ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

KHOA Toán - Tin Học

~~~~~~*~~~~~~

BÁO CÁO THỰC HÀNH NHẬP MÔN TRÍ TUỆ NHÂN TẠO

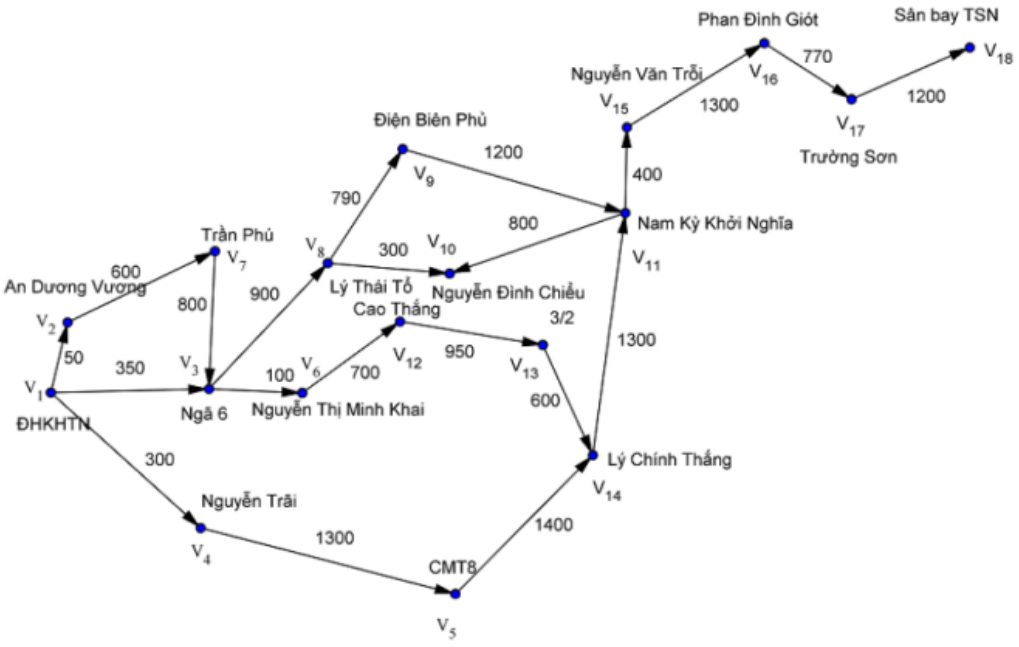
Sinh viên	**:**	**VÒNG VĨNH PHÚ**
MSSV	**:**	**19110413**
Môn Học	**:**	**NHẬP MÔN TRÍ TUỆ NHÂN TẠO**
Trường	**:**	**ĐẠI HỌC KHOA HỌC TỰ NHIÊN**

THÀNH PHỐ HỒ CHÍ MINH, THÁNG 12 NĂM 2021
~~~~~~

Nội dung

1. Ý TƯỞNG BÀI TOÁN

Đề bài : Tìm đường đi ngắn nhất từ trường Đại học Khoa học Tự nhiên (V_1) tới sân bay Tân Sơn Nhất (V_{18}) dùng các thuật toán : BFS,DFS,UCS



1.1 Tìm đường đi không dùng chi phí

Với việc tìm đường đi không dùng chi phí ta có thể sử dụng 2 thuật toán tìm kiếm theo chiều rộng (Breath First Search) và tìm kiếm theo chiều sâu (Depth First Search) để đưa ra giải pháp cho đường đi dựa theo đồ thị đã cho

1.2 Tìm đường đi sử dụng chi phí

Với việc tìm đường đi dùng chi phí ta có thể sử dụng thuật toán Tìm kiếm chi phí đều (Uniform Cost Search) để đưa ra giải pháp cho đường đi dựa theo đồ thị đã cho

2. Breath First Search (BFS)

2.1 Ý tưởng thuật toán

Breath First Search là thuật toán sử dụng cấu trúc dữ liệu hàng đợi (Queue) để lưu trữ thông tin trung gian thu được trong quá trình tiếp kiếm.

- Từ 1 gốc ban đầu xác định và lần lượt duyệt các đỉnh liền kề xung quanh đỉnh gốc vừa xét
    - Tiếp tục quá trình duyệt qua các đỉnh kề vừa xét cho đến khi đạt được kết quả cần tìm hoặc duyệt qua các đỉnh

2.2 Input/Output

- Input : file txt có các thông tin: (Graph, Start, Goal)
    - Graph: Một ma trận 18x18 cho biết vị trí V_i có thể đến được vị trí V_j hay không nếu có là 1 và nếu không là 0 (Với I là dòng và j là cột)
    - Start : Nơi bắt đầu 0 (1)
    - Goal : Nơi kết thúc 17 (18)
- Output : Giải pháp đi từ Start đến Goal với đường đi ngắn nhất

2.3 Xây dựng Hàm và thuật toán

- Khi lấy các thông tin trong file TXT ta sẽ có bộ dữ liệu gồm thông tin của bảng, điểm đầu ,điểm cuối, và ma trận nxn (với n là số node = 18)
- Ma trận nxn này còn gọi là ma trận kề (Adjacency Matrix) ta sẽ 1 hàm chuyển đổi về Danh sách kề (Adjancency List) để dễ dàng tổ chức thông tin cũng như lấy dữ liệu của node lưu trữ trong hàng đợi
- Khởi tạo 1 hàm để tìm đường đi BFS(graph,start,goal)
    - Xây dựng 1 list queue cho phép lưu trữ thông tin vào
    - Ta sẽ truyền thông tin [Start] vào trong queue để khởi tạo node đầu tiên

- Nếu queue chưa rỗng thì ta sẽ cho thực hiện 1 vòng lặp để tìm kiếm, ngược lại ta sẽ xuất ra thông báo không có đường đi dẫn tới kết quả
    - Ta lấy phần tử đầu tiên trong queue cho list(Path) để Path có node đầu tiên
    - Tạo 1 biến current là phần tử cuối của path để kiểm tra đã ở đích đến chưa nếu chưa thì tiếp tục tìm kiếm
    - Tạo 1 vòng lặp để lấy thông tin các đỉnh liền kề cho biến adj (for adj in graph.get(node, []))
    - Tạo 1 mảng explored để lấy thông tin mà path đã đi qua và thêm phần tử thông tin tìm kiếm liền kề (adj)
    - Truyền explored này vào queue để tạo 1 danh sách các đường đi đã explored được từ các node liền kề
    - Khi đã hết explored hết tất cả các đỉnh liền kề queue sẽ trả về cho path danh sách các đường đi và thực hiện như các bước trên đến khi current == goal (nếu danh sách path nào được đến được goal đầu tiên sẽ trả về cho hàm path đó)
- Kết thúc thuật toán

3. Depth First Search (DFS)

3.1 Ý tưởng thuật toán

Dept First Search là thuật toán sử dụng cấu trúc dữ liệu Ngăn xếp (Stack) để lưu trữ thông tin trung gian thu được trong quá trình tiếp kiếm.

- Duyệt đi xa nhất theo từng nhánh

- Khi nhánh đã duyệt hết, lùi về từng đỉnh để tìm và duyệt những nhánh tiếp theo

- Quá trình duyệt chỉ dừng lại khi tìm thấy đỉnh cần tìm hoặc tất cả đỉnh đều đã được duyệt qua

3.2 Input/Output

- Input : file txt có các thông tin: (Graph, Start, Goal)
  - Graph: Một ma trận 18x18 cho biết vị trí V_i có thể đến được vị trí V_j hay không nếu có là 1 và nếu không là 0 (Với I là dòng và j là cột)
  - Start : Nơi bắt đầu 0 (1)
  - Goal : Nơi kết thúc 17 (18)
- Output : Giải pháp đi từ Start đến Goal với đường đi đầu tiên tìm thấy

3.3 Xây dựng thuật toán

- Một số nguyên tắc: node là node hiện tại đang xét trong bài là biến (current)
- Khi lấy các thông tin trong file TXT ta sẽ có bộ dữ liệu gồm thông tin của bảng, điểm đầu ,điểm cuối, và ma trận nxn (với n là số node = 18)
- Hàm chuyển đổi về Danh sách kề (Adjancency List) để tổ chức lại thông tin
- Khởi tạo 1 hàm để tìm đường đi DFS(graph,start,goal)
  - Xây dựng 1 stack để lưu trữ thông tin
  - Thêm node đầu tiên, path đầu tiên vào stack (stack = [(start, [start])])
  - Khởi tạo 1 tập hợp explored để ghi nhớ những phần tử đã đi qua
  - Khi stack chưa rỗng thì ta sẽ thực hiện vòng lặp, ngược lại sẽ xuất thông báo tìm kiếm thất bại
    - Ta lấy node_current và path cuối cùng trong stack để tiến hành xử lý
    - Nếu node_current không bằng goal thì tiếp tục tìm kiếm và ngược lại ta sẽ trả về kết quả path

        - Nếu node_current chưa được explored thì ta sẽ thêm node vào mảng explored
        - Tạo 1 vòng lặp lấy các node của các đỉnh liền kề của node_current
        - Thêm node mới được tìm tháy vào trong path
        - Truyền (node mới được tìm, path) vào stack để thực hiện trả về xem node_current đã tới goal hay chưa nếu chưa thì tiếp tục tìm kiếm các node liền kề và thêm node mới vào path đến khi hết phần tử có thể tìm kiếm, thì lùi về kiểm tra node chưa explored để được kết quả cần tìm (current == goal)
- Kết thúc thuật toán

4. Uniform Cost Search(UCS)

4.1 Ý tưởng thuật toán

Giống như BFS, Uniform Cost Search (UCS) cũng sử dụng cấu trúc dữ liệu hàng đợi (Queue) để lưu trữ thông tin trung gian thu được trong quá trình tiếp kiếm nhưng dữ liệu get ra không phải là phần tử cuối cùng của queue mà là phần tử có trọng số nhỏ nhất (Weight/Cost)

- Từ 1 gốc ban đầu xác định và lần lượt duyệt các đỉnh liền kề xung quanh đỉnh gốc vừa xét
    - Tiếp tục quá trình duyệt qua các đỉnh kề vừa xét cho đến khi đạt được kết quả cần tìm hoặc duyệt qua các đỉnh

4.2 Input/Output

- Input : file txt có các thông tin: (Graph, Start, Goal)
    - Graph: Một ma trận 18x18 cho biết vị trí V_i có thể đến được vị trí V_j hay không và cho biết nếu có thì có trọng số chi phí K và nếu không là 0 (Với I là dòng và j là cột)
    - Start : Nơi bắt đầu 0 (1)
    - Goal : Nơi kết thúc 17 (18)
- Output : Giải pháp đi từ Start đến Goal có trọng số chi phí nhỏ nhất, Tổng trọng số chi phí

4.3 Xây dựng thuật toán

- Một số quy tắc: current node = biến current được sử dụng trong nội dung khởi tạo hàm
- Khi lấy các thông tin trong file TXT ta sẽ có bộ dữ liệu gồm thông tin của bảng, điểm đầu ,điểm cuối, và ma trận nxn (với n là số node = 18)
- Khởi tạo hàm chuyển đổi về Danh sách kề (Adjancency List) để tổ chức lại thông tin bao gồm danh sách các node từ A-> list[node] và trọng số chi phí tới các node đó
- Khởi tạo 1 hàm để tìm đường đi DFS(graph,start,goal)
    - Ta khởi tạo một danh sách queue sử dụng PriorityQueue cho phép lấy phần tử có trọng số nhỏ nhất có trong danh sách
    - Ta tiến hành Put phần tử cost và path chứa node đầu tiên vào queue (queue.put((0,[start])))
    - Nếu queue chưa rỗng ta thực hiện vòng lặp và ngược lại nếu queue rỗng ta sẽ thông báo tìm kiếm thất bại
        - Ta tiến hành lấy cost và list(path) đầu tiên từ queue để xử lý thông tin

- Khởi tạo 1 biến current lấy phần tử cuối của path để kiểm tra tới goal chưa nếu chưa thì tiếp tục và nếu đúng thì trả về path
- Tạo vòng lặp để lấy thông tin các đỉnh liền kề của current node vào biến adj (for adj in graph.get(current,[])) (lưu ý adj là biến tuple lưu path, cost)
- Khởi tạo biến explored = list(path) để lấy thông tin path các node đã đi qua và thêm thông tin adj[1] là thông tin node mới được tìm thấy
- Khởi tạo biến totalCost = cost + adj[0] để tính tổng chi phí các chặng đường đã đi và adj[0] là thông tin chi phí mới được tìm thấy (cost là tổng chi phí đã tính từ các node đã đi qua)
- Ta sẽ truyền tham số explored và total cost vào queue lúc này khi ta thực hiện lại bước 1 lấy thông tin cost và list(path) từ queue PriorityQueue sẽ lấy phần tử có cost nhỏ nhất trong list queue để được ưu tiên xử lý cứ như thế đến khi node cuối cùng của path (current) == goal ta trả về kết quả cost và path

- Kết thúc thuật toán

5. Các thư viên và hàm đã viết hỗ trợ thuật toán:

5.1 Thư viện

5.1.1 PriorityQueue trong Queue

Thư viện cho phép ta khởi tạo một danh sách (list_array) lưu trữ thông tin và lấy thông tin bằng cách chọn trọng số nhỏ nhất, phần tử trong danh sách là dạng tuple như trong hàm UCS thì nó sẽ xét phần tử đầu tiên (cost)
Tham khảo : https://docs.python.org/3/library/queue.html

5.1.2 Defaultdict trong collections

Thư viện cho phép ta khởi tạo một danh sách dạng từ điển bao gồm key,value khi ta gọi key thì cũng có thể truy xuất đến phần tử value nó có nét tương đồng với bài toán khi bài toán cho ta đi từ A->B thì defaultdict cho phép ta lấy thông tin tại node(key) -> node(value)

Tham khảo :

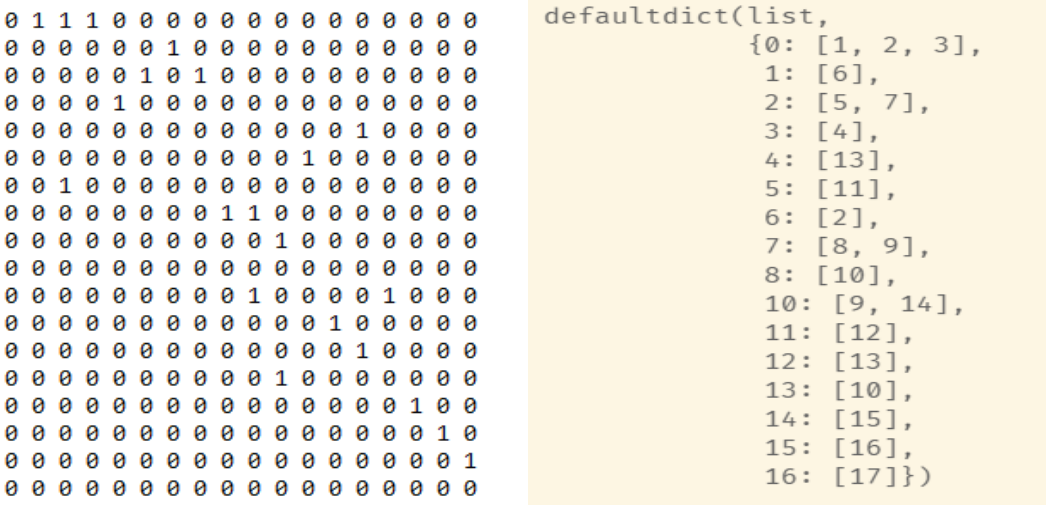
https://docs.python.org/3/library/collections.html#collections.defaultdict

5.2 Hàm hỗ trợ

5.2.1 adjm_to_adjl(matrix)

Hàm chuyển đổi từ ma trận kề (Adjacency Matrix) thành Danh sách kề (Adjancency List) bằng cách sử dụng hàm defaultdict(list)

Bằng cách này khi ta gán list = defaultdict(list) ứng với mỗi dòng I của list ta sẽ có key I và value [j] nếu matrix[i][j] = 1 (tìm thấy đường đi trong matrix)



```
0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
defaultdict(list,
            {0: [1, 2, 3],
             1: [6],
             2: [5, 7],
             3: [4],
             4: [13],
             5: [11],
             6: [2],
             7: [8, 9],
             8: [10],
             10: [9, 14],
             11: [12],
             12: [13],
             13: [10],
             14: [15],
             15: [16],
             16: [17]})
```

Ví dụ về hàm chuyển đổi Adjacency Matrix (bên trái) thành Adjancency List (bên phải)

5.2.2 adjm_to_adjlUCS(matrix)

Cũng là 1 hàm chuyển đổi từ Adjacency Matrix thành Adjancency List bằng cách sử dụng hàm defaultdict(list)

Nhưng khác với trên với mỗi vòng I của list ta sẽ có key I và value [giá trị matrix[i][j] và j] với matrix[i][j] là trọng số chi phí của đường đi

```
0 50 350 300 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 600 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 100 0 900 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1300 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1400 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 700 0 0 0 0 0
0 0 800 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 790 300 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1200 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 800 0 0 0 0 400 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 950 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 600 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1300 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1300 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 770 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1200
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
defaultdict(list,
            {0: [(50, 1), (350, 2), (300, 3)],
             1: [(600, 6)],
             2: [(100, 5), (900, 7)],
             3: [(1300, 4)],
             4: [(1400, 13)],
             5: [(700, 11)],
             6: [(800, 2)],
             7: [(790, 8), (300, 9)],
             8: [(1200, 10)],
             10: [(800, 9), (400, 14)],
             11: [(950, 12)],
             12: [(600, 13)],
             13: [(1300, 10)],
             14: [(1300, 15)],
             15: [(770, 16)],
             16: [(1200, 17)]})
```

Ví dụ về hàm chuyển đổi Adjacency Matrix có chi phí (bên trái) thành Adjancency List (bên phải)