BÀI BÁO CÁO MÔN PHÂN TÍCH VÀ THIẾT KẾ THUẬT TOÁN

Ngày 4 tháng 2 năm 2021

Mục lục

I Giới thiệu bài toán:

1 Mô tả:

Bài toán đường đi ngắn nhất của tất cả các cặp đỉnh là bài toán tìm một đường đi giữa cho tất cả các cặp đỉnh sao cho tổng các trọng số của các cạnh tạo nên đường đi đó là nhỏ nhất. Cho trước một đồ thị có trọng số (nghĩa là một tập đỉnh V, một tập cạnh E, và một hàm trong số có giá trị thực $f : E \to \mathbb{R}$), xét tất cả đỉnh v thuộc V, tìm một đường đi P từ v tới mỗi đỉnh v' thuộc V sao cho:

$$\sum_{p \in P}(f(p))$$

là nhỏ nhất trong tất cả các đường nối từ v tới v'.

- **Input:** : đồ thị có hướng có trọng số của cạnh.
- **Output:** :ịđường đi ngắn nhất giữa các cặp đỉnh trong đồ thị.

2 Lịch sử:

- Rất khó để truy ngược lịch sử của bài toán tim đường đi ngắn nhất của tất cả các cặp đỉnh.
- So với các bài toán tối ưu hóa tổ hợp khác: như cây bao trùm nhỏ nhất, bài toán vận chuyển, việc nghiên cứu toán học trong bài toán đường đi ngắn nhất bắt đầu khá muộn.
- Các phương pháp tiếp cận không tối ưu đã được nghiên cứu như: Rosenfeld [1956], người đã đưa ra phương pháp phỏng đoán để xác định một tuyến đường vận tải đường bộ tối ưu thông qua một mô hình tắc nghẽn giao thông nhất định.
- Việc tìm đường đi, đặc biệt là tìm kiếm trong mê cung, thuộc về các bài toán đồ thị cổ điển, và các tài liệu tham khảo cổ điển là Wiener [1873], Lucas [1882] và Tarry [1895], Lloyd và Wilson [1976]. Chúng tạo cơ sở cho các kỹ thuật tìm kiếm theo chiều sâu.
- Các vấn đề về đường đi cũng được nghiên cứu vào đầu những năm 1950 trong bối cảnh 'định tuyến thay thế', tức là tìm đường ngắn thứ hai nếu đường ngắn nhất bị chặn. Điều này áp dụng cho việc sử dụng đường cao tốc (Trueblood [1952]), cũng như định tuyến cuộc gọi điện thoại.

3 Ứng dụng:

- Ứng dụng trong viễn thông, bài toán đường đi ngắn nhất đôi khi được gọi là bài toán đường đi có độ trễ nhỏ nhất (min-delay path problem) và thường được gắn với một bài toán đường đi rộng nhất (widest path problem). ví dụ đường đi rộng nhất trong các đường đi ngắn nhất (độ trễ nhỏ nhất) hay đường đi ngắn nhất trong các đường đi rộng nhất.
- Kết hợp với bản đồ số.
- Ứng dụng trong bài toán xe buýt.
- Ứng dụng tron bài toán taxi.

4 Đề bài liên hệ với bản thân:

Bài toán tìm đường đi đến mấy chỗ giao hàng sao cho tốn ít chi phí nhất:

- Em đang thực hiện việc bán hàng online (tai nghe, chuột , cáp sạc), mọi người thường chọn mặt hàng mua sau đó gửi thông tin liên lạc gồm tên , số điện thoại , địa chỉ để em chốt đơn. Vì là sinh viên nghèo nên em không thể công tác với các công ty vận chuyển được mà phải tự mình đi giao hàng.
- Hôm nay, em có 4 đơn hàng ở Nhân Văn , Tự Nhiên , KTX A và KTX B. Từ trường, em sẽ tìm đường ngắn nhất đi tới Nhân văn , tiếp tục tìm đường ngắn nhất từ Nhân Văn đến Tự Nhiên và tiếp tục với KTX A và KTX B. Do cứ giao xong 1 đơn lại phải xem xét xem đường đi ngắn nhất tiếp theo là đường nào nên việc đi giao rất mất thời gian và có khi lại trễ hẹn với khách vì có rất nhiều đường giữa các địa điểm.
- - Chính vì thế em đã tìm hiểu trên mạng và thấy có các thuật toán có thể giải quyết bài toán tìm đường đi ngắn nhất giữa các điểm với nhau sao cho nó là ngắn nhất. Nhờ đó trước khi giao hàng em chỉ cần chạy thuật toán một lần lấy kết quả và chỉ cần tra lại kết quả khi đi giao hàng.

II Các thuật toán dùng để giải quyết bài toán

1 Thuật toán Floyd–Warshall:

1.1 Dẫn nguồn và lịch sử phát triển thuật toán:

1.1.1 Dẫn nguồn:

- Khái niệm, lịch sử, mã giả và chi tiết về thuật toán Floyd-warshall được tham khảo trên WikipediA https://en.wikipedia.org/wiki/Floyd\T5\textendashWarshall_algorithm.
- Source code được tham khảo trên Techie Delight

  https://www.techiedelight.com/pairs-shortest-paths-floyd-warshall-algorithm/?fbc

1.1.2 Lịch sử phát triển:

- Thuật toán Floyd – Warshall là một ví dụ về lập trình động, và được Robert Floyd công bố và được công nhận vào năm 1962.
- Về cơ bản, nó giống với các thuật toán được Bernard Roy công bố trước đó vào năm 1959 và cũng bởi Stephen Warshall vào năm 1962 để tìm ra điểm bắc cầu của một đồ thị, và có liên quan chặt chẽ với thuật toán của Kleene (đã xuất bản vào năm 1956) để chuyển đổi một automaton hữu hạn xác định thành một biểu thức chính quy.
- Công thức hiện tại của thuật toán dưới dạng ba vòng for lồng nhau lần đầu tiên được Peter Ingerman mô tả vào năm 1962.

1.2 Phương pháp đã dùng để thiết kế:

Thuật toán Floyd-Warshall được thiết kế bằng phương pháp quy hoạch động (Dynamic Programing). Nó chia bài toán tìm đường đi ngắn nhất giữa các cặp điểm, thành các bài toán tìm đường ngắn nhất giữa hai điểm là đường nối giữa hai điểm đó hoặc nó sẽ đi qua một vài điểm trung gian. Do tất cả các bài toán con được giải quyết trước, sau đó dùng kết quả để tìm lời giải cho bài toán lớn nên nó được tiếp cận theo hướng bottom-up.

1.3 Mã giả:

// graph là ma trận đồ thị có hướng cho trước chứa trọng số của các cạnh trong đồ thị.

// khởi tạo ma trận lưu chi phí đường đi cost.

$cost \leftarrow graph$

for $k \in V(G)$ do : // xét các đỉnh trung gian.

for $u \in V(G)$ do : // xét từng đỉnh bắt đầu.

for $v \in V(G)$ do : // xét từng đỉnh kết thúc

// xét đường đi ngắn nhất giữa đường hiện tại và đường đi qua trung gian k.

$cost[u,v]$ ¸$mincost[u,v], cost[u,k] + cost[k,v]$

// kết thúc thuật toán nếu phát hiện chu kỳ âm .

if $cost[v,v] < 0$ then :

return

1.4 Phân tích độ phức tạp bằng phương pháp toán học:

```
(1) for k in range(N):
(2)     for v in range(N):
(3)         for u in range(N):
(4)             if cost[v,k] != float('inf') and cost[k,u] != float('inf') and (cost[v,k] + cost[k,u] < cost[v,u]):
(5)                 cost[v,u] = cost[v,k] + cost[k,u]
(6)                 path[v,u] = path[k,u]
(7)         if cost[v,v] < 0:
(8)             return
```

Ta có:
$T(1) = 1(1 + 1 * 1)(1 + 9 * 1)$
$T(2) = 2(2 + 1 * 2)(2 + 9 * 2)$
$T(3) = 3(3 + 1 * 3)(3 + 9 * 3)$
$\Rightarrow T(n) = n(n + n)(n + 9n)$
$= 20n^3$
Đặt: $C = 20$
$T(n) = Cn^3$
$=> T(n) = O(n^3)$
Với n là số đỉnh của đò thị.

1.5 Mã nguồn cài đặt:

```
# chi phí nhỏ nhất di chuyển giữa các đỉnh
cost = adjMatrix.copy()

#  tạo ma trận đường đi rỗng có kích thước bằng kích thước ma trận ban đầu
path = np.asmatrix([[None for x in range(N)] for y in range(N)])

# tạo ma trận đường đi từ đỉnh v -> u thông qua các cạnh nối trên đồ thị
for v in range(N):
    for u in range(N):
        # chỉ có 1 đỉnh , gán giá trị 0
        if v == u:
            path[v,u] = 0
        # có cạnh nối từ v -> u , gán giá trị bằng cạnh bắt đầu là v
        elif (cost[v,u] != float('inf')):
            path[v,u] = v
        # không có cạnh nối giữa 2 đỉnh , gán giá trị -1
        else:
            path[v,u] = -1
# thuật toán Floyd-Warshall
# k đại diện cho đỉnh đang xét
# v, u lần lượt là cột và hàng của ma trận
# thuật toán sẽ tìm đường đi ngắn nhất của lần lượt từng đỉnh cho đến cái đỉnh còn lại
for k in range(N):
    for v in range(N):
        for u in range(N):
            # nếu có đường đi từ v -> u đi qua đỉnh trung gian k , sao cho cost(v -> k) + cost(k ->
            # hay nói cách khách đi đường v -> k -> u tốn ít chi phí hơn đi đường v -> u
            # thì cập nhật lại chi phí đi từ v -> u và cập nhật lại đỉnh đi đến u
            if cost[v,k] != float('inf') and cost[k,u] != float('inf') and (cost[v,k] + cost[k,u] <
                cost[v,u] = cost[v,k] + cost[k,u]
                path[v,u] = path[k,u]
        # nếu phần từ đường chéo trờ thành âm, đồ thị chứa chu kì trọng số âm
        if cost[v,v] < 0:
            #print("phát hiện trọng số ấm")
            return
```

1.6 Cách thức phát sinh Input/Output đã dùng để kiểm tra tính đúng đắn của cài đặt:

1.6.1 Kiểm tra tính đúng đắn của cài đặt : Ký hiệu:

- $V(G)$: tập hợp tất cả các nút trong đồ thị G.
- $D[u, v]$: chứa chi phí đường đi ngắn nhất của tất cả cặp u, v.
- S_k : tập hợp các nút trung gian k của đường đi từ v đến v'.
- $S_k - path$: đường đi đi qua nút trung gian trong S_k.

- D_k : chi phí đường đi qua các nút trung gian k.

Trường hợp cơ bản : $k = 0, S_0 = \emptyset$, luôn đúng.

Giả sử : sau $k \geq 0$ lần lặp kết quả vẫn đúng.

Bước quy nạp : chứng minh $D_k + 1[u, v]$ sau $k + 1$ lần lặp thì đường đi ngắn nhất từ u đến v là $S_{k+1} - path$. Giả sử x là nút cuối cùng được tìm thấy sau khi thực hiện vòng lặp, khi đó $S_{k+1} = S_k \cup x$.

- Cố định một cặp nút $u, v \in V(G)$ và L là độ dài của đường đi ngắn nhất từ u đến v tương đương với đường đi $S_{k+1} - path$, sao cho $L \leq D_{k+1}[u, v]$.
- Chọn một đường đi $S_{k+1}{}^{\smile} path$ đi từ u đến v là γ và chiều dài là L.
  - Nếu $x \notin \gamma$ thì kết quả đúng với giả thuyết quy nạp.
  - Nếu $x \in \gamma$ gọi γ_1 và γ_2 lần lượt là đường con đi từ u đến x và x đến v. khi đó γ_1 và γ_2 là các đường đi đúng với giả thuyết quy nạp :

$$L \geq |\gamma_1| + |\gamma_2| \geq D_k[u, x] + D_k[x, v] \geq D_k + 1[u, v]$$

1.6.2 Cách phát sinh Input/Output cho test case: Input: Đầu vào của bài toán là một tra trận đồ thị có hướng, trong đó trọng số của cạnh có thể chứa giá trị âm

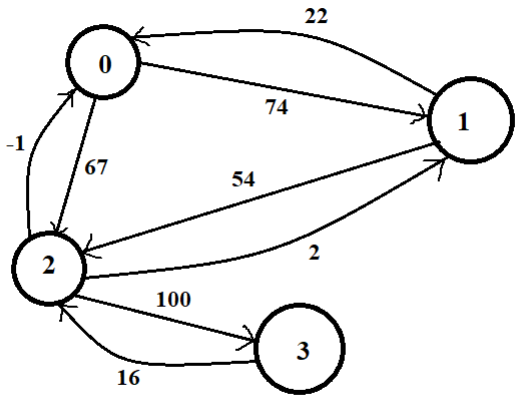
- Ta qui ước:
  - Đường chéo của ma trận có giá trị là 0.
  - Nếu không có cạnh nối giữa hai đỉnh của ma trận thì giá trị của nó là inf.
  - Nếu tồn tại cạnh nối giữa hai đỉnh thì giá trị của nó bằng chi phí của cạnh đó
- Ta sử dụng hàm random.random() để random ngẫu nhiên n đỉnh và p là khả năng xuất hiện cạnh nối giữa 2 đỉnh
- Sử dụng thư viện networkx.binomial_graph để tạo một đồ thị có hướng từ n và p
- Sau đó ta chuẩn hóa nó về dạng ma trận bằng hàm networkx.convert_matrix.to_numpy_matrix
- Tiếp tục chuẩn hóa ma trận thu được về dạng như ban đầu đã quy ước trong đó giá trị của cạnh sẽ được chọn ngẫu nhiên từ [-10,100]

Output:

- ma trận cost chứa giá trị đường đi ngắn nhất từ một đỉnh đến một đỉnh khác
- ma trận path chứa đỉnh là đường đi đến đỉnh đang xét sau cho chi phí là ngắn nhất
- thử nghiệm vẽ một số đồ thị có số đỉnh nhỏ để kiểm chứng

1.6.3 Kiểm tra thử với trường hợp cơ bản: Ta có đồ thị: Sau khi chuyển đổi qua ma trận bằng cách :

- Tạo ma trận G có kích thước bằng VxV với V là số đỉnh của đồ thị.
- Lần lượt xét từng đỉnh :
  - Gán $G[v, v] = 0$.
  - Nếu từ đỉnh đang xét v có cạnh hướng tới đỉnh u , với trọng số cạnh là p thì $G[v, u] = p$.
  - Với các đỉnh u không có cạnh nối từ v tới u thì $G[v, u] = \text{inf}$.




```
graph
 [[  0.  74.  67.  inf]
 [ 22.   0.  54.  inf]
 [ -1.   2.   0. 100.]
 [ inf  inf  16.   0.]]
```

Ma trận được chuyển từ đồ thị : Sau khi chạy thuật toán Floyd-Warshall ta thu được 2 kết quả :

- *cost* là ma trận chứa chi phí đường đi từ đỉnh bất kì đến các đỉnh khác.
- path là ma trận chứa đường đi từ đỉnh bất kì đến các đỉnh khác.

```
cost
 [[  0.  69.  67. 167.]
 [ 22.   0.  54. 154.]
 [ -1.   2.   0. 100.]
 [ 15.  18.  16.   0.]]
```

```
path
 [[0 2 0 2]
 [1 0 1 2]
 [2 2 0 2]
 [2 2 3 0]]
```

Truy xuất đường đi và chi phí từ đỉnh v đến đỉnh u:

- Gọi C và S lần lượt là chi phí và đường đi ngắn nhất giữa 2 đỉnh đang xét:
  - $v = 0$
    - $u = 1$
      - Xét $path[v, u] = path[0, 1] = 2 \Rightarrow$ đường đi ngắn nhất từ $v \rightarrow u$ phải đi qua đỉnh số $2 \Rightarrow S_1 = \{2, 1\}$.

· xét tiếp $path[v,2] = path[0,2] = 0 \Rightarrow$ đường đi ngắn nhất từ $0 \to 2$ là cạnh nối từ đỉnh 0 đến đỉnh $2 \Rightarrow S_2 = \{0,2,1\}$.

· $S = S_1 \cup S2 = \{0,2,1\} \Rightarrow$ đường đi ngắn nhất từ $0 \to 1$ là $0 \to 2 \to 1$.

· $C = cost[v,u] = cost[0,1] = 69$.

* $u = 2$

· Xét $path[v,u] = path[0,2] = 0 \Rightarrow$ đường đi ngắn nhất từ $0 \to 2$ là cạnh nối từ đỉnh 0 đến đỉnh $2 \Rightarrow S = \{0,2\}$.

· $S = \{0,2\} \Rightarrow$ đường đi ngắn nhất từ $0 \to 2$ là $0 \to 2$.

· $C = cost[v,u] = cost[0,2] = 67$.

* $u = 3$

· Xét $path[v,u] = path[0,3] = 2 \Rightarrow$ đường đi ngắn nhất từ $v \to u$ phải đi qua đỉnh số $2 \Rightarrow S1 = \{2,3\}$.

· Xét tiếp $path[v,2] = path[0,2] = 0 \Rightarrow$ đường đi ngắn nhất từ $0 \to 2$ là cạnh nối từ đỉnh 0 đến đỉnh $2 \Rightarrow S_2 = \{0,2,3\}$.

· $S = S_1 \cup S_2 = \{0,2,3\} \Rightarrow$ đường đi ngắn nhất từ $0 \to 1$ là $0 \to 2 \to 3$.

· $C = cost[v,u] = cost[0,3] = 167$.

– $v = 1$

* $u = 0$

· Xét $path[v,u] = path[1,0] = 1 \Rightarrow$ đường đi ngắn nhất từ $1 \to 0$ là cạnh nối từ đỉnh 1 đến đỉnh $0 \Rightarrow S = \{1,0\}$.

· $S = \{1,0\} \Rightarrow$ đường đi ngắn nhất từ $1 \to 0$ là $1 \to 0$.

· $C = cost[v,u] = cost[1,0] = 22$.

* $u = 2$

· Xét $path[v,u] = path[1,2] = 1 \Rightarrow$ đường đi ngắn nhất từ $1 \to 2$ là cạnh nối từ đỉnh 1 đến đỉnh $2 \Rightarrow S = \{1,2\}$.

· $S = \{1,2\} \Rightarrow$ đường đi ngắn nhất từ $1 \to 2$ là $1 \to 2$.

· $C = cost[v,u] = cost[1,2] = 54$.

* $u = 3$

· Xét $path[v,u] = path[1,3] = 2 \Rightarrow$ đường đi ngắn nhất từ $v \to u$ phải đi qua đỉnh số $2 \Rightarrow S1 = \{2,3\}$.

· Xét tiếp $path[v,2] = path[1,2] = 1 \Rightarrow$ đường đi ngắn nhất từ $1 \to 2$ là cạnh nối từ đỉnh 1 đến đỉnh $2 \Rightarrow S_2 = \{1,2\}$.

· $S = S_1 \cup S_2 = \{1,2,3\} \Rightarrow$ đường đi ngắn nhất từ $1 \to 3$ là $1 \to 2 \to 3$.

· $C = cost[v,u] = cost[1,3] = 154$.

– $v = 2$

* $u = 0$

· Xét $path[v,u] = path[2,0] = 2 \Rightarrow$ đường đi ngắn nhất từ $2 \to 0$ là cạnh nối từ đỉnh 2 đến đỉnh $0 \Rightarrow S = \{2,0\}$.

· $S = \{2,0\} \Rightarrow$ đường đi ngắn nhất từ $2 \to 0$ là $2 \to 0$.

· $C = cost[v,u] = cost[2,0] = -1$.

* $u = 1$

· Xét $path[v,u] = path[2,1] = 2 \Rightarrow$ đường đi ngắn nhất từ $2 \to 1$ là cạnh nối từ đỉnh 2 đến đỉnh $1 \Rightarrow S = \{2,1\}$.

· $S = \{2,1\} \Rightarrow$ đường đi ngắn nhất từ $2 \to 1$ là $2 \to 1$.

· $C = cost[v,u] = cost[2,1] = 2$.

* $u = 3$
  - Xét $path[v, u] = path[2, 3] = 3 \Rightarrow$ đường đi ngắn nhất từ $2 \rightarrow 3$ là cạnh nối từ đỉnh 2 đến đỉnh $3 \Rightarrow S = \{2, 3\}$.
  - $S = \{2, 1\} \Rightarrow$ đường đi ngắn nhất từ $2 \rightarrow 3$ là $2 \rightarrow 3$.
  - $C = cost[v, u] = cost[2, 3] = 100$.

– $v = 3$
* $u = 0$
  - Xét $path[v, u] = path[3, 0] = 2 \Rightarrow$ đường đi ngắn nhất từ v -> u phải đi qua đỉnh số $2 \Rightarrow S1 = \{2, 0\}$.
  - Xét tiếp $path[v, 2] = path[3, 2] = 3 \Rightarrow$ đường đi ngắn nhất từ $3 \rightarrow 2$ là cạnh nối từ đỉnh 3 đến đỉnh $2 \Rightarrow S_2 = \{3, 2\}$.
  - $S = S_1 \cup S_2 = \{3, 2, 0\} \Rightarrow$ đường đi ngắn nhất từ $3 \rightarrow 0$ là $3 \rightarrow 2 \rightarrow 0$.
  - $C = cost[v, u] = cost[3, 0] = 15$.
* $u = 2$
  - Xét $path[v, u] = path[3, 1] = 2 \Rightarrow$ đường đi ngắn nhất từ $v \rightarrow u$ phải đi qua đỉnh số $2 \Rightarrow S1 = 2, 1$.
  - Xét tiếp $path[v, 2] = path[3, 2] = 3 \Rightarrow$ đường đi ngắn nhất từ $3 \rightarrow 2$ là cạnh nối từ đỉnh 3 đến đỉnh $2 \Rightarrow S2 = \{3, 2\}$.
  - $S = S_1 \cup S_2 = \{3, 2, 1\} \Rightarrow$ đường đi ngắn nhất từ $3 \rightarrow 1$ là $3 \rightarrow 2 \rightarrow 1$.
  - $C = cost[v, u] = cost[3, 1] = 18$.
* $u = 3$
  - Xét $path[v, u] = path[3, 2] = 3 \Rightarrow$ đường đi ngắn nhất từ $3 \rightarrow 2$ là cạnh nối từ đỉnh 3 đến đỉnh $2 \Rightarrow S = \{3, 2\}$.
  - $S = \{3, 2\} \Rightarrow$ đường đi ngắn nhất từ $3 \rightarrow 2$ là $3 \rightarrow 2$.
  - $C = cost[v, u] = cost[3, 2] = 16$.

Kiểm tra lại so với đồ thị ta cũng thấy sau khi chạy thuật toán cho kết quả chính xác so với đồ thị.

1.7 Phân tích độ phức tạp bằng cài đặt:

1.7.1 Độ phức tạp thời gian:

- Sử dụng hàm CProfile để tiến hành đo thời gian thực hiện của các đồ thị có số đỉnh tăng dần:

```
for graph in list_Graph:
  cProfile.run('floydWarshall(graph,len(graph))')
```

- Sau khi thực hiện thu được 50 cặp giá trị là số đỉnh và thời gian thực thi
- Sử dụng LinearRegression để tìm hệ số X của các hàm độ phức tạp thuật toán cơ bản

```
1  print(result)
```

```
[['lg', array([204.559207])], ['sqrt', array([67.79153381])], ['N', array([2.76172225])], ['nlgn', array([0.30275637])], ['n**2', array([0.00485
```

- Thông qua việc tính toán ta tìm được mean square error của giá trị thời gian tính được và giá trị thời gian ban đầu , xét hàm nào có giá trị MSE nhỏ nhất thì có khả năng hàm đó là độ phức tạm của thuật toán cần tìm

665	2215.773		1918.195	88552.81	1748.181	218642.7	1836.545	143813.7	1887.942	107473.3	2145.561	4929.764	2232.398	276.3778
731	2950.225		1946.121	1008226	1832.88	1248459	2018.819	867517.2	2105.53	713510.1	2592.581	127909.5	2965.232	225.2236
		X	204.5592		67.79153		2.761722		0.302756		0.004852		7.59E-06	
		MSE		1.32E+06		2.78E+05		1.20E+05		1.01E+05		22214.65		5.89E+01
					Min	5.89E+01	suy ra	N**3 la do phuc tap can tim						

- Kết luận : độ phức tạp $O(n^3)$ chính là độ phức tạp thời gian cần tìm , khi so sánh với kết quả khi phân tích bằng toán học cũng chính xác.

1.7.2 Độ phức tạp bộ nhớ :

- Sử dụng hàm __sizeof__() để đo số byte bộ nhớ đã dùng để lưu ma trận dist, với số đỉnh của đồ thị tăng dần:

```
dataSize = []
for graph in list_Graph:
  dist, path = floydWarshall(graph,len(graph))
  size = dist.__sizeof__()
  dataSize.append(len(graph),size)
```

- Sau khi chạy ta thu được 50 cặp giá trị là số đỉnh và bộ nhớ để lưu giá trị của mảng *dist*.
- Sử dụng LinearRegression để tìm hệ số X của các hàm độ phức tạp thuật toán cơ bản.
- Thông qua việc tính toán ta tìm được mean square error của giá trị bộ nhớ đo được và bộ nhớ tính được, xét hàm nào có giá trị MSE nhỏ nhất thì có khả năng hàm đó là độ phức tạp bộ nhớ của thuật toán cần tìm.
- Kết luận : $O(n^2)$ chính là độ phức tạp bộ nhớ cần tìm.

	log(n)	log(n)*32	sqrt(n)	sqrt(n)*102979.6073	n*4001.71	nlog(n)	nlog(n)*4	n^2	n^2*6.595	n^3	n^3*0.009	n^4	n^4*1.424	n^5	n^5*1.986	2^n	2^n*-3.58544121E-13	
X		327262.6		102979.6		4001.713		4.33E+02		6.599302		0.009948		1.42E-05		1.99E-08		-3.59E-13	
MSE			2.96E+12		7.39E+11		1.03E+11		5.74E+10		2.89E+08		3.47E+10		9.17E+10		1.49E+11		3.347671E+413
						Min	2.89E+08	suy ra	n^2 là độ phức tạp cần tìm										

2 Thuật toán Dijkstra:

2.1 Dẫn nguồn và lịch sử phát triển thuật toán:

2.1.1 Dẫn nguồn:

- Khái niệm, lịch sử, mã giả và chi tiết về thuật toán Dijkstra được tham khảo trên WikipediA https://en.wikipedia.org/wiki/Dijkstra.
- Source code được tham khảo trên Techie Delight

  https://www.techiedelight.com/pairs-shortest-paths-floyd-warshall-algorithm/?fbc

2.1.2 Lịch sử phát triển:

- Dijkstra đã nghĩ về bài toán đường đi ngắn nhất khi làm việc tại Trung tâm Toán học ở Amsterdam năm 1956.
- Mục tiêu của ông là chọn cả một bài toán và giải pháp (sẽ được tạo bởi máy tính) mà những người không thuần tính toán vẫn có thể hiểu được.
- Ông đã thiết kế thuật toán đường đi ngắn nhất và sau đó triển khai nó cho ARMC(máy tính mới với tên ARMC) bằng bản đồ giao thông được đơn giản hóa một chút của 64 thành phố ở Hà Lan.
- Một năm sau, ông gặp một vấn đề khác từ các kỹ sư phần cứng làm việc trên máy tính tiếp theo của học viện: giảm thiểu lượng dây cần thiết để kết nối các chân trên bảng điều khiển phía sau của máy.
- Như một giải pháp, ông đã phát hiện lại thuật toán Prim (được biết đến trước đó với Jarník, và cũng được Prim khám phá lại). Dijkstra đã xuất bản thuật toán vào năm 1959, 2 năm sau Prim và 29 năm sau Jarník.

2.2 Phương pháp đã dùng để thiết kế:

Thuật toán Floyd-Warshall được thiết kế bằng phương pháp quy hoạch động (Dynamic Programing). Nó chia bài toán tìm đường đi ngắn nhất giữa các cặp điểm, thành các bài toán tìm đường ngắn nhất giữa hai điểm là đường nối giữa hai điểm đó hoặc nó sẽ đi qua một vài điểm trung gian. Do tất cả các bài toán con được giải quyết trước, sau đó dùng kết quả để tìm lời giải cho bài toán lớn nên nó được tiếp cận theo hướng bottom-up.

2.3 Mã giả:

// duyệt lần lượt các đỉnh nguồn trong tập đỉnh V.

for $k \in V(G)$ do : //(1)

    // khởi tạo cá biến

for $v \in V(G)$ do : //(2)

// ma trận chứa chi phí đường đi từ đỉnh nguồn tới các // đỉnh còn lại. $dist[v] \leftarrow float(`inf')$

// ma trận chứa các đỉnh đã xét.

$sptSet[v] \leftarrow Flase$

// xét đỉnh đầu tiên cũng là đỉnh nguồn k.

$dist[k] \leftarrow 0$

for $cout \in V(G)$ do : //(3)

// tìm đỉnh u chứa cạnh ngắn nhất từ đỉnh nguồn đang xét min ,$float(`inf')$

$u \leftarrow S$

for $v \in V(G)$ do : //(4)

// nếu cạnh ngắn nhất và không nằm trong tập đỉnh đang xét.

$if dist[v] < min$ and $sptSet[v] = Falsethen : min \leftarrow dist[v]$

$u \leftarrow S$

// cập nhật lại đường đi nếu tồn tại đường đi ngắn

// hơn đi từ đỉnh nguồn và đỉnh đó là đỉnh chưa xét

for $v \in V(G)$ do : //(5)

if $sptSet[v] = False$ then :

$dist[v] \leftarrow min\{dist[v], dist[u] + graph[u, v]\}$

2.4 Phân tích độ phức tạp bằng phương pháp toán học:

Theo như mả giã ta có thế thấy thuật toán thực hiện chủ yếu bằng 5 vòng for được đánh số từ (1) tới (5) , xét từ vòng for :

- for (5):

$T_5(1) = 1$
$T_5(2) = 2$
$T_5(3) = 3$
$\Rightarrow T_5(V) = V$

- for (4):

$T_4(1) = 1 * 2$
$T_4(2) = 2 * 2$
$T_4(3) = 3 * 2$
$\Rightarrow T_4(V) = 2 * V = V$

- for (3):

$T_3(1) = 1 * () = 1 * (V + V) = 2 * V$
$T_3(2) = 2 * (T_4(V) + T_5(V)) = 2 * (V + V) = 2 * 2 * V$
$T_3(3) = 3 * (T_4(V) + T_5(V)) = 3 * (V + V) = 3 * 2 * V$
$\Rightarrow T_3(V) = V * (2 * V) = 2 * V^2 = V^2$

- for (2):

$T_2(1) = 1 * 2$
$T_2(2) = 2 * 2$
$T_2(3) = 3 * 2$
$\Rightarrow T_2(V) = 2 * V = V$

- for (1):

$T_1(1) = 1 * (T_2(V) + 1 + T_3(V)) = 1 * (1 + V + V^2)$
$T_1(2) = 2 * (T_2(V) + 1 + T_3(V)) = 2 * (1 + V + V^2)$
$T_1(3) = 3 * (T_2(V) + 1 + T_3(V)) = 3 * (1 + V + V^2)$
$\Rightarrow T_1(V) = V * (T_2(V) + 1 + T_3(V)) = V * (V + V^2) = V^3 + V^2 + V$

$\Rightarrow T(V) = T_1(V) = O(V^3)$
Với V là số đỉnh của đồ thị.

2.5 Mã nguồn cài đặt:

```python
# thuật toán Dijkstra
def dijkstra(graph, V, S):
  # khởi tạo các biến
  dist = [float('inf')] * V
  dist[S] = 0
  sptSet = [False] * V

  for cout in range(V):
    min = sys.maxsize
    # chọn đỉnh có đường đi ngắn nhất từ đỉnh nguồm
    # u luôn bằng đỉnh nguồn ở lần chạy đầu tiên
    u = S;
    for v in range(V):
      if dist[v] < min and sptSet[v] == False:
        min = dist[v]
        u = v
    # cập nhật đỉnh u vào mảng các đỉnh đã xét
    sptSet[u] = True

    # chỉ cập nhật giá trị của những đỉnh liền kề với đỉnh đã chọn nếu khoảng cách hiện tại lớn hơn khoản
    # mới và đỉnh không nằm trong các đỉnh đã xét
    for v in range(V):
      if graph[u,v] > 0 and sptSet[v] == False and dist[v] > dist[u] + graph[u,v]:
        dist[v] = dist[u] + graph[u,v]

  # in đường đi từ đỉnh nguồn tới các đỉnh còn lại
  # printSolution(V,S,dist)
  return dist
```

```python
# tìm tất cả cặp đường đi ngắn nhất với Dijkstra
def All_pair_Dijkstra(graph):
  dist = []
  for v in range(len(graph)):
    d = dijkstra(graph,len(graph),v)
    dist.append(d)
  # print("dist\n",dist)
```

2.6 Cách thức phát sinh Input/Output đã dùng để kiểm tra tính đúng đắn của cài đặt:

2.6.1 Kiểm tra tính đúng đắn của thuật toán: Giả sử : với mọi $x \in V, d(x) = \delta(x)$ Chứng minh quy nạp :

- Trường hợp cơ bản : $V = 1$,Vì V chỉ phát triển về kích thước nên để $V = 1$ thì $V = \{s\}$ và $d(s) = 0 = delta(s)$, điều này đúng.
- Giả thuyết quy nạp : Gọi u là đỉnh cuối cùng được thêm vào V . Cho $V' = V \cup \{u\}$. Giả sử : với mỗi $x \in V`, d(x) = \delta(x)$. (1)
- Theo giả thuyết (1) với mọi đỉnh u không nằm trong $V`$, chúng ta sẽ có được chi phí của những đường đi . Chúng ta chỉ cần chứng minh rằng $d(u) = \delta(u)$ để hoàn thiện việc chứng minh
- Giả sử mâu thuẫn rằng đường đi ngắn nhất từ s đến u là Q và có độ dài :

$$L(Q) < d(u)$$

- Q bắt đầu từ $V`$ và tại một số điểm không nằm trong $V`$ (để đi tới u không nằm trong $V`$ ). Gọi $xy$ là cạnh đầu tiên thuộc $Q$ và không nằm trong $V`$. Gọi Q_x là đường con từ s đến x của Q, khi đó :

$$L(Q_x) + L(xy) \leq L(Q)$$

- Vì $d(x)$ là độ dài đường đi ngắn nhất từ s đến x theo giả thuyết (1) : $d(x) \leq L(Q_x)$, ta có:

$$d(x) + L(xy) \leq L(Q_x)$$

- Vì y liền kề với x , $d(y)$ sẽ được chấp nhận, vì vậy :

$$d(y) \leq d(x) + L(xy)$$

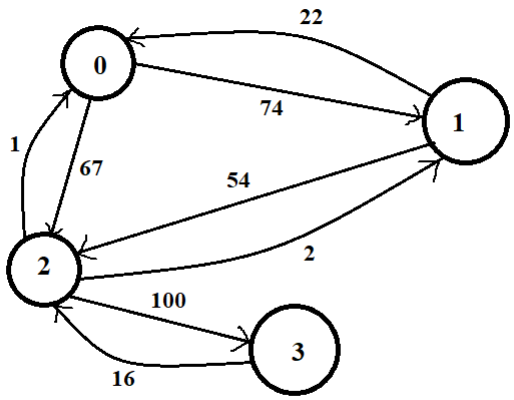
- Cuối cùng, do u đã được chọn từ đầu nên $d(u)$ phải là khoảng cách nhỏ nhất:

$$d(u) \leq d(y)$$

- Kết hợp tất cả các bất đẳng thức này theo hướng từ dưới lên, chúng ta sẽ bị mâu thuẫn rằng $d(x) \leq d(x)$. Do đó không tồn tại đường đi ngắn hơn Q và chính thì thế $d(u) = \delta(u)$, thuật toán được chứng minh.

2.6.2 Cách phát sinh Input/Output cho test case: Tương tự với cách phát sinh input/output của thuật toán Floyd-Warshall.

2.6.3 Kiểm tra thử với trường hợp cơ bản: Ta có đồ thị :



Sau khi chuyển đổi qua ma trận bằng cách :

- Tạo ma trận G có kích thước bằng VxV với V là số đỉnh của đồ thị.
- Lần lượt xét từng đỉnh:
  - Gán $G[v, v] = 0$.
  - Nếu từ đỉnh đang xét v có cạnh hướng tới đỉnh u , với trọng số cạnh là p thì $G[v, u] = p$.
  - Với các đỉnh u không có cạnh nối từ v tới u thì $G[v, u] = \text{inf}$

Ma trận được chuyển từ đồ thị :

```
graph
 [[  0.  74.  67.  inf]
 [ 22.   0.  54.  inf]
 [  1.   2.   0. 100.]
 [ inf  inf  16.   0.]]
```

Chạy thuật toán Dijkstra trên tất cả các đỉnh:

- Kết quả tay:
  - Từ đỉnh 0 đến :
    - 1 tốn 69
    - 2 tốn 67
    - 3 tốn 167
  - Từ đỉnh 1 đến :
    - 0 tốn 22
    - 2 tốn 54
    - 3 tốn 154
  - Từ đỉnh 2 đến :
    - 0 tốn 1
    - 1 tốn 2

S = 0	dist[x]			
	0	1	2	3
0	0	74	67	inf
0 2	-	69	67	167
0 2 1		69	-	167
0 2 1 3		-		167

S = 1	dist[x]			
	0	1	2	3
1	22	0	54	inf
1 0	22	-	54	inf
1 0 2	-		54	154
1 0 2 3			-	154

S = 2	dist[x]			
	0	1	2	3
2	1	2	0	100
2 0	1	2	-	100
2 0 1	-	2		100
2 0 1 3		-		100

S = 3	dist[x]			
	0	1	2	3
3	inf	inf	16	0
3 2	17	18	16	-
3 2 0	17	18	-	
3 2 0 1	-	18		

    - 3 tốn 100
  - Từ đỉnh 3 đến :
    - 0 tốn 17
    - 1 tốn 18
    - 2 tốn 16

- Kết quả chạy bằng code:

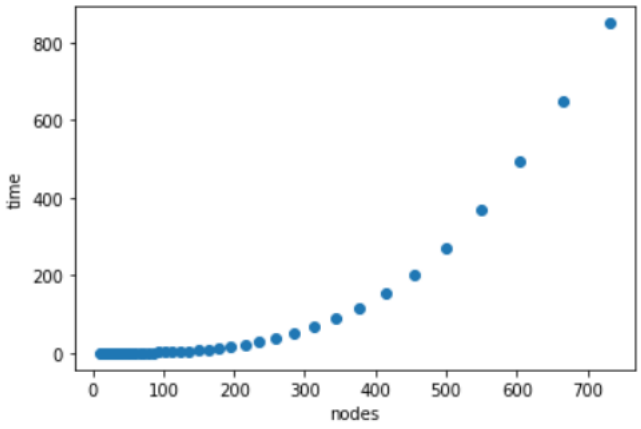
```
graph
 [[  0.  74.  67.  inf]
 [ 22.   0.  54.  inf]
 [  1.   2.   0. 100.]
 [ inf  inf  16.   0.]]


dist
 [[  0.  69.  67. 167.]
 [ 22.   0.  54. 154.]
 [  1.   2.   0. 100.]
 [ 17.  18.  16.   0.]]



khoảng cách từ đỉnh  0  đến đỉnh:
1 : 69.0
2 : 67.0
3 : 167.0
khoảng cách từ đỉnh  1  đến đỉnh:
0 : 22.0
2 : 54.0
3 : 154.0
khoảng cách từ đỉnh  2  đến đỉnh:
0 : 1.0
1 : 2.0
3 : 100.0
khoảng cách từ đỉnh  3  đến đỉnh:
0 : 17.0
1 : 18.0
2 : 16.0
```

- So sánh kết quả chạy tay và chạy code ta thấy cả 2 đều giống nhau.

2.7 Phân tích độ phức tạp bằng cài đặt:

2.7.1 a. Độ phức tạp thời gian:

- Cũng tương tự với thực nghiệm độ phức tạp của Floyd-Warshall, ta sẽ tạo bộ dữ liệu ma trận của đồ thị có hướng với số đỉnh ở mỗi lần lặp tăng 1%. Tập đỉnh thu được nằm trong khoảng $[11, 731]$.
- Sử dụng cProfile để tiến hành đo thời gian của thuật toán khi áp dụng cho bộ dữ liệu này, sau khi tiến hành thu thập ta được 50 bộ dữ liệu gồm 2 cột là số đỉnh và thời gian thực thi của thuật toán ứng với số đỉnh đó:



- Sau đó , ta đưa bộ dữ liệu vào Linear Regression để tiến hành dự đoán hệ số X của các hàm cơ bản

```
1  print(result)

5.66252309])], ['n**2', array([16.28963721])], ['n**3', array([0.02452095])], ['n**4', array([3.5073953e-05])], ['n**5', array([4.88955659e-08])],
```

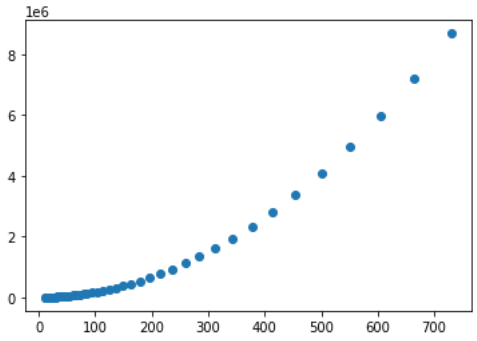
- Tiếp theo, ta đinh Mean Squard Error giữa các thời gian tính được và thời gian thu được , hàm nào có chỉ số MSE nhỏ nhất thì khả năng đó chính là độ phức tạp của thuật toán

	f(n) = log(n,2)		f(n) = sqrt(n)		f(n) = n		f(n) = n*log(n,2)		f(n) = n**2		f(n) = n**3		f(n) = n**4		f(n) = n**5	
X	59.238		19.623		0.7992		0.0876		0.0014		2.20E-06		3.23E-09		4.59E-12	
MSE		**110611.3**		**32103.884**		**10062**		**7330.9**		**1389.5**		**4.094389801**		**721.9754124**		**2137.478289**
					min	4.0944		suy ra n**3 la do phuc tap can tim								

- Theo như kết quả ta thấy hàm n^3 có chỉ số MSE nhỏ nhất nên độ phức tạp của thuật toán này là $O(n^3)$

2.7.2 Độ phức tạp bộ nhớ :

- Ta sử dụng hàm $__sizeof__()$ để có thể đo bộ nhớ được lưu của các ma trận có trong thuật toán cụ thể là các ma trận: $graph, dist$ và $sptSet$. Sau đó tính tổng 3 giá trị này để tính bộ nhớ đã sử dụng cho mỗi lần chạy thuật toán.
- Thu thập dựa trên bộ dữ liệu 50 đỉnh như thực nghiệm thời gian ta thu được bộ dữ liệu gồm 2 cột là số đỉnh của đồ thì và bộ nhớ cần lưu trữ khi thực hiện thuật toán.



- Sau đó, ta đưa bộ dữ liệu vào Linear Regression để tiến hành dự đoán hệ số X của các hàm cơ bản.

```
print(result)
```

```
[array([791028.68038671]), array([251128.23442502]), array([251128.23442502]), array([9836.42417798]), array([1066.66252309]), array([16.
28963721]), array([0.02452095]), array([3.5073953e-05]), array([4.88955659e-08]), array([-8.75779164e-13])]
```

- Tính chỉ số MSE như làm với thực nghiệm thời gian để tìm ra hàm phù hợp nhất.

	log(n)	sqrt(n)	n	nlog(n)	n^2	n^3	n^4	n^5
X	791028.7	251128.2	9836.424	1.07E+03	16.28964	0.024521	3.51E-05	4.89E-08
MSE	1.75E+13	3.78E+12	6.63E+11	3.72E+11	2.96E+08	2.12E+11	5.61E+11	9.05E+11
			Min	2.96E+08	suy ra	n^2 la do phuc tap can tim		

- Có thể thấy kết quả của độ phức tạp bộ nhớ là $O(n^2)$

3 Xử lí đa đồ thị:

Ta có đa đồ thị :
Đa đồ thị sau khi chuyển về ma trận sẽ có dạng ma trận 3 chiều: $n * n * m$ (với n là số đỉnh, m là tổng số đường đi lớn nhất giữa 2 đỉnh trong cả đồ thị).
Chuyển đa đồ thị trên về ma trận ta được ma trận 3 chiều D với kích thước $4 * 4 * 2$ như sau:

[[[0, 17, 6, inf],
[inf, 0, 3, 3],
[inf, 7, 0, 8],
[inf, inf, inf, 0]],

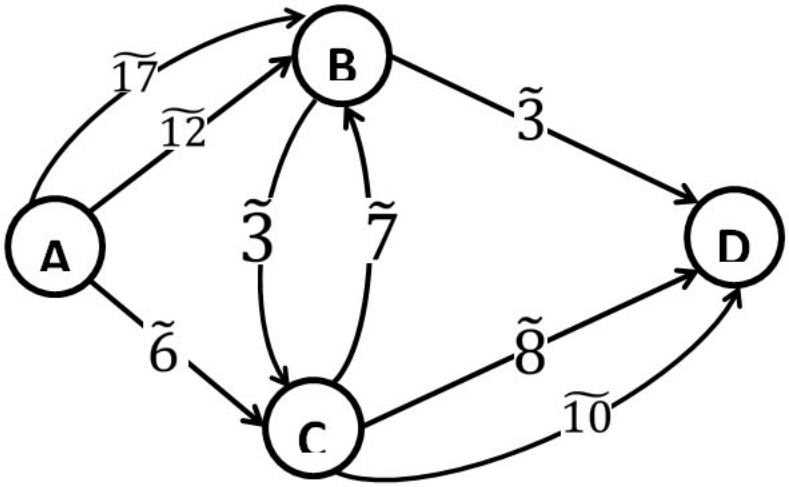


Figure 6 : A directed multigraph G with i-v fuzzy weighted arcs.

	A	B	C	D
A	0	17	6	inf
B	inf	0	3	3
C	inf	7	0	8
D	inf	inf	inf	0

+

	A	B	C	D
A	0	12	inf	inf
B	inf	0	inf	inf
C	inf	inf	0	10
D	inf	inf	inf	0

[[0, 12, inf, inf],
[inf, 0, inf, inf],
[inf, inf, 0, 10],
[inf, inf, inf, 0]]]

Mảng cost sẽ có kích thước $n * n$ (với n là số đỉnh) và có giá trị tại mỗi phần từ là min của tất cả đường đi giữa mỗi cặp đỉnh.

[[min(0, 0), min(17, 12), min(6, inf), min(inf,inf)].
[min(inf, inf), min(0, 0), min(3, inf), min(3, inf)],
[min(inf, inf), min(7, inf), min(0, 0), min(8, 10)],
[min(inf, inf), min(inf, inf), min(inf, inf), min(0, 0)]]

Suy ra mảng cost là:

[[0, 12, 6, inf],
[inf, 0, 3, 3],
[inf, 7, 0, 8],
[inf, inf, inf, 0]]

4 So sánh các thuật toán:

	Floyd-Warshall	Dijkstra
Độ phức tạp thời gian	$O(V^3)$	$O(V^3)$
Độ phức tạp bộ nhớ	$O(V^2)$	$O(V^2)$
Chạy được với trọng số âm	Có	Không

- Ta có thế thấy Floyd-Warshall và Dijkstra gần như giống nhau, tuy nhiên so với Dijkstra thì Floyd-Warshall dễ hiểu và dễ cài đặt hơn thuận lợi cho các bài toán lớn cần tính đường đi của tất cả các cặp nút với nhau. Nhưng khi bạn chỉ cần tính đường đi giữa 1 cặp duy nhất thì việc sử dụng Dijkstra sẽ nhanh chóng hơn.
- Sự khác biệt lớn nhất ta có thể thấy giữa 2 thuật toán là Floyd-Warshall có thể chạy được cạnh có trọng số âm. Việc trọng số âm được ứng dụng rất nhiều trong đời sống hiện nay nên thuật toán Floyd-Warshall khá phổ biến ở thời điểm hiện tại. Tuy nhiên, bản nâng cấp hơn của Dijkstra là Bellman-Ford đã khắc phục được nhược điểm này của Dijkstra nguyên bản và Dijkstra cũng có nhiều biến thể giúp giảm thời gian thực thi của thuật toán cũng như là bộ nhớ lưu trữ.