BÁO CÁO N_QUEEN

**TASK1: Steepest-ascend Hill Climbing Search**

**1. Mục tiêu**

- Cài đặt thuật toán steepest-ascent hill climbing (tìm kiếm leo đồi chọn bước tốt nhất).
- **Ý tưởng:** từ một trạng thái khởi tạo (bàn cờ), ở mỗi bước đánh giá toàn bộ tập các di chuyển lân cận (tức là thay đổi vị trí hàng của một quân trong mỗi cột) và chọn di chuyển làm giảm số xung đột nhiều nhất. Lặp lại cho đến khi không còn di chuyển cải thiện nào (cực trị cục bộ) hoặc đạt số xung đột = 0 (giải pháp).

**2. Code:**

```python
def steepest_ascent_hill_climbing(start_board, max_iters=1000):
    """Thuật toán steepest-ascent hill climbing.
    Ở mỗi bước, đánh giá tất cả các di chuyển thay đổi vị trí 1 quân trong một cột
    và chọn di chuyển cải thiện nhất (giảm số xung đột nhiều nhất).

    Trả về: final_board (np.array), history (danh sách số xung đột theo thời gian)
    """
    board = np.array(start_board).copy()
    n = len(board)
    history = [conflicts(board)]
    it = 0
    while it < max_iters:
        it += 1
        current_conf = conflicts(board)
        best_conf = current_conf
        best_board = None
```

```python
# đánh giá tất cả di chuyển đơn quân
        for col in range(n):
            orig = board[col]
            for row in range(n):
                if row == orig:
                    continue
                board[col] = row
                c = conflicts(board)
                if c < best_conf:
                    best_conf = c
                    best_board = board.copy()
            board[col] = orig
```

```python
# Nếu không tìm được cải thiện, đã tới cực trị cục bộ
        if best_board is None:
            break
        board = best_board
        history.append(best_conf)
        if best_conf == 0:
            break
    return board, history
```

```python
# Kiểm tra nhanh (ví dụ nhỏ)
if __name__ == '__main__':
    b = random_board(8)
    final, hist = steepest_ascent_hill_climbing(b)
    print('Xung đột ban đầu:', conflicts(b), 'Xung đột cuối:', conflicts(final))
```

**3. Kết quả:**

```
Xung đột ban đầu: 8 Xung đột cuối: 0
```

**4. Mô tả:**

**- Hàm và đầu vào/đầu ra**

- Hàm: **steepest_ascent_hill_climbing(start_board, max_iters=1000)**

**- Input:**

- **start_board:** mảng (list hoặc np.array) kích thước n, phần tử i là hàng của quân ở cột i (0-indexed).
- **max_iters:** số bước tối đa để tránh vòng lặp vô hạn.

**- Output:**

- **final_board:** mảng numpy chứa trạng thái cuối cùng (sau khi dừng).
- **history:** danh sách số xung đột theo từng bước (dùng để vẽ đồ thị/thống kê).

**- Luồng xử lý**

- Sao chép **start_board** vào biến **board** để không làm thay đổi đầu vào.
- Khởi tạo **history** bằng số xung đột ban đầu **(conflicts(board))**.
- Lặp **while** cho tới **max_iters**:
  - Tính **current_conf = conflicts(board)**.
  - Khởi **best_conf = current_conf**, **best_board = None**.
  - Duyệt mọi cột **col** từ **0..n-1**:
    - Lưu vị trí ban đầu **orig = board[col]**.
    - Duyệt mọi hàng **row** trong cột đó (0..n-1):
      - Nếu **row == orig**, bỏ qua (không phải là di chuyển).
      - Thay tạm **board[col] = row**, tính **c = conflicts(board)**.
      - Nếu **c < best_conf**, cập nhật **best_conf = c** và lưu **best_board = board.copy()**.
    - Khôi phục **board[col] = orig**.
  - Sau khi duyệt hết tất cả các di chuyển đơn quân:
    - Nếu **best_board is None** (không có di chuyển nào giảm conflict), **break** (đã tới cực trị cục bộ).
    - Gán **board = best_board**, **append best_conf** vào **history**.
    - Nếu **best_conf == 0**, **break** (đã tìm nghiệm tối ưu).
- Trả về **board, history**.

**5. Ghi chú về phức tạp và tối ưu**

**Phức tạp:** mỗi bước cần kiểm tra O(n * n) láng giềng (n cột × n hàng khả dĩ). Tính toán xung đột hiện tại (cấp toàn bộ) trong hàm conflicts đang là O(n) sử dụng đếm hàng/đường chéo (thực ra conflicts chạy O(n) sau khi đã tính tần suất). Do đó chi phí mỗi bước là ~O(n^2). Số bước tối đa phụ thuộc vào cấu trúc bài toán; worst-case là max_iters.

**Tối ưu tiềm năng:**

- Thay vì gọi conflicts toàn phần cho mỗi láng giềng, có thể duy trì thông tin bổ sung để cập nhật xung đột cục bộ nhanh hơn khi thay đổi 1 ô (cập nhật incremental).
- Hoán đổi logic để tránh copy toàn bộ bảng quá nhiều, hoặc hạn chế đánh giá các hàng hợp lý.

## TASK 2: Stochastic Hill Climbing 1 (Stochastic HC 1)

### 1. Mục tiêu:

- Cài đặt biến thể "stochastic hill climbing" mà tại mỗi bước thu thập toàn bộ các di chuyển cải thiện (tức là mọi láng giềng có số xung đột nhỏ hơn trạng thái hiện tại), rồi chọn ngẫu nhiên một trong số các di chuyển đó để thực hiện.
- So với steepest-ascent (chọn di chuyển tốt nhất), phương pháp này chọn ngẫu nhiên trong các cải thiện, giúp giảm bias và có thể tránh một vài bẫy cục bộ nhỏ.

### 2. Code:

```python
def stochastic_hill_climbing_all_neighbors(start_board, max_iters=1000):
    """Thuật toán stochastic HC 1:
    Ở mỗi bước, thu thập tất cả các di chuyển cải thiện và chọn ngẫu nhiên một trong số đó.
    Trả về board cuối cùng và lịch sử số xung đột.
    """
    board = np.array(start_board).copy()
    n = len(board)
    history = [conflicts(board)]
    it = 0
    while it < max_iters:
        it += 1
        current_conf = conflicts(board)
        improving_moves = []  # danh sách tuple (conf, col, row)
        for col in range(n):
            orig = board[col]
            for row in range(n):
                if row == orig:
                    continue
                board[col] = row
                c = conflicts(board)
                if c < current_conf:
                    improving_moves.append((c, col, row))
            board[col] = orig
        if not improving_moves:
            break
```

```python
# chọn ngẫu nhiên trong số các di chuyển cải thiện
        c, col, row = random.choice(improving_moves)
        board[col] = row
        history.append(c)
        if c == 0:
            break
    return board, history
```

```python
# kiểm tra nhanh
if __name__ == '__main__':
    b = random_board(8)
    final, hist = stochastic_hill_climbing_all_neighbors(b)
    print('Xung đột ban đầu:', conflicts(b), 'Xung đột cuối:', conflicts(final))
```

**3. Kết quả:**

```
Xung đột ban đầu: 8 Xung đột cuối: 1
```

**4. Mô tả:**

**- Hàm và đầu vào/đầu ra**

- Hàm: **stochastic_hill_climbing_all_neighbors(start_board, max_iters=1000)**

**- Input:**

- **start_board:** mảng vị trí hàng của hậu theo cột.
- **max_iters:** giới hạn số bước.

**- Output:**

- **final_board:** trạng thái cuối cùng (numpy array).
- **history:** danh sách số xung đột sau mỗi lần chọn di chuyển (bắt đầu với xung đột ban đầu).

**- Luồng xử lý**

- Sao chép **start_board** sang **board**, lấy **n = len(board)**.
- Khởi **history = [conflicts(board)]**.
- Lặp while tới max_iters:
  - Tính **current_conf = conflicts(board)**.
  - Tạo danh sách **improving_moves = []** để lưu các di chuyển làm giảm số xung đột.
  - Duyệt mọi cột col:
    - Lưu vị trí ban đầu **orig = board[col]**.
    - Duyệt mọi hàng row khác orig:
      - Thử **board[col] = row**, tính **c = conflicts(board)**.
      - Nếu **c < current_conf**, **append (c, col, row)** vào **improving_moves**.
    - Khôi phục **board[col] = orig**.
- Nếu **improving_moves** rỗng: dừng (đã tới cực trị cục bộ).
- Ngược lại: chọn ngẫu nhiên một **tuple (c, col, row)** từ **improving_moves**.
  - Thực hiện **board[col] = row.**
  - **Append c** vào **history**.
  - Nếu c == 0, dừng (đã tìm nghiệm).
- Trả về **board, history**.

**5. Điểm chú ý về thực thi và hiệu năng**

**Chi phí:** giống steepest về mặt xét toàn bộ láng giềng, vì vẫn duyệt O(n^2) láng giềng mỗi bước để phát hiện improving_moves. Tuy nhiên, sau khi thu thập, chỉ thực hiện 1 di chuyển.

**Ưu điểm:**

Tính ngẫu nhiên giúp thay đổi quá trình tìm kiếm, có thể vượt một vài bẫy cục bộ mà steepest hay rơi vào do luôn chọn cùng một bước tốt nhất.

**Nhược điểm:**

Vẫn tốn chi phí đánh giá vì phải thu thập tất cả các láng giềng; nếu muốn nhanh hơn, có thể dùng first-choice (Task 3) để sinh từng láng giềng ngẫu nhiên.

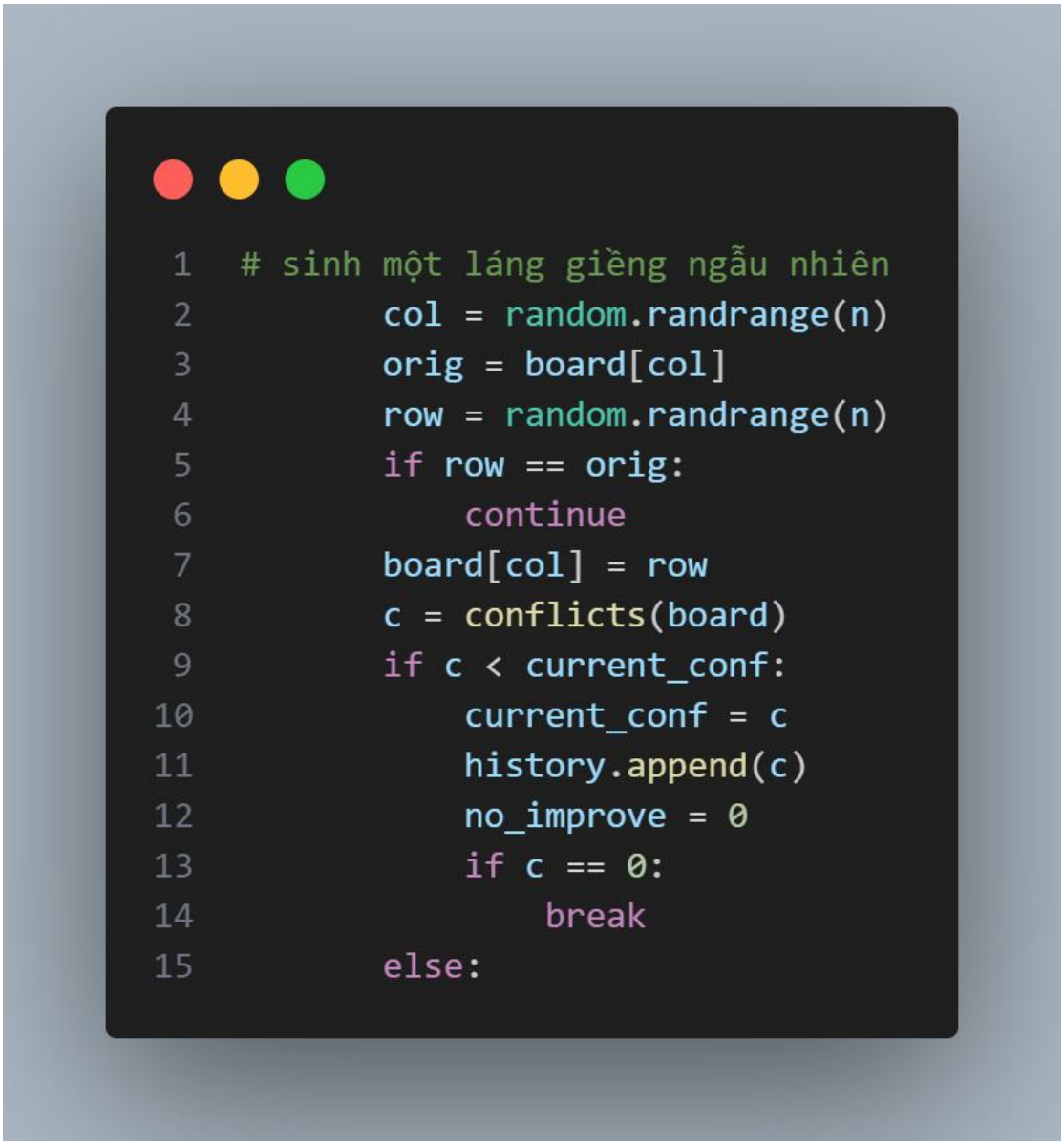
## TASK 3: Stochastic Hill Climbing 2 (First choice HC)

### 1. Mục tiêu

Cài đặt biến thể "First-choice hill climbing" (một dạng stochastic HC): thay vì liệt kê tất cả các láng giềng, thuật toán sinh từng láng giềng ngẫu nhiên một lần và chấp nhận ngay nếu nó cải thiện objective. Nếu không cải thiện nhiều lần liên tiếp (no_improve_limit), dừng lại. Phương pháp này tiết kiệm thời gian khi số láng giềng quá lớn.

### 2. Code

```
def first_choice_hill_climbing(start_board, max_iters=10000, no_improve_limit=1000):
    """First-choice hill climbing:
    Sinh ngẫu nhiên một láng giềng (chọn cột và hàng mới) và chấp nhận ngay nếu nó cải thiện.
    Dừng nếu không cải thiện trong một số lượng thử liên tiếp (no_improve_limit).
    """
    board = np.array(start_board).copy()
    n = len(board)
    history = [conflicts(board)]
    it = 0
    no_improve = 0
    current_conf = conflicts(board)
    while it < max_iters and no_improve < no_improve_limit:
        it += 1
```


1 # sinh một láng giềng ngẫu nhiên
2 col = random.randrange(n)
3 orig = board[col]
4 row = random.randrange(n)
5 if row == orig:
6 continue
7 board[col] = row
8 c = conflicts(board)
9 if c < current_conf:
10 current_conf = c
11 history.append(c)
12 no_improve = 0
13 if c == 0:
14 break
15 else:

```
# hoàn lại nếu không chấp nhận
            board[col] = orig
            no_improve += 1
    return board, history

```

```
# kiểm tra nhanh
if __name__ == '__main__':
    b = random_board(8)
    final, hist = first_choice_hill_climbing(b)
    print('Xung đột ban đầu:', conflicts(b), 'Xung đột cuối:', conflicts(final))
```

**3. Kết quả:**

```
Xung đột ban đầu: 7 Xung đột cuối: 3
```

**4. Mô tả:**

**- Mục tiêu của ô**

Cài đặt biến thể "First-choice hill climbing" (một dạng stochastic HC): thay vì liệt kê tất cả các láng giềng, thuật toán sinh từng láng giềng ngẫu nhiên một lần và chấp nhận ngay nếu nó cải thiện objective. Nếu không cải thiện nhiều lần liên tiếp (no_improve_limit), dừng lại. Phương pháp này tiết kiệm thời gian khi số láng giềng quá lớn.

**- Hàm và đầu vào/đầu ra**

Hàm: **first_choice_hill_climbing(start_board, max_iters=10000, no_improve_limit=1000)**

**- Input:**

- **start_board:** mảng vị trí hàng cho mỗi cột.
- **max_iters:** giới hạn tối đa số lần thử.
- **no_improve_limit:** số lần thử liên tiếp không thấy cải thiện để coi là đã tới cực trị cục bộ (dừng sớm).

**- Output:**

- **final_board:** trạng thái cuối cùng (numpy array).
- **history:** danh sách số xung đột theo thời gian.

**- Luồng xử lý (chi tiết)**

- Sao chép **start_board** sang **board**; lưu **n = len(board)**.
- Khởi **history = [conflicts(board)]**, **no_improve = 0**, **current_conf = conflicts(board)**.
- Vòng lặp **while it < max_iters and no_improve < no_improve_limit**:
  - Sinh một láng giềng ngẫu nhiên bằng cách:
    - Chọn ngẫu nhiên một cột **col = random.randrange(n)**.
    - Chọn một hàng mới **row = random.randrange(n)**; nếu **row == orig** (không thay đổi), bỏ qua (tiếp tục vòng).
    - Thực hiện **board[col] = row**, tính **c = conflicts(board)**.
  - Nếu **c < current_conf** (tức cải thiện):
    - Cập nhật **current_conf = c**, **history.append(c)**, **no_improve = 0**.
    - Nếu **c == 0**, dừng vì tìm nghiệm.
  - Nếu không cải thiện:
    - Khôi phục **board[col] = orig** và **no_improve += 1**.
- Trả về **board, history**.

**5. Tại sao dùng first-choice?**

- Khi mỗi trạng thái có nhiều láng giềng (ở đây n*(n-1) láng giềng nếu xét tất cả), liệt kê và đánh giá hết sẽ rất tốn. First-choice giảm chi phí bằng cách sinh cặp (col,row) ngẫu nhiên và chỉ đánh giá từng láng giềng một lần cho tới khi tìm được cải thiện.
- Nếu không có cải thiện trong nhiều lần thử liên tiếp (no_improve_limit), khả năng cao đã ở cực trị cục bộ -> dừng để tránh vòng lặp vô hạn.

**6. Các tham số và điều chỉnh**

- no_improve_limit: nhỏ → dừng sớm (có thể bỏ lỡ các cải tiến hiếm); lớn → cho phép tìm kiếm kỹ hơn. Thử nghiệm thường chọn giá trị tương đối so với max_iters.
- max_iters: dùng để bảo vệ chống infinite loop.

## TASK 4: Hill Climbing Search with Random Restarts

### 1. Mục tiêu

Random restarts (khởi động lại ngẫu nhiên) là kỹ thuật để giảm khả năng kẹt ở cực trị cục bộ. Ý tưởng: chạy thuật toán local search nhiều lần, mỗi lần bắt đầu từ một trạng thái ngẫu nhiên mới, và giữ nghiệm tốt nhất tìm được.

### 2. Code

```python
def random_restarts(algorithm_fn, n, restarts=100, **kwargs):
    results = []
    times = []
    for i in range(restarts):
        start = random_board(n)
        t0 = time.time()
        final, hist = algorithm_fn(start, **kwargs)
        t1 = time.time()
        results.append({'start_conf': conflicts(start), 'end_conf': conflicts(final), 'history': hist})
        times.append(t1 - t0)
    return results, times
```

```python
# Ví dụ kiểm tra nhanh với n=8 và 20 lần khởi động lại
if __name__ == '__main__':
    res, ts = random_restarts(steepest_ascent_hill_climbing, 8, restarts=20)
    print('Xung đột tối thiểu sau các lần chạy:', min(r['end_conf'] for r in res), 'Thời gian trung bình (s):', np.mean(ts))
```

### 3. Kết quả

```
Xung đột tối thiểu sau các lần chạy: 1 Thời gian trung bình (s): 0.0020501017570495605
```

### 4. Mô tả

**- Hàm và đầu vào/đầu ra**

- Hàm: **random_restarts(algorithm_fn, n, restarts=100, **kwargs)**
  - **algorithm_fn**: một hàm hill-climbing (ví dụ steepest_ascent_hill_climbing) nhận start_board là đầu vào và trả về (final_board, history).
  - **n**: kích thước bàn (số cột).
  - **restarts**: số lần chạy (ví dụ 100).
  - ****kwargs**: các tham số phụ chuyển tiếp cho algorithm_fn (ví dụ max_iters).
- **Output**:
  - **results**: danh sách dict cho mỗi lần chạy, chứa 'start_conf', 'end_conf', 'history'.
  - **times**: danh sách thời gian chạy (s) tương ứng với mỗi lần restart.

**- Luồng xử lý**

- Khởi **results = []**, **times = []**.
- Vòng lặp **for i in range(restarts)**:
  - Sinh **start = random_board(n)** (bàn khởi tạo ngẫu nhiên).
  - Ghi thời gian bắt đầu **t0 = time.time()**.

- Gọi **final, hist = algorithm_fn(start, **kwargs)** để chạy thuật toán từ khởi tạo đó.
- Ghi **t1 = time.time()** và tính thời gian **t1 - t0**.
- **Lưu kết quả**: append dict {'start_conf': conflicts(start), 'end_conf': conflicts(final), 'history': hist} vào results.
- Append thời gian vào times.

- Trả về **results, times**.

**5. Mục đích trong thí nghiệm**

Sử dụng random_restarts để:

- Tăng tỷ lệ tìm được nghiệm tối ưu so với chạy 1 lần.
- So sánh thuật toán nào có khả năng cải thiện nhiều nhờ restarts (ví dụ first-choice có thể cần ít restarts hơn so với steepest).
- Ở báo cáo, báo min(end_conf) (xung đột nhỏ nhất), mean(end_conf), thời gian trung bình, và số lần đạt optimal.

**TASK 5: Simulated Annealing**

**1. Mục tiêu**

Triển khai thuật toán Simulated Annealing (SA) cho bài toán n-Queens để cho phép thoát khỏi cực trị cục bộ bằng cách chấp nhận các bước xấu với xác suất tỉ lệ nghịch với độ xấu và tỉ lệ thuận với nhiệt độ T. Nhiệt độ giảm theo thời gian (annealing schedule) để dần chuyển từ khám phá sang khai thác.

**2. Code**

```python
def simulated_annealing(start_board, max_iters=10000, t0=1.0, alpha=0.995):
    board = np.array(start_board).copy()
    n = len(board)
    history = [conflicts(board)]
    current_conf = conflicts(board)
    T = t0
    for it in range(max_iters):
```

```python
# sinh một láng giềng ngẫu nhiên
        col = random.randrange(n)
        orig = board[col]
        row = random.randrange(n)
        if row == orig:
            continue
        board[col] = row
        c = conflicts(board)
        delta = c - current_conf
        if delta <= 0:
```

```
# chấp nhận nếu cải thiện
            current_conf = c
            history.append(c)
        else:
            # chấp nhận với xác suất exp(-delta/T)
            if T <= 0:
                accept = False
            else:
                accept = (random.random() < np.exp(-delta / T))
            if accept:
                current_conf = c
                history.append(c)
            else:
                board[col] = orig
```

```
# làm nguội
        T *= alpha
        if current_conf == 0:
            break
        if T < 1e-12:
            break
    return board, history
```

```python
# kiểm tra nhanh
if __name__ == '__main__':
    b = random_board(8)
    final, hist = simulated_annealing(b, max_iters=2000)
    print('Xung đột ban đầu:', conflicts(b), 'Xung đột cuối:', conflicts(final))
```

**3. Kết quả**

```
Xung đột ban đầu: 7 Xung đột cuối: 0
```

**4. Mô tả**

**- Hàm và đầu vào/đầu ra**

- Hàm: **simulated_annealing(start_board, max_iters=10000, t0=1.0, alpha=0.995)**
  - **start_board:** mảng khởi tạo (bảng).
  - **max_iters:** số vòng lặp tối đa.
  - **t0:** nhiệt độ ban đầu.
  - **alpha:** hệ số làm nguội (T <- T * alpha mỗi vòng). Giá trị gần 1 làm nguội chậm hơn.
- **Output:**
  - **final_board:** bảng sau khi dừng.
  - **history:** danh sách số xung đột theo thời gian (dùng để vẽ tiến trình).

**- Luồng xử lý**

- Sao chép **start_board** sang **board**. Khởi **history = [conflicts(board)], current_conf = conflicts(board), T = t0**.
- Vòng **for it in range(max_iters)**:
  - Sinh một láng giềng ngẫu nhiên bằng cách chọn cột **col = random.randrange(n)** và hàng ngẫu nhiên **row = random.randrange(n)**; nếu **row == orig**, bỏ qua.
  - Cập nhật **board[col] = row**, tính **c = conflicts(board)**.
  - Tính **delta = c - current_conf**.
    - Nếu **delta <= 0**: move cải thiện hoặc bằng; chấp nhận: current_conf = c và append c vào history.
    - Ngược lại **delta > 0** (move xấu): chấp nhận với xác suất exp(-delta / T) nếu T > 0. Nếu chấp nhận, cập nhật current_conf và history; nếu không, revert board[col] = orig.
  - **Làm nguội**: T *= alpha.
  - Nếu **current_conf == 0**, dừng sớm (giải).
  - Nếu **T < 1e-12**, dừng (nhiệt độ quá nhỏ).
- Trả về **board, history**.

**5. Giải thích các tham số và ảnh hưởng**

- **t0** (nhiệt độ ban đầu): càng lớn → càng dễ chấp nhận các bước xấu ban đầu, tăng tính khám phá.
- **alpha** (tỷ lệ giảm): 0.995 là lịch làm nguội chậm; alpha nhỏ hơn (vd. 0.9) làm nguội nhanh hơn, giảm lượng chấp nhận bước xấu.
- **max_iters:** cần đủ lớn để SA có thời gian khám phá; trong thực nghiệm, thường cần điều chỉnh t0/alpha để đạt kết quả tốt.

**TASK 6: Algorithm Behavior Analysis**

- **Mục tiêu** của phần so sánh là tóm tắt hiệu năng thực nghiệm: thời gian chạy trung bình, chất lượng nghiệm (số xung đột trung bình) và tỷ lệ chạy đạt nghiệm tối ưu (0 xung đột).
- **Code và kết quả:**

```python
algorithms = [
    ("Steepest", steepest_ascent_hill_climbing),
    ("StochasticAll", stochastic_hill_climbing_all_neighbors),
    ("FirstChoice", first_choice_hill_climbing),
    ("SimulatedAnnealing", simulated_annealing)
]
```

```python
def evaluate_algorithms(sizes=[4,8], runs=100, restarts=10):
    """Đánh giá các thuật toán theo thời gian chạy trung bình, số xung đột trung bình
    và tỷ lệ chạy đạt nghiệm tối ưu cho các kích thước bàn khác nhau.
    Trả về một DataFrame tóm tắt kết quả.
    """
    rows = []
    for size in sizes:
        for name, fn in algorithms:
            run_times = []
            end_confs = []
            success_count = 0
            for r in range(runs):
                start = random_board(size)
                t0 = time.time()
                final, hist = fn(start)
                t1 = time.time()
                run_times.append(t1 - t0)
                end_confs.append(conflicts(final))
                if conflicts(final) == 0:
                    success_count += 1
            rows.append({
                'Algorithm': name,
                'Board size': size,
                'Avg Run time': np.mean(run_times),
                'Avg conflicts': np.mean(end_confs),
                '% optimal': 100.0 * success_count / runs
            })
    return pd.DataFrame(rows)
```

```python
# ví dụ chạy nhanh (giảm số lần để demo)
if __name__ == '__main__':
    df = evaluate_algorithms(sizes=[4,8], runs=30)
    display(df)
```

| | Algorithm | Board size | Avg Run time | Avg conflicts | % optimal |
|---|---|---|---|---|---|
| 0 | Steepest | 4 | 0.000133 | 0.566667 | 46.666667 |
| 1 | StochasticAll | 4 | 0.000126 | 0.600000 | 46.666667 |
| 2 | FirstChoice | 4 | 0.002815 | 0.600000 | 50.000000 |
| 3 | SimulatedAnnealing | 4 | 0.000353 | 0.000000 | 100.000000 |
| 4 | Steepest | 8 | 0.001842 | 1.266667 | 10.000000 |
| 5 | StochasticAll | 8 | 0.002799 | 1.100000 | 20.000000 |
| 6 | FirstChoice | 8 | 0.008478 | 1.166667 | 6.666667 |
| 7 | SimulatedAnnealing | 8 | 0.006757 | 0.000000 | 100.000000 |

- Từ các thuật toán đã triển khai thường thu được những quan sát sau:

- **Steepest-ascent HC:** thường có cải thiện nhanh ban đầu (vì đánh giá toàn bộ không gian lân cận để chọn bước tốt nhất), nhưng dễ bị mắc kẹt tại cực trị cục bộ. Chi phí mỗi bước cao hơn (do xem xét tất cả các láng giềng), nên thời gian chạy trung bình có thể lớn hơn so với các biến thể mang tính ngẫu nhiên.
- **Stochastic HC 1** (chọn ngẫu nhiên từ các bước cải thiện): cho hiệu năng ổn định hơn so với steepest trong một số trường hợp, vì chọn ngẫu nhiên giữa các bước tốt giúp tránh một vài bẫy cục bộ nhỏ, nhưng vẫn có khả năng bị dừng ở cực trị cục bộ.
- **Stochastic HC 2 / First-choice**: thường có bước tiến nhanh hơn (ít phải đánh giá nhiều láng giềng), phù hợp với bài toán có số láng giềng lớn; tuy nhiên phụ thuộc mạnh vào may rủi của mẫu, nên phương pháp này có phương sai lớn hơn về kết quả cuối cùng.
- **Simulated Annealing**: chậm hơn từng bước so với first-choice nhưng có khả năng vượt khỏi cực trị cục bộ nhờ chấp nhận các bước xấu với xác suất giảm dần. Trong thực nghiệm, SA thường có tỷ lệ đạt nghiệm tối ưu cao hơn khi lịch làm lạnh và tham số được chỉnh hợp lý.

- **Kết luận thực tế**: không có thuật toán nào "tốt nhất" cho mọi kích thước; steepest tốt về chất lượng bước nhưng tốn thời gian, first-choice phù hợp khi cần chạy nhanh nhiều lần, và simulated annealing là lựa chọn

hợp lý khi muốn ưu tiên tìm nghiệm tối ưu hơn và chấp nhận chi phí tính toán cao hơn.

**ADVANCE TASK:**

**- Algorith Convergence: Hội tụ của thuật toán**

```
def plot_representative_runs():
    fig, ax = plt.subplots(figsize=(9,5))
    for name, fn in algorithms:
        start = random_board(8)
        final, hist = fn(start)
        ax.plot(hist, label=name)
    ax.set_xlabel('Số bước (Iteration)')
    ax.set_ylabel('Số xung đột (conflicts)')
    ax.set_title('Mô tả hội tụ: các run đại diện trên 8-Queens')
    ax.grid(True)
    ax.legend()
    plt.tight_layout()
    plt.show()

# Gọi hàm để hiển thị
if __name__ == '__main__':
    plot_representative_runs()
```

**Kết quả:**


**Nhận xét:**

**Steepest Ascent Hill Climbing (xanh dương)**

- Giảm xung đột rất nhanh chỉ sau khoảng 3 bước đã đạt 0.
- Thể hiện khả năng hội tụ nhanh nhất trong các thuật toán được so sánh.
- Tuy nhiên, do đặc tính "tham lam" (greedy), thuật toán này dễ mắc kẹt ở cực trị địa phương nếu không may mắn ở trạng thái ban đầu.

**Stochastic Hill Climbing (vàng cam)**

- Cũng giảm xung đột nhanh, hội tụ sau khoảng 5–6 bước.
- So với Steepest, kết quả tương đương nhưng ổn định hơn, vì có yếu tố ngẫu nhiên giúp tránh kẹt cục bộ.

**First Choice Hill Climbing (xanh lá)**

- Hiệu quả gần như Steepest, đạt nghiệm chỉ sau vài bước.
- Ưu điểm là nhanh và tiết kiệm tính toán hơn vì không cần đánh giá toàn bộ lân cận.

**Simulated Annealing (đỏ)**

- Giảm xung đột chậm hơn rõ rệt, dao động thất thường giữa các mức xung đột.
- Đặc trưng của thuật toán này là cho phép bước lùi tạm thời để thoát cực trị địa phương, nên đường biểu diễn có nhiều "lên xuống".
- Dù khởi đầu kém hơn, nhưng sau khoảng 90 bước cũng đạt được nghiệm (conflict = 0).
- Điều này cho thấy tính bền vững và khả năng tìm lời giải tốt dù tốn thời gian hơn.

**- Problem Size Scalability (Độ phóng to theo kích thước bài toán):**

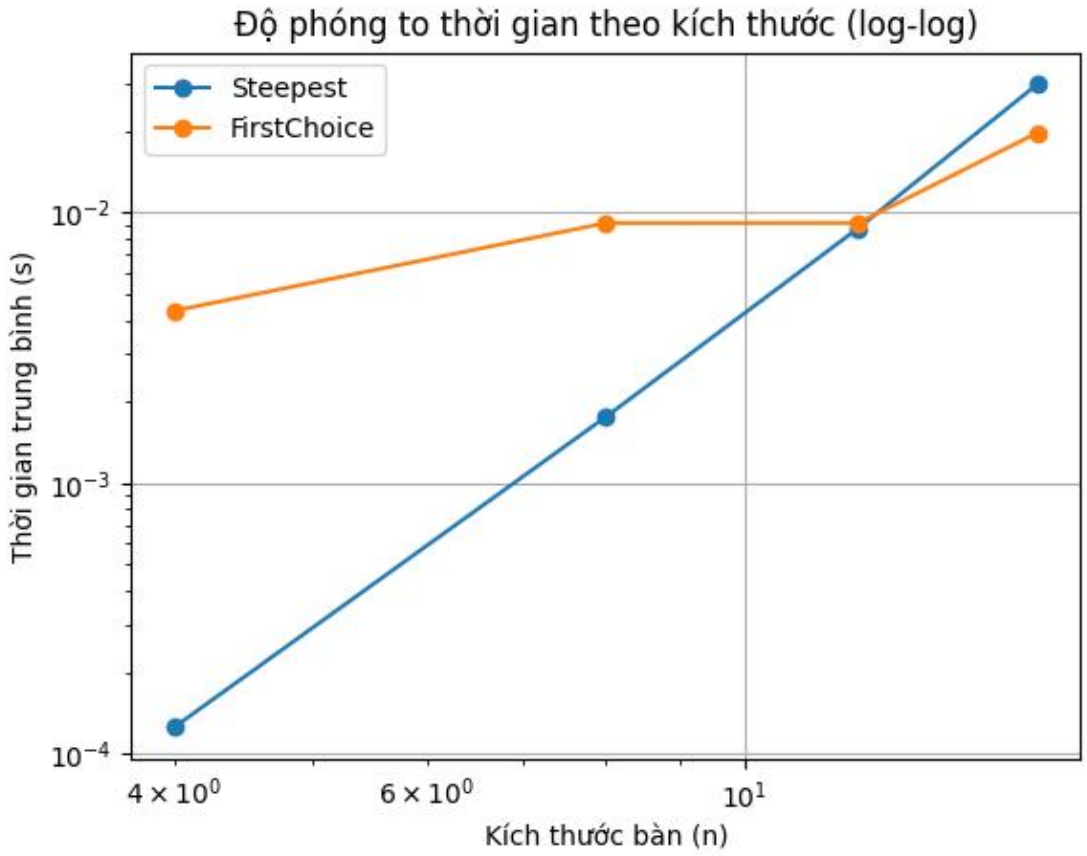
```
def measure_scaling(sizes=[4,8,12,16], runs=5, algorithms_to_test=None):
    """Trả về dict chứa kích thước và thời gian trung bình cho mỗi thuật toán.
    algorithms_to_test: list các tuple (label, function)
    """
    if algorithms_to_test is None:
        algorithms_to_test = [("Steepest", steepest_ascent_hill_climbing), ("FirstChoice",
first_choice_hill_climbing)]

    results = { 'size': list(sizes) }
    for name, _ in algorithms_to_test:
        results[name] = []
```

```
    for size in sizes:
        for name, fn in algorithms_to_test:
            times = []
            for r in range(runs):
                start = random_board(size)
                t0 = time.time()
                fn(start)
                t1 = time.time()
                times.append(t1 - t0)
            results[name].append(np.mean(times))
    return results
```

```
# Ví dụ chạy nhanh (chỉ dùng 4 kích thước để demo)
if __name__ == '__main__':
    res = measure_scaling(sizes=[4,8,12,16], runs=3)
    plt.loglog(res['size'], res['Steepest'], marker='o', label='Steepest')
    plt.loglog(res['size'], res['FirstChoice'], marker='o', label='FirstChoice')
    plt.xlabel('Kích thước bàn (n)')
    plt.ylabel('Thời gian trung bình (s)')
    plt.title('Độ phóng to thời gian theo kích thước (log-log)')
    plt.legend()
    plt.grid(True)
    plt.show()
```

**Kết quả:**




**Nhận xét:**

**Steepest Ascent (xanh dương)**

- Có thời gian tăng rất nhanh theo kích thước bàn (đường dốc lên rõ rệt).
- Khi nnn tăng, thời gian tăng gần như theo cấp số nhân.
- Điều này là do thuật toán phải đánh giá toàn bộ lân cận ở mỗi bước — chi phí tăng mạnh khi bàn cờ lớn.

**First Choice (cam)**

- Thời gian tăng chậm hơn, gần như ổn định ở các giá trị n=8n = 8n=8 đến n=16n = 16n=16.
- Chỉ khi nnn rất lớn (khoảng > 16), thời gian mới tăng đáng kể.
- Lý do: thuật toán chọn ngẫu nhiên và dừng sớm khi gặp nước đi tốt, nên giảm khối lượng tính toán đáng kể.

**-Exploring other Local Moves Operators**

Triển khai các Local Moves Operators khác:

```
def single_step_move(board):
    """Di chuyển một quân một ô lên hoặc xuống (wrap-around).
    - board: mảng 1D (numpy array hoặc list) biểu diễn vị trí hàng của quân theo cột.
    Trả về bảng mới (copy) sau di chuyển.
    """
    b = board.copy()
    n = len(b)
    col = random.randrange(n)
    direction = random.choice([-1, 1])
    b[col] = (int(b[col]) + direction) % n
    return b
```

```python
def column_swap_move(board):
    """Hoán đổi vị trí của hai cột (đổi chỗ 2 quân hậu giữa 2 cột).
    Trả về bảng mới.
    """
    b = board.copy()
    n = len(b)
    c1, c2 = random.sample(range(n), 2)
    b[c1], b[c2] = b[c2], b[c1]
    return b
```

```python
def dual_queen_move(board):
    """Chọn hai cột và đặt lại vị trí hàng của cả hai quân một cách ngẫu nhiên."""
    b = board.copy()
    n = len(b)
    c1, c2 = random.sample(range(n), 2)
    b[c1] = random.randrange(n)
    b[c2] = random.randrange(n)
    return b
```

```python
# Hàm phụ: tính số xung đột mà một quân ở cột `col` đang tạo ra
def local_conflicts(board, col):
    """Trả về số lượng quân khác đang xung đột với quân ở cột `col`.
    Dùng để xác định cột nào có vấn đề và nên tập trung.
    """
    n = len(board)
    cnt = 0
    for j in range(n):
        if j == col:
            continue
        # cùng hàng
        if board[j] == board[col]:
            cnt += 1
        # cùng đường chéo
        if abs(board[j] - board[col]) == abs(j - col):
            cnt += 1
    return cnt
```

```python
class AdaptiveMove:
    """Move adaptive: tập trung vào cột có nhiều xung đột.
    - move(board): trả về board mới sau một move áp dụng chiến lược thích ứng.
    """
    def __init__(self):
        self.history = []
```

```python
    def move(self, board):
        n = len(board)
        # tìm các cột có xung đột > 0
        conflict_cols = [col for col in range(n) if local_conflicts(board, col) > 0]
        if conflict_cols:
            col = random.choice(conflict_cols)
            new_board = board.copy()
            # thử đặt ngẫu nhiên một hàng mới cho cột chọn
            new_board[col] = random.randrange(n)
            return new_board
        # nếu không có cột nào xung đột, fallback về single step
        return single_step_move(board)
```

```python
# Sử dụng first-choice hill climbing với operator tuỳ chọn
def first_choice_with_operator(start_board, move_operator, max_iters=10000, no_improve_limit=1000):
    """Phiên bản first-choice hill climbing cho phép truyền move_operator(board)."""
    board = np.array(start_board).copy()
    n = len(board)
    history = [conflicts(board)]
    it = 0
    no_improve = 0
    current_conf = conflicts(board)
    while it < max_iters and no_improve < no_improve_limit:
        it += 1
        candidate = move_operator(board)
        c = conflicts(candidate)
        if c < current_conf:
            board = np.array(candidate)
```

```
            current_conf = c
            history.append(c)
            no_improve = 0
            if c == 0:
                break
        else:
            no_improve += 1
    return board, history
```

```
# Ví dụ nhanh
if __name__ == '__main__':
    b = random_board(8)
    final, hist = first_choice_with_operator(b, column_swap_move)
    print('Xung đột ban đầu:', conflicts(b), 'Xung đột cuối:', conflicts(final))
```

Kết quả:

```
Xung đột ban đầu: 8 Xung đột cuối: 5
```

Từ việc thử nghiệm các move operators khác nhau (single-step, column-swap, dual-queen, adaptive) ta rút ra:

- **Column-swap** hữu ích khi nhiều cột có cấu trúc xung đột lẫn nhau; việc hoán đổi đôi khi nhanh chóng khôi phục trạng thái không xung đột.
- **Single-step** phù hợp khi cần tinh chỉnh nhẹ vị trí của các quân, ít phá vỡ cấu trúc tổng thể.
- **Dual-queen move** và các move thay đổi nhiều biến cùng lúc có thể giúp thoát khỏi bẫy cục bộ nhưng cũng khiến quá trình tìm kiếm mất ổn định hơn.
- **Adaptive move** (tập trung vào cột có xung đột nhiều) thường cho hiệu quả thực nghiệm tốt vì nó kết hợp khám phá có định hướng và randomization.

**- Bài học chung:**

- Không có move operator tốt nhất cho mọi trường hợp; việc kết hợp vài operator và lựa chọn ngẫu nhiên theo trạng thái hiện tại thường cho kết quả khả quan.
- Việc đo lường và so sánh bằng thực nghiệm là thiết yếu — các giả thuyết lý thuyết chỉ dẫn hướng, nhưng thực nghiệm mới cho biết tham số tối ưu cho từng kích thước và mục tiêu.