به نام خدا

تمرین امتیازی - هوش مصنوعی **نیایش خانی ۹۹۵۲۱۲۳۵**

همانطور که در داک ذکر شده، در فایل های main و game تغییری ایجاد نمی کنیم.
در ai_solution فانکشن های موجود را کامل کردم که در ادامه به توضیح آن می پردازیم :
در فانکشن **init** این کلاس از متغیر های زیر استفاده شده است :

```
    def __init__(self, game):
        """
        Initialize a GameSolution instance.
        Args:
            game (Game): An instance of the Water Sort game.
        """
        self.game = game  # Store the game instance
        self.solution_found = False  # Flag to indicate if a solution is found
        self.moves = []  # List to store the sequence of moves
        self.visited_states = set()  # Set to store visited states for DFS
```

در متغیر game نمونه بازی ارائه شده در زمان اولیه را نگه می دارد.
سپس در solution_found یک flag داریم که برای نشان دادن این است که آیا راه حلی پیدا شده است یا خیر.
در moves هم لیستی برای ذخیره حرکات انجام شده داریم که منجر به حل می شوند.
و در آخر در visited_states ، استیت هایی که ویزیت شده اند را ذخیره می کنیم تا از محاسبات اضافی جلوگیری کنیم.

```
def is_victory(self, tube_colors):
    """
    Check if the current tube configuration is a victory.
    Args:
        tube_colors (List[List[int]]): Current state of the tubes.
    Returns:
        bool: True if the game is in a winning state, else False.
    """
    return self.game.check_victory(tube_colors)
```

در این تابع چک می کنیم که برنده شده ایم یا خیر. به این صورت که در ورودی tube_colors را می گیرد که در واقع وضعیت فعلی لوله ها است. سپس با استفاده از فانکشن check_victory برنده یا بازنده بودن چک می شود. اگر برنده شده باشیم، true و در غیر این صورت false برمی گرداند.

```python
def get_possible_moves(self, tube_colors):
    """
    Generate all possible moves from the current tube configuration.
    Args:
        tube_colors (List[List[int]]): Current state of the tubes.
    Returns:
        List[Tuple[int, int]]: List of possible moves (source, destination).
    """
    moves = []
    for i in range(len(tube_colors)):
        if tube_colors[i]:  # Source tube must have at least one color
            for j in range(len(tube_colors)):
                # Destination tube must be different, not full, and either empty or have the same color on top
                if i != j and (not tube_colors[j] or (len(tube_colors[j]) < self.game.NColorInTube and tube_colors[j][-1] == tube_colors[i][-1])):
                    moves.append((i, j))  # Append the move (source, destination)
    return moves
```

سپس در فانکشن **get_possible_moves** مانند قبل در اینپوت tube_colors را می گیریم. بوسیله این ورودی، تمامی حرکاتی که طبق قوانین بازی بوده باشند را ایجاد می کند. در آخر لیستی از تاپل ها که نشان دهنده حرکت های معتبر هستند را بر می گرداند که در آن هر تاپل یک جفت منبع، مقصد دارد.

در فانکشن بعدی به این صورت عمل می کنیم :

```python
def make_move(self, tube_colors, move):
    """
    Apply a move to the tube configuration.
    Args:
        tube_colors (List[List[int]]): Current state of the tubes.
        move (Tuple[int, int]): Move to apply (source, destination).
    Returns:
        List[List[int]]: New state of the tubes after the move.
    """
    new_tube_colors = [tube[:] for tube in tube_colors]  # Deep copy to avoid modifying the original state
    src, dst = move  # Unpack the move
    color = new_tube_colors[src].pop()  # Remove the top color from the source tube
    new_tube_colors[dst].append(color)  # Add the top color to the destination tube
    return new_tube_colors
```

در این فانکشن با داشتن وضعیت فعلی لوله ها و حرکتی که قرار است روی آنها اعمال شود، حرکت مشخص شده را انجام می دهد و یک استیت جدید ایجاد می کند. در آخر استیت جدید لوله ها پس از اعمال حرکت را بر می گرداند.

حال به سراغ پیاده سازی **solve** می رویم. (به دلیل طولانی بودن کد، در اینجا قرار نگرفته است.) بخش های این تابع :

tube_colors: حالت اولیه لوله ها.

تابع dfs: یک تابع کمکی که جستجوی عمقی را در ابتدا انجام می دهد.

current_colors: وضعیت فعلی لوله ها در پیمایش DFS.

path: مسیر حرکات انجام شده برای رسیدن به current_colors.

اگر حالت فعلی یک حالت برنده باشد، راه حل پیدا می شود و مسیر ثبت می شود.

این تابع به صورت بازگشتی تمام حرکات معتبر را بررسی می کند و حالت های جدید را به visited_states اضافه می کند.

اگر حرکتی به حالت برنده منتهی شود، بازگشت متوقف می شود و مسیر برمی گردد.
initial_state: حالت اولیه برای تغییرناپذیری به یک تاپل تبدیل می شود.
self.visited_states: حالت اولیه را ذخیره می کند تا از بازدید مجدد آن جلوگیری کند.

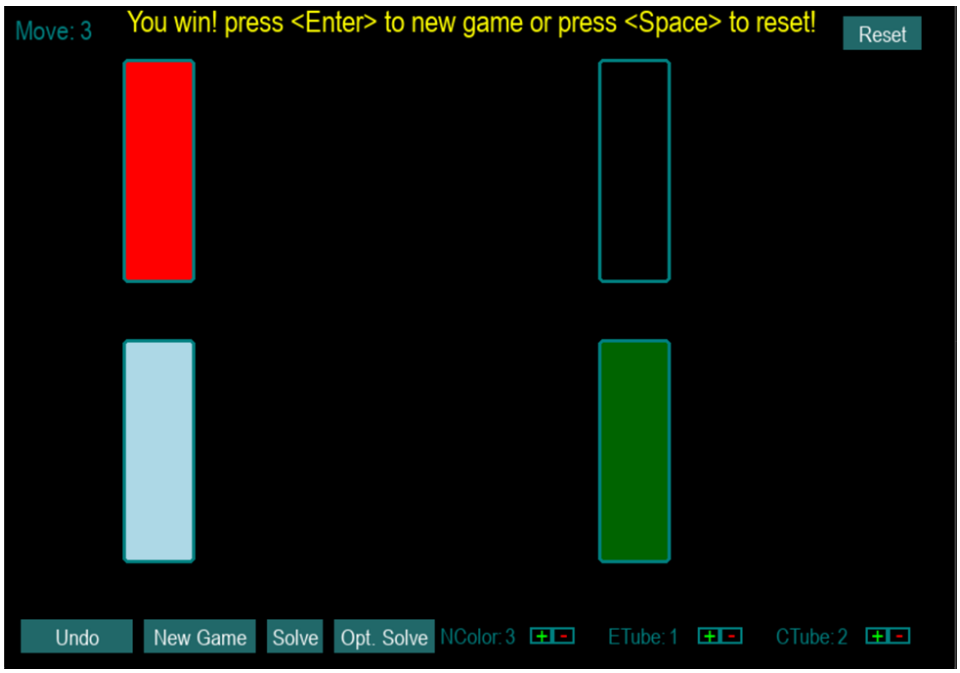
تابع بعدی **heuristic** است که شامل این موارد است :
tube_colors: وضعیت فعلی لوله ها.
Returns : فاصله تخمینی تا وضعیت هدف.
تابع هیوریستیک مجموع لوله هایی را محاسبه می کند که رنگ یکنواخت ندارند (لوله های اشتباه) و لوله هایی که به طور کامل با یک رنگ پر نشده اند (لوله های ناقص).
این هیوریستیک به اولویت بندی استیت هایی که در جستجوی A* به هدف نزدیک تر هستند کمک می کند.

و در بخش حل کردن بهینه مسئله در فانکشن **optimal_solve** تابع **a_star_search** را داریم که در واقع یک تابع کمکی که جستجوی A* را انجام می دهد. در این فانکشن داریم :
start_colors: حالت اولیه لوله ها.
initial_state: حالت اولیه برای تغییرناپذیری به یک تاپل تبدیل می شود.
g: دیکشنری که هزینه را از استیت شروع به هر استیت ذخیره می کند.
h: دیکشنری که تخمین هزینه هیوریستیک را در هدف هر استیت ذخیره می کند.
f: دیکشنری که کل هزینه تخمینی (g + h) را برای هر استیت ذخیره می کند.
priority_queue: صف اولویت برای کاوش استیت ها به ترتیب افزایش مقادیر
visited: برای ذخیره وضعیت های بازدید شده تنظیم کنید.
این تابع حالت ها را از صف اولویت باز می کند، تمام حرکات ممکن را اعمال می کند، هزینه ها را محاسبه می کند و حالت های جدید را تا زمانی که حالت هدف پیدا می شود به صف push می کند.

با ران کردن main بازی شروع می شود. علاوه بر امکان بازی کردن برای خود یوزر، امکان حل کردن مسئله به صورت عادی و حل کردن به صورت بهینه نیز وجود دارد که طبیعتا در حالت بهینه کد تلاش می کند تا راهی که تعداد حرکات کمتری نیاز دارد را در صورت وجود پیدا کند.
مثال :

به عنوان مثال راه حل زیر در حالت حل کردن عادی به دست آمده است که از ٤ حرکت تشکیل شده است.

با انتخاب گزینه پیدا کردن راه حل بهینه برای همین یازی، به این نتیجه خواهیم رسید :



همانطور که مشاهده می شود در این حالت با انجام ۳ حرکت، برنده شده ایم.