**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ ĐÔNG Á**



**BÀI TẬP LỚN**

**HỌC PHẦN:TOÁN RỜI RẠC**

**TÊN BÀI TẬP LỚN:LÝ THUYẾT ĐỒ THỊ**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sinh viên thực hiện** | **Khóa** | **Lớp** | **Mã sinh viên** |
| **Lê Quý Mùi** | **K12** | **DCCNTTK12.10.4** | **20211133** |
| **Nguyễn Văn Lâm** | **K12** | **DCCNTTK12.10.4** | **20211166** |
| **Nguyễn Sơn Dương** | **K12** | **DCCNTTK12.10.4** | **20211189** |
| **Nguyễn Hồng Nam** | **K12** | **DCCNTTK12.10.4** | **20210932** |
| **Vi Văn Tuấn** | **K12** | **DCCNTTK12.10.4** | **20211138** |

**Bắc Ninh, 11 tháng 7 năm 2022**

**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ ĐÔNG Á**

**(**trang phụ bìa số 2, bìa mềm)

**BÀI TẬP LỚN**

**HỌC PHẦN: TOÁN RỜI RẠC**

**Nhóm: 6**

**TÊN BÀI TẬP LỚN: LÝ THUYẾT ĐỒ THỊ**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **STT** | **Sinh viên thực hiện** | **Điểm bằng số** | **Điểm bằng chữ** | **Ký tên SV** |
| **1** | **Lê Lê Quý Mùi** |  |  |  |
| **2** | **Nguyễn Văn Lâm** |  |  |  |
| **3** | **Nguyễn Sơn Dương** |  |  |  |
| **4** | **Nguyễn Hồng Nam** |  |  |  |
| **5** | **Vi Văn Tuấn** |  |  |  |
|  |  |  |  |  |

|  |  |
| --- | --- |
| **CÁN BỘ CHẤM 1**  *(Ký và ghi rõ họ tên)* | **CÁN BỘ CHẤM 2**  *(Ký và ghi rõ họ tên)* |

**Bắc Ninh,11 tháng 7 Năm 2022**

**MỤC LỤC**

[PHẦN I. CƠ SỞ LÝ THUYẾT 5](#_Toc108130008)

[1. Khái niệm đồ thị 6](#_Toc108130009)

[Khái niệm chung: 6](#_Toc108130010)

[1.1. Khái niệm cơ bản về về đồ thị 6](#_Toc108130011)

[1.2. Đỉnh: 6](#_Toc108130012)

[1.3. Cạnh: 6](#_Toc108130013)

[1.4. Cạnh có hướng (cung): 6](#_Toc108130014)

[1.5. Cạnh vô hướng: 7](#_Toc108130015)

[1.6. Khuyên: 7](#_Toc108130016)

[1.7. Hai cạnh song song: 7](#_Toc108130017)

[1.8. Đồ thị có hướng: 7](#_Toc108130018)

[1.9. Đồ thị vô hướng: 7](#_Toc108130019)

[1.10. Đơn đồ thị: 7](#_Toc108130020)

[1.11. Đa đồ thị: 7](#_Toc108130021)

[1.12. Bậc: 7](#_Toc108130022)

[1.13. Ta có định lí: 7](#_Toc108130023)

[Hệ quả 7](#_Toc108130024)

[1.1. Đường đi và chu trình: 7](#_Toc108130025)

[1.2. Liên thông: 8](#_Toc108130026)

[1.3. Đồ thị phẳng 8](#_Toc108130027)

[1.4. Công thức Euler: 8](#_Toc108130028)

[1.5. biểu diễn đồ thị: 8](#_Toc108130029)

[2. Biểu diễn đồ thị trên máy tính: 8](#_Toc108130030)

[1.1. Ma trận kề: 9](#_Toc108130031)

[1.2. Ưu điểm: 9](#_Toc108130032)

[1.3. Nhược điểm: 9](#_Toc108130033)

[1.4. Danh sách cạnh: 10](#_Toc108130034)

[1.5. Ưu điểm: 10](#_Toc108130035)

[1.6. Nhược điểm: 10](#_Toc108130036)

[1.7. Danh sách kề: 11](#_Toc108130037)

[1.8. Ưu điểm: 11](#_Toc108130038)

[1.9. Nhược điểm: 11](#_Toc108130039)

[1.10. Ưu điểm của danh sách kề: 12](#_Toc108130040)

[3. Duyệt đồ thị FDS,BFS 13](#_Toc108130041)

[1.1. Giới thiệu 13](#_Toc108130042)

[1.2. Thuật toán duyệt đồ thị ưu tiên chiều rộng (Breadth-first search - BFS) 13](#_Toc108130043)

[1.3. Thuật toán duyệt theo chiều sâu (Deep-First Search-DFS) 13](#_Toc108130044)

[1. Thuật toán tìm đường đi ngắn nhất 13](#_Toc108130045)

[1.4. Giới thiệu 13](#_Toc108130046)

[1.5. Thuật toán Floyd 14](#_Toc108130047)

[1.6. Thuật toán Dijkstra 14](#_Toc108130048)

[1.7. Thuật toán Bellman - Ford 14](#_Toc108130049)

[1.8. Ý tưởng của thuật toán. 15](#_Toc108130050)

[1.9. Thuật toán Floyd-Warshall 15](#_Toc108130051)

[1.10. Ý tưởng của thuật toán. 15](#_Toc108130052)

[4. Thuật toán tim cây khung nhỏ nhất 15](#_Toc108130053)

[1.1. Thuật toán Kruskal 15](#_Toc108130054)

[1.2. Thuật toán Prim 16](#_Toc108130055)

[PHẦN II. CÀI ĐẶT THUẬT TOÁN 16](#_Toc108130056)

1. CƠ SỞ LÝ THUYẾT
   1. Khái niệm đồ thị

Khái niệm chung:Trong toán học và tin học, **đồ thị** là đối tượng nghiên cứu cơ bản của lý thuyết đồ thị. Một cách không chính thức, đồ thị là một tập các đối tượng gọi là **đỉnh** nối với nhau bởi các **cạnh**. Thông thường, đồ thị được vẽ dưới dạng một tập các điểm (đỉnh, nút) nối với nhau bởi các đoạn thẳng (cạnh). Tùy theo ứng dụng mà một số cạnh có thể có hướng.

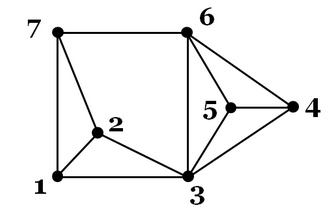
* + - 1. Khái niệm cơ bản về về đồ thị

Một cách không chính thức, đồ thị là một tập các đối tượng được gọi là các đỉnh nối với nhau bởi các cạnh.

Một đồ thị kí hiệu là: 🡺G=(V,E)

V là tập các đỉnh của đồ thị. Đặt ∣V∣=n∣V∣=n (số đỉnh).

E là tập các cạnh của đồ thị. Đặt ∣E∣=m∣E∣=m (số cạnh).



* + - 1. Đỉnh:

Đỉnh biểu diễn các đối tượng trong đồ thị, thường được đánh dấu bằng các số hoặc kí hiệu bằng các chữ cái in thường u,v,…

* + - 1. Cạnh:

Cạnh nối đỉnh x với đỉnh y là một tập gồm hai phần tử x,yx,y, thường được vẽ dưới dạng một đoạn thẳng nối hai đỉnh.

* + - 1. Cạnh có hướng (cung):

Là một cặp đỉnh có thứ tự. Trong mỗi cặp có thứ tự đó, đỉnh thứ nhất được gọi là đỉnh đầu, đỉnh thứ hai là đỉnh cuối.

* + - 1. Cạnh vô hướng:

Không quan tâm đến hướng và coi hai đỉnh như nhau.

* + - 1. Khuyên:

Là một cạnh nối một đỉnh với chính nó.

* + - 1. Hai cạnh song song:

Là hai cạnh cùng nối hai đỉnh u, v.

* + - 1. Đồ thị có hướng:

Là đồ thị mà tất cả các cạnh trong đồ thị đều có hướng.

* + - 1. Đồ thị vô hướng:

Là đồ thị mà tất cả các cạnh trong đồ thị đều vô hướng.

* + - 1. Đơn đồ thị:

Là đồ thị không có khuyên và không có cạnh song song.

* + - 1. Đa đồ thị:

Là đồ thị không phải là đơn đồ thị.

* + - 1. Bậc:

Trong đồ thị vô hướng, bậc của đỉnh v trong đồ thị G, ký hiệu dG(u)dG(u), là số cạnh liên thuộc với v, trong đó, khuyên được tính hai lần.

* + - 1. Ta có định lí:

Giả sử G=(V,E)G=(V,E) là đồ thị vô hướng, khi đó tổng các bậc đỉnh trong V sẽ bằng 2 lần số cạnh.

∑v∈VdG(v)=m∗2∑v∈VdG(v)=m∗2

Hệ quả: Trong đồ thị vô hướng, số đỉnh bậc lẻ là chẵn.

Trong đồ thị có hướng, ta định nghĩa **bán bậc ra** của u là số cung đi ra khỏi nó, kí hiệu d+G(u)dG+(u), **bán bậc vào** của u là số cung đi ra khỏi nó, kí hiệu d−G(u)dG−(u).

Giả sử G=(V,E)G=(V,E) là đồ thị có hướng, khi đó tổng các bán bậc vào bằng tổng các bán bậc ra và bằng số cung của đồ thị.

∑v∈Vd+G(v)=∑v∈Vd+G(v)=m∑v∈VdG+(v)=∑v∈VdG+(v)=m

* + - 1. Đường đi và chu trình:

Một dãy các đỉnh P = (p0,p1,…,pk) sao cho (Pi-1, Pi) ∊ E, ∀i: 1<=i<=k được gọi là một đường đi.

Một đường đi là chu trình khi p0 = pk.

* + - 1. Liên thông:

Một đồ thị vô hướng là liên thông nếu tồn tại đường đi giữa hai cặp đỉnh bất kì thuộc đồ thị.

Một đồ thị có hướng là liên thông nếu phiên bản vô hướng của đồ thị đó là liên thông.

Đồ thị phẳng:

Đồ thị phẳng là đồ thị có thể được vẽ trên mặt phẳng sao cho không có hai đỉnh nào trùng nhau và các cạnh nào trùng nhau hoặc cắt nhau.

* + - 1. Đồ thị phẳng

Gọi G là đồ thị phẳng nếu có thể biểu diễn G bằng một biểu đồ trong mặt phẳng sao cho các cạnh đôi một không cắt nhau. Các cạnh của đồ thị phẳng G chia mặt phẳng thành nhiều miền, mỗi miền gọi Là một mặt của G (trong số các mặt, luôn luôn có một và chỉ một mặt vô hạn).Chiều dài của chu trình ngắn nhất trong G gọi là đaicủa G, ký hiệu

là g; trường hợp G không có chu trình thì ta đặt g bằng số cạnh của G.Hiển nhiên rằng đồ thị con của đồ thị phẳng là đồ thị phẳng, do đó một đồ thị có đồ thị con không phẳng là đồ thị không phẳng

* + - 1. Công thức Euler:

Giả sử đồ thị vô hướng liên thông G=(V,E)G=(V,E) là đồ thị phẳng, với n đỉnh và m cạnh chia mặt phẳng thành e phần thì

n−m+e=2n−m+e=2

* + - 1. biểu diễn đồ thị:

- Biểu diễn hình học. Đồ thị thường được biểu diễn bằng hình vẽ, gọi là biểu đồ, hoặc hình biểu diễn bằng cách: Mỗi đỉnh biểu diễn thành một điểm và mỗi cạnh biểu diễn thành một cung nối hai đỉnht ương ứng với nó mà trên cung đó ghi trọng số nếu có và với biểu đồ của đồ thị có hướng thì mỗi cạnh có kèm một mũi tên hướng từ gốc đến ngọn.

* 1. Biểu diễn đồ thị trên máy tính:

Có nhiều cách để biểu diễn đồ thị trên máy tính, tùy thuộc vào tính chất của đồ thị hoặc thuật toán áp dụng với đồ thị… Ta cũng có thể lưu kèm theo các thông tin như trọng số, giá trị phù hợp với từng cạnh. Dưới đây là một vài cách phổ biến.

Ma trận kề:

Tạo một ma trận A kích thước n\*n trong đó n là số đỉnh của đồ thị. Ta gán a[u][v] = 0 nếu có cạnh cạnh nối hai đỉnh u, v.

Nếu đồ thị là đa đồ thị, ta có thể gán a[u][v] = số cạnh nối u và v.

Định nghĩa và gán tùy theo lập trình viên hiểu là vô hướng hay có hướng, đơn đồ thị hay đa đồ thị.

* + - 1. Ưu điểm:

Để kiểm tra hai đỉnh u, v có kề nhau không, ta chỉ cần kiểm tra trong độ phức tạp O(1)O(1).

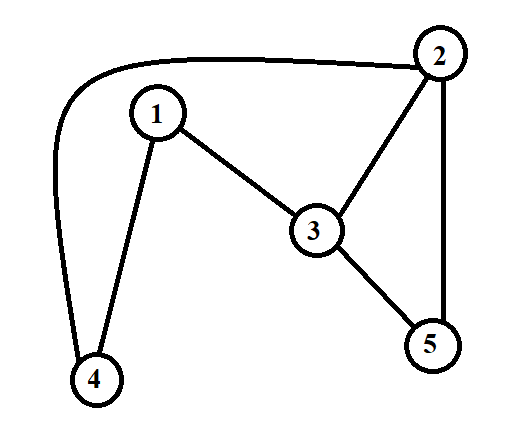
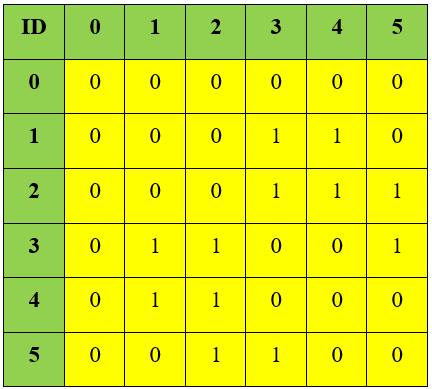
* + - 1. Nhược điểm:

Dù đồ thị có nhiều cạnh hay ít cạnh thì cũng phải mất n\*n ô nhớ để lưu.

Để duyệt tất cả các đỉnh kề với u, ta phải duyệt tất cả các đỉnh v ∊ V cho dù đỉnh u kề với ít hoặc không kề với đỉnh nào khác.

Biểu diễn bằng ma trận kề thường được dùng khi đồ thị có ít đỉnh, hoặc đồ thị dày, nhiều cạnh, hoặc thuật toán để thao tác trên đồ thị yêu cầu.

Ví dụ: Đồ thị G(V, E)G(V,E) dưới đây có 5 đỉnh, 6 cạnh:



Ma trận kề của nó sẽ có dạng như sau:

Ưu điểm của ma trận kề:

Đơn giản, dễ cài đặt.Để kiểm tra hai đỉnh uu và vv có kề nhau hay không, chỉ việc kiểm tra trong O(1)O(1) bằng phép so sánh a\_{u, v} \ne 0au,v​=0.

Nhược điểm của ma trận kề:

Luôn luôn tiêu tốn N^2N2 ô nhớ để lưu trữ ma trận kề, dù là trong trường hợp đồ thị ít cạnh hay nhiều cạnh.

Để xét một đỉnh uu kề với những đỉnh nào, buộc phải duyệt toàn bộ các đỉnh vv và kiểm tra điều kiện a\_{u, v} \ne 0au,v​=0. Như vậy kể cả đỉnh uu không kề với đỉnh nào, chúng ta vẫn phải duyệt mất O(N)O(N) để biết được điều đó.

Phù hợp khi nào: Trong các bài toán đồ thị có số lượng đỉnh ít (thường là không vượt quá 300300).

Danh sách cạnh:

Với đồ thị G=(V,E)G=(V,E) có n đỉnh, m cạnh, ta có thể liệt kê tất cả các cạnh của đồ thị bằng một danh sách tương ứng, mỗi phần tử của mảng tương ứng là một cặp (u,v) là một cạnh thuộc E, tùy theo người lập trình định nghĩa là có hướng hay vô hướng.

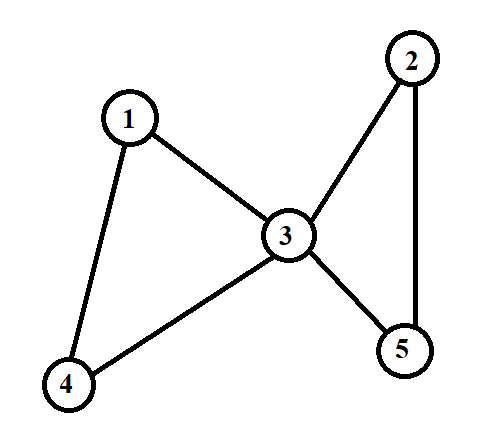
* + - 1. Ưu điểm:

Với đồ thị thưa, ta chỉ cần mất m (số lượng cạnh) ô nhớ để lưu đồ thị.

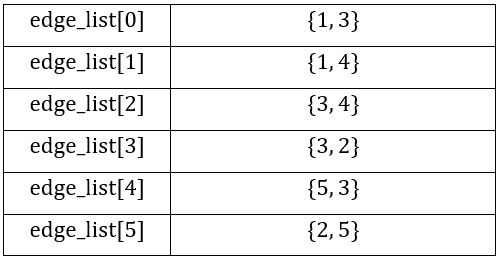
* + - 1. Nhược điểm:

Khi cần kiểm tra hai đỉnh u,v có kề nhau hay không, ta không thể kiểm tra nhanh trong //( O(1) //) như cách lưu bằng ma trận kề, mặc dù tùy theo cách lưu danh sách cạnh mà ta có thể kiểm tra trong //( O(logn) //) hoặc ít hơn.

Ví dụ: Đồ thị G(V, E)G(V,E) dưới đây 55 đỉnh, 66 cạnh theo thứ tự là: (1, 3), (1, 4), (3, 4), (3, 2), (5, 3), (2, 5)(1,3),(1,4),(3,4),(3,2),(5,3),(2,5):



Danh sách cạnh của nó được biểu diễn bằng một vector \text{edge\_list} như sau:



Ưu điểm của danh sách cạnh:

Trong trường hợp đồ thị ít cạnh, cách biểu diễn này sẽ giúp tiết kiệm không gian lưu trữ.

Ở một số trường hợp đặc biệt, ta phải xét tất cả các cạnh trên đồ thị thì phương pháp cài đặt này giúp việc duyệt cạnh dễ dàng hơn trong O(M)O(M) (ví dụ giải thuật tìm cây khung nhỏ nhất Kruskal).

Nhược điểm của danh sách cạnh:

Trong trường hợp cần duyệt các đỉnh kề với một đỉnh uu, bắt buộc phải duyệt qua mọi cạnh, lọc ra các cạnh có chứa đỉnh uu và xét đỉnh còn lại. Điều này sẽ tốn thời gian nếu đồ thị có nhiều cạnh.

Danh sách kề:

Với mỗi đỉnh của đồ thị, ta lưu một danh sách các đỉnh kề với đỉnh đó.

* + - 1. Ưu điểm:

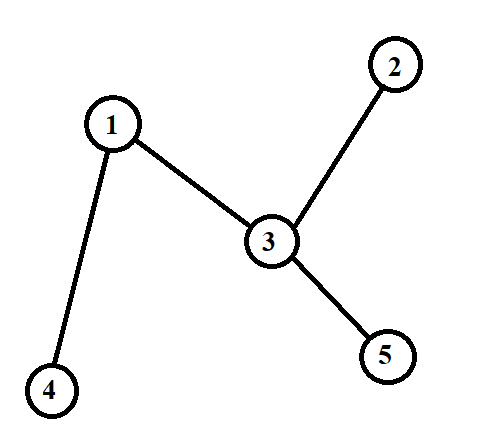
Với phương pháp này, việc duyệt tất cả các đỉnh kề với đỉnh u vô cùng dễ dàng.

* + - 1. Nhược điểm:

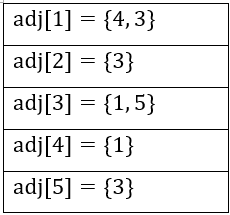
Khi cần kiểm tra hai đỉnh u,v có kề nhau hay không, ta không thể kiểm tra nhanh trong O(1)O(1) như cách lưu bằng ma trận kề, mặc dù tùy theo cách lưu danh sách cạnh mà ta có thể kiểm tra trong O(logn)O(logn) hoặc ít hơn.

Để khắc phục nhược điểm của ma trận kề và danh sách cạnh, người ta sử dụng danh sách kề (cũng là cách thường xuyên sử dụng nhất trong các bài toán đồ thị). Trong cách biểu diễn này, với mỗi đỉnh u của đồ thị, ta sẽ tạo ra một dánh sách adj\_uadju​ là các đỉnh kề với nó. Việc cài đặt adj\_uadju​ có thể thực hiện dễ dàng với vector.

Ví dụ: Đồ thị G(V, E)G(V,E) dưới đây gồm 55 đỉnh, 44 cạnh:



Danh sách kề của nó có thể biểu diễn bằng một mảng \text{adj}[6]adj[6] gồm các vector  mỗi vector \text{adj}[u]adj[u] lưu danh sách kề của đỉnh uu:



* + - 1. Ưu điểm của danh sách kề:

Duyệt đỉnh kề và các cạnh của đồ thị rất nhanh.

Tiết kiệm không gian lưu trữ, do vector là kiểu dữ liệu với bộ nhớ động, sẽ chỉ tạo ra các ô nhớ tương ứng với số lượng đỉnh kề.

Nhược điểm của danh sách kề: Khi cần kiểm tra (u, v)(u,v) có phải là một cạnh của đồ thị hay không thì bắt buộc phải duyệt toàn bộ danh sách kề của uu hoặc của vv.

Phù hợp khi nào: Hầu hết trong mọi bài toán đồ thị đều nên sử dụng, chỉ trừ các bài toán cần duyệt toàn bộ cạnh của đồ thị.

* 1. Duyệt đồ thị FDS,BFS

Giới thiệu

Khi giải quyết nhiều bài toán lý thuyết đồ thị, ta luôn phải duyệt qua tất cả các đỉnh của đồ thị đó. Cho nên, cần có thuật toán duyệt toàn bộ các đỉnh của đồ thị này. Gọi chung là thuật toán duyệt đồ thị. Trong đó có thuật toán duyệt theo chiều sâu và duyệt theo chiều rộng.

Thuật toán duyệt đồ thị ưu tiên chiều rộng (Breadth-first search - BFS)

 là một trong những thuật toán tìm kiếm cơ bản và thiết yếu trên đồ thị. Mà trong đó, những đỉnh nào gần đỉnh xuất phát hơn sẽ được duyệt trước.

Ứng dụng của BFS có thể giúp ta giải quyết tốt một số bài toán trong thời gian và không gian tối thiểu. Đặc biệt là bài toán tìm kiếm đường đi ngắn nhất từ một đỉnh gốc tới tất cả các đỉnh khác. Trong đồ thị không có trọng số hoặc tất cả trọng số bằng nhau, thuật toán sẽ luôn trả ra đường đi ngắn nhất có thể. Ngoài ra, thuật toán này còn được dùng để tìm các thành phần liên thông của đồ thị, hoặc kiểm tra đồ thị hai phía, …

Thuật toán duyệt theo chiều sâu (Deep-First Search-DFS)

:cho G = (V,E) là đồ thị có tập các đỉnh V và tập các cạnh E. v là một đỉnh trong V va u là đỉnh kề của v, sao cho u cũng thuộc V. Khi đó ta dán nhãn cho tất cả các đỉnh của đồ thị là 0. Chọn một đỉnh v thuộc tập V để bắt đầu duyệt. Gán nhãn đỉnh v này la 1-v đã được duyệt. Chọn đỉnh u trong tập V kề với đỉnh v mà nhãn là 0. Duyệt qua đỉnh u và gán nhãn u là 1. Tiếp tục quá trình duyệt đến khi tất cả các đỉnh đồ thị có nhãn là 1.

* 1. Thuật toán tìm đường đi ngắn nhất

Giới thiệu

Bài toán tìm đường đi ngắn nhất trên đồ thị là một trong những bài toán đa dạng, có nhiều ứng dụng thực tế (như trong Google Maps, hay các bài toán networking, …). Các dạng bài về tìm đường đi ngắn nhất cũng thường xuyên có mặt trong các kì thi lập trình.

Bài viết này sẽ giới thiệu ba thuật toán cơ bản của dạng bài tìm đường đi ngắn nhất:

•Thuật toán Bellman - Ford.

•Thuật toán Dijkstra.

•Thuật toán Floyd-Warshall (còn gọi là thuật toán Floyd).

Cần lưu ý rằng: [có một thuật toán thông dụng khác](https://vnoi.info/wiki/algo/basic/two-pointers.md#gi%E1%BA%A3i-ph%C3%A1p-3) cũng có tên thường gọi là thuật toán Floyd, dùng để tìm chu trình trong đồ thị có hướng. Bài viết này sẽ chỉ đề cập đến thuật toán tìm đường đi ngắn nhất.

Thuật toán Floyd

Thuật toán này cho phép chúng ta tìm đường đi ngắn nhất giữa mọi cặp đỉnh.

Nếu đỉnh k nằm trên đường đi ngắn nhất từ đỉnh i tới đỉnh j thì đoạn đường từ i tới k và từ k tới j phải là đường đi ngắn nhất từ i tới k và từ k tới j tương ứng. Do đó ta sử dụng ma trận A để lưu độ dài đường đi ngắn nhất giữa mọi cặp đỉnh.  
– Ban đầu ta đặt A[i,j] = C[i,j], tức là ban đầu A chứa độ dài đường đi trực tiếp (không đi qua đỉnh nào cả).  
– Sau đó thực hiện n lần lặp, sau lần lặp thứ k, ma trận A sẽ chứa độ dài đường đi ngắn nhất giữa mọi cặp đỉnh chỉ đi qua các đỉnh thuộc tập {1,2,..,k}. Như vậy, sau n lần lặp ta nhận được ma trận A chứa độ dài các đường đi ngắn nhất giữa mọi cặp đỉnh của đồ thị.

– Ký hiệu Ak là ma trận A sau lần lặp thứ k, tức là Ak[i,j] là độ dài đường đi ngắn nhất từ i đến j chỉ đi qua các đỉnh thuộc {1, 2,.., k}. Ak[i,j] được tính theo công thức như sau:    Ak[i,j] = min {Ak -1[i,j], Ak-1[i,k] + Ak-1[k,j]}.

– Trong quá trình lặp ta phải lưu lại vết đường đi, tức là đường đi ngắn nhất đi qua các đỉnh nào. Khi đó ta sử dụng mảng phụ P[nxn], trong đó P[i,j] lưu đỉnh k nếu đường đi ngắn nhất từ i đến j đi qua đỉnh k.  Ban đầu P[i,j]=0 với mọi i,j, vì lúc đó đường đi ngắn nhất là đường đi trực tiếp, không đi qua đỉnh nào cả.

Thuật toán Dijkstra

Về thuật toán Dijkstra có 2 loại là tìm đường đi ngắn nhất từ 1 đỉnh nguồn tới 1 đỉnh đích và tìm đường đi ngắn nhất từ 1 đỉnh nguồn tới các đỉnh còn lại của đồ thị, và ở đây mình sẽ nói về loại thứ 1. (loại thứ hai bạn có thể tìm trên mạng hoặc chỉ cần thay đổi dòng **while (s[b] == 0) (dòng 43 của code 1 & dòng 76 của code 2) thành vòng for duyệt từ 0 đến n-1 là sẽ tìm được tất cả các đỉnh).**

– Dùng 1 mảng Len[] – Len[i] là khoảng cách ngắn nhất từ đỉnh a tới đỉnh i.

– Dùng 1 mảng S đánh dấu các đỉnh i đặc biệt (các đỉnh i mà thời điểm hiện tại thì đường đi từ a tới i là ngắn nhất).

– Dùng mảng P[] đánh dấu đường đi. P[j] = i nếu i là đỉnh đi trước j trong đường đi ngắn nhất.

– Đặt lại giá trị vô cùng cho các cặp đỉnh không có đường đi.

– Khởi tạo tất cả các đường đi từ a đên các đỉnh khác bằng vô cùng.

– Khởi tạo đường đi từ a đến chính a = 0.

– Duyệt hết các đỉnh V của đồ thị

+ Tìm đỉnh i chưa nằm trong S mà đường đi từ a tới i là ngắn nhất để đưa vào S. Nếu không tìm được đỉnh nào nghĩa là đã duyệt hết các đỉnh có thể đi mà vẫn chưa thấy đỉnh đích => không thể đi được.

+ Nếu tìm được đỉnh i thì duyệt tất cả các đỉnh j chưa nằm trong S. Nếu Len[i] + G[i][j] < Len[j] (trong đó G[i][j] là khoảng cách từ đỉnh i tới đỉnh j) thì gán Len[j] = Len[i] + G[i][j]; và đánh dấu đường đi P[j] = i.

Thuật toán Bellman - Ford

Thuật toán Bellman-Ford dùng để giải quyết bài toán **đường đi ngắn nhất một nguồn** (Single-source shortest path), đồ thị **có thể có trọng số âm**.

* + - 1. Ý tưởng của thuật toán.

Xét trường hợp đơn giản hơn, khi đồ thị không có trọng số âm (tức đường đi ngắn nhất luôn tồn tại).

Thuật toán Bellman-Ford sẽ lặp nhiều lần. Ở mỗi vòng lặp, ta sẽ đi qua **tất cả** các cạnh (u,v)(u,v) trên đồ thị, so sánh đường đi S→vS→v đã tìm được với đường đi S→u→v

Thuật toán Floyd-Warshall

Thuật toán Floyd-Warshll dùng để giải quyết bài toán **đường đi ngắn nhất mọi cặp đỉnh** (All-pairs shortest path), đồ thị **có thể có trọng số âm**.

* + - 1. Ý tưởng của thuật toán.

Ý tưởng của thuật toán này là: "Liệu chúng ta có thể chèn một đỉnh kk vào đường đi ngắn nhất giữa 2 đỉnh uu và vv?".

Ví dụ như có một đường đi ngắn nhất từ 0→40→4 như sau: 0→1→2→3→40→1→2→3→4. Vậy việc tính đường đi ngắn nhất từ 0→40→4 hoàn toàn có thể được chia thành tính đường đi ngắn nhất từ 0→20→2 sau đó cộng với đường đi ngắn nhất từ 2→42→4. Tương tự thế đường đi ngắn nhất từ 0→20→2 và 2→42→4 lại tiếp tực được phân hoạch thành những đường đi ngắn nhất khác đơn giản hơn và tối ưu hơn.

Ta nhận thấy có một cấu trúc đệ quy, chia nhỏ bài toán ở đây. Ý tưởng này cho phép chúng ta thực hiện một thuật toán mang hương vị quy hoạch động như sau:

Gọi D(u,v,k)D(u,v,k) là đường đi ngắn nhất, trong đó ta chỉ được đi qua kk đỉnh đầu tiên (có số thứ tự từ 00 đến k−1k−1), ngoại trừ chính uu và vv. Ta có công thức truy hồi:

D(u,v,0)=Wu,vD(u,v,0)=Wu,v (không được dùng đỉnh nào ngoài chính u,vu,v).

D(u,u,k)=0D(u,u,k)=0

D(u,v,k)=D(u,v,k)= min của 2 trường hợp:

D(u,v,k−1)D(u,v,k−1): ta không dùng đỉnh kk làm trung gian, giữ nguyên đường đi cũ.

D(u,k,k−1)+D(k,v,k−1)D(u,k,k−1)+D(k,v,k−1): ta dùng đỉnh kk làm trung gian, từ đường đi u→vu→v thành đường đi u→k→vu→k→v.

Đến đây ta có thể sử dụng trực tiếp công thức quy hoạch động để cài đặt thuật toán. Tuy nhiên, để đảm bảo bộ nhớ, ta có thể tính các D(u,v,k)D(u,v,k) với kk lần lượt từ 11 đến NN, và khi cài đặt chỉ cần lưu lại D(u,v)D(u,v).

* 1. Thuật toán tim cây khung nhỏ nhất

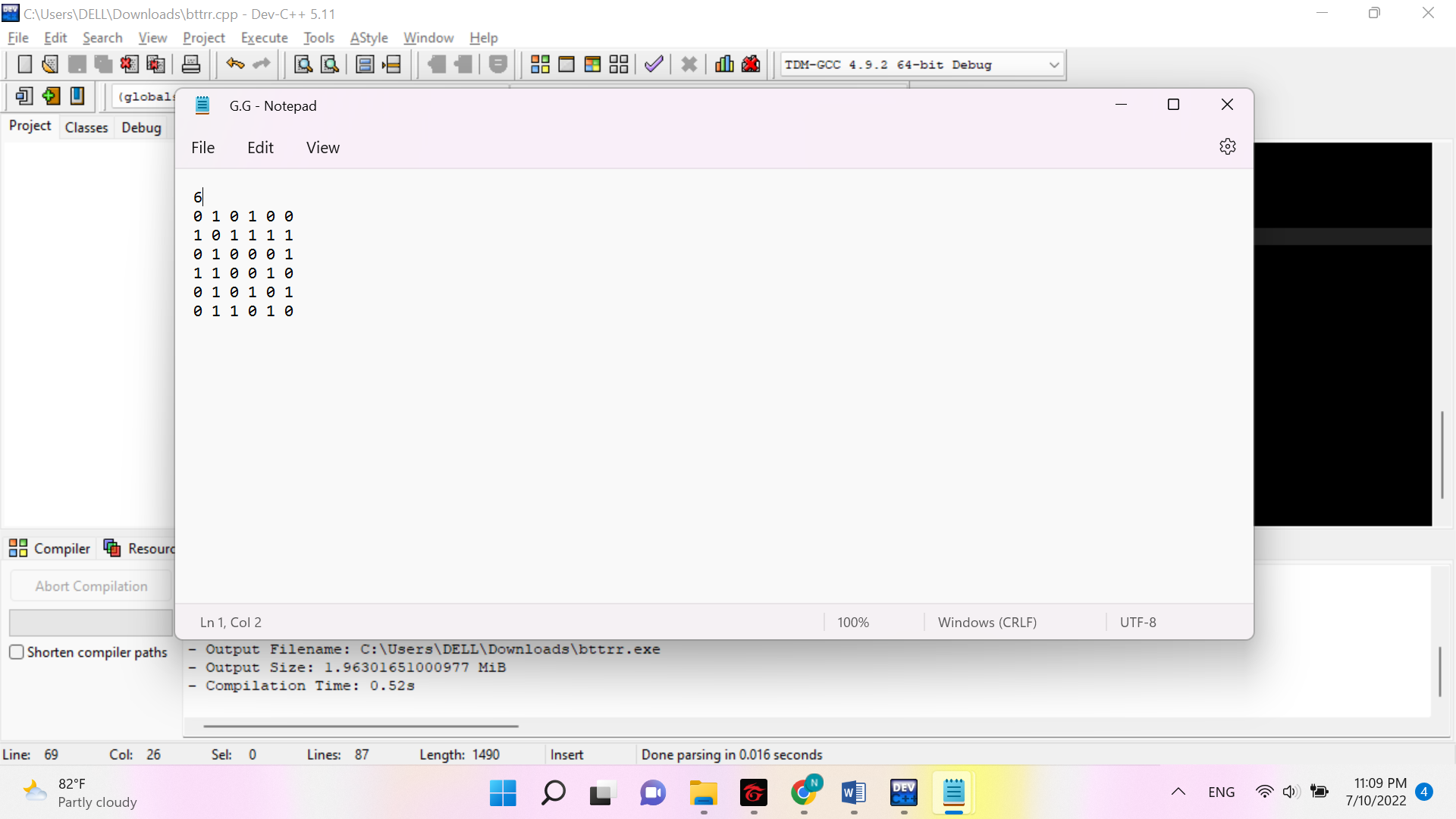
Thuật toán Kruskal

Trong quá trình thực hiện thuật toán, ở mỗi bước, ta có thể nhanh chóng chọn đỉnh và cạnh cần bổ sung vào cây khung, các đỉnh của đồ thị được sẽ được gán các nhãn. Nhãn của một đỉnh v gồm hai phần, [d[v], near[v]]. Trong đó, phần thứ nhất d[v] dùng để ghi nhận độ dài cạnh nhỏ nhất trong số các cạnh nối đỉnh v với các đỉnh của cây khung đang xây dựng. Phần thứ hai, near[v] ghi nhận đỉnh của cây khung gần v nhất.

Thuật toán Prim

Trong thuật toán này, bắt đầu tại một đỉnh tuỳ ý s của đồ thị, nối s với đỉnh y sao cho trọng số cạnh c[s, y] là nhỏ nhất. Tiếp theo, từ đỉnh s hoặc y tìm cạnh có độ dài nhỏ nhất, điều này dẫn đến đỉnh thứ ba z và ta thu được cây bộ phận gồm 3 đỉnh 2 cạnh. Quá trình được tiếp tục cho tới khi ta nhận được cây gồm n-1 cạnh, đó chính là cây bao trùm nhỏ nhất cần tìm.

1. CÀI ĐẶT THUẬT TOÁN



File notepad để lưu ma trận kề của trọng số G để tí gán vào code để chạy chương trình

1:Bài Code duyệt đồ thị:

#include <iostream>

#include <stdio.h>

#include <conio.h>

#define max 100

#define FileIn "F:\\G.txt"

using namespace std;

int chuaXet[max];

// A: ma tran ke cua G, n: so dinh

int A[max][max],n;

// doc file chua do thi G luu vao ma tran A

void Doc\_File(int A[max][max], int &n) {

FILE \*f = fopen(FileIn,"rb");

fscanf(f,"%d",&n);

cout<<"\n So dinh: "<<n<<"\n Ma tran ke: "<<endl;

for(int i =0;i<n;i++) {

for(int j =0;j<n;j++) {

fscanf(f,"%d",&A[i][j]);

cout<<A[i][j]<<" ";

}

cout<<endl;

}

fclose(f);

}

// Khoi tao chua xet

void KhoiTao\_ChuaXet(){

for (int i=0;i<max;i++)

chuaXet[i]=1;

}

// thuat toan DFS

void DFS(int u){

// xet dinh u

chuaXet[u]=0;

cout<<u<<"->";

for(int v=0;v<n;v++)

if(chuaXet[v]==1&&A[u][v]==1)

{

DFS(v);

}

}

// thuat toan BFS

void BFS(int u){

int queue[max], dau=0,cuoi=0;

for(int i=0;i<max;i++) queue[i]=0;

queue[cuoi]=u;

chuaXet[u]=0;

cout<<u<<"->";

while(dau>=cuoi)

{

int p=queue[cuoi];

cuoi++;

for(int v=0;v<n;v++)

if(chuaXet[v]==1&&A[p][v]==1)

{

dau++;

queue[dau]=v;

chuaXet[v] =0;

cout<<v<<"->";

}

}

}

// Kiem tra chuaXet

int KT\_ChuaXet(){

for (int i=0;i<n;i++)

if (chuaXet[i]==1) return i;

return -1;

}

// tim bac cac dinh

int Deg(int i){

int deg=0;

for(int j=0;j<n;j++)

{

deg +=A[i][j];

}

return deg;

}

// ham chinh

int main() {

// doc ma tran

Doc\_File(A,n);

// Duyet do thi DFS

KhoiTao\_ChuaXet();

cout<<"\n Duyet do thi DFS: ";

DFS(0);

// Duyet do thi BFS

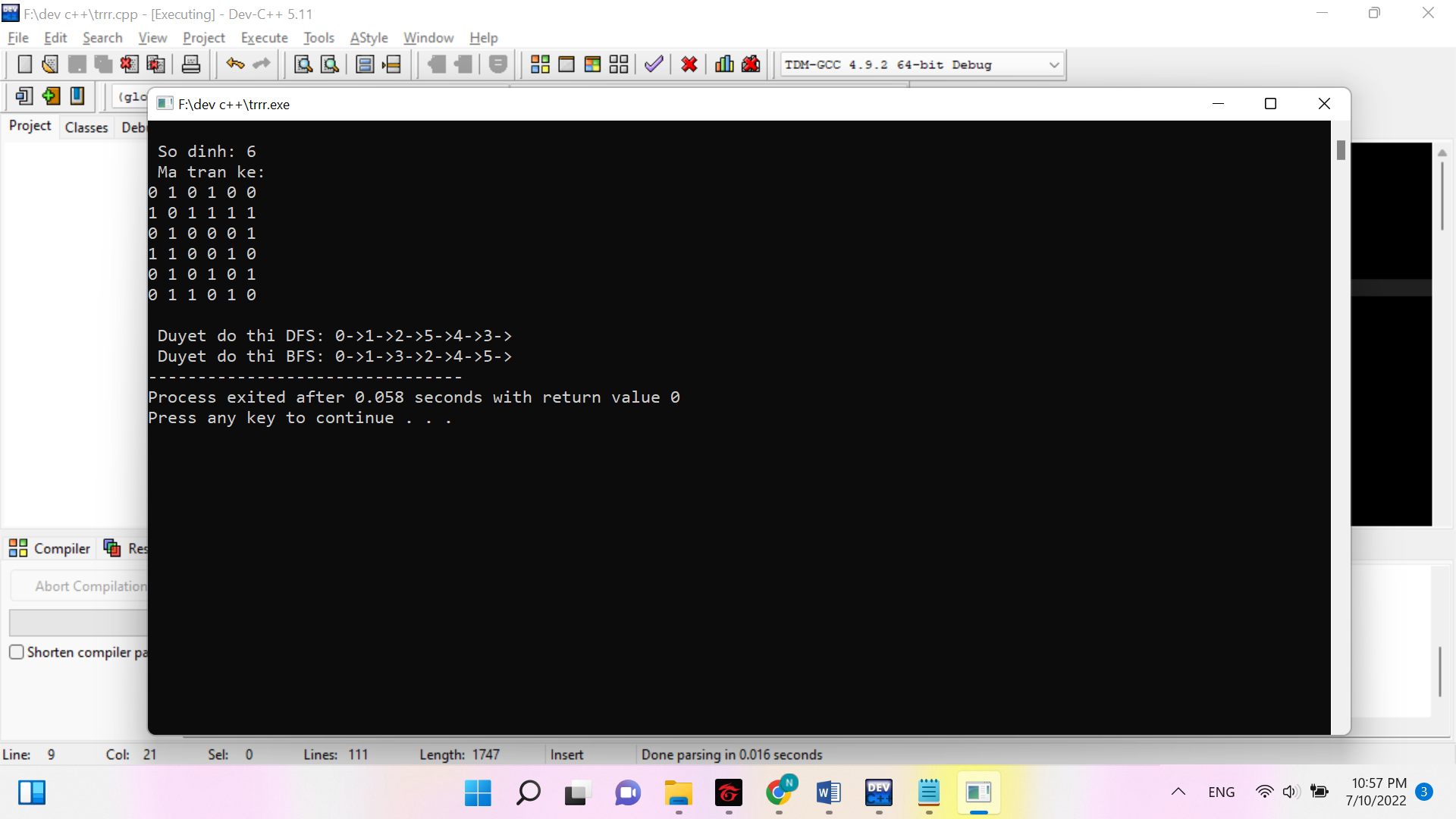
KhoiTao\_ChuaXet();

cout<<"\n Duyet do thi BFS: ";

BFS(0);

return 0;

}



Kết quả duyệt đồ thị DFS,BFS

Bttrr.cpp:

#define maxn 1000

#define maxf 1000000000

#include <iostream>

#include <fstream>

using namespace std;

int a[maxn][maxn],n,m,s,e;

int f[maxn],trace[maxn],checkt2[maxn];

void readf(){

fstream inp("input.txt");

inp>>n>>m;

inp>>s>>e;

for(int i=1;i<=n;i++)

for(int j=1;j<=n;j++) a[i][j]=-1;

for(int i=1;i<=m;i++){

int u,v;

inp>>u>>v;

inp>>a[u][v];

a[v][u]=a[u][v];

}

inp.close();

}

void dijkstra(){

for(int i=1;i<=n;i++){

f[i]=maxf;

checkt2[i]=true;

trace[i]=0;

}

f[s]=0;

trace[s]=0;

int v=s,fmin;

while(v!=e){

fmin=maxf;

for(int i=1;i<=n;i++)

if(checkt2[i] && fmin>f[i]){

fmin=f[i];

v=i;

}

if(fmin==maxf) break;

checkt2[v]=false;

for(int i=1;i<=n;i++)

if(a[v][i]>0 && f[i]>f[v]+a[v][i]){

f[i]=f[v]+a[v][i];

trace[i]=v;

}

}

}

void output(){

if(f[e]==maxf) cout<<"NO PATH";

else{

cout<<f[e]<<"\n";

int path[maxn],d=0;

d++;

path[d]=e;

while(trace[e]!=0){

e=trace[e];

d++;

path[d]=e;

}

for(int i=d;i>0;i--)

cout<<path[i]<<" ";

}

cout<<f[e];

}

int main(){

readf();

dijkstra();

output();

}

Input.txt:

6 7

1 4

1 2 4

1 6 2

2 3 2

3 4 20

3 6 3

4 5 5

5 6 1

