Universidad Internacional de La Rioja (UNIR)

ESIT

Máster Universitario en Inteligencia Artificial

Estudio de rendimiento de agentes inteligentes que juegan al juego del iwoki

Trabajo Fin de Máster

Presentado por: Videgaín Díaz, Santiago

Director/a: García Sánchez, Pablo

Ciudad: Madrid
Fecha: 13/02/2020

Resumen

El *iwoki maths* es un juego de mesa abstracto de colocación de losetas. Combina el cálculo de operaciones matemáticas simples con la percepción espacial de objetos bidimensionales. Se desarrollará una serie de agentes inteligentes con distintas capacidades de razonamiento, decisión y aprendizaje, basados en distintas técnicas de inteligencia artificial aplicada a la teoría de juegos. A su vez, el *iwoki maths* será utilizado como herramienta de investigación sobre la que se aplicarán distintas tecnologías y configuraciones. Tras la implementación del juego y de los agentes, se pondrá a prueba sus capacidades a base de jugar partidas entre ellos. Se evaluará uno de los agentes en una partida real jugando contra el autor del juego. Los resultados experimentales ratificarán conclusiones ya sabidas a nivel teórico y expondrán nuevo conocimiento que podrá ser la base de investigaciones futuras.

Palabras Clave: agente inteligente, inteligencia artificial, iwoki, Minimax, Reinforcement Learning.

Abstract

Iwoki maths is an abstract board game for laying tiles. It combines the calculation of simple mathematical operations with the spatial perception of two-dimensional objects. A series of intelligent agents with different reasoning, decision and learning capacities will be developed based on different artificial intelligence techniques applied to game theory. Moreover, *iwoki maths* will be used as a research tool on which different technologies and configurations will be applied. After the implementation of the game and the agents, their capabilities will be tested by playing games with each other. One of the agents will be evaluated in a real game playing against the author of the game. The experimental results will ratify conclusions already known at a theoretical level and will expose new knowledge that could be the basis for future research.

Keywords: artificial intelligence, intelligent agent, iwoki, Minimax, Reinforcement Learning.

Contenido

Índice de figuras

Índice de tablas

1. Introducción

Es de sobra conocida la capacidad de diversión de los juegos de mesa. Además refuerzan la creatividad y enseñan a tomar decisiones, así como a respetar normas.

El *iwoki maths* (en adelante iwoki) es un juego de mesa que combina estos beneficios con su carácter educativo. Refuerza la capacidad de análisis y activa las funciones cognitivas para poner en práctica la mejor estrategia a lo largo de las partidas.

Además, entrena la percepción espacial de objetos bidimensionales para dar paso al cálculo de operaciones matemáticas simples.

Iwoki es un juego de mesa abstracto que actualmente se encuentra en fase de prototipo, habiendo sido testeado en entornos lúdicos. Este juego resultó finalista en el concurso de prototipos Meeple Factory, que a nivel nacional organiza la Asociación Cultural CrossOver. Consecuentemente fue expuesto en el evento FicZone, en la Feria de Muestras de Armilla (Granada, España), durante el último fin de semana de abril de 2019.

Para poder seguir y entender el contenido de este trabajo, es necesario conocer con detalle las instrucciones del iwoki, disponibles en el Anexo I.

1.1. Motivación

Una de las principales características que debe tener un juego de mesa educativo, como es el iwoki, es que requiera capacidad de concentración, de abstracción y de activación cognitiva, todo ello en pro de un entrenamiento mental. Iwoki combina todo esto con las virtudes de la diversión y el entretenimiento (Bonilla, 2019).

Este trabajo de fin de máster aborda el desarrollo de varios agentes que compiten entre sí jugando al iwoki.

El carácter novedoso de este estudio comparativo y, por ende, de este trabajo de fin de máster, viene determinado por la originalidad del iwoki, que a día de hoy solamente se conoce en entornos lúdicos, como se ha comentado anteriormente.

En la Figura 1 se puede ver un ejemplo de jugada real del iwoki.

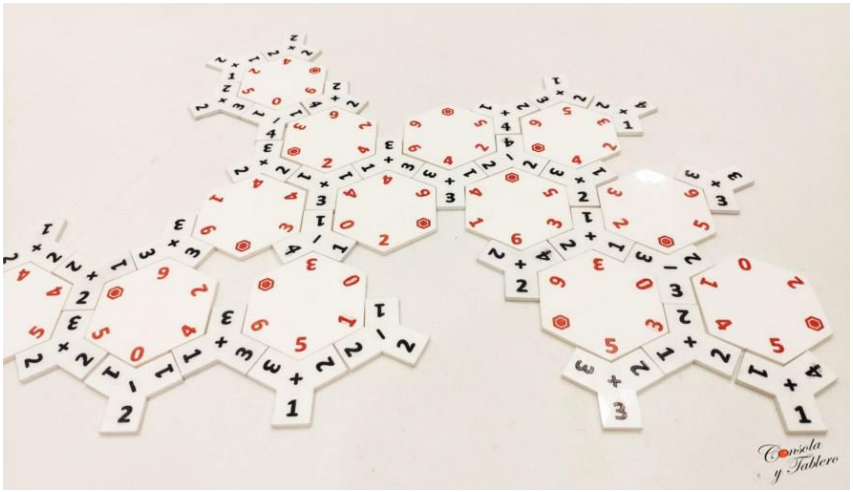


Figura 1. Ejemplo de partida del iwoki.
Fuente: (Bonilla, 2019).

1.2. Planteamiento del trabajo

Con el fin de acotar el alcance de este trabajo, el estudio se realiza sobre una versión simplificada del iwoki, donde en cada partida solo entran en liza dos jugadores. Las instrucciones del Anexo I incluyen esta modificación.

Existirán dos agentes con los que realizar enfrentamientos:

Agente 1. Elegirá sus jugadas de forma aleatoria.

Agente 2. Actuará como si fuera un jugador humano novato. Es decir, siguiendo una estrategia básica que le permita únicamente tomar decisiones a corto plazo.

Para enfrentarse a éstos, se desarrollarán otros dos agentes:

Agente 3. Tendrá en cuenta unas estrategias propias de un jugador humano experto.

Agente 4. Será capaz de aprender de su propia experiencia, poniendo en práctica el conocimiento adquirido a lo largo de sus partidas previas.

De forma adicional, el **Agente 5** hará de interfaz para permitir a un jugador humano enfrentarse al agente 3.

En este trabajo se persigue una triple finalidad:

- Por un lado se pretende estudiar cómo de efectivo es el agente 3 con respecto a los dos primeros, en función de las partidas que consigue ganar a sus contrincantes. Previsiblemente obtendrá mejores resultados, pues deberá:

    1. Evaluar la situación actual en cada turno.
    2. Razonar la jugada óptima en función de las diferentes estrategias conocidas.
    3. Decidir, en la medida de lo posible, la jugada que más le beneficie y a su vez más trabas ponga al jugador rival.

  Las estrategias tenidas en cuenta vienen explicadas en el Anexo II.

  Además, se realizará una comparativa entre instancias del agente 3 para obtener su mejor configuración, ejecutándose con distintos parámetros.

- Por otro lado, la finalidad será entrenar el agente 4 a base de jugar contra el primero, de forma que aprenda una política de qué movimiento debe realizar en cada turno para obtener la mayor puntuación al final de la partida. Una vez entrenado, se evaluará su aprendizaje enfrentándose al agente 2. Es de esperar que el rendimiento de este cuarto agente esté condicionado al entrenamiento al se someta con el primer agente.

- Finalmente se evaluará si el agente 3 es capaz de ganar al jugador humano creador del iwoki, que puede considerarse como un experto en el juego.

1.3. Estructura del trabajo

A continuación se describe la estructura de los capítulos de este trabajo de fin de máster:

- Contexto y estado del arte. Se expone el conocimiento existente relacionado con la solución propuesta, dentro del dominio de aplicación.

- Objetivo principal, objetivos específicos y metodología de trabajo utilizada.

- Desarrollo específico de la contribución. Comprende una descripción detallada de la implementación abstracta del iwoki, del modelado de los agentes y de las ejecuciones de las partidas ente ellos. Se incluye además una exposición objetiva de los resultados y una discusión de los mismos.

- Conclusiones y trabajo futuro. Se indica a modo de resumen el alcance y las conclusiones del trabajo realizado. Además se mencionan los puntos en los que se podría ampliar el estudio llevado a cabo.
- Referencias bibliográficas.
- Anexos. Se incluye información adicional de relevancia: instrucciones del iwoki, estrategias del juego, código implementado y trazas de una partida real contra el autor del iwoki.
- Artículo de investigación, en el que se excluye el segundo experimento por cuestión de espacio.

2. Contexto y estado del arte

En esta sección se expone el conocimiento aportado por varios autores, que sienta las bases de nuestro estudio y desarrollo.

2.1. Contexto

Este trabajo de fin de máster se enmarca de forma global en la teoría de juegos.

La teoría de juegos se basa en el análisis de comportamientos de varios agentes, centrados en perseguir un objetivo e interactuando entre ellos conforme a sus estrategias. Los agentes obtienen una ganancia positiva o negativa según el éxito o el fracaso de las interacciones realizadas. El ámbito de la teoría de juegos alcanza la economía, la historia, la política, las relaciones internacionales, la estrategia militar, etc. (Restrepo, 2009).

Según Aguilar (2008), podemos clasificar los juegos de diversa forma, atendiendo a algunas de sus características:

- En función del número de jugadores, el juego puede ser con o sin adversario:

  En un juego sin adversario interviene un solo agente. Los métodos más utilizados para su implementación son el de fuerza bruta sin heurística, o bien, si se quiere tener en cuenta el consumo de tiempo y de memoria como factores determinantes, el de búsqueda heurística mediante A* o IDA*. Ejemplos de juegos sin adversario son el 8-puzzle, el solitario, etc.

  En los juegos con adversario dos o más agentes compiten por un objetivo común. Ejemplos de juegos con un adversario son las 3 en raya, el ajedrez, el go, las damas, etc. Otros ejemplos de donde existe más de un adversario son el póker o el monopoly.

- Una característica relevante es el orden de los movimientos o acciones. Puede ser que el turno de los jugadores sea alternativo o por azar. También pueden ser simultáneos, en los que los agentes toman sus decisiones al mismo tiempo (piedra, papel o tijera), o secuenciales (parchís).

- El conocimiento de cada jugador sobre el resto cataloga el juego como de información perfecta, cuando no afecta el azar y la información de cada jugador se comparte para los demás (damas), o de información imperfecta, en donde sí interviene el azar y cada jugador oculta información al resto (black jack).

- El determinismo hace referencia a la intervención del azar. Así, un juego puede ser determinista si no hay espacio para el azar, o no determinista (o estocástico), si el azar interviene en alguna de las características anteriormente citadas.

- Juegos de suma cero o no cero. Von Newman y Morgenstern (1944) definen el concepto de suma cero para juegos con adversario, en los que todo aquello que consiga ganar un jugador corresponde con lo que pierde su rival. O lo que es lo mismo, la suma de lo que uno gana y el otro pierde es cero. Un ejemplo de juego de suma cero es un partido de tenis individual, donde cada punto de un jugador perjudica al contrincante. Sin embargo, en un partido de tenis de dobles, los puntos que consiga el compañero son aprovechados por uno mismo. Este caso es un ejemplo de suma no cero (Wright, 2006).

- Juegos cooperativos, en los que los participantes comparten intereses en común y actúan en beneficio de tales fines sin competir entre ellos. En los no cooperativos, cada agente se preocupa solo de sus propios intereses (Zagal, Rick, & Hsi, 2016).

- John Forbes Nash, Premio Nobel de Economía (1994) por sus investigaciones sobre la teoría de juegos y los procesos de negociación, añade un nuevo concepto para los casos de suma distinta de cero. Se trata del llamado Equilibrio de Nash, situación en la que a ninguno de los jugadores le interesa modificar su estrategia individual para no verse perjudicados, teniendo conocimiento de la de los adversarios (Mimbang, 2016).

- Aunque no será de aplicación en el desarrollo para los agentes del iwoki, por considerarse con información perfecta, otro concepto a tener en cuenta dentro de contexto de la teoría de juegos es el llamado Juego Bayesiano. Contempla como variables aleatorias la información no conocida del juego. Se aplica el teorema de Bayes estableciendo unas probabilidades sobre estados concretos del juego en función de información conocida previamente, tras una serie de acciones previas realizadas por los adversarios. El Juego Bayesiano, por tanto, tiene sentido para juegos de información imperfecta (Harsanyi, 2004).

Atendiendo a la naturaleza del iwoki, las características y propiedades que es preciso tener en cuenta de cara a la implementación de los agentes, son las siguientes:

- Se trata de un juego con adversario en el que van a intervenir dos agentes.

- Las reglas del juego son las mismas para los dos jugadores y conocidas por ambos.

- Los contrincantes jugarán su turno de forma secuencial y alternativa desde el inicio hasta el final de la partida.
- Es no cooperativo, puesto que cada agente vela por sus propios intereses y no tienen objetivos en común.
- Nos encontramos ante un juego de suma nula. Es decir, todo lo que gana un jugador en beneficio propio lo pierde el otro en igual proporción, y viceversa.
- Asimismo, podremos considerar el juego como determinista o no, según ciertas modificaciones necesarias para la implementación de algún agente en particular, que se verán a continuación.

Como se ha comentado anteriormente, el agente 1 elegirá de forma aleatoria (**Random**) el movimiento a realizar.

El agente 2 seguirá un método **Greedy,** también llamado de búsqueda voraz. Tomará la mejor decisión a corto plazo, como lo haría un jugador humano poco experimentado con el iwoki. Consiste en elegir la opción óptima en cada turno, sin tener en cuenta las consecuencias que dicha acción pueda acarrear en estados posteriores.

La implementación del agente 3, que jugará razonando de acuerdo con las estrategias de un jugador humano con mucha experiencia, utilizará el método **Minimax**. Es necesario para este caso que las fichas pequeñas de los jugadores permanezcan visibles en todo momento, pues no tiene sentido hablar de método de búsqueda Minimax si la información no es perfecta. Asimismo, se establece otra variación para que además sea determinista: el agente tendrá conocimiento del orden de las fichas que aún no se han repartido, para tenerlo en cuenta por si en su turno opta por robar.

Para el caso del agente 4, capaz de aprender de su propia experiencia previa, se va a llevar a cabo una implementación con **Reinforcement Learning**. A diferencia de la implementación para el agente Minimax, en este caso el juego mantendrá su naturaleza original:

- Estocástico, ya que las fichas se reparten de forma aleatoria.
- De información imperfecta, pues las fichas pequeñas de un jugador permanecerán ocultas para su adversario y viceversa.

2.2. Estado del arte

2.2.1. Minimax

Para juegos de dos jugadores en los que ambos intentan ganar y en donde las acciones de un jugador dependen de forma determinante de las del otro, necesitamos alguno de los algoritmos de búsqueda con adversario.

Minimax es el algoritmo básico de búsqueda con adversario, que tradicionalmente ha sido implementado de juegos de mesa con información perfecta (Yannakakis & Togelius, 2018).

Básicamente se resume en que cada jugador elige la mejor jugada para uno mismo, teniendo presente que le adversario ha escogido la opción más perjudicial para su oponente.

Para cada estado del juego el algoritmo genera un árbol completo de búsqueda, aplica a cada nodo terminal una función de utilidad (también llamada función de evaluación) y realiza una propagación de esos valores hacia el nodo raíz. Como se ilustra en la Figura 2, los nodos *Max* tendrán en cuenta los valores máximos de sus sucesores y los *Min*, lo mínimos. Al finalizar el proceso, desde el nodo raíz se tendrá visibilidad de cuál es la opción más prometedora.

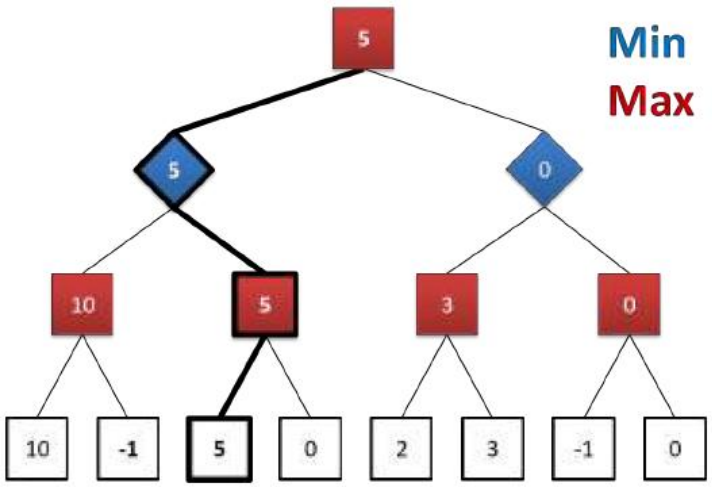


Figura 2. Árbol de búsqueda Minimax en el que se indica la estrategia óptima.
Fuente: (Yannakakis & Togelius, 2018).

El principal problema de este método es que el árbol aumenta exponencialmente con la profundidad y con el factor de ramificación del árbol. Es decir, con el número de movimientos simulados.

El orden de complejidad en tiempo es: $O(a^b)$

El orden de complejidad en espacio es: $O(ab)$

Siendo:

a: Factor de ramificación del árbol.

b: Profundidad del árbol (Número de *plys* explorados).

Por ejemplo, para el caso del ajedrez, si interpretamos que tiene un factor de ramificación promedio de 30 nodos (movimientos en cada posición) y se realizan 80 movimientos a lo largo de la partida, se explorarán 30^{80} nodos. Esta cantidad es mayor que el número de átomos que en existen en el universo.

La complejidad en tiempo y en espacio hace inviable un árbol que contemple todas las posibilidades en cada jugada.

Por regla general, las aplicaciones del algoritmo Minimax finalizan la búsqueda a una profundidad determinada. Los nodos de esa profundidad se convierten en terminales y usan una función de evaluación de estado que determina una heurística. A esta técnica se le denomina **Minimax con suspensión**.

También es posible limitar la búsqueda por tiempo de ejecución. El nodo raíz se quedará con la mejor opción encontrada hasta el momento en que se alcance el tiempo máximo.

Aplicando estas modificaciones sobre el algoritmo Minimax no se podrá garantizar que el agente encuentre la jugada óptima, pero resultará fiable si las heurísticas se aplican convenientemente.

Siguiendo estas modificaciones para solucionar el problema de la complejidad, existen diferentes técnicas de mejora de Minimax (Aguilar, 2008):

Poda Alfa-Beta:

Mediante la poda Alfa-Beta se reduce el número de nodos a explorar, al evitar pasar por nodos del árbol que no influyen en la decisión del agente. Como se puede ver en la Figura 3, cuando un nodo no aporta un valor mejor que el explorado hasta ese momento se poda la rama. (Edwards & Hart, 1963).

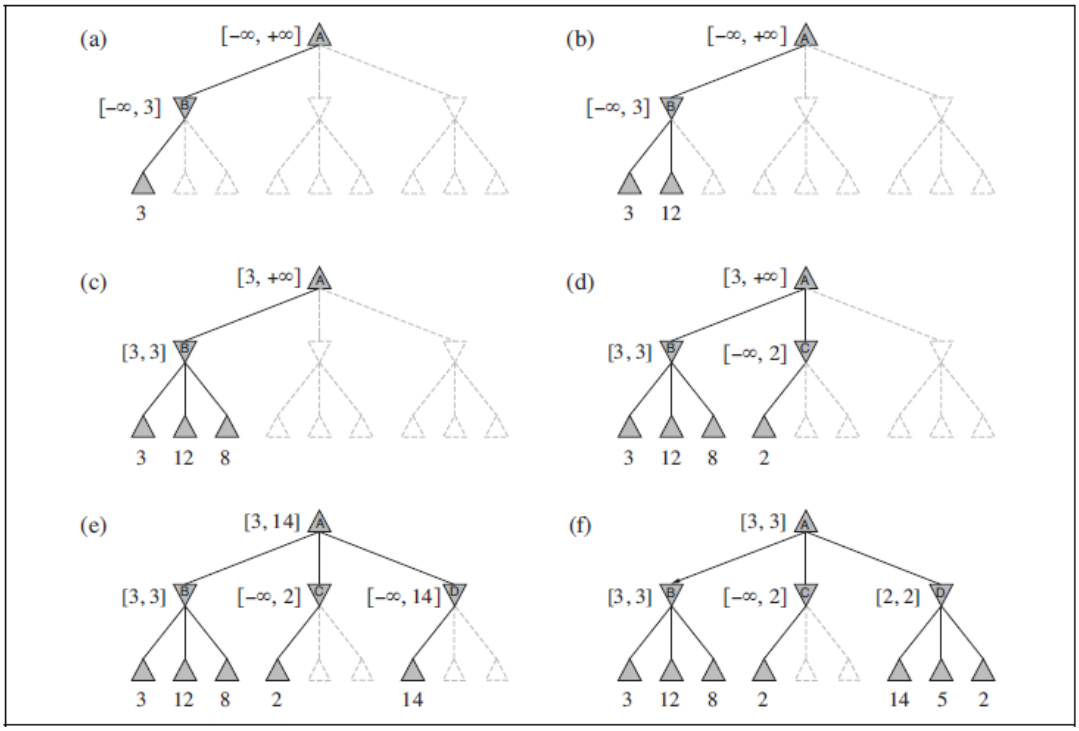


Figura 3. Construcción de un árbol Minimax con poda Alfa-Beta.
Fuente: (Russell & Norvig, 2010).

A continuación se citan variantes de la poda Alfa-Beta:

Aspiration Search:

Es una modificación de la poda Alfa-Beta en la que se utiliza un rango diferente al (–∞, +∞). Emplea una ventana más estrecha basada de la búsqueda anterior. Si el resultado es un valor dentro de esa ventana se habrá ahorrado tiempo. Sin embargo, si la búsqueda falla, se debe ampliar la ventana y buscar nuevamente.

Poda de inutilidades:

Es otra variante de la poda Alfa-Beta. En este caso, el nodo a evaluar no tiene porqué ser peor o igual que el anterior. Si no es suficientemente mejor, también se podará la rama. Es decir, aunque el nodo aporte un valor mejor que los vistos hasta ese momento, si no existe una mejora sustancial, se podrá prescindir de esa rama.

Habrá que determinar el umbral a partir del cual se considera que un valor es suficientemente mejor que otro.

Espera del reposo:

Pretende evitar el efecto horizonte, que se da cuando el árbol está limitado a una profundidad fija y se desconoce si la decisión que se toma es la correcta o hubiera sido distinta con una profundidad mayor.

Se trata de evaluar el estado del juego una vez evaluado un nodo, de forma que si cambia drásticamente, se deberá evaluar también el nodo siguiente.

Búsqueda secundaria:

También combate el efecto horizonte. Consiste en hacer una búsqueda adicional a la Minimax para evaluar si la elección del movimiento con Minimax ha sido buena. De esta forma, si se verifica que más adelante no mejora ninguna de las posibilidades, se deberá elegir otro camino.

Continuación heurística:

Es otro método más para evitar el efecto horizonte. Una vez construido el árbol hasta el nivel de profundidad predefinido, en lugar de seleccionar el mejor movimiento, ciertas heurísticas indicarán al algoritmo que debe explorar hasta una profundidad mayor. Por ejemplo, en el ajedrez, si el rey está en peligro, por lo crítico de la situación seguramente será necesario acceder a mayor profundidad en el árbol.

Movimientos de libro:

Se llaman así a las jugadas predefinidas que corresponden a determinados movimientos de la partida, como pueden ser al principio o al final.

Búsqueda sesgada:

En este caso el factor de ramificación cambia en beneficio de las jugadas más prometedoras. Así, si se sabe que un nodo ofrece información poco optimista, no merecerá la pena crear el árbol de búsqueda a partir de ahí.

Bajada progresiva:

Cuando es el tiempo el que pone cota a la búsqueda y no el nivel de profundidad del árbol, es necesario tomar una decisión dentro de un intervalo de tiempo establecido. Como se puede ver en la Figura 4, se recorren todos los nodos en amplitud hasta que la marca de tiempo llegue a su fin. En ese momento se devolverá la mejor opción del último nivel explorado por completo.

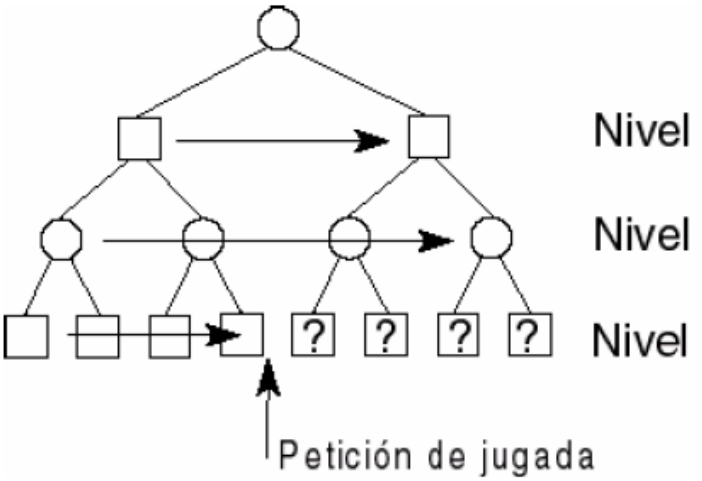


Figura 4. Generación de árbol de búsqueda con bajada progresiva.
Fuente: (Aguilar, 2008).

Poda por movimiento nulo:

La poda por movimiento nulo (Null-Move Forward Pruning) implica dar al oponente una jugada de ventaja. Si la situación en el nodo es suficientemente buena, es decir, Alfa es mayor que Beta, es de suponer que Alfa seguirá siendo mayor en nodos sucesivos, por lo que se poda esa rama. Permite reducir notablemente el número de ramificaciones, aun a riesgo de perder información que pudiera resultar relevante (David-Tabibi & Netanyahu, 2002).

Una variante es la poda por movimiento nulo verificado (Verified Null-Move Forward Pruning). En este caso no se hace un movimiento nulo, sino que se reduce la profundidad de la búsqueda (David-Tabibi & Netanyahu, 2002).

Algoritmo Megamax:

Se trata de una de una versión compacta del Minimax. La implementación de Minimax usa una subrutina para el jugador *Max* y otra para el *Min*. En lugar de esto, Megamax utiliza la puntuación negada del siguiente:

$$Max(a, b) = -Min(a, b)$$

También se le puede aplicar la poda Alfa-Beta.

Algoritmo MegaScout:

Es una mejora del Megamax que resulta óptimo cuando los nodos están ordenados. Como asume que el primer nodo es el mejor, producirá mayor número de podas.

Algoritmo SSS*:

Este algoritmo es otra alternativa a la poda Alfa-Beta. Simplifica aún más el espacio de búsqueda, ya que realiza podas en los mismos nodos que podaría el algoritmo Alfa-Beta, pero además lo hace también en otros nodos. Se basa en la utilización de subárboles del árbol de búsqueda (Plaat, Schaeffer, Pijls, & de Bruin, 1994).

Como desventaja, genera una complejidad espacial importante.

Algoritmo Scout:

Es otro algoritmo de poda. Su punto fuerte es que es más eficiente en términos computacionales que el algoritmo SSS*, al basarse en la comprobación de desigualdades. En este caso la poda se hace sobre ramas que no llevan a una mejor solución (Pearl, 1980).

2.2.2. Reinforcement Learning

El aprendizaje por refuerzo es un modelo en el que se interpreta un agente como una entidad que interactúa con el entorno y aprende de las consecuencias de sus acciones.

El **entorno** presenta al agente las distintas alternativas en cada **estado** concreto. Por cada una de estas alternativas el agente puede realizar una **acción**.

Como consecuencia de ejecutar determinada acción, el entorno reportará al agente una **recompensa** positiva (reward) o negativa (punishment) (Silver, 2015a).

La ejecución de esta acción nos llevará ante un nuevo estado con nuevas alternativas disponibles.

Este modelo viene representado en la Figura 5.

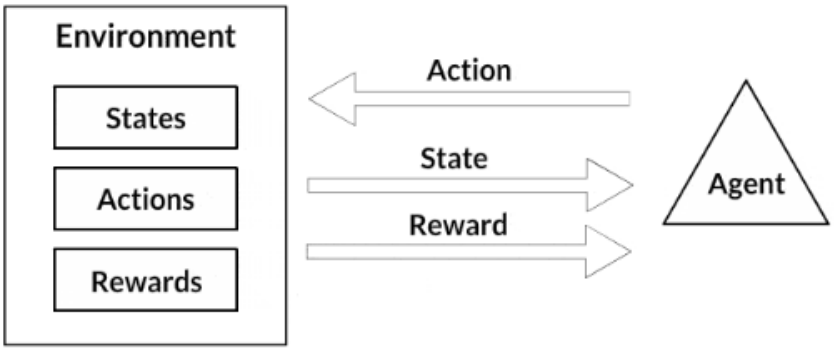


Figura 5. Esquema de aprendizaje por refuerzo.
Fuente: (Kansal & Martin).

El **Proceso de Decisión de Markov** (en inglés Markov Decision Proccess, MDP) formaliza de forma matemática los problemas de Reinforcement Learning (Andrew, 2017). Está definido por la tupla **(S, A(*s*), P, γ, R(*s, a*))**, donde:

S: Conjunto finito de estados.

A*(s)*: Conjunto finito de acciones posibles en cada estado *s*.

P: Especifica las transiciones entre estados. Es una distribución de probabilidad del estado ***s'*** al que se accede al ejecutar la acción *a* desde el estado actual *s*.

$$P(s', s \mid a)$$

γ: Es el ***discount factor*** *(factor de descuento)*. Indica cómo de importante es la recompensa a largo plazo. Toma valores entre 0 y 1. Cuanto más alto sea el valor de γ, más importancia tendrán las recompensas futuras. Por el contrario, un valor pequeño tenderá a solo tener en cuenta las recompensas inmediatas.

R*(s, a)*: Recompensas obtenidas a partir de un ejecutar la acción *a* desde un estado *s*. Es un número real. No depende de ningún estado ni acción anterior.

El agente elige acciones en base a una ***policy*** o política π, que define la conducta del agente. Determina qué acción se debe realizar en cada estado y depende únicamente del estado actual, no del histórico (Silver, 2015b). La política no cambia con el tiempo. Es decir, en un mismo estado se decantará por realizar siempre la misma acción.

La ***optimal policy*** π^* o política óptima es aquella que maximiza la suma acumulativa de recompensas. El objetivo del algoritmo es encontrar esta π^*.

Se define la ***Q-value function*** $\mathbf{Q^{\pi}(s, a)}$ como la recompensa acumulada esperada al elegir la acción *a* en el estado *s* y, después, seguir la política π.

La ***optimal Q-value function*** $\mathbf{Q^{*}(s, a)}$ obtiene el máximo valor de recompensas alcanzable con una política π a partir de un estado *s* y una acción *a*.

A medida que el agente obtiene recompensas, los *Q-values* se actualizan por medio de la ecuación de Bellman, hasta llegar al *optimal Q-value*.

Ecuación de Bellman:

Una vez ejecutada la acción *a* en el estado *s*, el sistema debe tener en cuenta el máximo valor de *Q* para cada posible acción ***a'*** en el estado ***s'***.

$$\boldsymbol{Q(s,a) \leftarrow (1 - lr) \cdot Q(s,a) + lr \cdot \left(R + \gamma \cdot \max_{a} Q(s', a')\right)}$$

lr (***learning rate)*** indica el grado en que los valores de *Q* se actualizan en cada iteración.

$\mathbf{max_a\, Q(s', a')}$ hace referencia a la estimación del valor futuro óptimo.

Cada valor de *Q(s,a)* idóneo se almacena en una tabla llamada ***Q-table***, para todos los pares estado-acción.

Cuando se utiliza la función *Q(s,a)* aprendida hasta ese momento para determinar la acción que se va a realizar, se dice que el algoritmo trabaja en modo **explotación**. Sin embargo, es muy común seleccionar acciones de forma aleatoria para poder tener en cuenta nuevas posibilidades. Se dice en este caso que el algoritmo funciona como **exploración** (Silver, 2015c).

El problema de los algoritmos de Reinforcement Learning es que sufren de *curse of dimensionality*, al crecer de forma exponencial el número de estados.

Para tratar la gran complejidad que supone un gran número de estados y acciones, se proponen dos soluciones:

MDP con descomposición Jerárquica:

Descompone el problema en un conjunto de subproblemas que son más fáciles de resolver. Establece una manera distinta de calcular *Q** que permite afrontar una gran profundidad del árbol de búsqueda de estados (Bai, Wu, & Chen, 2015).

Deep Reinforcement Learning:

Utiliza una red neuronal profunda para aproximar los valores de *Q(s, a, Θ).*

El estado *s* es el input de la red neuronal, el output es un *Q-value* por cada posible acción *a* y *Θ* son los parámetros de la red.

Deep Reinforcement Learning ha dado lugar a grandes logros en la historia reciente de la inteligencia artificial, como es el caso de AlphaGo, creado por DeepMind (Google). AlphaGo ganó en 2016 al 18 veces campeón del mundo de Go, Lee Sedol, 4 de las 5 partidas que disputaron. Hasta ese momento se pensaba que se trataba de una meta que la inteligencia artificial aún tardaría muchos años en conseguir (Silver et al., 2016)

Cabe mencionar el **Inverse Reinforcement Learning** (IRL, Aprendizaje por Refuerzo Inverso). En este caso se aprenden objetivos, valores o recompensas a base de observar el comportamiento de un agente experto. “Por ejemplo, podríamos observar el comportamiento de un humano en alguna tarea específica y aprender qué estados del entorno está tratando de lograr el humano y cuáles podrían ser los objetivos concretos” (Heidecke, 2018).

En el aprendizaje por refuerzo visto hasta ahora, al ejecutar una acción en un estado concreto se obtiene una recompensa, la cual se utiliza para aprender una política óptima. En el caso de aprendizaje por refuerzo inverso, nos basamos en la política del agente experto, del que se sabe que siempre elige la mejor acción posible, para obtener la función de recompensa que explique su comportamiento.

2.2.3. Conclusiones del contexto y del estado del arte

Hemos mencionado algunos tipos de algoritmos utilizados para modelar juegos de diversas características. Nos hemos centrado en los más apropiados para los dos agentes que incluyen comportamiento inteligente.

Como comentábamos en el capítulo Contexto, la implementación de los agentes 3 y 4 será con Minimax y Reinforcement Learning, respectivamente, pues a priori parece que estos algoritmos cubren el alcance funcional deseado en cada caso.

El agente Minimax con suspensión y poda Alfa-Beta presentará un razonamiento semejante al humano con elevada experiencia en el juego. El tiempo máximo será aplicado a la búsqueda en profundidad de cada una de las ramas del nodo raíz. Éste, junto con la profundidad del árbol, serán parámetros a considerar de forma experimental.

El agente QLearner (Reinforcement Learning) será capaz de aprovechar un conocimiento adquirido con la experiencia para elegir la mejor estrategia. La implementación seguirá un método iterativo.

3. Objetivos y metodología de trabajo

En este capítulo se van a definir los objetivos del trabajo, así como la metodología utilizada para alcanzarlos.

3.1. Objetivo general

Podemos interpretar este estudio a través de dos vertientes:

- Conocer qué implementaciones son más apropiadas para los agentes que juegan al iwoki con la finalidad de ganar y evaluar qué configuraciones optimizan los resultados.
- Por otro lado, aportar conocimiento en el campo de la inteligencia artificial aplicada a juegos, utilizando el iwoki como herramienta de investigación. Permitirá comparar diferentes agentes, establecer unas conclusiones determinadas sobre ellos y explorar nuevas líneas de investigación.

3.2. Objetivos específicos

Tras una serie de enfrentamientos entre los agentes, se sacarán las conclusiones de cuánto más efectivo es cada uno con respecto a sus rivales, según el número de partidas que consigue ganar y cuántos puntos de diferencia sea capaz de obtener.

Los experimentos se harán sobre dos líneas de investigación en paralelo:

1. Los agentes Random y Greedy serán los rivales del Minimax, en una modificación del juego que contempla información perfecta, pues, como hemos comentado anteriormente, para el agente Minimax es necesario que todas las fichas sean visibles por los dos jugadores.

   Se realizará una comparativa entre distintas instancias del mejor de los agentes (previsiblemente el mejor será el Minimax, que tiene en cuenta las estrategias propias de un jugador humano experto). Se tendrán en cuenta determinados parámetros, como el nivel de profundidad del árbol de búsqueda o el tiempo máximo de exploración de las ramas.

El autor del juego pondrá a prueba al agente Minimax con su mejor configuración en una serie de partidas Minimax vs. Human.

2. El QLearner (Reinforcement Learning) se enfrentará también a los agentes Random y Greedy, pero en este caso con la versión normal del iwoki (con información imperfecta), es decir, manteniendo ocultas las fichas pequeñas para que no las vea el oponente.

   Se hará una evaluación de la eficiencia en cuanto al tiempo de ejecución y al consumo de memoria.

De forma general, se determinará experimentalmente la mejor configuración para el iwoki.

3.3. Metodología de trabajo

Partimos de una modelización del iwoki que permita representar de forma abstracta todo el contenido del juego:

- Jugadores (agentes).
- Fichas hexagonales y pequeñas.
- Huecos donde se podrán colocar las fichas.
- Puntuaciones.
- Estado general del juego, que aglutina toda la información en un momento dado.

Asimismo, se realiza una implementación de los métodos que los agentes van a tener disponibles en el juego:

- Obtener aleatoriamente las fichas.
- Establecer el orden inicial del juego.
- Colocar una ficha pequeña o hexagonal en un hueco determinado.
- Robar una nueva ficha de alguno de los dos tipos.
- Cambiar el testigo rojo.
- Añadir un nuevo testigo blanco.

- Incrementar el contador de puntos obtenidos en una jugada.
- Hacer el recuento final de puntos.

Todo ello constituirá el espacio de juego.

Se hará distinción entre la jugada inicial, las acciones intermedias y la jugada final de la partida, de acuerdo a las diferentes estrategias que se deben tener en cuenta en los distintos momentos del juego. Estas estrategias están disponibles en el Anexo II.

A continuación se procederá al desarrollo de los cinco agentes: uno jugará de forma aleatoria, otro mediante el método Greedy, un tercero lo hará utilizando un algoritmo de búsqueda Minimax con suspensión y con poda Alfa-Beta, un QLearner aprovechará la experiencia obtenida en jugadas previas de entrenamiento y, por último, un agente que proporcione una API para que un humano pueda medirse a la efectividad de los anteriores.

Una vez finalizada la implementación, se dará paso a la fase experimental en la que los agentes realizarán enfrentamientos entre sí en partidas de dos jugadores. Cada partida podrá tener un histórico de todas y cada una de las jugadas que los agentes realicen en sus turnos. Esto permitirá hacer un seguimiento exhaustivo de la evolución de la partida e incluso poder hacer una simulación en la versión física del iwoki. Así se podrá contrastar las acciones de los agentes con lo que un humano hubiera decidido hacer en cada turno.

El número de partidas jugadas por los agentes serán las suficientes como para establecer una serie de conclusiones firmes relativas a la naturaleza de los agentes.

El conocimiento adquirido en todos los pasos anteriores permitirá identificar nuevas formas de profundizar más aún en el estudio de las soluciones implementadas, así como establecer nuevas líneas de investigación.

En la Figura 6 se indican de forma secuencial todos los pasos incluidos en la metodología.

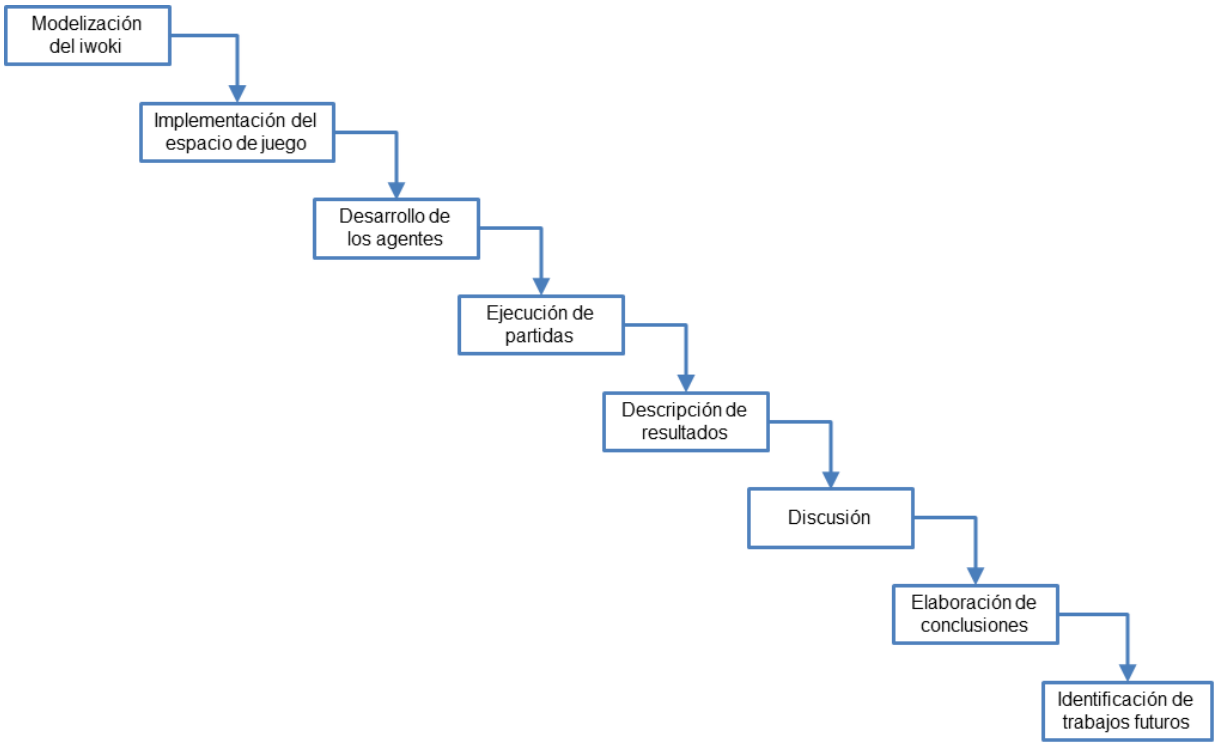


Figura 6. Esquema de la metodología de trabajo.

Fuente: elaboración propia.

4. Desarrollo específico de la contribución

A continuación se detalla cada uno de los pasos realizados en el desarrollo propuesto, conforme a lo especificado en los apartados anteriores.

4.1. Descripción detallada del experimento

El experimento se desarrolla íntegramente con lenguaje de programación Python, pues su sencillez en el desarrollo, su fácil interpretabilidad, su capacidad multiplataforma y demás características, lo hacen idóneo para este tipo de desarrollos.

El código completo de todas las clases, agentes y del resto de módulos se encuentra disponible en el Anexo III. Además está accesible desde https://github.com/videgain/iwoki.

4.1.1. Representación abstracta del iwoki

Partimos de la modelización de una representación abstracta de todo el entorno de juego, con el que los agentes puedan interactuar.

Por tanto, se realiza una implementación de las fichas pequeñas y hexagonales, de los huecos donde pueden colocarse y de los atributos y métodos comunes de los jugadores (puntuación, testigos rojo y blanco, acción de colocar fichas o de robar fichas, etc.).

En la Figura 7 se muestra un esquema de las clases que contiene la implementación del iwoki.

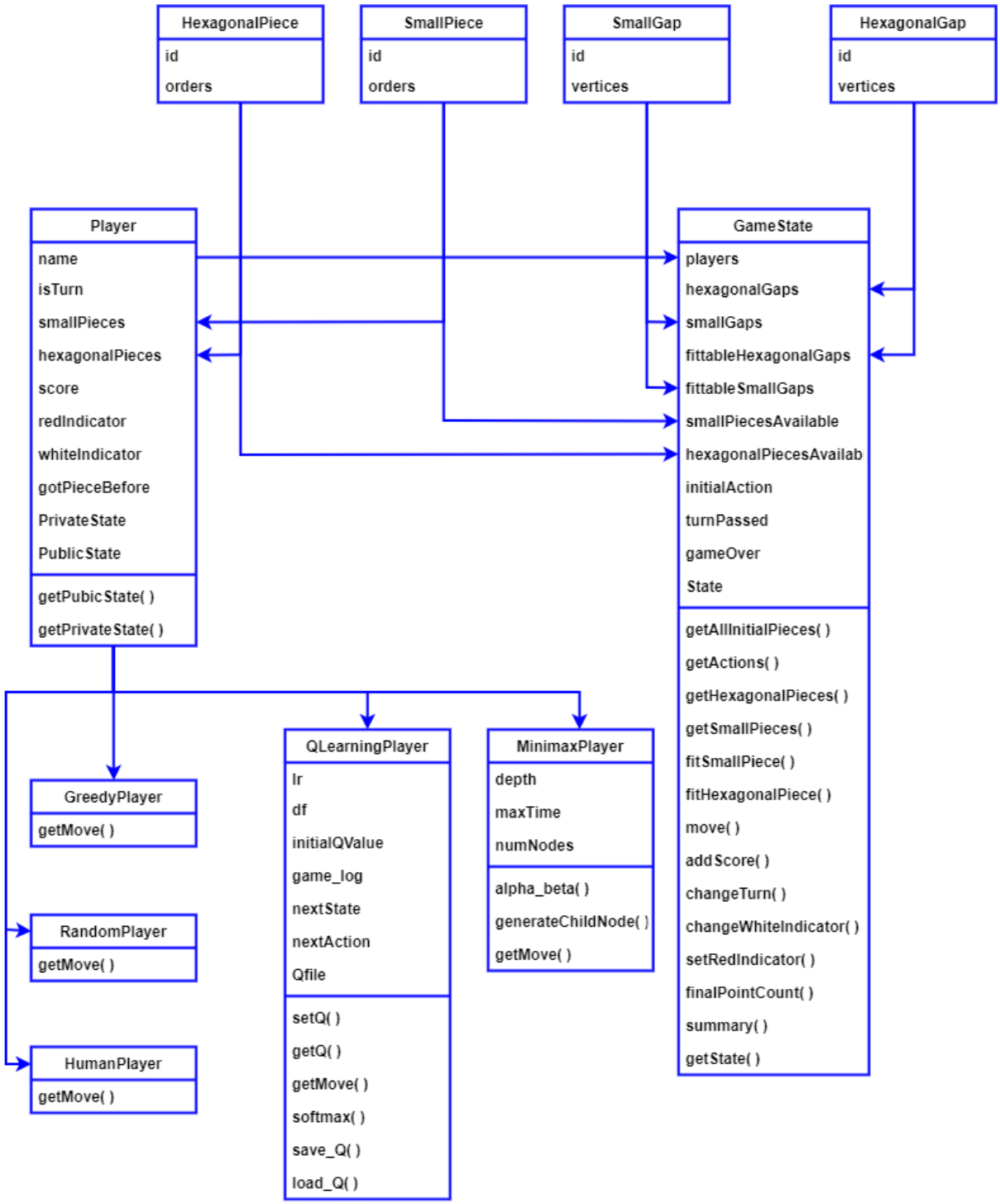


Figura 7. Esquema de las clases implementadas.

Fuente: elaboración propia.

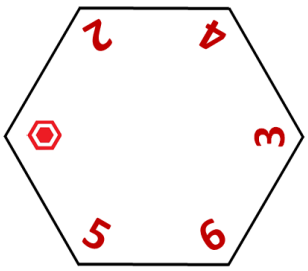
A continuación se detalla el planteamiento y contenido de cada una de las clases.

HexagonalPiece y SmallPiece se incluyen en el módulo **pieces.py**.

HexagonalPiece

Esta clase representa las fichas hexagonales, en las que en cada vértice hay un número del 1 al 6 o un comodín. Son los mismos valores por ambas caras, por lo que el orden presentado en una cara será el inverso en la otra.

Como vemos en la Figura 8, son tenidas en cuenta todas las 12 posibles variantes de cada ficha hexagonal, según los órdenes que puede presentar (6 por cada una de las caras).



Representación por ambas caras:

[2, 4, 3, 6, 5, *]	[*, 5, 6, 3, 4, 2]
[4, 3, 6, 5, *, 2]	[2, *, 5, 6, 3, 4]
[3, 6, 5, *, 2, 4]	[4, 2, *, 5, 6, 3]
[6, 5, *, 2, 4, 3]	[3, 4, 2, *, 5, 6]
[5, *, 2, 4, 3, 6]	[6, 3, 4, 2, *, 5]
[*, 2, 4, 3, 6, 5]	[5, 6, 3, 4, 2, *]

Figura 8. Representación de ficha hexagonal con sus 12 posiciones distintas.
Fuente: elaboración propia.

Los atributos *id* y *orders* hacen referencia al identificador de la ficha y a la lista con sus distintos órdenes, respectivamente.

SmallPiece

Se utiliza esta clase para modelar las fichas pequeñas. En cada una de sus 3 esquinas hay un número del 1 al 4. Cada cara de la ficha presenta los 3 mismos números y con el mismo orden. Por una cara viene representada la suma (+) y por la otra su resta (-).

En la Figura 9 se aprecian las 6 posibles variantes de cada ficha pequeña, según los órdenes que puede presentar (3 por cada una de las caras).

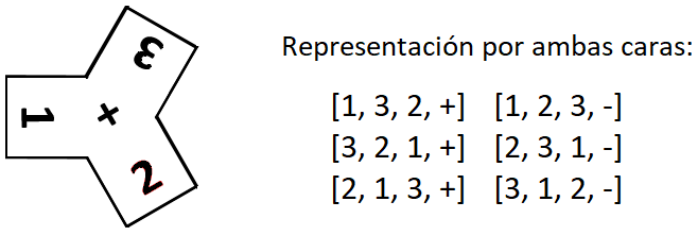


Figura 9. Representación de ficha pequeña con sus 6 posiciones distintas.
Fuente: elaboración propia.

Al igual que para la ficha hexagonal, los atributos *id* y *orders* se refieren al identificador de la ficha y a la lista con sus distintos órdenes, respectivamente.

Antes de explicar las clases HexagonalGap y SmallGap, que se encuentran desarrolladas en el módulo **gaps.py**, es preciso que quede claro el modo en que las fichas se van colocando a lo largo de la partida.

Como se ilustra en la Figura 10, el iwoki físico no requiere ningún tablero sobre el que posicionar las fichas. De cara al modelo a implementar, esto supone un gran inconveniente.

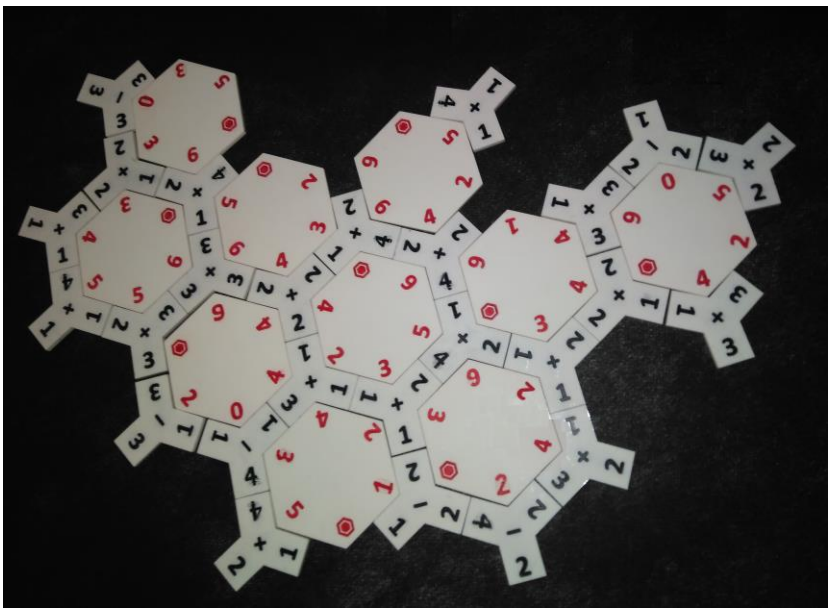


Figura 10. Colocación de fichas en una partida de iwoki.
Fuente: elaboración propia.

Para poder correlacionar las fichas, se modela un tablero virtual como el representado en la Figura 11. En él se define un conjunto de huecos fijos, donde se irán ubicando de forma ficticia las fichas pequeñas y las hexagonales.

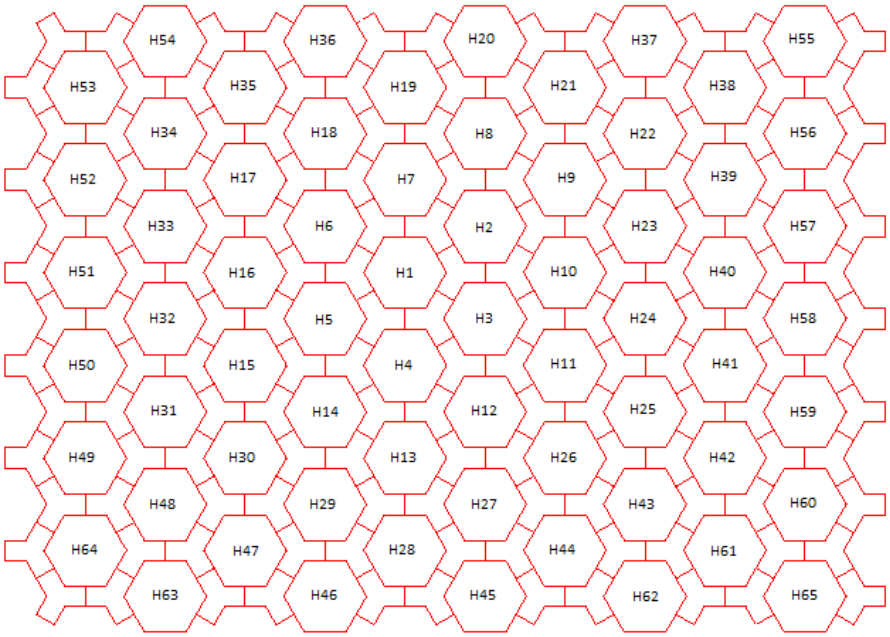


Figura 11. Representación de tablero virtual sobre el que se colocan las fichas.
Fuente: elaboración propia.

Todos estos huecos se identifican de forma unívoca.

Cada vértice del tablero es común a una ficha hexagonal y a una pequeña. Se identifica mediante las coordenadas (A, B), siendo:

- A: identificador del hueco hexagonal, que se puede apreciar en la Figura 12 con color negro (H1, H6, H7, etc.).
- B: número de orden del vértice del hueco A. Este número (del 1 al 6) será alguno de los representados en color azul en la Figura 12.

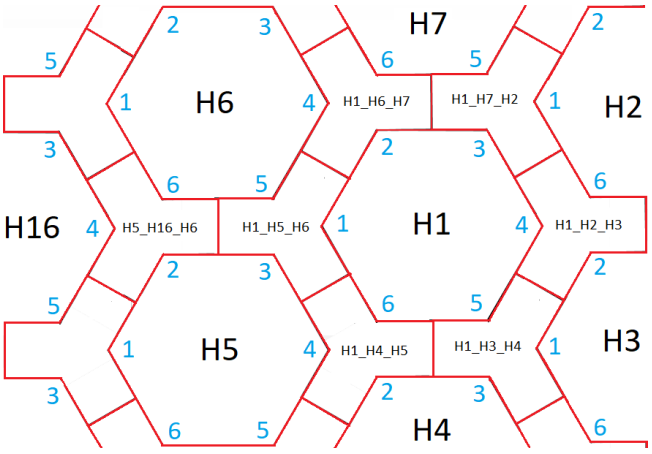


Figura 12. Identificación de los huecos y sus coordenadas.
Fuente: elaboración propia.

Podemos utilizar la Figura 12 para explicar el siguiente ejemplo:

- Los 6 vértices del hueco H1 tienen por coordenadas (H1, 1), (H1, 2), (H1, 3), (H1, 4), (H1, 5) y (H1, 6).
- El hueco pequeño identificado por H1_H5_H6 se encuentra en medio de los huecos hexagonales H1, H5 y H6 (de ahí su identificador). Sus tres vértices interiores tienen las coordenadas (H1, 1), (H5, 3) y (H6, 5).
- Es decir, el hueco de la ficha pequeña H1_H5_H6 tiene en común con el hueco hexagonal H1 las coordenadas (H1, 1), con el hueco H5 las coordenadas (H5, 3) y con el H6 las coordenadas (H6, 5).

Siguiendo ahora en la Figura 13, supongamos que se coloca la ficha pequeña [1, 1, 2, +], marcada más oscura que las demás, en el hueco H1_H5_H6. Las coordenadas de los vértices de ese hueco toman los siguientes valores:

(H6, 5) → 2 (resultado de 1 + 1)

(H1, 1) → 3 (resultado de 1 + 2)

(H5, 3) → 3 (resultado de 2 + 1)

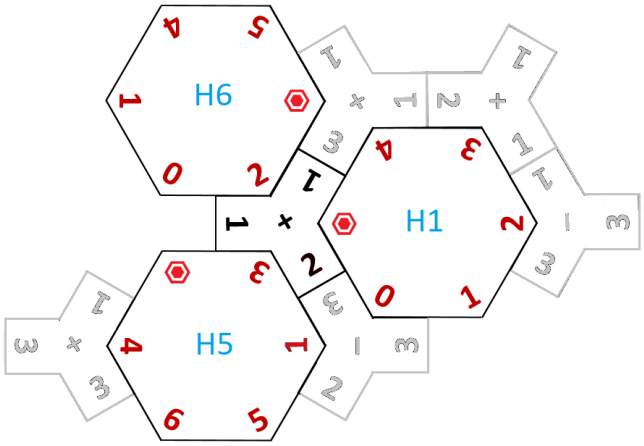


Figura 13. Ejemplo de colocación de ficha pequeña [1, 1, 2, +] en el hueco H1_H5_H6. Fuente: elaboración propia.

Es decir, como se puede observar, la ficha hexagonal que se sitúa en el hueco H6 tiene asignado en su vértice 5 el número 2. Por tanto, la ficha pequeña podrá cubrir ese vértice con el resultado de la suma 1 + 1, en las coordenadas (H6, 5). De igual forma, el resultado de la suma 2 + 1 se puede unir con el hexágono situado en la posición H5 por el vértice de coordenadas (H5, 3). Finalmente, el resultado de 1 + 2 es compatible con el comodín ( ), en el vértice de coordenadas (H1, 1). Por todo ello el sistema puede dar por válida la colocación de esa ficha pequeña en el hueco H1_H5_H6.

Veamos a continuación cómo se modelan los huecos que conforman el tablero virtual.

HexagonalGap

Es la clase que representa un hueco del tablero virtual reservado para una ficha hexagonal.

El atributo *id* de la clase hace referencia al identificador del hueco.

El atributo *vertices* define una lista de 6 elementos, cada uno de los cuales corresponde a un vértice del hexágono. Cada vértice incluye la siguiente información:

- Identificador del hueco hexagonal (H1, H2, etc.).
- Número de orden del vértice (del 1 al 6).
- Número asignado al vértice.
- Indicador del estado de ocupación del vértice ('O' → Occupied; 'V' → Vacant).

  'O' indica que el vértice del hueco hexagonal está cubierto por una ficha.

'V' quiere decir que, aunque aún no existe una ficha hexagonal cubriendo ese hueco, el vértice ya tiene asignado un número porque sí hay una ficha pequeña ocupando el hueco adyacente.

En el ejemplo de la Figura 14 se puede ver que el hueco H1 está ocupado por una ficha hexagonal. El vértice (H1, 6) tiene valor 0 (resultado de la resta 3 – 3) y su estado de ocupación es 'O' (Occupied).

Por otro lado, aunque el hueco H2 no está ocupado por ninguna ficha hexagonal, el vértice (H2, 1) tiene valor 2 (resultado de la suma 1 + 1). El estado de ocupación de este vértice es 'V' (Vacant).

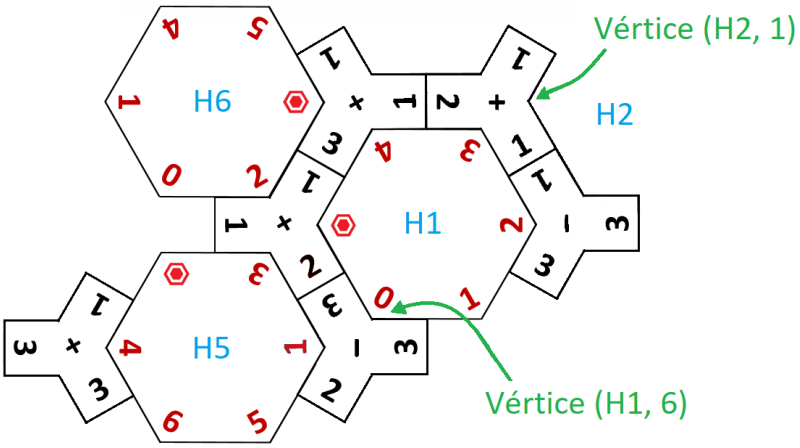


Figura 14. Vértices de huecos hexagonales.
Fuente: elaboración propia.

SmallGap

Es la clase utilizada para implementar los huecos del tablero virtual reservados para las fichas pequeñas.

El atributo *id* de la clase hace referencia al identificador del hueco.

El atributo *vértices* implementa una lista de 3 elementos que corresponden a los vértices interiores del hueco.

De forma análoga a la ficha hexagonal, la información que contiene cada vértice de una ficha pequeña es la siguiente:

- Identificador del hueco hexagonal adyacente (H1, H2, etc.).
- Número de orden del vértice del hueco hexagonal adyacente (del 1 al 6).
- Número asignado al vértice.
- Indicador del estado de ocupación del vértice ('O' → Occupied; 'V' → Vacant).

En el ejemplo de la Figura 15 observamos que el hueco H1_H6_H7 está ocupado por una ficha pequeña. El vértice (H1, 2) tiene valor 4 (resultado de la suma 3 + 1) y su estado de ocupación es 'O' (Occupied).

Por otro lado, aunque el hueco H5_H16_H6 no está ocupado por ninguna ficha pequeña, el vértice (H6, 5) tiene el valor 0 que le impone la ficha hexagonal del hueco H6. El estado de ocupación de este vértice es 'V' (Vacant).

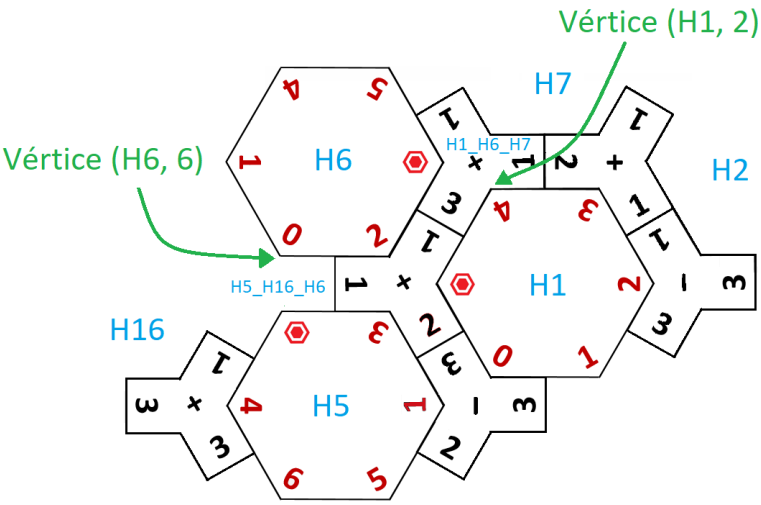


Figura 15. Vértices de huecos pequeños.
Fuente: elaboración propia.

Player

Permite modelar los jugadores. Define los siguientes atributos:

- *name*. Nombre del jugador (agente).
- *isTurn*. Flag que indica si es el turno actual del jugador.

- *smallPieces*. Lista de fichas pequeñas que tiene asignadas. Herencia de la clase SmallPiece.
- *hexagonalPieces*. Lista de fichas hexagonales que tiene asignadas. Herencia de la clase HexagonalPiece.
- *score*. Puntuación.
- *redIndicator*. Indicador o testigo rojo. Se activa en el agente para indicar al rival que le queda una única ficha pequeña.
- *whiteIndicator*. Indicador o testigo blanco. Sirve de contador del número de acciones realizadas para conseguir una nueva ficha hexagonal.
- *gotPieceBefore*. Flag que impide al jugador robar una ficha pequeña dos veces en un mismo turno.
- *PublicState*. Estado público del jugador. Está formado por todos los atributos excepto las fichas pequeñas, que el oponente no puede ver.
- *PrivateState*. Estado privado del jugador. Se incluyen todos los atributos.

Esta clase está implementada en el módulo **player.py**.

GameState

La clase GameState permite modelar toda la información del estado del juego al completo. Se definen los siguientes atributos:

- *players*. Lista de jugadores (agentes). Herencia de la clase Player.
- *hexagonalGaps*. Lista de huecos hexagonales del tablero virtual. Herencia de la clase HexagonalGap.
- *smallGaps*. Lista de huecos pequeños del tablero virtual. Herencia de la clase SmallGap.
- *fittableHexagonalGaps*. Lista de huecos hexagonales en los que se puede colocar alguna ficha hexagonal. Herencia de la clase HexagonalGap.
- *fittableSmallGaps*. Lista de huecos pequeños en los que es posible colocar alguna ficha pequeña. Herencia de la clase SmallGap.

Estos dos últimos atributos conforman los únicos huecos susceptibles de poder alojar fichas. Es decir, son los huecos que definen el perímetro del espacio de juego actual en donde los jugadores pueden colocar sus fichas.

- *hexagonalPiecesAvailable*. Lista de fichas hexagonales sin repartir. Herencia de la clase HexagonalPiece.
- *smallPiecesAvailable*. Lista de fichas pequeñas sin repartir. Herencia de la clase SmallPiece.
- *initialAction*. Flag que indica si se trata del primer turno de la partida.
- *turnPassed*. Indicador que controla la cantidad de veces que los jugadores pasan de turno. En el caso de que los dos hayan pasado de forma consecutiva en sus respectivos turnos, determina el fin de la partida y se procede al recuento final.
- *gameOver*. Flag que informa al sistema de que se ha llegado al final de la partida.
- *State*. Estado general de la partida en un momento determinado. Incluye los huecos donde se puede colocar alguna ficha (*fittableHexagonalGaps* y *fittableSmallGaps*), la información privada del jugador en cuestión (*playerInfo*) y la información pública del oponente (*opponentsInfo*).

  Como veremos posteriormente, este atributo representa el estado del sistema para el agente QLearner.

Las acciones que se implementan en esta clase son las siguientes:

- *getAllInitialPieces*. Función que reparte al inicio de la partida todas las fichas pequeñas y hexagonales a los jugadores.
- *getState*. Obtiene toda la información contenida en el atributo *State*.
- *getActions*. Devuelve todas las posibles acciones que el jugador puede realizar en el turno actual:
  - Colocar alguna de las fichas pequeñas, por el lado de la suma o de la resta, en cualquiera de sus órdenes.
  - Colocar alguna de las hexagonales disponibles, teniendo en cuenta también sus órdenes.
  - Robar una nueva ficha pequeña.

    - Pasar el turno al otro jugador.

  En cualquier caso, la función tiene en cuenta si es o no el inicio de la partida, pues en caso de que lo sea, el número de acciones a realizar es más limitado.

- *move*. Realiza una de las acciones posibles.
- *fitHexagonalPiece*. Coloca una ficha hexagonal en un hueco determinado del espacio de juego.
- *fitSmallPiece*. Ídem para una ficha pequeña.
- *getHexagonalPieces*. Permite obtener fichas hexagonales aleatoriamente de las que aún no han sido repartidas. La función es válida tanto para repartir las fichas hexagonales al inicio del juego como para robar una nueva.
- *getSmallPieces*. Igual que el anterior pero para fichas pequeñas.
- *addScore*. Incrementa la puntuación acumulada del jugador con un nuevo valor obtenido en su turno.
- *changeWhiteIndicator*. Actualiza el número de testigos blancos y, en caso de llegar a tres, roba una ficha hexagonal.
- *setRedIndicator*. Modifica el estado del testigo rojo, que indica al jugador rival que al agente le queda una sola ficha hexagonal.
- *changeTurn*. Cambia el turno al siguiente jugador.
- *finalPointCount*. Función que realiza el recuento final de puntos para cada jugador. Tal y como se especifica en las instrucciones del iwoki (Anexo I), a la cantidad de puntos que un jugador ha ido acumulando se le resta la suma de los números más altos de cada una de las fichas pequeñas que le hayan quedado sin colocar.
- *printFinalScore.* Presenta un resumen del resultado del juego una vez hecho el recuento final.
- *summary*. Muestra un resumen del juego.

La clase GameState se ubica en el módulo **gameState.py**.

Por último, existen tres módulos con funciones comunes necesarias para la modelización del juego:

initializations.py

Implementa dos funciones:

- *initializePieces* Inicializa todas las fichas hexagonales y pequeñas, que estarán disponibles a lo largo de la partida.
- *initializeGaps*. Hace lo mismo con todos los huecos de las fichas que conforman el tablero virtual.

utils.py

La función *drawPlayersOrder* establece al azar cuál de los jugadores empieza la partida.

iwoki.py

Es el módulo principal que invoca al resto. Los métodos implementados aquí son los siguientes:

- Función principal (*__main__*) desde la que se definen e inicializan los jugadores que van a participar en el juego; se crean y se reparten todas las fichas; se invoca al sorteo de qué jugador empezará la partida, y se llama al método *playGame*.

  El número de partidas que van a tener lugar se define con la constante *NUM_GAMES*.

  Se establece un *check point* para presentar un resumen de los datos relevantes hasta ese momento, a lo largo de las partidas que estén en ejecución. La Figura 16 ilustra un ejemplo de la información mostrada el *check point, que es la siguiente:*

  - Tiempo total empleado.
  - Número de partidas jugadas.
  - Tiempo medio de las partidas.
  - Promedio de turnos empleados por cada jugador.
  - Número de partidas que gana cada uno de los agentes y número de empates.
  - Tiempo medio que tarda el agente en decidir su movimiento (en segundos), solo para el agente Minimax.

- Promedio de nodos generados por partida, solo para el agente Minimax.
- Promedio de diferencia de puntos por partida.
- Número de *Q-values* actualizados.
- Número de *Q-values* totales en el *Q-file*.

Estos dos últimos datos solo se muestran cuando uno de los agentes es el QLearner.

Esta información también se carga en un *dataset* que permitirá realizar las métricas oportunas.

El número de partidas que se incluyen en cada *check point* viene determinado por la constante *CHECK_POINT*.

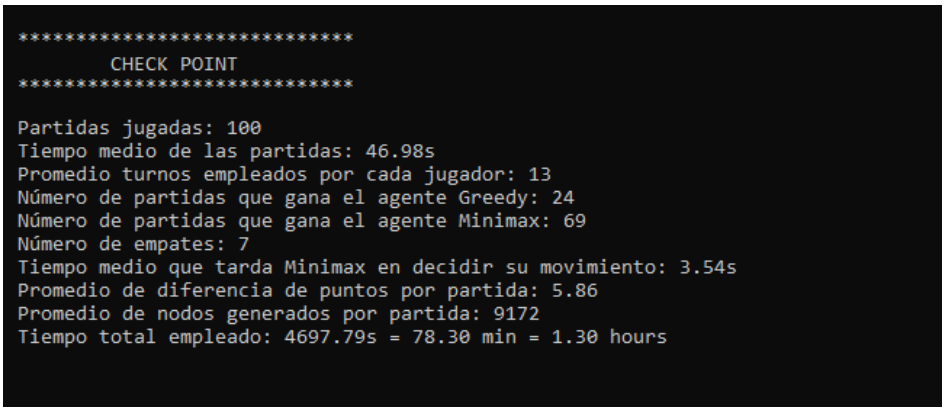


Figura 16. Check point a las 100 partidas de Minimax vs. Greedy.
Fuente: elaboración propia.

- *playGame*. Ejecuta el número de partidas especificado en *NUM_GAMES*, controlando los turnos de los dos jugadores.
- *saveMetrics*. Muestra por pantalla los *check points* y actualiza el *dataset* para las métricas.

4.1.2. Implementación de los agentes

En la Figura 17 se puede ver el *workflow* de una partida del iwoki tal y como se especifica en las instrucciones del juego.

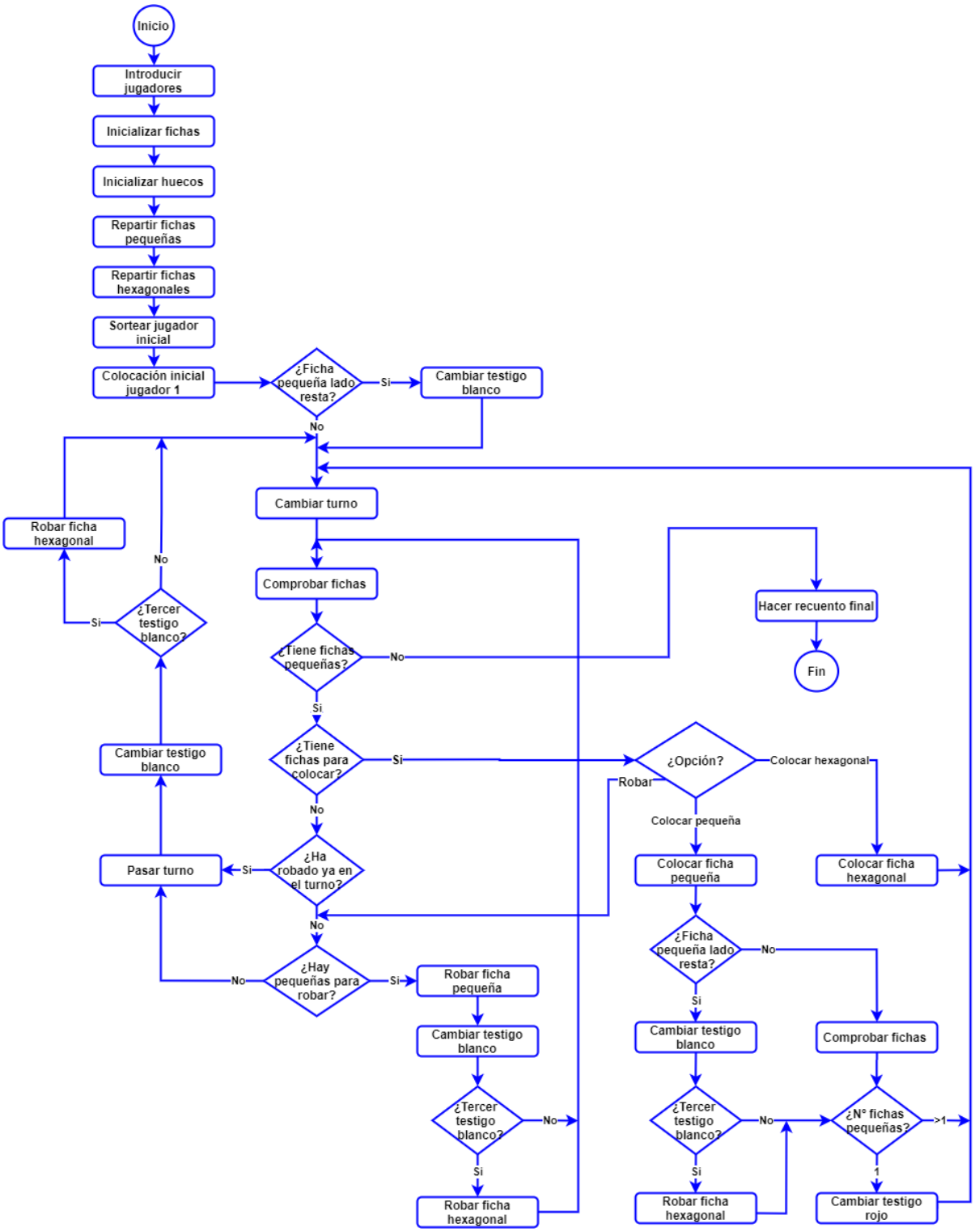


Figura 17. Workflow de una partida de iwoki.

Fuente: elaboración propia.

Agente Random

Este agente elige sus jugadas de forma aleatoria.

Su desarrollo se presenta en el módulo **random.py**.

El método *getMove* obtiene del *gameState* todas las posibles acciones a realizar en su turno y elige aleatoriamente una de ellas, sin tener en cuenta lo buena o mala que resulta la jugada.

Agente Greedy

El agente Greedy emula la manera de jugar de un humano novato. Pone en práctica la simple estrategia de elegir la opción que más puntos le proporcione de forma inmediata, sin tener en cuenta cómo de fácil le deje la jugada a su contrincante.

La implementación está en el módulo **greedy.py**.

En el método *getMove*, el *gameState* proporciona al agente todas las posibles acciones a tomar en su turno. A continuación selecciona la mejor, teniendo en cuenta la estrategia anterior.

En caso de tener distintas opciones que le reporten los mismos puntos, su preferencia será la siguiente:

1. Colocar una ficha pequeña por el lado de la resta. De esta forma se incrementa el testigo blanco, lo cual viene bien para conseguir una ficha hexagonal.
2. Colocar una ficha pequeña por el lado de la suma. En cualquier caso, colocando una ficha pequeña el jugador se acerca al final de la partida.
3. Colocar una ficha hexagonal.
4. Robar una ficha. Esto no le proporciona ningún punto, así que solo en última instancia opta por llevar a cabo esta acción. Obviamente, si ya acaba de robar, la opción será pasar el turno al otro jugador, tal y como especifican las reglas del iwoki.

Agente Minimax

Minimax aplica las estrategias que tiene un jugador humano con elevada experiencia. Tiene que anticiparse a las jugadas de su contrincante para conocer los posibles estados futuros del juego, desde el estado en que se encuentra hasta un horizonte (profundidad) establecido

por el atributo *depth*. Para ello, en cada turno debe generar un árbol de búsqueda Minimax con poda Alfa-Beta y suspensión en profundidad y en tiempo.

El código de este agente se encuentra en el módulo **minimax.py**.

El método *getMove* desencadena las siguientes acciones para obtener la mejor jugada en el turno actual:

1. Se invoca a la función *alpha_beta*, que se ejecuta de forma recursiva para los nodos *Max* y *Min*, desde el nodo raíz hasta los nodos hoja. A medida que se va explorando se generan nuevos nodos siguiendo estos pasos:

    a. El *gameState* obtiene para el estado (nodo) concreto todas las posibles acciones a realizar.

    b. Para cada una de ellas se crea un nuevo nodo de nivel inferior, invocando al método *generateChildNode*, que:

        - Hace un clon del *gameState*.
        - Ejecuta la acción sobre el nuevo *gameState*. Éste es el nuevo estado del juego asociado al nodo generado.

2. Se va ramificando el árbol recursivamente hasta que se encuentra un nodo hoja de dos formas distintas:

    - Llegando al nivel de profundidad establecido con el atributo *depth*.
    - Llegando al final de la partida en la correspondiente ramificación sin haber alcanzado el nivel de profundidad.

    La función de utilidad se halla mediante la resta de las puntuaciones tras haber realizado el recuento final de puntos.

3. Deshaciendo la recursividad se actualizan los valores de *Alpha* y *Beta* hasta dar valor al nodo raíz, a la vez que permiten hacer podas por las ramas que no van a presentar valores mejores que los encontrados hasta el momento.

El atributo *maxTime* determina el tiempo máximo de búsqueda en profundidad a lo largo de cada una de las ramas principales del nodo raíz.

Por ejemplo, para un estado del juego en que el agente Minimax tenga 6 posibles acciones y el valor de *maxTime* sea 10 segundos, si alguna de esas 6 ramas no se ha explorado por

completo en 10 segundos, se propagará la mejor opción encontrada hasta el momento y se seguirá por la siguiente rama del nodo raíz. De esta forma se limita la búsqueda total en ese turno a 60 segundos como máximo.

Al ser una búsqueda con suspensión, el algoritmo no garantiza encontrar la mejor acción en cada turno, pero estableciendo unos parámetros adecuados de tiempo (*maxTime*) y profundidad (*depth*), como se verá en los siguientes puntos, los resultados podrán ser satisfactorios.

Agente QLearner

Este agente encuentra su fortaleza a base de entrenar jugando numerosas partidas. Cuanto mayor es su entrenamiento se esperará de él un mayor rendimiento contra a sus rivales.

El módulo **qLearner.py** contiene el desarrollo, realizado con Reinforcement Learning.

El peso de la implementación recae sobre el método principal *getMove*, que realiza las siguientes acciones:

1. Para un estado concreto, el *gameState* proporciona todas las acciones que el agente puede realizar.
2. Para determinar la acción a ejecutar:
    a. El método *load_Q* obtiene de la *Q-table* los *Q-values* estado-acción que tuviera almacenados previamente. Los que no existan se inicializan con un hiperparámetro definido la constante *INITIALQVALUE*.
    b. En modo **exploración**, se selecciona la acción a realizar por medio de una función aleatoria ponderada sobre los *Q-values*. Éstos toman valor previamente por medio del método *softmax*, que obtiene una distribución de probabilidad. Así, los valores están comprendidos entre 0 y 1 y la suma de todos ellos es 1.
    c. En modo **explotación**, simplemente se selecciona la acción que tenga el mayor *Q-value*. En caso de tener varias acciones con este máximo valor, se toma una aleatoriamente.
3. Se ejecuta la acción seleccionada. Cada movimiento realizado a lo largo de la partida se almacena en una lista de valores estado-acción.
4. Al finalizar la partida:

a. Se calcula el valor de la recompensa como la diferencia entre los puntos conseguidos por el agente QLearner y los de su oponente, una vez realizado el recuento final de puntos.

b. A partir de esa recompensa se aplica la **ecuación de Bellman** de forma **iterativa** sobre la lista de valores estado-acción anterior, recorrida en sentido inverso (es decir, primero el estado-acción que ha llevado al final de la partida y a continuación el resto hasta llegar al estado-acción del inicio). Se obtienen así todos los *Q-values* de los movimientos ejecutados en la partida.

   En la ecuación de Bellman intervienen los hiperparámetros *Learning Rate* y *Discount Factor*, cuyos valores se cargan en los atributos *lr* y *df*, respectivamente.

c. Los *Q-values* se almacenan en la *Q-table*.

5. Tras la ejecución del número de partidas que conforma el entrenamiento del agente, la *Q-table* se almacena en un fichero *.pkl* con el fin de reutilizar el aprendizaje en futuras ejecuciones. Esto lo realiza el método *save_Q*.

Agente Human

El método *getMove* implementa una interfaz que permite a un jugador humano enfrentarse a cualquiera de los otros agentes. Un ejemplo de esta interfaz se puede ver en la Figura 18.

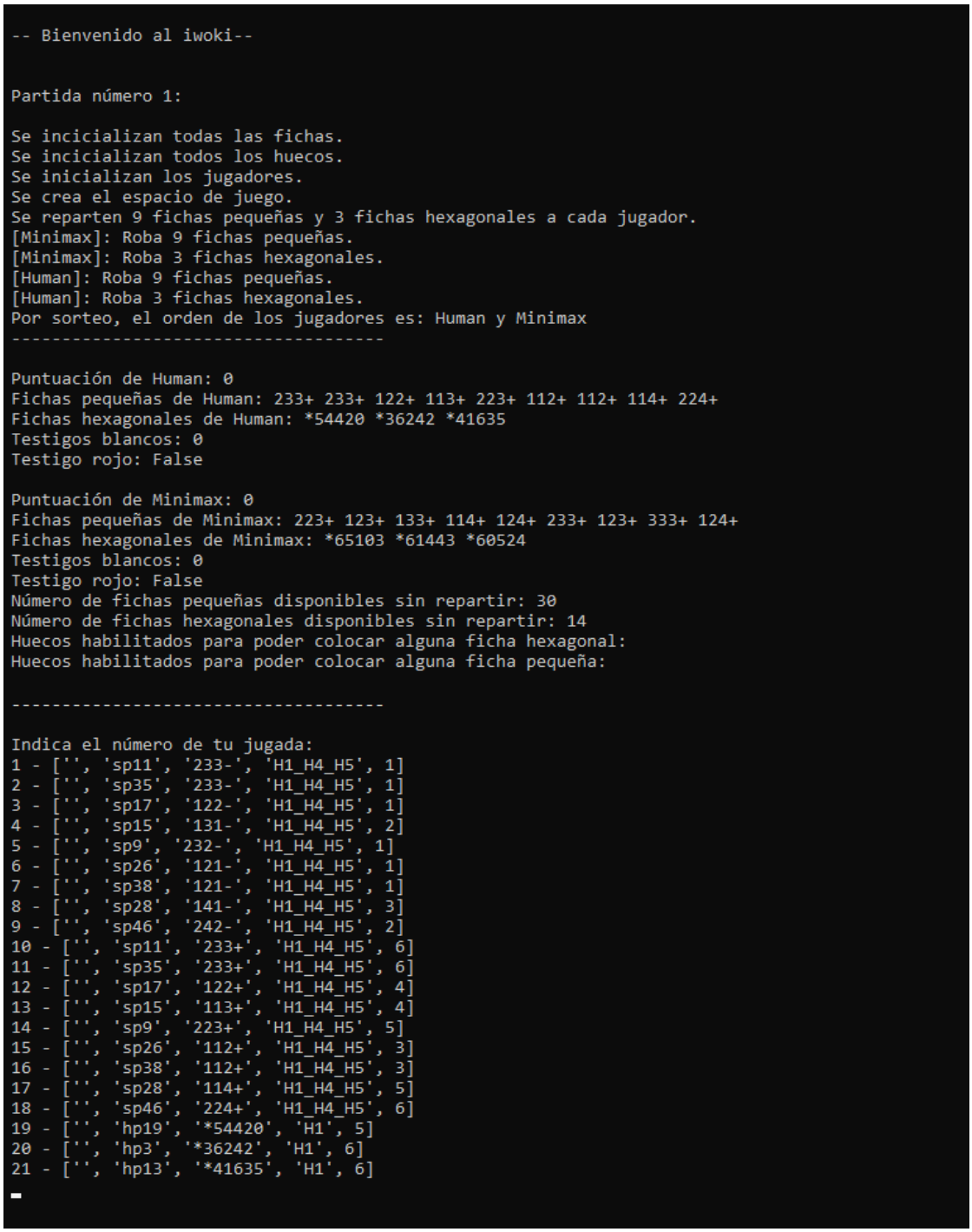


Figura 18. Interfaz de usuario para el jugador humano.

Fuente: elaboración propia.

4.1.3. Ejecución de las partidas

Para la correcta interpretación de los resultados, lo primero que se hace es un enfrentamiento de 100 partidas entre los agentes Greedy y Random. Así se podrá tener una referencia de cómo de bueno es el primero con respecto al segundo cuando otros agentes compitan contra ambos.

Como se ha comentado anteriormente, se van a llevar a cabo dos líneas experimentales de forma independiente, según la naturaleza de los agentes que van a intervenir en las partidas.

Experimento 1. Partidas con el agente Minimax

En esta línea experimental se utilizará una modificación del iwoki con el fin de conseguir que sea determinista y de información perfecta. Para ello:

- El estado de juego (*gameState*) no ocultará las fichas pequeñas y hexagonales que aún no se han repartido. Así el agente sabrá con antelación qué ficha es la que le corresponde robar si opta por esta acción en su turno y, del mismo modo, también sabrá cuál es la que su rival podría obtener en su caso. De este modo se consigue que sea determinista, eliminando por completo el azar una vez que se han repartido las fichas y la partida está iniciada.

- Las fichas pequeñas de los jugadores, al igual que las hexagonales, permanecerán visibles en todo momento. Así el juego será de información perfecta.

Minimax jugará partidas con los agentes Random y Greedy. Empleará diferentes configuraciones de acuerdo a los hiperparámetros que se van a utilizar, que son:

- **Nivel de profundidad del árbol de búsqueda (*depth*):** Se utilizarán valores de 1, 2, 3 y 4 niveles. Para los de profundidad 1, 2 y 3 se van a ejecutar 100 partidas, que permitirán evaluar los resultados y sacar determinadas conclusiones. Debido al alto conste computacional, solamente se ejecutarán 10 partidas cuando se aplique nivel de profundidad 4.

- **Tiempo máximo de búsqueda sobre cada una de las ramas principales del nodo raíz (*maxTime*):** Se aplicará un rango de valores que van desde 0,5 a 90 segundos. Como se verá en el siguiente apartado, solo tiene sentido aplicar este

hiperparámetro en partidas contra el agente Greedy sometidas a nivel de profundidad 3.

Tras jugar todas las partidas anteriores se determinará cuáles son las dos configuraciones de Minimax que son susceptibles de resultar óptimas para jugar contra un jugador humano, de forma que quede equilibrada su eficacia con el tiempo de espera en ejecutar su turno.

El agente Minimax jugará 20 partidas contra sí mismo aplicando estas dos configuraciones para determinar definitivamente cuál es la más apropiada.

Ésta será la que utilice el agente para jugar 5 partidas contra el creador del iwoki.

Experimento 2. Partidas con el agente QLearner

Inicialmente se lanzarán 10.000 partidas entre los agentes QLearner y Random y otras tantas entre QLearner y Greedy con el fin de establecer una referencia de cómo se comportan antes de realizar ningún entrenamiento.

A continuación se llevarán a cabo dos sub-experimentos con QLearner. En cada uno de ellos habrá dos fases:

1. En un primer paso se entrena el modelo para que adquiera la mayor experiencia posible. El entrenamiento pretende aportarle el conocimiento de la mejor política a seguir para determinar qué acción le conviene tomar en cada estado y conseguir así la mejor puntuación al final de la partida.

   El entrenamiento de QLearner se hará con partidas en modo exploración contra el agente Random, pues es el que explora todas las posibles acciones en cada uno de los estados. Si lo hiciera contra el Greedy su entrenamiento no sería tan exhaustivo, ya que solo aprendería del rival las mejores acciones a corto plazo.

2. Una vez entrenado el agente, la siguiente fase es poner en práctica su experiencia. Ahora las partidas de QLearner serán en modo explotación. El agente optará en cada estado por la acción que más beneficio a la larga le proporcione, tal y como habrá aprendido en el entrenamiento previo.

En el primer sub-experimento se ejecutará una batería de 100.000 partidas en modo exploración entre los agentes QLearner y Random. Una vez finalizado el entrenamiento:

1. Se jugarán 50.000 partidas QLearner vs. Random.

2. QLearner vs. Greedy se enfrentarán en otras 50.000 partidas.

Para el siguiente sub-experimento se opta por modificar el iwoki con el fin de reducir drásticamente el espacio de estados posibles. Al inicio se repartirán aleatoriamente las fichas pequeñas y hexagonales, pero se establece una semilla fija para que siempre sean las mismas en todas las partidas que se jueguen. Las que se roben a lo largo de la partida sí serán aleatorias sin semilla.

Al igual que en el anterior experimento, se entrena al QLearner con 100.000 partidas con QLearner vs. Random y posteriormente se realizan los mismos enfrentamientos (50.000 QLearner vs. Random y 50.000 QLearner vs. Greedy).

En cuanto a los hiperparámetros, en todos los casos se aplica un *Learning Rate* con valor 0,4 y un *Discount Factor* de 0,8.

4.2. Descripción de los resultados

Antes de nada, los agentes Greedy y Random se enfrentan en 100.000 partidas. El resultado se puede ver en la Tabla 1.

Tabla 1. Resultados de partidas Random vs. Greedy.

Victorias			Promedio por partida	
Greedy	Random	Empates	Duración Partida	Dif. puntos
100	0	0	0,16 seg	9,73

Fuente: elaboración propia.

Al igual que en el apartado anterior, los resultados se especifican por separado para las dos líneas experimentales.

Resultados del experimento 1 (agente Minimax)

Una vez ejecutadas las partidas detalladas en el apartado anterior, los resultados obtenidos pueden verse en la Tabla 2.

Tabla 2. Resultados de partidas de Minimax contra Random y Greedy.

		Parámetros Minimax			Victorias			Promedio por partida				
Vs		DEPTH	MAX_TIME	Nª Partidas	Minimax	Rival	Empates	Duración	Dif. puntos	Turnos	Tiempo decisión	Nodos
Minimax	Random	1	-	100	100	0	0	2,97 seg	24,50	13	0,22 seg	339
Minimax	Random	2	-	100	100	0	0	27,73 seg	26,94	14	1,97 seg	3.498
Minimax	Random	3	-	100	100	0	0	3,98 min	26,50	14	16,13 seg	44.200
Minimax	Random	4	-	10	10	0	0	25,58 min	27,20	13	1,94 min	278.857
Minimax	Greedy	1	-	100	47	49	4	3,87 seg	0,70	12	0,30 seg	455
Minimax	Greedy	2	-	100	71	25	4	44,81 seg	5,29	12	3,54 seg	6.315
Minimax	Greedy	3	-	100	90	8	2	11,65 min	7,50	13	50,27 seg	140.505
Minimax	Greedy	4	-	10	10	0	0	48,45 min	9,67	12	3,14 min	548.321
Minimax	Greedy	3	0,5 seg	100	46	49	5	9,81 seg	-1,05	12	0,77 seg	1.719
Minimax	Greedy	3	1 seg	100	41	57	2	15,20 seg	-1,88	12	1,18 seg	2.800
Minimax	Greedy	3	5 seg	100	49	43	8	58,39 seg	0,69	13	4,43 seg	11.563
Minimax	Greedy	3	10 seg	100	48	52	0	1,89 min	-0,82	13	8,52 seg	13.545
Minimax	Greedy	3	20 seg	100	60	28	12	3,21 min	1,36	13	13,96 seg	37.979
Minimax	Greedy	3	30 seg	100	60	28	12	4,41 min	3,28	13	19,70 seg	53.063
Minimax	Greedy	3	40 seg	100	56	32	12	6,03 min	5,20	13	27,67 seg	51.473
Minimax	Greedy	3	50 seg	100	76	20	4	6,72 min	7,80	13	29,90 seg	88.872
Minimax	Greedy	3	60 seg	100	60	38	2	7,93 min	6,94	14	33,52 seg	97.015
Minimax	Greedy	3	70 seg	100	82	16	2	8,00 min	9,17	13	36,67 seg	97.429
Minimax	Greedy	3	80 seg	100	80	19	1	9,51 min	10,70	14	39,89 seg	118.672
Minimax	Greedy	3	90 seg	100	95	1	4	9,85 min	16,60	13	44,43 seg	122.166

Fuente: elaboración propia.

En las gráficas que se presentan a continuación vemos cómo evolucionan los valores promedio de:

- Duración de las partidas.
- Diferencia de puntos.
- Tiempo de espera en tomar una decisión en cada turno.
- Número de nodos generados en el árbol de búsqueda.

Minimax vs. Random

Dado que el agente Random no consigue ni una sola victoria contra el Minimax, no tiene sentido realizar enfrentamientos modificando el hiperparámetro *maxTime* entre estos dos, pues no aportaría información relevante.

Los valores promedio a medida que aumenta la profundidad se pueden ver en la Figura 19.

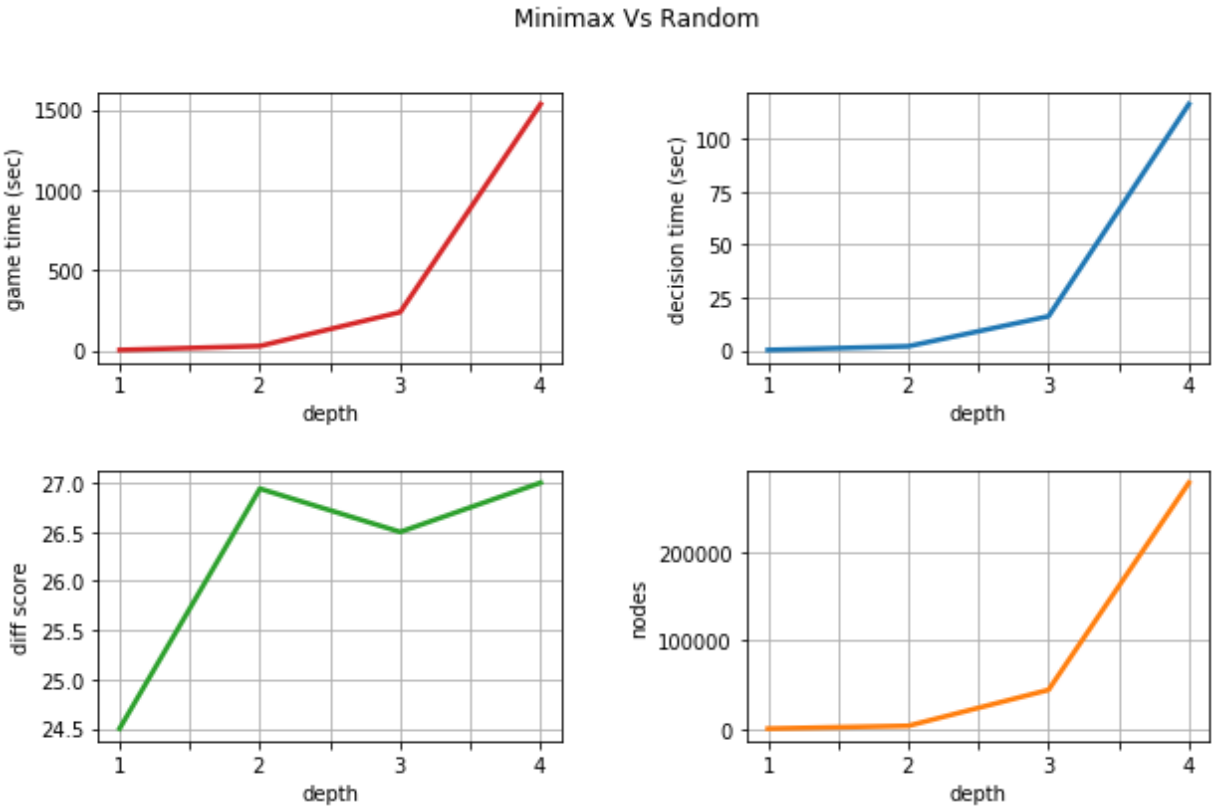


Figura 19. Valores promedio en partidas Minimax vs. Random al incrementar *depth*.
Fuente: elaboración propia.

Minimax vs. Greedy

Para este caso, la Figura 20 muestra la evolución de los valores promedio de las partidas según aumenta *depth*.

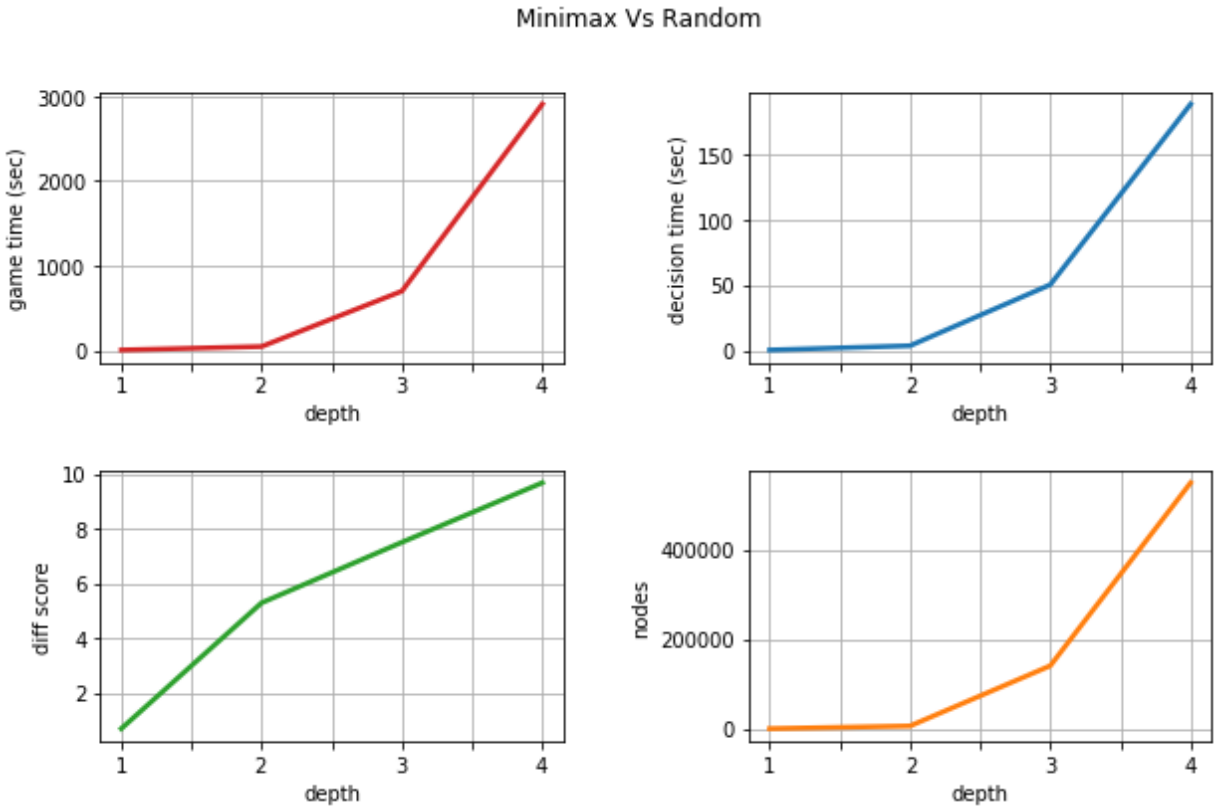


*Figura 20. V*alores promedio en partidas Minimax vs. Greedy al incrementar *depth*.

Fuente: elaboración propia.

Se observa en la Tabla 2 que la evolución de las partidas ganadas por Minimax frente Greedy es 47%, 71%, 90% y 100%, en partidas con profundidades 1, 2, 3 y 4, respectivamente. Se aprecia de forma gráfica en la Figura 21.

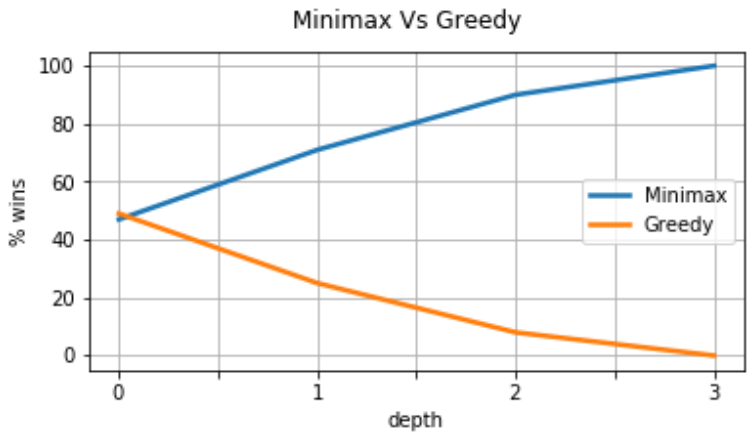


*Figura 21. V*ictorias de Minimax contra Greedy al incrementar *depth*.

Fuente: elaboración propia.

Cuando el parámetro que se incrementa es el *maxTime*, los valores promedio de las partidas evolucionan de acuerdo a lo representado en las gráficas de la Figura 22.

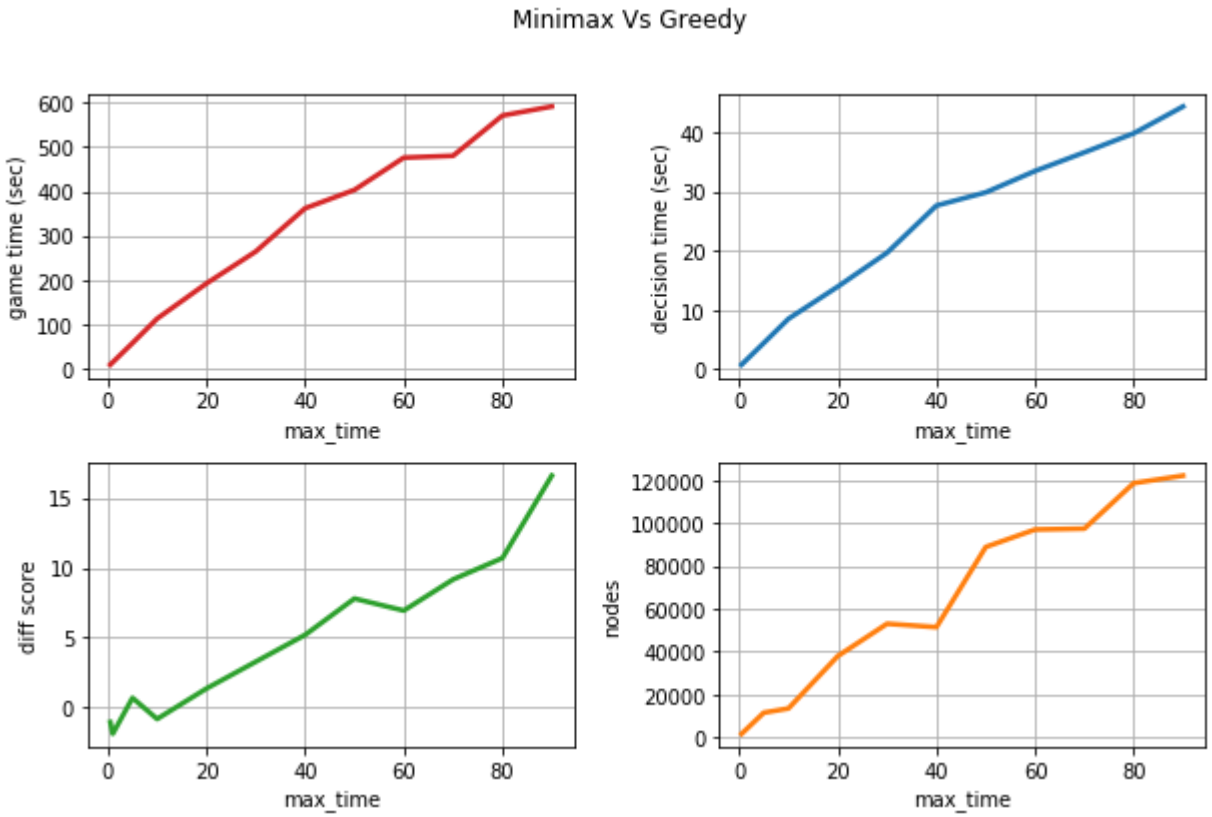


Figura 22. Valores promedio en partidas Minimax vs. Greedy al incrementar *maxTime*. *Fuente:* elaboración propia.

Minimax vs. Minimax

El experimento tiene lugar entre dos agentes Minimax con distintas configuraciones:

Agente Minimax 1: *depth = 3* y tiempo de espera ilimitado.

Agente Minimax 2: *depth= 3* y *maxTime = 90* segundos.

El resultado de las 20 partidas es el de la Tabla 3.

Tabla 3. Resultados de partidas entre dos agentes Minimax con distinta configuración.

Victorias			Promedio por partida	
Minimax 1	Minimax 2	Empates	Duración Partida	Dif. puntos
10	8	2	22,19 min	2,50

Fuente: elaboración propia.

Minimax vs. Human

Se ejecutan 5 partidas entre el autor del iwoki y el agente Minimax configurado con *depth= 3* y *maxTime = 90* segundos. Los resultados se pueden ver en la Tabla 4.

Tabla 4. Puntuaciones finales de las partidas Minimax vs. Human.

	Puntuaciones	
	Minimax	Human
Partida 1	48	43
Partida 2	51	47
Partida 3	59	54
Partida 4	51	45
Partida 5	65	62

Fuente: elaboración propia.

Durante el juego se comprueba que el tiempo de espera medido en cada turno corresponde con el obtenido en las partidas jugadas contra Greedy.

Resultados del experimento 2 (agente QLearner)

En la Tabla 5 se muestran los resultados de cada una de las baterías de partidas para los distintos sub-experimentos especificados en el apartado anterior.

Tabla 5. Resultados de partidas de QLearner contra Random y Greedy.

	Vs			Nª Partidas	Victorias Qlearner	Victorias Rival	Empates	Promedio por partida Dif. puntos	Promedio por partida Duración	Promedio por partida Turnos	Q-values Totales	Q-values Actualizados	Tamaño Q-file
	QLearner	Random	R	10.000	3.706	5.996	298	-5,11	0,21 seg	13	3.686.206	437.932	465,40 MB
	QLearner	Greedy	R	10.000	24	9.970	6	-32,68	0,19 seg	12	4.266.728	453.224	540,51 MB
*	QLearner	Random	R	100.000	37.305	59.536	3.159	-5,02	0,24 seg	13	36.967.241	4.379.700	4.652,88 MB
*	QLearner	Random	T	50.000	18.671	29.829	1.500	-5,07	0,25 seg	14	55.442.209	2.185.249	-
*	QLearner	Greedy	T	50.000	112	49.841	47	-32,64	0,24 seg	13	58.167.883	2.276.841	-
*	QLearner	Random	R	100.000	49.532	48.205	2.263	-0,20	0,20 seg	13	25.965.300	9.767.824	3.238,99 MB
*	QLearner	Random	T	50.000	22.777	26.208	1.015	-2,62	0,24 seg	13	39.591.285	4.625.747	-
*	QLearner	Greedy	T	50.000	41.607	8.386	7	-3,60	0,21 seg	12	28.707.442	16.570.374	-

* Sub-experimento 1. Se reparten al principio las fichas de forma aleatoria
* Sub-experimento 2. Se reparten al prinicipio las mismas fichas en todas las partidas
R Partidas en modo exploración
T Partidas en modo explotación

Fuente: elaboración propia.

Las ejecuciones de las primeras 10.000 partidas de QLearner vs. Random y de QLearner vs. Greedy muestran el comportamiento del agente QLearner, en modo exploración, sin realizar ningún entrenamiento previo.

Es necesario explicar que los *Q-values* totales son todas las tuplas (estado, acción) que se generan en todas las partidas. Es decir, habrá un *Q-value* por cada una de las acciones posibles en cada estado (turno de juego) de cada una de las partidas jugadas, siempre que no se haya generado antes para un mismo (estado, acción).

Mientras que los *Q-values* actualizados son los que, habiéndose generado previamente, se actualizan con un valor distinto al de inicialización.

En las figuras que se van a presentar a continuación vemos de forma gráfica el contenido de la Tabla 5. A medida que se completan las partidas de cada sub-experimento en cuestión, se puede apreciar la evolución de los siguientes valores:

- Partidas ganadas del agente QLearner con respecto a su oponente (Random o Greedy).
- Porcentaje de partidas ganadas de QLearner.
- Puntuación de los dos agentes.
- Diferencia de puntos.
- *Q-values* generados y actualizados.

- Porcentaje de *Q-values* actualizados con respecto a los totales generados.

Sub-experimento 1

No se establece ningún tipo de semilla en el procedimiento aleatorio de repartir fichas al inicio de la partida.

El entrenamiento de 100.000 partidas en modo exploración del agente QLearner contra Random presenta los resultados de la Figura 23.

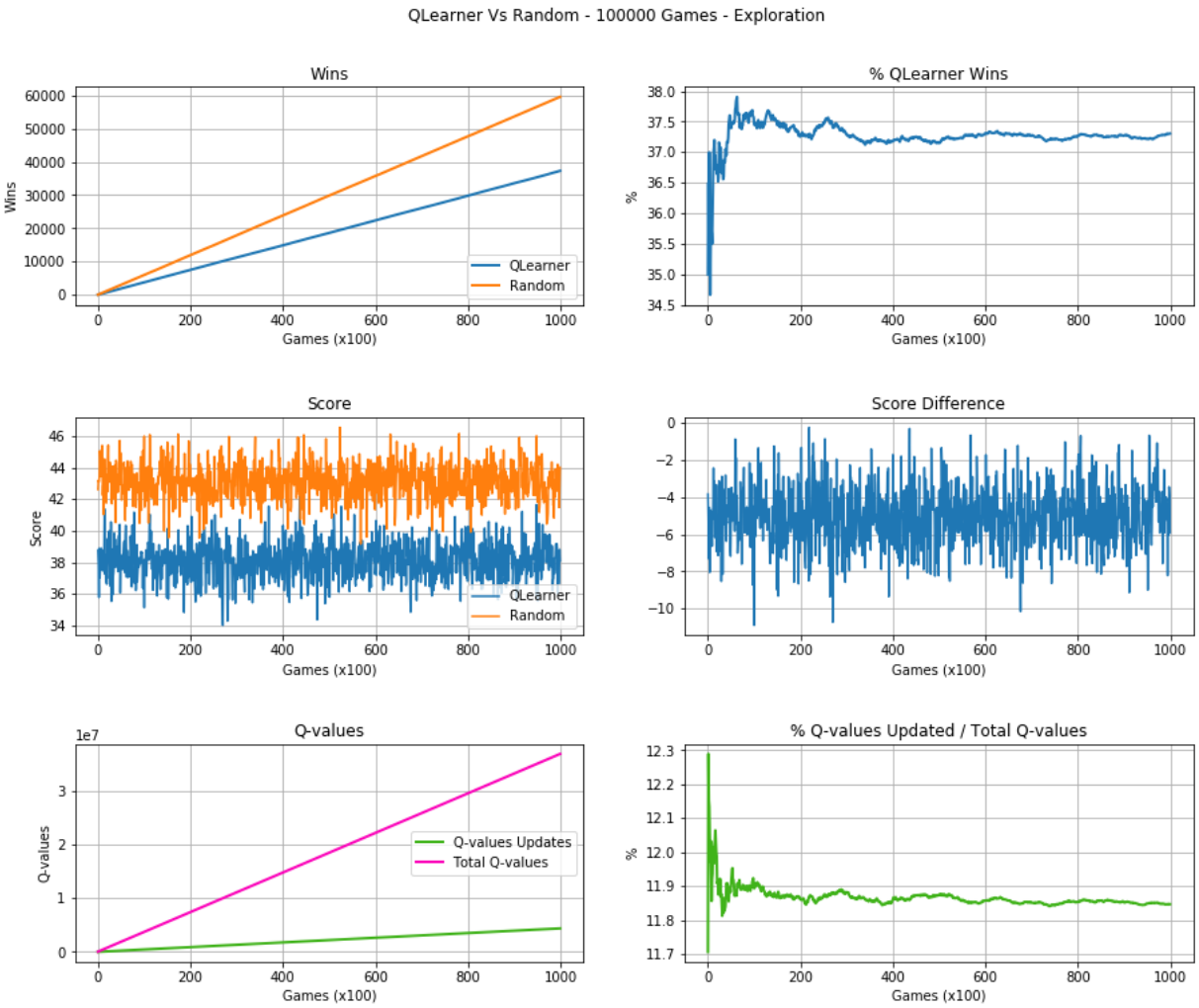


Figura 23. Métricas en el primer entrenamiento de QLearner vs. Random. *Fuente:* elaboración propia.

Tras el entrenamiento del agente QLearner, las métricas de las 50.000 partidas en modo explotación contra Random pueden verse en la Figura 24.

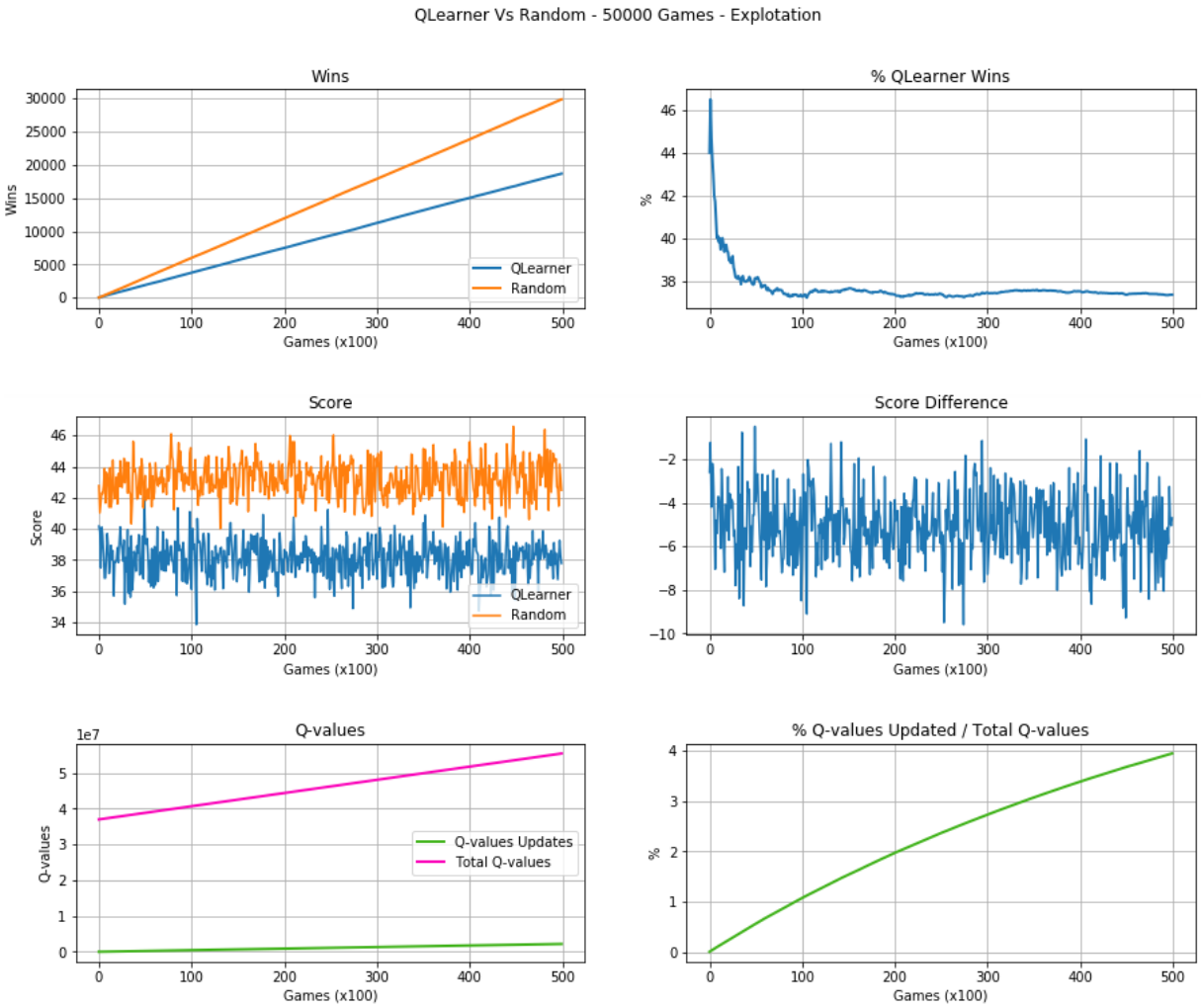


Figura 24. Métricas de QLearner vs. Random tras el primer entrenamiento.

Fuente: elaboración propia.

Y en la Figura 25 se muestran las métricas de las 50.000 partidas en modo explotación contra Greedy.

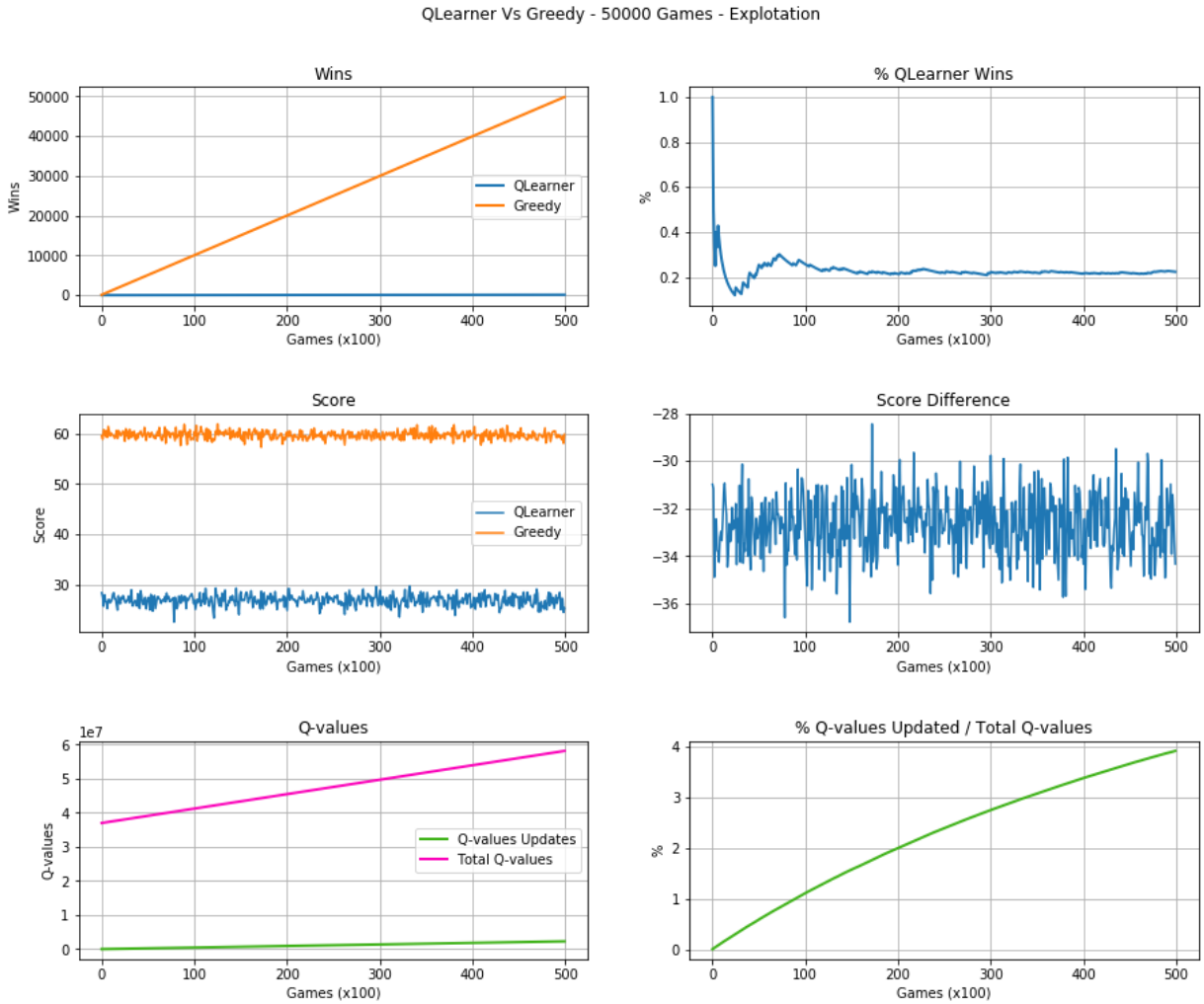


Figura 25. Métricas de QLearner vs. Greedy tras el primer entrenamiento.

Fuente: elaboración propia.

Sub-experimento 2

Se reparten siempre las mismas fichas al inicio de la partida.

Se entrena de nuevo el agente QLearner con otras 100.000 partidas en modo exploración contra Random. Se presentan los resultados en la Figura 26.

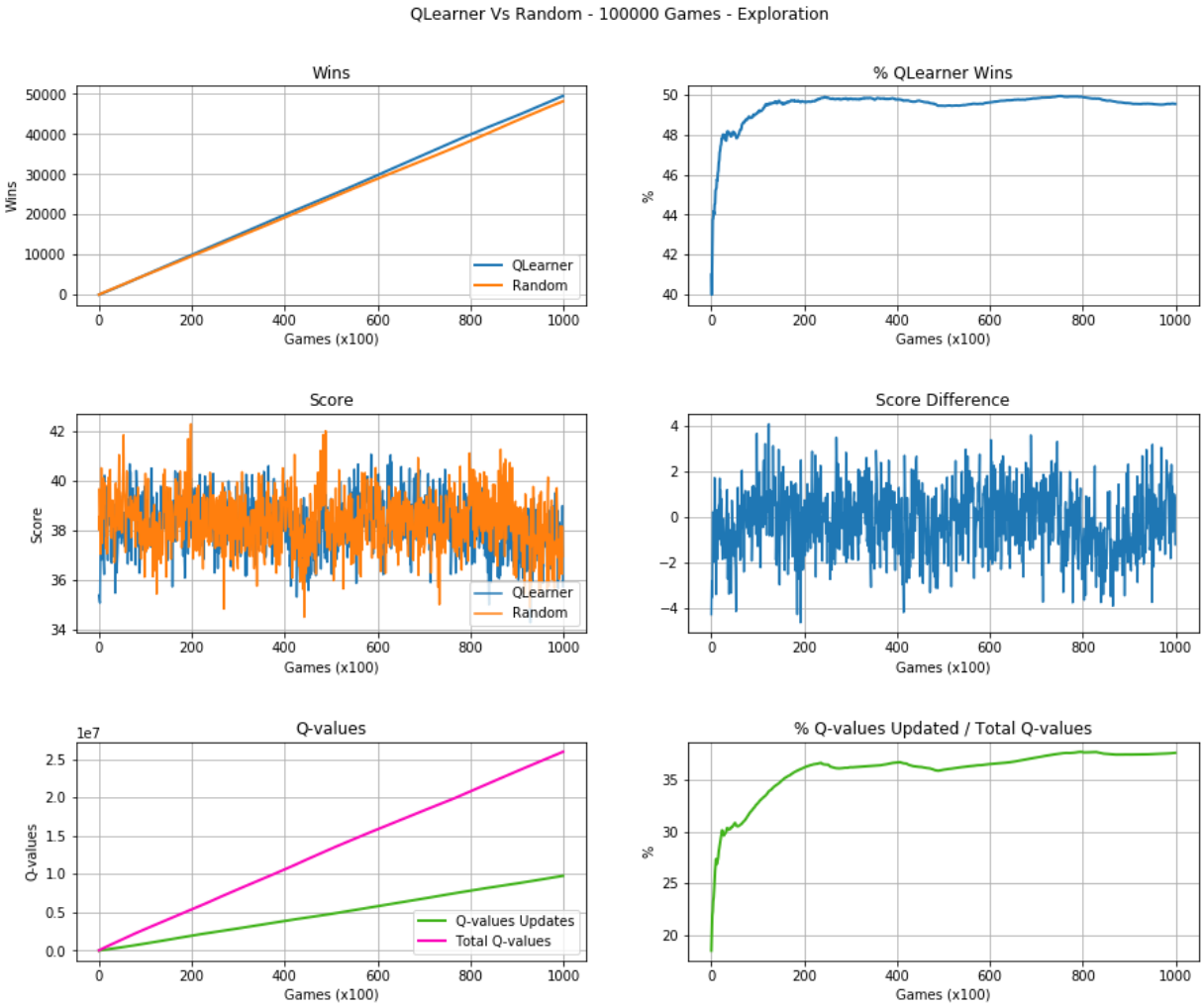


Figura 26. Métricas en el segundo entrenamiento de QLearner vs. Random.

Fuente: elaboración propia.

Una vez completado este entrenamiento de QLearner, las 50.000 partidas en modo explotación contra Random generan las métricas de la Figura 27.

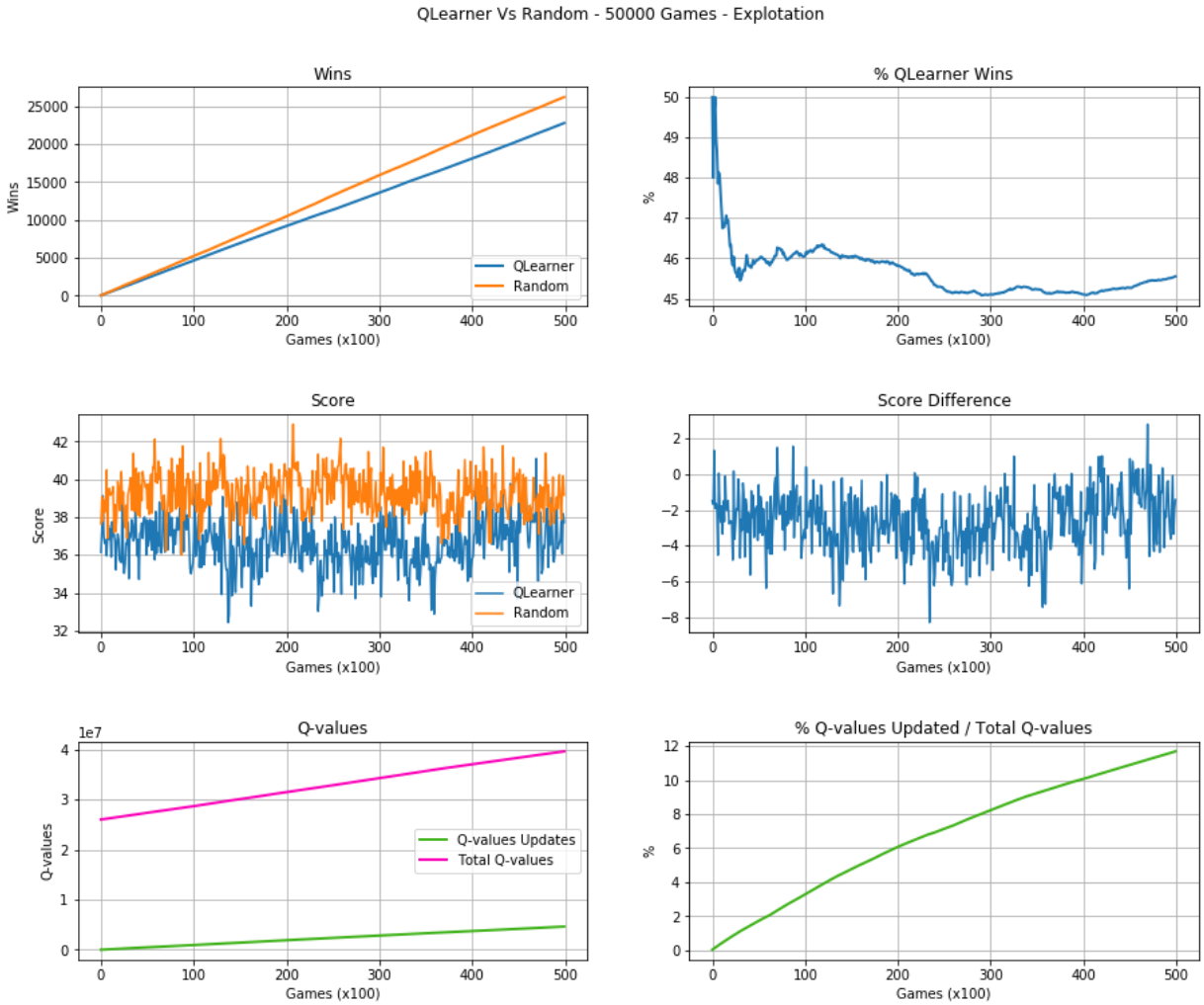


Figura 27. Métricas de QLearner vs. Random tras el segundo entrenamiento.

Fuente: elaboración propia.

Finalmente, en la Figura 28 se pueden ver las métricas de las 50.000 partidas en modo explotación contra Greedy.

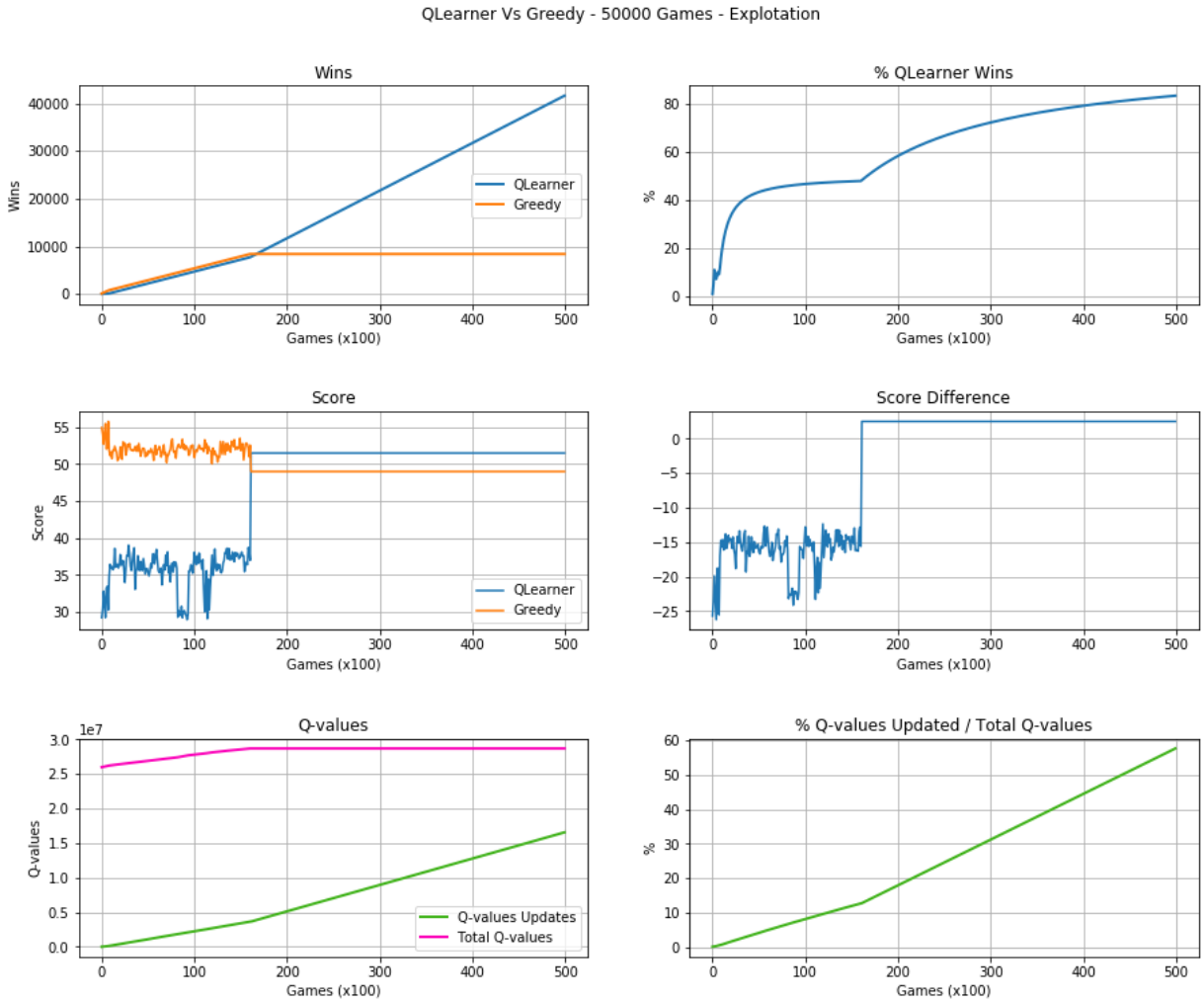


Figura 28. Métricas de QLearner vs. Greedy tras el segundo entrenamiento.
Fuente: elaboración propia.

4.3. Discusión

A continuación se presenta una valoración de los resultados obtenidos en el apartado anterior para las dos líneas de investigación.

Como punto de partida de ambas, cabe valorar los enfrentamientos entre los agentes Greedy y Random. Como cabía esperar, a tenor de los resultados presentados en la Tabla 1, se puede confirmar que:

El agente Greedy es ampliamente superior al Random.

Discusión sobre el experimento 1 (agente Minimax)

Los resultados mostrados en la Tabla 2 permiten elaborar una serie conclusiones generales:

- Por una parte, el número de turnos que se ejecutan de media es irrelevante, ya que en todos los casos es similar e independiente a los parámetros especificados.
- Además se observa que cualquiera que sea el valor del hiperparámetro *depth*, el agente Minimax siempre gana al Random y lo hace con una abultada diferencia de puntos. Se puede concluir que el Random está muy por debajo del nivel de Minimax.
- Como era de esperar, las partidas de Minimax con *depth = 1* contra Greedy quedan muy equilibradas. Se aprecia que el número de victorias de cada uno son muy similares (47 a 49) y que además solamente hay 0,7 puntos de diferencia de promedio. Queda confirmado de forma experimental algo que a nivel teórico era sabido:

La búsqueda Minimax que emplea un árbol de profundidad 1 es equivalente a la búsqueda voraz.

Cabe destacar la información vista en la Figura 21 acerca de la evolución de las partidas ganadas de Minimax frente Greedy, a medida que se aumenta la profundidad de búsqueda. Y es que con profundidades de 1, 2, 3 y 4, Minimax gana un 47%, 71%, 90% y 100% de partidas, respectivamente. La diferencia de puntos entre ambos crece de forma lineal, tal y como se puede apreciar en la Figura 20.

Como contrapartida, aumentar la profundidad conlleva incrementar exponencialmente el número de nodos generados en el árbol de búsqueda y, por ende, el tiempo que el agente tarda en decidir la acción que va a llevar a cabo. Esto se puede ver de forma gráfica en la Figura 20.

Se confirma que la complejidad del algoritmo Minimax crece de forma exponencial en tiempo y en espacio según aumenta la profundidad del árbol de búsqueda.

Con vistas a un enfrentamiento entre la mejor configuración de Minimax y un humano, *depth = 4* es un valor de profundidad que implica demasiado tiempo de espera en decidir la mejor jugada, a pesar de que el porcentaje de victorias contra Greedy es del 100%. Esa es la

razón por la cual se ha optado por ejecutar partidas de Minimax con distintos valores de *maxTime* solo contra Greedy con *depth = 3*.

El tiempo de decisión que se pretende mejorar es 50,27 segundos (Minimax vs. Greedy con profundidad 3), sin perder la efectividad de más de un 90% de partidas ganadas (9 victorias y dos empates).

Los valores promedio siguen una tendencia de crecimiento lineal a medida que se aumenta el parámetro *maxTime*, tal y como se ve en la Figura 22.

En la Tabla 2 vemos que cuando *maxTime = 90* segundos se alcanza el porcentaje de victorias de Minimax frente a Greedy y el tiempo que tarda el agente en decidir su jugada en cada turno es 44,43 segundos. Es decir, se reduce el tiempo un 11,62%. Si bien el experimento parece que ha mejorado el porcentaje de victorias, con un número mucho mayor de partidas se concluiría que con *maxTime = 90* segundos mejoraría el tiempo pero nunca llegaría a ser más efectivo en cuanto a porcentaje de victorias, ya que con tiempo ilimitado se explora el árbol por completo hasta el nivel de profundidad 3 en todas las ramas del nodo raíz.

Por tanto, parece que la configuración óptima del agente Minimax para jugar contra un humano, equilibrando la eficacia con el tiempo, es *depth = 3* y *maxTime = 90* segundos.

La Tabla 3 presenta el resultado de las partidas entre los dos agentes Minimax con profundidad 3, uno con tiempo máximo de exploración de 90 segundos y otro ilimitado. Ratifica en cierto modo los resultados que se habían obtenido en la Tabla 2:

- Están muy igualados en cuanto a efectividad: 10 a 8, con 2 empates y 2,50 puntos de diferencia de promedio.
- La duración media de las partidas es 22,19 minutos, que está próxima a la suma del valor de Minimax vs. Greedy (11,65 minutos) y la que emplea *maxTime = 90* segundos (9,85 minutos).

Se concluye, por tanto, que:

La mejor configuración de un agente Minimax con poda Alfa-Beta, para jugar al iwoki contra un humano, es con un nivel de profundidad 3 y con un tiempo máximo de búsqueda de 90 segundos sobre cada una de las ramas principales del nodo raíz.

Al ponerlo a prueba contra el autor del juego a lo largo de 5 partidas, resulta relevante que Minimax gana en todos los casos. En la Tabla 4 muestra la diferencia de puntos que obtienen. Es posible hacer un seguimiento de una de las partidas viendo el Anexo IV. Si se presta atención a los movimientos de ambos contrincantes vemos cómo el agente Minimax lleva la iniciativa de la partida. Minimax evita a toda costa que el humano coloque fichas hexagonales, manteniendo mayor puntuación que el humano en todo momento.

El agente Minimax se anticipa al humano y desbarata cualquier estrategia que éste intenta poner en práctica. Gana en todas las partidas disputadas.

Discusión sobre el experimento 2 (agente QLearner)

Según los resultados de la Tabla 5, y siguiendo las métricas ilustradas en las figuras del apartado anterior, es posible sacar una serie de conclusiones.

- En primer lugar, el promedio de turnos y la duración media de las partidas carece de interés, puesto que sus valores son muy similares en cualquiera de las tipologías de las partidas y para todos los agentes.
- Las 10.000 partidas iniciales que realiza el agente QLearner contra el Random no presentan información relevante. Se observa que el porcentaje de partidas ganadas por el primero es en torno a un 37% del total, con una media de diferencia de puntos que no es para nada abultada (5,11).
- Sin embargo, las otras instancias ejecutadas contra el agente Greedy ponen de manifiesto la poca efectividad del QLearner cuando aún no ha sido entrenado, pues solo consigue ganar 24 de las 10.000 partidas (un 0,24%) y la diferencia media de puntos es de 32,68.

En cuanto al primero de los sub-experimentos, el entrenamiento de QLearner es un ejercicio similar a las primeras 10.000 partidas pero con un volumen mayor. Se ve en la Figura 23 que la tendencia en cuanto a victorias y diferencia de puntos es la misma. Sin embargo, podemos encontrar relevancia en la cantidad de *Q-values* generados (cerca de 37 millones), siendo solo un 11,85% los actualizados a lo largo de las 100.000 partidas.

Una vez entrenado con instancias de exploración, el comportamiento del agente QLearner en modo explotación mantiene la tendencia:

- Las partidas contra el agente Random siguen la misma línea que en el entrenamiento, sin cambios de relevancia. Esto se puede ver en la Figura 24.
- Con respecto a los enfrentamientos contra Greedy, no presenta mejoría con relación a las 10.000 partidas iniciales, pues solamente consigue ganar 112 de las 100.000, (un 0,22%) y se mantiene la diferencia de puntos. La Figura 25 confirma de forma gráfica este dato.

El aprendizaje es de casi 37 millones de *Q-values*, lo que supone un *Q-file* de 4,54 GB, más alrededor de un 50% adicional de *Q-values* en la fase de explotación. A pesar de esto no se encuentran indicios de mejoría ni en los enfrentamientos contra Random ni contra Greedy.

Durante el experimento se ha probado a aumentar el número de partidas de entrenamiento, pero el consumo de recursos a nivel computacional hace inviable obtener buenos resultados. Por tanto se concluye que:

Es computacionalmente intratable entrenar un agente, implementado con Reinforcement Learning en modo iterativo, para que aprenda las políticas que le permitan ganar al iwoki.

Para poder llegar a unas conclusiones interesantes en este trabajo, se pone en práctica la modificación del iwoki anteriormente mencionada, que consiste en repartir siempre las mismas fichas al inicio de las partidas. Con esta medida se reduce considerablemente el espacio de posibles estados.

El segundo sub-experimento adopta esta modificación y a partir de ahora los resultados toman otro cariz.

Vemos en la Tabla 5 que a lo largo de las 100.000 partidas de entrenamiento existe una diferencia ínfima entre el entre QLearner y Random. Prácticamente el 50% de victorias y una diferencia media de puntos que puede considerarse nula (0,20). Se generan 25.965.300 *Q-values*, un 29,76% menos que en el primer experimento, y el porcentaje de *Q-values* actualizados se mantiene en torno al 39% con respecto a los totales. La evolución a nivel gráfico se aprecia en la Figura 26.

Tras dicho entrenamiento, se lanzan 50.000 instancias de explotación contra los dos contrincantes, dando lugar a las siguientes consideraciones:

- Las partidas contra el agente Random encuentran poca diferencia en sus valores de referencia. Por tanto, el cambio de exploración a explotación no supone alteración

significativa en la evolución de dichos valores. Las gráficas de la Figura 27 confirman esta tendencia.

- Al jugar contra Greedy el resultado es diferente. Las gráficas de la Figura 28 muestran que el rendimiento del agente QLearner ha mejorado hasta el punto de que a partir de las 16.200 partidas, gana siempre. Le toma la medida a Greedy, que en cada turno siempre elige la mejor jugada a corto plazo. Aprende qué política debe seguir para obtener la mayor cantidad acumulativa de puntos.

  Queda, por tanto, demostrado que:

En una versión del iwoki en la que se reparten siempre las mismas fichas al inicio de la partida, un agente QLearner, implementado con Reinforcement Learning iterativo, puede ser entrenado con 100.000 partidas exploratorias contra un agente Random, más otras 16.200 en modo explotación contra otro Greedy , para ganar a éste último.

5. Conclusiones y trabajo futuro

A lo largo del presente trabajo se ha dado a conocer el iwoki y sus características, lo cual ha permitido saber el modo en que se ha llevado a cabo la modelización e implementación de sus componentes.

Basado en el estudio del estado del arte, se han definido qué agentes serían los que iban a formar parte de los experimentos, según la naturaleza del iwoki, y se ha procedido a su desarrollo.

Durante la fase experimental se han ejecutado una serie de partidas que han puesto en liza a los distintos agentes. Se ha determinado cómo de bueno resulta el agente Minimax con respecto a Random y Greedy. También hemos podido observar la capacidad de aprendizaje que tiene el agente QLearner al entrenarlo con Random y después demostrar su conocimiento adquirido contra Greedy.

Finalmente hemos podido comprobar cómo el agente Minimax ha sido capaz de ganar al creador del iwoki, que a su vez es también el creador del algoritmo ganador.

Los resultados obtenidos cubren con la doble finalidad planteada al inicio del trabajo:

- Dar con la implementación apropiada para los agentes de las dos líneas de investigación desarrolladas, Minimax y Reinforcement Learning, y dotarles de la configuración óptima conforme a los resultados experimentales.
- Presentar el iwoki como una aportación novedosa al campo de la investigación en inteligencia artificial aplicada a juegos, estableciendo un punto de partida a partir del cual se desarrollen diversidad de agentes con distintas tecnologías.

Consecuentemente, se identifican algunas líneas de trabajo futuro a raíz de lo presentado en este trabajo:

- Existe la posibilidad de implementar las distintas **alternativas a la poda Alfa-Beta**, citadas en el capítulo de Estado del arte **(Megamax, SSS*, Scout, etc.)**.
- Dado que el iwoki original es de información imperfecta, pues la fichas pequeñas no son visibles para el resto de los jugadores, se podría desarrollar como un **juego bayesiano**. Así podrá tener en cuenta las probabilidades de que un adversario disponga de determinadas fichas, conociendo el número de veces que han salido previamente (es preciso recordar que existen cuatro ejemplares de cada ficha pequeña).

- Al ser también un juego estocástico y con un alto grado de ramificación, sería apropiado desarrollar método **Monte Carlo Tree Search**.

- El sistema de Reinforcement Learning desarrollado en este trabajo pone de manifiesto que, para la versión no modificada del iwoki, el sistema es demasiado complejo a nivel computacional. Para codificar un agente utilizando **Deep Reinforcement Learning**, el input de la red neuronal sería el estado actual (turno del juego) y el output sería un *Q-value* por cada posible acción en ese estado. El problema que surge es que hay un número distinto de acciones posibles en cada estado (existen millones de acciones distintas si tenemos en cuenta todos los estados), por lo que el número de neuronas de salida sería indeterminado. Se abre la puerta a una nueva línea de investigación para estudiar este caso concreto y modelar una red neuronal que permita tratar el problema del iwoki.

- Estudiar si es factible la implementación de una **Recurrent Neural Network** (RNN) para un nuevo agente **Vanilla**. La RNN sería similar a la que se emplea para modelos del lenguaje cuando obtiene una secuencia de palabras y devuelve una distribución de probabilidad de la palabra siguiente. El *hidden state* incluye el texto visto hasta el momento en cada *time step*. En un modelo generativo, la palabra *output* de cada *time step* se convierte en el *input* del siguiente.

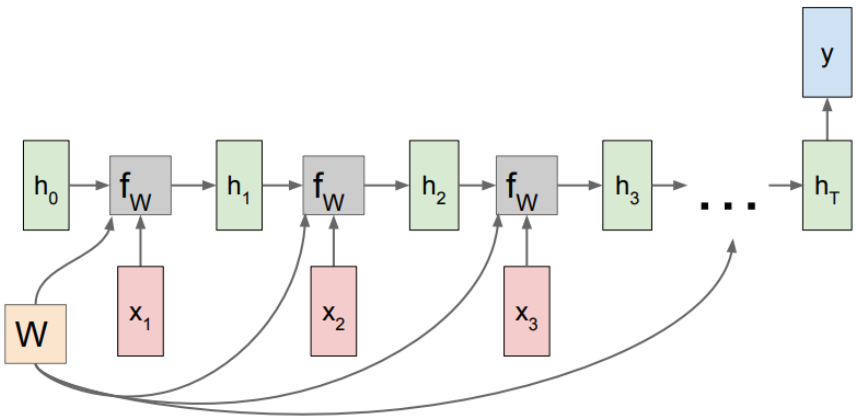


Figura 29. Grafo de computación de una RNN con un único output.
Fuente: *(Li, Johnson, & Yeung, 2018)*.

La misma representación de la Figura 29 podría ser válida para nuestro caso, pues existen las siguientes analogías:

  - La entrada a la red sería el conjunto de posibles acciones a realizar en un estado concreto.
  - La *hidden layer* se nutre con los estados que va adquiriendo el juego en cada *time step*, es decir, cada vez que el jugador realiza una acción concreta.
  - La salida sería una distribución de probabilidad de la siguiente acción a realizar, mediante una función *softmax*.
  - Esta acción sería la el *input* del siguiente *time step* del modelo.

  El agente Vanilla podía ser sustituido por **LSTM** o **GRU** en el caso de que se produjeran los problemas típicos de las RNN simples, como es el caso de *vanishing gradients* o *exploding gradients*.

- Por supuesto, sería interesante establecer un **entorno gráfico** que facilite a un jugador humano la ejecución de sus movimientos sin necesidad de reproducir las jugadas con la versión física del juego. Su aplicación sería extensible a dispositivos móviles.
- Como consecuencia, habilitar el juego también para jugadores humanos exclusivamente, mediante juego online. Se podría ampliar **hasta 6 jugadores**, tal y como es el iwoki original.

6. Bibliografía

Aguilar, P. (15 de junio de 2008). *IA - Algoritmos de Juegos.* Recuperado de https://www.cs.upc.edu/~bejar/ia/material/trabajos/Algoritmos_Juegos.pdf

Andrew, N. (2017). Reinforcement Learning and Control. *CS229 Lecture notes,* Part XIII.

Bai, A., Wu, F., & Chen, X. (julio de 2015). Online Planning for Large Markov Decision Processes with Hierarchical Decomposition. *ACM Transactions on Intelligent Systems and Technology*, 6(45). doi: 10.1145/2717316

Bonilla, J. (15 de marzo de 2019). Iwoki Maths: el cálculo matemático en su forma más cuidada. *Consola y Tablero*. Recuperado de https://consolaytablero.com/2019/03/15/iwoki-calculo-matematico/

David-Tabibi, O., & Netanyahu, N. (septiembre de 2002). Verified Null-Move Pruning. *ICGA Journal, International Computer Games Association,* 25(3), 153-161. https://arxiv.org/abs/0808.1125v1

Edwards, D., & Hart, T. (28 de octubre de 1963). The Alfa-Beta Heuristic. *Artificial Intelligence Project - RLE and MIT Computation Center, Memo 30*. doi: 1721.1/6098

Harsanyi, J. (1 de diciembre de 2004). Games with Incomplete Information Played by "Bayesian" Players, I–III Part I. The Basic Model. *InformsPubsOnLine. Management Science,* 50(12). doi: 10.1287/mnsc.1040.0270

Heidecke, J. (13 de febrero de 2018). Inverse Reinforcement Learning pt. I. *thinking wires*. Recuperado de https://thinkingwires.com/posts/2018-02-13-irl-tutorial-1.html

Kansal, S., & Martin, B. (s.f.). Reinforcement Q-Learning from Scratch in Python with OpenAI Gym. *Learndatasci, Ed.* Recuperado de https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/

Li, F.-F., Johnson, J., & Yeung, S. (3 de mayo de 2018). Lecture 10: Recurrent Neural Networks. *Stanford CS231N.* Recuperado de http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

Mimbang, J. (11 de abril de 2016). *La teoría de juegos: El arte del pensamiento estratégico*. 50minutos.es - Economía y Empresa.

Pearl, J. (1980). Scout: A Simple Game-Searching Algorithm With Proven Optimal Properties. *Association for the Advancement of Artificial Intelligence*. Los Angeles, California.

Plaat, A., Schaeffer, J., Pijls, W., & de Bruin, A. (Diciembre de 1994). α-β + TT. *Departament of Computing Science. The University of Alberta.* Edmonton, Alberta.

Restrepo, C. (diciembre de 2009). Aproximación a la teoría de juegos. *Revista Ciencias Estratégicas*, 17(22), 157-175. Medellín, Colombia. Recuperado de http://www.redalyc.org/pdf/1513/151313682002.pdf

Russell, S., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach (Third Edition)* (pp. 161-189). Penitence Hall Series in Artificial Intelligence.

Silver, D. (13 de mayo de 2015). *Reinforcement Learning Course by David Silver- Lecture 1: Introduction to Reinforcement Learning* [Video podcast]. Recuperado de https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PLzuuYNsE1EZAXYR4FJ75jcJseBmo4KQ9-&index=2&t=0s

Silver, D. (13 de mayo de 2015). *Reinforcement Learning Course by David Silver- Lecture 2: Markov Decision Process* [Video podcast]. Recuperado de https://www.youtube.com/watch?v=lfHX2hHRMVQ&list=PLzuuYNsE1EZAXYR4FJ75jcJseBmo4KQ9-&index=3&t=0s

Silver, D. (13 de mayo de 2015). *RL Course by David Silver - Lecture 9: Exploration and Exploitation* [Video podcast]. Recuperado de https://www.youtube.com/watch?v=sGuiWX07sKw

Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., van den Driessche, G., . . . Hassabis, D. (27 de enero de 2016). Mastering the game of Go with deep neural networks and tree search. *Nature,* 529, 484–489. doi: 10.1038/nature16961

Von Newman, J., & Morgenstern, O. (1944). *Theory of games and economic behavior* (pp. 670-674). Princeton University Press.

Wright, R. (2006). *Progess is not a zero-sum game*. [Video podcast]. Recuperado de https://www.ted.com/talks/robert_wright_progress_is_not_a_zero_sum_game#t-51770

Yannakakis, G., & Togelius, J. (11 de septiembre de 2018). *Artificial Intelligence and Games* (pp. 42-44, 71-76, 115-118). Springer. doi: 10.1007/s10710-018-9337-0

Zagal, J., Rick, J., & Hsi, I. (marzo de 2016). Collaborative games: Lessons learned from board games, 37(1), 24-40. *Simulation & Gaming*. doi: 10.1177/1046878105282279

7. Anexos

Anexo I Instrucciones del iwoki

En esta sección se especifican las instrucciones del iwoki adaptadas para solamente dos jugadores:

Resumen del juego

El *iwoki maths* es juego para combinar el cálculo de operaciones matemáticas simples con la percepción espacial de objetos bidimensionales.

Consiste en ir acoplando en cada turno las fichas pequeñas con las hexagonales, y viceversa, haciendo coincidir el resultado de las sumas o restas con los números de las fichas hexagonales.

A medida que el juego avanza irán aumentando las opciones para elegir la mejor estrategia.

El ganador será quien consiga sumar más puntos tras del recuento final.

Contenido

Fichas pequeñas: En cada una de las tres esquinas exteriores hay un número del 1 al 4 y en el centro un signo de suma (+) o resta (–). Cada cara presenta los mismos números, pero con el signo contrario.

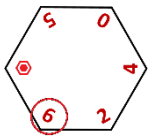
Fichas hexagonales: En los vértices hay un número del 1 al 6 o un comodín, que viene representado por  . Son los mismos valores por ambas caras, por lo que el orden presentado en una cara será el inverso en la otra.

Antes de empezar

- ⇨ Se introducen las fichas pequeñas un una bolsa y las hexagonales en la otra.
- ⇨ Cada jugador escoge al azar 9 fichas pequeñas y 3 hexagonales.
- ⇨ Se colocan las fichas pequeñas en el atril para no ser vistas por el oponente. Las hexagonales permanecerán visibles en todo momento.
- ⇨ Uno de los jugadores se encargará de rellenar una hoja de tanteo, anotando los puntos de ambos en cada jugada y haciendo el recuento final. Tendrá que poner en ella los nombres de los jugadores y rellenarla como se indica en el ejemplo de la propia hoja de tanteo.

Desarrollo de la partida

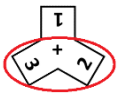
- ⇨ Se echa a suertes quién empieza la partida.
- ⇨ El jugador que empieza pone en la mesa de juego una de sus fichas.
  - Si elige empezar con una hexagonal, se anotará en su marcador el número más alto que contenga la ficha.



Por ejemplo, iniciando el juego con esta ficha, el jugador se anotará 6 puntos

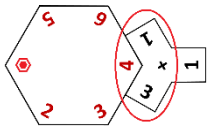
  - Si, por el contrario, elige empezar con una ficha pequeña, se anotará en su marcador el resultado de la suma o resta que más le convenga.

Por ejemplo, si la ficha con la que inicia la partida es  , lo más conveniente sería anotarse:

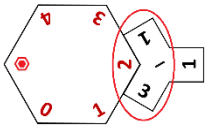


3 + 2 = 5 puntos , o bien  *3 – 1 = 2 puntos*

- ⇨ A continuación, de forma alternativa, cada jugador intentará colocar una ficha de cualquiera de los dos tipos, teniendo en cuenta lo siguiente:
  - El resultado de las sumas o restas de las fichas pequeñas debe coincidir con los números rojos de las hexagonales adyacentes.

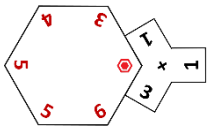


Ejemplo de suma: 3 + 1 = 4

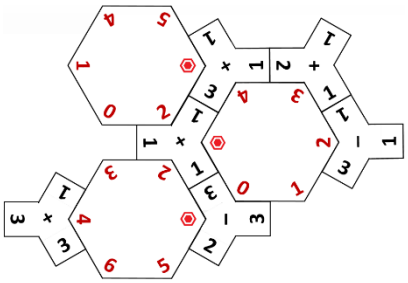


Ejemplo de resta: 3 – 1 = 2

- El comodín es válido como resultado de cualquier suma o resta, pero solo se podrá acoplar cuando la ficha pequeña cubra también algún número rojo. Mira estos ejemplos:



Colocación incorrecta. La ficha pequeña no se acopla sobre ningún número, solo sobre el comodín



Colocación correcta. En todos los casos, las fichas pequeñas que cubren los comodines también cubren algún número rojo de otra ficha hexagonal

⇨ ¿Cuándo se roban fichas pequeñas?:

- Cuando no sea posible poner fichas de ninguno de los dos tipos, se robará una ficha pequeña y se intentará colocar. Si no se puede se pasará el turno al otro jugador. Solo se puede robar una ficha pequeña en cada turno.
- En el caso de que no se pueda colocar ninguna ficha y no haya más para robar, se pasará el turno al otro jugador.
- También se podrá robar una ficha pequeña voluntariamente, aun teniendo la posibilidad de colocar alguna. Justo después de robar es obligatorio poner alguna de las fichas, pequeñas o hexagonales.

⇨ ¿Cuándo se roban fichas hexagonales?:

El jugador robará una ficha hexagonal cada vez que haya realizado tres veces cualquiera de las siguientes acciones:

- Colocar una ficha pequeña utilizando la cara de la resta en lugar de la suma.
- Robar una ficha pequeña, ya sea por no tener ninguna opción para colocar o porque lo haga voluntariamente.
- Pasar de turno al otro jugador ante el caso de no poder colocar ninguna ficha.

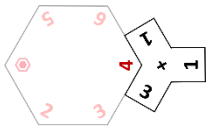
Las dos primeras veces que el jugador realice alguna de las acciones anteriores colocará un testigo blanco en la parte superior de su atril. De esta manera, tanto él

como su contrincante tendrán presente lo cerca que está de conseguir una nueva ficha hexagonal.

La tercera vez robará una ficha hexagonal y quitará los testigos blancos del atril para volver al estado inicial.

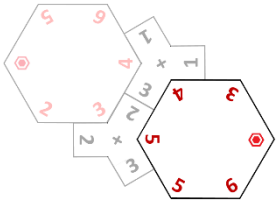
⇨ Cuando un jugador coloca alguna ficha, se rellena la hoja de tanteo sumando en su fila el número o números rojos que cubra. Mira estos ejemplos:

Caso 1. Si se coloca esta ficha pequeña:



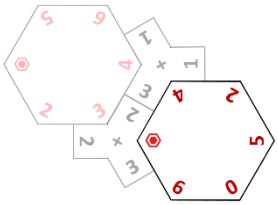
Se suman 4 puntos al marcador

Caso 2. Si se coloca la ficha hexagonal inferior:



Se suman 4 + 5 = 9 puntos al marcador

Hay que tener en cuenta que los comodines no suman puntos:



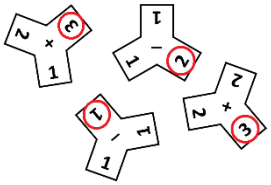
En este caso se suman 4 puntos al marcador

Fin de la partida

Cuando uno de los jugadores se quede sin fichas pequeñas, sin tener en cuenta las hexagonales, el otro jugará una última baza.

A continuación se realizará el recuento de puntos para ambos jugadores:

- La puntuación final de quien se haya quedado sin fichas pequeñas será la que haya acumulado en la hoja de tanteo hasta ese momento.
- A los puntos que haya acumulado el otro jugador hay que restarle el número más alto de cada una de las fichas que le queden. Fíjate en el ejemplo:



Si el jugador se queda con estas fichas tiene que restar 3 + 2 + 1 + 3 = 9 puntos a los que había acumulado en la partida

- No se deben tener en cuenta los puntos de las fichas hexagonales que no se hayan casado. Esos no cuentan.

El ganador de la partida será quien consiga más puntos después del recuento.

Hoja de tanteo

La hoja de tanteo sirve para llevar el control de los puntos que cada jugador va obteniendo en cada turno. Podemos ver el modo de rellenarla en la Figura 30.

Una vez finalizada la partida se realiza el recuento de puntos y se determina qué jugador es el ganador.

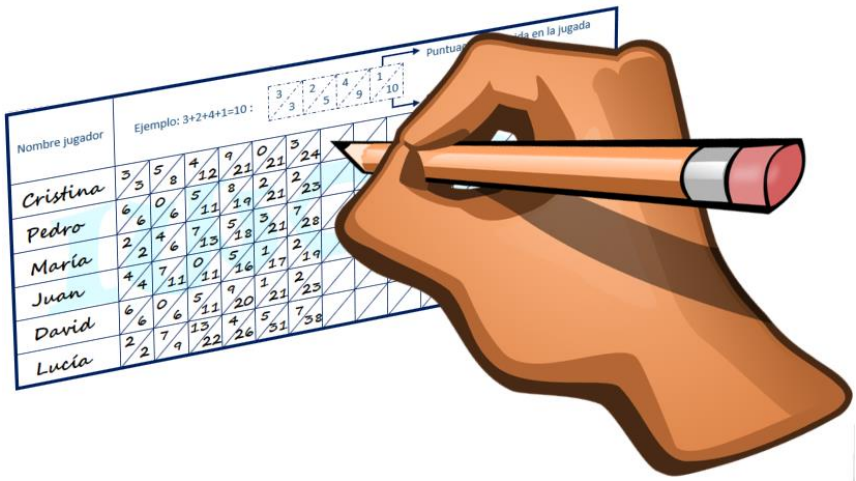


Figura 30. Ejemplo de cómo rellenar la hoja de tanteo, en su versión para 6 jugadores. *Fuente:* elaboración propia.

Anexo II Estrategias para jugar al iwoki

Puede ser de utilidad conocer algunas de las estrategias más importantes que un jugador humano va adquiriendo a media que juega a la versión física del iwoki. De esta forma se puede entender mejor las acciones que optan por realizar los agentes.

Al inicio de la partida

En contra de lo que pueda parecer, es buena opción iniciar la partida con una ficha pequeña utilizando la cara de la resta en lugar de la suma, ya que se le obliga al oponente a utilizar una ficha hexagonal para sumar muy pocos o incluso cero puntos. Además, con esta elección el jugador está más cerca de robar una ficha hexagonal.

En el desarrollo del juego

Por regla general es mejor colocar las fichas pequeñas utilizando la suma en lugar de la resta, con el fin de ir acumulando más puntos.

El jugador debe colocar cuanto antes las fichas pequeñas que tienen los números mayores. Así acumulará menos puntos para restar en el recuento final.

En igualdad de condiciones, el jugador debe tender a colocar una ficha pequeña que tenga todos los números iguales y quedarse con las que tiene números variados. Así le será más fácil casarla después, al poder acoplarla en más huecos disponibles.

En determinados casos es mejor robar voluntariamente o utilizar la resta en lugar de la suma en una ficha pequeña, con el fin de conseguir hacerse con una ficha hexagonal que, con suerte, permita al jugador sumar más puntos.

Cuando solo vaya a poder sumar pocos puntos el turno actual, el jugador debe aprovechar para realizar alguna acción que le proporcione un testigo blanco y acercarse así a conseguir una ficha hexagonal.

Debe ser consciente de sus puntos y de los de su oponente al valorar qué jugada le conviene hacer en cada turno:

- Colocar fichas pequeñas le acercan al final de la partida, pero por regla general esta acción le proporciona menos puntos.
- Colocar fichas hexagonales generalmente le permite obtener más puntos, pero no le acerca al final de la partida.

Cuando se acerca el final

El jugador debe saber que las fichas pequeñas sobrantes restan puntos en el recuento final. Si está próximo a quedarse sin fichas pequeñas y el oponente tiene más puntos que él, deberá intentar alargar la partida a base de robar fichas y colocar pequeñas con la resta en lugar de con la suma, para poder conseguir una ficha hexagonal que te pueda proporcionar los puntos necesarios.

Anexo III Módulos Python

pieces.py

```
import numpy as np
from utils import listToString

class HexagonalPiece:
    """
    Clase que representa las fichas hexagonales.
    En cada vértice hay un número del 1 al 6 o un comodín (representado por el carácter '*'). Son los mismos valores por ambas caras, por lo que el orden presentado en una cara será el inverso en la otra.
    Se tiene en cuenta todas las 12 posibles variantes de cada ficha hexagonal, según los órdenes que puede presentar (6 por cada una de las caras).

        Ejemplo:
            La representación de la ficha [2, 4, 3, 6, 5, *] tiene estas 12 posiciones distintas:

            [2, 4, 3, 6, 5, *]
            [4, 3, 6, 5, *, 2]
            [3, 6, 5, *, 2, 4]
            [6, 5, *, 2, 4, 3]
            [5, *, 2, 4, 3, 6]
            [*, 2, 4, 3, 6, 5]
            [*, 5, 6, 3, 4, 2]
            [2, *, 5, 6, 3, 4]
            [4, 2, *, 5, 6, 3]
            [3, 4, 2, *, 5, 6]
            [6, 3, 4, 2, *, 5]
            [5, 6, 3, 4, 2, *]

    """
    def __init__(self, name, numbers):
        self.id = name

        # Se tienen en cuenta los distintos órdenes de una ficha hexagonal por una cara:
        order1 = numbers[1:] + [numbers[0]]
        order2 = order1[1:] + [order1[0]]
        order3 = order2[1:] + [order2[0]]
        order4 = order3[1:] + [order3[0]]
        order5 = order4[1:] + [order4[0]]

        # Se tienen en cuenta también los diferentes órdenes de una ficha hexagonal por la otra cara:
        numbers_inv = numbers[::-1]
        order1_inv = order1[::-1]
        order2_inv = order2[::-1]
        order3_inv = order3[::-1]
        order4_inv = order4[::-1]
        order5_inv = order5[::-1]

        # self.orders contiene los distintos órdenes que pueden tomar los números de la ficha hegagonal, teniendo en cuenta las dos caras
        self.orders = np.array([listToString(numbers), listToString(order1), listToString(order2),
                                listToString(order3), listToString(order4), listToString(order5),
                                listToString(numbers_inv), listToString(order1_inv),  listToString(order2_inv),
                                listToString(order3_inv),  listToString(order4_inv),  listToString(order5_inv,)])


class SmallPiece:
    """
    Clase que representa las fichas pequeñas.
    En cada una de las 3 esquinas interiores hay un número del 1 al 4. Cada cara de la ficha presenta los 3 mismos números y con el mismo orden. Por una cara viene representada la suma (+) y por la otra su resta (-).
    Se tiene en cuenta todas las 6 posibles variantes de cada ficha pequeña, según los órdenes que puede presentar (3 por cada una de las caras).

        Ejemplo:
            La representación de la ficha [1, 2, 4] tiene estas 6 posibilidades distintas:

            [1, 2, 4, +]
            [2, 4, 1, +]
            [4, 1, 2, +]
            [1, 4, 2, -]
            [2, 1, 4, -]
            [4, 2, 1, -]

    """
    def __init__(self, name, numbers):
        self.id = name
```

```
        order1 = [numbers[0], numbers[1], numbers[2], '+']
        order2 = [numbers[1], numbers[2], numbers[0], '+']
        order3 = [numbers[2], numbers[0], numbers[1], '+']
        order4 = [numbers[0], numbers[2], numbers[1], '-']
        order5 = [numbers[1], numbers[0], numbers[2], '-']
        order6 = [numbers[2], numbers[1], numbers[0], '-']

        # self.orders contiene las posibles variantes que pueden tomar los números de la ficha pequeña, con los signos '+' y '-' de cada una
de las caras
        self.orders = np.array([listToString(order1), listToString(order2), listToString(order3),
                                listToString(order4), listToString(order5), listToString(order6)])
```

gaps.py

```
import numpy as np

class SmallGap:
    """
    Clase que representa un hueco del tablero de juego reservado para una ficha pequeña.
    """
    def __init__(self, vertices):
        self.vertices = vertices # lista en la que cada elemento corresponde con un vértice interior del hueco de la ficha
        idAux = ''
        for vertix in vertices:
            idAux = idAux + vertix[0] if idAux == '' else idAux + '_' + vertix[0]
        self.id = idAux


class HexagonalGap:
    """ Clase que representa un hueco del tablero de juego reservado para una ficha hexagonal.
        El identificador del hueco y el número de orden del vértice definen las coordenadas comunes del vértice que une a una ficha
hexagonal y otra pequeña.
    """
    def __init__(self, idHexagonal):
        self.id = idHexagonal
        # 'vertices' es una lista en la que cada elemento corresponde con un vértice del hexágono e incluye lo siguiente:
        # - Identificador del hueco hexagonal.
        # - Número de orden del vértice (del 1 al 6).
        # - Número asignado a ese vértice cuando se coloca en ese hueco una ficha hexagonal. '_' indica que el vértice aún no tiene valor
asignado.
        # - 'V' --> Vacant; 'O' --> Occupied.
        self.vertices = np.array([[idHexagonal, 1, '_', 'V'], [idHexagonal, 2, '_', 'V'], [idHexagonal, 3, '_', 'V'],
                        [idHexagonal, 4, '_', 'V'], [idHexagonal, 5, '_', 'V'], [idHexagonal, 6, '_', 'V']])
```

player.py

```
import numpy as np
from collections import namedtuple

class Player:
    """
    Clase que representa al agente.
    """
    def __init__(self, name):
        self.name = name # Nombre del jugador (agente)
        self.isTurn = False # Indica si es el turno del agente
        self.smallPieces = [] # Lista de fichas pequeñas que tiene el agente
        self.hexagonalPieces = [] # Lista de fichas hexagonales que tiene el agente
        self.score = 0 # Puntuación
        self.redIndicator = False # Indicador o testigo rojo, que se activa para indicar al oponentede que le queda una única ficha pequeña.
        self.whiteIndicator = 0 # Indicador o testigo blanco, que sirve de contador del número de acciones realizadas para conseguir una
nueva ficha hexagonal.
        self.gotPieceBefore = False # Indicador utilizado para impedir al agente robar una ficha pequeña dos veces en un mismo turno.
        self.numWins = 0 # Dato para la obtención de métricas
        self.numTurns = 0 # Dato para la obtención de métricas
        self.accumulatedScore = 0 # Dato para la obtención de métricas
```

```
        self.PublicState = namedtuple('PublicState',[ # namedtuple representa los atributos públicos del agente
            'hexagonalPieces', # tupla ordenada de strings ('hp1', 'hp2', ....)
            'numSmallPieces', # Integer
            'redIndicator', # Boolean
            'whiteIndicator', # Integer
        ])

        self.PrivateState = namedtuple('PrivateState',[ # namedtuple representa los atributos privados del agente
            'smallPieces',  # tupla ordenada de strings ('sp1', 'sp2', ....)
            'publicState' # namedtuple representa los atributos públicos del agente
        ])


    def getMove(self, gameState, *args, **kwargs):

        return None


    def gameOver(self, score, *args, **kwargs):

        return None


    def getPublicState(self):
        """
        Función que aglutina los atributos públicos del agente.
        RETURN:
            namedtuple PublicState
        """
        return self.PublicState(
            hexagonalPieces = tuple(sorted([
                piece.id
                for piece in self.hexagonalPieces
            ])), # tupla ordenada de strings ('hp1', 'hp2', ....)
            numSmallPieces = len(self.smallPieces), # Integer
            redIndicator = self.redIndicator, # Boolean
            whiteIndicator = self.whiteIndicator, # Integer
        )


    def getPrivateState(self):
        """
        Función que aglutina los atributos privados del agente.
        RETURN:
            namedtuple PrivateState
        """
        return self.PrivateState(
            smallPieces = tuple(sorted([
                'sp'+str(int(piece.id[2:])%12) # Para simplificar, se identifica la ficha con alguna de las 12 diferentes que existen
                for piece in self.smallPieces
            ])), # tupla ordenada de strings ('hp1', 'hp2', ....)
            publicState = self.getPublicState()
        )
```

gameState.py

```
import numpy as np
import player
from collections import namedtuple
from datetime import datetime

class GameState:
    """
    Clase que representa el estado de juego al completo.
    """
    def __init__(self, players, hexagonalPiecesAvailable, smallPiecesAvailable, hexagonalGaps, smallGaps,
                 fittableHexagonalGaps, fittableSmallGaps):
        self.players = np.array(players) # lista de jugadores (agentes)
        self.hexagonalPiecesAvailable = hexagonalPiecesAvailable # fichas hexagonales disponibles
        self.smallPiecesAvailable = smallPiecesAvailable # fichas pequeñas disponibles
        self.hexagonalGaps = hexagonalGaps # huecos hexagonales del espacio de juego
        self.smallGaps = smallGaps # huecos pequeños del espacio de juego
        self.fittableHexagonalGaps = fittableHexagonalGaps # huecos hexagonales en los que se puede colocar alguna ficha hexagonal
        self.fittableSmallGaps = fittableSmallGaps # huecos pequeños en los que se puede colocar alguna ficha pequeña
        self.initialAction = True # indica si es el primer movimiento de la partida
        self.turnPassed = 0 # número de turnos consecutivos que los jugdores han pasado. Evita un blucle infinito cuando ninguno de los
jugadores puede hacer otra acción que no sea pasar el turno
        self.gameOver = False # indicador de fin de la partida
        self.endProperly = False # indicador del modo de finalización de la partida
```

```
    self.State = namedtuple('State',[ # namedtuple representa el estado del mundo para un jugador
        # Tablero
        'fittableHexagonalGaps', # tupla ordenada de strings ('hg1', 'hg2', ....)
        'fittableSmallGaps', # tupla ordenada de strings ('hg1', 'hg2', ....)
        # Informnación pública de los jugadores
        'opponentsInfo', # Tupla del los estados de los oponentes
        # Informnación privada del jugador
        'playerInfo'#, # Tupla del los estados del jugador
    ])


def __hash__(self):
    return hash(self.State)


def __eq__(self, other):
    return other.State == self.State


def getAllInitialPieces(self, numSmall=9, numHexagonal=3):
    """
    Función que reparte al inicio de la partida todas las fichas pequeñas y hexagonales a los jugadores.
    PARAMETERS:
        numSmall: número de fichas pequeñas que se reparten.
        numHexagonal: número de fichas hexagonales que se reparten.
    """
    print("Se reparten {} fichas pequeñas y {} fichas hexagonales a cada jugador.".format(numSmall, numHexagonal))
    for i in range(len(self.players)):
        self.getSmallPieces(self.players[i], numSmall)
        self.getHexagonalPieces(self.players[i], numHexagonal)


def getActions(self, player):
    """
    Función que devuelve todas las posibles acciones a realizar para el jugador especificado como parámetro.
    """
    actions = []
    if self.initialAction: # Movimiento inicial de la partida
        # Fichas pequeñas por el lado de la resta
        for smallPiece in player.smallPieces:
            smallPieceOrder = [spo for spo in smallPiece.orders if spo[3] == '-'][0]
            smallPieceOrderSorted = sorted(smallPieceOrder[:3], reverse=True)
            score = int(smallPieceOrderSorted[0]) - int(smallPieceOrderSorted[2])
            actions = actions + [['', smallPiece.id, smallPieceOrder, 'H1_H4_H5', score]]

        # Fichas pequeñas por el lado de la suma
        for smallPiece in player.smallPieces:
            smallPieceOrder = [spo for spo in smallPiece.orders if spo[3] == '+'][0]
            smallPieceOrderSorted = sorted(smallPieceOrder[:3], reverse=True)
            score = int(smallPieceOrderSorted[0]) + int(smallPieceOrderSorted[1])
            actions = actions + [['', smallPiece.id, smallPieceOrder, 'H1_H4_H5', score]]

        # Fichas hexagonales
        for hexagonalPiece in player.hexagonalPieces:
            score = max([i for i in hexagonalPiece.orders[0] if i != '*'])
            actions = actions + [['', hexagonalPiece.id, hexagonalPiece.orders[0], 'H1', int(score)]]

    else: # Movimiento no incial de la partida

        # Fichas pequeñas por el lado de la resta
        for smallPiece in player.smallPieces:
            for smallGapF in self.fittableSmallGaps:
                sgVertices = [vertixSmallGap[2] for vertixSmallGap in self.fittableSmallGaps[smallGapF].vertices]

                for spOrder in smallPiece.orders:
                    if (spOrder[3] == '-'):
                        auxVertices = np.array([], dtype=int)
                        for i in range(3):
                            if i == 0:
                                auxVertices = np.append(auxVertices, abs(int(spOrder[i]) - int(spOrder[i+2])))
                            else: # i == 1 or i == 2
                                auxVertices = np.append(auxVertices, abs(int(spOrder[i]) - int(spOrder[i-1])))
                            i += 1
                        score = -1

                        for i in range(3):
                            if str(sgVertices[i]) == str(auxVertices[i]):
                                if int(sgVertices[i]) == 0:
                                    score = 0
                                elif score == -1:
                                    score += int(sgVertices[i]) + 1
                                else:
                                    score += int(sgVertices[i])
                            elif str(sgVertices[i]) != '_' and str(sgVertices[i]) != '*':
```

```
                            score = -1
                            break

                    if score > -1:
                        actions = actions + [['', smallPiece.id, spOrder, smallGapF, score]]

        # Fichas pequeñas por el lado de la suma
        for smallPiece in player.smallPieces:
            for smallGapF in self.fittableSmallGaps:
                sgVertices = [vertixSmallGap[2] for vertixSmallGap in self.fittableSmallGaps[smallGapF].vertices]

                for spOrder in smallPiece.orders:
                    if (spOrder[3] == '+'):
                        auxVertices = np.array([], dtype=int)
                        for i in range(3):
                            if i == 0:
                                auxVertices = np.append(auxVertices, int(spOrder[i]) + int(spOrder[i+2]))
                            else: # i == 1 or i == 2
                                auxVertices = np.append(auxVertices, int(spOrder[i]) + int(spOrder[i-1]))
                            i += 1
                        score = -1

                        for i in range(3):
                            if str(sgVertices[i]) == str(auxVertices[i]):
                                if int(sgVertices[i]) == 0:
                                    score = 0
                                elif score == -1:
                                    score += int(sgVertices[i]) + 1
                                else:
                                    score += int(sgVertices[i])
                            elif str(sgVertices[i]) != '_' and str(sgVertices[i]) != '*':
                                score = -1
                                break

                        if score > -1:
                            actions = actions + [['', smallPiece.id, spOrder, smallGapF, score]]

        # Fichas hexagonales
        for hexagonalPiece in player.hexagonalPieces:
            for hexGapF in self.fittableHexagonalGaps:
                hgVertices = [vertixHexGap[2] for vertixHexGap in self.fittableHexagonalGaps[hexGapF].vertices]
                for hpOrder in hexagonalPiece.orders:
                    score = -1

                    for i in range(6):
                        if str(hgVertices[i]) == str(hpOrder[i]) and str(hpOrder[i]) != '*':
                            if int(hgVertices[i]) == 0:
                                score = 0
                            elif score == -1:
                                score += int(hgVertices[i]) + 1
                            else:
                                score += int(hgVertices[i])
                        elif (str(hgVertices[i]) != '_' and str(hpOrder[i]) != '*'):
                            score = -1
                            break

                    if score > -1:
                        actions = actions + [['', hexagonalPiece.id, hpOrder, hexGapF, int(score)]]

        # Robar una ficha si no se ha robado otra antes en el mismo turno
        if not player.gotPieceBefore and len(self.smallPiecesAvailable) > 0:
            actions = actions + [['getSmallPiece', None, None, None, 0]]
        if player.gotPieceBefore and len(actions) == 0:
            actions = actions + [['passTurn', None, None, None, 0]]

        if len(actions) == 0:
            actions = actions + [['passTurn', None, None, None, 0]]

    return actions


def getHexagonalPieces(self, player, numPieces=1):
    """
    Función que selecciona al azar fichas hexagonales sin reemplazamiento.
    Es válida tanto para obtener las fichas hexagonales al inicio del juego como para robar una nueva.
    PARAMETERS:
        player: jugador que obtiene las fichas hexagonales.
        numPieces: número de fichas que se obtienen. Por defecto 1.
    """
    hexagonalPiecesSelected = []
```

```
        # Inicialización de la semilla para partidas con el agente QLearner
#         if numPieces == 1: # Las fichas se roban aleatoriamente
#             np.random.seed(int(datetime.now().strftime("%f")))
#         else:  # Se establece una semilla para repartir inicialmente
#             np.random.seed(20200104)

        # Inicialización de la semilla para partidas con el agente Minimax
        if numPieces == 1: # Se establece una semilla para robar fichas
            np.random.seed(20200121)
        else: # Se reparte inicialmente de forma aleatoria
            np.random.seed(int(datetime.now().strftime("%f")))


        if len(self.hexagonalPiecesAvailable) >= numPieces:
            hexagonalPiecesSelected = np.random.choice(self.hexagonalPiecesAvailable, numPieces, replace=False)

            # Se eliminan las fichas hexagonales asignadas de la lista de las que quedan disponibles
            self.hexagonalPiecesAvailable = [piece for piece in self.hexagonalPiecesAvailable if piece not in hexagonalPiecesSelected]

            # Se incluye la nueva ficha hexagonal en la lista de las fichas del jugador
            player.hexagonalPieces = np.append(player.hexagonalPieces, hexagonalPiecesSelected)

            print("[{}]: Roba {} ".format(player.name, numPieces), end = '')
            print("fichas hexagonales.") if numPieces > 1 else print("ficha hexagonal.")
        elif len(self.hexagonalPiecesAvailable) == 0:
            print("No existen fichas hexagonales disponibles.")
        elif len(self.hexagonalPiecesAvailable) == 1:
            print("Solamente existe 1 ficha hexagonal disponible.")
        else:
            print("Solamente existen {} fichas hexagonales disponibles.".format(len(self.hexagonalPiecesAvailable)))


    def getSmallPieces(self, player, numPieces=1):
        """
        Función que selecciona al azar fichas pequeñas sin reemplazamiento.
        Es válida tanto para obtener las fichas pequeñas al inicio del juego como para robar una nueva.
        PARAMETERS:
            player: jugador que obtiene las fichas pequeñas.
            numPieces: número de fichas que se obtienen. Por defecto 1.
        """
        player.gotPieceBefore = False
        smallPiecesSelected = []

        # Inicialización de la semilla para partidas con el agente QLearner
#         if numPieces == 1: # Las fichas se roban aleatoriamente
#             np.random.seed(int(datetime.now().strftime("%f")))
#         else:  # Se establece una semilla para repartir inicialmente
#             np.random.seed(20200104)

        # Inicialización de la semilla para partidas con el agente Minimax
        if numPieces == 1: # Se establece una semilla para robar fichas
            np.random.seed(20200121)
        else: # Se reparte inicialmente de forma aleatoria
            np.random.seed(int(datetime.now().strftime("%f")))


        if len(self.smallPiecesAvailable) >= numPieces:
            smallPiecesSelected = np.random.choice(self.smallPiecesAvailable, numPieces, replace=False)
            self.setRedIndicator(player, value=False) # Si estuviera activo el testigo rojo se desactiva
            if numPieces == 1:
                print("[{}]: Roba 1 ficha pequeña.".format(player.name))
                self.changeWhiteIndicator(player)
                player.gotPieceBefore = True
            else:
                print("[{}]: Roba {} fichas pequeñas.".format(player.name, numPieces))

            # Se eliminan las fichas pequeñas asignadas de la lista de las que quedan disponibles
            self.smallPiecesAvailable = [piece for piece in self.smallPiecesAvailable if piece not in smallPiecesSelected]

            # Se incluye la nueva ficha pequeña en la lista de las fichas del jugador
            player.smallPieces = np.append(player.smallPieces, smallPiecesSelected)

        elif len(self.smallPiecesAvailable) == 0:
            print("[{}]: Intenta robar una ficha pequeña. No hay ninguna disponible.".format(player.name))
        elif len(self.smallPiecesAvailable) == 1:
            print("Solamente existe 1 ficha pequeña disponible.")
        else:
            print("Solamente existen {} fichas pequeñas disponibles.".format(len(self.smallPiecesAvailable)))
```

```
def addScore(self, player, score):
    """
    Función que incrementa la puntuación acumulada del jugador con un nuevo valor.
    PARAMETERS:
        player: jugador.
        score: número de puntos que se incrementan.
    """
    player.score += score


def changeWhiteIndicator(self, player):
    """
    Función que actualiza el número de testigos blancos y, en caso de llegar a tres, roba una ficha hexagonal.
    PARAMETER:
        player: jugador.
    """
    player.whiteIndicator += 1
    if player.whiteIndicator == 1:
        print("[{}]: {} testigo blanco activo.".format(player.name, player.whiteIndicator))
    else:
        if player.whiteIndicator == 3: # Se roba una ficha hexagonal
            self.getHexagonalPieces(player)
            player.whiteIndicator = 0
        print("[{}]: {} testigos blancos activos.".format(player.name, player.whiteIndicator))


def setRedIndicator(self, player, value):
    """
    Función que modifica el estado del testigo rojo.
    PARAMETERS:
        player: jugador.
        value: nuevo estado al que se actualiza el testigo rojo.
    """
    if player.redIndicator or value:
        player.redIndicator = value
        if player.redIndicator:
            print("[{}]: Testigo rojo activo. Última ficha pequeña.".format(player.name))
        else:
            print("[{}]: Testigo rojo desactivado.".format(player.name))


def fitSmallPiece(self, player, piece, order, gap, score):
    """
    Función que coloca una ficha pequeña en uno de los huecos libres del espacio de juego.
    PARAMETERS:
        player: jugador.
        piece: identificador de la ficha pequeña que se va a colocar.
        order: lista que contiene el orden de los números de la ficha pequeña que se va a colocar, junto con el signo '+' o '-'.
        gap: identificador del hueco donde se va a colocar la ficha pequeña.
    """
    i = 0

    for vertixSmall in self.smallGaps[gap].vertices:
        if order[3] == '+':
            if i == 0:
                vertixSmall[2] = int(order[i]) + int(order[i+2])
            else:
                vertixSmall[2] = int(order[i]) + int(order[i-1])
        if order[3] == '-':
            if i == 0:
                vertixSmall[2] = abs(int(order[i]) - int(order[i+2]))
            else:
                vertixSmall[2] = abs(int(order[i]) - int(order[i-1]))

        for hexagonalGap in self.hexagonalGaps:
            for vertixHex in self.hexagonalGaps[hexagonalGap].vertices:
                if vertixHex[0] == vertixSmall[0] and int(vertixHex[1]) == int(vertixSmall[1]): # Coinciden las coordenadas
                    if vertixHex[2] == '_': # El hueco de la ficha hexagonal está sin ocupar
                        if order[3] == '+':
                            if i == 0:
                                vertixHex[2] = int(order[i]) + int(order[i+2])
                            else:
                                vertixHex[2] = int(order[i]) + int(order[i-1])
                        if order[3] == '-':
                            if i == 0:
                                vertixHex[2] = abs(int(order[i]) - int(order[i+2]))
                            else:
                                vertixHex[2] = abs(int(order[i]) - int(order[i-1]))

                        self.fittableHexagonalGaps[hexagonalGap] = self.hexagonalGaps[hexagonalGap]
                        if gap in self.fittableSmallGaps:
                            del self.fittableSmallGaps[gap]
```

```
                    else: # Si el hueco ya está ocupado por una ficha hexagonal se suman los puntos del vértice
                        vertixSmall[3] = 'O'
                        vertixHex[3] = 'O'
            i += 1

        player.smallPieces = [sp for sp in player.smallPieces if sp.id != piece]
        self.addScore(player, score)

        if order[3] == '-': # Si se pone la ficha por el lado de la resta se añade un testigo blanco
            self.changeWhiteIndicator(player)

        if len(player.smallPieces) == 1: # Última ficha pequeña
            self.setRedIndicator(player, value=True)


    def fitHexagonalPiece(self, player, piece, order, gap, score):
        """
        Función que coloca una ficha hexagonal en uno de los huecos libres del espacio de juego.
        PARAMETERS:
            player: jugador.
            piece: identificador de la ficha hexagonal que se va a colocar.
            gap: identificador del hueco donde se va a colocar la ficha hexagonal.
            order: lista que contiene el orden de los números de la ficha hexagonal que se va a colocar, incluyendo el comodín ('*').
        """
        i = 0

        for vertixHex in self.hexagonalGaps[gap].vertices:
            vertixHex[2] = order[i]
            for smallGap in self.smallGaps:
                for vertixSmall in self.smallGaps[smallGap].vertices:
                    if vertixSmall[0] == vertixHex[0] and int(vertixSmall[1]) == int(vertixHex[1]): # Coinciden las coordenadas
                        if vertixSmall[2] == '_': # El hueco de la ficha pequeña está sin ocupar
                            vertixSmall[2] = order[i]
                            self.fittableSmallGaps[smallGap] = self.smallGaps[smallGap]
                            if gap in self.fittableHexagonalGaps:
                                del self.fittableHexagonalGaps[gap]

                        else: # Si el hueco ya está ocupado por una ficha pequeña, se suman los puntos del vértice
                            vertixSmall[3] = 'O'
                            vertixHex[3] = 'O'
            i += 1

        player.hexagonalPieces = [hp for hp in player.hexagonalPieces if hp.id != piece]
        self.addScore(player, score)


    def move(self, chosenAction, player):
        """
        Función que ejecuta la acción que decide el jugador (agente).
        PARAMATERS:
            chosenAction: acción elegida.
            player: jugador
        """
        if len(player.smallPieces) == 0 or self.turnPassed == 2: # Si el jugador se ha quedado sin fichas pequeñas o ambos jugadores han
pasado en su turno, termina la partida
            self.printFinalScore()
            self.gameOver = True
            self.changeTurn(player)
        else:
            if chosenAction[0] == "getSmallPiece": # El jugador roba una ficha pequeña y juega de nuevo
                self.getSmallPieces(player)
                self.turnPassed = 0
            elif chosenAction[0] == "passTurn": # Pasa el turno al siguiente jugador
                print("[{}]: Pasa el turno".format(player.name))
                self.changeWhiteIndicator(player)
                self.turnPassed += 1
                self.changeTurn(player)
            else:
                if chosenAction[1][:2]=='hp': # El jugador coloca una ficha hexagonal
                    self.fitHexagonalPiece(player, chosenAction[1], chosenAction[2],
                                           chosenAction[3], chosenAction[4])
                    self.changeTurn(player)
                else: # El jugador coloca una ficha pequeña
                    self.fitSmallPiece(player, chosenAction[1], chosenAction[2],
                                       chosenAction[3], chosenAction[4])
                    self.changeTurn(player)
                self.turnPassed = 0

                print('[{}]: Coloca la ficha {}, con orden {} en el hueco {}'.format(player.name,
                                                                                     chosenAction[1],
                                                                                     chosenAction[2],
                                                                                     chosenAction[3]))
            player.accumulatedScore += chosenAction[4]
            self.initialAction = False
```

```
    def finalPointCount(self, player):
        """
        Función que realiza el recuento final de puntos.
        A la cantidad de puntos que el jugador ha ido acumulando se le resta la suma de los números más altos de cada una de las fichas
pequeñas que le hayan quedado sin colocar.
        PARAMETER:
            player: jugador.
        RETURNS:
            score: puntuación definitiva.
            subtraction: puntos que se le resta a la puntuación que tenía antes del recuento final.
        """
        subtraction = 0
        for piece in player.smallPieces:
            subtraction += max([int(i) for i in piece.orders[0] if i != '-' and i != '+'])
        player.score -= subtraction
        player.accumulatedScore -= subtraction

        return player.score, subtraction


    def printFinalScore(self):
        """
        Función que presenta el resultado final de la partida.
        """
        print("\nPuntuaciones finales de la partida:\n")
        winners = []
        winnerScore = 0
        maxScore = -1000

        for player in self.players:
            player.score, subtraction = self.finalPointCount(player)
            if player.score > maxScore:
                maxScore = player.score

            print("{}: {} puntos".format(player.name, player.score), end = '')
            if subtraction != 0:
                print(" (se le han restado {} puntos a los {} que tenía acumulados)".format(subtraction,
                                                                                            player.score + subtraction))
            else:
                print('\n')

        # Se comprueba si ha habido empate:
        for player in self.players:
            if player.score == maxScore:
                winners = np.append(winners, player)

    #     CBOLD = '\33[1m'
    #     CEND = '\033[0m'
        CBOLD = ''
        CEND = ''
        if len(winners) > 1:
            print(CBOLD + "Ha habido empate. LOS GANADORES SON: \n")
            i = 1
            for winner in winners:
                if i == 1:
                    print(winner.name, end = '')
                elif len(winners) == i:
                    print(" y " + str(winner.name))
                else:
                    print(", " + str(winner.name), end = '')
                i += 1
        else:
            print(CBOLD + "GANADOR:  {}".format(winners[0].name) + CEND)
            winners[0].numWins += 1

        print(CEND)
        print("\n-- Fin de la partida --")


    def summary(self):
        """
        Función que muestra el resumen del estado actual del juego.
        """
        print('-------------------------------------')
        for player in self.players:
            print("\nPuntuación de {}: {}".format(player.name, player.score))
            print("Fichas pequeñas de {}: ".format(player.name), end = '')
            for sp in player.smallPieces:
                print(str(sp.orders[0]) + " ", end = '')
            print("\nFichas hexagonales de {}: ".format(player.name), end = '')
            for hp in player.hexagonalPieces:
                print(str(hp.orders[0]) + " ", end = '')
            print("\nTestigos blancos: " + str(player.whiteIndicator))
            print("Testigo rojo: " + str(player.redIndicator))
```

```
        print("Número de fichas pequeñas disponibles sin repartir: " + str(len(self.smallPiecesAvailable)))
        print("Número de fichas hexagonales disponibles sin repartir: " + str(len(self.hexagonalPiecesAvailable)))

        print("Huecos habilitados para poder colocar alguna ficha hexagonal: ", end = '')
        for fhg in self.fittableHexagonalGaps:
            print(str(fhg) + "  ", end = '')
        print("\nHuecos habilitados para poder colocar alguna ficha pequeña: ")
        for fsg in self.fittableSmallGaps:
            print(str(fsg) + "  ", end = '')

        print('\n-------------------------------------')


    def changeTurn(self, player):
        """
        Función que cambia de turno para dar paso al siguiente jugador.
        """
        for p in self.players:
            if p == player:
                p.isTurn = False
            else:
                p.isTurn = True
            player.gotPieceBefore = False


    def getState(self, player):
        """
        Función que obtiene información del espacio de juego en un estado determinado.
        PARAMATER:
            player: jugador.
        RETUNR:
            El estado se devuelve transformado por una función hash.
        """
        s = self.State(
            # Tablero
            fittableHexagonalGaps = tuple(sorted([
                piece
                for piece in self.fittableHexagonalGaps
            ])), # tupla ordenada de strings ('hg1', 'hg2', ....)
            fittableSmallGaps =  tuple(sorted([
                piece
                for piece in self.fittableSmallGaps
            ])), # tupla ordenada de strings ('h1_h2_h3', 'h1_h3_h4', ....)

            # Informnación pública de los oponentes
            opponentsInfo = tuple([ # Tupla del los estados de los oponentes
                p.getPublicState()
                for p in self.players
                if p != player
            ]),
            # Informnación privada del jugador
            playerInfo = player.getPrivateState()#,
        )
        return s.__hash__()
```

initializations.py

```
import numpy as np
from gaps import HexagonalGap, SmallGap
from pieces import HexagonalPiece, SmallPiece
import player
import gameState

def initializePieces():
    """
    Función que crea/inicializa todas las fichas hexagonales y pequeñas.
    RETURNS:
        hexagonalPieces, smallPieces: listas de fichas hexagonales y pequeñas que van a estar disponibles a lo largo de la partida.
    """
    # Fichas Hexagonales:
    hp1 = HexagonalPiece('hp1', ['*', '5', '2', '4', '6', '6'])
    hp2 = HexagonalPiece('hp2', ['*', '2', '0', '4', '4', '6'])
    hp3 = HexagonalPiece('hp3', ['*', '3', '6', '2', '4', '2'])
    hp4 = HexagonalPiece('hp4', ['*', '4', '0', '3', '4', '6'])
    hp5 = HexagonalPiece('hp5', ['*', '5', '3', '4', '2', '1'])
    hp6 = HexagonalPiece('hp6', ['*', '2', '3', '4', '6', '5'])
    hp7 = HexagonalPiece('hp7', ['*', '6', '0', '5', '2', '4'])
    hp8 = HexagonalPiece('hp8', ['*', '6', '3', '0', '3', '5'])
    hp9 = HexagonalPiece('hp9', ['*', '2', '0', '1', '4', '5'])
    hp10 = HexagonalPiece('hp10', ['*', '6', '4', '2', '0', '3'])
```

```
    hp11 = HexagonalPiece('hp11', ['*', '0', '1', '2', '3', '4'])
    hp12 = HexagonalPiece('hp12', ['*', '6', '1', '4', '4', '3'])
    hp13 = HexagonalPiece('hp13', ['*', '4', '1', '6', '3', '5'])
    hp14 = HexagonalPiece('hp14', ['*', '4', '5', '2', '1', '4'])
    hp15 = HexagonalPiece('hp15', ['*', '4', '1', '0', '2', '2'])
    hp16 = HexagonalPiece('hp16', ['*', '6', '5', '5', '4', '3'])
    hp17 = HexagonalPiece('hp17', ['*', '6', '5', '1', '0', '3'])
    hp18 = HexagonalPiece('hp18', ['*', '5', '0', '4', '2', '6'])
    hp19 = HexagonalPiece('hp19', ['*', '5', '4', '4', '2', '0'])
    hp20 = HexagonalPiece('hp20', ['*', '4', '2', '3', '5', '6'])

    hexagonalPieces = [hp1,hp2,hp3,hp4,hp5,hp6,hp7,hp8,hp9,hp10,hp11,hp12,hp13,hp14,hp15,hp16,hp17,hp18,hp19,hp20]

    # Fichas Pequeñas:
    sp1 = SmallPiece('sp1', ['1','2','3'])
    sp2 = SmallPiece('sp2', ['1','1','2'])
    sp3 = SmallPiece('sp3', ['1','1','3'])
    sp4 = SmallPiece('sp4', ['1','1','4'])
    sp5 = SmallPiece('sp5', ['1','2','2'])
    sp6 = SmallPiece('sp6', ['1','3','3'])
    sp7 = SmallPiece('sp7', ['1','2','4'])
    sp8 = SmallPiece('sp8', ['2','2','2'])
    sp9 = SmallPiece('sp9', ['2','2','3'])
    sp10 = SmallPiece('sp10', ['2','2','4'])
    sp11 = SmallPiece('sp11', ['2','3','3'])
    sp12 = SmallPiece('sp12', ['3','3','3'])

    sp13 = SmallPiece('sp13', ['1','2','3'])
    sp14 = SmallPiece('sp14', ['1','1','2'])
    sp15 = SmallPiece('sp15', ['1','1','3'])
    sp16 = SmallPiece('sp16', ['1','1','4'])
    sp17 = SmallPiece('sp17', ['1','2','2'])
    sp18 = SmallPiece('sp18', ['1','3','3'])
    sp19 = SmallPiece('sp19', ['1','2','4'])
    sp20 = SmallPiece('sp20', ['2','2','2'])
    sp21 = SmallPiece('sp21', ['2','2','3'])
    sp22 = SmallPiece('sp22', ['2','2','4'])
    sp23 = SmallPiece('sp23', ['2','3','3'])
    sp24 = SmallPiece('sp24', ['3','3','3'])

    sp25 = SmallPiece('sp25', ['1','2','3'])
    sp26 = SmallPiece('sp26', ['1','1','2'])
    sp27 = SmallPiece('sp27', ['1','1','3'])
    sp28 = SmallPiece('sp28', ['1','1','4'])
    sp29 = SmallPiece('sp29', ['1','2','2'])
    sp30 = SmallPiece('sp30', ['1','3','3'])
    sp31 = SmallPiece('sp31', ['1','2','4'])
    sp32 = SmallPiece('sp32', ['2','2','2'])
    sp33 = SmallPiece('sp33', ['2','2','3'])
    sp34 = SmallPiece('sp34', ['2','2','4'])
    sp35 = SmallPiece('sp35', ['2','3','3'])
    sp36 = SmallPiece('sp36', ['3','3','3'])

    sp37 = SmallPiece('sp37', ['1','2','3'])
    sp38 = SmallPiece('sp38', ['1','1','2'])
    sp39 = SmallPiece('sp39', ['1','1','3'])
    sp40 = SmallPiece('sp40', ['1','1','4'])
    sp41 = SmallPiece('sp41', ['1','2','2'])
    sp42 = SmallPiece('sp42', ['1','3','3'])
    sp43 = SmallPiece('sp43', ['1','2','4'])
    sp44 = SmallPiece('sp44', ['2','2','2'])
    sp45 = SmallPiece('sp45', ['2','2','3'])
    sp46 = SmallPiece('sp46', ['2','2','4'])
    sp47 = SmallPiece('sp47', ['2','3','3'])
    sp48 = SmallPiece('sp48', ['3','3','3'])

    smallPieces = [sp1,sp2,sp3,sp4,sp5,sp6,sp7,sp8,sp9,sp10,sp11,sp12,sp13,sp14,sp15,sp16,
                   sp17,sp18,sp19,sp20,sp21,sp22,sp23,sp24,sp25,sp26,sp27,sp28,sp29,sp30,sp31,sp32,
                   sp33,sp34,sp35,sp36,sp37,sp38,sp39,sp40,sp41,sp42,sp43,sp44,sp45,sp46,sp47,sp48]

    return hexagonalPieces, smallPieces


def initializeGaps():
    """
    Función que crea/inicializa todos los huecos del tablero virtual.
    RETURNS:
        hexagonalGaps, smallGaps: listas de huecos para fichas hexagonales y pequeñas.
        fittableHexagonalGaps, fittableSmallGaps: listas de huecos, para fichas hexagonales y pequeñas, en los que se puede colocar alguna
ficha.
    """
    # Huecos para fichas hexagonales:
    h1 = HexagonalGap('H1')
    h2 = HexagonalGap('H2')
```

```
h3 = HexagonalGap('H3')
h4 = HexagonalGap('H4')
h5 = HexagonalGap('H5')
h6 = HexagonalGap('H6')
h7 = HexagonalGap('H7')
h8 = HexagonalGap('H8')
h9 = HexagonalGap('H9')
h10 = HexagonalGap('H10')
h11 = HexagonalGap('H11')
h12 = HexagonalGap('H12')
h13 = HexagonalGap('H13')
h14 = HexagonalGap('H14')
h15 = HexagonalGap('H15')
h16 = HexagonalGap('H16')
h17 = HexagonalGap('H17')
h18 = HexagonalGap('H18')
h19 = HexagonalGap('H19')
h20 = HexagonalGap('H20')
h21 = HexagonalGap('H21')
h22 = HexagonalGap('H22')
h23 = HexagonalGap('H23')
h24 = HexagonalGap('H24')
h25 = HexagonalGap('H25')
h26 = HexagonalGap('H26')
h27 = HexagonalGap('H27')
h28 = HexagonalGap('H28')
h29 = HexagonalGap('H29')
h30 = HexagonalGap('H30')
h31 = HexagonalGap('H31')
h32 = HexagonalGap('H32')
h33 = HexagonalGap('H33')
h34 = HexagonalGap('H34')
h35 = HexagonalGap('H35')
h36 = HexagonalGap('H36')
h37 = HexagonalGap('H37')
h38 = HexagonalGap('H38')
h39 = HexagonalGap('H39')
h40 = HexagonalGap('H40')
h41 = HexagonalGap('H41')
h42 = HexagonalGap('H42')
h43 = HexagonalGap('H43')
h44 = HexagonalGap('H44')
h45 = HexagonalGap('H45')
h46 = HexagonalGap('H46')
h47 = HexagonalGap('H47')
h48 = HexagonalGap('H48')
h49 = HexagonalGap('H49')
h50 = HexagonalGap('H50')
h51 = HexagonalGap('H51')
h52 = HexagonalGap('H52')
h53 = HexagonalGap('H53')
h54 = HexagonalGap('H54')
h55 = HexagonalGap('H55')

hexagonalGaps = {'H1': h1, 'H2': h2, 'H3': h3, 'H4': h4, 'H5': h5, 'H6': h6, 'H7': h7, 'H8': h8, 'H9': h9,
                 'H10': h10, 'H11': h11, 'H12': h12, 'H13': h13, 'H14': h14, 'H15': h15, 'H16': h16,
                 'H17': h17, 'H18': h18, 'H19': h19, 'H20': h20, 'H21': h21, 'H22': h22, 'H23': h23,
                 'H24': h24, 'H25': h25, 'H26': h26, 'H27': h27, 'H28': h28, 'H29': h29, 'H30': h30,
                 'H31': h31, 'H32': h32, 'H33': h33, 'H34': h34, 'H35': h35, 'H36': h36, 'H37': h37,
                 'H38': h38, 'H39': h39, 'H40': h40, 'H41': h41, 'H42': h42, 'H43': h43, 'H44': h44,
                 'H45': h45, 'H46': h46, 'H47': h47, 'H48': h48, 'H49': h49, 'H50': h50, 'H51': h51,
                 'H52': h52, 'H53': h53, 'H54': h54, 'H55': h55}

# Huecos para fichas pequeñas. Cada elemento de la lista está formado por los siguientes valores:
#    - id del hueco hexagonal del tablero.
#    - Número asignado a ese vértice. '_' indica que el vértice aún no tiene valor asignado.
#    - V --> Vacant; 'O' --> Occupied
h1_h2_h3 = SmallGap([['H1', '2', '_', 'V'], ['H2', '4', '_', 'V'], ['H3', '6', '_', 'V']])
h1_h3_h4 = SmallGap([['H1', '3', '_', 'V'], ['H3', '5', '_', 'V'], ['H4', '1', '_', 'V']])
h1_h4_h5 = SmallGap([['H1', '4', '_', 'V'], ['H4', '6', '_', 'V'], ['H5', '2', '_', 'V']])
h1_h5_h6 = SmallGap([['H1', '5', '_', 'V'], ['H5', '1', '_', 'V'], ['H6', '3', '_', 'V']])
h1_h6_h7 = SmallGap([['H1', '6', '_', 'V'], ['H6', '2', '_', 'V'], ['H7', '4', '_', 'V']])
h1_h7_h2 = SmallGap([['H1', '1', '_', 'V'], ['H7', '3', '_', 'V'], ['H2', '5', '_', 'V']])
h2_h7_h8 = SmallGap([['H2', '6', '_', 'V'], ['H7', '2', '_', 'V'], ['H8', '4', '_', 'V']])
h2_h8_h9 = SmallGap([['H2', '1', '_', 'V'], ['H8', '3', '_', 'V'], ['H9', '5', '_', 'V']])
h2_h9_h10 = SmallGap([['H2', '2', '_', 'V'], ['H9', '4', '_', 'V'], ['H10', '6', '_', 'V']])
h2_h10_h3= SmallGap([['H2', '3', '_', 'V'], ['H10', '5', '_', 'V'], ['H3', '1', '_', 'V']])
h3_h10_h11 = SmallGap([['H3', '2', '_', 'V'], ['H10', '4', '_', 'V'], ['H11', '6', '_', 'V']])
h3_h11_h12 = SmallGap([['H3', '3', '_', 'V'], ['H11', '5', '_', 'V'], ['H12', '1', '_', 'V']])
h3_h12_h4 = SmallGap([['H3', '4', '_', 'V'], ['H12', '6', '_', 'V'], ['H4', '2', '_', 'V']])
h4_h12_h13 = SmallGap([['H4', '3', '_', 'V'], ['H12', '5', '_', 'V'], ['H13', '1', '_', 'V']])
h4_h13_h14 = SmallGap([['H4', '4', '_', 'V'], ['H13', '6', '_', 'V'], ['H14', '2', '_', 'V']])
h4_h14_h5 = SmallGap([['H4', '5', '_', 'V'], ['H14', '1', '_', 'V'], ['H5', '3', '_', 'V']])
h5_h14_h15 = SmallGap([['H5', '4', '_', 'V'], ['H14', '6', '_', 'V'], ['H15', '2', '_', 'V']])
```

```
h5_h15_h16 = SmallGap([['H5', '5', '_', 'V'], ['H15', '1', '_', 'V'], ['H16', '3', '_', 'V']])
h5_h16_h6 = SmallGap([['H5', '6', '_', 'V'], ['H16', '2', '_', 'V'], ['H6', '4', '_', 'V']])
h6_h16_h17 = SmallGap([['H6', '5', '_', 'V'], ['H16', '1', '_', 'V'], ['H17', '3', '_', 'V']])
h6_h17_h18 = SmallGap([['H6', '6', '_', 'V'], ['H17', '2', '_', 'V'], ['H18', '4', '_', 'V']])
h6_h18_h7 = SmallGap([['H6', '1', '_', 'V'], ['H18', '3', '_', 'V'], ['H7', '5', '_', 'V']])
h7_h18_h19 = SmallGap([['H7', '6', '_', 'V'], ['H18', '2', '_', 'V'], ['H19', '4', '_', 'V']])
h7_h19_h8 = SmallGap([['H7', '1', '_', 'V'], ['H19', '3', '_', 'V'], ['H8', '5', '_', 'V']])
h8_h19_h20 = SmallGap([['H8', '6', '_', 'V'], ['H19', '2', '_', 'V'], ['H20', '4', '_', 'V']])
h8_h20_h21 = SmallGap([['H8', '1', '_', 'V'], ['H20', '3', '_', 'V'], ['H21', '5', '_', 'V']])
h8_h21_h9 = SmallGap([['H8', '2', '_', 'V'], ['H21', '4', '_', 'V'], ['H9', '6', '_', 'V']])
h9_h21_h22 = SmallGap([['H9', '1', '_', 'V'], ['H21', '3', '_', 'V'], ['H22', '5', '_', 'V']])
h9_h22_h23 = SmallGap([['H9', '2', '_', 'V'], ['H22', '4', '_', 'V'], ['H23', '6', '_', 'V']])
h9_h23_h10 = SmallGap([['H9', '3', '_', 'V'], ['H23', '5', '_', 'V'], ['H10', '1', '_', 'V']])
h10_h23_h24 = SmallGap([['H10', '2', '_', 'V'], ['H23', '4', '_', 'V'], ['H24', '6', '_', 'V']])
h10_h24_h11 = SmallGap([['H10', '3', '_', 'V'], ['H24', '5', '_', 'V'], ['H11', '1', '_', 'V']])
h11_h24_h51 = SmallGap([['H11', '2', '_', 'V'], ['H24', '4', '_', 'V'], ['H51', '6', '_', 'V']])
h11_h51_h25 = SmallGap([['H11', '3', '_', 'V'], ['H51', '5', '_', 'V'], ['H25', '1', '_', 'V']])
h11_h25_h12 = SmallGap([['H11', '4', '_', 'V'], ['H25', '6', '_', 'V'], ['H12', '2', '_', 'V']])
h12_h25_h26 = SmallGap([['H12', '3', '_', 'V'], ['H25', '5', '_', 'V'], ['H26', '1', '_', 'V']])
h12_h26_h13 = SmallGap([['H12', '4', '_', 'V'], ['H26', '6', '_', 'V'], ['H13', '2', '_', 'V']])
h13_h26_h27 = SmallGap([['H13', '3', '_', 'V'], ['H26', '5', '_', 'V'], ['H27', '1', '_', 'V']])
h13_h27_h28 = SmallGap([['H13', '4', '_', 'V'], ['H27', '6', '_', 'V'], ['H28', '2', '_', 'V']])
h13_h28_h14 = SmallGap([['H13', '5', '_', 'V'], ['H28', '1', '_', 'V'], ['H14', '3', '_', 'V']])
h14_h28_h29 = SmallGap([['H14', '4', '_', 'V'], ['H28', '6', '_', 'V'], ['H29', '2', '_', 'V']])
h14_h29_h15 = SmallGap([['H14', '5', '_', 'V'], ['H29', '1', '_', 'V'], ['H15', '3', '_', 'V']])
h15_h29_h30 = SmallGap([['H15', '4', '_', 'V'], ['H29', '6', '_', 'V'], ['H30', '2', '_', 'V']])
h15_h30_h31 = SmallGap([['H15', '5', '_', 'V'], ['H30', '1', '_', 'V'], ['H31', '3', '_', 'V']])
h15_h31_h16 = SmallGap([['H15', '6', '_', 'V'], ['H31', '2', '_', 'V'], ['H16', '4', '_', 'V']])
h16_h31_h32 = SmallGap([['H16', '5', '_', 'V'], ['H31', '1', '_', 'V'], ['H32', '3', '_', 'V']])
h16_h32_h17 = SmallGap([['H16', '6', '_', 'V'], ['H32', '2', '_', 'V'], ['H17', '4', '_', 'V']])
h17_h32_h54 = SmallGap([['H17', '5', '_', 'V'], ['H32', '1', '_', 'V'], ['H54', '3', '_', 'V']])
h17_h54_h33 = SmallGap([['H17', '6', '_', 'V'], ['H54', '2', '_', 'V'], ['H33', '4', '_', 'V']])
h17_h33_h18 = SmallGap([['H17', '1', '_', 'V'], ['H33', '3', '_', 'V'], ['H18', '5', '_', 'V']])
h18_h33_h34 = SmallGap([['H18', '6', '_', 'V'], ['H33', '2', '_', 'V'], ['H34', '4', '_', 'V']])
h18_h34_h19 = SmallGap([['H18', '1', '_', 'V'], ['H34', '3', '_', 'V'], ['H19', '5', '_', 'V']])
h19_h34_h35 = SmallGap([['H19', '6', '_', 'V'], ['H34', '2', '_', 'V'], ['H35', '4', '_', 'V']])
h19_h35_h20 = SmallGap([['H19', '1', '_', 'V'], ['H35', '3', '_', 'V'], ['H20', '5', '_', 'V']])
h20_h35_h36 = SmallGap([['H20', '6', '_', 'V'], ['H35', '2', '_', 'V'], ['H36', '4', '_', 'V']])
h20_h36_h37 = SmallGap([['H20', '1', '_', 'V'], ['H36', '3', '_', 'V'], ['H37', '5', '_', 'V']])
h20_h37_h21 = SmallGap([['H20', '2', '_', 'V'], ['H37', '4', '_', 'V'], ['H21', '6', '_', 'V']])
h21_h37_h38 = SmallGap([['H21', '1', '_', 'V'], ['H37', '3', '_', 'V'], ['H38', '5', '_', 'V']])
h21_h38_h22 = SmallGap([['H21', '2', '_', 'V'], ['H38', '4', '_', 'V'], ['H22', '6', '_', 'V']])
h22_h38_h39 = SmallGap([['H22', '1', '_', 'V'], ['H38', '3', '_', 'V'], ['H39', '5', '_', 'V']])
h22_h39_h40 = SmallGap([['H22', '2', '_', 'V'], ['H39', '4', '_', 'V'], ['H40', '6', '_', 'V']])
h22_h40_h23 = SmallGap([['H22', '3', '_', 'V'], ['H40', '5', '_', 'V'], ['H23', '1', '_', 'V']])
h23_h40_h50 = SmallGap([['H23', '2', '_', 'V'], ['H40', '4', '_', 'V'], ['H50', '6', '_', 'V']])
h23_h50_h24 = SmallGap([['H23', '3', '_', 'V'], ['H50', '5', '_', 'V'], ['H24', '1', '_', 'V']])
h26_h25_h52 = SmallGap([['H26', '2', '_', 'V'], ['H25', '4', '_', 'V'], ['H52', '6', '_', 'V']])
h26_h52_h41 = SmallGap([['H26', '3', '_', 'V'], ['H52', '5', '_', 'V'], ['H41', '1', '_', 'V']])
h26_h41_h27 = SmallGap([['H26', '4', '_', 'V'], ['H41', '6', '_', 'V'], ['H27', '2', '_', 'V']])
h27_h41_h42 = SmallGap([['H27', '3', '_', 'V'], ['H41', '5', '_', 'V'], ['H42', '1', '_', 'V']])
h27_h42_h43 = SmallGap([['H27', '4', '_', 'V'], ['H42', '6', '_', 'V'], ['H43', '2', '_', 'V']])
h27_h43_h28 = SmallGap([['H27', '5', '_', 'V'], ['H43', '1', '_', 'V'], ['H28', '3', '_', 'V']])
h28_h43_h44 = SmallGap([['H28', '4', '_', 'V'], ['H43', '6', '_', 'V'], ['H44', '2', '_', 'V']])
h28_h44_h29 = SmallGap([['H28', '5', '_', 'V'], ['H44', '1', '_', 'V'], ['H29', '3', '_', 'V']])
h29_h44_h45 = SmallGap([['H29', '4', '_', 'V'], ['H44', '6', '_', 'V'], ['H45', '2', '_', 'V']])
h29_h45_h30 = SmallGap([['H29', '5', '_', 'V'], ['H45', '1', '_', 'V'], ['H30', '3', '_', 'V']])
h30_h45_h46 = SmallGap([['H30', '4', '_', 'V'], ['H45', '6', '_', 'V'], ['H46', '2', '_', 'V']])
h30_h46_h47 = SmallGap([['H30', '5', '_', 'V'], ['H46', '1', '_', 'V'], ['H47', '3', '_', 'V']])
h30_h47_h31 = SmallGap([['H30', '6', '_', 'V'], ['H47', '2', '_', 'V'], ['H31', '4', '_', 'V']])
h31_h47_h55 = SmallGap([['H31', '5', '_', 'V'], ['H47', '1', '_', 'V'], ['H55', '3', '_', 'V']])
h31_h55_h32 = SmallGap([['H31', '6', '_', 'V'], ['H55', '2', '_', 'V'], ['H32', '4', '_', 'V']])
h34_h33_h53 = SmallGap([['H34', '5', '_', 'V'], ['H33', '1', '_', 'V'], ['H53', '3', '_', 'V']])
h34_h53_h48 = SmallGap([['H34', '6', '_', 'V'], ['H53', '2', '_', 'V'], ['H48', '4', '_', 'V']])
h34_h48_h35 = SmallGap([['H34', '1', '_', 'V'], ['H48', '3', '_', 'V'], ['H35', '5', '_', 'V']])
h35_h48_h49 = SmallGap([['H35', '6', '_', 'V'], ['H48', '2', '_', 'V'], ['H49', '4', '_', 'V']])
h35_h49_h36 = SmallGap([['H35', '1', '_', 'V'], ['H49', '3', '_', 'V'], ['H36', '5', '_', 'V']])

smallGaps = {'H1_H2_H3':h1_h2_h3, 'H1_H3_H4':h1_h3_h4, 'H1_H4_H5':h1_h4_h5, 'H1_H5_H6':h1_h5_h6,
             'H1_H6_H7':h1_h6_h7, 'H1_H7_H2':h1_h7_h2, 'H2_H7_H8':h2_h7_h8, 'H2_H8_H9':h2_h8_h9,
             'H2_H9_H10':h2_h9_h10, 'H2_H10_H3':h2_h10_h3, 'H3_H10_H11':h3_h10_h11, 'H7_H19_H8':h7_h19_h8,
             'H3_H12_H4':h3_h12_h4, 'H4_H12_H13':h4_h12_h13, 'H4_H13_H14':h4_h13_h14,
             'H5_H14_H15':h5_h14_h15, 'H5_H15_H16':h5_h15_h16, 'H5_H16_H6':h5_h16_h6,
             'H6_H17_H18':h6_h17_h18, 'H6_H18_H7':h6_h18_h7, 'H7_H18_H19':h7_h18_h19,
             'H8_H19_H20':h8_h19_h20, 'H8_H20_H21':h8_h20_h21, 'H8_H21_H9':h8_h21_h9,
             'H3_H11_H12':h3_h11_h12, 'H4_H14_H5':h4_h14_h5, 'H6_H16_H17':h6_h16_h17,
             'H9_H22_H23':h9_h22_h23, 'H9_H23_H10':h9_h23_h10, 'H10_H23_H24':h10_h23_h24,
             'H10_H24_H11':h10_h24_h11, 'H11_H24_H51':h11_h24_h51, 'H11_H51_H25':h11_h51_h25,
             'H11_H25_H12':h11_h25_h12, 'H12_H25_H26':h12_h25_h26, 'H12_H26_H13':h12_h26_h13,
             'H13_H26_H27':h13_h26_h27, 'H13_H27_H28':h13_h27_h28, 'H13_H28_H14':h13_h28_h14,
             'H14_H29_H15':h14_h29_h15, 'H15_H29_H30':h15_h29_h30, 'H15_H30_H31':h15_h30_h31,
             'H15_H31_H16':h15_h31_h16, 'H16_H31_H32':h16_h31_h32, 'H16_H32_H17':h16_h32_h17,
             'H17_H32_H54':h17_h32_h54, 'H17_H54_H33':h17_h54_h33, 'H17_H33_H18':h17_h33_h18,
             'H18_H33_H34':h18_h33_h34, 'H18_H34_H19':h18_h34_h19, 'H19_H34_H35':h19_h34_h35,
```

```
                'H19_H35_H20':h19_h35_h20, 'H20_H35_H36':h20_h35_h36, 'H20_H36_H37':h20_h36_h37,
                'H20_H37_H21':h20_h37_h21, 'H21_H37_H38':h21_h37_h38, 'H21_H38_H22':h21_h38_h22,
                'H22_H38_H39':h22_h38_h39, 'H22_H39_H40':h22_h39_h40, 'H22_H40_H23':h22_h40_h23,
                'H23_H40_H50':h23_h40_h50, 'H23_H50_H24':h23_h50_h24, 'H26_H25_H52':h26_h25_h52,
                'H26_H52_H41':h26_h52_h41, 'H26_H41_H27':h26_h41_h27, 'H27_H41_H42':h27_h41_h42,
                'H27_H42_H43':h27_h42_h43, 'H27_H43_H28':h27_h43_h28, 'H28_H43_H44':h28_h43_h44,
                'H28_H44_H29':h28_h44_h29, 'H29_H44_H45':h29_h44_h45, 'H29_H45_H30':h29_h45_h30,
                'H30_H45_H46':h30_h45_h46, 'H30_H46_H47':h30_h46_h47, 'H30_H47_H31':h30_h47_h31,
                'H31_H47_H55':h31_h47_h55, 'H31_H55_H32':h31_h55_h32, 'H34_H33_H53':h34_h33_h53,
                'H34_H53_H48':h34_h53_h48, 'H34_H48_H35':h34_h48_h35, 'H35_H48_H49':h35_h48_h49,
                'H35_H49_H36':h35_h49_h36, 'H14_H28_H29':h14_h28_h29, 'H9_H21_H22':h9_h21_h22}

    fittableHexagonalGaps = {}
    fittableSmallGaps = {}

    return hexagonalGaps, smallGaps, fittableHexagonalGaps, fittableSmallGaps


def initializeAll(players):
    """
    Función que inicializa todas fichas, los huecos y los jugadores.
    PARAMETER:
        players: lista de jugadores.
    RETURN:
        bameSpace: espacio de juego.
    """
    print("Se incicializan todas las fichas.")
    hexagonalPiecesAvailable, smallPiecesAvailable = initializePieces()

    print("Se incicializan todos los huecos.")
    hexagonalGaps, smallGaps, fittableHexagonalGaps, fittableSmallGaps = initializeGaps()

    print("Se inicializan los jugadores.")
    for player in players:
        player.isTurn = False
        player.smallPieces = []
        player.hexagonalPieces = []
        player.score = 0
        player.redIndicator = False
        player.whiteIndicator = 0
        player.gotPieceBefore = False

    print("Se crea el espacio de juego.")
    gameSpace = gameState.GameState(players, hexagonalPiecesAvailable, smallPiecesAvailable, hexagonalGaps, smallGaps,
                                    fittableHexagonalGaps, fittableSmallGaps)
    return gameSpace
```

utils.py

```
import numpy as np

def drawPlayersOrder(players):
    """
    Función que establece al azar el orden de intervención de los jugadores.
    PARAMETERS:
        players: lista de jugadores.
    RETURN:
        players: lista de jugadores con nuevo orden establecido.
    """
    np.random.shuffle(players)
    print("Por sorteo, el orden de los jugadores es: ", end = '')
    i = 1
    for player in players:
        player.isTurn = False
        if i == 1:
            player.isTurn = True
            print(player.name, end = '')
        elif i == len(players):
            print(" y " + str(player.name))
        else:
            print(", " + str(player.name), end = '' )
        i += 1

    return players
```

```
def listToString(s):
    """
    Función que convierte una lista a string.
    PARAMETERS:
        s: lista.
    RETURN:
        str1: string.
    """
    str1 = ""
    for element in s:
        str1 += element

    return str1
```

random.py

```
from player import Player
import gameState
import random

class RandomPlayer(Player):
    """
    Clase para implementar al jugador Random.
    """
    def __init__(self, name):
        super().__init__(name)
        self.Qfile = None


    def getMove(self, gameState, *args, **kwargs):
        """
        Función que juega el turno del agente Random. Elige de forma aleatoria la acción a ejecutar.
        PARAMETER:
            gameState: estado actual del juego.
        """
        if not gameState.gameOver:
            actions = gameState.getActions(self)
            chosenAction = random.choice(actions)
            gameState.move(chosenAction, self)
        else:
            gameState.changeTurn(self)
```

greedy.py

```
from player import Player
import gameState

class GreedyPlayer(Player):
    """
    Clase para implementar al jugador Greedy. Implementa un algoritmo voraz, que elige la mejor jugada a corto plazo en cada turno.
    """
    def __init__(self, name):
        super().__init__(name)
        self.Qfile = None


    def getMove(self, gameState, *args, **kwargs):
        """
        Función que juega el turno del agente Greedy.
        Se recorren las posibles acciones a realizar y se selecciona la que más puntos le aporte.
        Nunca va a optar por robar una ficha pequeña voluntariamente si existe la posibilidad de colocar alguna otra, ya sea pequeña o
hexagonal.
        A igualdad de puntos, el método escogerá la opción de colocar una ficha pequeña antes que una hexagonal.
        PARAMETER:
            gameState: estado actual del juego.
        """
        if not gameState.gameOver:
            actions = gameState.getActions(self)
            chosenAction = []
            maxScore = -1
```

```
            for action in actions:
                if int(action[4]) > maxScore:
                    maxScore = action[4]
                    chosenAction = action
            gameState.move(chosenAction, self)
        else:
            gameState.changeTurn(self)
```

minimax.py

```
from player import Player
import gameState
from copy import deepcopy
import time

class MinimaxPlayer(Player):
    """
    Clase para implementar al jugador Minimax, con con poda alfa-beta y suspensión en profundidad y en tiempo.
    """

    INFINITE = 10000
    DEPTH = 3

    def __init__(self, name):
        super().__init__(name)
        self.maxTime = 90000 # milisegundos. Tiempo máximo de exploración de cada una de las ramas que cuelgan del nodo raíz
        self.numNodes = 0 # Dato para la obtención de métricas.
        self.startTime = None # Establece el valor de tiempo inicial para determinar el momento en que se
                              # sobrepasa el límite de búsqueda en una rama principal del nodo raíz
        self.Qfile = None

        if self.gotPieceBefore:
            self.bestAction = [['passTurn', None, None, None, 0]]
        else:
            self.bestAction = ['getSmallPiece', None, None, None, 0]


    def generateChildNode(self, gameState, action, playerType):
        """
        Genera un clon del gameState y se ejecuta la acción sobre él.
        PARAMETERS:
            gameState: estado actual del juego que se va a clonar.
            action: acción que se va a ejecutar sobre el nuevo clon del gameState.
            playerType: MAX --> Minimax
                        min --> oponente
        RETURN:
            newGameState: estado del juego resultante.
        """

        newGameState = deepcopy(gameState) # Se clona el gameState
        if playerType == 'MAX':
            player = [p for p in newGameState.players if p.name == 'Minimax'][0]
        else:
            player = [p for p in newGameState.players if p.name != 'Minimax'][0]

        newGameState.move(action, player) # Se ejecuta la acción sobre newGameState
        self.numNodes += 1

        return newGameState


    def alpha_beta(self, gameState, depth=DEPTH, alpha=-INFINITE, beta=INFINITE, playerType='MAX'):
        """
        Función que explora de forma recursiva el árbol MINIMAX con poda ALFA-BETA.
        PARAMETERS:
            gameState: estado actual del juego.
            depth: profundidad de exploración. Por defecto DEPTH.
            alpha, beta: valores a actualizar en cada nodo. Por defecto -INFINITE y INFINITE, respectivamente.
            playerType: MAX --> Minimax
                        min --> oponente
        RETURN:
            alfa/beta: valor actualizado para el nodo actual.
            bestAction: acción seleccionada.
        """
        if depth == self.DEPTH:
            self.startTime = time.time()
        minimaxPlayer = [p for p in gameState.players if p.name == 'Minimax'][0]
        otherPlayer = [p for p in gameState.players if p.name != 'Minimax'][0]
```

```
        if gameState.gameOver or depth == 0: # Nodo terminal alcanzado por fin de partida o por límite de profundidad. El valor que devuelve es la diferencia de puntos entre el jugador y su oponente tras el recuento final de ambos
            return gameState.finalPointCount(minimaxPlayer)[0] - gameState.finalPointCount(otherPlayer)[0], None

        if time.time() > self.startTime + self.maxTime/1000: # Nodo terminal alcanzado por límite de tiempo. Devuelve la acción que mejor perspectivas tiene de las exploradas hasta el momento
            return alpha, self.bestAction

        if playerType == 'MAX':
            # Se expande el árbol a un nuevo nivel de profundidad. Por cada acción posible del gameState se genera un nodo hijo.
            for action in gameState.getActions(minimaxPlayer):
                newGameState = self.generateChildNode(gameState, action, playerType)
                alpha_prev = alpha

                if len(otherPlayer.smallPieces) == 0:
                    newGameState.gameOver = True

                alpha = max(alpha, self.alpha_beta(newGameState, depth-1, alpha, beta, 'min')[0])
                if alpha > alpha_prev and depth == self.DEPTH:
                    self.bestAction = action
                if alpha >= beta:
                    return alpha, self.bestAction # Se poda el árbol a partir de esta rama

            return alpha, self.bestAction

        else: # min
             # Se expande el árbol a un nuevo nivel de profundidad. Por cada acción posible del gameState se genera un nodo hijo.
            for action in gameState.getActions(otherPlayer):
                newGameState = self.generateChildNode(gameState, action, playerType)

                if len(minimaxPlayer.smallPieces) == 0:
                    newGameState.gameOver = True

                beta = min(beta, self.alpha_beta(newGameState, depth-1, alpha, beta, 'MAX')[0])
                if alpha >= beta:
                    return beta, None # Se poda el árbol a partir de esta rama

            return beta, None


    def getMove(self, gameState, *args, **kwargs):
        """
        Función que juega el turno del agente Minimax. Inicia la exploración del árbol MINIMAX para seleccionar la mejora acción y la ejecuta.
        PARAMETER:
            gameState: estado actual del juego.
        """
        if not gameState.gameOver:
            chosenAction = self.alpha_beta(gameState)[1]
            gameState.move(chosenAction, self)
        else:
            gameState.changeTurn(self)
```

qLearner.py

```
from player import Player
import numpy as np
import os
import pickle
from datetime import datetime

class QLearningPlayer(Player):
    """
    Clase para implementar al jugador QLearner. Usa Reinforcement Learning iterativo.
    """

    INITIALQVALUE = 0.5 # Valor con el que se inicializan los Q-values
```

```
    def __init__(self, name, lr=0.4, df=0.8, Qfile=None):
        super().__init__(name)
        self.lr = lr # learning rate
        self.df = df # discount factor
        self.numQUpdated = 0 # Dato para la obtención de métricas. Número de valores de Q actualizados tras aplicar la ecuación de Bellman
        self.game_log = []
        self.nextState = '0'
        self.nextAction = '0'

        self.Qfile = Qfile
        if self.Qfile != None:
            self.q = self.load_Q(self.Qfile)
        else:
            self.q = {} # (state, action)

        self.setQ('0', '0', 0)


    def softmax(self, x):
        """
        Función softmax para convertir una lista de valores numéricos en una distribución de probabilidad.
        PARAMETER:
            x: lista de valores numéricos.
        RETURN:
            players: Distribución de probabilidad. Los valores devueltos están comprendidos entre 0 y 1 y la suma de todos ellos es 1.
        """

        obtiene una distribución de probabilidad, de forma que los valores están  comprendidos entre 0 y 1 y la suma de todos ellos es 1.

        return np.exp(x) / np.sum(np.exp(x), axis=0)


    def setQ(self, state, action, qValue=INITIALQVALUE):
        """
        Función que inicializa todos los Q-values con INITIALQVALUE y se actualizan los Q-values existentes tras aplicarles la ecuación de
Bellman.
        PARAMETERS:
            state, action: clave de Q(state, action)
            qValue: valor que se le asigna a Q. Por defecto INITIALQVALUE.
        RETURN:
            qValue: valor que se le ha asignado a Q.
        """
        #
        qPrevValue = self.getQ(state, action)
        if not qPrevValue: # Nuevo Q-value
            self.q[state, tuple(action)] = qValue
            return qValue
        elif qPrevValue == self.INITIALQVALUE: # Se actualiza Q existente con el valor obtenido de la ecuación de Bellman
            self.q[state, tuple(action)] = qValue
            self.numQUpdated += 1
            return qValue
        elif qValue == self.INITIALQVALUE: # Se intenta inicializar pero ya tenía un valor. Prevalece en Q el valor que tenía anteriormente
            return qPrevValue
        else: # Se actualiza Q existente con el valor obtenido de la ecuación de Bellman
            self.q[state, tuple(action)] = qValue
            return qValue


    def getQ(self, state, action):
        """
        Función que obtiene un Q-value.
        PARAMETERS:
            state, action: clave del Q-value.
        RETURN:
            Valor de Q(state, action).
        """

        return self.q.get((state, tuple(action)))


    def getMove(self, gameState, train=True, *args, **kwargs):
        """
        Función que juega el turno del agente QLearner. Obtiene todas las acciones que el agente puede realizar y ejecuta la mejor de
ellas.
        PARAMETERS:
            gameState: estado actual del juego.
            train: True --> Se trata de un entrenamiento. Se ejecuta en modo exploración.
                   False --> Se ejecuta en modo explotación.
        """

        if gameState.gameOver:
            otherPlayer = [p for p in gameState.players if p.name != 'QLearner'][0]
            reward = gameState.finalPointCount(self)[0] - gameState.finalPointCount(otherPlayer)[0]
            i = 0
```

```
            for log in self.game_log[::-1]:
                if i == 0:
                    self.nextState = '0'
                    self.nextAction = '0'
                # Ecuación de Bellman
                bellman = (1 - self.lr) * self.getQ(log[0], log[1]) + self.lr * (reward + self.df * self.getQ(self.nextState,
self.nextAction))
                self.setQ(log[0], log[1], bellman)
                i += 1
                self.nextState = log[0]
                self.nextAction = log[1]

            gameState.endProperly = True
            self.game_log = []

        else:
            state = gameState.getState(self)
            possibleActions =  gameState.getActions(self)
            qValues = [self.setQ(state, a) for a in possibleActions]

            if train: # Exploración
                softQValues = self.softmax(qValues) # Se ponderan los Q-values. Valores entre 0 y 1 y cuya suma es 1.
                action = possibleActions[int(np.random.choice(len(possibleActions), p=softQValues))]
            else: # Explotación
                maxQ = max(qValues)
                if qValues.count(maxQ) > 1: # Si existe una acción con el valor máximo se escoge una aleatoriamente
                    bestOptions = [i for i in range(len(possibleActions)) if qValues[i] == maxQ]
                    i = np.random.choice(bestOptions)
                else:
                    i = qValues.index(maxQ)
                action = possibleActions[i]
            gameState.move(action, self)
            if not gameState.gameOver:
                self.game_log = self.game_log + [(state, action)]

            gameState.changeTurn(self)


    def save_Q(self):
        """
        Función que guarda la Q-table en un fichero .pkl.
        """

        file_name = datetime.now().strftime('qtable_%Y_%m_%d_%H_%M.pkl')  # fichero .pkl para almecenar la Q-table
        q_dir = 'qfiles'
        if not os.path.exists(q_dir):
            os.mkdir(q_dir)

        print('Se guardan los {} valores de Q en el fichero {}.\n'.format(len(self.q), file_name))
        with open(os.path.join(q_dir, file_name), 'wb') as file:
            pickle.dump(self.q, file)


    def load_Q(self, file_name):
        """
        Función que carga la Q-table desde un fichero.
        PARAMETER:
            file_name: fichero.
        RETURN:
            qSaved: Q-table almacenada en el fichero.
        """
        print('Obteniendo valores de Q del fichero {} para el agente QLearner.......'.format(os.path.join('qfiles', file_name)))
        with open(os.path.join('qfiles', file_name), 'rb') as file:
            qSaved = pickle.load(file)
            print('{} valores de Q leídos.'.format(len(qSaved)))
        return qSaved
```

human.py

```
from player import Player
import gameState

class HumanPlayer(Player):
    """
    Clase para implementar al jugador Human.
    """
    def __init__(self, name):
        super().__init__(name)
        self.Qfile = None


    def getMove(self, gameState, *args, **kwargs):
        """
        Función que implementa una API sencilla para permitir al jugador humano ejecutar su turno.
        PARAMETER:
            gameState: estado actual del juego.
        """
        if not gameState.gameOver:
            actions = gameState.getActions(self)
            while True:
                print ("\nIndica el número de tu jugada:")
                num = 1
                for action in actions:
                    print ("{} - {}".format(num, action))
                    num += 1
                optionMenu = input()
                if int(optionMenu) > 0 and int(optionMenu) <= num:
                    chosenAction = actions[int(optionMenu)-1]
                    break
                else:
                    print("Opción incorrecta")
                    continue
            gameState.move(chosenAction, self)
        else:
            gameState.changeTurn(self)
```

iwoki.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import os
import numpy as np
import time
from datetime import datetime
from utils import drawPlayersOrder
from initializations import initializeAll
from player import Player
from agents.greedy import GreedyPlayer
from agents.random import RandomPlayer
from agents.qlearner import QLearningPlayer
from agents.minimax import MinimaxPlayer
from agents.human import HumanPlayer

NUM_GAMES = 100 # Número de partidas
CHECK_POINT = 10 # Número de partidas para capturar datos generales que informan sobre la evolución de las partidas

def playGame(players, gameSpace):
    """
    Función que controla el turno de los jugadores y finaliza la partida.
    PARAMETERS:
        players: lista de jugadores.
        gameSpace: espacio de juego.
    """
    while True:
        for player in players:
            if player.isTurn:
```

```
                if gameSpace.gameOver:
                    agents[player.name].getMove(gameSpace, train=True)
                    if gameSpace.endProperly:
                        return None
                    gameSpace.endProperly = True
                    break
                elif player.name == 'Human':
                    gameSpace.summary()

                agents[player.name].getMove(gameSpace, train=True)
                player.numTurns += 1
                break


def saveMetrics(file_name, time, games_played, players):
    """
    Función que en el estado de Check Point almacena y muestra datos generales que sobre la evolución de las partidas.
    PARAMETERS:
        file_name: fichero donde se almacena la información.
        time: tiempo desde el anterior check point.
        games_played: número de partidas jugadas desde el inicio hasta el check point.
        players: lista de jugadores
    """
    print('****************************')
    print('        CHECK POINT')
    print('****************************')
    mean_time = time / games_played
    print('Partidas jugadas: {}'.format(games_played))
    print(f'Tiempo medio de las partidas:{mean_time: .2f}s')
    print('Promedio turnos empleados por cada jugador: {}'.format(int(players[0].numTurns / games_played)))
    winsAny = 0
    nonMinimaxScore=0
    for player in players:
        print('Número de partidas que gana el agente {}: {}'.format(player.name, player.numWins))
        winsAny += player.numWins
        if player.name != 'Minimax':
            nonMinimaxScore += player.accumulatedScore
    print('Número de empates: {}'.format(games_played-winsAny))
    if any([p for p in players if p.name == 'QLearner']):
        print('Número de Q-values actualizados: {}'.format(agents['QLearner'].numQUpdated))
        print('Número de Q-values TOTALES en Qfile: {}'.format(len(agents['QLearner'].q)))
    if any([p for p in players if p.name == 'Minimax']):
        print(f'Tiempo medio que tarda Minimax en decidir su movimiento:{(check_time - start_all) / players[0].numTurns: .2f}s')
        print('Promedio de diferencia de puntos por partida: {}'.format(int(agents['Minimax'].accumulatedScore - nonMinimaxScore) /
games_played))
        print('Promedio de nodos generados por partida: {}'.format(int(agents['Minimax'].numNodes / games_played)))

    with open(os.path.join(q_dir, file_name), 'a') as file:
        file.write(str(games_played) +','
        + str(mean_time) +','
        + str(agents['QLearner'].numWins) + ','
        + str(agents['Random'].numWins) + ','
        + str(agents['Greedy'].numWins) + ','
        + str(agents['Minimax'].numWins) + ','
        + str(games_played-agents['Random'].numWins - agents['Greedy'].numWins - agents['QLearner'].numWins - agents['Minimax'].numWins) +
','
        + str(agents['QLearner'].accumulatedScore / CHECK_POINT) + ','
        + str(agents['Random'].accumulatedScore / CHECK_POINT) + ','
        + str(agents['Greedy'].accumulatedScore / CHECK_POINT) + ','
        + str(agents['Minimax'].accumulatedScore / CHECK_POINT) + ','
        + str(agents['QLearner'].numQUpdated) + ','
        + str(len(agents['QLearner'].q)) + '\n')


if __name__=="__main__":
    start_all = time.time()

    # Se crea el dataset para las métricas
    file_name = datetime.now().strftime('iwoki_%Y_%m_%d_%H_%M.csv')  # dataset
    q_dir = 'datasets'
    if not os.path.exists(q_dir):
        os.mkdir(q_dir)
    with open(os.path.join(q_dir, file_name), 'a') as file:
        file.write('numGames,time,QLearnerWins,RandomWins,GreedyWins,MinimaxWins,draws,'
               'scoresQLearner,scoresRandom,scoresGreedy,scoresMinimax,updatedQ,totalQ\n')

    os.system('cls')
    print ("\n-- Bienvenido al iwoki--\n")
    agents = {
    'Random' : RandomPlayer('Random'),
    'Greedy' : GreedyPlayer('Greedy'),
    'QLearner' : QLearningPlayer('QLearner', lr=0.5, df=0.8, Qfile=None),
    'Minimax' : MinimaxPlayer('Minimax'),
    'Human' : HumanPlayer('Human'),
    }
```

```
    players = [agents['Greedy'], agents['Minimax']]

    previous_check_time = start_all
    for gameId in range(NUM_GAMES): # Número de partidas
        start = time.time()
        print('\nPartida número {}:\n'.format(gameId+1))

        gameSpace = initializeAll(players)
        gameSpace.getAllInitialPieces(numSmall=9, numHexagonal=3)

        players = drawPlayersOrder(gameSpace.players)
        print("Empieza la partida el jugador " + players[0].name)
        print("\n-- Se inicia la partida --")
        playGame(players, gameSpace)

        end = time.time()
        print(f'\nDuración de la partida:{end - start: .2f}s = {(end - start)/60:.2f} min = {(end - start)/3600:.2f} hours\n')
        if (gameId+1) % CHECK_POINT == 0:
            check_time = time.time()
            saveMetrics(file_name, check_time - previous_check_time, gameId+1, players)
            if not any([p for p in players if p.name == 'Minimax']): # 'Minimax' no es ninguno de los jugadores
                for player in players:
                    player.accumulatedScore = 0

    if any([p for p in players if p.name == 'QLearner']):
        agents['QLearner'].save_Q()
    end_all = time.time()
    print(f'Tiempo total empleado:{end_all - start_all: .2f}s = {(end_all - start_all)/60:.2f} min = {(end_all - start_all)/3600:.2f}
hours')
```

Anexo IV Resumen de partida entre el autor del iwoki y el agente Minimax

En este anexo se muestran las trazas de la ejecución de una de las partidas entre el autor del juego y el agente Minimax.

En la Figura 31 podemos observar la disposición de las fichas al final de la partida.

```
-- Bienvenido al iwoki--

Se inicializan todas las fichas.
Se inicializan todos los huecos.
Se inicializan los jugadores.
Se crea el espacio de juego.
Se reparten 9 fichas pequeñas y 3 fichas hexagonales a cada jugador.
[Minimax]: Roba 9 fichas pequeñas.
[Minimax]: Roba 3 fichas hexagonales.
[Human]: Roba 9 fichas pequeñas.
[Human]: Roba 3 fichas hexagonales.
Por sorteo, el orden de los jugadores es: Human y Minimax
Empieza la partida el jugador Human

-- Se inicia la partida --

-------------------------------------
Puntuación de Human: 0
Fichas pequeñas de Human: 114+ 113+ 333+ 122+ 222+ 222+ 122+ 223+ 124+
Fichas hexagonales de Human: *41022 *53421 *63035
Testigos blancos: 0
Testigo rojo: False

Puntuación de Minimax: 0
Fichas pequeñas de Minimax: 123+ 114+ 113+ 333+ 122+ 223+ 224+ 124+ 224+
Fichas hexagonales de Minimax: *65103 *54420 *45214
Testigos blancos: 0
Testigo rojo: False
-------------------------------------
[Human]: Coloca la ficha sp20, con orden 222- en el hueco H1_H4_H5.
[Human]: 1 testigo blanco activo.
[Minimax]: Coloca la ficha hp19, con orden 4420*5 en el hueco H1.
-------------------------------------
Puntuación de Human: 0
Fichas pequeñas de Human: 114+ 113+ 333+ 122+ 222+ 122+ 223+ 124+
Fichas hexagonales de Human: *41022 *53421 *63035
Testigos blancos: 1
Testigo rojo: False

Puntuación de Minimax: 0
Fichas pequeñas de Minimax: 123+ 114+ 113+ 333+ 122+ 223+ 224+ 124+ 224+
Fichas hexagonales de Minimax: *65103 *45214
Testigos blancos: 0
Testigo rojo: False
-------------------------------------
[Human]: Coloca la ficha sp28, con orden 114+ en el hueco H1_H6_H7.
[Minimax]: Coloca la ficha hp17, con orden 3*6510 en el hueco H7.
-------------------------------------
Puntuación de Human: 5
Fichas pequeñas de Human: 113+ 333+ 122+ 222+ 122+ 223+ 124+
Fichas hexagonales de Human: *41022 *53421 *63035
Testigos blancos: 1
```

```
Testigo rojo: False

Puntuación de Minimax: 5
Fichas pequeñas de Minimax: 123+ 114+ 113+ 333+ 122+ 223+ 224+ 124+ 224+
Fichas hexagonales de Minimax: *45214
Testigos blancos: 0
Testigo rojo: False
-------------------------------------
[Human]: Coloca la ficha sp15, con orden 113+ en el hueco H1_H2_H3.
[Minimax]: Coloca la ficha sp46, con orden 242+ en el hueco H1_H7_H2.
-------------------------------------
Puntuación de Human: 9
Fichas pequeñas de Human: 333+ 122+ 222+ 122+ 223+ 124+
Fichas hexagonales de Human: *41022 *53421 *63035
Testigos blancos: 1
Testigo rojo: False

Puntuación de Minimax: 15
Fichas pequeñas de Minimax: 123+ 114+ 113+ 333+ 122+ 223+ 124+ 224+
Fichas hexagonales de Minimax: *45214
Testigos blancos: 0
Testigo rojo: False
-------------------------------------
[Human]: Coloca la ficha hp8, con orden 035*63 en el hueco H2.
[Minimax]: Coloca la ficha hp14, con orden *41254 en el hueco H3.
-------------------------------------
Puntuación de Human: 15
Fichas pequeñas de Human: 333+ 122+ 222+ 122+ 223+ 124+
Fichas hexagonales de Human: *41022 *53421
Testigos blancos: 1
Testigo rojo: False

Puntuación de Minimax: 19
Fichas pequeñas de Minimax: 123+ 114+ 113+ 333+ 122+ 223+ 124+ 224+
Fichas hexagonales de Minimax:
Testigos blancos: 0
Testigo rojo: False
-------------------------------------
[Human]: Coloca la ficha sp45, con orden 223+ en el hueco H2_H10_H3.
[Minimax]: Coloca la ficha sp27, con orden 311+ en el hueco H3_H10_H11.
-------------------------------------
Puntuación de Human: 20
Fichas pequeñas de Human: 333+ 122+ 222+ 122+ 124+
Fichas hexagonales de Human: *41022 *53421
Testigos blancos: 1
Testigo rojo: False

Puntuación de Minimax: 23
Fichas pequeñas de Minimax: 123+ 114+ 333+ 122+ 223+ 124+ 224+
Fichas hexagonales de Minimax:
Testigos blancos: 0
Testigo rojo: False
-------------------------------------
[Human]: Coloca la ficha sp19, con orden 214- en el hueco H3_H12_H4.
[Human]: 2 testigos blancos activos.
[Minimax]: Coloca la ficha sp41, con orden 122+ en el hueco H2_H9_H10.
-------------------------------------
Puntuación de Human: 22
Fichas pequeñas de Human: 333+ 122+ 222+ 122+
Fichas hexagonales de Human: *41022 *53421
Testigos blancos: 2
Testigo rojo: False

Puntuación de Minimax: 26
Fichas pequeñas de Minimax: 123+ 114+ 333+ 223+ 124+ 224+
Fichas hexagonales de Minimax:
Testigos blancos: 0
Testigo rojo: False
```

```
-------------------------------------
[Human]: Coloca la ficha sp17, con orden 122+ en el hueco H2_H7_H8.
[Minimax]: Coloca la ficha sp7, con orden 421- en el hueco H7_H19_H8.
[Minimax]: 1 testigo blanco activo.
-------------------------------------
Puntuación de Human: 25
Fichas pequeñas de Human: 333+ 122+ 222+
Fichas hexagonales de Human: *41022 *53421
Testigos blancos: 2
Testigo rojo: False

Puntuación de Minimax: 29
Fichas pequeñas de Minimax: 123+ 114+ 333+ 223+ 224+
Fichas hexagonales de Minimax:
Testigos blancos: 1
Testigo rojo: False
-------------------------------------
[Human]: Coloca la ficha hp15, con orden 22*410 en el hueco H8.
[Minimax]: Coloca la ficha sp13, con orden 321- en el hueco H8_H20_H21.
[Minimax]: 2 testigos blancos activos.
-------------------------------------
Puntuación de Human: 30
Fichas pequeñas de Human: 333+ 122+ 222+
Fichas hexagonales de Human: *53421
Testigos blancos: 2
Testigo rojo: False

Puntuación de Minimax: 31
Fichas pequeñas de Minimax: 114+ 333+ 223+ 224+
Fichas hexagonales de Minimax:
Testigos blancos: 2
Testigo rojo: False
-------------------------------------
[Human]: Coloca la ficha sp24, con orden 333- en el hueco H2_H8_H9.
[Human]: Roba 1 ficha hexagonal.
[Human]: 0 testigos blancos activos.
[Minimax]: Coloca la ficha sp34, con orden 422- en el hueco H8_H21_H9.
[Minimax]: Roba 1 ficha hexagonal.
[Minimax]: 0 testigos blancos activos.
-------------------------------------
Puntuación de Human: 30
Fichas pequeñas de Human: 122+ 222+
Fichas hexagonales de Human: *53421 *36242
Testigos blancos: 0
Testigo rojo: False

Puntuación de Minimax: 33
Fichas pequeñas de Minimax: 114+ 333+ 223+
Fichas hexagonales de Minimax: *40346
Testigos blancos: 0
Testigo rojo: False
-------------------------------------
[Human]: Coloca la ficha sp5, con orden 212- en el hueco H8_H19_H20.
[Human]: 1 testigo blanco activo.
[Human]: Testigo rojo activo. Última ficha pequeña.
[Minimax]: Coloca la ficha sp21, con orden 232- en el hueco H6_H18_H7.
[Minimax]: 1 testigo blanco activo.
-------------------------------------
Puntuación de Human: 30
Fichas pequeñas de Human: 222+
Fichas hexagonales de Human: *53421 *36242
Testigos blancos: 1
Testigo rojo: True

Puntuación de Minimax: 34
Fichas pequeñas de Minimax: 114+ 333+
Fichas hexagonales de Minimax: *40346
Testigos blancos: 1
```

```
Testigo rojo: False
-------------------------------------
[Human]: Roba 1 ficha pequeña.
[Human]: 2 testigos blancos activos.
[Human]: Testigo rojo desactivado.
-------------------------------------
Puntuación de Human: 30
Fichas pequeñas de Human: 222+ 112+
Fichas hexagonales de Human: *53421 *36242
Testigos blancos: 2
Testigo rojo: False

Puntuación de Minimax: 34
Fichas pequeñas de Minimax: 114+ 333+
Fichas hexagonales de Minimax: *40346
Testigos blancos: 1
Testigo rojo: False
-------------------------------------
[Human]: Coloca la ficha sp44, con orden 222- en el hueco H7_H18_H19.
[Human]: Roba 1 ficha hexagonal.
[Human]: 0 testigos blancos activos.
[Human]: Testigo rojo activo. Última ficha pequeña.
[Minimax]: Coloca la ficha hp4, con orden 40346* en el hueco H5.
-------------------------------------
Puntuación de Human: 30
Fichas pequeñas de Human: 112+
Fichas hexagonales de Human: *53421 *36242 *23465
Testigos blancos: 0
Testigo rojo: True

Puntuación de Minimax: 34
Fichas pequeñas de Minimax: 114+ 333+
Fichas hexagonales de Minimax:
Testigos blancos: 1
Testigo rojo: False
-------------------------------------
[Human]: Roba 1 ficha pequeña.
[Human]: 1 testigo blanco activo.
[Human]: Testigo rojo desactivado.
-------------------------------------
Puntuación de Human: 30
Fichas pequeñas de Human: 112+ 122+
Fichas hexagonales de Human: *53421 *36242 *23465
Testigos blancos: 1
Testigo rojo: False

Puntuación de Minimax: 34
Fichas pequeñas de Minimax: 114+ 333+
Fichas hexagonales de Minimax:
Testigos blancos: 1
Testigo rojo: False
-------------------------------------
[Human]: Coloca la ficha sp29, con orden 212+ en el hueco H5_H14_H15.
[Human]: Testigo rojo activo. Última ficha pequeña.
[Minimax]: Coloca la ficha sp40, con orden 141+ en el hueco H1_H3_H4.
[Minimax]: Testigo rojo activo. Última ficha pequeña.
-------------------------------------
Puntuación de Human: 34
Fichas pequeñas de Human: 112+
Fichas hexagonales de Human: *53421 *36242 *23465
Testigos blancos: 1
Testigo rojo: True

Puntuación de Minimax: 41
Fichas pequeñas de Minimax: 333+
Fichas hexagonales de Minimax:
Testigos blancos: 1
Testigo rojo: True
```

```
-------------------------------------
[Human]: Coloca la ficha hp5, con orden 53421* en el hueco H4.
[Minimax]: Roba 1 ficha pequeña.
[Minimax]: Testigo rojo desactivado.
[Minimax]: Coloca la ficha sp30, con orden 133+ en el hueco H1_H5_H6.
[Minimax]: Testigo rojo activo. Última ficha pequeña.
-------------------------------------
Puntuación de Human: 42
Fichas pequeñas de Human: 112+
Fichas hexagonales de Human: *36242 *23465
Testigos blancos: 1
Testigo rojo: True

Puntuación de Minimax: 45
Fichas pequeñas de Minimax: 333+
Fichas hexagonales de Minimax:
Testigos blancos: 2
Testigo rojo: True
-------------------------------------
[Human]: Coloca la ficha hp3, con orden *24263 en el hueco H14.
[Minimax]: Coloca la ficha sp48, con orden 333+ en el hueco H5_H15_H16.
-------------------------------------
Puntuación de Human: 45
Fichas pequeñas de Human: 112+
Fichas hexagonales de Human: *23465
Testigos blancos: 1
Testigo rojo: True

Puntuación de Minimax: 51
Fichas pequeñas de Minimax:
Fichas hexagonales de Minimax:
Testigos blancos: 2
Testigo rojo: True
-------------------------------------
[Human]: Coloca la ficha sp26, con orden 121+ en el hueco H14_H28_H29.
-------------------------------------

Puntuaciones finales de la partida:

Human: 47 puntos

Minimax: 51 puntos

GANADOR:  Minimax

-- Fin de la partida --
```

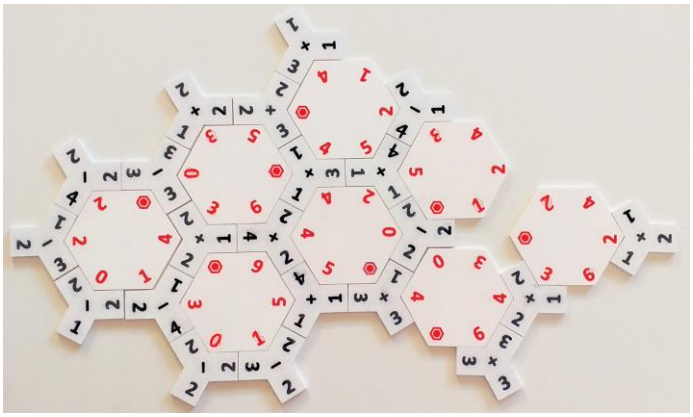


Figura 31. Imagen final de la partida entre el autor del iwoki y el agente Minimax.

Fuente: elaboración propia.

Performance study of intelligent agents playing the iwoki game

Santiago Videgain Diaz

Universidad Internacional de la Rioja, Logroño (España)

February, 13th 2020

ABSTRACT

Iwoki maths is an abstract board game for laying tiles. It combines the calculation of simple mathematical operations with the spatial perception of two-dimensional objects. A series of intelligent agents with different reasoning and decision capacities will be developed based on different artificial intelligence techniques applied to game theory. Moreover, *iwoki maths* will be used as a research tool on which different technologies and configurations will be applied. After the implementation of the game and the agents, their capabilities will be tested by playing games with each other. One of the agents will be evaluated in a real game playing against the author of the game. The experimental results will ratify conclusions already known at a theoretical level and will expose new knowledge that could be the basis for future research.

KEYWORDS

artificial intelligence, intelligent agent, game theory, iwoki, Minimax.

I. INTRODUCTION

It is well known that board games can provide a lot of fun. They also reinforce creativity and teach decision making, as well as respect for rules.

The *iwoki maths* (hereinafter iwoki) is a game that combines this benefits with its educational character. It reinforces the capacity for analysis and activates the cognitive functions to put into practice the best strategy throughout the games.

It also trains the spatial perception of two-dimensional objects to make way for the calculation of simple mathematical operations.

Iwoki is an abstract board game that is currently in the prototype phase, having been tested in playful environments. This game was a finalist in the Meeple Factory prototype competition, organized nationally by the CrossOver Cultural Association. Consequently, it was exhibited at the FicZone event, at the Armilla Trade Fair (Granada, Spain), during the last weekend of April 2019.

An example of a real iwoki game can be seen in Fig. 1.

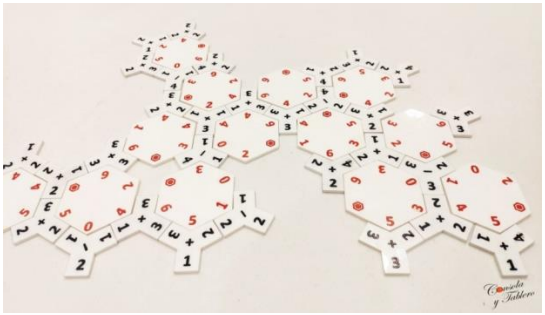


Fig. 1. Example of an iwoki game [1].

This article discusses the development of various agents who compete with each other by playing iwoki.

In order to limit the scope of this work, the study is carried out on a simplified version of the iwoki, where only two players enter each game.

There will be two agents with whom to engage in confrontation:

Agent 1 will choose its moves in a **Random** way.

Agent 2 will act as if it were a novice human player. That is, following a basic strategy that allows it to make only short-term decisions. It will follow a **Greedy** method.

To face these, **Agent 3** will play according to the strategies of a very experienced human player. It will use the **Minimax** method. It is necessary for this case that the small pieces of the players remain visible at all times, because it does not make sense to talk about the Minimax search method if the information is not perfect. Likewise, another variation is established so that it is also deterministic: the agent will know the order of the pieces that have not been distributed yet, to take it into account in case in his turn he chooses to get one.

Additionally, **Agent 4** will act as an interface to allow a **Human** player to face Agent 3.

In this work, a double purpose is pursued:

On the one hand, it is intended to study how effective Agent 3 is in relation to the first two, depending on the games he manages to win against his opponents. In addition, a comparison between instances of Agent 3 will be made to obtain its best configuration, being executed with different parameters.

On the other hand, it will be evaluated if Agent 3 is able to beat the human player who created the iwoki, who can be considered an expert in the game.

II. STATE OF THE ART

This article is framed in a global way in the game theory.

Game theory is based on the analysis of the behavior of various agents, focused on pursuing a goal and interacting with each other according to their strategies. The agents obtain a positive or negative

gain according to the success or failure of the interactions carried out. The scope of game theory reaches the economy, history, politics, international relations, military strategy, etc. [2].

According to Aguilar, P. [3], we can classify games in different ways, according to some of their characteristics:

- Depending on the number of players, the game can be with or without an opponent:

In a game without an opponent, only one player is involved. The most used methods for its implementation are the brute force method without heuristics, or, if we want to take into account the time and memory consumption as determining factors, the heuristic search by means of A* or IDA*. Examples of games without an opponent are 8-puzzle, solitaire, etc.

In games with an opponent two or more agents compete for a common goal. Examples of games with an opponent are 3-in-a-row, chess, go, checkers, etc. Other examples of where there is more than one opponent are poker or monopoly.

- A relevant feature is the order of the moves or actions. It may be that the players' turns are alternate or random. They may also be simultaneous, in which the players make their decisions at the same time (rock, paper or scissors), or sequential (Parcheesi).

- The knowledge of each player about the rest catalogues the game as one of perfect information, when it does not affect randomness and the information of each player is shared for the others (checkers), or of imperfect information, where randomness does intervene and each player hides information from the rest (black jack).

- Determinism refers to the intervention of randomness. Thus, a game can be deterministic if there is no room for randomness, or non-deterministic (or stochastic) if randomness intervenes in any of the above characteristics.

- Zero-sum or non-zero-sum games. Von Newman and Morgenstern [4] define the concept of zero-sum games with an opponent, in which everything a player manages to win corresponds to what his opponent loses. In other words, the sum of what one wins and the other loses is zero. An example of a zero-sum game is an individual tennis match, where each player's point is detrimental to the opponent. However, in a doubles tennis match, the points scored by your partner are used by you. This case is an example of non-zero-sum play [5].

- Cooperative games, in which participants share common interests and act for the benefit of such ends without competing with each other. In non-cooperative games, each player is concerned only with his own interests [6].

- John Forbes Nash, Nobel Prize in Economics (1994) for his research on game theory and negotiation processes, adds a new concept for non-zero-sum cases. This is the so-called Nash Equilibrium, a situation in which none of the players is interested in modifying their individual strategy so as not to be disadvantaged, knowing their opponents' one [7].

- Although it will not be applicable in the development for the iwoki agents, because it is considered to have perfect information, another concept to take into account within the context of the game theory is the so-called Bayesian Game. It considers as random variables the unknown information of the game. Bayes' theorem is applied by establishing probabilities about concrete states of the game based on previously known information, after a series of previous actions carried out by the opponents. Bayesian Game, therefore, makes sense for games with imperfect information [8].

Given the nature of iwoki, the characteristics and properties that must be taken into account when implementing the agents are as follows:

- It is a game with an opponent in which two agents are involved.

- The rules of the game are the same for both players and known by both.

- The opponents will play their turn in a sequential and alternative way from the beginning to the end of the game.

- It is uncooperative, since each agent looks after his own interests and has no common objectives.

- This is a zero-sum game. In other words, everything a player wins for his own benefit is lost by the other player in the same proportion, and vice versa.

- We may also consider the game as deterministic or not, according to certain modifications necessary for the implementation of a particular agent.

Minimax

For two-player games in which both players try to win and where one player's actions depend decisively on the other's, we need one of the adversarial search algorithms.

Minimax is the basic opponent search algorithm, which has been traditionally implemented from perfect information board games [9].

Basically, it means that each player chooses the best move for himself, keeping in mind that the opponent has chosen the most harmful option for his opponent.

For each state of the game the algorithm generates a complete search tree, applies to each terminal node a utility function (also called evaluation function) and performs a propagation of those values to the root node. As illustrated in Fig. 2, the Max nodes will take into account the maximum values of their successors and the Min, the minimum. At the end of the process, the root node will have visibility of which is the most promising option.

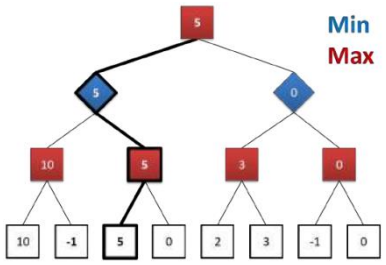


Fig. 2 Minimax search tree with optimal strategy [9].

The main problem with this method is that the tree increases exponentially with depth and with the branching factor of the tree. That is, with the number of simulated movements.

The order of complexity in time is: $O(a^b)$.

The order of complexity in space is: $O(ab)$.

Being:

a: Tree branching factor.

b: Tree depth (Number of plys scanned).

For example, in the case of chess, if we interpret that it has an average branching factor of 30 nodes (moves on each position) and 80 moves are made throughout the game, 30^{80} nodes will be explored. This is more than the number of atoms in the universe.

The complexity in time and space makes a tree that contemplates all the possibilities in each move unfeasible.

As a rule, the applications of the Minimax algorithm finish the search at a certain depth. The nodes of that depth become terminals and use a state evaluation function that determines a heuristic. This technique is called **Minimax** with **suspension**.

It is also possible to limit the search by run time. The root node will stay with the best option found until the maximum time is reached.

Applying these modifications on the Minimax algorithm will not be able to guarantee that the agent will find the optimal play, but it will be reliable if the heuristics are applied properly.

Following these modifications to solve the problem of complexity, there are different techniques to improve Minimax [3]:

Alpha-Beta pruning:

Alpha-Beta pruning reduces the number of nodes to be explored by avoiding passing through nodes in the tree that do not influence the decision of the agent. As it can be seen in Fig. 3, when a node does not provide a better value than the explored one until that moment, the branch is pruned [10].

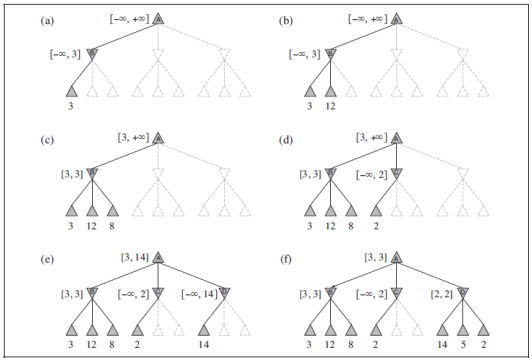


Fig. 3. Construction of a Minimax tree with Alpha-Beta pruning [11].

The following are variants of the Alfa-Beta pruning:

Aspiration Search:

It is a modification of the Alpha-Beta pruning in which a different range is used than the (-∞, +∞). It uses a narrower window based on the previous search. If the result is a value within that window you will have saved time. However, if the search fails, you should widen the window and search again.

Pruning for uselessness:

This is another variant of Alpha-Beta pruning. In this case, the node to be evaluated does not have to be worse or the same as the previous one. If it is not better enough, the branch will also be pruned. That is, even if the node contributes a better value than those seen until that moment, if there is no substantial improvement, that branch can be dispensed with.

The threshold above which one value is considered to be sufficiently better than another must be determined.

Waiting-for-rest method:

It aims to avoid the horizon effect, which occurs when the tree is limited to a fixed depth and it is not known whether the decision taken is the right one or would have been different with a greater depth.

It is about evaluating the state of the game once a node has been evaluated, so that if it changes drastically, the next node should also be evaluated.

Secondary search:

It also fights the horizon effect. It consists in making an additional search to the Minimax to evaluate if the movement choice with Minimax has been good. In this way, if it is verified that later on it does not improve any of the possibilities, another path should be chosen.

Heuristic continuation:

This is another method of avoiding the horizon effect. Once the tree is built to the predefined depth level, instead of selecting the best movement, certain heuristics will indicate to the algorithm that it should explore to a greater depth. For example, in chess, if the king is in danger, the critical situation will probably require access to greater depth in the tree.

Book moves:

This is the name given to the predefined moves that correspond to certain movements in the game, such as at the beginning or end.

Biased search:

In this case the branching factor changes to the benefit of the most promising moves. Thus, if a node is known to offer unoptimistic information, it is not worth creating the search tree from it.

Progressive descent:

When it is time that sets the search standard and not the depth of the tree, a decision must be made within a set time interval. As you can see in Fig. 4, all the nodes are run in amplitude until the time stamp is completed. At that point, the best choice of the last level fully scanned will be returned.

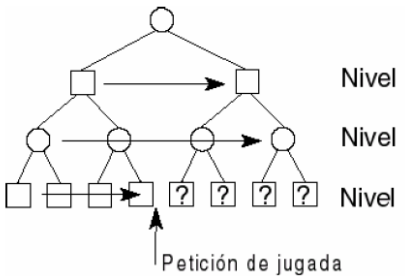


Fig. 4. Generating a search tree with a progressive descent [3].

Null-Move Forward Pruning:

It involves giving the opponent a turn ahead. If the situation in the node is good enough, i.e., Alpha is bigger than Beta, it can be assumed that Alpha will remain bigger in successive nodes, so that branch is pruned. It allows to reduce significantly the number of branches, even at the risk of losing information that could be relevant [12].

A variant is the Verified Null-Move Forward Pruning. In this case, no null movement is made, but the depth of the search is reduced [12].

Megamax algorithm:

This is one of a compact version of the Minimax. The Minimax implementation uses a subroutine for the Max player and another for the Min. Instead, Megamax uses the negated score of the next one:

$$Max(a, b) = - Min(a, b)$$

Alpha-Beta pruning can also be applied.

MegaScout algorithm:

This is an improvement on the Megamax that is optimal when the nodes are ordered. As it assumes that the first node is the best, it will produce more pruning.

SSS* algorithm:

This algorithm is another alternative to Alpha-Beta pruning. It simplifies even more the search space, since it performs pruning in the same nodes that the Alpha-Beta algorithm would prune, but it also does it in other nodes. It is based on the use of sub-trees of the search tree [13].

As a disadvantage, it generates an important spatial complexity.

Scout algorithm:

It's another pruning algorithm. Its strength is that it is more efficient in computational terms than the SSS* algorithm, since it is based on the verification of inequalities. In this case the pruning is done on branches that do not lead to a better solution [14].

We have mentioned some types of algorithms used to model games of various characteristics. We have focused on the most appropriate ones for the agent that includes intelligent behavior.

The Minimax agent with Alpha-Beta suspension and pruning will present reasoning similar to the human with high experience in the game. The maximum time will be applied to the in-depth search of each of the branches of the root node. This, as well as the depth of the tree, will be parameters to be considered experimentally.

III. Objectives and Methodology

This study can be interpreted in two ways:

- To know which implementations are most appropriate for agents who play iwoki in order to win and to evaluate which configurations optimize results.

- On the other hand, to provide knowledge in the field of artificial intelligence applied to games, using iwoki as a research tool. It will allow us to compare different agents, establish certain conclusions about them and explore new lines of research.

After a series of confrontations between the agents, conclusions will be drawn as to how much more effective each one is in relation to his rivals, according to the number of games he manages to win and how many points of difference he is able to obtain.

The experiment will follow the following steps:

1. The agents Random and Greedy will be the rivals of the Minimax, in a modification of the game that contemplates perfect information, since, as we have commented previously, for the Minimax agent it is necessary that all the chips are visible by both players.

2. A comparison will be made between different instances of the best of the agents (predictably the best will be the Minimax, which takes into account the strategies of an expert human player). Certain parameters will be determinated, such as the depth level of the search tree or the maximum time of exploration of the branches.

3. The author of the game will test the Minimax agent with its best configuration in a series of Minimax vs. Human games.

The following methodology will be used:

We start by modelling the iwoki so that it can represent all the content of the game in an abstract way.

Likewise, an implementation of the methods that the agents will have available in the game is carried out, which will constitute the game space.

Next, the four agents will be developed: one will play randomly, another one by the Greedy method, a third one will do it using a Minimax search algorithm with suspension and Alpha-Beta pruning and, finally, an agent that provides an API so that a human can measure the effectiveness of the previous ones.

Once the implementation is completed, the agents will play against each other in two-player games. Each game may have a history of each and every one of the moves that the agents make in their turns. This will allow to make an exhaustive pursuit of the evolution of the game and even to be able to make a simulation in the physical version of the iwoki. This way it will be possible to contrast the actions of the agents with what a human would have decided to do in each turn.

The number of games played by the agents will be enough to establish a series of firm conclusions regarding the nature of the agents.

The knowledge acquired in all the previous steps will allow identifying new ways to go deeper into the study of the implemented solutions, as well as to establish new lines of research.

IV. Contribution

Once the components of the game space have been implemented, the detail of the agents is described as follows.

Agent Random

This agent chooses his moves randomly. It gets from the game space all the possible actions to do in its turn and chooses randomly one of them, without taking into account how good or bad the move is.

Agent Greedy

It emulates the playfulness of a human novice. He implements the simple strategy of choosing the option that gives him the most points immediately, regardless of how easy the move leaves his opponent.

It gets from the game space all the possible actions to take in its turn. Then he selects the best one, taking into account the previous strategy.

In case of different options that give it the same points, its preference will be the following one:

1. Place a small piece on the subtraction side. This increases the white indicator, which is good for getting a hexagonal piece.

2. Place a small piece on the addition side. In any case, placing a small piece brings the player closer to the end of the game.

3. Place a hexagonal piece.

4. Take a piece. This does not give it any points, so it only ultimately chooses to take this action. Obviously, if it has just taken a piece, the option will be to pass the turn to the other player, as specified by the iwoki rules.

Agent Minimax

It applies the strategies that a highly experienced human player has. He has to anticipate his opponent's moves to know the possible future states of the game, from the state he is in to a horizon (depth). To do so, in each turn it must generate a Minimax search tree with Alpha-Beta pruning and suspension in depth and time.

To get the best move in the current turn:

1. The alpha_beta function is invoked, that is executed in a recursive way for the Max and Min nodes, from the root node to the leaf nodes. As it is explored, new nodes are generated following these steps:

a. It gets from the game space all the possible actions to be carried out.

b. For each of them, a new lower level node is created, invoking a method that:

- Makes a clone of the game space.

- Executes the action on the new game space. This is the new game state associated to the generated node.

2. The tree is branched out recursively until a leaf node is found in two different ways:

- Reaching the established depth level.

- Reaching the end of the game in the corresponding branch without having reached the depth level.

The utility function is found by subtracting the scores after the final point count.

3. Undoing the recursion updates the values of Alpha and

Beta until the root node is given a value. At the same time, it allows to prune the branches that will not present better values than those found so far.

The maximum time for deep searching along each of the main branches of the root node is determined.

For example, for a game state where the Minimax agent has 6 possible actions and the maximum time set is 10 seconds, if any of those 6 branches has not been fully explored in 10 seconds, it will propagate the best option found so far and follow it by the next branch of the root node. This limits the total search in that turn to a maximum of 60 seconds.

As it is a search with suspension, the algorithm does not guarantee to find the best action in each turn, but establishing some suitable parameters of time and depth, as it will be seen in the following points, the results will be able to be satisfactory.

Human Agent

It implements an interface that allows a human player to face any of the other agents.

The composition of the items in the experiment is explained as follows.

For the correct interpretation of the results, the first thing that is done is a 100-game match between the Greedy and Random agents. This way we can have a reference of how good the first one is with respect to the second one when other agents compete against both.

In this experiment, a modification of the iwoki will be used in order to make it deterministic and information perfect. To this purpose:

- Small and hexagonal pieces that have not yet been distributed will not be hidden. This way the agent will know in advance which piece he will get if he chooses this action in his turn and, in the same way, he will also know which piece his opponent could get in his case. In this way, the agent is deterministic, eliminating completely the randomness once the pieces have been distributed and the game is started.

- The players' small pieces, as well as the hexagonal ones, will remain visible at all times. Thus the game will be one of perfect information.

Minimax will play games with the agents Random and Greedy. It will use different configurations according to the hyperparameters that will be used, which are:

- **Search tree depth**: 1, 2, 3 and 4 level values will be used. For depth 1, 2 and 3, 100 games will be executed, which will allow evaluating the results and drawing certain conclusions. Due to the high computational cost, only 10 games will be executed when depth level 4 is applied.

- **Maximum search time on each of the main branches of the root node** (*maxTime*): A range of values from 0.5 to 90 seconds will be applied. As we will see in the next section, it only makes sense to apply this hyperparameter in games against the Greedy agent under depth 3.

After playing all of the above games, the two Minimax settings that are likely to be optimal for playing against a human player will be determined so as to balance their effectiveness with the time it takes to execute their turn.

Minimax agent will play 20 games against itself applying these two settings to determine definitively which one is the most appropriate.

This will be the one used by the agent to play 5 games against the creator of the iwoki.

V. Results

First of all, agents Greedy and Random are facing off in 100,000 games. The result can be seen in Table I.

TABLE I

RANDOM VS. GREEDY GAME RESULTS

Wins			Game average	
Greedy	Random	Draws	Duration	Dif. puntos
100	0	0	0.16 seg	9.73

Once the items detailed in the previous section have been executed, the results obtained can be seen in Table II.

TABLE II

GAME RESULTS OF MINIMAX VS. RANDOM AND GREEDY

		Parameters Minimax			Wins			Game average				
Vs		DEPTH	MAX_TIME	Num. Games	Minimax	Rival	Draws	Duration	Dif. Scores	Turns	Decision time	Nodes
Minimax	Random	1	-	100	100	0	0	2.97 sec	24.50	13	0.22 sec	339
Minimax	Random	2	-	100	100	0	0	27.73 sec	26.94	14	1.97 sec	3,498
Minimax	Random	3	-	100	100	0	0	3.98 min	26.50	14	16.13 sec	44,200
Minimax	Random	4	-	10	10	0	0	25.58 min	27.20	13	1.94 min	278,857
Minimax	Greedy	1	-	100	47	49	4	3.87 sec	0.70	12	0.30 sec	455
Minimax	Greedy	2	-	100	71	25	4	44.81 sec	5.29	12	3.54 sec	6,315
Minimax	Greedy	3	-	100	90	8	2	11.65 min	7.50	13	50.27 sec	140,505
Minimax	Greedy	4	-	10	10	0	0	48.45 min	9.67	12	3.14 min	548,321
Minimax	Greedy	3	0.5 sec	100	46	49	5	9.81 sec	-1.05	12	0.77 sec	1,719
Minimax	Greedy	3	1 sec	100	41	57	2	15.20 sec	-1.88	12	1.18 sec	2,800
Minimax	Greedy	3	5 sec	100	49	43	8	58.39 sec	0.69	13	4.43 sec	11,563
Minimax	Greedy	3	10 sec	100	48	52	0	1.89 min	-0.82	13	8.52 sec	13,545
Minimax	Greedy	3	20 sec	100	60	28	12	3.21 min	1.36	13	13.96 sec	37,979
Minimax	Greedy	3	30 sec	100	60	28	12	4.41 min	3.28	13	19.70 sec	53,063
Minimax	Greedy	3	40 sec	100	56	32	12	6.03 min	5.20	13	27.67 sec	51,473
Minimax	Greedy	3	50 sec	100	76	20	4	6.72 min	7.80	13	29.90 sec	88,872
Minimax	Greedy	3	60 sec	100	60	38	2	7.93 min	6.94	14	33.52 sec	97,015
Minimax	Greedy	3	70 sec	100	82	16	2	8.00 min	9.17	13	36.67 sec	97,429
Minimax	Greedy	3	80 sec	100	80	19	1	9.51 min	10.70	14	39.89 sec	118,672
Minimax	Greedy	3	90 sec	100	95	1	4	9.85 min	16.60	13	44.43 sec	122,166

In the graphs below we see the progress of the average values of

- Duration of the games.
- Difference of scores.
- Waiting time to make a decision in each turn.
- Number of nodes generated in the tree search.

A. Minimax vs. Random

As Random agent does not get a single victory against Minimax, it does not make sense to make confrontations by modifying the *maxTime* hyperparameter between these two, as it would not provide relevant information.

The average values while the depth increases can be seen in Fig. 5.

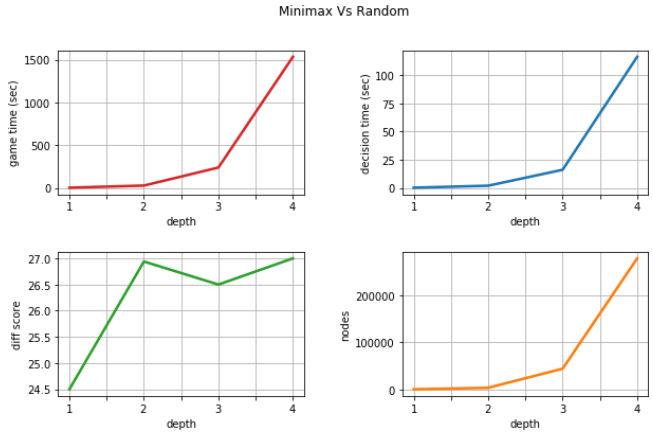


Fig. 5. Average values in Minimax vs. Random games by increasing *depth*.

B. Minimax vs. Greedy

For this case, Fig. 6 shows the evolution of the average values of the items while increasing their depth.

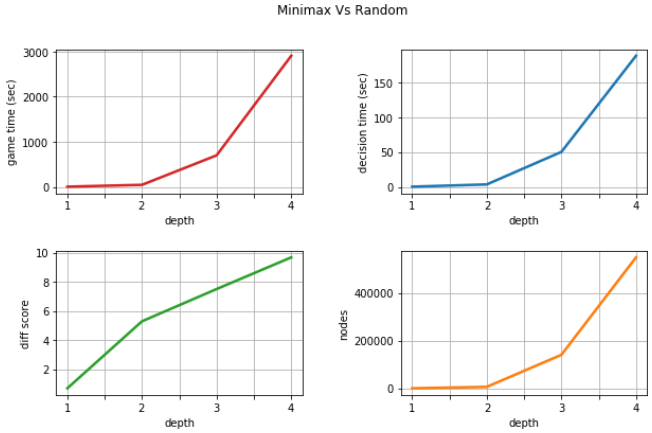


Fig. 6. Average values in Minimax vs. Greedy games when increasing *depth*.

We can see in Table II that the evolution of the games won by Minimax against Greedy is 47%, 71%, 90% and 100%, in games with depths 1, 2, 3 and 4, respectively. It can be seen graphically in Fig. 7.

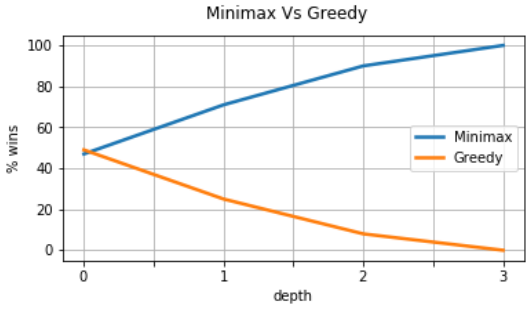


Fig. 7. Minimax wins against Greedy by increasing *depth*.

When the parameter that is increased is the *maxTime*, the average values of the items change according to the graphs in Fig. 8.

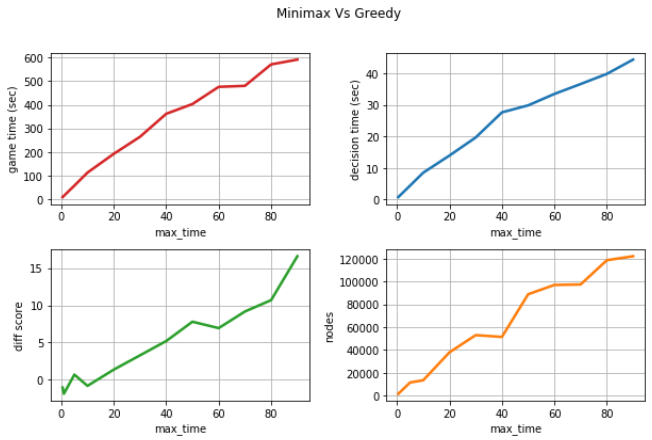


Fig. 8. Average values in games Minimax vs. Greedy by increasing *maxTime*.

C. Minimax vs. Minimax

The experiment takes place between two Minimax agents with different configurations:

- Minimax agent 1: *depth = 3* and unlimited timeout.
- Agent Minimax 2: *depth = 3* and *maxTime = 90 seconds*.

The result of the 20 games is shown in Table III.

TABLE III
GAME RESULTS BETWEEN TWO MINIMAX AGENTS WITH DIFFERENT CONFIGURATIONS

Wins			Game average	
Minimax 1	Minimax 2	Draws	Duration	Dif. Scores
10	8	2	22.19 min	2.50

D. Minimax vs. Human

Five games are executed between the iwoki author and the Minimax agent configured with *depth= 3* and *maxTime = 90 seconds*. The results can be seen in Table IV.

TABLE IV
FINAL SCORES OF MINIMAX VS. HUMAN GAMES

	Scores	
	Minimax	Human
Game 1	48	43
Game 2	51	47
Game 3	59	54
Game 4	51	45
Game 5	65	62

It is checked during the game that the waiting time measured in each turn corresponds to that obtained in the games played against Greedy.

VI. DISCUSSION

The following is an assessment of the results obtained in the previous section

First of all, we should consider the matches between agents Greedy and Random. As expected, based on the results presented in Table I, it can be confirmed that

Agent Greedy is vastly superior to Random.

The results shown in Table II provide a number of general conclusions:

- On the one hand, the number of turns that are executed on average is irrelevant, since in all cases it is similar and independent of the specified parameters.

- In addition, it is observed that whatever the value of the hyperparameter *depth* is, the Minimax agent always beats the Random with a big difference of points. We can conclude that Random is far below the level of Minimax.

- As expected, Minimax games with *depth = 1* against Greedy are very balanced. We can see that the number of victories of each one is very similar (47 to 49) and that there is only 0.7 points difference on average. It is confirmed in an experimental way something that was known on a theoretical level:

The Minimax search using a tree of depth 1 is equivalent to the Greedy search.

It is worth emphasizing the information seen in Fig. 7 about the evolution of the games won by Minimax versus Greedy, while increasing the search depth. With depths of 1, 2, 3 and 4, Minimax wins 47%, 71%, 90% and 100% of games, respectively. The difference in scores between the two grows in a linear trend, as can be seen in Fig. 6.

As a counterpart, increasing the depth means increasing exponentially the number of nodes generated in the search tree and, therefore, the time that the agent takes to decide the action to be carried out. This can be seen in a graphic way in Fig. 6.

It is confirmed that the complexity of the Minimax algorithm grows in an exponential way in time and space as the depth of the search tree increases.

For a match between the best Minimax configuration and a human, *depth = 4* is a value that implies too much time waiting to decide the best move, even though the winning percentage against Greedy is 100%. That's why we have chosen to run Minimax games with different *maxTime* values only against Greedy with *depth = 3*.

The decision time to be improved is 50.27 seconds (Minimax vs. Greedy with *depth = 3*), without losing the effectiveness of more than 90% of won games (9 wins and 2 draws).

The average values follow a linear growth trend as the *maxTime* parameter is increased, as shown in Fig. 8.

In Table II we see that when *maxTime = 90 seconds* the percentage of wins of Minimax versus Greedy is reached and the time it takes the agent to decide his move in each turn is 44.43 seconds. That is, the time is reduced by 11.62%. While the experiment seems to have improved the win percentage, a much larger number of games would conclude that *maxTime = 90 seconds* would improve the time but would never become more effective in terms of win percentage, since unlimited time explores the entire tree down to depth level 3 through all the branches of the root node.

Therefore, it seems that the optimal configuration of the Minimax agent to play against a human, balancing efficiency with time, is *depth = 3* and *maxTime = 90 seconds*.

Table III presents the result of the games between the two Minimax agents with *depth = 3*, one with maximum scan time of 90 seconds and another unlimited. It ratifies in some way the results that had been obtained in Table II:

- They are very evenly matched in terms of effectiveness: 10 to 8, with 2 draws and 2.50 points difference on average.

- The average duration of the games is 22.19 minutes, which is close to the sum of the value of Minimax vs. Greedy (11.65 minutes) and the one using *maxTime = 90 seconds* (9.85 minutes).

It is therefore concluded that:

The best configuration of a Minimax agent with Alpha-Beta pruning, to play iwoki against a human, is with a depth level 3 and with a maximum search time of 90 seconds on each of the main branches of the root node.

By testing it against the author of the game over 5 games, it is relevant that Minimax wins in all cases. Table IV shows the difference in points they get. Throughout the game it is Minimax that takes the lead. He avoids at all costs that the human places hexagonal pieces, maintaining higher score than the human at all times.

The Minimax agent anticipates the human and disrupts any strategy the human tries to implement. It wins in all games played.

VII. Conclusion

In the course of this work, iwoki and its characteristics have been made known, which has made it possible to know how the modelling and implementation of its components has been carried out.

Based on the study of the state of the art, we have defined which agents would be part of the experiments, according to the nature of the iwoki, and we have proceeded to develop them.

During the experimental phase, a series of games have been carried out that have put the different agents into competition. It has been determined how good the Minimax agent is with respect to Random and Greedy.

The results obtained cover the twofold purpose set out at the beginning of the work:

- To find the appropriate implementation for the Minimax agent and to provide it with the optimal configuration according to the experimental results.

- To present iwoki as a new contribution to the field of research in artificial intelligence applied to games, establishing a starting point from which to develop a diversity of agents with different technologies.

Consequently, some lines of future work are identified as a result of what is presented in this paper:

- There is the possibility of implementing the different **variants of Alpha-Beta** pruning, mentioned in the chapter on the state of the art.

- Since the original iwoki is of imperfect information, as the small pieces are not visible to the rest of the players, it could be developed as a **Bayesian game**. This way you can take into account the probability that an opponent has certain pieces, knowing the number of times they have been played previously (it must be remembered that there are four copies of each small piece).

- Iwoki is also a stochastic game with a high degree of branching, so it would be appropriate to develop the **Monte Carlo Tree Search** method.

- It is feasible to develop an agent with **Reinforcement Learning** in an iterative mode, who is able to learn from his own experience, putting into practice the knowledge acquired throughout his previous training games.

- To code an agent using **Deep Reinforcement Learning**, the input of the neural network would be the current state (game turn) and the output would be a *Q-value* for each possible action in that state. The problem that arises is that there are a different number of possible actions in each one of the millions of existing states, so the number of output neurons would be undetermined. This opens the door to a new line of research to study this specific case and model a neuronal network that will allow the problem of iwoki to be treated.

- Study if it is feasible to implement a **Recurrent Neural Network** (RNN) for a new **Vanilla** agent. The RNN would be similar to the one used for language models when obtaining a sequence of words and returning a probability distribution of the next word. The hidden state includes the text seen so far at each time step. In a generative model, the word output of each time step becomes the input of the next one.

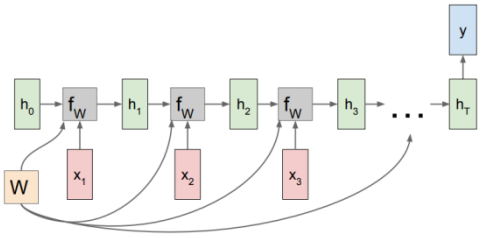


Fig. 9. Computing graph of an RNN with a single output [15].

The same representation of Fig. 9 could be valid for our case, since the following analogies exist:

- The input to the network would be the set of possible actions to be performed in a particular state.
- The hidden layer is fed with the states that the game acquires in each time step, that is, each time the player performs a specific action.
- The output would be a probability distribution of the next action to be performed, through a *softmax* function.
- This action would be the input of the next time step of the model.

Agent Vanilla could be replaced by **LSTM** or **GRU** in case the typical problems of simple RNNs occur, such as vanishing gradients or exploding gradients.

- Of course, it would be interesting to establish a **graphic environment** that makes it easier for a human player to execute his moves without having to reproduce the moves with the physical version of the game. Its application would be extendable to mobile devices.

- As a consequence, to enable the game also for human players exclusively, through online play. It could be extended to **up to 6 players**, just like the original iwoki.

REFERENCES

[1] J. Bonilla Sierra, "Iwoki Maths: el cálculo matemático en su forma más cuidada," *Consola y Tablero*, 2019. [Online]. Available: https://consolaytablero.com/2019/03/15/iwoki-calculo-matematico/. [Accessed: 13-Sep-2019].

[2] R. C. Carlos Alberto, "Aproximación a la teoría de juegos," *Rev. Ciencias Estratégicas*, vol. 17, no. 22, pp. 157–175, 2009.

[3] P. Aguilar, "IA – Algoritmos de Juegos," 2008.

[4] J. von Neumann and O. Morgenstern, *Theory of games and economic behavior*. 1944.

[5] R. Wright, "Progess is not a zero-sum game," 2006.

[6] J. P. Zagal, J. Rick, and I. Hsi, "Collaborative games: Lessons learned from board games," *Simul. Gaming*, vol. 37, no. 1, pp. 24–40, 2006.

[7] J. B. Mimbang, "La teoría de juegos: El arte del pensamiento estratégico," *50minutos.es - Econ. y Empres.*, 2016.

[8] J. C. Harsanyi, "Games with Incomplete Information Played by 'Bayesian' Players, I–III Part I. The Basic Model," *informsPubsOnLine*, 2004.

[9] G. N. Yannakakis and J. Togelius, *Artificial Intelligence and Games*. 2018.

[10] D. J. Edwards and T. P. Hart, "The Alfa-Beta Heuristic," *Artif. Intell. Proj. - RLE MIT Comput. Center, Memo 30*, 1963.

[11] S. J. Russell and P. Norvig, *Artificial Intelligence A Modern Approach; PearsonEducation*. 2010.

[12] O. David-Tabibi and N. S. Netanyahu, "Verified Null-Move Pruning," *ICGA Journal, Int. Comput. Games Assoc.*, vol. 25, 2002.

[13] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin, "Alpha-Beta + TT," *Dep. Comput. Sci. Univ. Alberta*, no. December, 1994.

[14] J. Pearl, "SCOUT: A Simple Game-Searching Algorithm With Proven Optimal Properties," *Cogn. Syst. Lab. Sch. Eng. Appl. Sci. Univ. Calif.*, 1980.

[15] F.-F. Li, J. Johnson, and S. Yeung, "Lecture 10: Recurrent Neural Networks," in *Stanford CS231N*, 2018.