A picture containing application

Description automatically generated**TRƯỜNG ĐẠI HỌC SÀI GÒN**

**KHOA CÔNG NGHỆ THÔNG TIN**

Icon

Description automatically generated with low confidence

**BÁO CÁO**

**HỌC PHẦN: TRÍ TUỆ NHÂN TẠO NÂNG CAO**

**LAB 03**

**Sinh viên thực hiện**

Nguyễn Văn Anh Tú – 3122410445

Trương Hữu Nghĩa - 3122410263

Đỗ Khôi Nguyên - 3122410266

Lai Tấn Tài - 3122410366

**TP. HCM THÁNG 09/2025**

1. The N-Queen Prolem

Task 1: Steepest-ascend Hill Climbing Search

**Ý tưởng:** Ở mỗi bước, thử tất cả các nước đi có thể (di chuyển 1 hậu sang hàng khác) và chọn nước đi nào giảm nhiều nhất số lượng xung đột (conflicts).

**Đặc điểm hội tụ:** Ban đầu cải thiện nhanh, sau đó dừng sớm khi đạt **local optimum** (điểm cực trị cục bộ).

**Ưu điểm:** Tìm nghiệm tốt nếu khởi tạo thuận lợi.

**Nhược điểm:** Dễ bị kẹt ở local optimum.

Code:

def steepest\_ascent\_hill\_climbing(n, max\_iters=1000):

    board = random\_board(n)

    current\_conflicts = conflicts(board)

    steps = [current\_conflicts]  # để lưu lại quá trình hội tụ

    for \_ in range(max\_iters):

        best\_board = board.copy()

        best\_conflicts = current\_conflicts

        # thử di chuyển từng hậu sang các hàng khác

        for col in range(n):

            for row in range(n):

                if board[col] == row:

                    continue

                new\_board = board.copy()

                new\_board[col] = row

                new\_conflicts = conflicts(new\_board)

                if new\_conflicts < best\_conflicts:

                    best\_board = new\_board

                    best\_conflicts = new\_conflicts

        # nếu không có cải thiện, dừng lại

        if best\_conflicts >= current\_conflicts:

            break

        # cập nhật board tốt nhất

        board = best\_board

        current\_conflicts = best\_conflicts

        steps.append(current\_conflicts)

        # nếu đạt nghiệm (0 conflict) thì thoát

        if current\_conflicts == 0:

            break

    return board, current\_conflicts

Task 2: Stochastic Hill Climbing 1

**Ý tưởng**: Thay vì chọn nước đi tốt nhất, thuật toán chọn ngẫu nhiên một nước đi tốt hơn hiện tại (có ít xung đột hơn).

**Đặc điểm hội tụ**: Tăng tính ngẫu nhiên → giúp thoát local optimum đôi khi. Tuy nhiên hội tụ chậm hơn Steepest-Ascent

**Ưu điểm**: Có thể tìm nghiệm tốt hơn nếu may mắn.

**Nhược điểm:** Không ổn định, kết quả thay đổi theo seed.

Code:

def stochastic\_hill\_climbing\_1(n, max\_iters=1000):

    board = random\_board(n)

    current\_conflicts = conflicts(board)

    steps = [current\_conflicts]

    for \_ in range(max\_iters):

        improving\_neighbors = []

        for col in range(n):

            for row in range(n):

                if board[col] == row:

                    continue

                new\_board = board.copy()

                new\_board[col] = row

                new\_conflicts = conflicts(new\_board)

                if new\_conflicts < current\_conflicts:

                    improving\_neighbors.append((new\_board, new\_conflicts))

        if not improving\_neighbors:

            break

        board, current\_conflicts = random.choice(improving\_neighbors)

        steps.append(current\_conflicts)

        if current\_conflicts == 0:

            break

    return board, current\_conflicts

Task 3: Stochastic Hill Climbing 2

**Ý tưởng**: Chạy Steepest-Ascent nhiều lần từ các board khởi tạo ngẫu nhiên khác nhau. Nếu bị kẹt, khởi động lại.

**Đặc điểm hội tụ**: Dạng “brute force thông minh” — thử nhiều hướng, tăng xác suất tìm ra nghiệm 0 conflict.

**Ưu điểm**: Hầu như luôn tìm được nghiệm đúng.

**Nhược điểm**: Mất thời gian nếu số restart nhiều.

Code:

def stochastic\_hill\_climbing\_2(n, max\_iters=10000, no\_improve\_limit=1000):

    board = random\_board(n)

    current\_conflicts = conflicts(board)

    steps = [current\_conflicts]

    no\_improve\_count = 0

    for \_ in range(max\_iters):

        # Sinh 1 neighbor ngẫu nhiên

        new\_board = board.copy()

        col = random.randint(0, n - 1)

        new\_row = random.randint(0, n - 1)

        while new\_row == new\_board[col]:

            new\_row = random.randint(0, n - 1)

        new\_board[col] = new\_row

        new\_conflicts = conflicts(new\_board)

        # Nếu neighbor tốt hơn → chấp nhận

        if new\_conflicts < current\_conflicts:

            board = new\_board

            current\_conflicts = new\_conflicts

            steps.append(current\_conflicts)

            no\_improve\_count = 0

        else:

            no\_improve\_count += 1

        # Dừng nếu không cải thiện sau nhiều lần thử

        if no\_improve\_count > no\_improve\_limit or current\_conflicts == 0:

            break

    return board, current\_conflicts

Task 4: Hill Climbing Search with Random Restarts

Như ta được biết thì các nhược điểm của Hill Climbing thông thường là có thể bị kẹt ở local optimum (tốt hơn xung quanh nhưng chưa phải nghiệm toàn cục). Cũng có thể bị kẹt ở plateau (các trạng thái có giá trị giống nhau, không có tiến bộ).

Hill Climbing Search with Random Restarts sinh ra để khắc phục việc mắc kẹt, ta chạy nhiều lần Hill Climbing từ các trạng thái khởi tạo ngẫu nhiên khác nhau.

Code:

def get\_neighbours(board):

    """Sinh tất cả láng giềng bằng cách di chuyển 1 quân hậu trong cột khác hàng"""

    n = len(board)

    neighbors = []

    for col in range(n):

        for row in range(n):

            if board[col] != row:  # chỉ thay đổi nếu khác hàng

                new\_board = board.copy()

                new\_board[col] = row

                neighbors.append(new\_board)

    return neighbors

Ý tưởng:

Sinh ra tất cả trạng thái lân cận (neighbor) bằng cách di chuyển một quân hậu sang một hàng khác trong cùng cột. Vì trong biểu diễn N-Queens, board[i] = j nghĩa là quân hậu ở cột i hàng j.

Tính chất:

* Có đúng neighbors.
* Mỗi neighbor khác board hiện tại ở đúng 1 cột.
* Đây là “không gian lân cận ” mà Hill Climbing sẽ tìm kiếm.

Code:

def random\_restart\_hill\_climbing(n, max\_restarts=100, max\_steps=1000, show=False):

    best\_board = None

    best\_conf = float("inf")

    for restart in range(max\_restarts):

        board = random\_board(n)

        current\_board = board.copy()

        current\_conf = conflicts(current\_board)

        for step in range(max\_steps):

            if current\_conf == 0:

                print(f"Found solution at restart {restart}, step {step}")

                if show: show\_board(current\_board)

                return current\_board

            neighbors = get\_neighbours(current\_board)

            neighbor\_conflicts = [conflicts(nb) for nb in neighbors]

            min\_conf = min(neighbor\_conflicts)

            if min\_conf >= current\_conf:

                break  # mắc kẹt → restart mới

            best\_neighbor = neighbors[neighbor\_conflicts.index(min\_conf)]

            current\_board = best\_neighbor

            current\_conf = min\_conf

        # lưu nghiệm tốt nhất trong trường hợp không tìm được nghiệm hoàn hảo

        if current\_conf < best\_conf:

            best\_conf = current\_conf

            best\_board = current\_board

    print(f"No perfect solution found after {max\_restarts} restarts. Best conflicts = {best\_conf}")

    if show and best\_board is not None:

        show\_board(best\_board)

    return best\_board

Như đã biết thì với Hill Climbing Random Restart sẽ có cách hoạt động sau:

* Nếu bị kẹt do conflict => tạo board ngẫu nhiên mới => thử lại.
* Mỗi lần thử có cơ hội rơi gần một nghiệm toàn cục => tìm được lời giải sau vài chục lần.

Cấu trúc chính của code:

1. Sinh trạng thái khởi tạo ngẫu nhiên (random\_board)
2. Leo đồi bằng cách chọn neighbor tốt nhất (ít xung đột nhất).
3. Nếu bị kẹt (không có neighbor tốt hơn) => restart.
4. Lặp lại đến khi tìm được nghiệm hoặc hết số lần restart.

Giải thích chi tiết code:

1. Khởi tạo giá trị ban đầu

Lưu nghiệm tốt nhất, ít xung đột nhất tìm được qua tất cả restart. Nếu không có lời giải hoàn hảo, ta vẫn có thể tả về board “gần đúng nhất”

    best\_board = None

    best\_conf = float("inf")

1. Vòng lặp restart

for restart in range(max\_restarts):

        board = random\_board(n)

        current\_board = board.copy()

        current\_conf = conflicts(current\_board)

Mỗi lần restart:

* Tạo board ngẫu nhiên mới.
* Tính số xung đột ban đầu.
* Bắt đầu leo đồi từ đây.

1. Leo đồi

for step in range(max\_steps):

            if current\_conf == 0:

                print(f"Found solution at restart {restart}, step {step}")

                if show: show\_board(current\_board)

                return current\_board

Nếu đạt trạng thái hoàn hảo thì dừng và trả về

1. Sinh và đánh giá neighbors

neighbors = get\_neighbours(current\_board)

            neighbor\_conflicts = [conflicts(nb) for nb in neighbors]

1. Tìm neighbor tốt nhất

if min\_conf >= current\_conf:

                break  # mắc kẹt → restart mới

            best\_neighbor = neighbors[neighbor\_conflicts.index(min\_conf)]

            current\_board = best\_neighbor

            current\_conf = min\_conf

Sinh tất cả láng giềng, tính số xung đột cho từng neighbor

1. Lưu lại nghiệm tốt nhất và kết thúc

if current\_conf < best\_conf:

            best\_conf = current\_conf

            best\_board = current\_board

    print(f"No perfect solution found after {max\_restarts} restarts. Best conflicts = {best\_conf}")

    if show and best\_board is not None:

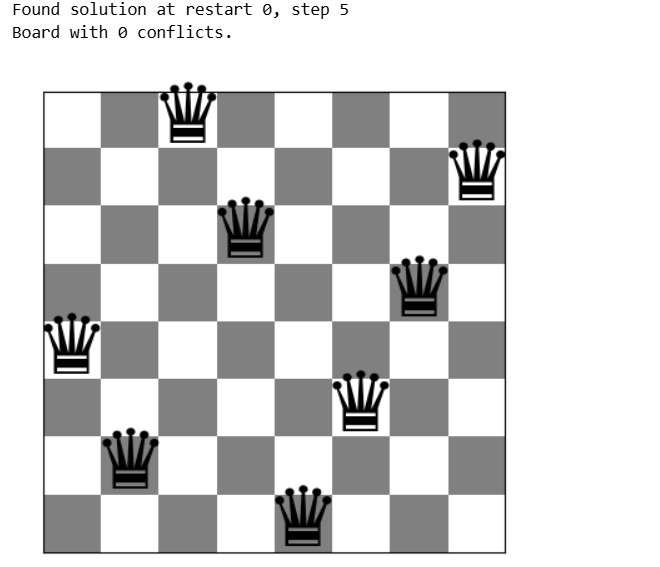
        show\_board(best\_board)

    return best\_board

 Nếu **không có neighbor nào tốt hơn** (tức local optimum hoặc plateau) → dừng vòng lặp và restart.

 Nếu có, **chuyển sang neighbor tốt nhất** (leo lên đỉnh cao hơn).  
→ Đây chính là **Hill Climbing “steepest-ascent”** (leo đồi dốc nhất).

**Kết quả code:**

****

Task 5: Simulated Annealing

Code:

def get\_neighbours(board):

    """Sinh tất cả láng giềng bằng cách di chuyển 1 quân hậu trong cột khác hàng"""

    n = len(board)

    neighbors = []

    for col in range(n):

        for row in range(n):

            if board[col] != row:  # chỉ thay đổi nếu khác hàng

                new\_board = board.copy()

                new\_board[col] = row

                neighbors.append(new\_board)

    return neighbors

def neighbour(board):

    """Chọn ngẫu nhiên một láng giềng từ danh sách tất cả các láng giềng"""

    neighbors = get\_neighbours(board)

    return random.choice(neighbors)

def tempestimation(board, E, m=1000):

    values = []

    for \_ in range(m):

        nb = neighbour(board)

        values.append(E(nb))

    mean\_val = np.mean(values)

    variance = np.mean([(x - mean\_val) \*\* 2 for x in values])

    std\_dev = math.sqrt(variance)

    return std\_dev if std\_dev > 0 else 1.0

def simulated\_annealing(n, kmax=10000, alpha=0.95, show=False):

    # trạng thái ban đầu

    s = random\_board(n)

    e = conflicts(s)

    emax = 0   # nghiệm lý tưởng là không xung đột

    k = 0

    # nhiệt độ ban đầu

    T = tempestimation(s, conflicts)

    while k < kmax and e > emax:

        sn = neighbour(s)

        en = conflicts(sn)

        # xác suất chấp nhận

        if en < e:

            accept = True

        else:

            prob = math.exp((e - en) / T)

            accept = (random.random() < prob)

        if accept:

            s, e = sn, en

        # giảm nhiệt độ

        T = alpha \* T

        k += 1

    if show:

        show\_board(s)

    print(f"Finished after {k} steps, conflicts = {e}")

    return s

Giống hill climbing, thuật toán cũng cần hàm get\_neighbor()

SA **không duyệt toàn bộ** neighbor mà **chọn ngẫu nhiên một trạng thái lân cận** mỗi lần.  
Điều này giúp **tăng tính ngẫu nhiên và tiết kiệm thời gian**, phù hợp với ý tưởng "thử nghiệm" trong SA.

Hàm tempestimation ước lượng nhiệt độ ban đầu T0​ bằng **độ lệch chuẩn của năng lượng (conflicts)** giữa các trạng thái lân cận ngẫu nhiên.

Ý nghĩa:

* Nếu biến thiên năng lượng cao → cần nhiệt độ lớn để “cho phép di chuyển tự do”.
* Nếu biến thiên nhỏ → hệ đã ổn định hơn → nhiệt độ ban đầu có thể nhỏ hơn.

Thuật toán chính:

Khởi tạo

* Trạng thái ban đầu ngẫu nhiên.
* Tính “năng lượng” hiện tại (số xung đột).
* Đặt nhiệt độ ban đầu theo tempestimation

Tạo một trạng thái hàng xóm mới.

Tính năng lượng tương ứng.

Nếu trạng thái mới tốt hơn thì luôn chấp nhận

Nếu tệ hơn, chấp nhận với xác suất:

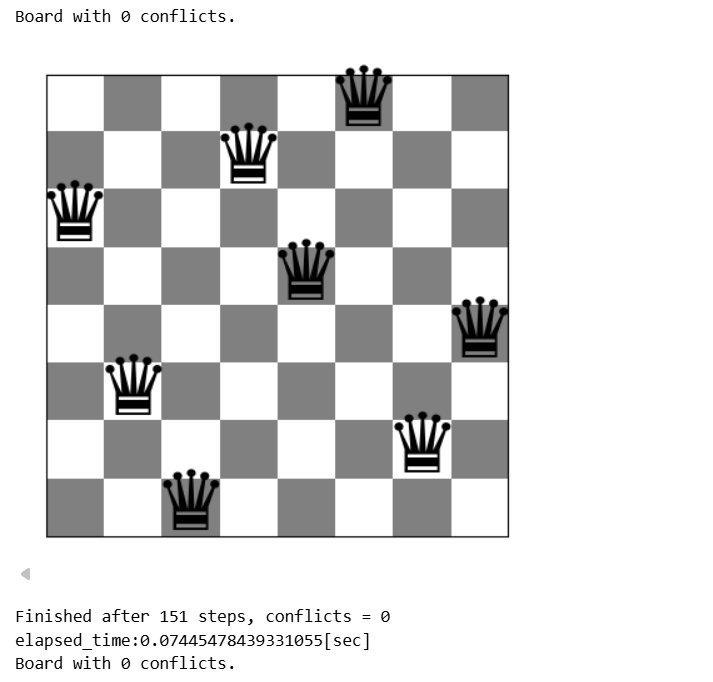
Khi T lớn => P gần 1 => dễ bước lùi

Khi T nhỏ => P gần 0 => hành vi giống HC

Nhiệt độ giảm dần theo:

Với alpha thuộc (0,1) ví dụ 0.95 => giảm 5% mỗi bước. Đây gọi là lịch làm nguội hình học.

Kết quả code:



Task 6: Compare Performance

Code:

def benchmark(algorithm, n, runs=100):

    times = []

    conflicts\_list = []

    successes = 0

    for \_ in range(runs):

        start = time.time()

        board, conf = algorithm(n)

        end = time.time()

        times.append(end - start)

        conflicts\_list.append(conf)

        if conf == 0:

            successes += 1

    avg\_time = np.mean(times)

    avg\_conflicts = np.mean(conflicts\_list)

    success\_rate = (successes / runs) \* 100

    return avg\_time, avg\_conflicts, success\_rate

algorithms = {

    "Steepest asc. HC": steepest\_ascent\_hill\_climbing,

    "Stochastic HC 1": stochastic\_hill\_climbing\_2,

    "Stochastic HC 2": stochastic\_hill\_climbing\_2,

    "Simulated Annealing": simulated\_annealing\_run

}

for n in [4, 8]:

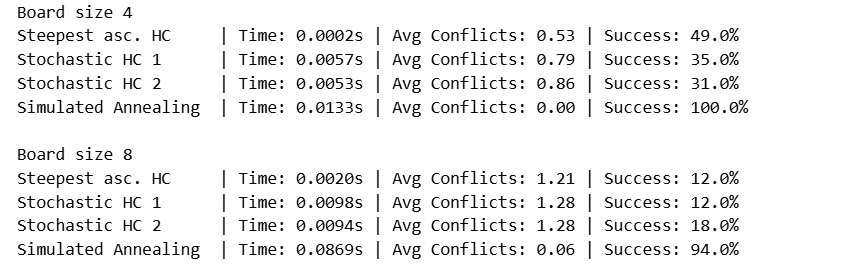
    print(f"\nBoard size {n}")

    for name, algo in algorithms.items():

        avg\_time, avg\_conflicts, success\_rate = benchmark(algo, n, runs=100)

        print(f"{name:20s} | Time: {avg\_time:.4f}s | Avg Conflicts: {avg\_conflicts:.2f} | Success: {success\_rate:.1f}%")

Kết quả:



Đánh giá

|  |  |  |
| --- | --- | --- |
| **Đặc điểm** | **Hill Climbing** | **Simulated Annealing** |
| **Khả năng khám phá không gian nghiệm** | Thấp – dễ mắc kẹt tại cực trị cục bộ | Cao – có thể thoát khỏi cực trị cục bộ nhờ chấp nhận bước “xấu” theo xác suất |
| **Tốc độ hội tụ** | Nhanh ban đầu nhưng nhanh chóng bị chững lại | Chậm hơn nhưng ổn định, cải thiện dần theo thời gian |
| **Khả năng mở rộng khi kích thước bài toán tăng (scalability)** | Giảm mạnh khi ( n ) lớn | Ổn định, ít bị ảnh hưởng bởi kích thước |
| **Chất lượng nghiệm cuối cùng** | Thường dừng ở nghiệm cục bộ, chưa tối ưu | Gần như luôn đạt nghiệm tối ưu (conflicts ≈ 0) |