Buscaminas de Cristian Rodríguez

Repositorio de git: https://github.com/CrisRPia/Buscaminas_Terminal

Análisis del problema

En esta sección se detallarán los objetivos del programa.

Problema

Se requiere diseñar un tablero de buscaminas básico que deberá poderse jugar sin errores. Esto implica diseñar una estructura de datos capaz de mantener toda la información posible de una celda, y guardar una matriz de celdas. Además, este proyecto se realizó con la intención de añadir comandos para facilitar el uso, renderizar el tablero como una sola ventana y con íconos precisos, y utilizar proyectos anteriores para generar tableros de juego disfrutables.

Posibles soluciones

Tablero

El tablero inicialmente se implementó como una lista de listas conteniendo objetos de la clase `Cell`, que almacena toda la información requerida para cada celda. Eventualmente esta estructura, que estaba suelta en `main.py`, se almacenó dentro de la clase `Minesweeper` que permite crear y simular tableros facilmente.

Interfaz

El usuario interactúa con el programa mediante comandos de línea. Esto es forzado por la premisa de la tarea. En base a esto se podría imprimir múltiples tableros en cadena para mostrar el estado de juego al usuario. Sin embargo, se optó por sobreescribir lo ya impreso moviendo el cursor mediante código ANSI para posibilitar animaciones sin parpadeo y para poder centrar elementos en pantalla. La ilegibilidad del código ANSI se simplifica mediante clases estáticas.

Comandos

Los comandos que se detallarán en la siguiente sección fueron preferidos frente a un único comando para descubrir una celda para evitar repetición monótona y proteger al usuario de leves errores de ingreso.

Diseño, descripción y justificación de la solución de la propuesta

En esta sección se detalla el diseño a mayor profundidad; se usarán ejemplos de código y se discutirá la distribución e implementación de diversos archivos y clases.

Tablero

El tablero se almacena en `Minesweeper`, y las celdas como `Cell`

Cell

```
class Cell():
    def __init__(self, value=0):
        if not (0 <= value <= 8):
            raise ValueError(f"Invalid value {value}. " +
                             "It must be an integer between 0 and 8.")
        self.flagged = False
        self.is_mine = False
```

```
        self.discovered = False
        self.value = value


    def __str__(self) -> str:
        if self.flagged:
            return f"{ANSIColors.GREEN}{icons.FLAG[icons.legacy]}"
        if not self.discovered:
            return f"{ANSIColors.WHITE}{icons.FOG[icons.legacy]}"
        if self.is_mine:
            return f"{ANSIColors.RED}{icons.MINE[icons.legacy]}"
        if self.value == 0:
            return f"{ANSIColors.WHITE}{icons.BLANK}"
        return ANSIColors.WHITE + str(self.value)
```

Figura 3.1: Clase Cell.

Como se aprecia en la figura 3.1, la clase cell contiene información sobre si la celda esta marcada como bandera, mina o descubierta, y un valor que representa la cantidad de minas vecinas. Desde esta clase no se puede calcular la cantidad de minas vecinas ya que la clase misma no tiene noción alguna del tablero, ese es el trabajo del contenedor de `Cell`. Además, se sobreescribe el método `__str__()` para que al imprimir la celda se aplique su ícono y su color correspondiente.

Minesweeper

```
class Minesweeper:
    board: List[List[Cell]] = []
    size: int
    mines: int
    solveable: bool


    def __init__(self, size: int, mines: int):
        self.size = size
        self.mines = mines
        self.solveable = False
        for i in range(ATTEMPTS):
            self.__prepare_board()
            self.reveal_help(auto_neighbours=False)
            cp = copy.deepcopy(self)
            if winnable(cp):
                self.solveable = True
                break
            elif i % 100 == 0:
                Renderer.print_board(cp.board, clear_prompt=False)
                Renderer.print_message(
                    f"Generando tablero{'.' * (i % 3000 // 1000 + 1)}\n" +
                    f"Intento {i} de {ATTEMPTS}." +
                    f"{i / ATTEMPTS * 100: 5.1f}%"
                )
        # This way it looks pretty
        if self.solveable:
            self.__animate_start()
    # Para ver el resto de la clase, abra minesweeper.py
```

Figura 3.2.1: Segmento fundamental de la clase Minesweeper.

En la figura 3.2 se puede ver tanto los atributos como el constructor de `Minesweeper`. Esta clase almacena el tablero, su tamaño, su cantidad de minas y si el mismo se puede resolver. Al iniciar la clase intenta generar un tablero ganable. Para esto se genera uno y se simula el juego. Si el juego no es ganable, simula nuevamente hasta encontrar uno ganable o exceder el límite de intentos. Los tableros se simulan mediante la función de la figura 3.2.2:

```
def winnable(game: Minesweeper) -> bool:
    size = len(game.board)

    # A game that's won from the begginning is pointless
    if game.won():
        return False

    # Initialize knowledge base
    ai = MinesweeperAI(size)
    for i in range(size):
        for j in range(size):
            cell = game.board[i][j]
            if cell.discovered:
                ai.add_knowledge((i, j), cell.value)
    for _ in range(size ** 2):
        # Make a move
        move = ai.make_safe_move()
        if move == None:
            if game.won():
                return True
            else:
                return False
        cell = game.board[move[0]][move[1]]
        cell.discovered = True
        if not ai.add_knowledge_safe(move, cell.value):
            return False
    return False
```

Figura 3.2.2: Función que determina si un tablero es ganable.

> *Aclaración:* La IA que juega al buscaminas fué desarrollada como parte de CS50AI. un curso de inteligencia artificial. Por lo tanto, el trabajo fue guiado y Cristian Rodríguez no reclama crédito por el algoritmo. La intervención del curso es inevitable ya que el alumno realizó el ejercicio antes siquiera de conocer la premisa de este proyecto.

Para entender esta función, primero es necesario entender el funcionamiento de la IA. La misma almacena toda la información inferible de las celdas descubiertas.

```
Tablero:
    A B C
    D 1 E
    F G H

Interpretación 1:
    Or(
        And(A, Not(B), Not(C), Not(D), Not(E), Not(F), Not(G), Not(H)),
        And(Not(A), B, Not(C), Not(D), Not(E), Not(F), Not(G), Not(H)),
        And(Not(A), Not(B), C, Not(D), Not(E), Not(F), Not(G), Not(H)),
        And(Not(A), Not(B), Not(C), D, Not(E), Not(F), Not(G), Not(H)),
```

```
        And(Not(A), Not(B), Not(C), Not(D), E, Not(F), Not(G), Not(H)),
        And(Not(A), Not(B), Not(C), Not(D), Not(E), F, Not(G), Not(H)),
        And(Not(A), Not(B), Not(C), Not(D), Not(E), Not(F), G, Not(H)),
        And(Not(A), Not(B), Not(C), Not(D), Not(E), Not(F), Not(G), H)
    )

Interpretación 2:
    { A, B, C, D, E, F, G, H } = 1
```

Figura 3.2.3: Posibles interpretaciones de la información de una celda.

La primera interpretación del tablero visible en la figura 3.2.3 es válida y sería definitivamente posible simular todas las tablas de verdad de la lógica demostrada e inferir toda la información posible del tablero actual. No obstante, dicho proceso aumenta en complejidad temporal exponencialmente cada vez que se añade nueva información, y por ello no se utilizó para representar la información del juego.

En su lugar, se utiliza la interpretación dos, representando cada inferencia como una instancia de la clase Sentence - puede ver el código de la clase en `Not_completely_mine/AI.py`. Mediante esta interpretación, se puede inferir toda la información disponible en el tablero mediante tres reglas:

1. De toda sentencia con cero minas se puede inferir que sus celdas son seguras.
2. De toda sentencia cuyo número de celdas equivale al de minas se puede inferir que todas sus celdas son minas.
3. De cada par de sentencias `set1 = count1` y `set2 = count2` en las cuales `set1` es un subset de `set2` se puede inferir que `set2 - set1 = count2 - count1`.

Una vez se determina que una celda es segura, se puede remover de todas las sentencias y volver a verificar las tres reglas. De forma similar, cuando se descubre una mina, se puede remover esa celda de todas las sentencias en la que se incluye y remover el valor de la sentencia en 1.

Volvamos a la figura 3.2.2

```python
def winnable(game: Minesweeper) -> bool:
    size = len(game.board)

    # A game that's won from the begginning is pointless
    if game.won():
        return False

    # Initialize knowledge base
    ai = MinesweeperAI(size)
    for i in range(size):
        for j in range(size):
            cell = game.board[i][j]
            if cell.discovered:
                ai.add_knowledge((i, j), cell.value)
    for _ in range(size ** 2):
        # Make a move
        move = ai.make_safe_move()
        if move == None:
            if game.won():
                return True
            else:
                return False
        cell = game.board[move[0]][move[1]]
        cell.discovered = True
        if not ai.add_knowledge_safe(move, cell.value):
```

```
            return False
    return False
```

Figura 3.2.2: Función que determina si un tablero es ganable.

`winnable` recibe un `Minesweeper` y determina si es ganable. Para esto, primero se cargan todas las celdas descubiertas en la base de conocimiento de `MinesweeperAI`, proceso mediante el cual la IA hace sus inferencias iniciales. Luego se le pide a la IA movimientos seguros y se le da la información obtenida de los movimientos. Si la IA es incapaz de realizar otro movimiento seguro y el juego no se ha ganado, entonces el tablero no es ganable.

Como se puede ver, este algoritmo afecta al parámetro inicial, por lo que es necesario pasarle una copia del tablero para que el jugador no gane automáticamente. Esta es una decisión de diseño intencionada; el mayor costo de la función suele ser el hecho de requerir una copia profunda de `Minesweeper`, por lo que se requiere que el usuario lo copie manualmente. De esta forma no hay sorpresas. Además, si el usuario desea mostrar el tablero ganado, puede hacerlo pasando un `Minesweeper` sin copiarlo.

Renderizado

La clase estática `Renderer` se utiliza como interfaz para abstraer las funciones que se encargan de controlar la disposición del tablero, el prompt y los mensajes en la terminal. Veamos un ejemplo:

```
class Renderer():
    # ...

    T = TypeVar('T')
    @staticmethod
    def print_prompt(printer: Callable[..., T],
                     warning: str = "") -> T:
        # Infer requirements
        term_size = ANSICursor.get_terminal_size()

        # Calculate end of board
        first_line = term_size.lines // 2 + 2

        # Erase old prompt
        ANSICursor.clear_from_row(first_line)

        # Render warning right below the board
        ANSICursor.move_cursor(first_line + 1, 0)
        print(warning)

        # Render main prompt
        ANSICursor.move_cursor(first_line + 3, 0)
        try:
            return printer()
        except KeyboardInterrupt:
            print("\n\n¡Hasta luego!")
            sys.exit()
```

Figura 3.3.1: Función que ubica el prompt en el lugar correcto.

La figura 3.3.1 muestra la interfaz para establecer el lugar del prompt del usuario. Fíjese que esta función en ningún momento imprime el prompt, sino que deja ello en manos del parámetro printer, y devuelve el valor de printer. De esta manera, uno puede usar sus propias funciones personalizadas para pedir información al usuario, y esta función meramente ubica el cursor en la posición correspondiente, imprimiendo una advertencia si es necesario.

¿Porqué es conveniente esto? Vea los casos de la figura 3.3.2:

```
# ...
# Set legacy mode
if Renderer.print_prompt(
    lambda: input("Modo de legado (Enter vacío para no aplicarlo): "),
):
    icons.legacy = True
# ...

def get_starters() -> Tuple[int, int]:
    size, mines = 0, 0
    while size < 3 or size > 10:
        size = Renderer.print_prompt(
            lambda: ask_int("Tamaño del tablero (de 3 a 10): ")
        )
    while mines < 1 or mines > size ** 2 - 1:
        mines = Renderer.print_prompt(
            lambda: ask_int(f"Cantidad de minas (de 1 a {size ** 2 - 1}): ")
        )
    return size, mines

# ...

def ask_int(s: str) -> int:
    """
        Imprime s y devuelve un entero que pide por consola al usuario.
    """
    try:
        return int(input(s))
    except Exception as e:
        if e is KeyboardInterrupt:
            raise e
        print("Error, el valor ingresado debe ser un número entero.")
        return ask_int(s)
```

Figura 3.3.2: Usos de print_prompt.

En main, `print_prompt` efectivamente es un input posicionado, nada especial. Ahora veamos `get_starters`. El mismo utiliza `ask_int`, que pide un ingreso al usuario hasta que este sea un número entero. El utilizar una función como parametro permite que el prompt imprima fácilmente lo que el usuario desee, y exija también lo que el usuario desee. De esta forma, se pueden utilizar funciones de impresión estándares sin tener que readaptarlas a este formato de impresión. Y, como bonus, al usar la declaración de genéricos, el *Protocolo de Servidor de Lenguaje* puede entender con exactitud el tipo de la función, incluso cuando este puede ser cualquiera.

Comandos

Los comandos disponibles al usuario son:

- Try: Descubrir una celda.
- Flag: Marcar una celda como mina.
- Expand: Descubrir o marcar las celdas alrededor de una descubierta si es fácilmente inferible.
- Help: Pide ayuda a la IA en un movimiento. Si es inferible, se le avisa al usuario de ello. Sino, se desbloquean celdas hasta que haya un movimiento inferible o se gane el juego.

```
    def act(self, action: Action, row: int, column: int) -> str:
        row = row - 1
        column = column - 1
        cell = self.board[row][column]

        # TRY
        if action == Action.TRY:
            if cell.flagged:
                warning = ("No se puede investigar una celda con bandera. " +
                           f"Ejecuta flag {column + 1} {row + 1} para desmarcarla.")
                return warning
            cell.discovered = True
            if cell.value == 0:
                self.discover_neighbours(row, column)

        # FLAG
        elif action == Action.FLAG:
            if cell.discovered:
                return f"¡La celda {column + 1} {row + 1} ya está descubierta!"
            cell.flagged = not cell.flagged

        # EXPAND
        elif action == Action.EXPAND:
            flags = self.neighbouring_flags(row, column)
            fog = self.neighbouring_fog(row, column)
            if (
                cell.discovered
                and cell.value != 0
                and (
                    fog + flags == cell.value
                    or flags == cell.value
                )
            ):
                # If any flags are wrong, lose the game
                if self.neighbouring_correct_flags(row, column) != flags:
                    self.expand_lose(row, column)
                    return ""
                self.clear_safes(row, column)
            else:
                return "Celda inválida"
        return ""
```

Figura 3.4.1: Método act.

La función de la figura 3.4.1 recibe, además de las coordenadas por las que actuar, un `Action`. Este tipo es un `Enum` que contiene las posibles acciones. Se utiliza esto en lugar de meras strings para evitar errores de escritura y para poder aceptar las acciones como parametro mas ninguna otra posibilidad en `act`.

El método `act` verifica que las coordenadas para cada acción sean válidas, en cuyo caso las ejecuta. El propósito de la mayoría de funciones es evidente con excepción de una, `self.discover_neighbours`, que contiene una animación. Veamos.

```
    def discover_neighbours(self, row: int, column: int, animated=True):
        """
            I was assisted throughout the troubleshooting of this algorithm by
            Phind. It also taught me about the existance of 'deque's.
```

```
            This algorithm reveals all obvious cells to prevent the user from
            having to input 8 commands each time they encounter a zero cell.
            It does so through layered and staggered Breadth First Search,
            meaning that it kinda reveals them by path distance from the
            starting point, in order to show a satisfying animation.
        """
        queue = deque([(row, column)])

        while queue:
            if animated:
                time.sleep(0.03)
            for _ in range(len(queue)):
                row, column = queue.popleft()
                for r in range(row - 1, row + 2):
                    for c in range(column - 1, column + 2):
                        # Skip the cell if r or c is out of bounds
                        if not (0 <= r < self.size) or not (0 <= c < self.size):
                            continue

                        cell = self.board[r][c]

                        # Skip the cell if it's the current cell, discovered, or a
mine
                        if (r == row and c == column) or cell.discovered or
cell.is_mine:
                            continue

                        cell.discovered = True

                        # If the cell value is 0, append it to the queue
                        if cell.value == 0:
                            queue.append((r, c))

            Renderer.print_board(self.board)
```

Figura 3.4.2: Método discover_neighbours

Como se puede ver en la figura 3.4.2, este método limpia recursivamente todas las celdas de valor cero en la frontera descubierta. Sin embargo, se hizo un esfuerzo extra para implementarla mediante BFS. Esto se debe a que el descubrimiento de la frontera puede ser animado, y se decidió que este algoritmo es el de apariencia más orgánica para este propósito.

Viendo la figura 3.4.1 nuevamente, uno puede notar que no hay caso para tratar `Action.HELP`. Esto se debe a que este caso tiene un método dedicado, `self.help`, que verá en la figura 3.4.3.

```
    def help(self) -> bool:
        size = len(self.board)
        output = True
        while True:
            # Initialize knowledge base
            ai = MinesweeperAI(size)
            for i in range(size):
                for j in range(size):
                    cell = self.board[i][j]
                    if cell.discovered:
                        ai.add_knowledge((i, j), cell.value)
```

```
            # Reveal cell if necessary
            if ai.make_safe_move() == None and not self.won():
                self.reveal_help()
                output = False
            else:
                break
        return output
```

Figura 3.4.3: Método help

En la definición del método se encuentra el motivo de su exclusión: devuelve un booleano. Pero, ¿cuál es el punto de este método? Al generar el tablero se garantiza que sea ganable, ¿no? Pues no. Pruebe generar un tablero de tamaño diez y cincuenta minas; si se le ha generado un tablero ganable, se enfrenta usted a un milagro estadístico. Pero que no se pueda garantizar un juego ganable no es motivo para garantizar un juego aburrido. `help` le da la información justa al usuario para jugar un tablero ridículo sin ser injusto. El juego, en cierta forma, se vuelve un buscaminas inverso. En lugar de buscar las pocas minas, se ha de buscar las pocas celdas seguras.

Manual de usuario

Requisitos

Para ejecutar el juego deberá tener python 3.10 en adelante instalado junto con todas sus librerías estándar (que suelen venir junto a la instalación normal). Además, deberá ejecutarlo desde una terminal que sea compatible con el código ANSI y tenga una fuente monoespaciada. Este programa ya fué probado en Alacritty y la nueva terminal de Windows, ambas funcionando correctamente. Por último, si usted desea ejecutarlo sin el modo de legado, deberá de instalar una Nerd Font en su terminal. El programa fue diseñado específicamente para `JetBrainsMono Nerd Font`, pero cualquier Nerd Font monoespaciada sin ligaturas debería funcionar.

> ATENCIÓN: El programa no espera redistribución del tamaño de ventana, y asume espacio igual o mayor a dos tableros, tanto vertical como horizontalmente.

Ejecución

En su terminal, entre al directorio de los archivos del programa y ejecute `python main.py` o `python3 main.py`.

Juego

El objetivo del buscaminas es descubrir todas las celdas que no son minas de un tablero. Al descubrir una celda segura, podrá ver la cantidad de celdas vecinas que son minas. Usted deberá utilizar esta información para inferir todas las minas.

Una vez ejecuta el juego, verá un menú para elegir el modo gráfico. Si usted ha instalado una Nerd Font en su terminal, presione enter sin ingresar texto, o habiendo borrado todo texto ingresado. De lo contrario, ingrese algún caracter y presione enter para jugar en el modo de legado. Todas las peticiones de ingreso se confirman con enter.

Ahora tendrá que ingresar el tamaño del tablero y acto seguido la cantidad de minas. Lo recomendado es que el 20% del tablero sean minas. De esta forma se podrá generar un tablero que no requiera adivinar mientras sigue siendo desafiante.

Una vez ingresada la configuración deseada, el programa generará tableros hasta encontrar uno justo o llegar a 100.000 intentos sin suerte. Este proceso puede llevar algunos minutos si la configuración tiene demasiadas minas, recuerde que siempre puede salir del programa apretando el par de teclas `ctrl+c`.

Si el programa puede generar un tablero justo, verá lo siguiente debajo del tablero:

```
Este tablero se puede resolver con seguridad.

Acción (man para explicación):
```

Y ejecutar man mostrará lo siguiente:

```
Sintaxis: acción coordenada_horizontal coordenada_vertical

Ejemplo: flag 5 5

Posibles acciones:

    flag: Marcar la celda con una bandera. Es útil para recordar las minas ya
          inferidas.

    help: Pedirle ayuda al programa en un movimiento. La IA determinará si es
          posible realizar un movimiento seguro y lo dirá. De no ser posible,
          se desbloquearán celdas hasta que un movimiento seguro sea posible.

    try: Descubre la celda. Si la celda es una mina, el juego se pierde. Si la
         celda no tiene minas vecinas, efectuará try en las celdas vecinas. Esta
         acción se ejecuta por defecto; '5 5' es equivalente a 'try 5 5'. Una
         vez se descubre la última celda sin minas, el juego se gana.

    expand: Al usarse en una celda descubierta con valor numérico, descubre o
            aplica bandera a las celdas vecinas correctas si el número de
            celdas vecinas sin descubrir es equivalente al valor de la celda
            electa menos la cantidad de banderas vecinas.
```

Estos son los comandos que tiene disponible.

De no poder generar un tablero justo, considere ejecutar `help`. Este comando le dará suficiente información para poder inferir alguna o algunas celda o celdas.

Finalmente, al terminar una partida se le preguntará si desea continuar jugando. En caso negativo, ingrese algún texto y presione Enter.

Conclusiones del trabajo

Siendo el único integrante del grupo, Cristian Rodríguez desarrolló este proyecto placenteramente, únicamente reprochando su incapacidad para añadir soporte de ratón al juego. Fue ampliamente satisfactoria la integración de la IA en el proyecto, siendo que incluso se logró optimizar ampliamente sobre el diseño original.

Durante el desarrollo del proyecto ha sido necesario más de un replanteo considerable del código y su distribución en clases. Sin embargo, no considero esto como un fallo dado que el conocimiento ganado de los primeros bocetos fueron de gran ayuda en encaminar hacia el diseño final. Todavía existen aspectos a mejorar, como replantearse que aspectos de `ANSICursor` deberían ir realmente en `Renderer` (la rara distribución nace del ingreso tardío de `Renderer` al proyecto).

Pese a esto, siendo que el estado actual del programa es perfectamente funcional, este parece un punto oportuno para darle fin a su desarrollo; se han completado todos los objetivos iniciales del programa, se ha aspirado a más, y se a conseguido más. Ahora es momento de buscar más que este programa.

Ayudas externas

El proyecto fué desarrollado en su mayoría en base a los conocimientos existentes de Cristian Rodríguez. Sin embargo, Phind se utilizó como suplemento y cazador de errores en ciertas funciones. Todas las funciones en las que se utilizó para más ayuda que simplemente ser un motor de búsqueda de documentación se encuentran comentadas indicando ello.

Además, Not_completely_mine/AI.py se basa considerablemente en un proyecto del mismo alumno de CS50AI.