```
# @title
import copy
from heapq import heappush, heappop

rows = [1, 0, -1, 0]
cols = [0, -1, 0, 1]

# Tạo một lớp hàng đợi ưu tiên (priority queue)
class priorityQueue:
    def __init__(self):
        self.heap = []

    # Chèn một khóa mới vào hàng đợi
    def push(self, key):
        heappush(self.heap, key)

    # Hàm để lấy phần tử nhỏ nhất từ Hàng đợi Ưu tiên
    def pop(self):
        return heappop(self.heap)

    # Hàm để kiểm tra xem Hàng đợi có trống hay không
    def empty(self):
        return not self.heap

# Cấu trúc của node (nút)
class nodes:
    def __init__(self, parent, mats, empty_tile_posi, costs, levels):
        # Lưu trữ node cha của node hiện tại, giúp truy vết đường đi khi tìm thấy nghiệm
        self.parent = parent

        # Lưu trữ ma trận trạng thái hiện tại
        self.mats = mats

        # Lưu trữ vị trí của ô trống trong ma trận
        self.empty_tile_posi = empty_tile_posi

        # Lưu số ô đặt sai vị trí (heuristic cost)
        self.costs = costs

        # Lưu số bước di chuyển cho đến nay (depth/level)
        self.levels = levels

    # Hàm này được sử dụng để so sánh các node trong hàng đợi ưu tiên
    def __lt__(self, nxt):
        return (self.costs + self.levels) < (nxt.costs + nxt.levels)

# Phương thức tính số ô đặt sai vị trí (số ô không phải ô trống không ở vị trí cuối cùng)
def calculateCosts(mats, final) -> int:
    count = 0
    for i in range(n):
        for j in range(n):
            if (mats[i][j] and (mats[i][j] != final[i][j])):
                count += 1
    return count

# Tạo node mới
def newNodes(mats, empty_tile_posi, new_empty_tile_posi, levels, parent, final) -> nodes:
    # Sao chép dữ liệu từ ma trận cha sang ma trận hiện tại
    new_mats = copy.deepcopy(mats)

    # Di chuyển ô bằng 1 vị trí (hoán đổi ô trống với ô lân cận)
    x1 = empty_tile_posi[0]
    y1 = empty_tile_posi[1]
    x2 = new_empty_tile_posi[0]
    y2 = new_empty_tile_posi[1]
    new_mats[x1][y1], new_mats[x2][y2] = new_mats[x2][y2], new_mats[x1][y1]

    # Thiết lập số ô đặt sai vị trí
    costs = calculateCosts(new_mats, final)

    new_nodes = nodes(parent, new_mats, new_empty_tile_posi, costs, levels)
    return new_nodes

# Hàm tìm vị trí của số 0 trong ma trận
def find_empty_position(mat, n):
```

```
    for i in range(n):
        for j in range(n):
            if mat[i][j] == 0:
                return [i, j]
    return None

# Hàm in ma trận N x N
def printMatrix(mats):
    for i in range(n):
        for j in range(n):
            print("%d " % (mats[i][j]), end = " ")
        print()

# Hàm kiểm tra (x, y) có phải là tọa độ ma trận hợp lệ hay không
def isSafe(x, y):
    return x >= 0 and x < n and y >= 0 and y < n

# In đường đi từ node gốc đến node đích
def printPath(root):
    if root == None:
        return
    printPath(root.parent)
    printMatrix(root.mats)
    print()

# Phương thức giải bài toán N*N - 1 puzzle sử dụng kỹ thuật Nhánh và Cận
# empty_tile_posi là vị trí ô trống ban đầu
def solve(initial, empty_tile_posi, final):
    # Tạo hàng đợi ưu tiên để lưu trữ các node đang xét của cây tìm kiếm
    pq = priorityQueue()

    # Tạo node gốc
    costs = calculateCosts(initial, final)
    root = nodes(None, initial, empty_tile_posi, costs, 0)

    # Thêm node gốc vào danh sách các node đang xét
    pq.push(root)

    # Tìm kiếm node với chi phí nhỏ nhất, thêm các node con vào danh sách đang xét
    # và cuối cùng xóa nó khỏi danh sách
    while not pq.empty():
        # Tìm node với chi phí ước tính nhỏ nhất và xóa khỏi danh sách
        minimum = pq.pop()

        # Nếu node này là nghiệm (tất cả ô đều đúng vị trí)
        if minimum.costs == 0:
            # In đường đi từ gốc đến đích
            print("Đường đi đến nghiệm:")
            printPath(minimum)
            return

        # Tạo tất cả các node con khả thi
        for i in range(4):  # 4 hướng di chuyển
            new_tile_posi = [
                minimum.empty_tile_posi[0] + rows[i],
                minimum.empty_tile_posi[1] + cols[i],
            ]

            if isSafe(new_tile_posi[0], new_tile_posi[1]):
                # Tạo một node con
                child = newNodes(minimum.mats,
                                 minimum.empty_tile_posi,
                                 new_tile_posi,
                                 minimum.levels + 1,
                                 minimum, final)

                # Thêm node con vào danh sách các node đang xét
                pq.push(child)

# Code chính

# --- SỬA PHẦN CODE CHÍNH ---

if __name__ == "__main__":
    # 1. Nhập kích thước N
    print("Nhập kích thước n của bàn cờ (ví dụ 3, 4, 5): ")
    n = int(input()) # Biến n này sẽ cập nhật cho toàn bộ các hàm ở trên sử dụng
    if n == 3 :
```

```
      input_puzzel = [[1, 2, 3],
           [5, 6, 0],
           [7, 8, 4]]

      output_puzzel = [[1, 2, 3],
         [5, 8, 6],
         [0, 7, 4]]
    if n == 4:
      input_puzzel = [
      [1, 2, 3, 4],
      [5, 6, 7, 8],
      [9, 10, 11, 12],
      [13, 0, 14, 15]
      ]

      output_puzzel = [
      [1, 2, 3, 4],
      [5, 6, 7, 8],
      [9, 10, 11, 12],
      [13, 14, 15, 0] # Số 0 về cuối
      ]

    if n == 5:
        input_puzzel = [
        [1, 2, 3, 4, 5],
        [6, 7, 8, 9, 10],
        [11, 12, 13, 14, 15],
        [16, 17, 18, 19, 20],
        [21, 22, 0, 23, 24]
      ]

        output_puzzel = [
            [1, 2, 3, 4, 5],
            [6, 7, 8, 9, 10],
            [11, 12, 13, 14, 15],
            [16, 17, 18, 19, 20],
            [21, 22, 23, 24, 0]
          ]
    # 4. Tự động tìm vị trí ô trống (số 0)
    ViTriOBanDau = find_empty_position(input_puzzel, n)

    if ViTriOBanDau is None:
        print("Lỗi: Không tìm thấy số 0 trong ma trận bắt đầu!")
    else:
        print("\nĐang giải bài toán...")\

        solve(input_puzzel, ViTriOBanDau, output_puzzel)
```

```
Nhập kích thước n của bàn cờ (ví dụ 3, 4, 5):
3

Đang giải bài toán...
Đường đi đến nghiệm:
1  2  3
5  6  0
7  8  4

1  2  3
5  0  6
7  8  4

1  2  3
5  8  6
7  0  4

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

```
from collections import deque
import builtins # Import the builtins module to access original functions

class DoThi:
    def __init__(self, danh_sach_ke, H_heuristic):
        """
        Khởi tạo đồ thị với danh sách kề.
        danh_sach_ke: Một dictionary chứa các đỉnh và trọng số đến các đỉnh hàng xóm.
        H_heuristic: Một dictionary chứa các giá trị heuristic cho mỗi nút
```

```
        H_heuristic: Một dictionary chứa các giá trị heuristic cho mỗi nút.
        """
        self.danh_sach_ke = danh_sach_ke
        self.H = H_heuristic # Store the heuristic dictionary as an instance attribute

    def lay_hang_xom(self, v):
        """Lấy danh sách các đỉnh kề (hàng xóm) của đỉnh v."""
        return self.danh_sach_ke[v]

    def ham_heuristic(self, n):
        """
        Hàm Heuristic (H): Ước lượng khoảng cách từ nút hiện tại đến đích.
        """
        return self.H.get(n, 1) # Mặc định trả về 1 nếu không tìm thấy nút

    def thuat_toan_a_star(self, diem_bat_dau, diem_ket_thuc):
        # --- KHỞI TẠO ---

        # tap_mo (Open List): Các nút đã được nhìn thấy nhưng chưa được kiểm tra kỹ (hàng xóm).
        tap_mo = set([diem_bat_dau])

        # tap_dong (Closed List): Các nút đã được kiểm tra hoàn tất.
        tap_dong = set([])

        # g_score: Lưu chi phí thực tế từ điểm bắt đầu đến nút hiện tại.
        # Giá trị mặc định là vô cùng, riêng điểm bắt đầu là 0.
        g_score = {}
        g_score[diem_bat_dau] = 0

        # cha (Parents): Lưu vết đường đi (nút này đến từ nút nào) để truy vết lại sau khi tìm thấy đích.
        cha = {}
        cha[diem_bat_dau] = diem_bat_dau

        # --- BẮT ĐẦU VÒNG LẶP ---
        while len(tap_mo) > 0:
            n = None

            # 1. Tìm nút trong tap_mo có giá trị f() thấp nhất để xử lý trước.
            # Công thức: f(n) = g(n) + h(n)
            # g(n): Chi phí thực tế từ nguồn.
            # h(n): Chi phí ước lượng đến đích.
            for v in tap_mo:
                if n == None or g_score[v] + self.ham_heuristic(v) < g_score[n] + self.ham_heuristic(n):
                    n = v

            if n == None:
                print('Không tồn tại đường đi!')
                return None

            # 2. Kiểm tra nếu nút hiện tại chính là đích
            if n == diem_ket_thuc:
                duong_di = []

                # Truy vết ngược lại từ đích về nguồn thông qua biến 'cha'
                while cha[n] != n:
                    duong_di.append(n)
                    n = cha[n]

                duong_di.append(diem_bat_dau)
                duong_di.reverse() # Đảo ngược lại để có thứ tự từ Bắt đầu -> Kết thúc

                print('Đã tìm thấy đường đi: {}'.format(duong_di))
                return duong_di

            # 3. Duyệt qua tất cả hàng xóm của nút n hiện tại
            for (m, trong_so) in self.lay_hang_xom(n):
                # m: tên nút hàng xóm
                # trong_so: khoảng cách từ n đến m

                # Nếu nút hàng xóm chưa có trong cả tập mở và tập đóng -> Thêm vào tập mở
                if m not in tap_mo and m not in tap_dong:
                    tap_mo.add(m)
                    cha[m] = n
                    g_score[m] = g_score[n] + trong_so

                # Nếu đã từng thấy nút này, kiểm tra xem đi đường mới này có ngắn hơn không?
                else:
                    if g_score[m] > g_score[n] + trong_so:
```

```
                        # Nếu đường mới ngắn hơn, cập nhật lại chi phí và cha
                        g_score[m] = g_score[n] + trong_so
                        cha[m] = n

                        # Nếu nút này lỡ bị đưa vào tập đóng rồi, phải lôi nó ra lại tập mở để xét lại
                        if m in tap_dong:
                            tap_dong.remove(m)
                            tap_mo.add(m)

            # 4. Sau khi xét hết hàng xóm của n, chuyển n từ tập mở sang tập đóng
            tap_mo.remove(n)
            tap_dong.add(n)

        print('Không tồn tại đường đi!')
        return None

# --- PHẦN CHẠY THỬ (DEMO) ---
if __name__ == "__main__":
    # Tạo dữ liệu đồ thị mẫu (Tương ứng với các điểm A, B, C, D trong hàm heuristic)
    du_lieu_do_thi = {
        'A': [('B', 1), ('C', 3), ('D', 7)],  # Từ A đi được đến B (chi phí 1) và C (chi phí 3)
        'B': [('D', 5)],            # Từ B đi được đến D (chi phí 5)
        'C': [('D', 12)],           # Từ C đi được đến D (chi phí 12)
        'D': []                     # D là đích, không đi đâu nữa
    }
    H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
            }
    # NHẬP TỪ BÀN PHÍM ---
    print("Mời nhập điểm bắt đầu (ví dụ A): ", end="")
    diem_xuat_phat = builtins.input().strip().upper() # Sử dụng builtins.input()
    print("Mời nhập điểm kết thúc (ví dụ D): ", end="")
    diem_dich = builtins.input().strip().upper() # Sử dụng builtins.input()


    # ----------------------------------
    # 2. In Input ra màn hình cho mọi người xem
    print("=" * 40)
    print("DEMO THUẬT TOÁN A* (A STAR)")
    print("=" * 40)
    print("1. DỮ LIỆU ĐẦU VÀO (INPUT):")
    print(f"   - Điểm bắt đầu: {diem_xuat_phat}")
    print(f"   - Điểm kết thúc: {diem_dich}")
    print("   - Cấu trúc đồ thị (Danh sách kề):")
    for nut, hang_xom in du_lieu_do_thi.items():
        print(f"     + Tại {nut} có đường đi tới: {hang_xom}")

    print("-" * 40)
    print("2. QUÁ TRÌNH XỬ LÝ:")

    # 3. Chạy thuật toán
    do_thi = DoThi(du_lieu_do_thi, H) # Pass the H dictionary to the constructor
    do_thi.thuat_toan_a_star(diem_xuat_phat, diem_dich)
    print("=" * 40)
```

```
Mời nhập điểm bắt đầu (ví dụ A): A
Mời nhập điểm kết thúc (ví dụ D): D
========================================
DEMO THUẬT TOÁN A* (A STAR)
========================================
1. DỮ LIỆU ĐẦU VÀO (INPUT):
   - Điểm bắt đầu: A
   - Điểm kết thúc: D
   - Cấu trúc đồ thị (Danh sách kề):
     + Tại A có đường đi tới: [('B', 1), ('C', 3), ('D', 7)]
     + Tại B có đường đi tới: [('D', 5)]
     + Tại C có đường đi tới: [('D', 12)]
     + Tại D có đường đi tới: []
----------------------------------------
2. QUÁ TRÌNH XỬ LÝ:
Đã tìm thấy đường đi: ['A', 'B', 'D']
========================================
```