Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona

Práctica PA1

Programación y algoritmia I

Autores:
Violeta Bonet y Pablo González

20 de enero de 2023

Índice

1. Introducción

Dado que no se ha cedido ninguna guía para la realización de este documento, y se permite un formato libre, la estructura escogida se presenta a continuación.

Hemos pensado que el proceso de desarrollo de esta práctica es tanto o más importante que el resultado final, y que, por tanto, se explicarán en el documento las distintas fases por las que pasó el proyecto, así como la metodología que seguimos durante su desarrollo.

Esta decisión hace que la explicación tenga un tono más cercano con el lector, y por eso ambos redactores nos referiremos a Pablo y a Violeta por el nombre propio, pero siempre manteniendo la formalidad en la redacción. El apartado de desarrollo es la parte del documento en la que se hablará de todo el proceso y resultado de realización de la práctica. Está dividido en cuatro subapartados: abs_board, bot, main_txt y main_gui, correspondientes a los principales bloques de trabajo del proyecto. Luego, en el apartado de manual de usuario, se explica cómo usar el programa sin entrar en detalles técnicos. Por último, el trabajo cierra con una conclusión y una bibliografía.

También cabe destacar respecto al código fuente, que ha sido programado en inglés (por facilidad a la hora de dar nombre a las variables y funciones) pero comentado en español, y la interacción con el usuario se hace también en este último idioma.

Respecto a las referencias usadas, hemos consultado principalmente la página web de Stack Overflow [2] y los tutoriales y referencias de Pygame [1].

2. Desarrollo

En los primeros días, tras la publicación del enunciado de la práctica, Pablo, con experiencia previa en programación, comenzó a planear todo lo que quería incluir dentro de esta y empezó a estructurar los ficheros que serían necesarios y la forma en que estarían diferenciadas las funciones, basándose en los ficheros de ejemplo cedidos junto con el enunciado. La estructura tendría en efecto cuatro ficheros: un fichero de constantes que usaría el programa, un fichero que contendría una función que generase las funciones usadas por los drivers, y dos ficheros (drivers) que servirían para poder usar el programa desde la terminal y desde una interfaz gráfica. Pablo tomó también unas cuantas decisiones más referentes al programa, como que las piedras se representarían simplemente como números en una matriz de dos dimensiones (tablero).

Violeta no tenía apenas experiencia en programación antes de comenzar el proyecto, pero quería también formar parte de la toma de decisiones y puso freno a la situación. En una pequeña reunión, Pablo defendió que la forma de programar en equipo es crear funciones o pequeños fragmentos de código individualmente y, tras pasarles pruebas exhaustivas, añadirlos al programa principal. Violeta, sin embargo, hizo notar que la diferencia de nivel entre ambos no permitiría esa metodología de trabajo, por lo que finalmente acordamos trabajar de una manera distinta. Dado que la práctica de programación está pensada para aprender, Violeta se enfrentaría a los retos de la práctica programando cada función con la ayuda de Pablo, es decir, programando juntos. Pablo, por su parte, crearía un bot contra el que pudieras jugar, para así tener que enfrentarse también a nuevos retos.

Con la estructura de ficheros montada y una dinámica de trabajo en equipo, iniciamos el desarrollo.

2.1. Abs_board

Durante la planificación ya habíamos decidido no renunciar a la complejidad en este proyecto, y la forma de implementar esta parte del código definiría el resto de la práctica, por lo que era crucial pensarlo bien. En este punto, Violeta aún no tenía casi nada de experiencia programando, así que

Pablo se ocupó de sentar las bases y comenzamos a trabajar juntos.

Para ello teníamos que decidir qué variantes iba a permitir nuestro programa, para pensar cómo hacer la función `board_setup` lo suficientemente flexible como para aceptarlas a todas. Pensamos en incluir todas las propuestas. Esta función está basada en la función `set_board_up` del fichero *abs_board* cedido junto con el enunciado, por lo que su propósito es el de generar las funciones que serán usadas posteriormente por los drivers (*main_txt* y *main_gui*). Lo primero que había que generar era, cómo no, el tablero. Para ello habría que especificar sus dimensiones al llamar a la función. Mirando todas las variantes que queríamos implementar, parecía claro que la función tenía que tener bastantes más argumentos que solo las dos dimensiones, y de hecho nos dimos cuenta de que al estar pasando esos argumentos libremente, podíamos crear la variante que quisiéramos simplemente modificándolos a placer. Finalmente, decidimos trabajar usando los siguientes argumentos modificables:

- Dimensiones del tablero (i y j)
- Longitud de la línea (tres en raya, cuatro en raya, ¿cinco en raya?)
- Número de piedras por jugador (el tres en raya *misery* tiene 4 por cabeza)
- El tipo de movimiento (puede ser el normal del tres en raya, el de la variante en la que solo puedes mover a posiciones adyacentes o el del cuatro en raya, que tiene gravedad y es completamente distinto)
- Si es *misery* o no
- El número de jugadores de la partida (para crear partidas de más de dos jugadores)

Si lográbamos implementar las funciones que devolvería `board_setup` de forma que funcionasen para todos estos argumentos, obtendríamos una enorme flexibilidad para poder inventar variantes de todo tipo.

La primera función que fue implementada, fue la de comprobar si existe una raya en el tablero por parte de algún jugador. En este punto aún no

habíamos tenido la reunión aclaratoria sobre la forma de trabajar, por lo que como parecía claro que iba a ser una función necesaria en algún punto del programa, Pablo decidió comenzar a implementarla por su cuenta.

Comprobar si existe una raya de dimensión n en el tablero parecía una tarea muy fácil en un principio, pero todas las ideas para hacer dicha comprobación eran, por decirlo así, poco elegantes. Finalmente, la mejor solución fue comprobar la raya desde la posición del último movimiento realizado, ya que, si se había generado una raya nueva, solo podía pasar por esa casilla, y si existía una raya anterior, ya se debería haber comprobado antes. Para ello, a partir de la casilla del último movimiento, se comprueba si hay una raya en las ocho direcciones.

El resto de funciones fueron discutidas e implementadas en común.

La función `board_setup` sufrió muchos cambios a lo largo del desarrollo de la práctica, adaptándose a las necesidades que veíamos al ir implementando paralelamente el driver de *main_txt*. Por ese motivo, en este caso se explicará el resultado final más que el proceso. Aparte de la función `nline_checker` anteriormente explicada, la función de `board_setup` contiene otras diez funciones más, incluyendo las que están asociadas exclusivamente al bot, que se explican más adelante. Sin embargo, solo se devuelven cinco funciones y dos variables.

La función `end_checker` es la encargada de comprobar si la partida ha acabado. Para ello, usa la función `nline_checker` para ver si hay una raya o no en el tablero, y en caso negativo también debe comprobar si al jugador siguiente le quedan movimientos posibles. Para comprobar si a un jugador le quedan movimientos posibles, existe la función `possible_moves` que dependiendo del tipo de movimiento y de si el número de piedras restantes para ese jugador es igual a cero, calcula todos los movimientos posibles en forma de una lista de tuplas.

Para llevar la cuenta de las piedras restantes por cabeza, se hace uso de una variable llamada `bag`, que es una lista con dos elementos: el número de piedras por cabeza en el turno actual y la cantidad de jugadores que aún no ha movido en ese turno. Cada vez que un jugador hace un movimiento, se resta una unidad al segundo elemento del `bag`. Para evitar que esta variable contenga valores incorrectos, existe una función llamada `bag_resolver`, que

actualiza los valores de la lista en caso de que sea necesario.

Los movimientos en el tablero se realizan haciendo uso de la función `move`, que dependiendo de la longitud de la tupla que contiene al movimiento, sabe qué tipo de movimiento es y lo realiza adecuadamente. Como la función `move` es usada también para que mueva el bot usando su propia función `bot_move`, existe una función específica para que mueva el jugador, llamada `player_move`, que actualiza el `bag` y usa la función `move`.

Por último, la función `check_stone` se encarga de comprobar si en una posición hay una piedra del jugador que se pasa como parámetro. Esta función es usada para validar movimientos introducidos por el usuario.

La función `board_setup` devuelve el tablero, el `bag` y las funciones de `check_stone`, `possible_moves`, `player_move`, `bot_move` y `end_checker`, que serán usadas por los drivers.

Después de ver las posibilidades que ofrecía el poder crear variantes nuevas solo modificando los parámetros de entrada a la función de `board_setup`, decidimos explotar esta funcionalidad al máximo, permitiendo la creación de variantes custom. Sin embargo, había que poner limitaciones a los valores que podía introducir el usuario en la función de `board_setup`, por lo que se creó la función `custom_board_checker`, que hace una serie de validaciones para crear una partida al menos coherente, como por ejemplo que el tamaño de la raya no supere a las dimensiones del tablero. Todas las variantes propuestas en el enunciado se almacenaron en el fichero de constantes como tuplas con los parámetros adecuados para pasar a la función de `board_setup`.

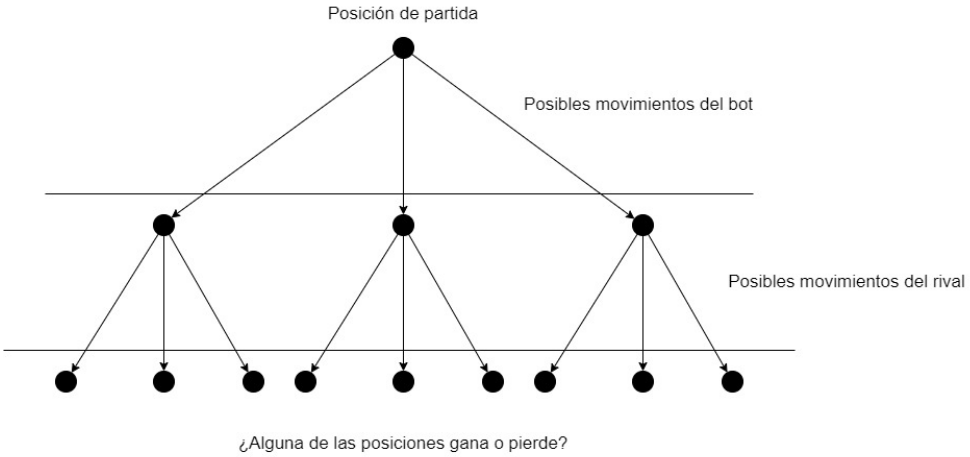
2.2. Bot

Como se ha explicado previamente, desde un principio consideramos que el objetivo de la práctica de programación era aprender todo lo posible durante su desarrollo. Por eso, y dada la diferencia de nivel de partida entre nosotros, tomamos la decisión de incluir un bot (un algoritmo capaz de jugar a un nivel decente) que fuera desarrollado por Pablo para dejar que Violeta se enfrentase a los retos del programa principal, aunque siempre trabajando juntos. Por tanto, esta parte del programa es la única que se ha desarrollado de forma individual por parte de Pablo para que así ambos pudiéramos

aprender con el proyecto.

Crear un bot capaz de jugar al tres en raya no es excesivamente difícil, ya que no hay muchos movimientos posibles y dado que el juego es infinito si se juega bien, se puede programar para que el algoritmo detecte ciertas posiciones y simplemente no pierda nunca. Sin embargo, el reto de este bot en particular, era que fuera capaz de jugar a todas las variantes posibles del juego que incluye nuestro programa, creadas y por crear.

La idea, desde un principio, se basaba en crear un algoritmo que valorase todos los posibles movimientos desde una posición en el tablero, y para cada movimiento valorase todos los posibles movimientos del rival en esa nueva posición, para volver a la situación inicial. Esto se puede pensar como un árbol de posibilidades en el que cada nodo representa una posición, y cada rama un posible movimiento.



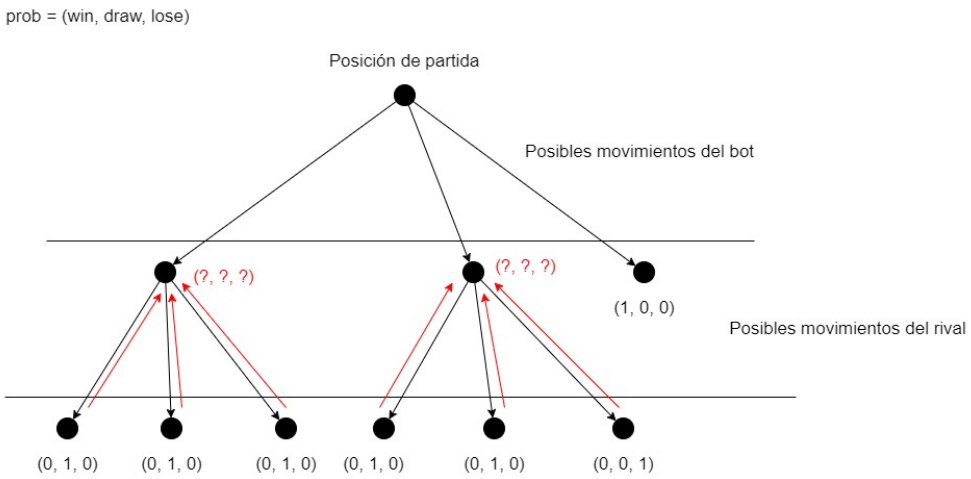
Este hipotético algoritmo se ejecutaría recursivamente un número finito de veces. A este número de veces se le llamó profundidad o `depth`, ya que determina cuán profunda es la búsqueda de movimientos o, en otras palabras, hasta dónde bajan las ramas del árbol.

Sin embargo, el problema aún estaba lejos de estar resuelto. Una de las cuestiones por tratar era cómo podía el bot determinar, tras hacer esta búsqueda, cuál era el mejor movimiento. La primera idea fue hacer que contara para cada rama inicial todos los nodos que acababan en victoria y escogiera la rama que más "victorias" tenía. Eso llevó a replantear el esquema

inicial de árbol, ya que quedó en evidencia que había algunos nodos (aquellos que acababan en posición de victoria o derrota) que no continuarían expandiendo ramas hasta llegar al nivel de profundidad.

Sin embargo, la solución de contar las victorias no era muy buena, ya que un movimiento que perdía tras una jugada podía ganar en cualquier otra situación y ser considerado como el mejor. Por tanto, también se tenían que contar de alguna forma las "derrotas" para poder valorar el "peligro" de hacer un movimiento, pero tampoco se podía hacer un contador sin más, porque en la situación anterior, el movimiento que perdía en una jugada seguía siendo el mejor, ya que solo había un nodo de derrota mientras que había múltiples nodos (a niveles más profundos) que llegaban a victoria. Además, todos los nodos del nivel más bajo de profundidad que ni ganaban ni perdían tenían que ser de alguna forma considerados como empates.

La forma de solventar ese problema fue asignar a cada posible movimiento tres valores: la cantidad de victoria, la cantidad de empate y la cantidad de derrota. A esta tupla en aquel momento se le llamó "probabilidad", y ha arrastrado ese nombre hasta el programa final. El algoritmo, por tanto, funcionaría de la siguiente forma: extendería las ramas del árbol buscando posibles movimientos recursivamente y sus probabilidades, y luego realizaría algún cálculo para asignar a cada nodo anterior, en dirección contraria, otra serie de probabilidades (los tres valores de la tupla) hasta llegar a los nodos que salen directamente del nodo de la posición inicial para poder decidir qué movimiento es mejor.



Para calcular las probabilidades en dirección inversa, una cosa estaba clara, en el turno del bot siempre se escogería el mejor de los movimientos, por lo que, si las ramas del árbol que se iban a recorrer en dirección inversa representaban movimientos del bot, el nodo padre iba a tener la mejor de las probabilidades de los nodos hijo. Sin embargo, si el movimiento correspondía al rival, no se sabía cuál escogería, por lo que se podía simplemente hacer la media de las probabilidades.

Esta media supone que, si una probabilidad contiene algo de derrota, existe la posibilidad de perder por esa rama si el rival juega bien. Teniendo eso en cuenta, faltaban por implementar las funciones que calculasen la mejor probabilidad y la probabilidad media dada una lista. La probabilidad media se calculó, evidentemente, haciendo la media de victoria, empate y derrota. Para la mejor probabilidad, dado el peligro que suponía que existiera algo de derrota, aunque fuera ínfima, se optó por escoger aquellas probabilidades con la menor derrota posible y descartar el resto. Luego, de entre las elegidas, se escogería la que tuviera mayor probabilidad de ganar.

En este punto, quedaba por pensar cómo adaptar este concepto abstracto de bot al programa que estábamos desarrollando. Había que tener en cuenta muchas posibilidades. Por ejemplo, el número de jugadores puede ser mayor que dos, existen variantes en las que pierdes si te quedas sin posibles movimientos (por ejemplo si tienes un número infinito de piedras) y hay variantes (*misery*) en las que pierdes si consigues hacer raya.

Se valoraron los posibles problemas que podrían surgir. Para un número de jugadores mayor que dos, se considerarían todos como rivales y se aplicaría la media de las probabilidades múltiples veces seguidas, una para cada rival. Además, cada nivel de profundidad englobaría un turno completo, es decir, los movimientos de todos los jugadores de la partida. Si un jugador se queda sin movimientos posibles, pierde, por tanto, si se va a calcular la media de probabilidades sobre una lista vacía, se considerará como victoria, ya que esta función aplica sobre rivales, y si se va a calcular el mejor movimiento sobre una lista vacía, se devolverá derrota. En caso de que la variante sea *misery* y hacer raya suponga perder, no debería haber problema si existe una función que compruebe la victoria o derrota de una posición según el modo de juego.

Había que implementar unas cuantas funciones auxiliares al bot, como una función que mueva una pieza en un tablero, una función que compruebe si una posición es ganadora o perdedora, una función que calcule los posibles movimientos independientemente del modo de juego, y las funciones de media de probabilidades y mejor probabilidad.

Muchas de estas funciones eran también necesarias para el juego principal (mover, comprobar si se ha ganado o no, etc.) por lo que se decidió usar las mismas adaptándolas sutilmente. Estas adaptaciones consisten principalmente en añadir a las funciones argumentos opcionales que tienen por defecto las variables del juego principal (como el tablero en el que se juega) pero que aceptan variables distintas para modificarlas o comprobar cosas en ellas sin afectar al juego. Por ese motivo, se cambiaron las posibles variables `nonlocal` de cada una de las funciones por estos argumentos opcionales.

También surgió el problema de que los posibles movimientos dependen de cuántas piedras tiene el jugador en ese momento en el `bag` explicado en *abs_board*, por lo que se usó la misma estructura replicándola en cada iteración recursiva.

Después de unos cuantos días programando, haciendo pruebas por consola, y descubriendo jugadas ganadoras pensadas por el bot, el único problema visible era su lentitud. Hacer un cálculo exponencial, incluso para profundidades bajas, eternizaba la partida. Había formas de hacerlo más eficiente, como almacenar toda la estructura de árbol en memoria para no recalcularla cada vez o descartar ciertas ramas en el árbol de búsqueda para no tener que computar tanto. Incluso se valoró la opción de usar *cython* para hacerlo más rápido, pero todas estas opciones no mejoraban mucho el rendimiento porque el problema seguía siendo que el algoritmo era exponencial, y las mejoras pensadas o bien se salían aún más de lo esperado de la práctica o bien suponían un esfuerzo que era necesario dedicar en otras partes del proyecto, por lo que se descartaron.

También existía el problema contrario: cuando había pocos movimientos posibles, como por ejemplo al final de una partida de cuatro en raya, el bot movía tan rápido que no se podía ni siquiera ver el movimiento. Solucionar este problema, sin embargo, era mucho más fácil. Simplemente se añadió un tiempo mínimo a la hora de hacer cada movimiento.

La última modificación al bot se hizo días después, cuando jugando contra él quedaba en evidencia un problema: en una posición ganadora, escogía el primero de los movimientos que ganaba, que no tenía por qué ser el más rápido. Esto suponía que, si el primer movimiento ganaba en dos turnos y escogía ese, tras recalcular las probabilidades en el siguiente turno, el primero de los movimientos ganadores podía ser otro que ganara en dos turnos también, es decir, que no seguía el camino elegido en el turno anterior, por lo que nunca perdía, pero tampoco ganaba hasta que se veía forzado a ello.

La solución a este problema fue sencilla: cuando un nodo del árbol equivalía a una posición ganadora, en vez de asignar la tupla `(1, 0, 0)` a la probabilidad de ese movimiento, cambiaría el 1 por el valor de `depth` de forma que quedaría como `(depth, 0, 0)`. Puesto que la función recursiva que despliega el árbol de búsqueda almacena el valor de `depth` de manera inversa, es decir que comienza la recursividad con `depth` en su valor máximo y lo va disminuyendo hasta llegar a cero, este cambio hace que las victorias más rápidas tengan una mayor prioridad en la función que obtiene la mejor de las probabilidades. De esta forma quedaba resuelto el problema.

El resultado es un algoritmo capaz (aunque con tiempo) de jugar a un nivel aceptable a cualquiera de las variantes que se le propongan. Esta afirmación es teórica, porque en la práctica, usar un tablero grande o jugar contra múltiples jugadores hace que la cantidad de cálculos se dispare y no acabe nunca. Por eso se decidió limitar la profundidad máxima a 3, ya que las variantes preprogramadas funcionan en un tiempo más o menos aceptable para ese valor.

2.3. main_txt

Desarrollamos el driver *main_txt* al mismo tiempo que *abs_board*. Eso añadió un nivel más de dificultad al trabajo porque en ciertas partes del *main_txt* necesitábamos funciones del *abs_board* que aún no habían sido creadas.

La primera vez que nos reunimos, nos pusimos directamente a programar. Empezamos por crear la función que te mostraba el tablero en la consola, porque era bastante independiente de las otras funciones y por su baja di-

ficultad. Decidimos crear esta función dentro del *main_txt* y no dentro del *abs_board* (como se había sugerido en el archivo cedido junto con el enunciado) porque no era una función que pudiera utilizar el *main_gui*, servía únicamente para imprimir el tablero en la consola.

En segundo lugar, hicimos la función `move_txt` y eso nos obligó a crear también la función `select_stone_org_txt`. La primera función se divide en dos bloques: mover para las variantes con un tipo de movimiento distinto a *gravity* y mover para las variantes con el tipo de movimiento *gravity*. La función pide a los jugadores por consola las posiciones donde quieren mover sus piezas y le va pasando esas coordenadas a la función `player_move` del *abs_board* para que mueva la piedra a esa posición. Si el tipo de movimiento no es *gravity* cuando el jugador se queda sin piedras, la función le pide que seleccione la piedra que quiere mover, y aquí es donde entra en juego la función `select_stone_org_txt`. Esta función pide las coordenadas y comprueba tanto que hayan sido bien introducidas como que correspondan a una piedra del jugador que quiere mover y te retorna las coordenadas de origen del movimiento. En cambio, si el tipo de movimiento es *gravity*, solo necesitamos saber a qué columna quiere mover el jugador.

Seguidamente, pasamos a darles forma a los menús del programa. Los que desarrollamos primero fueron el menú principal y el menú de la variante custom. En el menú principal se despliegan los diferentes modos de juego con una breve explicación de cada uno de ellos:

1. Tres en raya clásico
2. Tres en raya clásico *misery*
3. Tres en raya adyacente
4. Tres en raya adyacente *misery*
5. Cuatro en raya
6. Custom

Las opciones de la 1 - 5 conducen hasta el siguiente menú, pero la opción 6 conduce al menú que explica la variante custom, donde se pueden ver las

condiciones de uso de la variante. En este punto tuvimos que pararnos a pensar qué límites queríamos ponerle al juego customizable (por ejemplo, que el número de jugadores no fuera mayor que 10) y también sobre qué condiciones era necesario advertir al jugador para que su juego fuera posible (por ejemplo que el número de piezas de un mismo jugador que hace raya tiene que ser menor o igual al máximo del número de filas y el número de columnas). En este mismo menú también aparecen instrucciones del formato a seguir para introducir los valores del juego custom de manera que el programa los entienda.

Una vez introducidas las coordenadas, se traducen para poder pasarlas a la función `board_setup` del *abs_board*.

En este punto las 6 variantes muestran otro menú, que sirve para escoger si quieres jugar como multijugador o contra la IA (bot). Si escoges la primera opción puedes empezar a jugar directamente. En cambio, si escoges jugar contra la IA primero te pregunta en qué posición quieres jugar y qué nivel de dificultad quieres que tenga el bot.

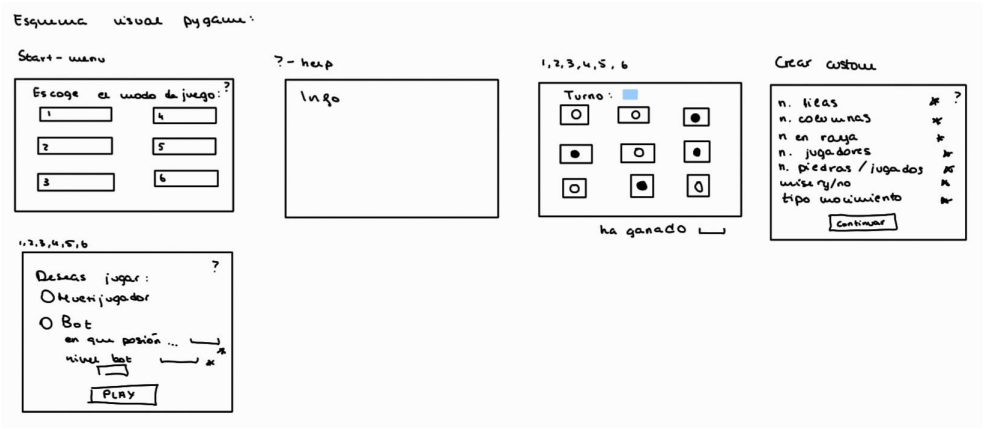
Para finalizar este fichero solo faltaba que el bucle principal del juego combinara todas las funciones y llevara el curso de la partida. Este bucle se encarga de gestionar los movimientos tanto del jugador como del bot, y comprueba para cada movimiento si alguien ha ganado el juego. Si no hay aún un ganador, se repite el bucle hasta que alguien gane. En el caso de que sí lo haya, se muestra el ganador por consola utilizando la función `print_winner`.

2.4. main_gui

En un principio no nos habíamos planteado hacer la versión del tres en raya usando la librería *pygame*, porque pensábamos que era opcional, y por eso decidimos ampliar mucho el driver de *main_txt*. Más tarde vimos que sí que era necesario hacer esta versión si queríamos hacer un trabajo completo y nos vimos obligados a implementar en *main_gui* la misma versión ampliada que habíamos hecho en *main_txt*.

Lo primero que pensamos al iniciar el desarrollo de esta versión fue de qué manera queríamos mostrar nuestro programa en la interfaz gráfica. Podíamos hacer que el jugador escogiera y configurara la partida a través de los mismos

menús que en *main_txt* y luego abrir el tablero correspondiente al juego de su elección con *pygame*. Sin embargo, no nos acabó de convencer esta opción porque pensamos que si hacíamos una interfaz gráfica, debería poder usarse todo desde la misma. Por ese motivo decidimos investigar cuál era la mejor forma de realizar cambios completos de pantallas en *pygame*. Antes de saber si era posible realizar estos cambios, hicimos un esquema inicial de las pantallas que queríamos crear:



Este esquema no es el definitivo, pero nos ayudó a saber qué es lo que queríamos que tuviera nuestro programa.

A continuación, empezamos a hacer nuestros primeros programas con *pygame* para aprender cómo funcionaba y la estructura que tenía. Investigamos también las opciones que teníamos para hacer las pantallas. Podíamos utilizar una librería nombrada *pygame-menu* para los menús, pero no nos convenció ni el diseño ni su funcionamiento, de manera que empezamos a buscar formas de cambiar la pantalla y vimos que lo más fácil sería establecer un color de fondo y, a cada cambio de pantalla, limpiarla rellenándola con el color de fondo y añadirle posteriormente todo lo necesario.

Delante de este planteamiento, iniciamos el proceso de programación del *main_gui*. Este proceso lo hicimos a distancia, comunicándonos a través de reuniones online, por lo que fue bastante diferente al de creación del *main_txt*. En vez de programarlo todo juntos, dado que Violeta ya había adquirido más destreza con la programación, empezamos a desarrollar código de manera paralela repartiendo trabajos.

La estructura del programa es la siguiente: primero hay una sección de funciones y luego una de variables globales. A continuación, se inicializa pygame, se definen todos los botones y se crea el diccionario `button_master`. Se definen algunas fuentes y, por último, el bucle principal del juego.

En el apartado de funciones, existen algunas que se relacionan directamente con las diferentes pantallas del programa. Son las siguientes:

- `start_screen`: pantalla de inicio con los modos de juego a escoger y un botón que da acceso al manual de juego.
- `help_screen`: pantalla que muestra el manual de juego.
- `select_bot_multi`: pantalla para escoger jugar multijugador o contra el bot.
- `setup_bot`: pantalla para escoger la dificultad del bot y el turno del jugador.
- `custom_board`: pantalla para customizar tu juego de tres en raya.
- `setup_game`: pantalla que contiene el tablero de la versión de juego escogida.

Todas estas funciones, menos la de `setup_game` cumplen la estructura de cambiar la variable de `actual_interface`, limpiar la pantalla, dibujar los rectángulos de los botones y, si hay texto, escribirlo.

Para explicar las siguientes funciones tenemos que hablar primero del `button_master`. Esta variable es un diccionario que contiene 6 llaves, tantas como las pantallas existen. El valor de estas llaves es una lista de tuplas; cada tupla contiene un botón de esa interfaz y una función asociada a ese botón. Como la librería *pygame* no dispone de botones, hemos tenido que crearlos. Cada botón es un rectángulo con un texto encima. Después, dentro del bucle principal, cuando se detecta el evento de clic en el ratón, el programa itera sobre las tuplas del `button_master` correspondientes a la interfaz actual, comprobando si el ratón está situado sobre alguno de los botones que contiene esa pantalla. Si es así, se llama a la función almacenada como segundo elemento de la tupla que contiene el botón. Por ese motivo, si la función

se tiene que llamar con parámetros, el `button_master` almacena funciones lambda que llaman a la función adecuada con el parámetro correspondiente. Debido a este planteamiento tuvimos que hacer una función para cada botón de nuestro programa. La mayoría de estas funciones se usan en las funciones `custom_board` y `setup_bot` para establecer los valores con los que se quiere jugar, pero también existen funciones para escoger el modo de juego, la opción multijugador o la opción contra el bot, poder navegar adelante y atrás en el manual de juego y volver a la pantalla de inicio. Estas últimas funciones, a diferencia de las otras, solo hacen un cambio de pantalla y de interfaz.

En las pantallas correspondientes a las funciones de `setup_bot` y de `custom_board` además de crear unas funciones para los botones de suma y resta de valores, tuvimos que crear dos funciones que actualizaran, cada vez que se hace clic, dichos valores. Estas funciones se llaman `update_values_bot` y `update_values_custom`.

Además de las funciones ya mencionadas, existen funciones ya no tan relacionadas con el diseño de la interfaz sino con el juego. Dos de ellas traducen las coordenadas del ratón para poder relacionarlas con las casillas del tablero. Otra se ocupa de la actualización y representación del tablero, teniendo en cuenta todos los posibles factores (si hay una casilla seleccionada, si hay un ganador, etc.). También hay una que se encarga de reiniciar el juego, y algunas otras menos importantes que se usan como funciones auxiliares.

El fichero acaba con el bucle principal. En la primera parte del bucle se comprueba si se ha hecho clic, y lo primero que se hace es revisar el `button_master`, como se ha explicado anteriormente. Luego, si no se ha pulsado ningún botón y está iniciada la partida, se hacen todas las gestiones necesarias para que la interacción se procese y el juego funcione correctamente.

Por último, independientemente de si se ha hecho clic en un botón o no, si está iniciada la partida y es el turno del bot, se inicia el proceso del movimiento de la IA.

3. Manual de usuario

El programa se puede ejecutar desde los dos drivers, tanto desde consola usando *main_txt* como a través de una interfaz gráfica usando *main_gui*.

3.1. Uso por consola

Se debe ejecutar el fichero *main_txt.py*. Se mostrará el menú principal que permite escoger (introduciendo números con el teclado y presionando enter) la variante a jugar, explicando brevemente cada una de ellas. Si se escoge una variante predefinida (es decir, no la custom) se mostrará un mensaje preguntando si se desea jugar en modo multijugador o contra la IA. En el caso de escoger multijugador, se inicia la partida, y en el caso de escoger contra la IA, se solicitará la posición en la que el usuario desea jugar en cada turno y el nivel que quiere que tenga el bot.

En caso de escoger la variante custom, se solicitarán los datos que definirán la partida. Estos datos tienen un formato y ciertas restricciones que se explican en un mensaje. Tras la customización de la variante, se mostrará, igual que en los otros casos, la pantalla que permite elegir si jugar multijugador o contra la IA.

Tras estas configuraciones previas se llega al inicio de la partida. En cada movimiento se muestra el tablero y el jugador que debe mover. Para introducir un movimiento, existen tres posibilidades:

1. La primera es que el modo de juego sea con gravedad, en cuyo caso se considera que todos los jugadores tendrán piedras infinitas. En este modo, solo se debe introducir un entero para realizar un movimiento. Este entero va desde el cero hasta el número de columnas menos uno, correspondiendo el cero a la columna de la izquierda, y el número máximo a la de la derecha.

2. Si el modo no es con gravedad y el jugador aún tiene piedras en la bolsa (bag), deberá seleccionar con dos coordenadas (enteros separados por un espacio) una casilla libre a la que mover. Las coordenadas empiezan en cero, y el origen de coordenadas es la esquina superior izquierda.

3. Si el modo no es con gravedad y el jugador ya ha puesto todas sus piedras en el tablero, para realizar un movimiento deberá escoger una casilla de origen usando las coordenadas explicadas en el apartado anterior y posteriormente una casilla de destino usando el mismo sistema de coordenadas.

Cuando un jugador hace raya o un jugador no tiene movimientos posibles, acaba el juego, y se da el ganador o los ganadores en base al modo de juego.

3.2. Uso de la interfaz gráfica

Se debe ejecutar el archivo *main_gui.py*. Se mostrará el menú principal, con las diferentes variantes para seleccionar. Si se selecciona una que no sea la variante custom, dejará escoger entre jugar multijugador o contra el bot. En caso de escoger jugar contra el bot, se deberá seleccionar la posición que tendrá el jugador en cada turno y la dificultad del bot. Si la variante seleccionada es la custom, se solicitarán los datos necesarios para crear la partida y después se mostrará la misma pantalla para seleccionar jugar contra el bot o en modo multijugador.

Una vez iniciada la partida, se deberá hacer clic con el ratón en la casilla en la que se quiere poner la piedra. Si el jugador no tiene piedras en la bolsa, deberá seleccionar una del tablero haciendo clic sobre ella y luego hacer clic sobre la casilla a la que la quiere mover. Si el modo de juego es con gravedad, hacer clic en cualquier casilla de una columna, pondrá la piedra en dicha columna.

Durante la partida se muestra un color en la zona inferior que indica el jugador que debe mover. Cuando acaba la partida, el color inferior indica el jugador o jugadores que han ganado. Al finalizar la partida, hacer clic en cualquier lugar de la pantalla redirigirá al menú principal.

En caso de tener alguna duda, existe un manual (botón con forma de interrogación) en el menú principal.

4. Conclusión

Como conclusión, este trabajo ha sido una experiencia enriquecedora tanto a nivel personal como técnico, ya que ambos hemos podido alcanzar nuestro principal objetivo de aprender y mejorar nuestras habilidades en programación. A pesar de los desafíos iniciales a los que nos enfrentamos al trabajar en equipo debido a la falta de experiencia en trabajo grupal de Pablo y el bajo nivel de programación inicial de Violeta, logramos superarlos y presentar finalmente un trabajo completo y de calidad. En general, consideramos que ha sido una experiencia valiosa y positiva para nosotros en múltiples aspectos.

Referencias

[1] *Pygame front page*. URL: https://www.pygame.org/docs/.

[2] *Where developers learn, share, and build careers*. URL: https://stackoverflow.com/.