گزارش کار تمرین عملی :

صبا باقری _ ۴۰۱۵۲۱۰۷۵

تابع solve :

```
if(self.ws_game.check_victory(current_state)):
            self.solution_found = True
            return
```

در این قسمت ابتدا با استفاده از تابع check_victory در game.py چک میکنیم که بازی به پایان رسیده است یا خیر. اگر بازی به پایان رسیده بود از تابع return میکنیم.

در غیر اینصورت:

```
for this in range(len(current_state)):
    for that in range(len(current_state)):
```

در بین tube های موجود در current_state نگاه میکنیم تا دو tube مناسب پیدا شود. This ایندکس source tube و that ایندکس distination tube ، در current_state است.

```
if this != that:
   length = 1
   chain = True
   color_on_top = 100
   color_to_move = 100
   if len(current_state[this]) > 0:
      color_to_move = current_state[this][-1]
      for col in range(1, len(current_state[this])):
          if chain:
             if current_state[this][-1 - col] == color_to_move:
                 length += 1
             else:
                 chain = False
```

در این قسمت ، رنگی که باید از current_state[this] خارج شود و طول آن محاسبه میشود و در color_to_move رنگ موردنظر ذخیره میشود. همچنین طول آن هم در length ذخیره میشود. متغیر chain هم برای این است که بررسی کنیم آیا طول آب انتخابی برای انتقال ۱ است یا بیشتر.

```
if len(current_state[that]) < self.ws_game.NColorInTube:
   if len(current_state[that]) == 0:
      color_on_top = color_to_move
   else:
      color_on_top = current_state[that][-1]
```

در این قسمت هم در current_state[that] tube بررسی میکنیم که آیا فضای کافی برای انتقال رنگ وجود دارد یا خیر. و آخرین رنگ موجود در آن را در color_on_top ذخیره میکنیم.

در قسمت بعدی در یک شرط if بررسی میکنیم که شرایط برای انتقال فراهم باشد و color_to_move == color_on_top باشد. و اگر این شرایط برقرار بود ، ابتدا یک کپی از current_state در new_st ذخیره میکنیم و انتقال رنگ را در new_st اعمال میکنیم و اگر این new_st در self.visited_tubes ها نبود ، آن را اضافه میکنیم و current_state را برابر با new_st قرار میدهیم. سپس دوباره تابع solve را برای current_state جدید صدا میزنیم و در نهایت اگر self.solution_found=true باشد ، true ریترن میکنیم در غیر اینصورت ، حرکت را از self.moves حذف میکنیم.

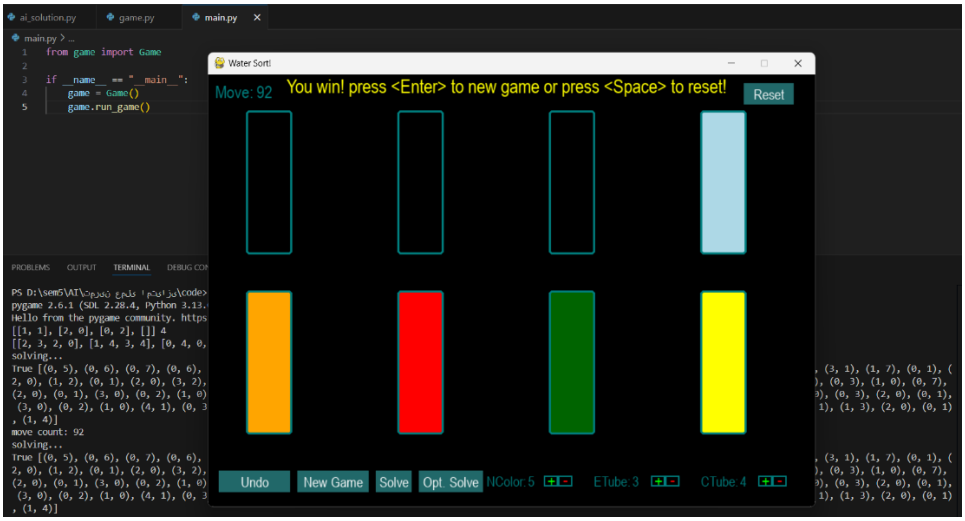
```
If length <= self.ws_game.NcolorInTube - len(new_st[that]):
   if length > 1 :
     for I in range(length):
        new_st[that].append(color_to_move)
    else:
        new_st[that].append(color_to_move)
    if length>1:
      for I in range(length):
        new_st[this].pop(-1)
   else:
       new_st[this].pop(-1)
if tuple(map(tuple, new_st)) not in self.visited_tubes:
   self.visited_tubes.add(tuple(map(tuple, new_st)))
   self.moves.append((this,that))
   current_state = new_st
   self.solve(current_state)
   if self.solution_found:
      return True
  else:
      self.moves.remove((this,that))
      return False
```

اجرای تابع solve :

قبل از شروع بازی :

بعد از اتمام بازی :

تعداد حرکات : ۹۲



تابع optimal_solve :

توضیح تابع heuristic :

```
def heuristic(self, states):
        ms = 0
        for tube in states:
            if len(tube) > 0:
                unique_colors = set(tube)
                if len(unique_colors) > 1:
                    ms += len(tube)
        return ms
```

در هر tube در states که در واقع همان current_state است تعداد mismatch در رنگ ها را پیدا میکنیم. و در واقع برای تخمین از این استفاده میکنیم که کدام state دارای tube های مرتب تر است و آن را انتخاب میکنیم.

توضیح تابع optimal_solve :

```
        sp = PriorityQueue()
        h = {}
        g = {tuple(map(tuple, current_state)): 0}
        self.visited_tubes = set()

        sp.put((self.heuristic(current_state), current_state, []))
```

sp یک priority queue برای ذخیره state ها است ، که برای هر عضو آن ، بخش اول prioriy بخش دوم خود state و بخش سوم self.moves مربوط به آن است. h, g هم دو دیکشنری برای ذخیره سازی مقدار ها هستند. در ابتدا هم state فعلی و heuristic آن و حرکات که مربوط به آن است را در sp ذخیره میکنیم.

سپس در حلقه while :

```
while(not sp.empty()):
      _, current_state, current_moves = sp.get()
      if self.ws_game.check_victory(current_state):
         self.moves = current_moves
         self.solution_found = True
         return
```

تا وقتی که sp خالی نشده باشد ادامه میدهیم. بالاترین priority را با دستور get میگیریم.

با check_victory چک میکنیم که آیا بازی به پایان رسیده است یا خیر . و اگر به پایان رسیده بودیم ، حرکاتی که انجام داده ایم و از sp گرفتیم و در current_move ذخیره کرده ایم را در self.moves ذخیره میکنیم و self.solution_found را برابر با true قرار میدهیم و return میکنیم.

اگر که به پایان بازی نرسیده بودیم :

```
 self.visited_tubes.add(tuple(map(tuple, current_state)))
```

state فعلی را در self.visited_tube ذخیره میکنیم.

```
for this in range(len(current_state)):
    for that in range(len(current_state)):
        if this != that:
           length = 1
           chain = True
           color_to_move = 100
           if len(current_state[this]) > 0:
              color_to_move = current_state[this][-1]
              for col in range(1, len(current_state[this])):
                  if chain and current_state[this][-1 - col] == color_to_move:
                     length += 1
                  else:
                     chain = False
         if len(current_state[that]) < self.ws_game.NColorInTube and
len(current_state[this]) > 0:
             if (len(current_state[that]) == 0 or current_state[that][-1] ==
color_to_move):
                new_st = deepcopy(current_state)
                for i in range(length):
                    if len(new_st[that]) < self.ws_game.NColorInTube:
                       new_st[that].append(color_to_move)
```

```
                    new_st[this].pop(-1)
```

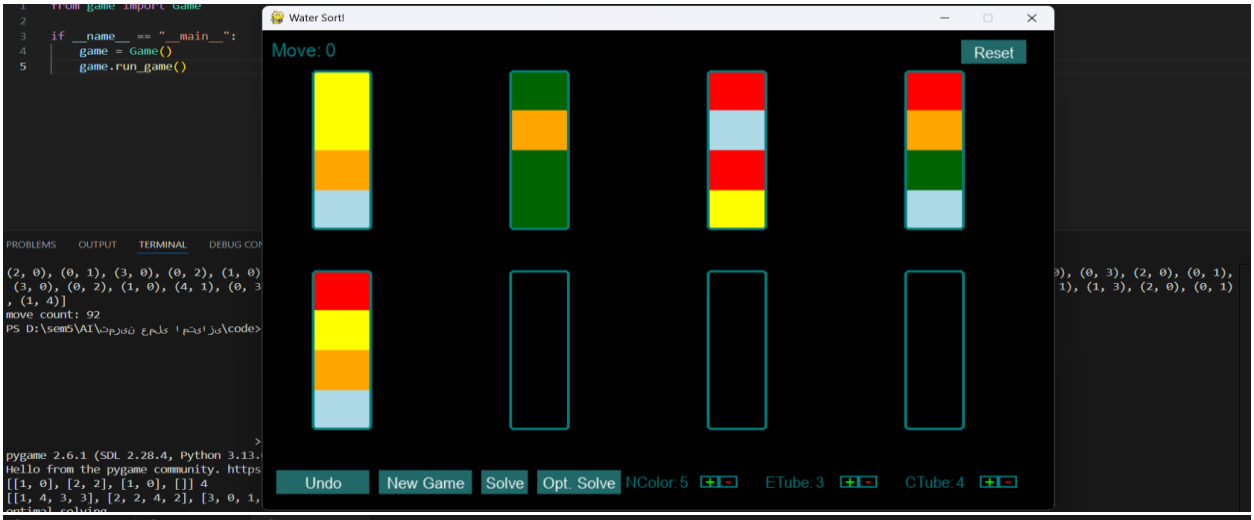
بخش بالا کاملا مشابه با تابع solve است. تنها تفاوت در بخش زیر است :

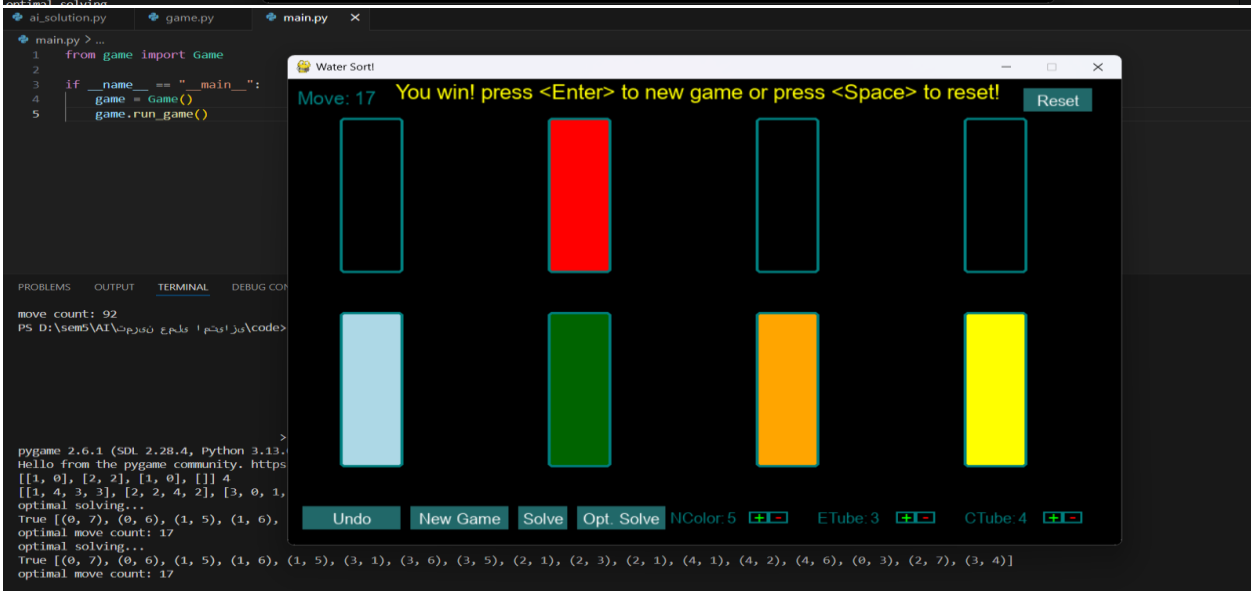
```
if tuple(map(tuple, new_st)) not in self.visited_tubes:
    new_g = g[tuple(map(tuple, current_state))] + 1
    new_h = self.heuristic(new_st)
    priority = new_g + new_h
    sp.put((priority, new_st, current_moves + [(this, that)]))
    g[tuple(map(tuple, new_st))] = new_g
```

اگر new_st در self.visited_tube نبود ، new_g را با اضافه کردن یک به g استیت فعلی ، new_h با محاسبهheuristic و priority را با جمع این دو محاسبه میکنیم و در sp به همراه move ذخیره میکنیم. و مقدار g مرتبط با state جدید را در دیکشنری g آپدیت میکنیم.

اجرای تابع optimal_solve با همان تنظیمات ولی رنگ های متفاوت :

تعداد حرکت : ۱۷





نتیجه : با تنظیمات مشابه :

تعداد حرکات در تابع solve : ۹۲

تعداد حرکات در تابع optimal_solve : ۱۷