علیرضا خانی

توضیحات در مورد مسئله ۸ پازل به وسیله الگوریتم تپه نوردی با ۲ هیوریستیک

[ریپازیتوری گیتهاب]

⚙ نحوه تنظیم روی لپ تاپ خودتان

```
* Clone this repo to your local machine.
* Using the terminal, navigate to the cloned repo.
* Creation of virtual environments
* Active Your "Activate.ps1"
* Upgrade your pip
pip install streamlit
streamlit run main.py
```

در ابتدا یک «آموزش تصویری» داده ام و سپس «توضیحات مربوط به کد» را از صفحه ۵ به بعد ارائه داده ام.

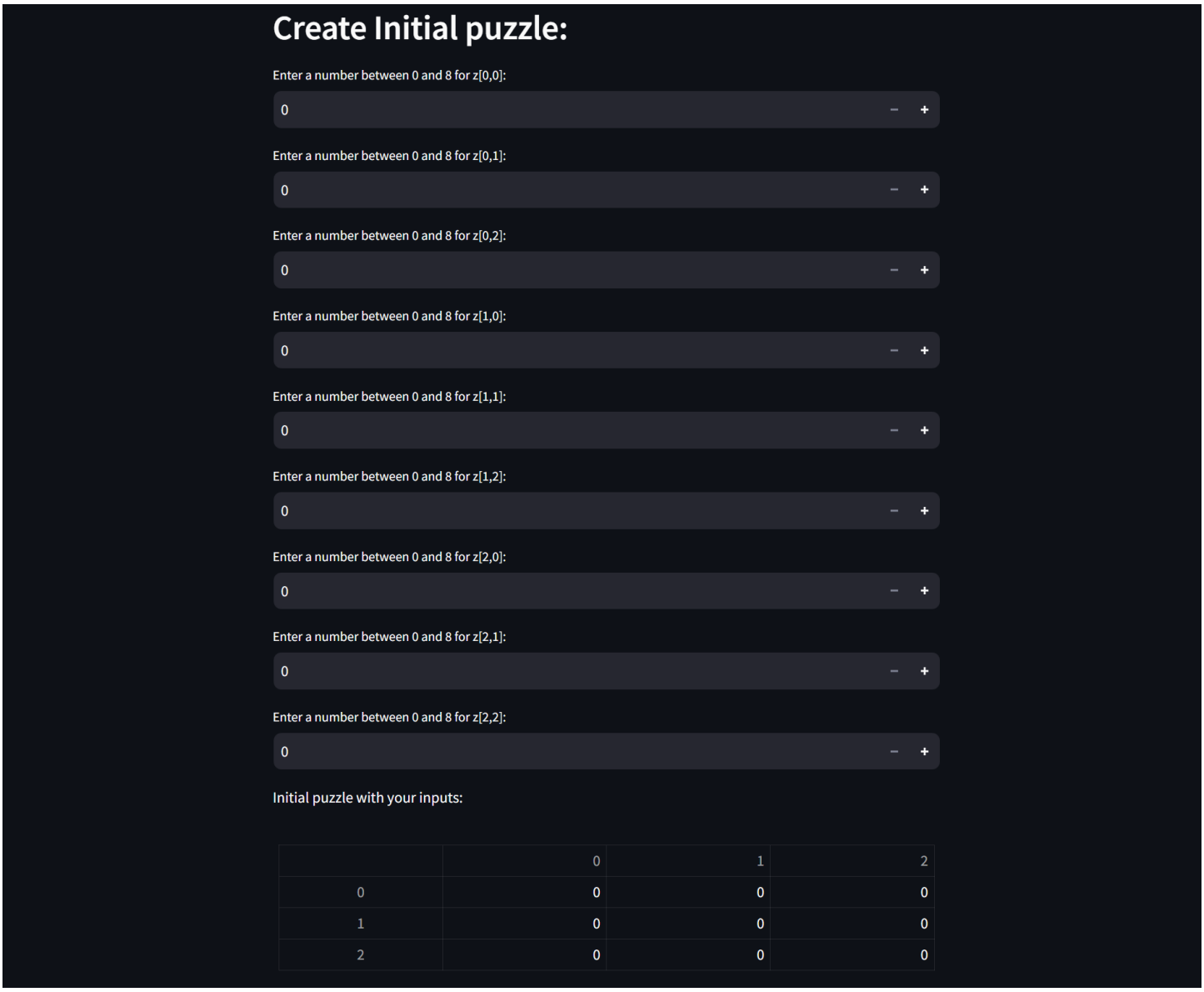
🏫 آموزش تصویری:

بعد از اجرای " streamlab run main.py " در کامندلاین خود، مرورگر شما باز خواهد شد و به آدرس localhost معمولا با پورت 8501 صفحه ای مانند عکس زیر نمایش میابد. در ابتدا Goal State را نمایش داده ام.

که همانطور که مشاهده می‌شود اسم هر درایه با z[i,j] مشخص شده است.

بنابر این برای وارد کردن مقادیر ۸ پازل خود در فیلد های زیر باید عدد مورد نظر خودتان را قرار دهید. (توجه داشته باشید که باید عدد بین ۰ تا ۸ باشد و از قرار دادن عدد تکراری اکیدا پرهیز کنید)

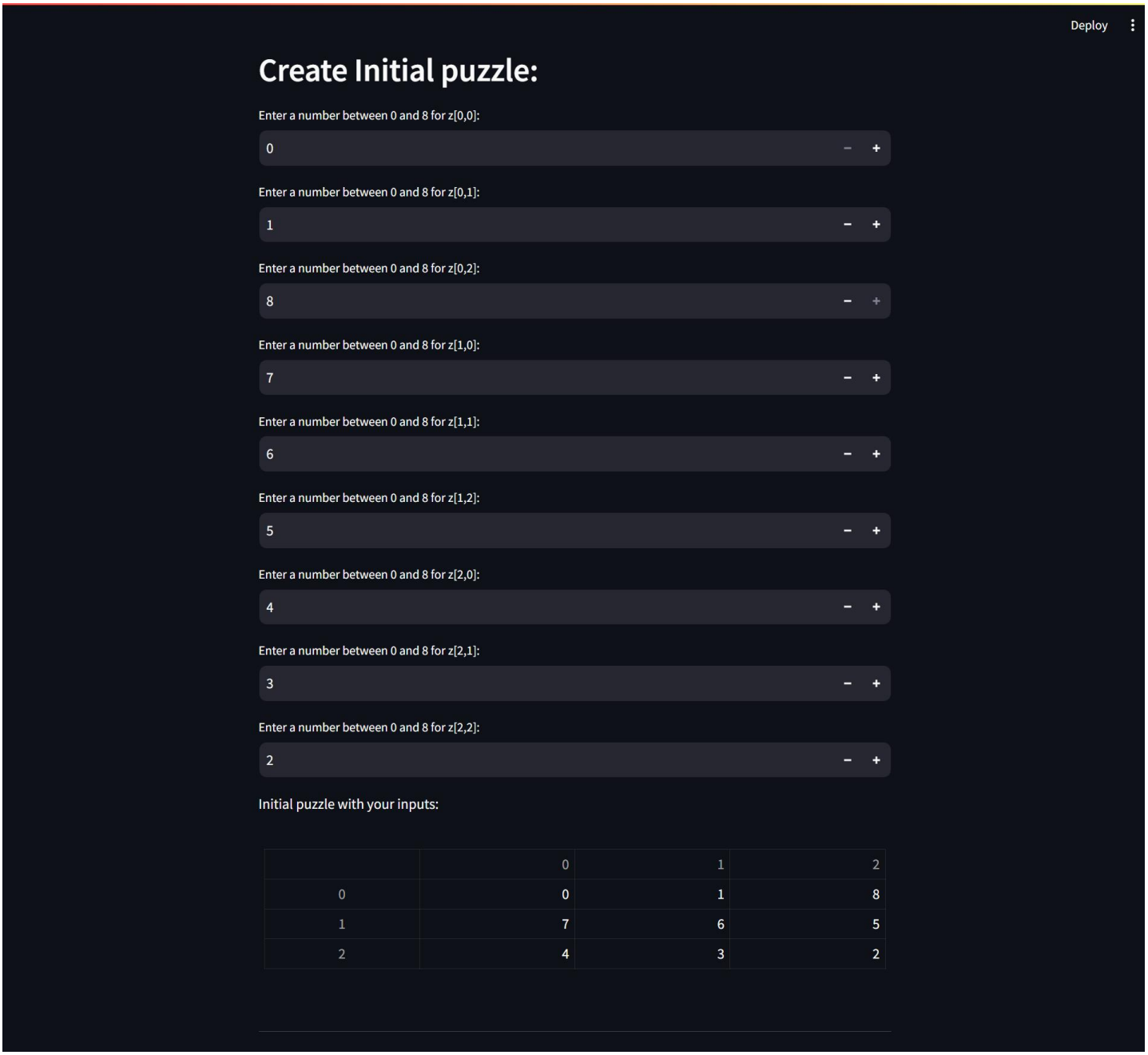
و بعد از وارد کردن مقادیر مورد نظر خود، زیر خط **Initial puzzle with your input** جدولی که وارد کردید را مشاهده خواهید کرد.



برای مثال من اعداد زیر را واردکردم و همانطورکه مشاهده می‌کنید **initial puzzle** من نمایش داده می‌شود: ⬇

0	**1**	**8**
7	**6**	**5**
4	**3**	**2**

جدول بالا را با جدول پایین تصویر زیر مقایسه کنید و نحوه واردکردن هر درایه را متوجه خواهید شد.



حال برای h1، جدول نهایی را بعد از انجام الگوریتم (Final Puzzle mode) را نشان داده ام و بعد از آن تعداد حرکت برای رسیدن به حالت نهایی (Number of moves made) را مشخص نموده ام و همچنین مسیر (Path) انجام این الگوریتم هم آورده شده است:

Results for h1 heuristic:

Final puzzle mode:

	0	1	2
0	1	0	8
1	7	6	5
2	4	3	2

Number of moves made: 1

Path:

step 0:

	0	1	2
0	0	1	8
1	7	6	5
2	4	3	2

step 1:

	0	1	2
0	1	0	8
1	7	6	5
2	4	3	2

و دقیقا برای h2 نیز از همین سبکِ نمایش استفاده شده است

📝 توضیحات مربوط به کد:

```python
import copy
import streamlit as st
```

وارد کردن توابع مورد نیاز

```python
def create_initial_state():
    initial_state = [[0] * 3 for _ in range(3)]
    index = 0
    for i in range(3):
        for j in range(3):
            num = st.number_input(f"Enter a number between 0 and 8 for z[{i},{j}]: ", min_value=0, max_value=8, key=f"num_{i}_{j}")
            if 0 <= num <= 8 and num not in initial_state:
                initial_state[i][j] = num
            else:
                st.error("❌Invalid input❌. Please enter a number between 0 and 8 for the next try! 👀")
                st.stop()
            index += 1
    return initial_state
```

تابع `create_initial_state` برای ایجاد یک وضعیت اولیه (initial state) برای یک پازل از کاربر استفاده می‌کند. در اینجا توضیحات جزیی به کارهایی که این تابع انجام می‌دهد، ارائه شده است:

۱ :`initial_state = [[0] * 3 for _ in range(3)]` : .یک لیست دوبعدی به نام `initial_state` ایجاد می‌شود که ابعاد آن ۳ در ۳ است. این لیست با مقادیر اولیه ۰ پر می‌شود.

۲ :`index = 0` :.یک متغیر `index` ایجاد می‌شود که در ادامه برای شمارش موقعیت‌های درخواستی از کاربر مورد استفاده قرار می‌گیرد.

۳ :`for i in range(3):` : .یک حلقه for برای تعیین مقدارهای سطرها (i) شروع می‌شود.

٤ :`for j in range(3):` : .یک حلقه for دیگر برای تعیین مقدارهای ستون‌ها (j) شروع می‌شود.

۵ :`num = st.number_input(f"Enter a number between 0 and 8 for z[{i},{j}]: ", min_value=0, max_value=8, key=f"num_{i}_{j}")` : .از کاربر خواسته می‌شود تا یک عدد صحیح بین ۰ تا ۸ وارد کند. این ورودی توسط `st.number_input` در Streamlit گرفته می‌شود و در `num` ذخیره می‌شود.

٦ :`if 0 <= num <= 8 and num not in initial_state:` : .شرطی بررسی می‌شود که اگر `num` بین ۰ تا ۸ باشد و قبلاً در `initial_state` قرار نگرفته باشد، مقدار `num` در ماتریس `initial_state` در موقعیت مشخص شده (i, j) ذخیره می‌شود.

۷ :`else:` : .در صورتی که شرط بالا برقرار نباشد) یعنی `num` خارج از محدوده یا قبلاً در `initial_state` وجود داشته باشد(، وارد بلوک `else` می‌شود.

۸ :`st.error("❌Invalid input❌. Please enter a number between 0 and 8 for the next try! 👀")` : .یک پیام خطا با استفاده از `st.error` به کاربر نمایش داده می‌شود.

۹ :`st.stop()` : .اجرای برنامه متوقف می‌شود و کاربر باید دوباره اجرای برنامه را آغاز کند.

۱۰ :`index += 1` : .مقدار `index` یک واحد افزایش می‌یابد.

۱۱ . حلقه‌های `for` به ازای ستون‌ها و سطرها ادامه پیدا می‌کنند.

۱۲. `return initial_state`: نهایتاً ماتریس `initial_state` که حاوی اعدادی است که کاربر وارد کرده است، به عنوان وضعیت اولیه برنامه بازمی‌گردانده می‌شود.

```python
def print_table(state):
    custom_css = """
            <style>
                .tabless {
                    text-align: center;
                }
                td, th {
                 text-align: center;
             }
            </style>
        """
    st.markdown(custom_css, unsafe_allow_html=True)
    st.table(state)
```

Python

این کد یک تابع به نام `print_table` ایجاد می‌کند که برای نمایش یک جدول در Streamlit استفاده می‌شود. در زیر توضیحات جزیی به این کد ارائه شده است:

۱ :`custom_css` : .یک رشته HTML است که حاوی کدهای CSS است. این کدها به وسیله تگ `<style>` تعریف شده‌اند.

۲ :`.tabless { text-align: center; }` : .یک کلاس به نام `tabless` تعریف شده است که مشخص می‌کند که متن (text) داخل عناصر با این کلاس (مانند `p`, `div` و...) باید وسط چین باشد.

۳ :`td, th { text-align: center; }` : .مشخص می‌کند که متن داخل سلول‌های `td` و `th` باید وسط چین باشد.

٤ :`st.markdown(custom_css, unsafe_allow_html=True)` : .با استفاده از`st.markdown` ، کدهای CSS به صورت HTML در Streamlit نمایش داده می‌شوند. مهم است که `unsafe_allow_html=True` را اضافه کنید تا Streamlit به شما این امکان را بدهد که کدهای HTML را اجرا کنید (به دلیل محدودیت‌های امنیتی، این امکان به صورت پیش‌فرض غیرفعال است).

۵ :`st.table(state)` : .از `st.table` برای نمایش جدول استفاده شده است که `state` به عنوان ورودی گرفته می‌شود. این تابع در Streamlit یک جدول از اطلاعات را نمایش می‌دهد.

در نهایت، این کد با اعمال کدهای CSS، متن داخل سلول‌ها را وسط چین می‌کند تا جدول به نمایش در Streamlit بیاید.

```python
def heuristic_1(state, goal_state):
    count = sum(1 for i in range(3) for j in range(3) if state[i][j] != goal_state[i][j] and state[i][j] != 0)
    return count
```

Python

درمورد heuristic_1 باید بگم که این هیوریستیک براساس کمترین مکان های اشتباه کار می کند. به عبارتی تمامی ورودی هارا چک میکند که آیا ورودی ها در جای درست خودشان طبق goal_state قرار دارند یا خیر.

این کد یک تابع به نام `heuristic_1` برای محاسبه مقدار هیوریستیک مربوط به الگوریتم جستجوی مطلع (Informed Search) استفاده می‌شود. در اینجا توضیحات جزیی به این کد ارائه شده است:

۱ :`def heuristic_1(state, goal_state)` : .تابع `heuristic_1` با دو ورودی `state` و `goal_state` تعریف شده است.

۲ :`count = sum(1 for i in range(3) for j in range(3) if state[i][j] != goal_state[i][j] and state[i][j] != 0)` : .این خط کد یک عبارت تکراری با استفاده از `sum` اجرا می‌کند تا تعداد خانه‌هایی را که در `state` با `goal_state` متفاوت هستند (برای مکان‌های غیر صفر) را بشمارد. به این تعداد خانه‌ها "مکان‌های اشتباه" می‌گویند.

۳ : `state[i][j] != goal_state[i][j] and state[i][j] != 0` : .این شرط بررسی می‌کند که آیا مقدار متناظر با خانه مورد نظر در `state` با مقدار متناظر در `goal_state` متفاوت است و همچنین مقدار متناظر با آن خانه در `state` صفر نیست. این به این دلیل است که در حالتی که مقدار خانه در `state` صفر باشد، به معنای این است که جایگاه خالی است و نباید به عنوان مکان اشتباه محسوب شود.

٤ : `sum(1 for i in range(3) for j in range(3) if ... )` : .از `sum` برای جمع‌آوری تعداد مکان‌های اشتباه استفاده شده است.

٥ : `return count` : .تعداد مکان‌های اشتباه به عنوان مقدار هیوریستیک برگردانده می‌شود.

به طور خلاصه، این هیوریستیک با شمارش تعداد مکان‌هایی که مقدار آنها در `state` با مقدار متناظر در `goal_state` متفاوت است، کمک می‌کند تا میزان اشتباه در حالت فعلی را اندازه‌گیری کند.

```python
def heuristic_2(state, goal_state):
    distance = sum(abs(row - i) + abs(col - j) for i in range(3) for j in range(3) if state[i][j] != 0
                   for row, col in [divmod(state[i][j] - 1, 3)])
    return distance
```

Python

این تابع به نام `heuristic_2` یک هیوریستیک مبتنی بر فاصله منهتن (Manhattan) برای الگوریتم‌های جستجوی مطلع (Informed Search) استفاده می‌شود. در زیر توضیحات جزیی به این تابع ارائه شده است:

١ : `def heuristic_2(state, goal_state)` : .تابع `heuristic_2` با دو ورودی `state` و `goal_state` تعریف شده است.

٢ : `distance = sum(abs(row - i) + abs(col - j) for i in range(3) for j in range(3) if state[i][j] != 0 for row, col in [divmod(state[i][j] - 1, 3)])` : .این خط کد یک عبارت تکراری با استفاده از `sum` اجرا می‌کند تا فاصله منهتن را محاسبه کند. این فاصله توسط مجموع مطلق اختلافات در مختصات سطر و ستون هر خانه از ماتریس `state` تا مختصات مربوط به آن خانه در `goal_state` به دست می‌آید.

٣ : `for i in range(3) for j in range(3) if state[i][j] != 0` : .یک حلقه دوتایی برای تمام خانه‌های غیر صفر در `state` شروع می‌شود.

٤ : `for row, col in [divmod(state[i][j] - 1, 3)]` .یک حلقه تکتایی برای تعیین مختصات مربوط به مقدار خانه (`state[i][j]`) در `goal_state` استفاده می‌شود. مختصات این خانه در `goal_state` از روی مقدار خانه در `state` به دست می‌آید.

٥ : `abs(row - i) + abs(col - j)` .اختلاف مطلق مختصات سطر و ستون بین موقعیت خانه در `state` و موقعیت متناظر در `goal_state`.

٦ : `sum(...)` : .جمع تمام مقادیر فاصله منهتن محاسبه شده.

٧. : `return distance` : مقدار فاصله منهتن به عنوان مقدار هیوریستیک برگردانده می‌شود.

```python
def generate_neighbors(current_state):
    neighbors = []
    zero_row, zero_col = next((i, j) for i, row in enumerate(current_state) for j, val in enumerate(row) if val == 0)
    for move in [(0, -1), (0, 1), (-1, 0), (1, 0)]:
        new_row, new_col = zero_row + move[0], zero_col + move[1]
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_state = copy.deepcopy(current_state)
            new_state[zero_row][zero_col], new_state[new_row][new_col] = new_state[new_row][new_col], \
                new_state[zero_row][zero_col]
            neighbors.append(new_state)
    return neighbors
```

تابع `generate_neighbors` در یک الگوریتم جستجوی محلی(Local Search) ، به خصوص در الگوریتم Hill Climbing کاربرد دارد. این تابع وظیفه تولید حالات مجاور یک حالت فعلی (current state) را برعهده دارد. در اصطلاحات الگوریتم‌های جستجو، این حالات مجاور "مهمان‌ها" یا "همسایگان" نامیده می‌شوند. در اینجا توضیحات جزیی به کد ارائه شده است:

۱ `def generate_neighbors(current_state)` : .تابع `generate_neighbors` با ورودی `current_state` تعریف شده است.

۲ `neighbors = []` : .یک لیست خالی به نام `neighbors` برای ذخیره حالات مجاور ایجاد می‌شود.

۳ `zero_row, zero_col = next((i, j) for i, row in enumerate(current_state) for j, val in enumerate(row) if val == 0)` : .مختصات سطر و ستون خانه خالی (صفر) در `current_state` را به دست می‌آورد. این اطلاعات برای جابه‌جایی مکان خالی با موقعیت‌های مجاور آن استفاده می‌شود.

٤ `for move in [(0, -1), (0, 1), (-1, 0), (1, 0)]:` : .یک حلقه `for` برای تعیین حرکت‌های ممکن به سمت‌های بالا، پایین، چپ، و راست (مشخص شده توسط `(۰, -۱)`, `(۰, ۱)`, `(-۱, ۰)`, و `(1, 0)`) شروع می‌شود.

۵ `new_row, new_col = zero_row + move[0], zero_col + move[1]` : .با استفاده از مختصات مکان خالی و حرکت تعیین شده، مختصات مکان جدید را محاسبه می‌کند.

٦ `if 0 <= new_row < 3 and 0 <= new_col < 3:` : .یک شرط بررسی می‌کند که آیا مکان جدید داخل محدوده ماتریس ۳ در ۳ است یا خیر.

۷ `new_state = copy.deepcopy(current_state)` : .یک کپی عمیق از `current_state` ایجاد می‌شود تا تغییرات در مکان‌های جدید اعمال شود.

۸ `new_state[zero_row][zero_col], new_state[new_row][new_col] = new_state[new_row][new_col], new_state[zero_row][zero_col]` : .مکان خالی با مکان جدید جابه‌جا می‌شود.

۹ `neighbors.append(new_state)` : .حالت جدید به لیست `neighbors` اضافه می‌شود.

۱۰ `return neighbors` : .لیست حالات مجاور تولید شده به عنوان خروجی تابع برگردانده می‌شود.

در کل، تابع `generate_neighbors` وظیفه ایجاد و بازگرداندن حالات مجاور یک حالت فعلی را دارد. این کار به کمک جابه‌جایی مکان خالی با مکان‌های مجاور آن انجام می‌شود.

```python
def hill_climbing(state, heuristic, goal_state):
    current_state = state
    path = [current_state]
    moves = 0
    while True:
        neighbors = generate_neighbors(current_state)
        neighbor_scores = [(neighbor, heuristic(neighbor, goal_state)) for neighbor in neighbors]
        neighbor_scores.sort(key=lambda x: x[1])
        if neighbor_scores[0][1] >= heuristic(current_state, goal_state):
            break
        else:
            current_state = neighbor_scores[0][0]
            path.append(current_state)
            moves += 1
    return current_state, moves, path
```
Python

این کد یک تابع به نام `hill_climbing` برای اجرای الگوریتم جستجوی تپه‌ای (Hill Climbing) با استفاده از یک هیوریستیک مشخص شده است. در زیر توضیحات جزیی به این کد ارائه شده است:

۱ `def hill_climbing(state, heuristic, goal_state)`: . تابع `hill_climbing` با سه ورودی تعریف شده است (`state`: حالت اولیه، `heuristic`(تابع هیوریستیک مورد استفاده) و `goal_state` حالت هدف.)

۲ `current_state = state`: . حالت فعلی با حالت اولیه مساوی می‌شود.

۳ `path = [current_state]`: . یک لیست به نام `path` ایجاد می‌شود که حالت اولیه به عنوان اولین عنصر آن افزوده می‌شود. این لیست برای ذخیره مسیر پیموده شده توسط الگوریتم استفاده می‌شود.

٤ `moves = 0`: . تعداد حرکات انجام شده به صورت اولیه صفر تنظیم می‌شود.

٥ `while True:`: . یک حلقه بی‌پایان آغاز می‌شود که تا زمانی که یک شرط خاص فعال باشد، ادامه می‌یابد.

٦ `neighbors = generate_neighbors(current_state)`: . حالات مجاور حالت فعلی با استفاده از تابع `generate_neighbors` تولید می‌شوند.

۷ `neighbor_scores = [(neighbor, heuristic(neighbor, goal_state)) for neighbor in neighbors]`: . برای هر حالت مجاور، امتیاز محاسبه شده توسط تابع هیوریستیک برای آن حالت ذخیره می‌شود.

۸ `neighbor_scores.sort(key=lambda x: x[1])`: . لیست حالات مجاور بر اساس امتیازها مرتب می‌شود.

۹ `if neighbor_scores[0][1] >= heuristic(current_state, goal_state):`: . اگر امتیاز حالت مجاور با بالاترین امتیاز حالت فعلی یا مساوی باشد، حلقه متوقف می‌شود.

۱۰ `else:`: . در غیر این صورت:

- `current_state = neighbor_scores[0][0]`: حالت فعلی با حالت مجاور با بالاترین امتیاز جابه‌جا می‌شود.
- `path.append(current_state)`: حالت فعلی به مسیر اضافه می‌شود.
- `moves += 1`: تعداد حرکات انجام شده افزایش می‌یابد.

۱۱ `return current_state, moves, path`: حالت نهایی، تعداد حرکات انجام شده، و مسیر پیموده شده تا حالت هدف به عنوان خروجی تابع برگردانده می‌شود.

در کل، تابع `hill_climbing` الگوریتم جستجوی تپه‌ای را با استفاده از تابع هیوریستیک مشخص شده اجرا می‌کند تا به حالت هدف برسد. این الگوریتم در هر مرحله بهترین حالت مجاور را انتخاب می‌کند و اگر به حالتی برسد که امتیاز آن با امتیاز حالت فعلی یا بیشتر باشد، الگوریتم متوقف می‌شود.

```python
def main():
    st.title("Puzzle Solver with hill climbing")
    st.header("Goal State:")
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    print_table(goal_state)

    st.header("Template Puzzle:")
    template_state = [["z[0,0]", "z[0,1]", "z[0,2]"], ["z[1,0]", "z[1,1]", "z[1,2]"], ["z[2,0]", "z[2,1]", "z[2,2]"]]
    print_table(template_state)

    st.write("\n--------------------------\n")
    st.header("Create Initial puzzle:")
    initial_state = create_initial_state()
    st.write("Initial puzzle with your inputs:")
    print_table(initial_state)
    initial_state_h1 = copy.deepcopy(initial_state)
    result_h1, moves_h1, path_h1 = hill_climbing(initial_state_h1, heuristic_1, goal_state)
    st.write("\n--------------------------\n")
    st.header("Results for h1 heuristic:")
    st.write("Final puzzle mode:")
    print_table(result_h1)
    st.write("Number of moves made:", moves_h1)
    st.write("Path:")
    for i, state in enumerate(path_h1):
        st.write(f"step {i}:")
        print_table(state)

    initial_state_h2 = copy.deepcopy(initial_state)
    result_h2, moves_h2, path_h2 = hill_climbing(initial_state_h2, heuristic_2, goal_state)
    st.write("\n--------------------------\n")
    st.header("Results for h2 heuristic:")
    st.write("Final puzzle mode:")
    print_table(result_h2)
    st.write("Number of moves made:", moves_h2)
    st.write("The path obtained:")
    for i, state in enumerate(path_h2):
        st.write(f"step {i}:")
        print_table(state)
```

Python

```python
if __name__ == "__main__":
    main()
```

Python