Escuela Técnica Superior de
Ingeniería Informática

Trabajo Fin de Grado

Implementación de una biblioteca para resolución de juegos con información perfecta e imperfecta

Realizado por
Pedro Luis Soto Santos

Para la obtención del título de
Grado en Ingeniería Informática - Ingeniería del Software

Dirigido por
Ignacio Pérez Hurtado de Mendoza

Realizado en el departamento de
Ciencias de la Computación e Inteligencia Artificial

Junio, Curso 2022/23

Agradecimientos

En primer lugar, me gustaría agradecer a todos y cada uno de los profesores que se han cruzado en mi camino y que han contribuido a mi formación como estudiante y persona. En especial, agradezco a mi tutor Ignacio por motivarme a ir siempre un paso más allá y sacar la mejor versión de mí.

También quiero agradecer a mi familia, sobre todo a mis padres y hermana, por brindarme su amor y apoyo constante e incondicional, a mi pareja por comprenderme y acompañarme en todo momento, y a mis amigos y compañeros de clase por su empatía y ayuda.

A todos ellos, gracias.

> "El juego es la forma más elevada de investigación."
>
> Albert Einstein

> "Un juego es una situación conflictiva en la que uno debe tomar una decisión sabiendo que los demás también toman decisiones, y que el resultado del conflicto se determina, de algún modo, a partir de todas las decisiones realizadas."
>
> John von Neumann

Resumen

En este trabajo se estudian distintos algoritmos de búsqueda para la toma de decisiones en una amplia variedad de juegos de mesa, tanto de información perfecta como imperfecta, con el objetivo de implementar una biblioteca pública que incluya tales algoritmos. De esta forma, se facilita y apoya el desarrollo de juegos de ordenador en el marco del software libre. Además de la implementación de la biblioteca, en este trabajo se incluye el desarrollo de una extensa batería de juegos para ilustrar mediante ejemplos el uso de la biblioteca, realizando así la validación experimental de los algoritmos, comparativas y análisis estadísticos. Por último, como caso de estudio especial, cabe destacar que se ha implementado un algoritmo genético para la obtención de configuraciones iniciales de fichas en el juego del *Stratego*.

Palabras clave: Minimax, MCTS, SO-ISMCTS, MO-ISMCTS, árbol de búsqueda, teoría de juegos, algoritmos genéticos, información perfecta, información imperfecta, conjunto de información.

Abstract

In this work we explore different search algorithms for decision making in a wide variety of board games, both with perfect and imperfect information, in order to implement a public library that includes such algorithms. In this way, the development of computer games is facilitated and supported within the open source software environment. In addition to the implementation of the library, this work includes the development of an extensive battery of games to illustrate through examples the use of the library, thus performing experimental validation of the algorithms, comparative and statistical analysis. Finally, as a special case study, a genetic algorithms has been implemented to obtain the initial game board for the board game *Stratego*.

Keywords: Minimax, MCTS, SO-ISMCTS, MO-ISMCTS, tree search, game theory, genetic algorithms, perfect information, imperfect information, information set.

Índice General

Índice de Figuras

1. Introducción

1.1. Contextualización

Para comprender el contexto de este trabajo primero se introducirá brevemente la Teoría de Juegos.

La Teoría de Juegos es una rama de las matemáticas aplicadas que se utiliza para modelizar y analizar situaciones en las que múltiples agentes interactúan entre sí, cada uno de ellos con objetivos y restricciones diferentes. Se empezó a desarrollar como una herramienta que permitiese alcanzar una mejor comprensión del comportamiento existente en los fenómenos sociales[12].

Este campo de las matemáticas comienza, tal y como lo percibimos en la actualidad, con la publicación del libro *Theory of Games and Economic Behavior*[13] del matemático *John von Neumann* y el economista *Oskar Morgenstern*, en el que, por primera vez, de forma rigurosa se usan los conceptos básicos de esta teoría, todo ello desde un punto de vista puramente económico.

Pero no solo se ha hecho uso de esta teoría como herramienta para el estudio en el campo económico, también se ha usado en otros campos como la política, la biología y, obviamente, en los juegos de mesa. A continuación y durante el transcurso del trabajo nos centraremos en este último campo mencionado, los juegos de mesa.

En esta teoría, se distinguen múltiples tipos de juegos, pero nosotros nos vamos a centrar en dos de estos tipos: los juegos de información perfecta y los juegos de información imperfecta.

En un juego de información perfecta, cada jugador conoce todas las decisiones tomadas por los demás jugadores en el pasado, así como todas las posibles acciones disponibles en el presente. En otras palabras, los jugadores tienen un conocimiento completo del estado actual y pasado del juego. En este tipo de juegos, la estrategia óptima para cada jugador puede ser determinada a través de análisis matemático riguroso, ya que se dispone de toda la información necesaria para ello. Algunos ejemplos de juegos de mesa de este tipo son el *ajedrez*, las *damas* o el *Go*.

Por otro lado, en un juego de información imperfecta, cada jugador tiene información incompleta sobre el estado actual del juego o sobre las acciones tomadas por los demás jugadores. En este caso, la estrategia óptima para cada jugador es más difícil de determinar, ya que las decisiones de los demás jugadores pueden ser impredecibles o desconocidas. Algunos ejemplos de juegos de mesa de este tipo son la mayoría de juegos de cartas, ya que en casi todos ellos no tenemos el conocimiento de las cartas de los rivales o las cartas restantes del mazo, también hay algunos juegos de tablero con información imperfecta como podría ser el *Stratego*.

Gracias a la versatilidad de la Teoría de Juegos, los juegos de información imperfecta son de especial relevancia, ya que, en la mayoría de campos en los que se aplica dicha teoría, casi todas las situaciones incluyen ciertas incógnitas que nos hacen

imposible tener un conocimiento completo del estado, por ejemplo, en el campo de la economía, es imposible saber la mejor estrategia de inversión, ya que no se tiene el conocimiento completo y no se puede calcular como evolucionará el mercado en un futuro.

1.2. Objetivos

El objetivo principal de este Trabajo de Fin de Grado es el desarrollo de una biblioteca pública para *Python* que implemente ciertos algoritmos para la resolución de dos tipos de juegos de mesa: juegos con información perfecta y juegos con información imperfecta. Se pretende crear una herramienta accesible y fácil de usar para que cualquier programador interesado en desarrollar este tipo de juegos pueda hacerlo sin tener que hacer una implementación propia de los algoritmos fundamentales.

Para cada tipo de juego, la biblioteca incluirá un conjunto diferente de algoritmos que podrán ser utilizados para su resolución, de los cuales hablaremos detalladamente más adelante. El objetivo es que la biblioteca sea completa y ofrezca soluciones para cada uno de estos tipos de juego. También se busca que sea fácil de utilizar, por lo que, se proporcionará un manual de uso detallado que explicará el contenido de la biblioteca, en que casos debe ser utilizado cada uno de los algoritmos y cómo deben ser usados.

Para poder validar el correcto funcionamiento de la biblioteca y comprobar que se han implementado correctamente los algoritmos, se desarrollarán varios juegos de mesa para cada tipo de juego, cada uno de los cuales pondrá en práctica uno o varios de los algoritmos implementados. Cada juego será elegido teniendo en cuenta si es un juego de información perfecta o imperfecta, para saber cuál o cuáles de los algoritmos implementados debe ser aplicado. De esta manera, los futuros usuarios de la biblioteca tendrán varios ejemplos prácticos, para cada uno de los tipos de juegos y algoritmos, en los cuales se pueden fijar como caso de uso para aprender a utilizarla correctamente.

Por último, se realizará un análisis estadístico de los algoritmos incluidos en la biblioteca. Este análisis permitirá evaluar el rendimiento y la eficacia de cada algoritmo en la resolución de los juegos correspondientes.

1.3. Vista General

El resto de este Trabajo Fin de Grado se estructura como sigue:

En primer lugar, tendremos el marco teórico(2), en el cual se describirán los diferentes algoritmos y las investigaciones previas que han servido como base y soporte teórico para el desarrollo de la biblioteca. A continuación, se abordará el estado del arte(3), profundizando en el estado actual de algunos proyectos relacionados con estos algoritmos. Continuaremos con los detalles sobre el desarrollo y la publicación de la biblioteca(4) y los casos de estudio(5), en el que se expondrán los juegos desarrollados y los algoritmos seleccionados para la resolución de estos. Por último, se incluirán las

secciones de experimentación y pruebas(6), en los que se realizará el análisis de estos algoritmos, y las conclusiones y el trabajo futuro(7), en el que se plantearán ciertas áreas de mejora para la biblioteca y las conclusiones obtenidas durante el desarrollo del proyecto. Además, como anexo, se incluirá un manual de uso(A) de la biblioteca, para aprender a instalarla y utilizarla.

2. Marco Teórico

Esta sección contendrá una introducción teórica sobre los algoritmos que se incluyen en la biblioteca. Estos algoritmos se dividen en dos secciones dependiendo de si pueden ser aplicados a juegos de información perfecta o a juegos de información imperfecta.

2.1. Juegos de Información Perfecta

Como se mencionó en la introducción, un juego de información perfecta es aquel en el que toda la información sobre su estado está visible para todos los jugadores durante todo el desarrollo del juego. Esto significa que cada jugador puede ver todas las acciones que han ocurrido y pueden ser realizadas por los demás jugadores y como estas han afectado o afectarán al estado del juego, así como todas las posibles acciones que puede realizar en su turno y el efecto que tendrán sobre el estado del juego. Esto permite a cualquiera de los jugadores crear diferentes estrategias en base al análisis del estado del juego y las posibilidades de los rivales.

Por ejemplo, el *Tic-Tac-Toe* es un juego de información perfecta, ya que en todo momento se pueden ver las posiciones en las que el rival puede colocar su siguiente ficha, lo que permite al jugador calcular la acción a realizar teniendo en cuenta las acciones que se podrían realizar después. Otros ejemplos de juegos de información perfecta son el *ajedrez* y las *damas*, en los que ocurren lo mismo que en el caso del *Tic-Tac-Toe* solo que, al ser mucho más complejos, hay muchos más posibles movimientos y con ello multitud de estrategias.

2.1.1. Minimax

El primer algoritmo que será introducido en esta sección es el algoritmo de *minimax* [9]. Este es uno de los algoritmos de decisión más conocidos y se puede aplicar a juegos de información perfecta de dos jugadores y de suma cero, esto quiere decir que cuando un jugador gana el otro pierde o, por otro lado, ambos jugadores empatan. Si traducimos la victoria como una recompensa de un punto positivo para el jugador ganador, la derrota como una recompensa de un punto negativo para el jugador perdedor y el empate como una recompensa de cero puntos para ambos jugadores, si tras finalizar la partida sumamos las recompensas de ambos jugadores, para todas las posibles partidas, siempre el total de la suma será cero.

El algoritmo de *minimax* se centra en obtener el movimiento o la acción que minimice la pérdida máxima para uno de los jugadores, en otras palabras, este algoritmo se basa en suponer que cada uno de los jugadores siempre elegirá la jugada que maximice su recompensa, teniendo en cuenta que su rival hará lo mismo. Debido a que *minimax* se aplica a juegos de suma cero, la jugada que maximiza la recompensa de uno de

los jugadores es la misma que minimiza la recompensa del rival, por ello se usan los nombres de *jugador max*, para el jugador que realiza el primer movimiento, y *jugador min*, para su rival. Esto se debe a que, el *jugador min* busca maximizar su propia recompensa, pero a su vez, está minimizando la recompensa del jugador rival o *jugador max*.

El funcionamiento interno de este algoritmo se basa en la creación de un árbol de búsqueda en profundidad en el que cada nodo representa un estado del juego y sus aristas son las posibles acciones o jugadas que se pueden realizar desde este estado. El nodo raíz debe representar al estado actual del juego, por lo que, el jugador actual, es decir, el jugador al cual le toca jugar o realizar una acción en este estado, será el *jugador max*. Es muy importante tener claro el término *jugador actual*, ya que será un término que se irá alternando constantemente. Desde el nodo raíz saldrán, como aristas, todas las acciones que puede realizar el jugador actual en este turno, siendo una de ellas la acción que devolverá el algoritmo la cual minimizará la pérdida máxima del *jugador max*.

Para calcular la acción óptima del *jugador max*, el árbol se expandirá explorando todas las posibles jugadas y estados del juego, llegando en todas las ramas del árbol a un nodo terminal o nodo hoja, que representará un estado final de la partida en el que habrá un jugador ganador o un empate entre ambos jugadores. Cada uno de los nodos tendrá asociado un jugador, este jugador será el jugador actual del estado del juego que representa ese nodo, habiendo así *nodos max* y *nodos min*, dependiendo si representan a un *jugador max* o a un *jugador min*.

A cada uno de los nodos terminales se le adjudicará el valor correspondiente a la recompensa obtenida por el *jugador max*. A partir de aquí, se calculará el valor de todos los nodos superiores dependiendo de los valores de sus nodos hijos. Para los *nodos max* se elegirá el valor máximo entre los valores de todos sus hijos, esto significa que el *jugador max* siempre elige la mejor acción posible para el mismo, por otro lado, para los *nodos min* se elegirá el valor mínimo entre todos los valores de sus hijos, esto muestra como el *jugador min* siempre elige la acción que minimiza la recompensa del *jugador max* y que a la vez maximiza su propia recompensa. Una vez calculados los valores de todos los nodos, desde los nodos hoja hasta el nodo raíz, se obtiene la acción que dirige al nodo con mayor valor entre los hijos del nodo raíz, siendo esta la acción óptima que devolverá el algoritmo.

En la siguiente imagen se puede ver un ejemplo de como quedaría el árbol de búsqueda para el final de una partida del juego *3 en raya* en el que el *jugador max* sería la X y las recompensas son 1 por ganar y -1 por perder.

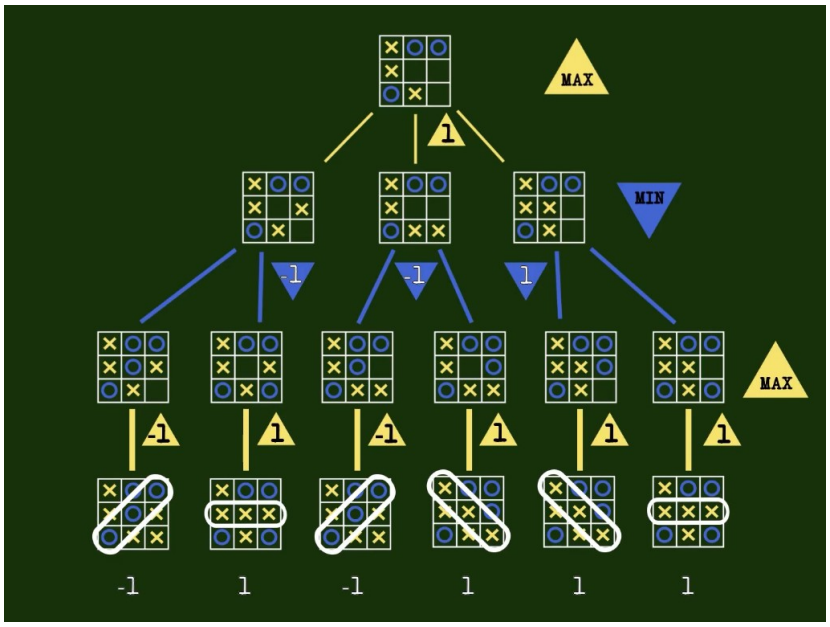

Figura 2.1: *Minimax*
Árbol de búsqueda de *minimax* para el final de una partida de *3 en raya*.

Función de Evaluación Heurística

El algoritmo *minimax* puede llegar a calcular el árbol de búsqueda completo para juegos sencillos como el *Tic-Tac-Toe* en un tiempo viable, pero cuando se aplica a juegos complejos como lo puede ser el *ajedrez*, en el que hay un gran número de posibles partidas, el coste y el tiempo de computación necesario para poder calcular todas estas partidas es enorme. Ya en el año 1950 el matemático estadounidense *Claude Shannon* propuso, en un artículo titulado *Programing a computer for Playing Chess*[10], usar una función de evaluación *heurística* para poder evaluar los estados del juego. Esta función dará una puntuación numérica según lo bueno que sea el estado del juego para el *jugador max*. En caso de ser un estado final de juego, la función debe devolver el valor máximo o el valor mínimo dependiendo de si es una victoria o una derrota para el *jugador max*, así el algoritmo siempre elegirá la victoria e intentará evitar la derrota.

Gracias a esta función se puede limitar la profundidad del árbol de búsqueda usando como recompensa en los nodos terminales el resultado de la función de evaluación *heurística*, dando la posibilidad de aplicar *minimax* en juegos complejos en los que se requeriría demasiado tiempo y coste de computación para obtener un árbol de búsqueda completo con todos los estados finales del juego.

Para realizar una función de evaluación *heurística* se debe tener un conocimiento experto sobre el juego al que se quiere aplicar, ya que la evaluación que realiza esta función es la parte que más influye en el algoritmo sobre la decisión del mejor movimiento posible para el *jugador max*. Esto puede suponer un impedimento a la hora de aplicar este algoritmo a juegos de los cuales no se tiene un gran conocimiento, ya que requeriría de un estudio previo para conocer el juego y poder realizar una buena función de evaluación.

Poda Alfa Beta

Del mismo modo que la función de evaluación *heurística*, también existen otras técnicas que hacen más eficiente al algoritmo *minimax*, una de ellas es la *poda alfa-beta* [8], la cual es aplicada a prácticamente todas las implementaciones de *minimax*. Esta técnica reduce el número de nodos que tienen que ser evaluados por el algoritmo *minimax* gracias a la eliminación, o poda, de algunas de las ramas del árbol de búsqueda que no serán relevantes para el cálculo de los valores de los nodos de niveles superiores. La eficiencia de la *poda alfa-beta* depende de la ordenación inicial de las acciones, siendo más eficiente si primero se exploran las acciones más prometedoras.

Igualmente, cabe destacar que la técnica de *poda alfa-beta* nunca afectará negativamente a la eficiencia del algoritmo *minimax*, por el contrario, siempre mejorará o igualará, en el peor de los casos con la peor ordenación inicial posible.

Esta técnica usa dos valores, el primero de ellos es *alfa* y representa el valor mínimo asegurado para el *jugador max*, el segundo valor es *beta* y representa el valor máximo asegurado para el *jugador min*. Ambos valores se inician suponiendo el peor de los casos para ambos jugadores, siendo $-$ ínf para *alfa* y $+$ ínf para *beta*. Estos valores se irán actualizando de la siguiente forma: siempre que se encuentre un valor mayor que *alfa* en un *nodo max*, *alfa* se actualizará con tal valor, y siempre que se encuentre un valor menor que *beta* en un *nodo min*, *beta* se actualizará con ese valor.

Estos valores servirán para evitar explorar nodos que no tengan posibilidad de ser elegidos. Existen dos tipos de podas: la *poda alfa* y la *poda beta*.

La *poda alfa* ocurre durante la evaluación de un *nodo max*, donde *alfa* representa la mejor opción encontrada hasta el momento. Un *nodo min* puede ser podado si uno de sus hijos tiene un valor menor o igual a **alfa**. Esto se debe a que el valor de **alfa** indica que ya se ha encontrado un hermano mejor, el cual será elegido por el *nodo max* padre antes que el *nodo min* que se está evaluando.

Por otro lado, la *poda beta* es similar a la **poda alfa**, pero ocurre durante la evaluación de un *nodo min*. En este caso, un *nodo max* puede ser podado cuando se encuentran valores mayores o iguales a *beta*. Esto indica que ya ha sido explorado un hermano con un valor menor, que será elegido sobre el nodo actual por el *nodo min* padre.

A continuación, se muestra una imagen en la que se puede ver un árbol de búsqueda al que se le aplica una *poda alfa* y una *poda beta*.

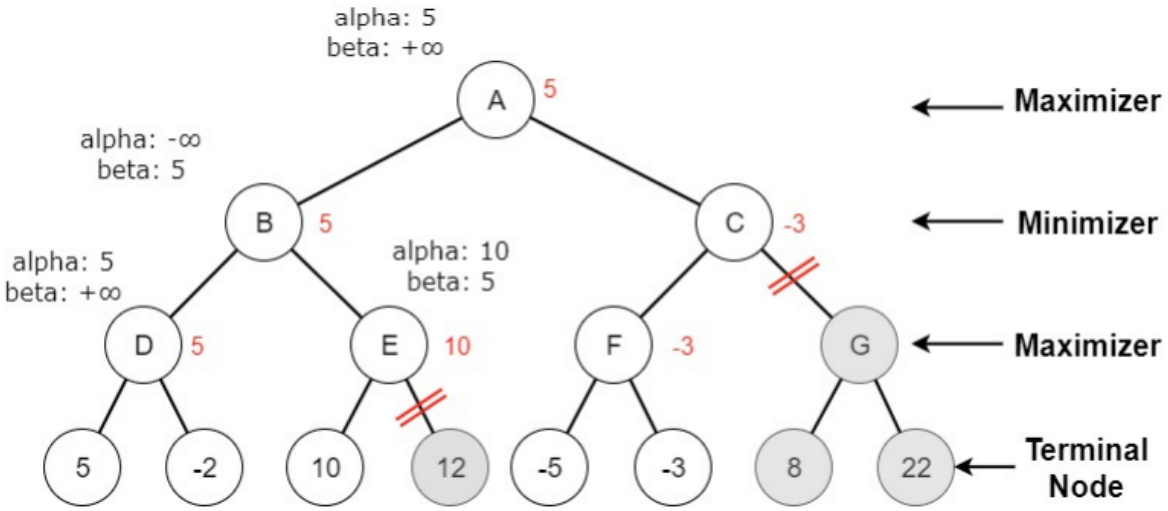

Figura 2.2: *Poda alfa-beta*
Ejemplo de una *poda alfa* y una *poda beta* en un árbol de búsqueda de *minimax*.

Por último, cabe destacar porque el algoritmo de *minimax* solo se debe aplicar a juegos de suma cero para dos jugadores, esto se debe a que el *jugador min* elige sus movimientos fijándose en la recompensa del *jugador max* sin tener en cuenta la recompensa propia. Esto provocaría que en el caso de no ser un juego de suma cero, por ejemplo, un juego cooperativo, el *jugador min* solo hiciese jugadas para molestar al *jugador max* sin tener en cuenta si estas jugadas conllevan una recompensa negativa para el mismo, por lo que obviaría estrategias en las que ambos jugadores obtendrían recompensas positivas.

Algo parecido ocurre cuando se usa *minimax* en juegos con más de 2 jugadores, y es que, en estos casos el algoritmo solo contempla un *jugador max*, por lo que, supone que todos los demás jugadores, al ser tratados como *jugadores min*, escogerán las acciones que conlleven a una menor recompensa del *jugador max*, siendo esto irracional, ya que cada jugador debería centrarse en obtener la mayor recompensa posible independientemente de las recompensas que obtuviesen los rivales. Ambos problemas se solucionan en algunas de las variantes del algoritmo *minimax*, como por ejemplo, el algoritmo max^n[7], en el que cada jugador se centra en obtener la mayor recompensa posible sin tener en cuenta las recompensas obtenidas por los otros jugadores.

En la biblioteca se desarrollará el algoritmo *minimax* limitado en profundidad con el uso de una función de evaluación *heurística* para obtener una evaluación de los estados de juego en los nodos terminales y se implementará la técnica de *poda alfa-beta* para aumentar la eficiencia al disminuir el número de nodos evaluados, y con ello, disminuirá el tiempo de ejecución necesario.

2.1.2. MCTS

El algoritmo *Monte Carlo Tree Search (MCTS)* [1] es la aplicación del *método de Montecarlo* en árboles de búsqueda. El *método de Montecarlo* es un método no deter-

minista que utiliza el azar para obtener resultados aproximados a problemas complejos de resolver de manera exacta. Un método determinista es aquel que, con los mismos datos de entrada, siempre obtiene el mismo resultado, por otro lado, un método no determinista, aún con los mismos datos de entrada, puede obtener diferentes resultados debido al uso del azar en el cálculo de estos. Esto quiere decir que para cada ejecución del algoritmo *MCTS*, sobre el mismo estado del juego inicial, se crea un árbol de búsqueda diferente, a diferencia del algoritmo *minimax*, en el que siempre se crea el mismo árbol de búsqueda, ya que es un algoritmo determinista que no hace uso de la aleatoriedad.

Al igual que en el algoritmo *minimax*, el árbol de búsqueda del algoritmo *MCTS* está formado por nodos que representan estados del juego y por aristas que representan acciones, siendo el nodo raíz el estado actual del juego. Además, el algoritmo *MCTS* solo es aplicable a juegos de información perfecta, ya que, el algoritmo debe saber como puede evolucionar el estado del juego, sin embargo, a diferencia del algoritmo *minimax*, si es aplicable a juegos de más de dos jugadores, debido a que no se hace distinción entre *jugador max* y *jugador min*.

Por último, la mayor ventaja que aporta *MCTS* frente a *minimax* es la ausencia del uso de la función de evaluación *heurística*, esto permite que se pueda usar *MCTS* en juegos en los que no se tenga el conocimiento necesario para la creación de una buena *heurística*. Finalmente, cabe añadir que cada uno de los nodos del árbol de búsqueda creado por el algoritmo *MCTS* debe guardar cierta información, que se irá actualizando a lo largo de las distintas iteraciones del mismo, como el número de veces que ha sido visitado o la recompensa acumulada que ha ido obteniendo, estos datos serán utilizados en una de las cuatro fases de las que consta una iteración completa del algoritmo *MCTS*. Estas cuatro fases son las siguientes:

- **Selección**: En esta primera fase se selecciona el nodo del árbol de búsqueda que será explorado durante las siguientes fases.
- **Expansión**: Una vez seleccionado el nodo se expande el árbol de búsqueda añadiendo un nodo hijo al nodo seleccionado anteriormente, siempre y cuando, el nodo seleccionado no represente a un estado final del juego, en este caso, no podría ser expandido con un nuevo hijo, ya que un estado final de juego no tiene posibles acciones para aplicar.
- **Simulación**: En la tercera fase se hace una simulación de juego, mediante la realización de movimientos aleatorios, desde el estado de juego que representa el nuevo nodo expandido, hasta llegar a un estado final de juego. Una vez llegado al estado final de juego se comprueban las recompensas obtenidas por los jugadores, según si han sido ganadores, perdedores o ha habido un empate.
- **Retro propagación**: Por último, obtenidas las recompensas de los jugadores, se asignan estas recompensas al nodo hijo, expandido en la segunda fase, y se propagan desde este nodo hacia el nodo raíz actualizando el valor de la recompensa acumulada de cada uno de los nodos que se recorran. Estos valores se usarán en la fase de selección para saber cuál será el nodo seleccionado para ser explorado.

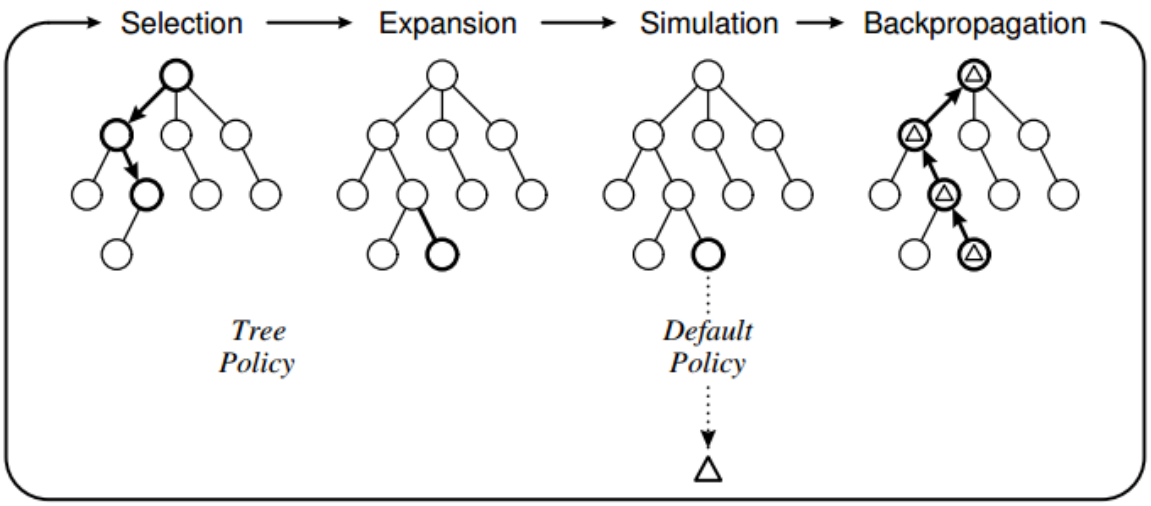

Figura 2.3: Fases *MCTS*
Secuencia de ejecución de las 4 fases en una iteración del algoritmo *MCTS*.

UCT

El problema que conlleva este algoritmo está relacionado con decidir cómo se hace la selección del nodo que será explorado. Si se seleccionan nodos de manera completamente aleatoria, podrían existir estados en los que haya muchas acciones posibles, pero solamente un pequeño subconjunto de estas serían realmente buenas, por lo que sería muy poco probable que sean elegidas las mejores acciones y, por tanto, el algoritmo perdería mucho tiempo explorando acciones que no merecen la pena.

Para solucionar este problema existe una variante de *MCTS* llamada *Upper Confidence bounds applied to Trees (UCT)* [4] la cual resuelve el problema añadiendo una función que aplica una fórmula, derivada del algoritmo *UCB*, en la fase de selección de *MCTS* para elegir el siguiente nodo a explorar en el árbol de búsqueda. Esta función y el uso de esta fórmula permite llevar un equilibrio entre exploración y explotación, es decir, entre explorar nodos poco visitados o explorar nodos que han sido más visitados y han acumulado buenas recompensas.

Esta función aplica su fórmula a todos los nodos hijos del nodo actual (nodo padre), siendo elegido, como mejor hijo a explorar, el nodo que obtenga un mayor valor. Si más de un nodo hijo obtiene el valor máximo se elige uno de ellos aleatoriamente. Esta fórmula hace uso de la información guardada en cada nodo sobre el número de veces que ha sido visitado y la recompensa acumulada que ha ido obteniendo, como se puede ver a continuación:

$$UCT(s) = \frac{r(s)}{n(s)} + K\sqrt{\frac{2\ln n(s_0)}{n(s)}}$$

Donde s es el nodo hijo, $r(s)$ es su recompensa acumulada, $n(s)$ es el número de veces que ha sido visitado, $n(s_0)$ es el número de veces que ha sido visitado su nodo

padre y K es una constante, llamada *constante de exploración*, mayor que 0 que ajustará la relación entre exploración y explotación del algoritmo *MCTS*. Se recomienda usar el valor $K = 1/\sqrt{2}$ cuando las recompensas se sitúan en el rango $[0, 1]$ [1].

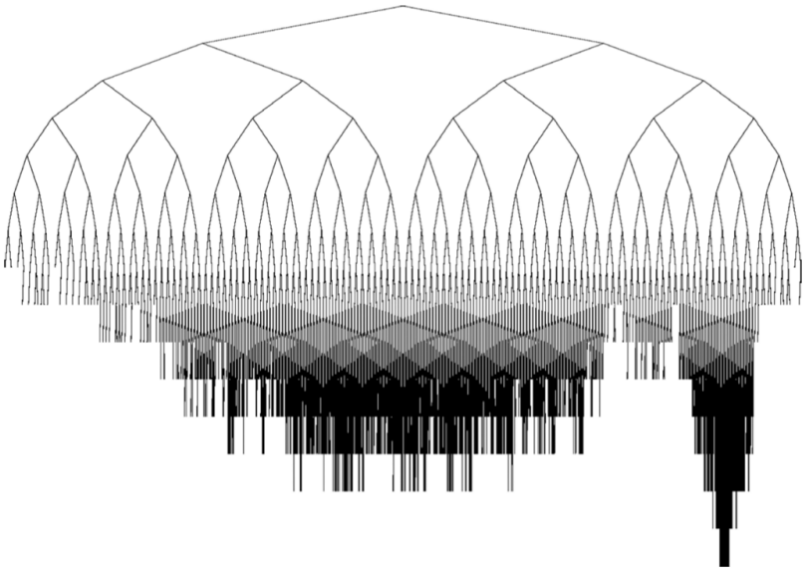

Figura 2.4: Árbol de búsqueda *MCTS*
Árbol de búsqueda de *MCTS* tras múltiples iteraciones afectado por la relación entre exploración y explotación provocada por el uso de *UCT*.

En la imagen superior se puede apreciar la uniformidad en la expansión del árbol de búsqueda, en donde hay una mayor explotación en la parte derecha del árbol, pero aun así, se mantiene una gran expansión en la parte central. Esto se debe a que el valor de la *constante de exploración* mantiene una relación equilibrada entre explotación y exploración, si fuese cambiado el valor de esta constante esta relación cambiaría, y con ello, la expansión del árbol de búsqueda. Por último, cabe destacar que la variante *UCT*, con el tiempo y memoria de computación necesario, permite que el algoritmo *MCTS* converja al árbol de búsqueda de *minimax*, obteniendo así una acción óptima para cada estado.

En la biblioteca se desarrollará el algoritmo *MCTS* basado en la variante *UCT* al que se le aplicará un limite en el tiempo de computación, tras el cual, devolverá la mejor acción encontrada hasta el momento.

2.2. Juegos de Información Imperfecta

Hasta ahora se han explicado los dos primeros algoritmos que serán desarrollados e implementados en la biblioteca, pero como se mencionó anteriormente, estos solo son aplicables a juegos de información perfecta. A partir de aquí, se presentarán algoritmos diseñados para ser aplicados a juegos de información imperfecta, es decir, en juegos en los que hay cierta información oculta para todos o algunos jugadores.

En estos juegos, cada jugador solo tiene acceso a una parte de la información del estado del juego, siendo por ello difícil crear buenas estrategias. Por ejemplo, en el *blackjack* no se conoce el orden de las cartas del mazo de robo, por lo que no se puede saber qué carta será la siguiente en ser sacada por el crupier. Aunque se puedan calcular las probabilidades teniendo en cuenta las cartas que ya han sido jugadas, siempre habrá un margen de error, lo que impide seguir una estrategia de juego perfecta.

Existen varios tipos de elementos que caracterizan un juego de información imperfecta, pero nos centraremos en dos de ellos:

- Conjunto de información: Conjunto de estados de juego que aparecen cuando un jugador tiene oculta cierta información sobre el estado actual del juego. Cada jugador tiene su propio conjunto de información y este contiene todos los posibles estados que son indistinguibles para este jugador debido a la información oculta. Esto ocurre, por ejemplo, en los juegos de cartas en los que los rivales ocultan sus cartas a los demás jugadores, o como se ha ejemplificado anteriormente, en el *blackjack*, ya que no se conoce el orden de las cartas del mazo de robo. Siguiendo con el ejemplo del *blackjack*, el conjunto de información de un jugador contendría todos los estados correspondientes a todas las permutaciones posibles del orden de las cartas del mazo de robo. Cabe destacar que un jugador sabe en que conjunto de información se encuentra, pero no en que estado dentro de ese conjunto.
- Movimientos parcialmente observables: Un juego tiene movimientos parcialmente observables si un jugador puede realizar movimientos en los que queda oculta toda o parte de la información a uno o varios jugadores rivales. Por ejemplo, si en un juego de cartas se permite a un jugador intercambiar una carta de su mano con el mazo de robo sería un movimiento parcialmente observable, ya que los rivales pueden ver que ha ocurrido el intercambio, pero no saben qué carta se ha añadido al mazo ni cuál es la carta que ha obtenido el rival.

Una de las técnicas para hacer frente a esta falta de información es la técnica de la determinación, también conocida como búsqueda con *Perfect Information Monte Carlo (PIMC)* [6], en la que se crea una muestra de varias determinaciones del estado actual del juego y se analiza cada una de ellas con algoritmos de búsqueda para juegos de información perfecta. Una determinación es una instancia del juego en el que el estado actual es uno de los estados del conjunto de información del jugador actual, por lo que, no hay información oculta para ninguno de los jugadores, convirtiéndose en un juego de información perfecta.

Por ejemplo, una determinación de un juego de cartas sería una instancia del juego en el que las cartas de los jugadores y la baraja son visibles para todos los jugadores. De esta manera, los algoritmos diseñados específicamente para juegos de información perfecta, como *minimax* o *MCTS*, pueden ser utilizados para analizar cada determinación, y con ello, posteriormente, obtener una estrategia común para el juego de información imperfecta.

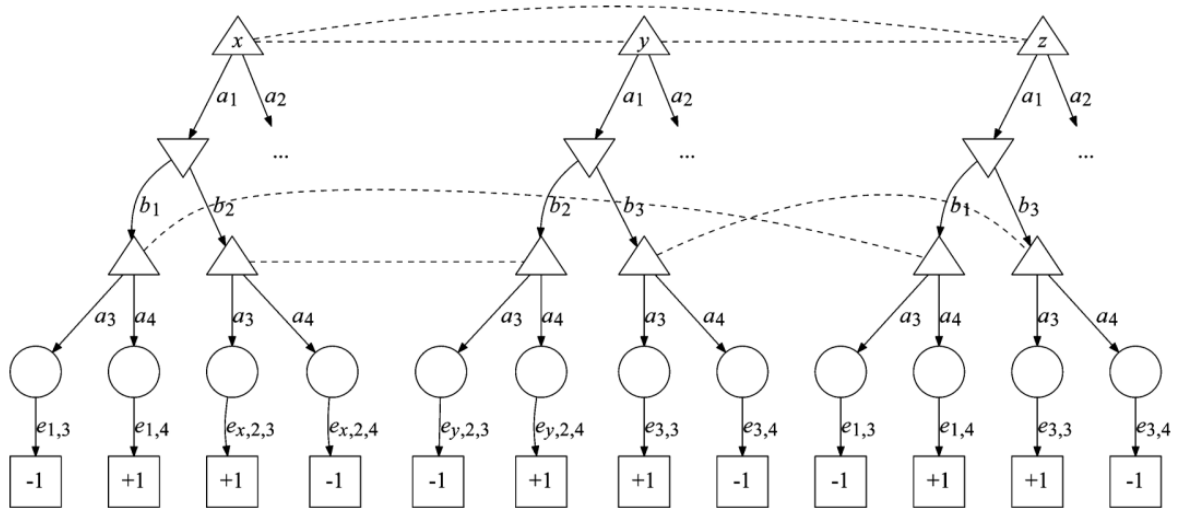

Figura 2.5: Determinación
Ejemplo de árbol de búsqueda para 3 determinaciones (x, y, z). Los nodos representan estados de juego. Los $\triangle$ representan estados de decisión del jugador 1, los $\triangledown$ representan estados de decisión del jugador 2, los $\bigcirc$ representan estados del entorno y los $\square$ representan estados finales con las recompensas del jugador 1. Las líneas discontinuas representan que los nodos que unen están en el mismo conjunto de información para el jugador 1.

A continuación, se verá una familia de algoritmos llamada *Information Set Monte Carlo Tree Search (ISMCTS)* [2], la cual surge para abordar alguna de las debilidades de la técnica de la determinación. Además, se explicará el porqué surgen estas debilidades y como serán resueltas por estos algoritmos.

2.2.1. SO-ISMCTS

Uno de los puntos débiles de la técnica de la determinación es conocido como fusión de estrategias y se debe a que al tener múltiples árboles de búsqueda, uno por cada determinación, puede haber nodos en distintos árboles que representen a estados de juego pertenecientes al mismo conjunto de información, es decir, que desde el punto de vista del jugador estos estados son iguales, pero la búsqueda los trata como nodos distintos, ya que no tiene la información oculta que el jugador tendría. Esto provoca que pueda haber incoherencias a la hora de obtener una estrategia común a todas las determinaciones, ya que el algoritmo erróneamente asume que puede tomar decisiones diferentes en función de la información que tiene sobre el estado del juego, pero realmente esta información está oculta para el jugador. Además, no se comparte información entre los árboles de búsqueda, lo que impide unificar los nodos que se tengan en común y obliga a dividir la fuerza de cómputo entre todos los árboles.

El primer algoritmo que fue diseñado para arreglar estas debilidades es conocido como *Single Observer Information Set Monte Carlo (SO-ISMCTS)*. Este algoritmo en lugar de crear un árbol de búsqueda para cada determinación, crea un único árbol de búsqueda en el que los nodos representan conjuntos de información en vez de estados. Gracias a tener un único árbol de búsqueda, en el que se guardan todas las estadísticas

de los movimientos, no se tiene que dividir la fuerza de computo y se puede utilizar de manera más eficiente para obtener una mayor profundidad en el único árbol de búsqueda.

Este algoritmo está basado en el *MCTS*, como bien indica su nombre, por lo que cada iteración de este algoritmo contiene las mismas 4 fases que el *MCTS* (Figura 2.3). La diferencia es que para cada iteración del *SO-ISMCTS* se obtiene aleatoriamente una determinación del juego en base al conjunto de información del jugador actual, es decir, del jugador del que se quiere obtener el mejor movimiento.

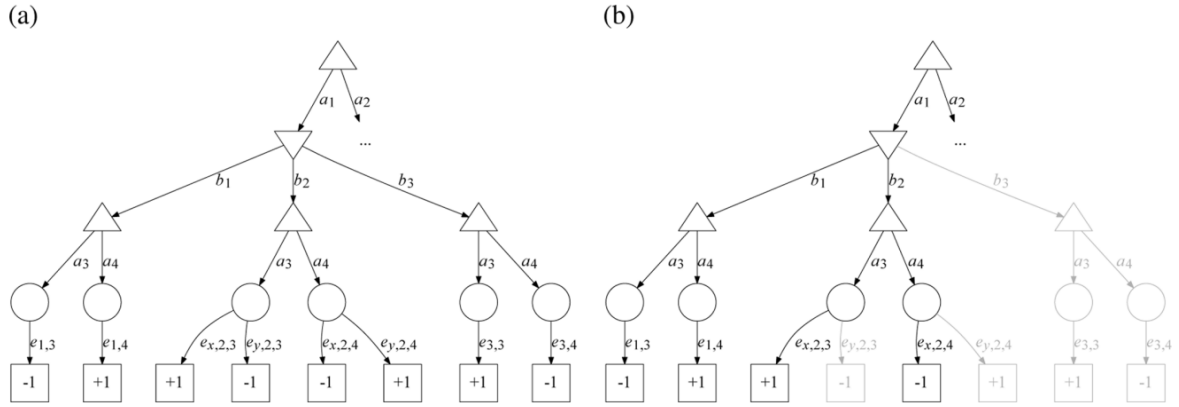

Figura 2.6: *SO-ISMCTS*
Árbol de búsqueda con conjuntos de información para el juego de la Figura 2.5. La imagen (a) representa el árbol de búsqueda completo. La imagen (b) representa el árbol de búsqueda restringido para la determinación x de la Figura 2.3.

En la imagen anterior (a) se puede ver como han sido juntados los nodos que en la Figura 2.5 estaban en el mismo conjunto de información (nodos unidos por líneas discontinuas), mostrando como es posible representar múltiples determinaciones con un único árbol de búsqueda. Esto también permite que el algoritmo de búsqueda no se centre en una sola determinación y sea capaz de explorar acciones que son buenas para múltiples determinaciones, reduciendo o eliminando por completo, dependiendo del juego, los efectos de la fusión de estrategias. Esto se debe a que los nodos guardarán la recompensa acumulada obtenida para todas las determinaciones, por lo que las decisiones no serán tomadas teniendo en cuenta solo la determinación actual, sino que tendrán importancia todas las determinaciones anteriormente exploradas.

Además, se usará la variante *UCT* de *MCTS* en la fase de selección de *SO-ISMCTS*, pero con una pequeña variación, ya que las acciones disponibles para un mismo nodo pueden diferir en cada iteración del algoritmo. Es decir, habrá determinaciones que tengan acciones disponibles que otras determinaciones del mismo conjunto de información no tienen, por lo que estas acciones han podido ser exploradas en anteriores iteraciones y por ello se ven representadas en el árbol de búsqueda, pero no deben tenerse en cuenta en las iteraciones en las que estas acciones no están disponibles para la determinación correspondiente.

Por ello, para cada iteración se obtiene una determinación y se restringe el árbol

de búsqueda a las zonas disponibles para esta determinación, como se puede ver en la Figura 2.6 (b). Esto provoca que deba ser reemplazado el atributo que representa el número de veces que ha sido visitado el nodo padre por un atributo que representará el número de veces que el nodo ha estado disponible para ser elegido en la fase de selección, ya que el nodo padre ha podido tener muchas visitas, pero esto no quiere decir que este nodo hijo haya estado disponible para ser seleccionado en todas ellas.

2.2.2. SO-ISMCTS + POM

Sin embargo, el *SO-ISMCTS* no siempre evita por completo los efectos de la fusión de estrategias, ya que si se aplicara el *SO-ISMCTS* a algún juego que contenga movimientos parcialmente observables, el algoritmo los trataría como movimientos completamente observables, ya que las aristas representan las acciones tomadas desde el punto de vista del jugador que las toma, por lo que el algoritmo usaría esta información para crear el árbol de búsqueda, pero esta información es desconocida por los demás jugadores al no obtener estos jugadores rivales la observación completa que obtendría el jugador que realiza el *movimiento parcialmente observable*.

Esto provoca que se genere un nuevo nodo, que representaría un nuevo conjunto de información, para cada una de las acciones tomadas por los rivales, lo cual no es correcto. Estas acciones son indistinguibles desde el punto de vista de cada jugador rival, es decir, el jugador obtiene la misma observación para todas las posibles acciones parcialmente observables que pueda realizar el rival, por lo que todas ellas llevan al mismo conjunto de información. Estas acciones deberían ser representadas con un único nodo, para todas ellas, en el árbol de búsqueda.

Para solucionar este problema se diseñó una variante para el *SO-ISMCTS* llamada *Single Observer Monte Carlo Tree Search with Partially Observable Moves (SO-ISMCTS+POM)* en la que las aristas del árbol de búsqueda pasan de representar las acciones desde el punto de vista del jugador que las realiza, a representar las acciones desde el punto de vista del jugador actual del nodo raíz, es decir, el jugador al cual el algoritmo le está buscando el mejor movimiento.

Esta pequeña variación trata correctamente las acciones parcialmente observables, ya que crea una única arista representando que todos los movimientos parcialmente observables son indiferenciables desde el punto de vista del jugador actual del nodo raíz, es decir, que el jugador actual del nodo raíz obtendrá la misma observación independientemente del movimiento que realice el rival. Esta única arista se dirigirá a un único nodo el cual representa correctamente el nuevo conjunto de información del jugador, ya que este obtiene un único conjunto de información al obtener la misma información para todos los movimientos parcialmente observables.

Sin embargo, al no crearse un nodo para cada una de las acciones parcialmente observables posibles para el rival, no se guarda información sobre las recompensas acumuladas que obtiene el rival al realizar cada una de estas acciones, por lo que no se puede tomar una decisión informada en la fase de selección, sino que se selecciona aleatoriamente una acción entre todas las acciones parcialmente observables disponibles para el rival. Este es el gran inconveniente de esta variante, y es que, se soluciona el

problema de la aparición de movimientos parcialmente observables, pero se sacrifica la racionalidad de las jugadas de los rivales, es decir, suponiendo que los rivales jugarán de manera aleatoria sin seguir una estrategia cuando tengan que elegir entre un conjunto de acciones parcialmente observables.

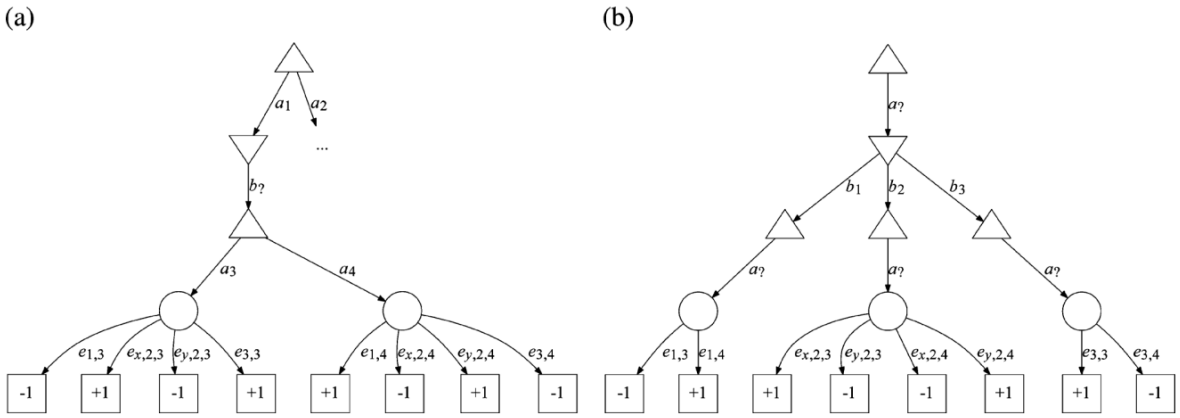

Figura 2.7: *SO-ISMCTS+POM* y *MO-ISMCTS*
Árbol de búsqueda con conjuntos de información para el juego de la Figura 2.5 con movimientos parcialmente observables en los que el jugador 1 no puede diferenciar entre las jugadas realizadas por el jugador 2 y viceversa. La imagen (a) representa el árbol de búsqueda del *SO-ISMCTS+POM* para el jugador 1. Las imagenes (a) y (b) representan la pareja de árboles de búsqueda para el *MO-ISMCTS*, donde (a) sería el punto de vista del jugador 1 y (b) el punto de vista del jugador 2.

En la imagen anterior (a) se puede apreciar como han quedado unificadas todas las acciones parcialmente observables, para los rivales, del jugador 2 (b_1,b_2,b_3) en una única arista ($b_?$), que representa como el jugador 1 no puede distinguir entre los movimientos que realiza el jugador 2. Esto hace que en la fase de selección de cada iteración se elija aleatoriamente entre una de las acciones disponibles para la determinación de esa iteración, ya que no se guarda información individualizada para cada acción, por lo que el algoritmo no puede saber cuál es la mejor jugada para el jugador 2.

2.2.3. MO-ISMCTS

Como se ha visto anteriormente la variante *SO-ISMCTS+POM* soluciona el problema de la aparición de movimientos parcialmente observables a consta de asumir que el rival seguirá una estrategia aleatoria a la hora de decidir entre todos los posibles movimientos parcialmente observables. Esta variante puede llegar a funcionar en juegos en los que sea poco frecuente la aparición de decisiones entre varios movimientos parcialmente observables. Sin embargo, para juegos en los que todas las acciones realizadas por un jugador son indistinguibles para todos los demás jugadores rivales, como en el caso de la Figura 2.7, esta variante asumirá que el rival jugará de manera completamente aleatoria.

Por ejemplo, imaginemos un juego en el que el rival tiene múltiples acciones y una

de ellas le otorga una clara ventaja. En el caso del algoritmo *MCTS*, gracias a guardar la información sobre las recompensas acumuladas de cada acción, podrá identificar esta situación y prever que el rival, si juega de forma racional, realizará la acción ventajosa. Debido a esto, el algoritmo intentará evitar que se llegue a esta situación del juego.

Sin embargo, el algoritmo *SO-ISMCTS+POM* al no guardar información sobre las recompensas y elegir las acciones de manera aleatoria, existe la posibilidad de que nunca elija la acción que le otorga gran ventaja al rival. Esto provocaría que el algoritmo no detectará que se trata de una situación desfavorable y como resultado, podría calcular estrategias que lleven a esta situación, lo que afectaría negativamente su rendimiento en el juego.

Para solucionar este problema se diseñó un algoritmo llamado *Multiple Observer Information Set Monte Carlo Tree Search (MO-ISMCTS)*. Este algoritmo crea un árbol de búsqueda para cada jugador, en los que los nodos representan los conjuntos de información de su jugador y las aristas representan las acciones desde el punto de vista de su jugador. En cada iteración se desciende simultáneamente por todos los árboles de búsqueda y en la fase de selección se utilizan las estadísticas del árbol del jugador que va a realizar la acción. Una vez elegida y realizada la acción se desciende en todos los árboles por la rama correspondiente a la observación que obtiene el jugador dueño de cada árbol. Si un árbol no tiene ninguna rama que se corresponda con esta observación, entonces la crea.

Gracias a tener un árbol de búsqueda para cada jugador, se puede guardar información sobre los movimientos de cada uno de ellos, aunque estos sean indiferenciables para los rivales, así en la fase de selección el algoritmo de *UCT* puede usar esta información para realizar una decisión informada, teniendo en cuenta las estadísticas de todas las posibles acciones. Debido a ello no se asume que el jugador sigue una estrategia aleatoria, ya que se dejan de elegir aleatoriamente entre las acciones parcialmente observables.

Cabe destacar que la unión de los árboles de búsqueda de todos los jugadores en un único árbol de búsqueda da como resultado el árbol de búsqueda para el algoritmo *SO-ISMCTS*, como se puede comprobar observando la Figura 2.6 y la Figura 2.7. Esto demuestra que el algoritmo guarda toda la información del juego repartida entre sus árboles de búsqueda, por lo que no hay pérdida de información alguna.

En la biblioteca se desarrollarán los algoritmos *SO-ISMCTS* y *MO-ISMCTS* con el uso de un limite en el tiempo de computación, tras el cual, se devolverá la mejor acción encontrada hasta el momento. Se ha decidido dejar fuera de la biblioteca la implementación del algoritmo *SO-ISMCTS+POM* debido a que es un algoritmo que está a medio camino entre el *SO-ISMCTS* y el *MO-ISMCTS*, ya que resuelve el problema de los movimientos parcialmente observables que surge en el algoritmo *SO-ISMCTS* pero asume nuevos problemas, en cambio, el algoritmo *MO-ISMCTS* está diseñado para resolver el problema de los movimientos parcialmente observables sin asumir ningún problema extra.

3. Estado del Arte

En el ámbito de la programación en *Python*, las bibliotecas juegan un papel fundamental en el desarrollo de software eficiente y funcional. Estas bibliotecas contienen una gran variedad de funcionalidades y herramientas que permiten a los programadores resolver problemas específicos de manera más eficiente y rápida, sin tener que desarrollar todas las funcionalidades desde cero.

Hay todo tipo de bibliotecas, desde bibliotecas centradas en el análisis o visualizado de datos, como *matplotlib* o *numPy*, hasta bibliotecas para el desarrollo de videojuegos, como *pyGame*. Gracias a las bibliotecas y todas las funcionalidades que estas aportan de forma abierta, *Python* se ha convertido en uno de los lenguajes de programación más usados en la actualidad y es esta una de las razones por las que he decidido realizar este Trabajo de Fin de Grado.

Este Trabajo de Fin de Grado se centra en el desarrollo de una nueva biblioteca para *Python*, enfocada en la implementación de varios algoritmos de resolución de juegos. Para comprender el valor y la relevancia de este desarrollo, es necesario realizar un análisis del estado del arte en este campo, en el que se identifiquen bibliotecas similares y se analicen sus fortalezas y debilidades.

El análisis inicial se centró en la búsqueda de bibliotecas para *Python* que implementaran el algoritmo *minimax*, uno de los algoritmos de resolución de juegos de información perfecta seleccionados para formar parte de la nueva biblioteca en desarrollo. Sin embargo, a pesar de que el algoritmo *minimax* es ampliamente conocido y utilizado, no se encontró ninguna biblioteca específica que se incluyese la posibilidad de utilizar tal algoritmo.

Aunque se descartó la existencia de bibliotecas dedicadas al algoritmo *minimax*, se hallaron algunas bibliotecas que implementan juegos de mesa sencillos, como el *Tic-Tac-Toe* o *3 en raya* en español, las cuales contienen implementaciones internas del algoritmo *minimax*. Aunque parezca extraño, puede llegar a tener sentido considerando que en internet se encuentran numerosos artículos y páginas web que explican en detalle cómo realizar una implementación del algoritmo *minimax*.

Sin embargo, a pesar de la existencia de múltiples recursos educativos, muchos de estos principalmente se centran en explicar el algoritmo en sí, sin hablar sobre aspectos de eficiencia y optimización. En muchos casos, no se incluye la técnica de *poda alfa-beta*, que es esencial para aumentar significativamente el rendimiento del algoritmo. Además, algunos de los juegos de ejemplo utilizados en estos recursos son demasiado simples y no se profundiza en la importancia de utilizar una buena función *heurística* para aplicar el algoritmo *minimax* en juegos de mesa más complejos, en los que el cálculo del árbol de búsqueda completo en un tiempo óptimo puede ser inviable, por lo que el uso de una buena función *heurística* es esencial.

Gracias a este primer análisis, queda claro que el desarrollo de una biblioteca que implemente eficientemente el algoritmo *minimax*, utilizando la técnica *poda alfa-beta* y permitiendo el uso de funciones *heurísticas* para poder ser usado en todo tipo de

juegos de mesa de información perfecta, es una gran oportunidad de mejora en cuanto al estado actual, por lo que aportará gran valor a la comunidad de desarrolladores, ya que ahorrará el tiempo de buscar e implementar desde cero el algoritmo *minimax* a los desarrolladores que quieran hacer uso de este algoritmo en sus propios juegos.

Tras realizar este análisis inicial, se continuó con la investigación sobre el estado del arte de las bibliotecas que implementan el algoritmo *MCTS*. Durante esta búsqueda, se encontraron varias bibliotecas que ya implementaban este algoritmo y estaban diseñadas para que los usuarios pudieran utilizarlo en sus propios juegos. Entre las más destacadas se encuentran en *mcts*[11] y *mctspy*[3], las cuales están disponibles tanto en el repositorio de software oficial de *Python*, llamado *PyPI*, como en *GitHub*, la famosa plataforma de alojamiento de código. Ambos repositorios de *GitHub* cuentan con cierto número de visitas y diversos contribuyentes.

Sin embargo, se observó que ambas bibliotecas presentan una falta considerable de documentación. Apenas se proporcionan instrucciones básicas sobre cómo importar las bibliotecas, sin profundizar en detalles sobre su uso. Además, es importante destacar que la biblioteca *mctspy* obliga a los usuarios a utilizar las implementaciones internas de la biblioteca para desarrollar sus propios juegos para que estos puedan hacer uso de la implementación del algoritmo *MCTS*. Esta obligación, sumada a la falta de documentación, puede suponer un obstáculo significativo para los desarrolladores menos experimentados.

Aunque se haya encontrado un par de bibliotecas que ya implementan el algoritmo *MCTS* para juegos de mesa, la actividad que estas bibliotecas presentan reflejan el interés y el uso de estas bibliotecas por parte de la comunidad, a pesar de la escasa documentación proporcionada. Por lo tanto, sigue siendo una excelente opción desarrollar una nueva biblioteca que implemente el algoritmo *MCTS*, pero esta vez acompañada de una buena documentación en forma de manual de usuario.

Continuando con el análisis, se procedió a investigar la existencia de bibliotecas para *Python* relacionadas con la implementación de los algoritmos para la resolución de juegos de información imperfecta de la familia *information set Monte Carlo Tree Search (ISMCTS)*. En particular, se enfocó en la búsqueda de los algoritmos *SO-ISMCTS* y *MO-ISMCTS*, que son los seleccionados para ser desarrollados e implementados en la biblioteca que será desarrollada para este Trabajo de Fin de Grado.

Lamentablemente, durante esta búsqueda no se encontraron bibliotecas específicas que implementaran los algoritmos *SO-ISMCTS* y *MO-ISMCTS* en *Python*. Sin embargo, se encontraron algunos trabajos de investigación que mencionaban la implementación de un algoritmo llamado *ISMCTS*, sin mencionar mayor detalle sobre la implementación ni especificar si se trataba de alguno de los algoritmos específicos que estábamos buscando. Por lo tanto, no se pudo encontrar una implementación específica de los algoritmos *SO-ISMCTS* y *MO-ISMCTS* en *Python* en el estado del arte actual.

Este análisis confirma que el desarrollo de la biblioteca propuesta, que incluirá implementaciones eficientes de los algoritmos *SO-ISMCTS* y *MO-ISMCTS*, junto con una documentación completa a modo de manual de usuario, será una contribución valiosa para la comunidad de desarrolladores. Debido a que no se ha encontrado ninguna biblioteca que implemente estos algoritmos. La biblioteca propuesta será la primera

en traer estas implementaciones al estado del arte, ofreciendo nuevas posibilidades y oportunidades a los desarrolladores, sobre todo para los interesados en la resolución de juegos de información imperfecta.

En resumen, la biblioteca propuesta tiene un gran valor debido a varias razones. En primer lugar, sería, presumiblemente, una de las primeras bibliotecas en *Python* en ofrecer una implementación eficiente del algoritmo *minimax*, que es ampliamente utilizado en la resolución de juegos de información perfecta. Esto permitirá a los desarrolladores aprovechar este algoritmo de toma de decisiones de manera sencilla y rápida, sin tener que buscar o implementar soluciones desde cero.

Además, la biblioteca también incluirá implementaciones de los algoritmos *SO-ISMCTS* y *MO-ISMCTS*, que son algoritmos avanzados para la resolución de juegos de información imperfecta. Estos algoritmos son escasos en el estado del arte actual, lo que hace que la biblioteca propuesta sea aún más valiosa para aquellos interesados en abordar problemas dentro de este dominio.

Otro punto destacado es que la biblioteca se acompañará de una extensa documentación a modo de manual de usuario para el uso de todos y cada uno de los algoritmos. Esto es un punto fuerte en comparación con otras bibliotecas existentes, que como se ha podido apreciar en la investigación previa, a menudo carecen de una documentación clara y completa. La presencia de un manual de usuario detallado y accesible facilitará el uso de la biblioteca por parte de desarrolladores de todos los niveles de experiencia.

En conjunto, la biblioteca propuesta destaca por ofrecer implementaciones eficientes de los algoritmos *minimax*, *MCTS*, *SO-ISMCTS* y *MO-ISMCTS*, cubriendo tanto juegos de información perfecta como juegos de información imperfecta. Esta combinación de algoritmos centralizados en una única biblioteca junto con una documentación de calidad, la distingue de otras bibliotecas existentes y la convierte en una herramienta única, poderosa y accesible para afrontar desafíos en el dominio de la resolución de juegos, siendo un aporte de gran valor para la comunidad de desarrolladores de *Python*.

4. Biblioteca *pyplAI*

En esta sección, hablaremos sobre la biblioteca desarrollada durante este Trabajo de Fin de Grado, la cual lleva el nombre de *pyplAI*. El nombre fue elegido cuidadosamente para transmitir el propósito y la funcionalidad de la biblioteca.

El prefijo *py* es comúnmente utilizado en muchas bibliotecas de *Python*, en referencia al propio lenguaje de programación. Por otro lado, *plAI* es una combinación del verbo en inglés *play*, que se asocia con el acto de jugar, y la abreviatura *AI* de *artificial intelligence* (inteligencia artificial).

La elección de este nombre tiene como objetivo transmitir que la biblioteca está relacionada con la implementación de algoritmos de inteligencia artificial que se pueden aplicar en el contexto de juegos, utilizando el lenguaje de programación *Python*.

En primer lugar, es importante destacar que la biblioteca *pyplAI* ha sido desarrollada exclusivamente en *Python* y está diseñada para ser utilizada en proyectos que utilicen este mismo lenguaje de programación. También, como aspecto importante, se ha minimizado el uso de bibliotecas externas, lográndose que la biblioteca carezca de dependencias, ya que solamente han sido utilizadas bibliotecas incluidas en la biblioteca estándar de *Python*. Esto evita que tengan que ser descargadas bibliotecas adicionales para el uso de la biblioteca *pyplAI*.

Además, la biblioteca ha sido diseñada y desarrollada con un enfoque en la eficiencia. Por ello, han sido optimizadas las implementaciones de los algoritmos, buscando mejorar su rendimiento. Como base para el desarrollo de las implementaciones de los distintos algoritmos, se han utilizado diferentes artículos de investigación sobre los algoritmos implementados en la biblioteca.

Para el algoritmo *minimax*, se ha utilizado como base un artículo de investigación de la universidad *Yachay Tech* de Ecuador [8], en la que se realiza un análisis en profundidad sobre el rendimiento del algoritmo *minimax* con la técnica *poda alfa-beta*. Sin embargo, este artículo se centra en investigaciones con juegos de mesa simples y no diseña un algoritmo limitado en profundidad, como el que ha sido desarrollado para esta biblioteca.

Por lo tanto, este artículo ha servido como base teórica y práctica para el diseño de la implementación propia del algoritmo *minimax* con la *poda alfa-beta* en la biblioteca *pyplAI*. Sin embargo, se han realizado algunas modificaciones y ajustes sobre las implementaciones recomendadas por el artículo para mejorar su rendimiento. Esto incluye la incorporación de una función de evaluación *heurística* que permite limitar la búsqueda en profundidad para adaptar el algoritmo a juegos más complejos.

En cuanto al algoritmo *MCTS*, se ha utilizado un artículo de investigación realizado por un grupo de 10 profesores de distintas universidades del Reino Unido, la mayoría de ellos miembros del *IEEE*, que aborda varios métodos de búsqueda en árboles con Monte Carlo.

Entre los métodos estudiados en este artículo se incluye el algoritmo *MCTS* con

la variante *UCT*. Esto sirvió como base para el desarrollo de una implementación propia de este algoritmo. Aunque no se partió desde cero, el artículo utilizado como base proporcionaba poca información sobre como realizar estas implementaciones, limitándose a ofrecer algunos pseudocódigos de alto nivel. Esto se debe a que el artículo se centra en múltiples variantes del algoritmo *MCTS* y no profundiza demasiado en implementaciones específicas.

Para las implementaciones de los algoritmos *SO-ISMCTS* y *MO-ISMCTS*, se ha utilizado como referencia teórica y práctica un artículo escrito por tres investigadores y profesores del Departamento de Inteligencia Artificial de la Universidad de Bradford en el Reino Unido [2], Peter I. Cowling, Edward J. Powley y Daniel Whitehouse. En dicho artículo se introduce por primera vez la familia de algoritmos *ISMCTS* como una solución para aplicar *MCTS* a juegos con información imperfecta. Cabe destacar que, estos investigadores también participaron en la creación del artículo usado como referencia para el algoritmo *MCTS* mencionado anteriormente.

En el artículo, se presentan los tres nuevos algoritmos de la familia *ISMCTS* de manera teórica y se realiza un análisis experimental. Cada algoritmo se acompaña de pseudocódigos de alto nivel que pueden servir como base para futuras implementaciones por parte de otros desarrolladores. Dicha información proporcionada en el artículo ha sido la única referencia utilizada para el desarrollo de las implementaciones de los algoritmos *SO-ISMCTS* y *MO-ISMCTS* en la biblioteca *pyplAI*. Esto se debe a que, a diferencia de los algoritmos *minimax* y *MCTS*, la familia de algoritmos *ISMCTS* es poco conocida, lo que se refleja en la falta de información, como se mencionó en el análisis del estado del arte de la sección anterior.

Además, es importante destacar que también se ha realizado una documentación completa que acompañará a la biblioteca. Esta documentación incluye una breve introducción sobre las funcionalidades que ofrece y dónde puede ser utilizada, así como un amplio manual de uso que proporciona instrucciones detalladas desde la importación de la biblioteca hasta la utilización de cada uno de los algoritmos. Esta documentación será de gran valor para los usuarios de la biblioteca, ya que como se ha podido observar en la sección anterior durante el análisis del estado del arte, no es algo común en las bibliotecas de este dominio.

Finalmente, una vez completadas las implementaciones y verificado el correcto funcionamiento de cada algoritmo en la biblioteca, se procedió a organizar y crear los archivos necesarios para convertir el código *Python* en una biblioteca. Se siguió la estructura de directorios específica para la creación de una biblioteca y se crearon los archivos obligatorios para la configuración de la biblioteca, como los archivos *init.py* y *setup.py*. Además de los archivos obligatorios, se agregaron archivos opcionales, como un *README.md* que contiene la documentación de la biblioteca, y un *LICENSE.md* que establece la licencia de software libre permisiva elegida para el proyecto, en este caso, la licencia *MIT*.

Una vez completados estos pasos, la biblioteca se empaquetó a través del uso de la consola de comandos, lo que generó los archivos de distribución necesarios. Estos archivos se utilizarían posteriormente en el proceso de publicación de la biblioteca en el repositorio de software oficial de *Python*, conocido como *PyPi*.

Antes de la publicación en el repositorio *PyPi*, se realizó una prueba en el repositorio de pruebas llamado *PyPi test*. El proceso de carga es similar al del repositorio oficial, pero la biblioteca solo queda visible para el desarrollador que realizó la carga. Una vez cargada, se realizaron pruebas a fondo para asegurar el correcto funcionamiento de la biblioteca y sus algoritmos. Después de verificar su funcionamiento en el repositorio de pruebas, se publicó en el repositorio oficial de *PyPi*, haciéndola pública y accesible para cualquier desarrollador. Se puede acceder a la biblioteca a través del siguiente enlace: `https://pypi.org/project/pyplAI/` o, directamente, se puede descargar e instalar la biblioteca desde la consola usando el comando `pip install pyplAI`. Una vez descargada e instalada, se puede importar en un archivo de *Python* usando `import pyplAI`.

Así mismo, se ha creado un repositorio público en *GitHub* con toda la biblioteca disponible `https://github.com/plss12/pyplAI`. Los usuarios son libres de usar la biblioteca para sus propios proyectos o colaborar en ella aportando mejoras y modificaciones. En dicho repositorio de *GitHub* también se encuentran los juegos desarrollados para probar el correcto funcionamiento de los algoritmos. Estos juegos se presentarán en la siguiente sección, casos de estudio, y serán utilizados para realizar las pruebas que se verán en la sección de experimentación y pruebas (6). De esta manera, los juegos desarrollados no solo sirven como comprobación del correcto uso de los algoritmos y ejemplos de uso, sino que también serán utilizados para la evaluación de su rendimiento.

Por último, se decidió no incluir estos juegos en la biblioteca subida a *PyPi* para hacerla más ligera y evitar dependencias externas con otras bibliotecas. Por ello, los juegos desarrollados se pueden ver en el repositorio de *GitHub* y pueden ser usados como ejemplos de uso para ver cómo ha sido utilizada la biblioteca en cada uno de ellos. Además, hay ejemplos para todos los algoritmos de la biblioteca, lo que puede ser de gran ayuda para los desarrolladores que tengan dudas sobre cómo integrar la biblioteca en sus propios juegos.

5. Casos de Estudio

En esta sección serán introducidos los juegos de mesa que han sido desarrollados para hacer uso de los diferentes algoritmos implementados en la biblioteca. La selección de juegos ha estado centrada en la búsqueda de varios juegos de información perfecta, juegos de información imperfecta y juegos de información imperfecta con movimientos parcialmente observables, estos tipos de juegos serán usados para el estudio de los algoritmos *minimax* y *MCTS*, el algoritmo *SO-ISMCTS* y el algoritmo *MO-ISMCTS* respectivamente.

Además, la selección ha estado enfocada en la búsqueda de juegos sencillos y juegos más complejos, para poder analizar el comportamiento de los distintos algoritmos en juegos de diferentes características.

Por último, cabe destacar que se incluye un caso de estudio especial para el juego del *Stratego* en el que se hace uso de un algoritmo genético para la obtención de varias configuraciones iniciales del tablero.

5.1. Algoritmos *Minimax* y *MCTS*

5.1.1. Tic-Tac-Toe

El primero de los juegos desarrollados ha sido el *Tic-Tac-Toe* o *tres en raya* en español, un juego de mesa muy simple escogido para hacer uso de los algoritmos para juegos de información perfecta, *minimax* y *MCTS*.

Para jugar a este juego solo se necesitan dos jugadores y un tablero de 3x3. Cada jugador tiene asignado un símbolo, generalmente una X o un O, y el objetivo es ser el primer jugador en colocar tres de sus símbolos en línea recta, de manera horizontal, vertical o diagonal.

El juego comienza con el tablero vacío y los jugadores se turnan para colocar su símbolo en alguna de las casillas aún vacías. Si el tablero queda completo, es decir, no hay ninguna casilla vacía, y ningún jugador ha logrado conseguir una línea recta de 3 de sus símbolos, la partida acaba en empate.

Debido a que este juego ha sido elegido para implementar el algoritmo *minimax*, se debe establecer una función de evaluación, aunque, gracias a la simplicidad del juego, se puede calcular el árbol de búsqueda completo en un tiempo de computación viable. La función heurística para la evaluación de los estados no terminales contará las líneas rectas de tres casillas con dos símbolos de un mismo jugador y un espacio vacío, en otras palabras, las posibles líneas en las que cada jugador tiene la opción de hacer un tres en raya. El resultado final de la función de heurística será la diferencia entre los posibles tres en raya para el jugador al que se le está buscando el mejor movimiento y el rival.

Gracias a su simplicidad, el *Tic-Tac-Toe* se ha convertido en el juego por excelencia para enseñar los conceptos básicos de muchos de los algoritmos de búsqueda, como se puede apreciar en la imagen 2.1. Esta es una de las razones por las que ha sido elegido este juego frente a otros juegos sencillos de información perfecta.

5.1.2. Ultimate Tic-Tac-Toe

El siguiente juego desarrollado es una variante bastante más compleja del *Tic-Tac-Toe*, llamada *Ultimate Tic-Tac-Toe* [5]. Este nuevo juego, al igual que el *Tic-Tac-Toe* es un juego de información perfecta para dos jugadores, por lo que se ha escogido para hacer uso de los algoritmos *minimax* y *MCTS*.

En este juego, se utiliza un tablero compuesto por nueve tableros de 3x3, que a su vez conforman un tablero más grande de 3x3, es decir, cada casilla de este tablero grande es un tablero de 3x3 del *Tic-Tac-Toe*. El objetivo es el mismo que en el *Tic-Tac-Toe*, es decir, conseguir una línea recta de 3 de tus símbolos en el tablero grande, pero para marcar con tu símbolo una de las casillas del tablero grande se debe ganar la partida que está dentro de la casilla. Además, cada movimiento determinará en que tablero se debe jugar en la siguiente jugada, si el tablero en el que se debe jugar esta completo o ya ha sido ganado por uno de los dos jugadores, el jugador puede elegir en que tablero va a jugar. Además, en el primer turno, el primer jugador elige libremente en el tablero que desea jugar, ya que no hay ningún movimiento previo que determine el tablero de juego.

Por ejemplo, si el primer jugador marca la casilla superior derecha en uno de los tableros pequeños, el segundo jugador, en el siguiente turno, debe jugar en el tablero superior derecho. Si este tablero estuviese completo o si ya ha sido ganado por algún jugador, el segundo jugador puede elegir marcar una de las casillas de otro tablero, siempre y cuando aún no este completo o ganado. En la siguiente imagen se puede ver este ejemplo de manera gráfica.

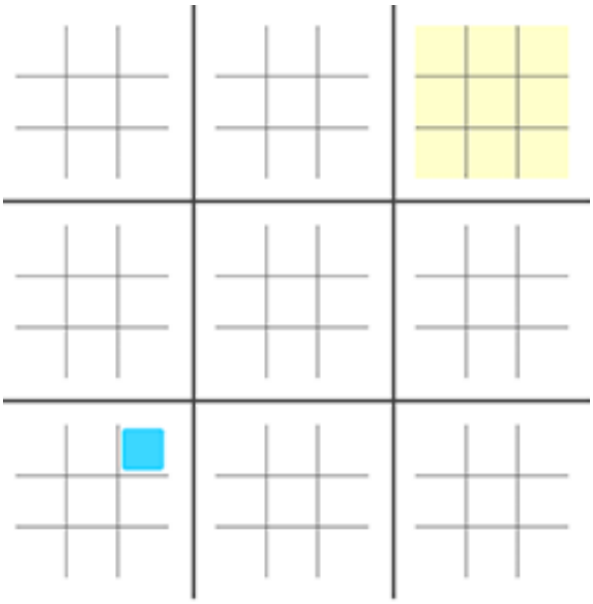

Figura 5.1: *Ultimate Tic-Tac-Toe*
Tablero con el comienzo de una partida del *Ultimate Tic-Tac-Toe*, en azul se muestra la casilla seleccionada por el primer jugador y en amarillo se destaca el tablero en el que debe jugar el segundo jugador, determinado por el movimiento anterior.

Al igual que para el *Tic-Tac-Toe*, al ser aplicado el algoritmo *minimax* en este juego, se debe crear una función heurística para la evaluación de los estados de juego no terminales. Además, debido al aumento de la complejidad respecto al juego base, esta función heurística se vuelve obligatoria al ser inviable calcular el árbol de búsqueda completo.

La función heurística usada ha sido elegida entre una de las propuestas por el estudio realizado por Eytan Lifshitz y David Tsurel, de la Universidad Hebrea de Jerusalén [5]. Esta función comprueba ciertas situaciones en el tablero y da puntos si se cumplen: se suman 5 puntos por cada tablero pequeño ganado, si se ha ganado el tablero central se suman 10 puntos, por cada tablero situado en una esquina ganado se suman 3 puntos, por cada casilla central marcada en cada uno de los tableros pequeños se suman otros 3 puntos. Además, al igual que en el *Tic-Tac-Toe*, se cuentan las líneas con opciones de ser un tres en raya, en otras palabras, las líneas que contienen 2 símbolos propios y una casilla vacía, y se otorgan 4 puntos por cada una de estas líneas en el tablero grande y 2 puntos por cada una de estas líneas en los tableros pequeños. Estas situaciones también serán comprobadas para el rival, aportando la misma puntuación negativa a la suma total.

Este juego ha sido elegido gracias a que, a pesar de ser un juego complejo, es fácil de explicar y jugar, debido en gran parte, por tener las mismas mecánicas de juego que el *Tic-Tac-Toe*. Además, uno de los puntos que ha sido decisivo para la elección de este juego, es la manera en la un movimiento no solo afecta al estado del juego presente, sino que además afecta directamente al estado del juego futuro, ya que determina el tablero en el que se deberá jugar en el siguiente turno. Esto provoca que se requieran estrategias de juego que tengan muy en cuenta las consecuencias directas de los movimientos.

5.1.3. Damas

Por último, en juegos de información perfecta, se utilizará el clásico juego de las *damas*. Un juego para dos jugadores cuyo objetivo es capturar todas las fichas del rival, llamadas peones o damas, o hacer que este no pueda realizar ningún movimiento.

En el juego de las *damas* se hace uso de un tablero de ajedrez, es decir, un tablero de 8x8 casillas alternando entre casilla negra y casilla blanca. Cada jugador tendrá 12 peones de su color, normalmente se suelen usar peones blancos y negros, pero también existen peones rojos. El primer jugador en mover es el que utiliza las fichas blancas. Los peones de cada jugador comenzarán situándose en las casillas negras de las 3 filas más cercanas a su jugador. Los peones solo se pueden mover una casilla diagonalmente hacia delante, izquierda o derecha, siempre y cuando la casilla esté libre.

Para capturar a un peón, el jugador debe mover su peón diagonalmente saltando sobre un peón rival y colocándose justo en la casilla vacía siguiente en la diagonal, si esta casilla no estuviera vacía, no se podría realizar la captura. Se pueden encadenar múltiples capturas, siempre y cuando haya casillas vacías disponibles para realizar las múltiples capturas.

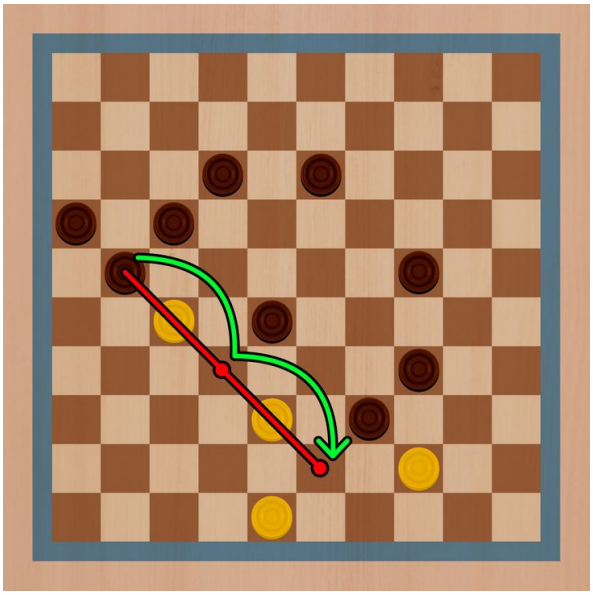

Figura 5.2: Captura peón
Tablero para el juego de las damas en el que se puede ver como se realiza una captura múltiple en las damas, más concretamente, como un peón negro puede enlazar una doble captura.

También, se puede convertir un peón en dama o reina, la cual se puede mover múltiples casillas diagonalmente en cualquier dirección. Esto se consigue al llegar con un peón al extremo opuesto del tablero.

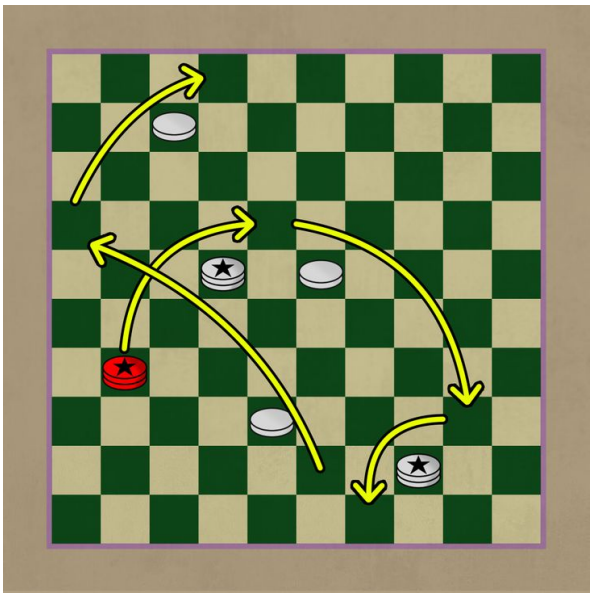

Figura 5.3: Captura dama
En este tablero se puede ver como una dama, ficha roja marcada con una estrella, puede enlazar múltiples capturas gracias a la libertad de movimiento tanto en dirección como en número de casillas.

Cabe destacar que si es posible capturar alguna pieza rival el jugador esta obligado a hacerlo, de lo contrario, si se realiza un movimiento sin captura y el rival se da cuenta, en el siguiente turno quitará la pieza que no ha realizado el movimiento para capturar del tablero y continuará jugando. Además, se debe realizar el movimiento que realice un mayor número de capturas.

Por último, si ningún jugador puede realizar un movimiento, se declarará un empate. También, se puede determinar un número limitado de turnos a jugar para que no se alargue demasiado la partida, si llegado a este turno ningún jugador ha podido vencer, la partida terminará con empate. El límite por número de turnos será utilizado en las pruebas que se realicen con los algoritmos de la biblioteca para este juego, para evitar entrar en partidas demasiado largas.

Además, al igual que los dos juegos anteriores, para que se pueda hacer uso del algoritmo *minimax* en este juego, se debe establecer una función heurística para la evaluación de los estados de juego no terminales. En este caso se ha optado por una función simple, pero eficaz, la cual compara la superioridad o inferioridad en cuanto a número de peones y número de damas, dándole mayor importancia a las damas, por su amplia libertad de movimientos. Más concretamente se calculará la diferencia entre el número de damas del jugador y las del rival y se multiplicará por 4, para darle mayor importancia, a este resultado se le sumará la diferencia entre peones del jugador y peones del rival, sin multiplicar por ningún valor.

Se ha elegido esta función frente a las múltiples posibilidades porque es simple y no requiere un conocimiento experto en el juego. Además, recoge a la perfección el objetivo del juego, es decir, capturar fichas rivales y evitar que el rival capture, dando especial importancia a convertir peones en damas y en evitar o capturar a las damas rivales.

5.2. Algoritmo *SO-ISMCTS* y Algoritmo Genético

5.2.1. Escoba

El primero de los juegos elegido para ser desarrollado y hacer uso del algoritmo *SO-ISMCTS* es el juego de la *escoba*, un juego de cartas, de baraja española, de 2 a 4 jugadores. Cada jugador comienza con 3 cartas en su mano, otras 4 cartas se colocan boca arriba en la mesa, visibles para todos los jugadores, y las cartas restantes se colocan en una pila de robo, oculta para todos los jugadores.

El juego consiste en realizar capturas utilizando una carta propia y las cartas necesarias en la mesa como para formar un grupo de 15 puntos. Si no es posible realizar ninguna captura, es decir, no existe ninguna combinación en la que una de las cartas de la mano del jugador y las cartas de la mesa se sumen 15 puntos, el jugador debe dejar una carta de su mano sobre la mesa. Siempre sea posible, es decir, mientras queden cartas en la pila de robo, después de realizar una captura o dejar una de las cartas sobre la mesa, el jugador debe robar una carta de la pila de robo. Cabe destacar que cuando se hace una captura con todas las cartas de la mesa, se le llama *escoba*, de ahí viene el nombre del juego, y se otorga un punto, por cada escoba, durante el recuento de puntos al final de la partida.

Las puntuaciones de las cartas están ligadas a su número, en otras palabras, los unos valen uno, los doses valen dos, y así sucesivamente, excepto en las figuras, cuyos valores son 8, 9 y 10 para las sotas, los caballos y los reyes respectivamente. Esto se debe a que en la baraja española no se utilizan los ocho ni los nueve.

Una vez agotada la pila de robo, se debe continuar la partida sin robar hasta que los jugadores jueguen todas las cartas de sus manos. Si todos los jugadores se quedan sin cartas en las manos y aún sigue habiendo cartas sobre la mesa, debido a que no se han podido capturar, las cartas de la mesa serán otorgadas al último jugador que haya realizado una captura. Tras agotar la pila de robos y jugar todas las cartas de las manos los jugadores, se realiza el recuento de puntos para saber quien es el ganador.

Durante el recuento de puntos cada jugador cuenta el número total de cartas capturadas, el número de oros capturados y el número de siete capturados. Se otorga un punto al jugador con más cartas capturadas, un punto al jugador con más oros capturados y un punto al jugador con más siete capturados. En caso de empate entre varios jugadores en cuanto al mayor número de cartas totales, oros o siete capturados, se otorga un punto a cada jugador. A esta puntuación, cada jugador se suma el número de escobas realizadas durante la partida. Por último, el siete de oros, también llamado *guindis*, tiene un valor especial, ya que suma un punto adicional al jugador que lo haya capturado. Gana la partida el jugador con mayor puntuación total.

Este juego ha sido elegido por varios motivos. El primero de ellos es la posibilidad de estudiar el funcionamiento del algoritmo en un juego de más de dos jugadores. El segundo motivo viene dado por la naturaleza del juego, que permite identificar fácilmente si un jugador está siguiendo una estrategia óptima, dado que es importante de realizar el mayor número de escobas posibles, capturar la mayor cantidad de oros y siete, así como obtener el siete de oros o *guindis*, es fácil de ver cuando un jugador

está tomando decisiones racionales en el juego o no. Por ejemplo, si un jugador tiene la oportunidad de hacer una escoba, pero elige realizar otro movimiento, queda claro que el jugador no está jugando de manera óptima, ya que hacer una escoba siempre es la mejor jugada posible para un jugador. Esto permitirá evaluar si el algoritmo *SO-ISMCTS* sigue buenas estrategias buscando y priorizando estos movimientos sobre otros.

5.2.2. Blackjack

El siguiente juego desarrollado para el algoritmo *SO-ISMCTS* es el famoso juego del *blackjack*, también conocido como *veintiuno*. El *blackjack* es un juego de cartas propio de los casinos en el que participan uno o más jugadores, todos ellos enfrentándose al crupier.

El objetivo del juego es obtener una mano cuyo valor sea lo más cercano a 21, pero sin pasarse. Se juega con una baraja inglesa de 52 cartas, sin los comodines, en la que el valor de cada carta está ligado a su número, excepto las figuras, que valen 10, y el uno o as, que puede valer 1 u 11, según le convenga al jugador.

El juego comienza con una apuesta igual para todos los jugadores de la mesa. Se reparten dos cartas a cada uno de los jugadores boca arriba, quedando visibles para el resto. Además, se reparten dos cartas para el crupier, de las cuales solamente se muestra una boca arriba, para que pueda ser vista por todos los jugadores, la otra queda oculta. En este punto, cada jugador puede decidir realizar cuatro opciones:

- Robar: El jugador roba una carta adicional y puede continuar robando o pasar.
- Pasar: El jugador decide no robar ninguna carta más y espera a que se resuelva la ronda.
- Doblar: El jugador dobla la apuesta inicial, pero se le obliga a robar una carta y no poder pasar ni robar de nuevo.
- Retirarse: El jugador recupera la mitad de la apuesta y pierde la posibilidad de seguir jugando esa ronda.

Si se elige robar, después de robar una carta adicional, el jugador solo puede seguir robando o pasar, ya no puede doblar ni retirarse. Si un jugador roba una carta adicional y el valor total de su mano supera los 21, queda fuera de la partida.

Una vez que los jugadores hayan tomado todas sus decisiones, el crupier revela su carta oculta y comienza a jugar. El juego del crupier se rige por reglas específicas:

- Debe seguir robando si su mano tiene un valor igual o menor a 16.
- Debe plantarse si su mano tiene un valor de 17 o más.

Si el crupier se pasa de 21, todos los jugadores que aún estén en la partida ganan automáticamente. De lo contrario, se comparan las manos de cada uno de los jugadores con la del crupier: si un jugador tiene una mano con un valor más cercano a 21 que la mano del crupier, ese jugador gana; de lo contrario, el jugador pierde. Si el valor de

la mano del jugador y la del crupier es igual, se considera un empate y se devuelve la apuesta al jugador. Si un jugador vence al crupier, recibe el doble de su apuesta.

Durante el primer reparto de cartas, puede ocurrir una jugada especial conocida como *blackjack*. Si un jugador recibe un as junto con una carta de valor 10 (10, J, Q, K) en sus dos primeras cartas, obtiene un *blackjack* y gana automáticamente, a menos que el crupier también hubiese obtenido un *blackjack*. En ese caso, sería un empate y el jugador recuperaría su apuesta. Además, obtener un *blackjack* conlleva una recompensa extra, y es que, si un jugador gana gracias a obtener un *blackjack* recibe su apuesta multiplicada por 2.5, a diferencia de la victoria normal en la que se recibe el doble de su apuesta, es decir, la apuesta multiplicada por 2.

Cabe destacar que estas son las reglas generales del *blackjack*, y existen muchas variantes con pequeñas variaciones en algunos detalles del juego. Una de las diferencias más significativas se encuentra en la frecuencia de barajado de las cartas, la cual puede variar según las reglas específicas de cada casino o modalidad de juego. Hay dos métodos principales de barajado: el barajado continuo y el barajado manual.

En el barajado continuo, se utiliza una máquina mezcladora automática que baraja constantemente las cartas después de cada ronda. Esto significa que se juega con una baraja virtual y no hay interrupciones en el juego para barajar físicamente las cartas. En este caso, no hay un momento específico en el que se baraje la baraja, ya que se mezcla de manera continua durante todo el juego.

En el barajado manual, que es más común en los casinos tradicionales, se utilizan varias barajas de cartas (generalmente entre 4 y 8 dependiendo del casino o el tipo de mesa), y se barajan de manera periódica. El momento exacto en el que se baraja la baraja puede variar, pero generalmente ocurre después de que se haya jugado un porcentaje determinado de las cartas, normalmente suele ser aproximadamente el 75 % de la baraja. De esta forma, se pueden jugar rondas rápidas sin la interrupción continua para realizar el barajado.

En el barajado manual, las cartas jugadas en una ronda no vuelven al mazo de robo, sino que se descartan para la próxima ronda. Es por ello que, en el *blackjack* se suele jugar con más de una baraja. De lo contrario, sería fácil contar las cartas que han sido descartadas y las que son visibles para calcular las probabilidades de ganar al crupier, como se puede apreciar en la famosa película *21 blackjack*.

En nuestro caso nos centraremos en el *blackjack* con todas las reglas generales explicadas anteriormente, y el barajado manual, con la posibilidad de modificar el número de mazos de la partida y con una frecuencia del barajado alrededor del 75 % de las cartas de la baraja jugadas. Se ha elegido el barajado manual en lugar del barajado continuo, para evaluar el rendimiento del algoritmo *SO-ISMCTS* conforme aumenta la información obtenida sobre las cartas descartadas a lo largo de las rondas. Esta información obtenida hace que cada vez sea menor el conjunto de información a la hora de elegir una determinación, por lo que el rendimiento del algoritmo debería mejorar considerablemente conforme avance la partida.

5.2.3. Stratego

El último juego de información imperfecta desarrollado para hacer uso del algoritmo *SO-ISMCTS* será el *Stratego*. Un juego de tablero estratégico que simula una batalla entre dos ejerctitos. En este juego, cada jugador cuenta con un conjunto de fichas de diferentes rangos, desde el rango más bajo (1) hasta el rango más alto (10), además de una bandera que debe proteger y algunas bombas que tienen características especiales. Cabe destacar que puede haber varias fichas del mismo rango, pero solamente habrá una única bandera.

El objetivo del juego es capturar la bandera del rival antes de que este capture la bandera propia, o bien, dejar al rival sin la posibilidad de realizar movimiento alguno. El tablero tiene un tamaño fijo que depende de la variante del juego, ya que,hay variantes que varían en tamaño de tablero y número de fichas. Además, en el centro del tablero suelen encontrarse lagos o algún tipo de estructura que hacen la labor de obstáculos. Estos obstáculos ocupan varias casillas quedando inutilizadas por las fichas de ambos jugadores.

Antes de comenzar el juego, cada jugador coloca estratégicamente sus fichas en su mitad de tablero, ocultando al rival el rango o tipo de cada una de las fichas.

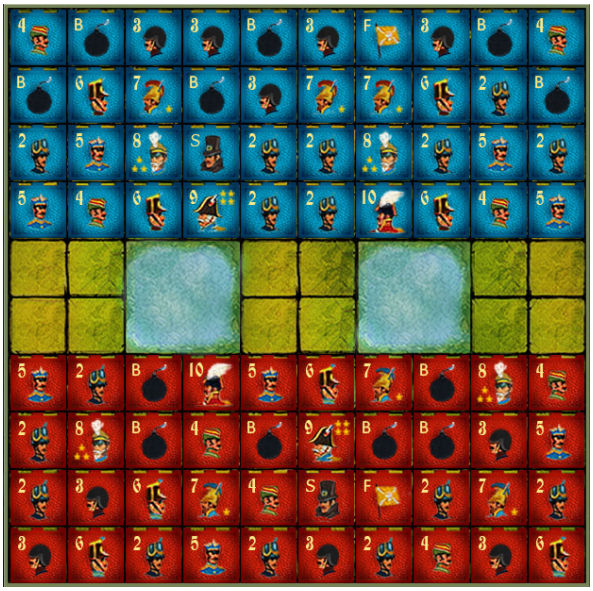

Figura 5.4: Tablero *Stratego*
Ejemplo de un tablero de *Stratego* listo para empezar una partida. En el tablero se muestran las fichas de ambos jugadores, pero en realidad, durante el juego, cada jugador solo podrá ver las posiciones de las fichas del jugador rival, sin saber qué fichas son específicamente.

En cada turno, el jugador debe elegir entre dos tipos de acciones: puede mover alguna de sus fichas o puede atacar una ficha enemiga con una de las suyas. No siempre es posible realizar un ataque a una ficha enemiga, ya que como veremos más adelante, dependerá de la posición de las fichas en el tablero.

Las fichas pueden moverse una casilla por el tablero, ya sea de manera horizontal o vertical, pero nunca en en diagonal. Existen algunas excepciones en cuanto

al movimiento. En primer lugar las bombas y la bandera no se pueden mover de su posición inicial. Además, hay una ficha especial llamada explorador, de rango 2, que puede moverse en línea recta por cualquier número de casillas siempre que no encuentre obstáculos en su camino. Cabe destacar que si un jugador realiza un movimiento de múltiples casillas con el explorador, esta ficha quedaría automáticamente identificada como un explorador, ya que es la única ficha con la habilidad de realizar este tipo de movimientos.

Para atacar a una ficha rival, se debe tener una ficha propia, sin contar la bandera o las bombas, adyacente a la ficha que se desea atacar, a excepción de los exploradores, ya que como pueden realizar movimientos de múltiples casillas, también pueden realizar ataques a distancia. Un ataque se divide en atacante, el que realiza el ataque, y defensor, el que recibe el ataque. Gana la batalla la ficha con mayor rango, y la ficha perdedora es capturada y eliminada del juego.

Si la ficha atacante gana la batalla se coloca en la posición de la ficha defensora, si es la defensora la que gana la batalla, mantiene la posición en la que estaba. Además, si en un ataque ambas fichas tienen el mismo rango, ambas son capturadas y eliminadas del tablero. Cabe añadir, que tras un ataque queda revelada la identidad de la ficha ganadora, debido a que es necesario mostrar el rango de cada una de las fichas de la batalla para saber cual es la ganadora.

Al igual que con el movimiento, hay algunas excepciones en las reglas de ataque. Si una espía, de rango 1, ataca a un mariscal, de rango 10, el espía gana y captura y elimina al mariscal. Esta regla solo se aplica cuando el espía es el atacante y el mariscal es el defensor, de lo contrario, se aplican las reglas normales y el mariscal ganaría la batalla por su superioridad en el rango.

Otro excepción para el ataque ocurre con las bombas. Estas fichas no pueden atacar ni moverse, pero si son atacadas, captura y eliminan a la ficha atacante, manteniéndose la bomba en su posición. Solo hay una ficha que puede ganar el ataque a una bomba. El minador, de rango 3, puede desactivar la bomba y ganar la batalla, capturando y eliminando la bomba del tablero y ocupando su posición. Además, si un jugador ataca y descubre que la ficha objetivo es la bandera, ese jugador gana la partida.

En nuestro caso, se ha elegido una variante específica del *Stratego*. Esta variante hace uso de 30 fichas para cada jugador y un tablero de dimensiones 10x8. A continuación, se mistara una tabla con todos los tipo de fichas, mostrando el rango de estas así como la cantidad de estas fichas que tendrá cada uno de los jugadores en esta variante seleccionada del *Stratego*.

Ficha	Rango	Cantidad
Espía	1	1
Explorador	2	5
Minador	3	4
Sargento	4	2
Teniente	4	2
Capitán	4	3
Comandante	4	3
Coronel	4	2
General	4	1
Mariscal	4	1
Bandera	-	1
Bomba	-	5

Por último, se ha optado por implementar un límite de turnos, lo que significa que si ningún jugador logra la victoria antes de que se alcance dicho límite, el juego terminará en un empate. Esta limitación en la duración de las partidas tiene como objetivo evitar que el algoritmo entre en simulaciones de partidas demasiado largas, lo cual podría afectar al rendimiento del algoritmo. Gracias a limitar la duración de las partidas, el algoritmo podrá realizar un mayor número de simulaciones por iteración, por lo que debería mejorar su rendimiento.

Algoritmo Genético

La colocación inicial del tablero supone un gran problema en el juego del *Stratego*. Uno de los factores más importantes para ganar la partida es la configuración inicial del tablero. Por ejemplo, si no se coloca la bandera en una zona protegida, será fácil que el rival la encuentre. Por ello, se suele colocar la bandera al final del tablero, protegida por la colocación de bombas cercanas o fichas con un gran rango. Esto hace que elegir una configuración inicial de manera aleatoria no sea una buena solución.

Para obtener una buena configuración inicial, se planteó tomar la colocación de cada ficha como un movimiento del juego. De esta forma, la partida comenzaría con el tablero vacío y cada jugador, en su turno, debería elegir una ficha y la posición en la que la colocaría. Así, se podría utilizar el algoritmo *SO-ISMCTS* para obtener una colocación inicial. Sin embargo, este planteamiento presenta un problema, y es que la longitud del árbol de búsqueda crecería de manera excesiva, ya que los primeros 60 movimientos de la partida (30 piezas colocadas por cada jugador) solo servirían para formar el tablero inicial. Por lo tanto, el algoritmo *SO-ISMCTS* necesitaría mucho tiempo de computación para calcular un árbol de búsqueda lo suficientemente profundo como para obtener un buen posicionamiento de las fichas. Debido a esta limitación, se descartó esta solución.

Después de descartar la solución anterior, se llegó a la conclusión de que usar algoritmos genéticos sería una buena alternativa para calcular un conjunto de configuraciones iniciales. De esta manera, el algoritmo *SO-ISMCTS* podría elegir aleatoriamente una de las configuraciones iniciales obtenidas con el algoritmo genético al comienzo de las partidas. Esto aseguraría tener una buena configuración inicial sin necesidad de

utilizar una configuración aleatoria y evitando el cálculo de la configuración inicial con el algoritmo *SO-ISMCTS* al principio de cada partida, lo cual llevaría mucho tiempo de cálculo, como ya hemos mencionado anteriormente.

El algoritmo genético implementado utiliza una población inicial compuesta por un número predefinido de cromosomas elegido por el usuario. Cada cromosoma representa una configuración inicial del tablero de juego para un jugador y está formado por una lista de genes, donde cada gen es un número que representa una ficha del juego.

Cada número en el cromosoma está relacionado con el rango de la ficha que representa. Por ejemplo, si hay 5 exploradores por jugador y cada explorador tiene un rango de 3, entonces en el cromosoma correspondiente a ese jugador el número 3 se repetirá 5 veces para representar los 5 exploradores. Además, las fichas sin rango, como lo son las bombas y la bandera, han sido relacionadas con un número diferente al de las demás fichas para que también se vean representadas en cada cromosoma.

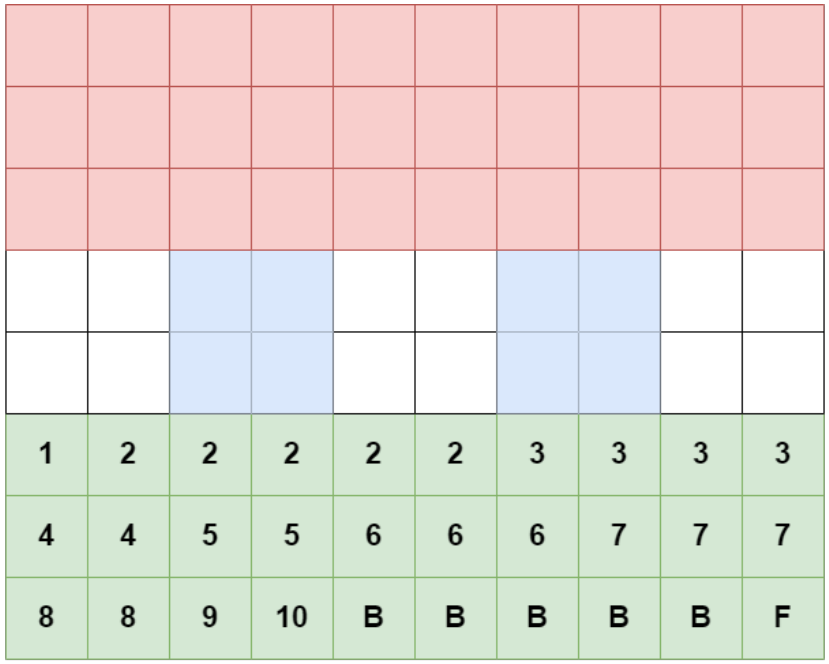

Figura 5.5: Cromosoma de un tablero de *Stratego*
En esta figura se puede ver un tablero completo de *Stratego*, en el que la zona roja es la zona del jugador enemigo, la zona azul es la zona de agua y la zona verde es la zona del jugador. En la zona verde se puede ver una configuración inicial de fichas, en la que cada número representa el rango, las **B** las bombas y la **F** la bandera. Justo abajo del tablero se muestra el cromosoma que representa esta configuración inicial.

La evaluación de los cromosomas se realiza mediante una función de evaluación

que calcula la recompensa acumulada obtenida después de simular 30 partidas utilizando la configuración inicial representada por la lista de genes del cromosoma y otra configuración inicial aleatoria distinta para cada una de las partidas. Las partidas se dividen en dos grupos de 15 partidas cada uno, alternando los jugadores iniciales y ofreciendo igualdad de oportunidades a ambas configuraciones iniciales.

Esta evaluación por medio de simulaciones permite medir el desempeño de cada configuración inicial y determinar qué cromosomas presentan mejores resultados en términos de recompensa acumulada. De esta manera, se busca encontrar la configuración óptima que maximice la recompensa obtenida en el juego.

Una vez que la población inicial ha sido creada y evaluada, se calcula el número de individuos que serán cruzados en cada generación. Este cálculo se realiza utilizando el tamaño total de la población y una proporción de cruce, que es una constante definida por el usuario. La proporción de cruce determina la cantidad de cromosomas que se seleccionarán para el cruce y la cantidad de cromosomas que no serán seleccionados.

Tras obtener el número de cromosomas que se cruzarán y que no se cruzarán, en cada una de las generaciones se seleccionan dos poblaciones. Estas poblaciones están formadas por cromosomas de la población inicial en el caso de ser la primera generación, o por los cromosomas de la población resultante de la generación anterior en generaciones sucesivas.

Es importante destacar que el tamaño de las poblaciones seleccionadas cruce se basa en los números calculados previamente. Si la población inicial tiene un total de 100 cromosomas, de los cuales se ha calculado que van a ser cruzados un total de 40, la población que será cruzada estará formada por 40 cromosomas y la población que no será cruzada estará formada por 60 cromosomas. Esto hace que entre ambas poblaciones se forme una población del tamaño de la población inicial.

Para la selección de cromosomas de una población se utiliza un tipo de selección probabilística conocido como selección por ruleta. Esta estrategia de selección permite determinar qué cromosomas de la población serán seleccionados para continuar en las próximas generaciones, favoreciendo a aquellos individuos con mejores valores en la función de evaluación pero brindando oportunidades a los demás cromosomas.

En la selección por ruleta, se asigna a cada cromosoma una probabilidad de ser seleccionado proporcional a su valor de en la función de evaluación, cuanto mayor sea este valor en comparación con los demás cromosomas, mayor será su probabilidad de ser elegido. Esto significa que los cromosomas con una mayor aptitud tienen más posibilidades de ser seleccionados para la reproducción y, por lo tanto, sus características beneficiosas tienen más probabilidades de transmitirse a las generaciones futuras.

Es importante destacar que este tipo de selección es adecuado para problemas de maximización, es decir, cuando se busca maximizar la función de evaluación. La selección por ruleta se basa en la idea de que los individuos más aptos tienen una mayor probabilidad de supervivencia en términos de selección y reproducción. Sin embargo, en problemas de minimización, donde se busca minimizar la función de evaluación, se pueden utilizar estrategias de selección alternativas, como la selección por torneo.

Una vez han sido creadas las poblaciones mediante la selección por ruleta de sus

cromosomas, se procede a realizar el cruce entre los cromosomas de la población seleccionada. Este cruce se realiza mediante la combinación de parejas de cromosomas de esta población, generando así dos cromosomas hijos o descendientes. Existen diferentes métodos para realizar los cruces, sin embargo en nuestro caso específico se necesita un método que solamente altere el orden de los genes, manteniendo los mismos genes.

Lo que buscamos, en realidad, es una permutación de los genes. De lo contrario, si se intercambiara alguno de los genes, se podrían generar cromosomas que representarán configuraciones iniciales inválidas para el *Stratego*, ya que el número y el tipo de fichas es siempre fijo. Por ejemplo, siempre hay una sola bandera, por lo que si un cruce intercambiase la ficha de la bandera por cualquier otra ficha, el cromosoma resultante no representaría una configuración inicial válida para una partida de *Stratego*. Por lo tanto, es importante utilizar un método de cruce que altere el orden de los genes, pero que mantenga su integridad, es decir, una permutación de los genes.

Para cumplir con la propiedad de permutación, se ha optado por implementar el cruce basado en orden. Este tipo de cruce, dado dos cromosomas padres, selecciona aleatoriamente dos puntos de corte. Utilizando estos puntos de corte, se obtiene el segmento comprendido entre el primer y segundo punto de corte del primer cromosoma padre y se copia en el primer cromosoma hijo. Para completar el primer cromosoma hijo, empezando desde el segundo punto de corte, se copian los genes que aún no han sido añadidos a él siguiendo el orden en que aparecen en el segundo cromosoma padre. Si es necesario, se vuelve a empezar por el principio de la lista de genes del segundo cromosoma padre. Para crear el segundo hijo, se sigue el mismo procedimiento, pero se intercambian los padres.

Este cruce presenta un problema para nuestros cromosomas, ya que el cruce basado en orden está diseñado para realizar permutaciones sin repetición, como podría ser en una lista con números del 1 al 10. Sin embargo, en nuestro caso, la lista de genes contiene números repetidos, por ello se necesitaría un método de cruce diseñado para permutaciones con repetición.

Para solucionar este problema, se ha creado una tabla de frecuencias que contiene el número de veces que cada número debe aparecer en el cromosoma hijo resultante del cruce. Con esta tabla de frecuencias, se puede adaptar el cruce basado en orden para permutaciones con repetición. En lugar de comprobar si un número ya ha sido añadido al nuevo cromosoma hijo, ahora se comprueba si el nuevo cromosoma hijo tiene la cantidad necesaria de este número. Si es así, se pasa al siguiente gen, de lo contrario, se añade este gen, aunque ya haya sido añadido un número igual anteriormente. Con esta solución, se garantiza un cruce cuyo resultado es una configuración inicial valida.

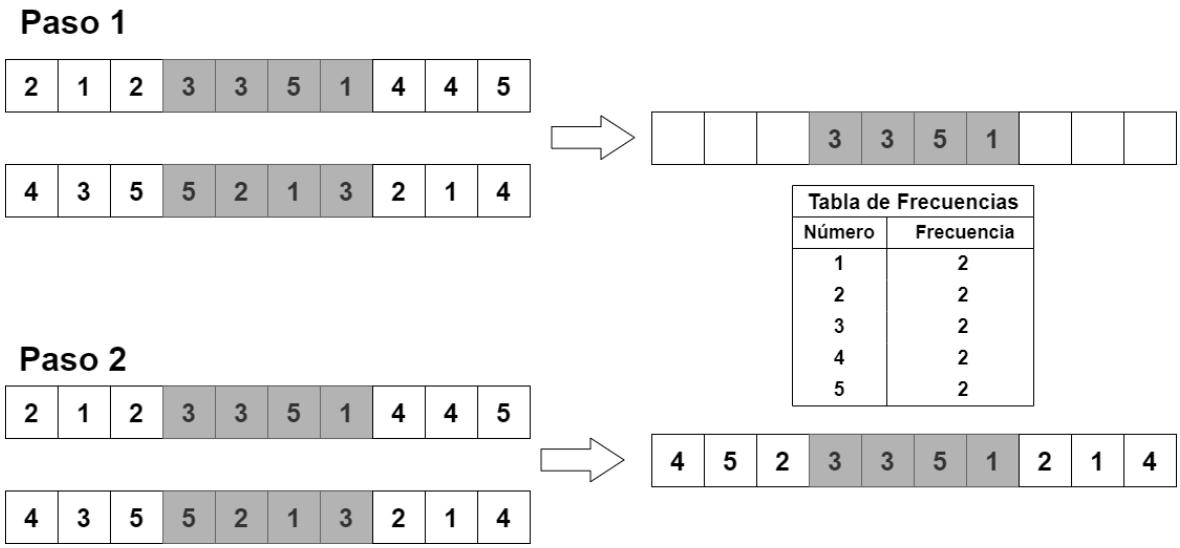

Figura 5.6: Cruce Basado en Orden con Tabla de Frecuencias
En esta figura se pueden ver los dos pasos para realizar un cruce basado en orden entre dos cromosomas, añadiendo la modificación del uso de una tabla de frecuencia, con la que se comprueba si ya se ha cumplido la frecuencia en el cromosoma hijo para añadir o no el número del cromosoma padre.

Después de haber realizado todos los cruces entre los cromosomas de la población seleccionada, se genera una nueva población combinando los descendientes resultantes de los cruces y la población con los cromosomas que no fueron seleccionados para el cruce. Una vez creada esta nueva población, se aplica una mutación a algunos de los cromosomas, cuya probabilidad de mutación será una constante proporcionada por el usuario. La mutación que se utiliza es la de intercambio, en la cual se seleccionan dos genes de un cromosoma y se intercambian sus posiciones. Esta mutación cumple con la propiedad de permutación, lo que garantiza que no se obtendrán configuraciones iniciales inválidas. Posteriormente, se realiza una nueva evaluación para obtener el nuevo valor de la función de evaluación de los cromosomas de la nueva población.

El proceso completo del algoritmo genético incluye la selección de individuos, mutación de los seleccionados, creación de una nueva población, y mutación y evaluación de los individuos de esa población. Cada repetición de este proceso se conoce como una generación y el resultado final es una nueva población mutada y evaluada. Cabe destacar que el usuario es el que elige cuántas generaciones se ejecutarán, al igual que con el número de individuos en cada población y las probabilidades de mutación y cruce. A continuación, se mostrará un pseudocódigo de alto nivel del algoritmo genético:

Algorithm 1 Algoritmo Genético

Require: Tamaño de población N, número de generaciones G, proporción de cruce p_c, probabilidad de mutación p_m

Ensure: Mejores 10 individuos de la última población P

$g \leftarrow 0$

$P(g) \leftarrow$ Inicialización población aleatoria de tamaño N

Evalúa $P(g)$

Calcular el número de individuos a cruzar n multiplicando la proporción de cruce p_c por el tamaño de la población N: $n \leftarrow \lfloor p_c N \rfloor$

while $g < G$ **do**

$P1 \leftarrow$ Selección por ruleta de n individuos de $P(g)$ para cruzamiento

$P2 \leftarrow$ Selección por ruleta de $N - n$ individuos de $P(g)$ para no ser cruzados

$P3 \leftarrow$ Cruza $P1$

$P4 \leftarrow$ Unión de $P2$ y $P3$

$P(g+1) \leftarrow$ Mutación de los individuos de $P4$ con probabilidad p_m

Evalúa $P(g+1)$

$g \leftarrow g + 1$

end while

Seleccionar los 10 mejores individuos de $P(g)$

return Lista de los 10 mejores individuos

Es importante tener en cuenta que cuanto mayor sea el número de individuos por población y el número de generaciones, mejores serán los resultados. Sin embargo, esto también conlleva un aumento en el costo y tiempo de computación.

En nuestro caso, el algoritmo genético implementado se ejecutará con una población de 100 individuos durante 1000 generaciones. Como resultado, se obtendrán los 10 mejores individuos según el valor de la función de evaluación en la generación final. Estas 10 configuraciones iniciales del tablero de un jugador se guardarán en un archivo y serán utilizadas por el algoritmo *SO-ISMCTS* en los inicios de sus partidas. Estos resultados serán expuestos en detalle en la siguiente sección de experimentación y pruebas.

Se ha elegido desarrollar el juego del *Stratego* para su estudio con el algoritmo *SO-ISMCTS* por su gran complejidad estratégica, la cual se ve reducida conforme se revela información durante el desarrollo de la partida. Además, ofrece una gran oportunidad para poder realizar un caso de estudio especial utilizando algoritmos genéticos con el objetivo de encontrar configuraciones iniciales óptimas del tablero. Esto permitirá ver como se comporta el algoritmo *SO-ISMCTS* al hacer uso y enfrentarse a diferentes configuraciones iniciales aleatorios u obtenidas mediante el uso del algoritmo genético.

5.3. Algoritmo *MO-ISMCTS*

5.3.1. Phantom (4,4,4)

El primero de los juegos de información imperfecta con movimientos parcialmente observables elegido es el *Phantom (4,4,4)*. Los juegos (m, n, k) son juegos de dos jugadores con tableros de m x n casillas. Los jugadores se turnan para marcar una de las casillas del tablero. El primer jugador en lograr una línea horizontal, vertical o diagonal de k casillas es el ganador del juego. Por ejemplo, el *Tic-Tac-Toe*, visto anteriormente, es un juego (3, 3, 3), ya que, se juega en un tablero 3x3 y el primero en ganar es el jugador que logre una linea horizontal, vertical o diagonal de 3 casillas.

Un juego *Phantom (m, n, k)* es un juego (m, n, k) en el que ningún jugador puede ver donde ha marcado el jugador rival. Si un jugador intentase marcar una casilla ocupada por el rival, se le comunica que es un movimiento inválido, ya que esa casilla está ocupada por el rival. Además, se le permite elegir una nueva casilla para marcar sin ningún tipo de penalización. Realizar movimientos inválidos es la única manera de obtener información sobre los movimientos previos del rival.

Cuando un jugador realiza un movimiento inválido, el rival no puede observar en que casilla ha ocurrido este movimiento inválido, y por lo tanto, el jugador ha obtenido información acerca de la posición de alguna de las marcas del rival, sin que el rival sepa cuál es la marca conocida ahora por el jugador.

Es importante destacar que, para este tipo de juegos, todos los movimientos realizados por un jugador son parcialmente observables para el rival. Esto significa que cuando un jugador marca una casilla o realiza un movimiento inválido, el rival solamente observa que tipo de movimiento se ha realizado, pero de esta observación no obtiene ningún tipo de información sobre cuál es la casilla marcada. Por lo tanto, todos los posibles movimientos del jugador son indistinguibles para el rival, esto lo hace perfecto para hacer uso del algoritmo *MO-ISMCTS*.

Por esta razón, se ha elegido el *Phantom (4, 4, 4)* como juego para analizar el comportamiento del algoritmo *MO-ISMCTS*. El tamaño del tablero es lo suficientemente grande para que los algoritmos de búsqueda puedan hacer uso de múltiples determinaciones y crear árboles de búsqueda variados. Esto permitirá realizar una evaluación precisa del rendimiento del algoritmo.

5.3.2. Holjjak

El último juego de información imperfecta con movimientos parcialmente observables elegido es el *Holjjak*, un juego de canicas tradicional originario de Corea. Para jugar a este juego se necesitan dos jugadores y un número determinado de canicas que se repartirán de manera igualitaria para cada jugador.

Este juego consiste en que un jugador elige un número de canicas a apostar entre todas las canicas que pose. El rival debe adivinar si la cantidad que ha elegido es par o

impar, si el rival acierta se lleva las canicas seleccionadas por el rival, de lo contrario, si falla, debe dar ese mismo número de canicas al otro jugador.

El juego va turnando a los jugadores entre las distintas posiciones, es decir, entre elegir el número de canicas a apostar o adivinar si el número de canicas elegido por el rival es par o impar. Un jugador gana cuando ha conseguido todas las canicas del rival, es decir, el rival queda con cero canicas en su propiedad.

Este juego es un caso de estudio especial, ya que, claramente, se pueden diferenciar dos tipos de movimientos. El primero de ellos es la elección del número de canicas que elegimos. Para el jugador que realiza esta acción el juego es un juego de información perfecta, ya que el jugador sabe el número exacto de canicas que posee el rival, por lo que no hay ningún tipo de información oculta para el jugador que debe realizar la acción de elegir el número de canicas a apostar.

Por otro lado, el otro tipo de movimiento, es cuando el rival debe adivinar si el número de canicas seleccionadas por el rival es par o impar. En este caso, para el jugador que debe adivinar, el juego es un juego de información imperfecta, ya que está oculta la información sobre el número de canicas que ha elegido el rival.

Además, para el jugador que debe adivinar, el movimiento que realiza el rival al elegir el número de canicas a apostar es parcialmente observable, ya que la observación que obtiene es que el rival elige un número de canicas, pero le falta información sobre cuál es el número de canicas seleccionado. Esto quiere decir que desde el punto de vista del jugador, que debe adivinar, no hay ninguna diferencia entre que el rival coja un número determinado de canicas u otro totalmente diferente, ya que la observación que obtiene para ambos movimientos es exactamente igual.

Por último, cabe destacar que este juego ha sido elegido por su simpleza pero sobre todo por su peculiaridad al ser una mezcla entre un juego de información perfecta y un juego de información imperfecta con movimientos parcialmente observables. Gracias a esta peculiaridad, se va a poder estudiar el uso alternado entre el algoritmo para juegos de información perfecta, *MCTS*, y el algoritmo para juegos de información imperfecta con movimientos parcialmente observables, *MO-ISMCTS*, ya que se utilizará el algoritmo *MCTS*, para los turnos en los que se deba elegir el número de canicas a apostar, y el algoritmo *MO-ISMCTS*, para los turnos en los que se deba adivinar si el número de canicas elegido por el rival es par o impar.

6. Experimentación y Pruebas

En esta sección se describen en detalle las pruebas realizadas para evaluar el rendimiento de los distintos algoritmos aplicados a los juegos elegidos, que se explicaron en la sección anterior. Las pruebas varían en función del algoritmo utilizado, por lo que se emplean diferentes pruebas para cada uno de ellos. En los próximos apartados se describen en detalle las pruebas específicas utilizadas para cada uno de los algoritmos, incluyendo los resultados obtenidos.

Todas estas pruebas han sido realizadas en la misma computadora, cuyas especificaciones son: procesador I7-12700H 2.7 GHz, tarjeta gráfica RTX 3050 Ti, 16 GB de memoria RAM y disco duro SSD de 500 GB de almacenamiento.

6.1. *Minimax* y *MCTS*

Para comparar el rendimiento de los algoritmos *minimax* y *MCTS*, se han realizado enfrentamientos entre ellos, pero previamente se ha llevado a cabo un estudio para determinar el tiempo medio de computación que utiliza *minimax* para calcular un movimiento dada una profundidad de búsqueda. Este estudio se realiza individualmente para cada juego, ya que el tiempo de cálculo varía con la complejidad del juego. Para obtener la media de tiempo de computación utilizado por *minimax*, con una profundidad dada, se simula un número determinado de partidas y se hace la media de los tiempos de computación de las ejecuciones del algoritmo *minimax*.

El objetivo de este estudio previo, es establecer un tiempo de computación para *MCTS* que sea equivalente al tiempo utilizado por *minimax* para la profundidad seleccionada para el enfrentamiento, de esta forma ambos algoritmos competirán en igualdad de condiciones.

Después de obtener el tiempo de computación equivalente para *MCTS*, se llevan a cabo varios enfrentamientos en los que se prueban distintas profundidades y, por lo tanto, distintos tiempos de computación para *MCTS*. En cada uno de los enfrentamientos se juegan varias partidas enfrentando ambos algoritmos, alternando el algoritmo que empezará como jugador inicial para asegurar la igualdad de condiciones. Finalmente, se obtienen los resultados de partidas ganadas por cada algoritmo y los empates para cada enfrentamiento.

El objetivo de esta prueba es comparar el rendimiento de ambos algoritmos enfrentándolos entre sí, y evaluar cómo evoluciona el desempeño de cada algoritmo en función de diferentes tiempos de computación y profundidades. A continuación, se presentarán los resultados obtenidos en ambas pruebas para cada uno de los juegos seleccionados como casos de estudio:

6.1.1. Tic-Tac-Toe

En el juego del *Tic-Tac-Toe*, se realizó un estudio para determinar el tiempo de computación medio que utiliza el algoritmo *minimax* dado un conjunto de profundidades (3, 4, 5, 6) y simulando 100 partidas para cada una de las profundidades. A continuación se muestra una tabla con los resultados:

Profundidad	Tiempo medio de computación
3	0.06
4	0.14
5	0.23
6	0.8

Después de determinar los tiempos de computación para *MCTS* equivalentes a las distintas profundidades en el estudio previo, se llevaron a cabo 100 enfrentamientos entre los algoritmos *minimax* y *MCTS* utilizando el mismo conjunto de profundidades y los tiempos de computación obtenidos para *MCTS*. A continuación se presentan los resultados obtenidos:

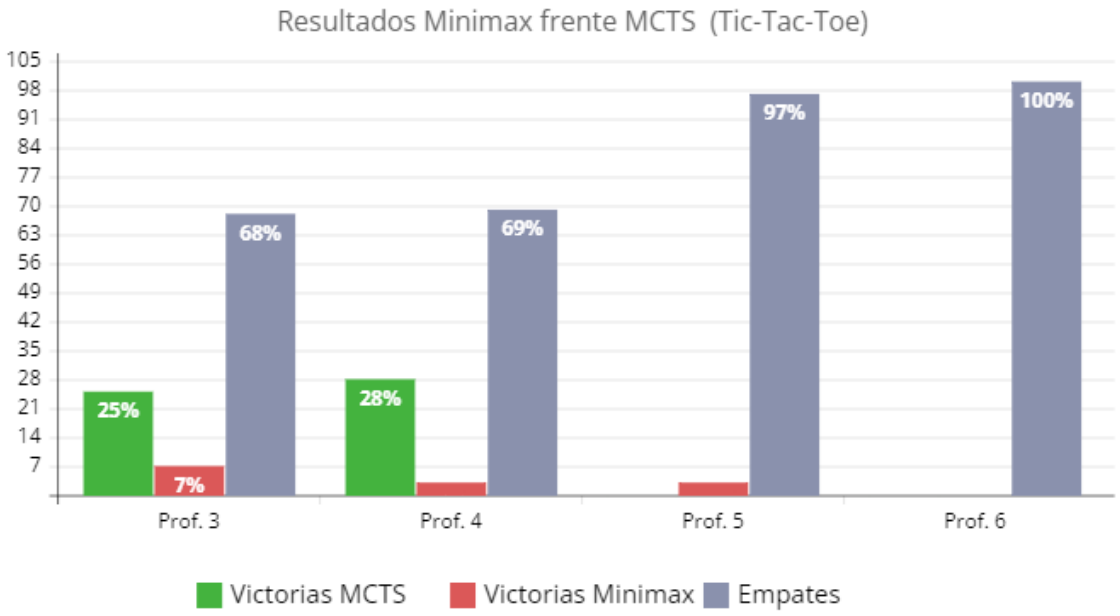

Figura 6.1: Resultados *MCTS* vs *Minimax* (*Tic-Tac-Toe*)

Como cabría esperar, a medida que aumenta el nivel de profundidad en el algoritmo *minimax* y, por tanto, el tiempo de computación en el algoritmo *MCTS*, se producen un mayor número de empates. En concreto, se observa que el porcentaje de empates alcanza el 100 % cuando se llega a una profundidad de 6. Es interesante destacar que, a pesar de esto, el algoritmo *MCTS* es capaz de ganar en algunas ocasiones en niveles más bajos de profundidad (3 y 4). Sin embargo, en niveles de profundidad más altos, el algoritmo *minimax* es capaz de prever y evitar las estrategias ganadoras de *MCTS*.

6.1.2. Ultimate Tic-Tac-Toe

En el juego del *Ultimate Tic-Tac-Toe*, se realizó el mismo estudio para determinar el tiempo de computación medio que utiliza el algoritmo *minimax* y simulando 100 partidas para cada una de las profundidades dadas (2, 3, 4, 5). Los resultados de este estudio se muestran en la siguiente tabla:

Profundidad	Tiempo medio de computación
2	0.25
3	0.79
4	4.02

Después de obtener los tiempos de computación equivalentes para *MCTS*, se realizaron 100 enfrentamientos entre los algoritmos *minimax* y *MCTS* utilizando el mismo conjunto de profundidades y los tiempos de computación obtenidos para *MCTS*. A continuación se muestran los resultados obtenidos:

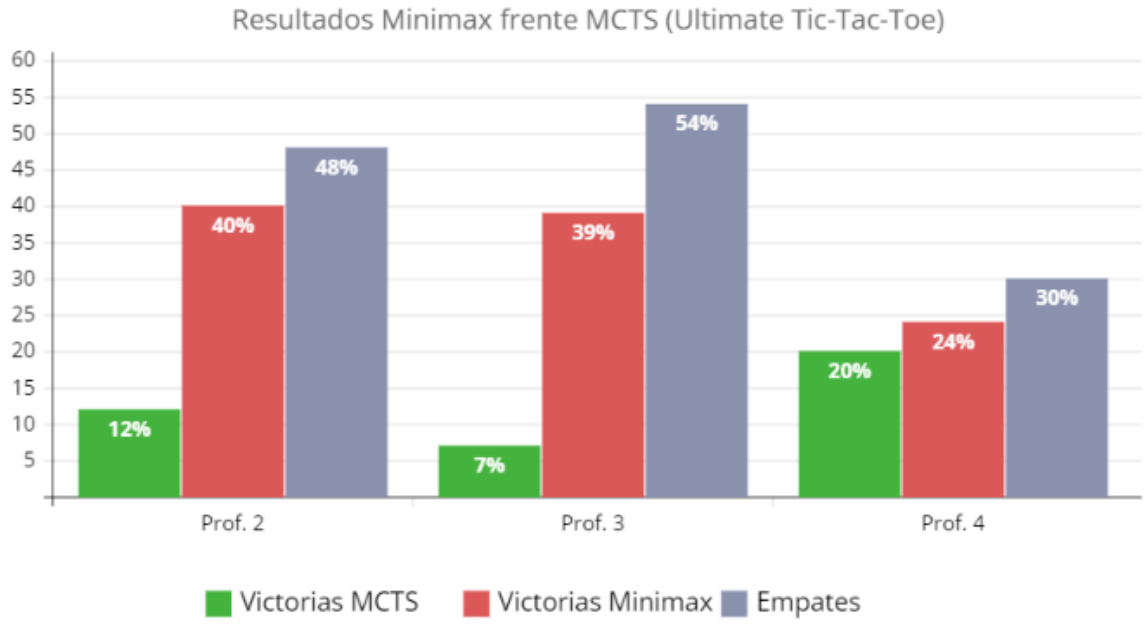

Figura 6.2: Resultados *MCTS* vs *Minimax* (*Ultimate Tic-Tac-Toe*)

En estos resultados se puede observar cómo el algoritmo *minimax* tiene una clara ventaja sobre el algoritmo *MCTS* en niveles de profundidad bajos (2 y 3). Esto puede ser debido a la gran complejidad del juego, ya que aunque haya múltiples posibles movimientos, el algoritmo *minimax* obtiene buenos movimientos basados en el valor de la función de evaluación. Por otro lado, para el algoritmo *MCTS*, puede resultar más complicado obtener un buen movimiento al estar limitado a un tiempo de computación reducido.

Sin embargo, se puede apreciar como en el nivel de profundidad 4 los resultados son bastante igualados entre las victorias de *MCTS* y *minimax*. Esto podría deberse

al aumento considerable en el tiempo de cómputo utilizado para el algoritmo *MCTS*, lo que le permite explorar más a fondo el árbol de búsqueda.

6.1.3. Damas

Por último, se realizó el estudio para determinar el tiempo de computación medio que utiliza el algoritmo *minimax* para el juego de las *damas*, con la simulación de 100 partidas para cada una de las profundidades dadas (2, 3, 4, 5). La siguiente tabla muestra los resultados obtenidos:

Profundidad	Tiempo medio de computación
2	0.15
3	0.32
4	0.85
5	1.77

Después de obtener los tiempos de computación equivalentes para *MCTS*, se realizaron 100 enfrentamientos entre los algoritmos *minimax* y *MCTS* haciendo uso del mismo conjunto de profundidades y los tiempos de computación obtenidos en el estudio para *MCTS*. A continuación se presentan los resultados obtenidos:

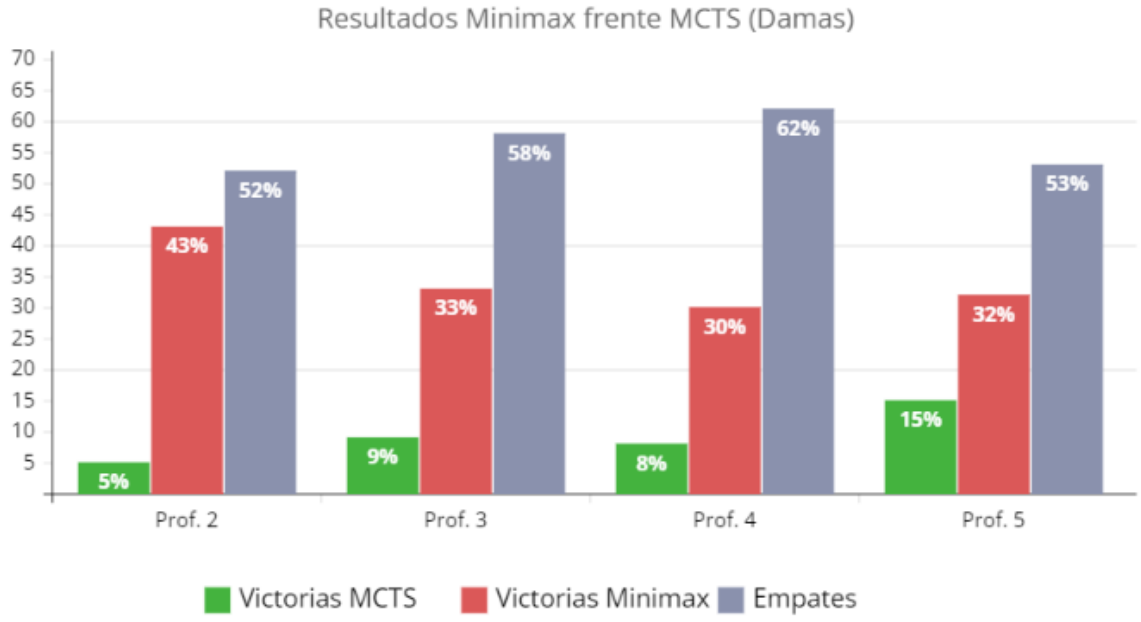

Figura 6.3: Resultados *MCTS* vs *Minimax* (*Damas*)

En la gráfica se puede apreciar la clara superioridad del algoritmo *minimax* sobre el algoritmo *MCTS*. Esta superioridad puede deberse a la heurística utilizada por *minimax*, la cual se basa en la diferencia de fichas con el rival, lo que hace que el algoritmo busque jugadas de captura y evite perder fichas. Durante los enfrentamientos, se ha observado cómo el algoritmo *minimax* es superior en el final de las partidas, incluso

llegando a provocar empates en partidas en las que el rival tenía una clara superioridad en cuanto al número de fichas.

Además, en la gráfica se puede apreciar cómo esta ventaja del algoritmo *minimax* sobre el algoritmo *MCTS* se va reduciendo a medida que se aumenta la profundidad de *minimax* y, por lo tanto, se incrementa el tiempo de cómputo de *MCTS*. Esto puede deberse a que el incremento en el tiempo de cómputo de *MCTS* permite a este explorar más a fondo el árbol de búsqueda.

6.2. *SO-ISMCTS* y Algoritmo Genético

Para evaluar el rendimiento del algoritmo *SO-ISMCTS* se han diseñado distintas pruebas específicas para cada uno de los juegos seleccionados como casos de estudio. Esto se debe a que cada juego es muy distinto entre sí, por lo que no tendría sentido diseñar una prueba común para todos. A continuación, se entrará en detalle sobre las pruebas que realizadas para cada uno de los juegos y se mostrarán los resultados de las mismas.

6.2.1. Escoba

Para el juego de la *escoba*, se han diseñado dos pruebas. La primera consiste en realizar una serie de enfrentamientos entre el algoritmo *SO-ISMCTS* y un agente que elige movimientos aleatorios. En esta prueba, se realizarán varios enfrentamientos con distintos tiempos de computación para el algoritmo *SO-ISMCTS*. Cabe destacar que el jugador inicial irá rotando para mantener la igualdad de condiciones en este aspecto. El objetivo es evaluar la evolución del rendimiento del algoritmo *SO-ISMCTS* dado distintos tiempos de computación teniendo una ventaja clara sobre el oponente.

La segunda prueba está diseñada para evaluar el rendimiento del algoritmo *SO-ISMCTS* en partidas de más de dos jugadores. En esta prueba, cada partida tendrá cuatro jugadores, dos de los cuales realizarán movimientos aleatorios y los otros dos utilizarán el algoritmo *SO-ISMCTS*. Los dos jugadores que utilizan el algoritmo *SO-ISMCTS* tendrán el mismo tiempo de computación para el cálculo de los movimientos. Además, para mantener la igualdad de condiciones en cuanto al orden de juego, se irá rotando el jugador inicial y el orden de turno de los demás jugadores.

El objetivo de estas dos pruebas es evaluar el rendimiento del algoritmo *MO-ISMCTS* en diferentes situaciones del juego de la *escoba*, incluyendo aquellas en las que el algoritmo se enfrenta a un solo rival sobre el que se tiene ventaja y también cuando se enfrenta a múltiples oponentes, algunos en igualdad de condiciones y otros en una relativa inferioridad. Además, se busca analizar la evolución del rendimiento del algoritmo con diferentes tiempos de computación.

Para la primera prueba, se han seleccionado cuatro tiempos de computación (0.25, 0.5, 1, 2 segundos) y se han llevado a cabo 100 enfrentamientos entre el algoritmo *SO-ISMCTS* y un agente aleatorio para cada uno de estos tiempos. A continuación se presentan los resultados obtenidos:

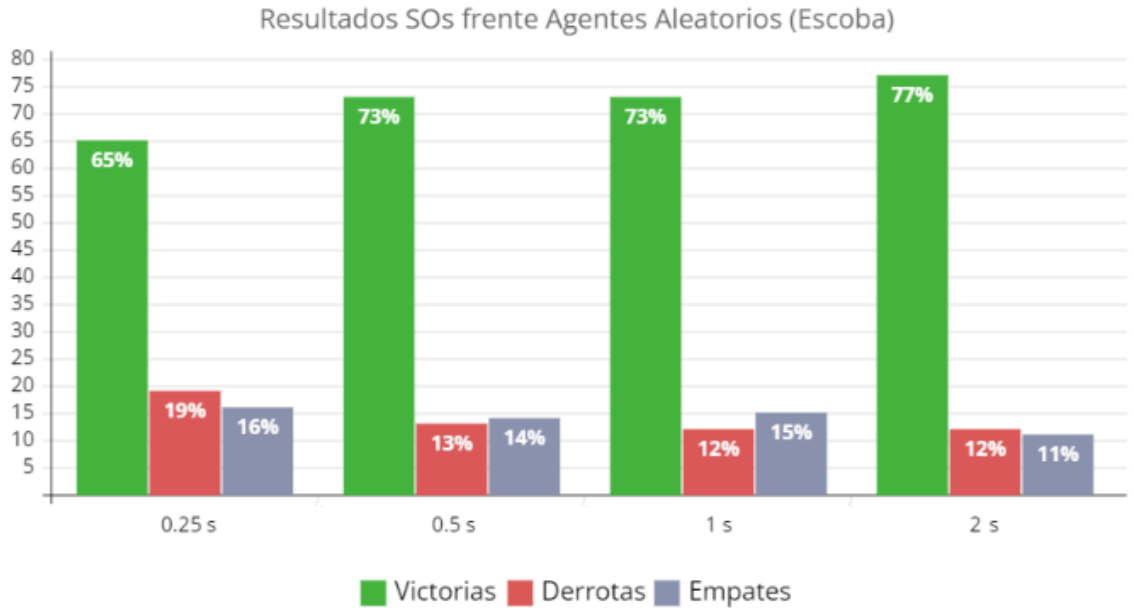

Figura 6.4: Resultados *SO-ISMCTS* vs Agente Aleatorio (*Escoba*)

Para la segunda prueba, también se han seleccionado cuatro tiempos de computación (0.25, 0.5, 1, 2 segundos) y se han realizado 100 partidas para cada uno de ellos. En cada una de las partidas, han participado dos agentes que han utilizado el algoritmo *MO-ISMCTS* y otros dos que han realizado movimientos aleatorios. Se han obtenido los siguientes resultados:

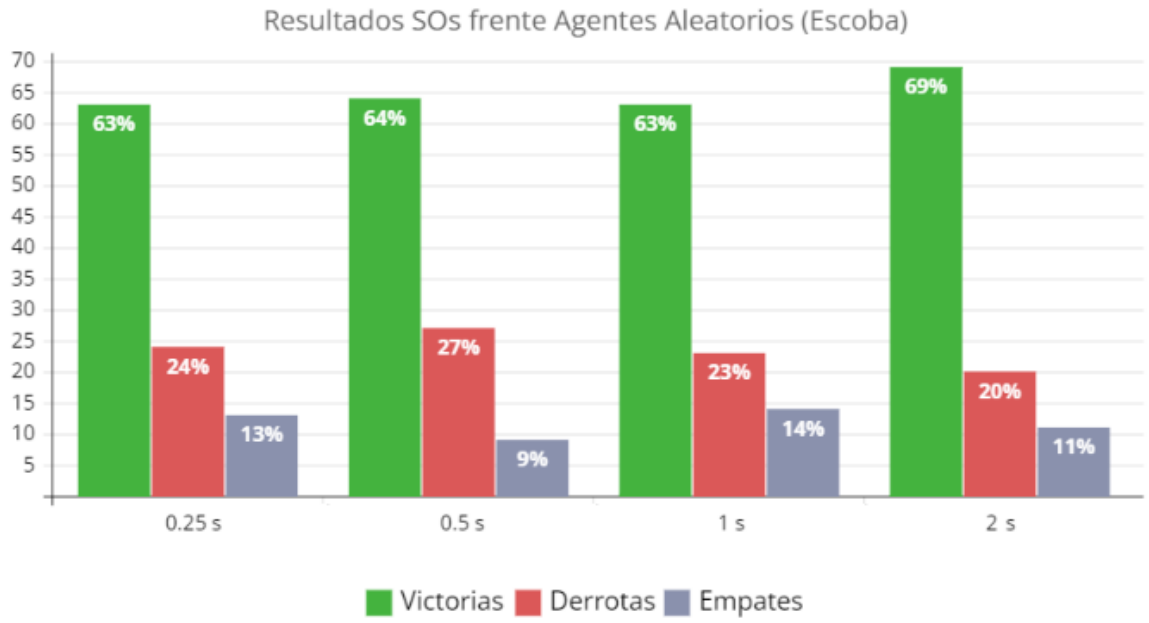

Figura 6.5: Resultados 2 *SO-ISMCTS* vs 2 Agentes Aleatorios (*Escoba*)

Los resultados anteriores muestran la clara superioridad del algoritmo *SO-ISMCTS* frente a los agentes aleatorios en el juego de la *escoba*. A pesar de que este juego tiene un gran componente de azar, por el reparto de las cartas, el algoritmo *SO-ISMCTS* logra ganar alrededor del 73 % de las partidas en la primera prueba, alcanzando este porcentaje de victorias desde los 0.5 segundos de tiempo de computación.

Sin embargo, se puede observar que el rendimiento del algoritmo *SO-ISMCTS* se ve ligeramente afectado en la segunda prueba, donde la tasa de victorias disminuye hasta alrededor del 65 % de las partidas. Esto podría deberse a que en la segunda prueba, en cada partida, se enfrentan dos jugadores que utilizan el algoritmo *SO-ISMCTS* compitiendo entre sí y también contra los agentes aleatorios. Esto podría beneficiar en cierto modo a los agentes aleatorios, haciendo que puedan lograr un mayor número de victorias. Aun así, la proporción de victorias sigue siendo bastante buena y cercana a la de la primera prueba.

6.2.2. Blackjack

El juego del *blackjack* se diferencia de otros juegos de casino en el sentido de que los jugadores no compiten directamente entre sí, sino que juegan contra el crupier. Además, el crupier no tiene libertad en cuanto a las decisiones que toma en el juego, ya que estas están determinadas por un conjunto específico de reglas las cuales ya han sido explicadas en la sección donde se introduce el *blackjack* 5.2.2.

Por ello, el *blackjack* se puede considerar como un juego en solitario, ya que un jugador normalmente realiza movimientos en base a sus cartas independientemente de lo que hagan los demás jugadores, es decir, cada jugador compite contra el crupier de forma individual. Por esta razón, se ha diseñado una prueba para poder analizar el rendimiento del *SO-ISMCTS* en un juego solitario, en el que el jugador no compite contra otros jugadores, sino solo contra el crupier.

La prueba consiste en enfrentar al algoritmo *SO-ISMCTS* contra el crupier en varias partidas de *blackjack*. El jugador, representado por el algoritmo, comenzará cada partida con 100 monedas y realizará una apuesta inicial de 2 monedas en cada ronda. El resultado se obtendrá como la diferencia entre el número de monedas inicial y final del jugador al terminar la partida. Además, como se explicó en la sección de introducción al juego del *blackjack* 5.2.2, cada casino utiliza un número específico de mazos, en este caso se utilizarán 4 mazos.

Es importante destacar que, en esta prueba, se considerará que la partida acaba cuando se han jugado el 75 % de las cartas de los 4 mazos utilizados, que es el punto en el que se suele parar para barajar los mazos en la mayoría de los casinos. Esto se debe a que el algoritmo necesita tener un final de partida claro para poder realizar las simulaciones y obtener una recompensa en el final de esta. En un juego real de *blackjack*, un jugador puede abandonar la mesa en cualquier momento, pero en esta prueba se debe realizar esta suposición para poder hacer uso del algoritmo *SO-ISMCTS*.

Además, cabe mencionar que se realizarán varias partidas con distintos tiempos de computación del algoritmo cuyo objetivo es poder analizar la evolución de las ganancias o pérdidas medias en función del tiempo de computación utilizado. Con esta prueba, se

espera obtener una evaluación del rendimiento del algoritmo *SO-ISMCTS* en un juego solitario, en el que el jugador no compite contra otros jugadores, sino solo contra el crupier.

Se han elegido cuatro tiempos de computación para el algoritmo *SO-ISMCTS* (0.25, 0.5, 1, 2 segundos), y se han llevado a cabo 100 partidas para cada uno de ellos, con el objetivo de obtener las medias de beneficios. Los resultados obtenidos obtenidos son los siguientes:

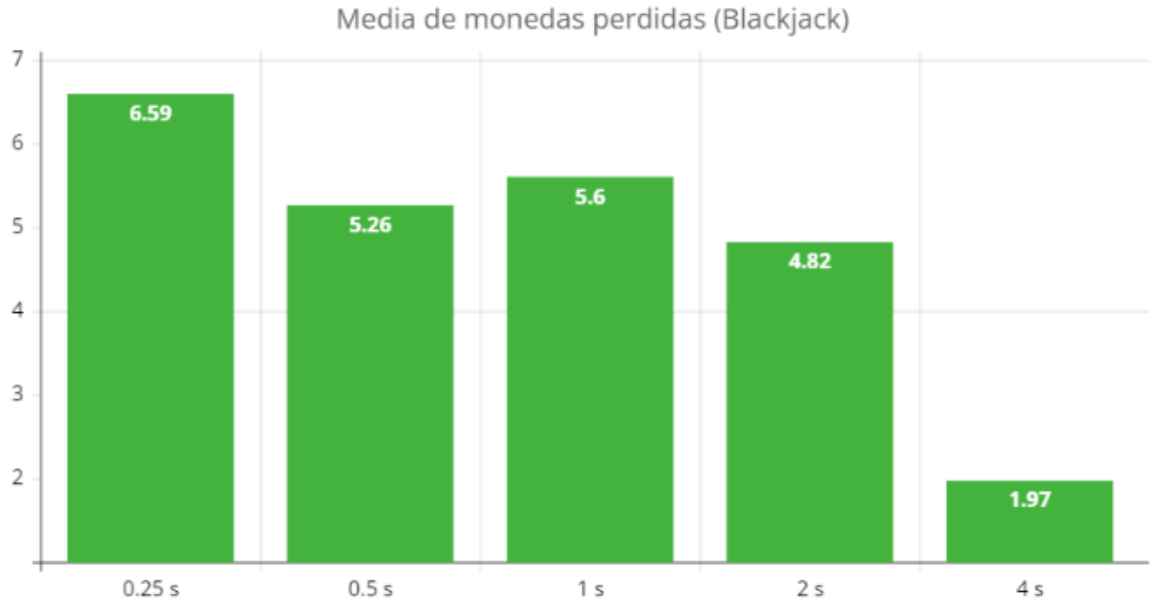

Figura 6.6: Media de monedas perdidas por *SO-ISMCTS* en el *Blackjack*

Por último, con el fin de hacer una comparativa con los resultados anteriores, se han simulado 100 partidas en las que se ha enfrentado a un agente aleatorio contra el crupier, obteniéndose una perdida de 28.58 monedas de media por partida. Este dato, junto con la gráfica de resultados, muestra el buen rendimiento del algoritmo *SO-ISMCTS*.

Sin embargo, a pesar de que el rendimiento del algoritmo mejora considerablemente a medida que se aumenta el tiempo de cómputo, en promedio no se obtiene ningún beneficio, sino que se pierden monedas. Esto podría deberse a la ventaja del crupier sobre los jugadores, ya que se estima que en el *blackjack*, el crupier tiene una ventaja de entre el 1 % y el 2 % sobre los jugadores, que es precisamente el rango alcanzado por el algoritmo *SO-ISMCTS* con 4 segundos de tiempo de cómputo.

6.2.3. Stratego

Para la evaluación del algoritmo *SO-ISMCTS* en el juego *Stratego*, se han realizado dos pruebas. Estas pruebas están diseñadas en base al uso de las configuraciones

iniciales obtenidas por la ejecución del algoritmo genético. Sin embargo, antes de introducir y mostrar los resultados de las pruebas, se presentará el resultado obtenido de la ejecución del algoritmo genético.

El algoritmo genético se ejecutó con una configuración de 1000 generaciones, una población de 100 individuos, una probabilidad de cruce del 60 % y una probabilidad de mutación del 20 %. De la población resultante de la última iteración del algoritmo genético se seleccionaron las 10 mejores configuraciones iniciales, según la función de evaluación, y se guardaron para hacer uso de estas en las pruebas de evaluación del rendimiento del algoritmo *SO-ISMCTS*.

Además, para analizar la distribución de las configuraciones iniciales, se ha creado un mapa de calor que representa la sección del tablero en la que un jugador coloca sus fichas, en este caso una sección del tablero de 3x10. En este mapa, se utiliza la escala de color rojo para mostrar las zonas en las aparece con frecuencia la bandera, oscureciendo el tono a medida que aumenta el número de apariciones. Asimismo, las zonas con mayor frecuencia de aparición de las bombas se representa mediante una escala de colores azules. A continuación, se muestra la imagen del mapa de calor mencionado:

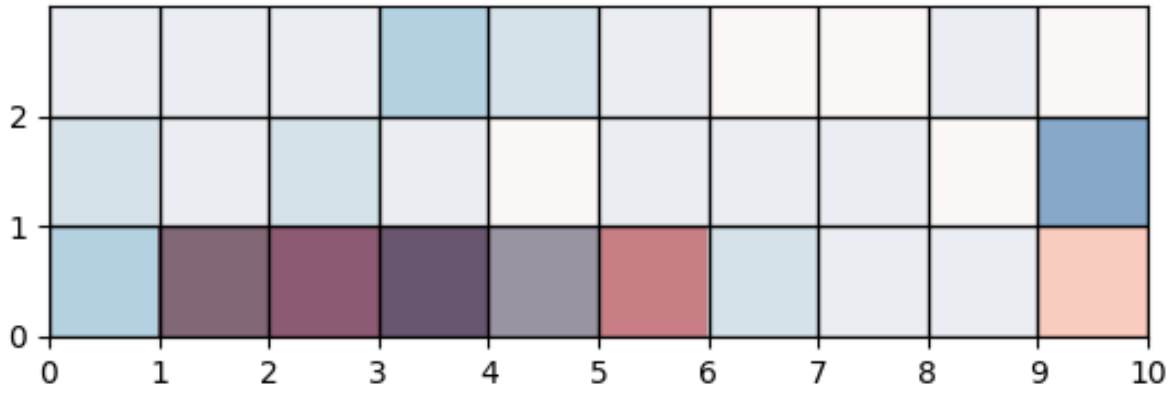

Figura 6.7: Mapa de Calor Configuraciones Iniciales (Algoritmo Genético)

Gracias al mapa de calor, se puede apreciar como los resultados del algoritmo genético han dado lugar a configuraciones iniciales en las que se coloca la bandera en la última fila siendo protegida por las bombas las cuales normalmente son colocadas en los alrededores.

La primera de las pruebas consiste en enfrentar al algoritmo *SO-ISMCTS*, haciendo uso de las configuraciones iniciales obtenidas con el algoritmo genético, contra un agente que realiza movimientos de manera aleatoria y que hace uso de configuraciones iniciales aleatorias. Se limitará el algoritmo *SO-ISMCTS* con diferentes tiempos de computación para evaluar la evolución de su rendimiento con distintos límites de tiempo y con una ventaja aparente sobre el oponente.

Por último, la segunda prueba se ha diseñado para evaluar la evolución del rendimiento del algoritmo *SO-ISMCTS* en partidas en las que se parte con una menor ventaja frente al oponente. Para ello, se enfrentará al algoritmo consigo mismo, con

la diferencia de que uno de ellos utilizará configuraciones iniciales aleatorios y el otro utilizará las configuraciones iniciales obtenidas con el algoritmo genético. Además, se utilizará el mismo tiempo de computación para ambos algoritmos, siendo seleccionados varios tiempos de computación.

Es importante destacar que se ha establecido un límite de 500 turnos en todos los enfrentamientos de ambas pruebas. Si ninguno de los jugadores logra la victoria antes de alcanzar este límite, la partida acaba con empate. En principio se estableció un límite de 250 turnos, sin embargo, los resultados reflejaron que más del 70 % de las partidas terminaban en empate. Por esta razón, se decidió aumentar el límite de turnos a 500 para reducir el número de empates en los resultados.

En la primera prueba se realizaron 100 enfrentamientos para cada uno de los tiempos de computación seleccionados (0.5, 1, 2 segundos). A continuación, se muestra una gráfica con los resultados obtenidos:

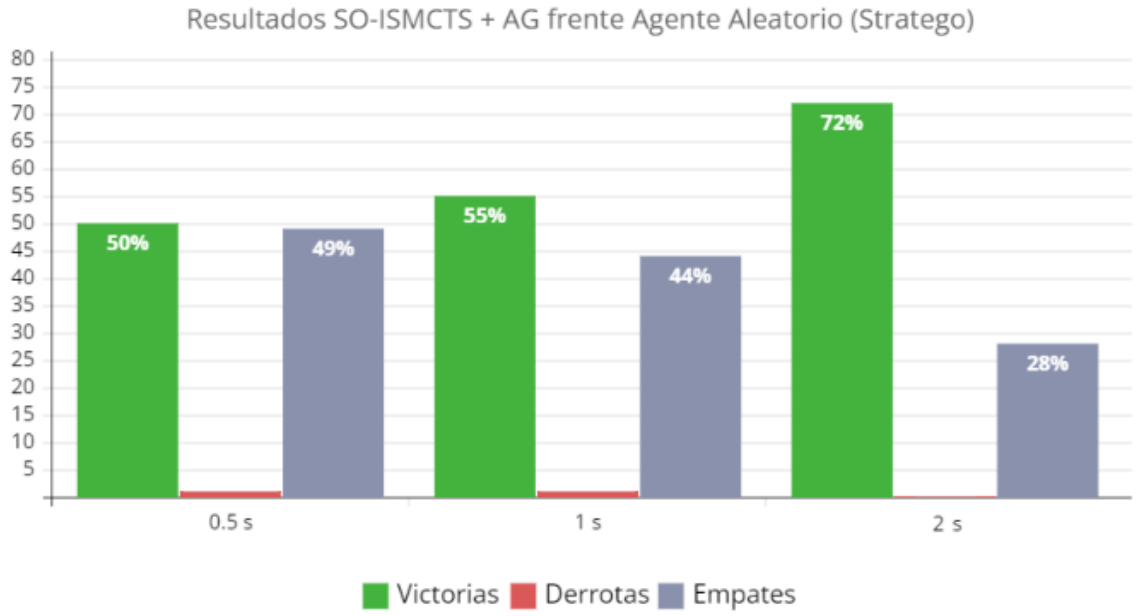

Figura 6.8: Resultados *SO-ISMCTS + AG* vs Agente Aleatorio (*Stratego*)

Igualmente, en la segunda prueba se realizaron 100 enfrentamientos para cada uno de los tiempos de computación elegidos (0.5, 1, 2 segundos). Los resultados obtenidos se muestran en la siguiente gráfica:

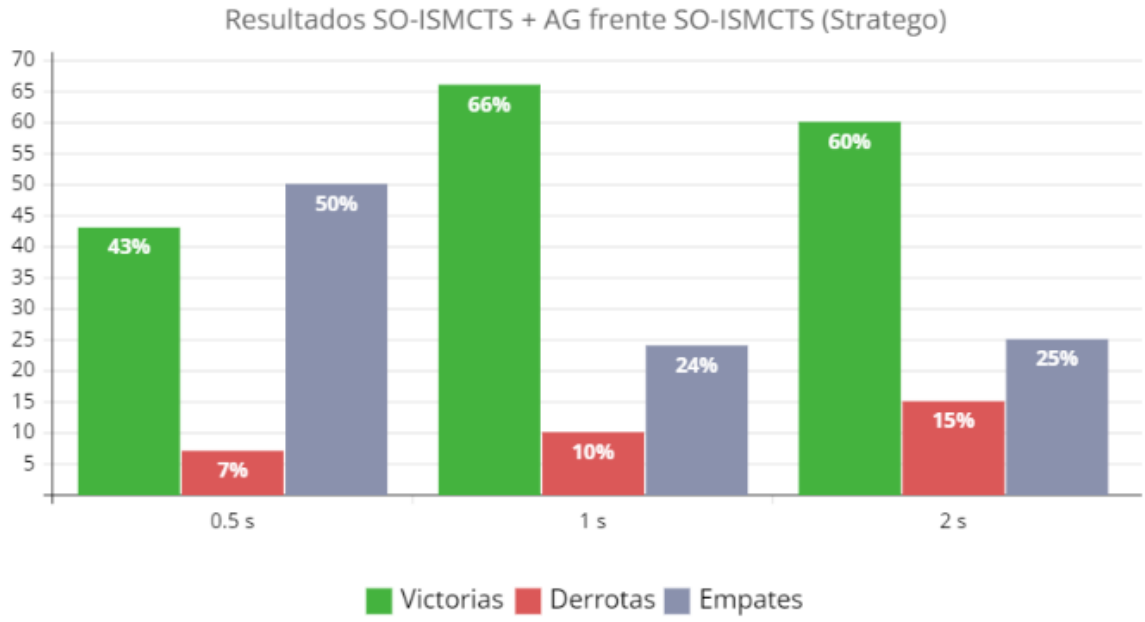

Figura 6.9: Resultados *SO-ISMCTS + AG* vs *SO-ISMCTS* (*Stratego*)

Los resultados de las pruebas muestran el buen rendimiento del algoritmo *SO-ISMCTS*. En la primera prueba, se puede apreciar su superioridad frente al agente aleatorio, logrando anular las victorias de este último y obteniendo un porcentaje de victorias superior al 70 % con solo 2 segundos de tiempo de computación.

Además, gracias a los resultados de la segunda prueba, se puede apreciar cómo el algoritmo *SO-ISMCTS* es muy superior cuando utiliza las configuraciones iniciales obtenidas por el algoritmo genético, logrando una tasa de victorias cercana al 60 %, lo cual es significativamente superior a las tasa de victorias obtenida por el algoritmo *SO-ISMCTS* utilizando configuraciones iniciales aleatorias, las cuales en ningún caso superaron el 15 %.

6.3. *MO-ISMCTS*

Por último, para evaluar el rendimiento del algoritmo *MO-ISMCTS*, se llevarán a cabo un par de pruebas en los juegos seleccionados como casos de estudio. A diferencia del algoritmo *SO-ISMCTS*, se usarán las mismas pruebas para el algoritmo *MO-ISMCTS* en todos los juegos.

La primera prueba constará de varios enfrentamientos entre el algoritmo *MO-ISMCTS* y un agente que elige movimientos aleatorios. En cada uno de los enfrentamientos, se utilizarán diferentes tiempos de computación para *MO-ISMCTS* con el fin de evaluar la evolución del rendimiento con distintas limitaciones de tiempo en una situación donde tiene una ventaja clara sobre el oponente.

La segunda prueba consistirá en una serie de enfrentamientos entre el algoritmo *MO-ISMCTS* y el algoritmo *SO-ISMCTS*. En cada uno de los enfrentamientos se utilizará el mismo tiempo de computación para ambos algoritmos. El objetivo de esta prueba es evaluar el rendimiento del *MO-ISMCTS* frente al *SO-ISMCTS* en juegos con movimientos parcialmente observables, ya que el *MO-ISMCTS* fue diseñado específicamente para mejorar el rendimiento de *SO-ISMCTS* en este tipo de juegos, como se explicó en el marco teórico en la sección 2.2.2.

Estas dos pruebas tienen como objetivo estudiar el rendimiento del algoritmo *MO-ISMCTS* en distintas situaciones del juego, partiendo en algunas con ventaja. Cabe destacar que, al igual que en las pruebas diseñadas para los demás algoritmos, se alternará el jugador inicial en los enfrentamientos para mantener la igualdad de condiciones en este aspecto.

6.3.1. Phantom (4,4,4)

Para la primera prueba, se han seleccionado 4 tiempos de computación (0.25, 0.5, 1, 1.5 segundos) y para cada uno de ellos se han realizado 100 enfrentamientos entre el algoritmo *MO-ISMCTS* y un agente aleatorio. Los resultados obtenidos se muestran a continuación:

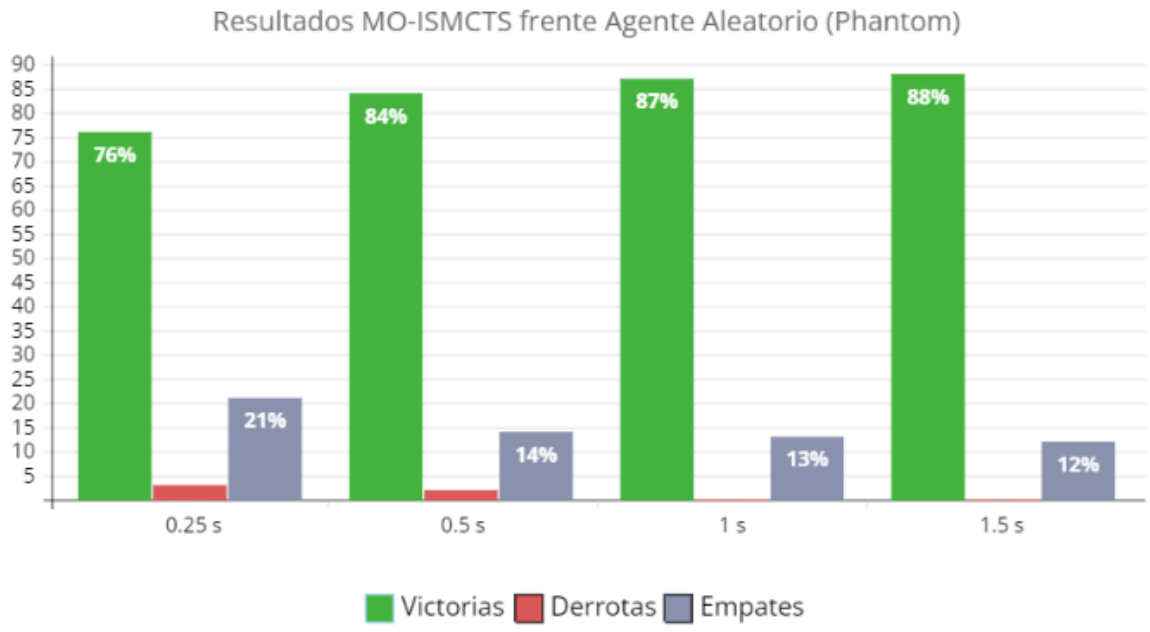

Figura 6.10: Resultados *MO-ISMCTS* vs Agente Aleatorio (*Phantom*)

Para la segunda prueba, también se han seleccionado cuatro tiempos de computación (0.5, 1, 2, 3 segundos) y para cada uno de ellos se han realizado 100 enfrentamientos entre el algoritmo *MO-ISMCTS* y el algoritmo *SO-ISMCTS*. A continuación, se comentan los resultados obtenidos:

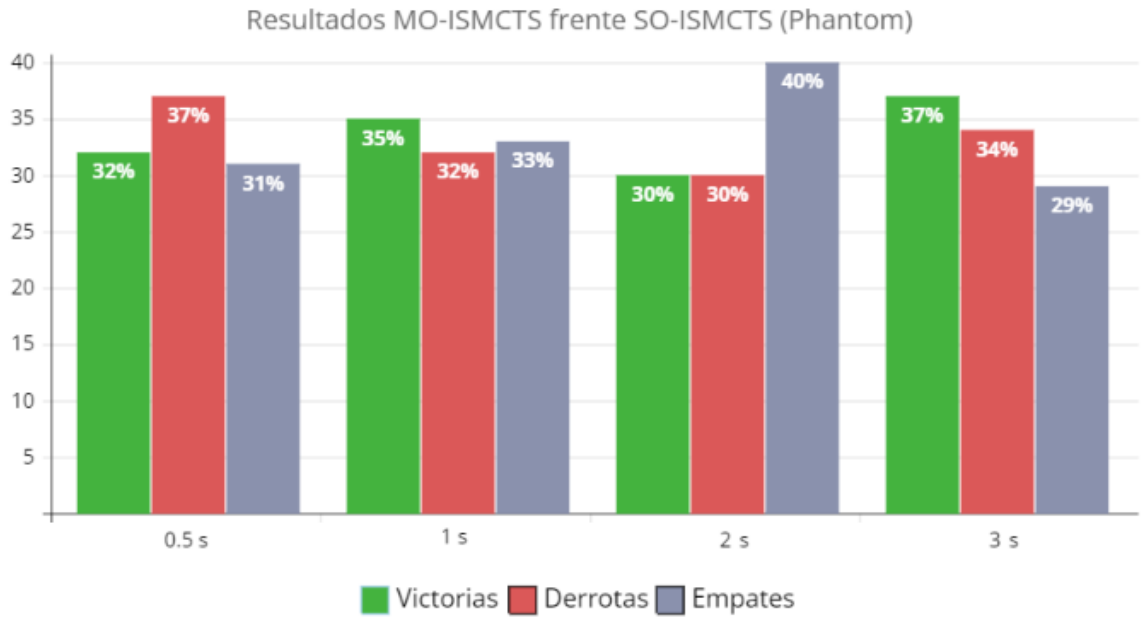

Figura 6.11: Resultados *MO-ISMCTS* vs *SO-ISMCTS* (*Phantom*)

Como cabría esperar, el algoritmo MO-ISMCTS es claramente superior al agente aleatorio. En la primera prueba, MO-ISMCTS logra alcanzar una gran proporción de victorias desde el primer instante de computación, y con tan solo 1 segundo de computación, consigue anular por completo cualquier victoria del agente aleatorio.

Por otro lado, en los resultados de la segunda prueba se puede apreciar cómo, en promedio, el rendimiento del algoritmo *MO-ISMCTS* es ligeramente superior al rendimiento del algoritmo *SO-ISMCTS*, aunque la tasa de victorias de ambos está muy igualada. Esto podría deberse a que, aunque el algoritmo *MO-ISMCTS* está diseñado para mejorar el rendimiento del *SO-ISMCTS* en este tipo de juegos, el algoritmo *SO-ISMCTS* es más simple, ya que solo construye un árbol de búsqueda, lo que le permite realizar entre un 15 % y un 30 % más iteraciones que el algoritmo *MO-ISMCTS*, para el mismo tiempo de computación.

6.3.2. Holjjak

Este juego es un caso de estudio especial, ya que hay momentos durante la partida en los que el juego se convierte en un juego de información perfecta para el jugador que debe realizar el siguiente movimiento. Esto ocurre cuando el jugador debe elegir el número de canicas a apostar, ya que en este momento no hay ninguna información oculta para el jugador, el cual sabe el número de canicas que tiene tanto él como su rival. Todo esto se comenta con mayor detalle en la sección de casos de estudio dedicada a este juego (5.3.2).

Es por ello que se decidió utilizar una combinación de algoritmos para elegir los distintos movimientos en este juego. En concreto, se utilizó el algoritmo *MCTS* para

la elección del número de canicas a apostar y el algoritmo *MO-ISMCTS* o el algoritmo *SO-ISMCTS* para el resto de movimientos. En las pruebas realizadas, se mantuvo el uso del algoritmo *MCTS* con el mismo tiempo de computación que se dio a los otros algoritmos, ya sea el *SO-ISMCTS* o el *MO-ISMCTS*. Sin embargo, para la prueba en la que se utilizó un agente que realizaba movimientos aleatorios, se decidió eliminar el uso del algoritmo *MCTS*, para mantener la aleatoriedad completa del agente.

En la primera prueba, se llevaron a cabo 100 enfrentamientos entre el algoritmo *MO-ISMCTS* y un agente aleatorio para cada uno de los cuatro tiempos de computación elegidos (0.25, 0.5, 1, 1.5 segundos). A continuación se presentan los resultados obtenidos:

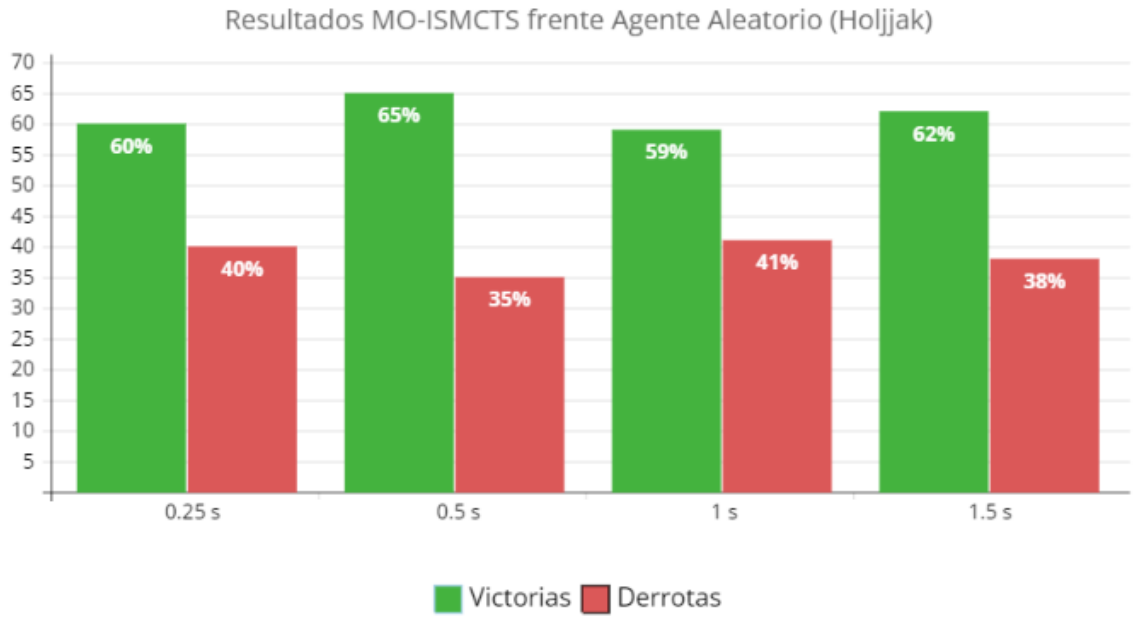

Figura 6.12: Resultados *MO-ISMCTS* vs Agente Aleatorio (*Holjjak*)

En la segunda prueba, se llevaron a cabo 100 enfrentamientos entre el algoritmo *MO-ISMCTS* y el algoritmo *SO-ISMCTS* para cada uno de los cuatro tiempos de computación seleccionados (0.25, 0.5, 1, 1.5 segundos). Se han obtenido los siguientes resultados:

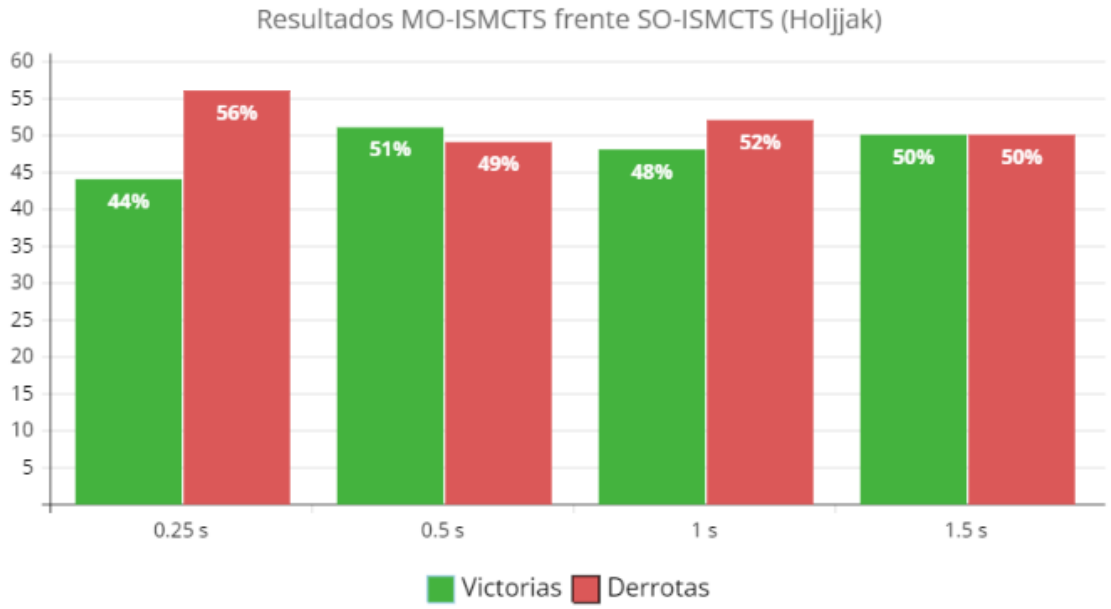

Figura 6.13: Resultados *MO-ISMCTS* vs *SO-ISMCTS* (*Holjjak*)

Debido a que el juego *holjjak* es bastante simple y ofrece pocas opciones, se podría esperar que incluso un agente aleatorio se acercase considerablemente a la tasa de victorias del algoritmo *MO-ISMCTS*. Sin embargo, los resultados de la primera prueba muestran que el algoritmo *MO-ISMCTS* es capaz de mantener una tase de victorias en torno al 62 % frente al agente aleatorio.

Por último, en la segunda prueba, se puede apreciar una igualdad en el rendimiento de los algoritmos *MO-ISMCTS* y *SO-ISMCTS*. Esta igualdad se hace absoluta dados 1.5 segundos de computación, en los que ambos obtienen una tasa de victorias del 50 %.

7. Conclusiones y Trabajo Futuro

Este Trabajo de Fin de Grado ha consistido en el desarrollo de una biblioteca para *Python* incluyendo la implementación de algoritmos de búsqueda para resolver juegos con información perfecta e imperfecta. Además, se han desarrollado varios juegos de ambos tipos para demostrar el correcto funcionamiento de la biblioteca y se han realizado varias pruebas experimentales para obtener un análisis estadístico del rendimiento de los distintos algoritmos en diferentes situaciones.

Por último, se ha creado un detallado manual de uso, para que la biblioteca sea fácilmente accesible por los usuarios. Todo esto ha sido publicado en el repositorio oficial de software de *Python*, conocido como *PyPi*, para que esté disponible y abierto a cualquier desarrollador interesado.

Además, se ha creado un repositorio público en *GitHub* que incluye la biblioteca, su manual y todos los juegos desarrollados. Estos juegos también pueden ser utilizados como ejemplos para que los desarrolladores puedan ver cómo integrar la biblioteca en sus propios juegos.

Durante el análisis del estado del arte, se pudo comprobar la falta de una biblioteca de este tipo, lo que motivó a continuar con el análisis y estudio de cada uno de los algoritmos que serían implementados. Luego de este análisis, se seleccionaron los algoritmos *minimax* y *MCTS* para juegos de información perfecta, y *SO-ISMCTS* y *MO-ISMCTS* para juegos de información imperfecta. Además, se definieron los juegos de mesa que serían desarrollados como casos de estudio para cada uno de estos algoritmos y se diseñaron las pruebas experimentales que serían realizadas para evaluar el rendimiento de cada uno de los algoritmos en diferentes situaciones.

Un paso previo a la realización de pruebas fue hacer un estudio, para cada juego, entre el nivel de profundidad del algoritmo *minimax* y su tiempo de computación. Esto se realizó para que las partidas entre *minimax* y *MCTS* estuvieran en igualdad de condiciones, ya que el algoritmo *MCTS* recibe el tiempo de cómputo como parámetro.

Gracias a estas pruebas se ha podido comprobar, en general, como la superioridad inicial del algoritmo *minimax* frente al algoritmo *MCTS* se va reduciendo progresivamente, llegando incluso a igualarse, conforme se aumenta el tiempo de computación de ambos algoritmos. Esto puede deberse a que, dado el suficiente tiempo de cómputo, *MCTS* converge al árbol de búsqueda completo generado por *minimax*.

Por otro lado, las pruebas realizadas muestran el excelente rendimiento del algoritmo *SO-ISMCTS* en juegos de información imperfecta, logrando una alta tasa de victorias en enfrentamientos contra agentes aleatorios. Además, en el caso de estudio especial del juego *Stratego*, se ha desarrollado adicionalmente un algoritmo genético para decidir la configuración inicial de fichas. Se ha encontrado que la utilización de dicho algoritmo genético permite mejorar aún más el rendimiento del algoritmo *SO-ISMCTS*. Esto se ha comprobado al enfrentar el *SO-ISMCTS* utilizando las configuraciones iniciales generadas por el algoritmo genético con el *SO-ISMCTS* utilizando configuraciones iniciales aleatorias, donde se observa una clara superioridad del primero en tales en-

frentamientos directos. Esto demuestra la importancia de las configuraciones iniciales en el *Stratego* y cómo la utilización de un algoritmo genético, en ciertos apartados de juegos complejos, puede ser una herramienta útil para mejorar el rendimiento de los algoritmos de búsqueda.

Finalmente, las pruebas realizadas sobre el algoritmo *MO-ISMCTS* han demostrado su buen rendimiento en juegos de información imperfecta con movimientos parcialmente observables, obteniendo porcentajes de victorias similares e incluso superiores en algunas ocasiones al enfrentarse contra el algoritmo *SO-ISMCTS*. Sin embargo, es importante tener en cuenta que *SO-ISMCTS*, debido a su simplicidad y uso de un único árbol de búsqueda, es capaz de realizar alrededor de un 25 % más de iteraciones que *MO-ISMCTS* en el mismo tiempo de cómputo, lo que le da una ventaja significativa en este aspecto.

Como trabajo futuro, se propone ampliar la biblioteca con más algoritmos para todo tipo de juegos, de manera que los usuarios tengan una amplia variedad de opciones. También se pretende agregar más información sobre los resultados de la experimentación en el manual de uso, para que los usuarios puedan identificar los algoritmos que mejor se ajusten a sus necesidades específicas. Además, sería conveniente aumentar la visibilidad de la biblioteca *pyplAI* dentro de la comunidad de desarrolladores de código abierto de *Python*, especialmente los interesados en juegos de mesa, para que puedan hacer uso de ella y contribuir en su repositorio de *GitHub*.

Por último, como objetivo más ambicioso, se podría adaptar la biblioteca para la resolución de problemas de la Teoría de Juegos, de manera que tenga un enfoque más amplio y no se centre únicamente en la resolución de juegos de mesa, lo que podría beneficiar a una comunidad mucho más amplia.

8. Bibliografía

[1] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. doi: 10.1109/TCIAIG.2012.2186810. URL https://ieeexplore.ieee.org/document/6145622.

[2] Peter I. Cowling, Edward J. Powley, and Daniel Whitehouse. Information set monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):120–143, 2012. doi: 10.1109/TCIAIG.2012.2200894. URL https://ieeexplore.ieee.org/document/6203567.

[3] Kamil Czarnogórski. Pypi - mctspy, Jul 2019. URL https://pypi.org/project/mctspy/.

[4] Levente Kocsis, Csaba Szepesvari, and Jan Willemson. Improved monte-carlo search. 2006. URL https://www.semanticscholar.org/paper/Improved-Monte-Carlo-Search-Kocsis-Szepesvari/b2c20a877a891ea97179658c06a6d552b50cba6e.

[5] Eytan Lifshitz and David Tsurel. Ai approaches to ultimate tic-tac-toe. page 5, 2013. URL https://www.cs.huji.ac.il/w~ai/projects/2013/UlitmateTic-Tac-Toe/files/report.pdf.

[6] Jeffrey Richard Long, Nathan R Sturtevant, Michael Buro, and Timothy Furtak. Understanding the success of perfect information monte carlo sampling in game tree search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2010. URL https://www.semanticscholar.org/paper/Understanding-the-Success-of-Perfect-Information-in-Long-Sturtevant/011e2c79575721764c127e210c9d8105a6305e70.

[7] Carol A. Luckhardt and Keki B. Irani. An algorithmic solution of n-person games. page 158–162, 1986. URL https://dl.acm.org/doi/10.5555/2887770.2887795.

[8] Patricio Mendoza. Alpha-beta pruning algorithm: The intelligence behind strategy games. page 9, 2022. URL https://www.researchgate.net/publication/360872512.

[9] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[10] Claude E. Shannon. Programming a computer playing chess. *Philosophical Magazine*, Ser.7, 41(314), 1950. URL https://vision.unipv.it/IA1/ProgrammingaComputerforPlayingChess.pdf.

[11] Paul Sinclair. Pypi - mcts, Apr 2019. URL https://pypi.org/project/mcts/.

[12] Ángel F. Tenorio Villalón and Ana M. Martín Caraballo. Un paseo por la historia de la teoría de juegos. *Boletín de Matemáticas*, 22(1):77–95, ene. 2015. URL https://revistas.unal.edu.co/index.php/bolma/article/view/51847.

[13] J. von Neumann and O. Morgenstern. *Theory of games and economic behavior*. Princeton University Press, 1947.

A. Manual de Uso de *pyplAI*

Introducción

Esta biblioteca para *Python* es el resultado final de un Trabajo de Fin de Grado centrado en la implementación de algoritmos de *Monte Carlo Tree Search (MCTS)* y *minimax* para diferentes tipos de juegos de mesa, entre ellos, según clasifica la teoría de juegos, los juegos de información perfecta, como lo son *el ajedrez* o *las damas*, en los que en todo momento cualquier jugador puede ver toda la información del juego, como los posibles movimientos del rival, y juegos de información imperfecta, como lo son la mayoría de juegos de cartas como por ejemplo el *Uno* o la *brisca*, en el que los jugadores solo conocen información propia, como sus cartas, o información general del juego, como el estado de la mesa, pero desconocen la información del rival, como las cartas de los demás jugadores.

Juegos de Información Perfecta

Para este tipo de juegos se han implementado los algoritmos de *MCTS con UCT* [1] y *minimax* con la técnica de *poda de alfa-beta* [8]. Además, como ejemplos para ver el correcto uso de la biblioteca, se han creado diferentes juegos de mesa, entre ellos, el *Tic-Tac-Toe* (3 en raya), el *Ultimate Tic-Tac-Toe* [5] y las *damas*.

Juegos de Información Imperfecta

Para este segundo tipo de juegos se han implementado dos algoritmos, el *Single Observer Information Set Monte Carlo Tree Search (SO-ISMCTS)* [2] y el *Multiple Observer Information Set Monte Carlo Tree Search (MO-ISMCTS)* [2]. El primero de estos algoritmos genera un único árbol desde el punto de vista del jugador al que le toca jugar. Por otro lado, el segundo algoritmo crea un árbol para cada uno de los jugadores y agrupa las acciones que son indiferenciables desde el punto de vista de este jugador, es decir, si un jugador puede realizar varios movimientos que no aportarían información a los rivales, agrupa estos movimientos como si fuesen uno solo, creando un único nodo en el árbol de los jugadores rivales.

Un ejemplo serían los juegos en los que un jugador intercambia una de las cartas de su mano con cualquier carta de la baraja, cuando el jugador efectúa este intercambio los demás jugadores no reciben ningún tipo de información sobre el intercambio de cartas, por lo que su conocimiento sobre el estado del juego no cambia.

Para estos algoritmos también se han desarrollado varios juegos como ejemplos de uso de la biblioteca, aunque ambos algoritmos se pueden usar en todos los juegos de información imperfecta, se ha diferenciado entre juegos para el *SO-ISMCTS* y el *MO-ISMCTS*. Para el primer algoritmo se ha creado el juego de la *escoba*, el *Stratego* y el *blackJack*, y para el segundo se ha desarrollado el *holjjak* (juego de adivinar las canicas

del rival) y el *phantom* (variante del 4 en raya en el que no se ven los movimientos del rival).

Manual de Uso

El primer paso necesario para poder usar esta biblioteca es descargarla. Se puede descargar la biblioteca desde la consola usando el comando *pip install pyplAI*. Tras esto, ya se puede importar en el archivo de *Python* del juego al que queramos implementarla.

```
import pyplAI
```

Clase sobre el Estado del Juego

Para poder usar esta biblioteca es imprescindible que haya una clase que guarde la información necesaria sobre el estado del juego. Aunque se puede dar a esta clase el nombre que se desee, en este manual se utilizará una clase llamada *Juego* a modo de ejemplo.

```
class Juego:
    ⋮
```

Atributo Obligatorio *jugadorActual*

Debido a que los algoritmos necesitan saber cuál es el jugador al cual le toca jugar en cada uno de los turnos, es obligatorio añadir un atributo *jugadorActual* en la clase del juego. Además, es muy importante que este atributo tenga este mismo nombre.

```
class Juego:
    def __init__(self):
        self.jugadorActual=1
                ⋮
```

Este atributo debe contener un valor numérico que representa el índice del jugador al que le toca jugar (empezando por 1). También, debe ir actualizándose conforme se avanza en el juego, por ejemplo, en el *3 en raya*, este atributo se debe iniciar con el valor 1, pero cuando el primer jugador realice su movimiento, se actualiza al valor 2, indicando que el segundo jugador es el que debe realizar su jugada.

Además del atributo para identificar el jugador actual del estado del juego, esta clase deberá implementar ciertos métodos para poder interactuar y obtener información sobre el estado del juego. Algunos métodos son comunes para todos los algoritmos, mientras que otros algoritmos requieren de métodos específicos. Cabe destacar que estos métodos pueden ser nombrados de forma distinta a como se mostrará a continuación, pero es totalmente necesario que reciban los parámetros de entrada y devuelvan la salida tal y como se especifica en este manual.

Métodos Generales

Los siguientes 4 métodos serán obligatorios para todos los algoritmos que se implementan en esta biblioteca:

- **obtiene_movimientos(self)**: Devolverá una lista con todos los movimientos posibles del jugador actual del estado del juego, es recomendable devolver estos movimientos en un orden aleatorio.

```
class Juego:
        ⋮
    def obtiene_movimientos(self):
            ⋮
        return movimientos
```

- **aplica_movimiento(self, movimiento)**: Aplicará el movimiento dado al estado del juego y devolverá el estado resultante (*self*).

```
class Juego:
        ⋮
    def aplica_movimiento(self, movimiento):
            ⋮
        return self
```

- **es_estado_final(self)**: Comprueba el estado actual del juego, devolviendo *True* o *False* dependiendo de si es un estado final o no.

```
class Juego:
        ⋮
    def es_estado_final(self):
            ⋮
        return True
            o
        return False
```

- **gana_jugador(self, jugador)**: Comprueba si el jugador dado gana el juego en el estado actual de este, devolviendo *True* en caso de que sea ganador, o *False* en caso contrario. La variable *jugador* debe ser un número entero positivo, y se debe situar en el rango de 1, para el primer jugador del juego, hasta n, siendo n el número total de jugadores.

```
class Juego:
        ⋮
    def gana_jugador(self, jugador):
            ⋮
        return True
            o
        return False
```

Métodos SO/MO-ISMCTS

Además de los 4 métodos anteriores, los algoritmos de *SO-ISMCTS* y *MO-ISMCTS* necesitan un método adicional para su uso, el cuál se explica a continuación:

- **determinacion(self)**: Devuelve una determinación aleatoria del estado del juego, desde el punto de vista del jugador actual.

```
class Juego:
        ⋮
    def determinacion(self):
            ⋮
        return determinacion
```

Por ejemplo, para un juego de cartas en el que cada jugador tiene una mano, hay una mesa con cartas visibles, una pila de descartes y un mazo de robo inicial, el jugador actual no tiene conocimiento sobre las cartas del rival ni las cartas del mazo de robo, pero si sabe cuáles son sus cartas, las de la mesa y las que han sido mandadas a la pila de descartes, por lo que tiene la información sobre cuáles son las cartas restantes, pero aún no sabe donde se encuentran.

Con esta información, este método debe dar un estado del juego en el que aleatoriamente se distribuyan las cartas que el jugador no tiene localizadas entre las zonas en las que no sabe que cartas hay, en este caso, las zonas serían las manos de los rivales y el mazo de robo.

Métodos MO-ISMCTS

Además, para el funcionamiento del algoritmo *MO-ISMCTS* se necesitará otro método adicional:

- **es_movimiento_visible(movimiento)**: Este método, dado un movimiento, devuelve *True* si es un movimiento visible para los rivales, o *False* en caso contrario.

```
class Juego:
        ⋮
    def es_movimiento_visible(movimiento):
            ⋮
        return True
           o
        return False
```

Por ejemplo, un movimiento visible para los rivales sería jugar una carta de tu mano sobre la mesa, por lo que, haría saber a los rivales el lugar exacto de esa carta, cosa que antes no sabían. Sin embargo, un movimiento no visible sería intercambiar una carta de tu mano con el mazo de robo sin revelar ninguna de las dos. Esta acción no mostraría nueva información a ningún rival, ya que no se muestran las cartas y los rivales seguirán sin saber que cartas hay en tu mano ni que cartas hay en el mazo.

Métodos Minimax

En el caso del algoritmo de *minimax*, además de los 4 métodos generales, se necesita un método que devuelva una heurística sobre el estado del juego, es decir, que evalúe su estado para saber como de bueno es desde el punto de vista de un jugador dado:

- **heuristica(self, jugador)**: Este método debe devolver un número entero que refleje una evaluación sobre como de bueno es el estado del juego para un jugador dado. Esta evaluación debe ser mayor cuanto mejor sea el estado del juego para este jugador.

```
class Juego:
        ⋮
    def heuristica(self, jugador):
            ⋮
        return evaluacion
```

Por ejemplo, para el juego de las *damas* una posible heurística sería contar el número de fichas del jugador y restarle el número de fichas del rival. Hay multitud de posibles heurísticas para cada uno de los juegos, y se debe tener un conocimiento sobre el juego para poder crear una buena heurística, ya que la heurística tendrá un gran peso a la hora de decidir cuál es el mejor movimiento.

Al igual que en el método *gana_jugador*, el argumento *jugador* debe ser un número entero positivo, desde el 1, para el primer jugador, hasta n, para el último jugador, siendo n el número total de jugadores.

Constructor Algoritmo

Una vez tengamos todos estos métodos solamente debemos llamar a la biblioteca y al constructor del algoritmo que se desee utilizar. A esta llamada le pasaremos como argumentos los métodos necesarios para el uso de este algoritmo junto con algunas otras variables que detallaremos a continuación:

- **numeroJugadores**: Este argumento es obligatorio para todos los algoritmos, representará el número de jugadores que contiene el juego, siendo este un número entero mayor que cero.
- **tiempoEjecución**: Este argumento es común para todos los algoritmos de Monte Carlo, ya que, estos algoritmos necesitan un tiempo límite de ejecución antes de devolver el mejor movimiento encontrado. Este argumento debe ser un número real mayor que cero y representa el tiempo de ejecución en segundos.
- **profundidadBusqueda**: Este argumento solo es necesario en la llamada al constructor del algoritmo de *minimax*, y sirve para limitar la profundidad en el árbol de búsqueda. Este argumento debe ser un número entero mayor que cero.

Sabiendo todo esto ya podemos ver como se deben hacer las llamadas a la biblioteca y los constructores para cada uno de los tipos de algoritmos. Es muy importante

que se siga el orden mostrado en los argumentos de entrada. Además, recordar que *Juego* es la clase de ejemplo que contiene la información sobre el estado del juego, como el atributo *jugadorActual*, y los métodos explicados anteriormente.

- **MCTS**:

```
mcts = pyplAI.MCTS(
       Juego.aplica_movimiento,
       Juego.obtiene_movimientos,
       Juego.es_estado_final,
       Juego.gana_jugador,
       numeroJugadores,
       tiempoEjecucion)
```

- **Minimax**:

```
minimax = pyplAI.Minimax(
          Juego.aplica_movimiento,
          Juego.obtiene_movimientos,
          Juego.es_estado_final,
          Juego.gana_jugador,
          Juego.heuristica,
          numeroJugadores,
          profundidadBusqueda)
```

- **SO-ISMCTS**:

```
so_ismcts = pyplAI.SOISMCTS(
            Juego.aplica_movimiento,
            Juego.obtiene_movimientos,
            Juego.es_estado_final,
            Juego.gana_jugador,
            Juego.determinacion,
            numeroJugadores,
            tiempoEjecucion)
```

- **MO-ISMCTS**:

```
mo_ismcts = pyplAI.MOISMCTS(
            Juego.aplica_movimiento,
            Juego.obtiene_movimientos,
            Juego.es_estado_final,
            Juego.gana_jugador,
            Juego.determinacion,
            Juego.es_movimiento_visible,
            numeroJugadores,
            tiempoEjecucion)
```

- **Modo *verbose***:

  Si queremos ver algunos detalles cuando ejecutemos el algoritmo, se debe añadir un *True* como último argumento en la llamada al constructor. Este modo mostrará información útil del algoritmo por consola, como por ejemplo, tiempo de ejecución, número de nodos creados y visitados y demás características propias de cada uno de los algoritmos. Un ejemplo de como quedaría la creación del algoritmo de *MCTS* con este argumento extra sería el siguiente:

```
mcts = pyplAI.MCTS(
       Juego.aplica_movimiento,
       Juego.obtiene_movimientos,
       Juego.es_estado_final,
       Juego.gana_jugador,
       numeroJugadores,
       tiempoEjecucion,
       True)
```

Ejecución Algoritmo

Una vez tengamos el objeto del algoritmo que se quiere utilizar solamente se debe llamar a su método *ejecuta* y pasarle como argumento el objeto que contiene el estado actual del juego, esto devolverá el mejor movimiento encontrado durante el tiempo de computación dado, para los algoritmos de *MCTS*, o la profundidad de búsqueda, en el caso del algoritmo *minimax*. A continuación, se mostrará el código de ejemplo en el que se usa el algoritmo de *MCTS* para obtener un movimiento y seguidamente aplicarlo al juego, obteniendo así un nuevo estado del juego:

```
def main():
    juego = Juego()
    movimiento = mcts.ejecuta(juego)
    juego = juego.aplica_movimiento(movimiento)
```

En caso de que el estado del juego no tenga ningún movimiento posible para aplicar los algoritmos devolverán *None*, y si solo hay un posible movimiento para aplicar, para ahorrar cálculos innecesarios, se devolverá el único movimiento posible.

Si se tienen dudas sobre como integrar la biblioteca a juegos propios se recomienda ver los juegos del repositorio de *GitHub* (*https://github.com/plss12/pyplAI*) como ejemplos de uso. Este repositorio contiene la biblioteca y todos los juegos nombrados en la introducción de este manual, incluyendo las implementaciones de los algoritmos correspondientes con cada uno de ellos.

Contacto

En caso de tener alguna duda, idea o aportación sobre la biblioteca por favor contactar al siguiente correo: pepoluis712@gmail.com