Diseño de Algoritmos

Implementación del Juego del Molino

Grupo G1

Javier Loro Carrasco

Diego Cordero Contreras

Mohamed Essalhi Ahamyan

Índice:

1. Introducción

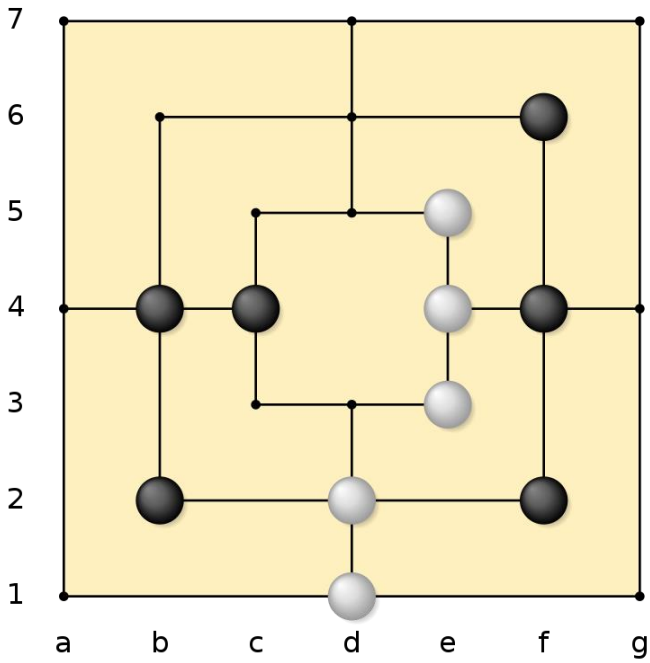
En este proyecto se ha desarrollado un programa Cliente-Servidor para jugar al Juego del Molino, con el fin de aplicar los conceptos estudiados en teoría acerca del Diseño de Algoritmos en un escenario real. Para ello, se han llevado a cabo cuatro tareas fundamentales, las cuales incluyen la definición de los estados, acciones, entidades y la lógica del juego, así como la creación de diversos agentes que puedan jugar de manera inteligente.

La implementación de este juego supone un reto por la libertad que se ha proporcionado a la hora de cualquier elección, es decir, no se ha establecido ningún lenguaje predeterminado, no se ha establecido un servidor común o una forma de representar el tablero. No obstante, se han dado diversas indicaciones que nos ayudaron en gran medida a implementarlo.

Algunas cosas que daremos por hecho para el trabajo será que como lenguaje hemos elegido Python 3.12, se ha establecido un servidor con sockets, el tablero se ha creado como un array de 24 posiciones, en el que se trabajará con aritmética modular, módulo 8 (más adelante esto se detallará); y además todos los objetos que haya serializados se tratarán con json.

En el transcurso de este trabajo se detallará el proceso de diseño e implementación del sistema, se mencionarán las tecnologías utilizadas y se presentarán los resultados obtenidos.

2. Juego Del Molino



El Juego del Molino es un juego de mesa estratégico para dos jugadores que se juega en un tablero de tres en tres intersecciones unidas por líneas, formando así una red de 24 líneas. El objetivo del juego es crear molinos, que son tres piezas del mismo color alineadas en una línea recta, permitiéndole al jugador que lo forma retirar una de las piezas del oponente.

Cada jugador comienza con nueve fichas de su color, y se turnan para colocar sus fichas en las intersecciones libres del tablero. Una vez que todas las fichas han sido colocadas, los jugadores se turnan para mover sus fichas a una intersección adyacente que esté vacía. Si un jugador forma un molino, puede retirar una ficha del oponente que no forme parte de un molino. Si el oponente sólo tiene dos fichas restantes, entonces el jugador puede retirar una ficha de molino del oponente.

Una vez que todas las fichas han sido colocadas, los jugadores deben mover sus fichas a una intersección adyacente hasta que uno de los jugadores tenga menos de tres fichas o no pueda mover sus fichas. En este punto, el otro jugador es declarado ganador.

2.1. Representación del Juego

Para representar el tablero del juego en el código, hemos definido el tablero como un estado. Debemos entender que el tablero se puede entender como un array de 24 posiciones numeradas del 0 al 23. Para poder definir esta estructura y que nos sea más sencillo el trabajo con las variables, el estado se define con los siguientes campos:

1. **Free:** Array que indica las posiciones libres en el tablero
2. **Gamer:** Matriz de dos por el número de fichas vivas del jugador. La primera dimensión es utilizada para diferenciar las fichas de cada jugador. La segundad dimensión representa las fichas vivas del jugador, al jugarse con 9 fichas, el máximo será de dos por nueve, aunque depende de las fichas colocadas y que hayan sido eliminadas.
3. **Chips:** Array con dos valores que indica el número de piezas sin colocar en al tablero para cada jugador.
4. **Turn:** Entero que indica el turno del jugador que debe mover

Estos campos compondrán una estructura JSON que denominaremos estado.

También debemos representar las acciones, es decir, los movimientos al realizar una partida, para ello se genera una acción con los siguientes campos:

1. **POS_INIT:** entero que indica la posición inicial desde la que se mueve una ficha. Tendrá el valor -1 si esa ficha se coloca.
2. **NEXT_POS:** entero que indica la posición final a la que se mueve una ficha.
3. **KILL:** entero que en caso de que se elimine una ficha rival indica su posición, en caso contrario su valor es -1.

Estos campos compondrán una estructura JSON que denominaremos movimiento.

Se ha creado una estructura JSON con el estado y el movimiento para unificar todos los datos:

1. **STATE:** estado del que se parte
2. **MOVE:** acción realizada por el jugador
3. **NEXT_STATE:** estado resultante tras la acción del jugador

Con esta definición llamada sucesor trabajaremos como definición del tablero.

3. Servidor

Si bien es cierto que no es necesario implementar un servidor como parte del dominio del problema, se ha realizado de cara a poder jugar con otros jugadores cada uno desde su dispositivo y hacer más enriquecedora la experiencia.

Para ello, se ha utilizado una implementación cliente-servidor basada en sockets para el paso de mensajes.

Servidor en Azure: Se ha desplegado el servidor en una máquina virtual de Azure de manera que se puede jugar de dos maneras

1. **Local:** Entre dos jugadores que utilizarán el mismo dispositivo
2. **Online:** Entre dos jugadores que pueden estar en distintos dispositivos, conectándose mediante el servidor de Azure.

Debemos tener en cuenta que las horas que ofrece el servicio gratuito de Azure son limitadas, por lo que el servidor Online no siempre está activo

Se ha realizado una implementación sencilla en el paso de mensajes. Estos mensajes se envían con un identificador definido como primer parámetro para hacer más sencillo el tratamiento de la información, seguido del resto de los campos del mensaje.

Las funciones del sistema son funciones básicas de cualquier sistema cliente-servidor que intente implementar un juego como este, estas funciones son:

1. **Iniciar Sesión:** con un nombre de usuario que es único y una contraseña
2. **Registrarse:** darse de alta en la base de datos con un nombre de usuario que es único y una contraseña
3. **Darse De Baja:** eliminar los datos de tu usuario de la base de datos
4. **Modificar Contraseña:** cambiar la contraseña del usuario
5. **Consultar tus estadísticas:** te mostrará el número de partidas, de victorias, el tiempo que empleas en realizar cada partida y un resumen de las partidas indicando el rival y quién gano.
6. **Crear una partida:** método para jugar una partida, deberás introducir el tipo de jugador que quieres ser (manual, torpe, Montecarlo…) y te devolverá un identificador y quedarás a la espera de que otro jugador se una a la partida con dicho identificador
7. **Unirse a una partida:** método para jugar una partida, deberás introducir el tipo de jugador que quieres ser (manual, torpe, Montecarlo…) y un identificador para poder conectarte.
8. **Salir:** cerrar la conexión

Además, el servidor comprueba que el usuario no pueda conectarse 2 veces al mismo servidor para que no existan problemas a la hora de jugar partidas.

4. Agentes

Uno de los objetivos de la práctica es conocer e implementar distintos algoritmos para intentar simular el razonamiento de como jugaría un ser humano, o ser capaz de superarlo. Por ello, se han definido distintos agentes con distintos algoritmos capaces de jugar con mayor o menor efectividad una partida.

4.1. Humano

No es un agente en sí, es la manera que tendremos de jugar de manera manual una partida, tratamos de que la interfaz sea lo más cómoda posible, añadiendo mensajes de errores, jugadas que realiza el rival, en qué fase se encuentra el jugador (colocar fichas o mover fichas)…

4.2. Aleatorio

Este agente totalmente autónomo recibe el estado del tablero, a partir de ahí genera todos los posibles sucesores con ese estado y elige uno de manera aleatoria

4.3. Monte Carlo

Este agente implementa un árbol de Monte Carlo y elige la jugada con mejor valoración. Un árbol de Monte Carlo es un algoritmo de búsqueda heurística para algunos tipos de procesos de decisión. En este caso, la decisión será la jugada con mayor valoración que hayamos conseguido dado un tiempo de cómputo.

Se le dará un tiempo como argumento y con ese tiempo realizará tantas veces el bucle. Cada iteración del bucle consta de 4 partes:

1. Selección: se elige el nodo de mayor valoración que no haya sido expandido y no sea terminal
2. Expansión: de entre todos los sucesores del nodo seleccionado, si en uno de ellos ganamos, lo elegimos, si no, elegimos de manera aleatoria.
3. Simulación: para el sucesor expandido, simulamos una partida de manera aleatoria, debe devolvernos un valor, 1 victoria, 0 empate, -1 derrota.
4. Propagación: propagamos el resultado al padre del nodo de manera recursiva hasta que llegamos a la raíz. Para propagar el resultado haremos uso del criterio UCT, donde la valoración del nodo es:

$$UCT = \left(\frac{nj_{ganadas} - nj_{perdidas}}{n}\right) + \frac{2}{\sqrt{2}} * \sqrt{\frac{2 * \log_2 n_{padre}}{n}}$$

$$n = número\ de\ veces\ que\ se\ ha\ visitado\ el\ nodo$$

$$n_{padre} = número\ de\ veces\ que\ se\ ha\ visitado\ el\ nodo\ padre$$

$$nj_{ganadas} = número\ de\ veces\ ganadas\ en\ el\ nodo$$

$$nj_{perdidas} = número\ de\ veces\ perdidas\ en\ el\ nodo$$

Además, ha sido implementado con dos variantes distintas que nos dan distintos resultados.

1. Monte Carlo 1: árbol de Monte Carlo visto en el laboratorio de la asignatura. En la selección se elige el nodo con mayor valoración, que haya sido anteriormente expandido y no sea final.
2. Monte Carlo 2: árbol de Monte Carlo explicado en el artículo de Cameron B. Browne: *A Survey of Monte Carlo Tree Search Methods.*
   Difiere ligeramente con el visto en el laboratorio de la asignatura, pero el principio es el mismo. Empezando con la raíz, en la selección se mantiene el nodo y se expande ese mismo nodo hasta que ha sido expandido por completo, el siguiente nodo que se expande es el mejor hijo y comenzaría de nuevo el bucle. Finalmente, se devuelve el mejor hijo del nodo inicial, es decir, la siguiente jugada con mejor valoración.

4.4. Minimax

Este agente implementa un árbol minimax con todas las jugadas posibles hasta una profundidad dada y, tras aplicar el conocido algoritmo,

Como tal, un árbol minimax es una estructura de datos que representa un conjunto de jugadas (en este universo de juego que nos encontramos) que dos jugadores pueden realizar a partir de una jugada padre. La primera jugada sería el nodo raíz del árbol. Como tal, esta estructura es únicamente válida para juegos de dos jugadores con jugadas finitas. Este árbol trataría de llegar a las últimas jugadas posibles o a una profundidad delimitada con anterioridad, como es en nuestro caso. En estas jugadas aplicaría una función de evaluación, distinta para cada juego, que otorgará una puntuación positiva si va ganando max, 0 para empate o una puntuación negativa si va ganando min, porque cada nivel del árbol va asociado a un jugador, max o min.

Esta puntuación calculada en el último nivel se propagará al padre con el algoritmo minimax, esto es, si el nivel es max, solo cogerá la puntuación mayor de sus hijos, si es min, la puntuación menor, de modo que cada vez que visita un hijo solo se actualiza si el valor es mejor que el anterior que tenía del otro hijo. Esto hará que se propaguen todas las puntuaciones hacia arriba, de modo que el nodo raíz solo tendrá que elegir el siguiente nodo con la mejor puntuación para obtener la mejor jugada.

En nuestro caso hemos implementado el árbol con la poda alfa beta, esto añade otros dos valores al margen el valor del nodo, alfa y beta. Estos valores en un principio se establecen como, alfa el valor más bajo posible y beta el mayor valor posible, imaginemos que son -999 y 999. Estos valores son así en un principio para el nodo raíz y se propagan para abajo con sus hijos. Estos valores cambiarán en la fase de propagación hacia arriba, cuando un nodo min

actualice su valor también actualizará su valor beta si este es menor, lo contrario pasaría con un nodo max y su alfa. Estos valores se propagan hacia arriba, de modo que cuando se explore otra rama, los valores alfa y beta ya no serán los primeros si no los obtenidos en otras ramas. Si alfa es mayor o igual que beta en una actualización, no se sigue explorando esa rama, ya que el mejor valor ya ha sido obtenido. Esto es a lo que se le denomina poda.

Una vez tenemos un breve concepto explicado de lo que es el algoritmo minimax, explicaremos un poco como ha sido su implementación para éste nuestro juego del molino. En primer lugar, explicaremos cuál ha sido la función de evaluación. Lo más común sería haber hecho la típica función en estos juegos de contar 1 si gana max, -1 si ganara min o 0 si no ha ganado nadie todavía, pero esto sería inútil de momento, ya que, habiendo tantos sucesores, la profundidad se ve muy limitada y en pocas ocasiones se llega hasta el final de las jugadas, salvo en jugadas finales. Con lo que nuestra función de evaluación ha sido una más inmediata, fichas de max – fichas de min, de modo que siempre se elegirá el camino que vaya hacia un molino o hacia evitar otro molino.

Esta función nos ha ahorrado profundidad y además nos ha creado una gente inteligente que casi siempre evitará molinos o un mla mayor a la par que siempre busca las mejores opciones para tener más fichas y, por tanto, ganar.

Realmente con respecto a nuestra implementación no hay mucho más que destacar, pues el resto sigue los principios básicos de este algoritmo. El nodo raíz es la jugada actual, y sus hijos son los sucesores generados por nuestra función de generar sucesores. La profundidad que le hemos podido dar, por complejidad computacional, es de 4 si queremos que el tiempo de jugada no exceda los 30 segundos, puesto que con esto ya estamos hablando de más de 2000000 nodos expandidos.

4.5. Aprendizaje

Este agente está basado en el aprendizaje por refuerzo. Este es un tipo de aprendizaje automático en el que un agente aprende a tomar decisiones a través de la interacción con un entorno. El objetivo del agente es maximizar una recompensa numérica a largo plazo que recibe del entorno en función de las acciones que toma. El agente no tiene información previa sobre el entorno, por lo que debe explorar y experimentar para aprender la mejor manera de obtener la máxima recompensa. Durante este proceso, el agente aprende a asociar acciones específicas con estados del entorno y recompensas esperadas.

Esta implementación para este juego, como realmente cualquier implementación de este aprendizaje, nos supuso un reto. Ya que, como tal, hay mucha libertada a la hora de implementarlo. Nosotros supusimos que lo mejor

era almacenar un estado, la acción a tomar y su recompensa. De esta manera, el agente iría explorando todos los estados posible scon sus acciones y almacenando una recompensa al respecto que en algún momento actualizaría al volver por ella. Esto supone una gran dificultad ya que, volviendo al ejemplo anterior, si con solo 4 de profundidad ya teníamos 200000 de pares estado-acción/sucesor, tratar de almacenar todos los estados-acción posibles se trata de algo imposible. Para este mismo motivo se toman unas cuantas acciones al respecto. Primero se implementa un fichero json que el par estado-acción lo almacena como una clave codificada en md5 para que ocupe menos (pasando un archivo de 300000 de líneas que antes ocupaba una giga a ocupar solo cien megas); se eliminan filas que nunca se usan y que por tanto siempre tienen la recompensa por defecto.

Pero para explicar más optimizaciones realizadas, es necesario pararnos a ver el proceso que hemos seguido para almacenar 4estas recompensas. Lo primero de todo, decir que el archivo de código donde está el agente está dividido en dos partes, el entrenador y el agente como tal. El entrenador es una clase que lo que hace es simular partidas de un agente q learning con o sin experiencia, contra un jugador aleatorio o a elegir (se escogió el aleatorio por la fluidez de sus movimientos y la rapidez de elección). Esto hace que el agente almacene muchos movimientos y sus recompensas. Cabe a añadir que se implementó de manera que, si se quisiera acelerar este aprendizaje, se podría poner a jugar a dos agentes cualesquiera a jugar, ya sea Montecarlo, aleatorio o incluso manual para volver este aprendizaje por refuerzo en imitation learning, ya que este agente tomaría los movimientos usados por uno de los agentes y simplemente almacenaría sus valores en cuanto a recompensas. Para ejecutar este entrenador, solo se tiene que llamar a esta clase con Python y darle como argumento cuantas partidas quieres que se simulen. También en el código está resaltada el argumento para cambiar los tipo de jugadores que quieres que se enfrenten

La otra parte del código es el agente con el que se puede jugar, este directamente coge los movimientos del fichero json y almacena los nuevos no conocidos con la recompensa por defecto o, si hay un molino, con la recompensa correspondiente. Esto acelerar considerablemente, no solo el proceso de aprendizaje, si no también el proceso de elección, ya que en ocasiones solo hay estados que es la primera vez que se visitan p0roque son demasiados como para poder almacenarlos todos con los métodos que conocemos.

La función de evaluación usada sería la de comprobar si por una acción se llega a un molino o si por el contrario se evita. La recompensa mayor se da, si se gana y la menor si se pierde. Cada acción va con un penalizador de 0.1 para que no se quede en bucle repitiendo acciones que no sabe a donde llevan.

Podemos destacar de este agente que, para su máxima optimización, finalmente hemos hecho que, no solo no se almacenan tantos estados como en

un principio había con la recompensa por defecto, sino que además almacena en dos archivos separados por el turno de la acción, es decir, el jugador uno va en un archivo y el jugador dos van en toro archivo, esto reduce a la mitad el tiempo de exploración dentro de un archivo por turno. Esto sumado a que no está el estado completo dentro del json si no que está en md5, hace que el tiempo de este agente haya sido el menor, y en ratio tiempo/victorias, el mejor.

4.6. Integración con Otros Equipos

A modo de poder comparar las diferentes implementaciones que hemos hecho los distintos equipos, ya que inicialmente no teníamos un servidor unificado ni definíamos el mismo tipo de mensajes, ha sido una tarea extra realizada para poder comprobar la eficiencia de los distintos algoritmos realizados.

Para ello, se ha establecido con el grupo G3 compuesto por Enrique, Ignacio y Pablo una adaptación en nuestro servidor que nos permita jugar contra sus algoritmos de MonteCarlo y de Q-Learning. Además, se pueden elegir como uno de los modos a la hora de jugar partidas.

4.7. Estadísticas

Una vez implementados los distintos agentes, hemos realizado distintas pruebas para comprobar la eficacia de las diferentes implementaciones, comprobando tanto el número de victorias que obtenemos con cada agente como el tiempo que necesita por partida, ya que el tiempo es una variable limitada y debemos dar buenas respuestas en cortos periodos de tiempo.

A continuación, mostramos la tabla a modo resumen con las pruebas realizadas.

Jugador 1	Jugador 2	Jugadas por Partida	Resultados (J1 – Empates – J2)	Tiempo medio (J1 – J2)
MonteCarlo1	Torpe	53	93 – 0 – 7	0:44 – 0:00
MonteCarlo2	Torpe	48	99 – 0 – 1	0:32 – 0:00
Minimax	Torpe	38	99 – 1 – 0	1:44 – 0:00
Q-Learning	Torpe	35	100 – 0 – 0	0:02-0:00
MonteCarlo2	MonteCarlo1	49	82 – 0 – 18	0:34 – 0:40
Minimax	MonteCarlo1	50	88 – 10 – 2	1:48 – 0:27
Q-Learning	MonteCarlo1	36	100 – 0 – 0	0:03 – 0:29
MonteCarlo2	Minimax	46	2 – 2 – 96	0:26 – 1:52
MonteCarlo2	Q-Learning	41	9 – 0 – 91	0:30 – 0:05
Minimax	Q-Learning	195	5 – 95 – 0	1:19 – 0:30
MonteCarlo1	MC1 Enrique	105	3 – 18 – 79	0:44 – 0:50
MonteCarlo1	MC2 Enrique	81	8 – 4 – 88	0:48 – 0:28
MonteCarlo2	MC2 Enrique	57	93 - 0 - 7	0:35 – 0:30
MonteCarlo2	MC1 Enrique	59	97 – 0 – 3	0:36 – 0:50
Q-Learning	Q-Learning Enrique	44	86 – 0 – 14	0:07 – 0:17

5. Manual de Usuario

A modo de poder ejecutar de manera más sencilla el Juego y que no existan problemas, queremos explicar de manera resumida los pasos para iniciar el Juego.

Nos situamos en la carpeta juego-lab-g1 del repositorio y ejecutamos el comando: 'python3 Código/server.py' de esta manera abriremos el servidor para la comunicación entre los jugadores.

En la misma carpeta, ejecutamos dos instancias del jugador, con el comando: python3 Código/jugador.py

Primero se nos pedirá el tipo de conexión. Asumiendo que el servidor de Azure no está disponible, elegimos la opción 1. LOCAL introduciendo un 1 por teclado.

```
ELIJA METODO DE CONEXION:
        1. LOCAL
        2. ONLINE
1
```

A continuación, iniciamos sesión con nuestro usuario y contraseña o nos registramos si no tenemos cuenta.

```
ACCESO AL SISTEMA:
        1. INICIAR SESION
        2. REGISTRARSE
```

Accederíamos al menú principal donde tendríamos todas las opciones

```
----------
Inicio de Sesión Correcto
Usuario: Mohamed
ACCEDIENDO AL MENU

Opciones del menú:
        1.Darse de baja
        2.Modificar contraseña
        3.Consultar mis estadisticas
        4.Crear una partida
        5.Unirse a una partida
        6.Salir
Introduce una opcion:
```

Una vez en este menú, el primer jugador deberá Crear una partida y el segundo Unirse a dicha partida. El primer paso para ambos será elegir el jugador entre todos los implementados:

```
Tipos de jugador:
        1. Jugador Manual
        2. Jugador Torpe
        3. Jugador MinMax
        4. Jugador MonteCarlo 1
        5. Jugador MonteCarlo 2
        6. Jugador MonteCarlo 1 Enrique
        7. Jugador MonteCarlo 2 Enrique
        8. Jugador Q_Learning
        9. Jugador Q_Learning Enrique
Determinar tipo de jugador:
```

Al jugador que cree la partida le aparecerá un Código:

```
ID_JUEGO: 205236
ESPERANDO RIVAL...
```

El jugador que se une a la partida deberá introducir el Código:

```
Introduce el id del juego
ID_JUEGO: 205236
CONECTANDO A LA PARTIDA....
```

Finalmente, la partida se habrá creado, un jugador iniciará y el otro esperará al rival:

```
1º JUGADOR: Mohamed
--------------------------------
Tienes el primer turno--> O
----------------------------------------
        TABLERO          ||       POSICIONES
[ ] -- -- [ ] -- -- [ ]  ||  00 -- -- 01 -- -- 02
 |         |         |   ||  |        |        |
 | [ ] -- [ ] -- [ ] |   ||  |  08 -- 09 -- 10 |
 |  |      |      |  |   ||  |  |     |     |  |
 |  | [ ] [ ] [ ] |  |   ||  |  |  16 17 18 |  |
 |  |  |        |  |  |   ||  |  |  |     |  |  |
[ ][ ][ ]     [ ][ ][ ]  ||  07 15 23     19 11 03
 |  |  |        |  |  |   ||  |  |  |     |  |  |
 |  | [ ] [ ] [ ] |  |   ||  |  |  22 21 20 |  |
 |  |      |      |  |   ||  |  |     |     |  |
 | [ ] -- [ ] -- [ ] |   ||  |  14 -- 13 -- 12 |
 |         |         |   ||  |        |        |
[ ] -- -- [ ] -- -- [ ]  ||  06 -- -- 05 -- -- 04

COLOCAR FICHA
Seleccione una casilla del 0 al 23:
```

```
--------------------------------
Tienes el segundo turno --> X
ESPERANDO JUGADA RIVAL...
```

Una vez dentro, para jugar una partida, deberás colocar fichas y mover fichas teniendo en cuenta que cada ficha del tablero se representa con un valor del 0 al 23 como se indica a la derecha.

6. Conclusiones

6.1. Javier

Uno de los mayores retos a la hora de afrontar el laboratorio comparado con otras asignaturas era la libertad que de primeras se te plantea para desarrollar ciertas funcionalidades. Para mi este ha sido también el gran punto a favor.

El desarrollo de la practica conlleva trabajar sobre muchos aspectos aprendidos a lo largo de la carrera. Creo que hemos realizado un gran desarrollo, pero descuidando también ciertos aspectos de buenas prácticas a la hora de escribir código que nos ha lastrado según hemos ido avanzando en la práctica y tenido que retocar y adaptar respectivamente según íbamos avanzando. Una mejor definición del funcionamiento final nos hubiera llevado a preparar desde el comienzo un mejor código que debido al desconocimiento de las funcionalidades futuras no nos permitió escribir.

La parte de no conocer el juego nos ha permitido tener una perspectiva amplia a la hora de optimizar los recursos, usar elementos interesantes más allá de una matriz de elementos como aprovechar las propiedades del módulo para determinar las posiciones, movimientos o el turno

La implementación de los algoritmos de teoría ha sido algo enrevesado de comprender, pero es satisfactorio verlos en funcionamiento, y estaría queda como interesante la posibilidad de desarrollar un nuevo agente aprovechando el potencial de cada uno dependiendo de la fase del juego

6.2. Mohamed

Este laboratorio ha sido uno de los más divertidos de la carrera. Se nos pedía un gran reto como es implementar un juego desde cero y, además implementar distintos jugadores con algoritmos vistos en clase. Este método, de utilizar un juego y en base a ello, aprender cómo funcionan distintos algoritmos me parece un enfoque acertado que hace la práctica más amena. Además, al ser grupos de tres personas y siendo esta la única tarea del laboratorio hace que se pueda dividir el trabajo de manera balanceada y se pueda seguir las distintas lecciones que se imparten sin que haya muchos retrasos si el grupo se organiza adecuadamente.

Con respecto al juego, personalmente el hecho de no conocerlo no me parece que sea relevante en el desarrollo de la práctica, ya que lo que se pide son agentes que sepan jugar independientemente con cierto tipo de algoritmos y que, en principio, el conocimiento no afectaría a su desarrollo.

Además, es satisfactorio ver como poco a poco ves que dichos jugadores que se implementan funcionan y consiguen buenos resultados incluso contra un humano.

Como único *“pero”* al laboratorio sería el hecho de implementar un servidor, al hacerlo cada uno de manera separada y no un servidor unificado con el mismo tipo de mensajes y mecanismos, hace que no se puedan comparar de manera sencilla los jugadores implementados si no se ponen de acuerdo grupos distintos y el poder compararlos podría aumentar la competitividad por ver quien lo hace mejor.

6.3. Diego

Con este laboratorio me he sentido muy realizado porque es de los pocos que hemos tenido en la carrera con los que llegabas a casa y realmente me apetecía ponerme a investigar como podríamos hacer algo de éste o implementarlo mejor. Esto es algo a lo que le doy mucho valor porque no solo resulta divertido si no que, además, te permite aplicar lo aprendido en clase de forma realmente práctica, viéndole una utilidad y, desde luego, sentando mucho mejor sus bases.

Como tal el mayor reto y problema para mi punto de vista es el mismo, la libertad que te ofrece. El hecho de que realmente solo empieces con pequeñas indicaciones al principio de cómo se deben implementar ciertos detalles e inmediatamente después ya se te de “salida” a hacer lo que quieras es, sin duda, lo mejor de este laboratorio, pero entiendo también que en algunos momentos también haya podido suponer grandes contratiempos sobre todo a la hora de estandarizarlo para poder probarlo con los compañeros. No obstante, si se le arrebatara este grado de libertad que tiene perdería también cierto encanto

En conclusión, este laboratorio ha sido muy realizador (por como avanza todo y finalmente ves un producto realmente terminado), abrumador tal vez en ocasiones por la cantidad de tareas que requería, pero, sobre todo, muy entretenido y realmente útil para los conceptos que la asignatura requiere

7. Bibliografía

Even, G., Garg, N., Könemann, J., Ravi, R., & Sinha, A. (2004). Min–max tree covers of graphs. *Operations Research Letters*, *32*(4), 309-315.

ISO 690

Even-Dar, E., Mansour, Y., & Bartlett, P. (2003). Learning Rates for Q-learning. *Journal of machine learning Research*, *5*(1).

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*, 279-292.

Soboĺ, I. M. (1990). Quasi-monte carlo methods. *Progress in Nuclear Energy*, *24*(1-3), 55-61.

Hunt, J., & Hunt, J. (2019). Sockets in Python. *Advanced Guide to Python 3 Programming*, 457-470.

Eggen, R., & Eggen, M. (2019). Thread and process efficiency in python. In *Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA)* (pp. 32-36). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).