TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN TOÁN ỨNG DỤNG VÀ TIN HỌC

o0o

BÁO CÁO ĐỒ ÁN I

LÝ THUYẾT ĐỒ THỊ VÀ ỨNG DỤNG TRONG MẠNG LƯỚI GIAO THÔNG

Học phần: **Đồ án I**

Giảng viên hướng dẫn: **TS. ĐOÀN DUY TRUNG**

Sinh viên: **Nguyễn Công Hiếu - 20195016**

Lớp: **Toán tin 02 - khóa 64**

HÀ NỘI, 06/2022

NHẬN XÉT CỦA GIẢNG VIÊN HƯỚNG DẪN

....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....

Hà Nội, ngày ... tháng ... năm 2022

Giảng viên hướng dẫn

Lời nói đầu

Trong toán học, lý thuyết đồ thị là một lĩnh vực nghiên cứu về đồ thị - một cấu trúc được sử dụng để mô hình hóa mối quan hệ giữa từng cặp đối tượng. Đồ thị đóng một vai trò quan trọng trong nhiều lĩnh vực nghiên cứu của các bộ môn khoa học. Chẳng hạn như trong khoa học máy tính, các mạng lưới truyền thông, các thiết bị tính toán hiệu năng cao đều được biểu diễn dưới dạng đồ thị. Hay trong vật lý và hóa học, đồ thị được sử dụng để nguyên cứu về mô hình tự nhiên của một phân tử.

Chính vì lý thuyết đồ thị có vai trò quan trọng như vậy đối với đời sống thực tiễn nên thông qua học phần **đồ án I**, em đã chọn lĩnh vực này để nghiên cứu, bắt đầu từ những kiến thức cơ bản nhất. Khi nghiên cứu *Lý thuyết đồ thị*, ta sẽ phải học một lượng lớn các kiến thức thiên về lý thuyết rất khô khan. Do vậy, để không mất đi sự hứng thú ban đầu em đã chọn một chủ đề đơn giản có ứng dụng sát tới thực tiễn và có liên quan mật thiết tới đồ thị, đó chính là **mạng giao thông**.

Phần lớn bài báo cáo được em tham khảo qua cuốn *Introduction to Graph Theory* của *Douglas B West*. Bộ câu hỏi trong phần ứng dụng được tham khảo trong khóa học *Large-Scale Data Mining: Design and Algorithms* của Đại học California, Los Angeles (*UCLA*). Bài báo cáo nên được xem cùng với file notebook trong đây: *Uber-Traffic-Network*.

Cuối cùng, em xin cảm ơn **TS. Đoàn Duy Trung** - người thầy đã hướng dẫn em thực hiện bài báo cáo đồ án đầu tiên trong quá trình học đại học. Trong quá trình viết báo cáo, rất khó tránh khỏi sai sót, mong rằng thầy sẽ góp ý để em cái thiện ở các học phần đồ án sau.

Hà Nội, tháng 6 năm 2022

Tác giả đồ án

Hiếu

Nguyễn Công Hiếu

Tóm tắt

Bài báo cáo này được chia thành 3 chương chính nghiên cứu về lý thuyết đồ thị và ứng dụng của nó xoay quanh vấn đề về giao thông. Ngoài ra, cũng giới thiệu về một package Python dùng để trực quan hóa dữ liệu dạng đồ thị.

Chương 1: Cơ sở về lý thuyết đồ thị

Phần này sẽ đi vào những kiến thức căn bản về đồ thị như khái niệm, định nghĩa, tính chất, ... để cung cấp cho người đọc một góc nhìn tổng quan về lý thuyết đồ thị. Một vài định lý, mệnh đề hay hệ quả quan trọng sẽ được đưa ra cùng với cách thức chứng minh để giúp người đọc rèn luyện một tư duy nhạy bén trong các vấn đề về đồ thị.

Chương 2: Trực quan hóa đồ thị với NetworkX API

Với mục tiêu ứng dụng lý thuyết đồ thị vào trong các bài toán thực tiến, ta cần một phương thức để biểu diễn và hình dung nó trên máy tính. Trong bài báo cáo này, chúng em xin phép được chọn ngôn ngữ python và thư viện NetworkX để giải quyết vấn đề nêu trên.

Chương 3: Ứng dụng trong mạng lưới giao thông của Uber

Sau khi nắm được nền tảng lý thuyết và cách thức xử lý đồ thị trên máy tính, ta sẽ ứng dụng chúng vào phân tích bộ dữ liệu mạng lưới giao thông của Uber. Ở phần này, chúng em sẽ tự đặt ra các vấn đề và tự trả lời chúng dựa trên các kiến thức và thuật toán đã được nghiên cứu ở chương 1, chương 2 và ở những tài liệu tham khảo.

Mục lục

Chương 1

Cơ sở về lý thuyết đồ thị

1.1 Đồ thị và một số khái niệm cơ bản

1.1.1 Đồ thị vô hướng

Định nghĩa 1.1.1. **Đồ thị** G là một đối tượng được cấu thành từ 3 thành phần: tập đỉnh $V(G)$, tập cạnh $E(G)$ và mối liên hệ giữa 2 đỉnh trong $V(G)$ (không nhất thiết phải phân biệt) thông qua một cạnh trong $E(G)$.

Định nghĩa 1.1.2. **Khuyên** là một cạnh nối từ một đỉnh vào chính nó. **Đa cạnh** là những cạnh có cùng một cặp đỉnh. Một **đơn đồ thị** là đồ thị không có khuyên và không có đa cạnh.

Ví dụ 1.1.1. Cho

$$G = (V(G), E(G), \psi_G)$$

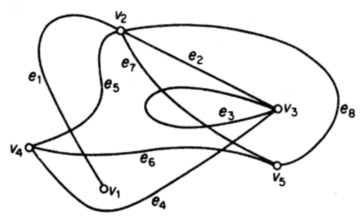
trong đó

$$\begin{aligned} V(G) &= \{v_1, v_2, v_3, v_4, v_5\} \\ E(G) &= \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\} \end{aligned}$$

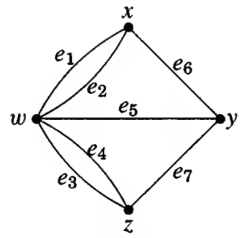
và ψ_G được xác định bởi

$$\psi_G(e_1) = v_1v_2, \psi_G(e_2) = v_2v_3, \psi_G(e_3) = v_3v_3, \psi_G(e_4) = v_3v_4$$

$$\psi_G(e_5) = v_2v_4, \psi_G(e_6) = v_4v_5, \psi_G(e_7) = v_2v_5, \psi_G(e_8) = v_2v_5$$



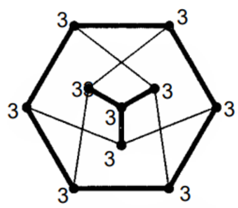
Định nghĩa 1.1.3. *Xét đồ thị $G = (V, E)$ và cạnh $e = (u, v) \in E(G)$. Ta nói rằng hai đỉnh u và v là* ***kề nhau*** *và cạnh e này* ***liên thuộc*** *với hai đỉnh u và v.*



Như trong đồ thị thị trên, ta nói rằng w và y là kề nhau và e_5 là liên thuộc với w, y.

Định nghĩa 1.1.4. Với một đỉnh $u \in V(G)$, ta ký hiệu **bậc của** u là $d_G(u)$ và bằng số cạnh liên thuộc với u, ngoại trừ khuyên do được tính hai lần. **Bậc lớn nhất** của đồ thị G là $\Delta(G)$, **bậc nhỏ nhất** là $\delta(G)$. Một đồ thị được gọi là k**-chính quy** nếu $\Delta(G) = \delta(G) = k$.

Định nghĩa 1.1.5. Bậc của đồ thị G, ký hiệu $n(G)$ là tổng số đỉnh trong đồ thị G. **Kích thước của đồ thị** G, ký hiệu $e(G)$ là tổng số cạnh trong G.



Hình 1.1: Đồ thị Petersen

Đồ thị Petersen trên là đồ thị 3-chính quy do $\Delta(G) = \delta(G) = 3$ và bậc của G bằng 10, kích thước của nó bằng 15.

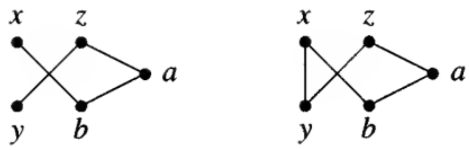
Định lý 1.1.1 (Công thức tổng bậc)**.** *Cho đơn đồ thị G, ta có*

$$\sum_{v \in V(G)} d(v) = 2e(G)$$

Chứng minh. Do mỗi cạnh của đồ thị G đều liên thuộc với 2 đỉnh phân biệt nên phép lấy tổng các bậc sẽ đếm mỗi cạnh 2 lần. ■

Hệ quả 1.1.1.1. *Mọi đồ thị đều có một số chẵn các đỉnh bậc lẻ. Không tồn tại một đồ thị với số lẻ các đỉnh là chính quy với bậc của đỉnh là lẻ.*

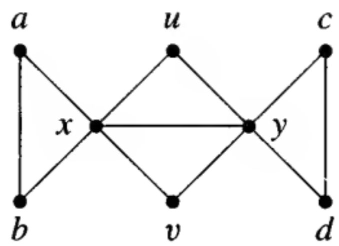
Định nghĩa 1.1.6. *Một **path** là một đồ thị đơn với các đỉnh được sắp xếp sao cho 2 đỉnh là kề khi và chỉ khi chúng là liên tiếp nhau trong một danh sách nào đó. Chu trình là một path với đỉnh đầu và đỉnh cuối trùng nhau.*



Hình 1.2: Path và Cycle

Định nghĩa 1.1.7. *Cho đồ thị G. **Walk** là một danh sách các đỉnh và cạnh $v_0, e_1, v_1, ..., e_k, v_k$ sao cho, với $1 \leq i \leq k$, thì cạnh e_i liên thuộc với v_{i-1} và v_i. **Trail** là một walk nhưng không có sự lặp lại về cạnh. Khi ta nói u, v-walk hay u, v-trail tức là u, v là 2 đỉnh đầu cuối của nó. Một u, v**-path** là một đường đi từ u đến v mà các đỉnh bậc 1 của nó là u và v. Các đỉnh còn lại ngoài u và v được gọi là các **đỉnh trong**.*

Độ dài của một walk, trail, path, cycle là bằng số cạnh của nó. Ta gọi walk, trail là đóng (closed) nếu đỉnh đầu và cuối của nó trùng nhau.

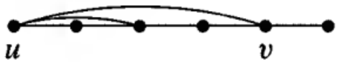


Ta minh họa các khái niệm trong hình trên. Dễ thấy một closed walk với độ dài 12 là danh sách $(a, x, a, x, u, y, c, d, y, v, x, b, a)$. Nếu bỏ đi 2 đỉnh đầu trong danh sách ta sẽ được một closed trail với độ dài 10.

Đồ thị trên có 5 chu trình: (a, b, x), (c, y, d), (u, x, y), (x, y, v), (u, x, v, y). Danh sách (u, y, c, d, y, x, v) tạo thành một u, v-trail chứa các cạnh của một u, v-path u, y, x, v nhưng không chứa u, v-path u, y, v.

Định lý 1.1.2. *Cho đồ thị G với $\Delta(G) = 2$ thì G chứa một chu trình.*

Chứng minh. Giả sử P là path dài nhất có thể trong G và u là một đỉnh đầu cuối của P. Do P không thể được kéo dài thêm nữa nên mọi đỉnh kề với u phải là một đỉnh trong P. Mặt khác, u có số bậc nhỏ nhất bằng 2 nên tồn tại 1 đỉnh v kề với u thông qua một cạnh không thuộc P. Cạnh này sẽ tạo thành một chu trình trong đồ thị. ∎

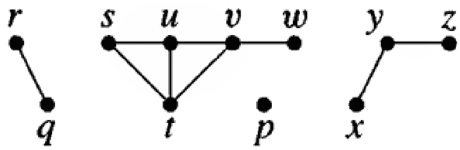


Định nghĩa 1.1.8. *Một **đồ thị con** của dồ thi G là một đồ thị H với $V(H) \subseteq V(G)$ và $E(H) \subseteq E(G)$. Khi đó, ta có thể ký hiệu $H \subset G$.*

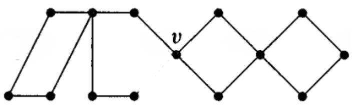
Đồ thị G được gọi là liên thông nếu mọi cặp đỉnh trong G đều thuộc một path; ngược lại, G là không liên thông.

Định nghĩa 1.1.9. *Các **thành phần liên thông** của đồ thị G là các đồ thị con liên thông lớn nhất có thể của nó. Một đỉnh được gọi là **đỉnh cô lập** nếu bậc của nó bằng 0.*

Đồ thị dưới đây có 4 thành phần liên thông và một đỉnh cô lập.



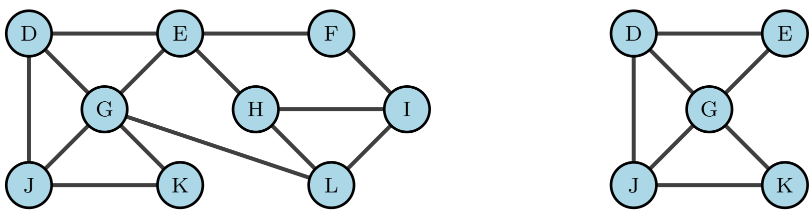
Định nghĩa 1.1.10. ***Đỉnh cắt** hay **cạnh cắt** của một đồ thị là một đỉnh hoặc một cạnh mà nếu ta loại bỏ nó thì sẽ làm tăng số thành phần liên thông.*



Hình 1.3: Đồ thị với đỉnh cắt v và cạnh cắt bên trái của v.

Định nghĩa 1.1.11. *Một **đồ thị con cảm sinh** là đồ thị con thu được bởi phép toán xóa đi một tập các đỉnh.*

Ký hiệu, $G[T] = G - \overline{T}$ với $\overline{T} = V(G) - T$ là đồ thị con của G **cảm sinh** bởi T.



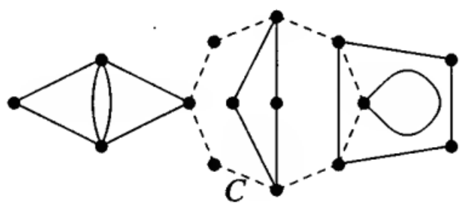
Hình 1.4: Đồ thị bên phải được cảm sinh từ đồ thị bên trái với $T = \{D, E, G, J, K\}$

Định nghĩa 1.1.12. *Một đồ thị được gọi là **Eulerian** nếu nó có một trail đóng chứa tất cả các cạnh. Một trail đóng còn được gọi là một **circuit** nếu chúng ta không muốn chỉ rõ đâu là đỉnh bắt đầu. Một **Eulerian circuit/Eulerian trail** trong một đồ thị là một circuit/trail chứa tất cả các cạnh.*

Định lý 1.1.3. *Đồ thị G là Eulerian khi và chỉ khi nó là liên thông và các đỉnh của nó đều có bậc chẵn.*

Chứng minh. Điều kiện cần: Hiển nhiên G phải liên thông. Giả sử G có một Eulerian circuit C. Dễ thấy muốn đi qua một đỉnh của C phải sử dụng 2 cạnh liên thuộc với nó do đó mọi đỉnh phải có bậc chẵn.

Điều kiện đủ: Với điều kiện liên thông và các đỉnh có số bậc chẵn. Chúng ta xây dựng một Eulerian circuit dựa vào quy nạp trên số cạnh m.
Bước cơ sở: $m = 0$, ta có một trail đóng bao gồm một đỉnh.
Bước quy nạp: $m > 0$. Với số bậc chẵn thì mọi đỉnh trong G có số bậc nhỏ nhất bằng 2. Theo định lý 1.1.2 thì G chứa một chu trình C. Cho $G^{'} = G - E(C)$, hiển nhiên các thành phần của $G^{'}$ cũng là các đồ thị với số bậc của đỉnh là chẵn. Do các thành phần của G' cũng thỏa mãn điều kiện quy nạp nhưng với số cạnh ít hơn m nên ta áp dụng giả thuyết quy nạp dẫn đến mọi thành phần của $G^{'}$ đều có một Eulerian circuit. Để xây dựng được Eulerian circuit của G chúng ta duyệt qua C, nhưng khi đi đến đỉnh thuộc một thành phần của $G^{'}$ (lần đầu tiên) thì ta sẽ duyệt theo Eulerian ciruit của thành phần đó rồi lại duyệt tiếp C. Circuit của G sẽ kết thúc tại đỉnh đầu tiên chúng ta bắt đầu duyệt C. ∎



Hình 1.5: Eulerian circuit

1.1.2 Phép đẳng cấu trong đồ thị

Định nghĩa 1.1.13. Một phép **đẳng cấu** từ một đồ thị đơn G đến một đồ thị H là một song ánh $f : V(G) \to V(H)$ sao cho với mỗi $uv \in E(G)$ khi và chỉ khi $f(u)f(v) \in E(H)$. Lúc đó, ta nói rằng "G là đẳng cấu với H", ký hiệu $G \cong H$.

Ví dụ 1.1.2. Cho G và H là 2 path có 4 đỉnh. Xác định hàm số $f : V(G) \to V(H)$ với $f(w) = a$, $f(x) = d$, $f(y) = b$, $f(z) = c$. Để chỉ ra f có là một đẳng cấu, ta có thể kiểm tra xem f có bảo toàn sự hiện diện của một cạnh qua hai đỉnh bất kì trong G. Chú ý rằng, nếu ta thực hiện vài biến đổi trên $A(G)$ bằng việc thay đổi thứ tự trên dòng và cột thành w, y, z, x ta sẽ được $A(H)$. Như vậy, f là một đẳng cấu. Một phép đẳng cấu khác biến w, x, y, z thành c, b, d, a.

$$\begin{array}{c|cccc} & w & x & y & z \\ \hline w & 0 & 1 & 0 & 0 \\ x & 1 & 0 & 1 & 0 \\ y & 0 & 1 & 0 & 1 \\ z & 0 & 0 & 1 & 0 \end{array} \qquad \begin{array}{c|cccc} & w & y & z & x \\ \hline w & 0 & 0 & 0 & 1 \\ y & 0 & 0 & 1 & 1 \\ z & 0 & 1 & 0 & 0 \\ x & 1 & 1 & 0 & 0 \end{array} \qquad \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 0 & 0 & 1 \\ b & 0 & 0 & 1 & 1 \\ c & 0 & 1 & 0 & 0 \\ d & 1 & 1 & 0 & 0 \end{array}$$

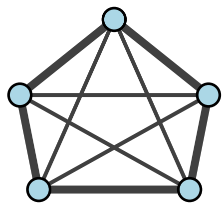
Chú ý 1.1.1. *Tìm kiếm phép đẳng cấu.* Ta có thể thực hiện một phép giao hoán cho cả dòng và cột của $A(G)$. Nếu ma trận mới thu được giống với $A(H)$ thì phép giao hoán đó trả về một phép đẳng cấu.

1.1.3 Phân rã đồ thị và một số dạng đồ thị đặc biệt

Định nghĩa 1.1.14. ***Phần bù*** $\overline{G}$ *của một đơn đồ thị G là một đơn đồ thị với tập đỉnh $V(G)$ xác định các cạnh $uv \in E(\overline{G})$ khi và chỉ khi $uv \notin E(G)$.*

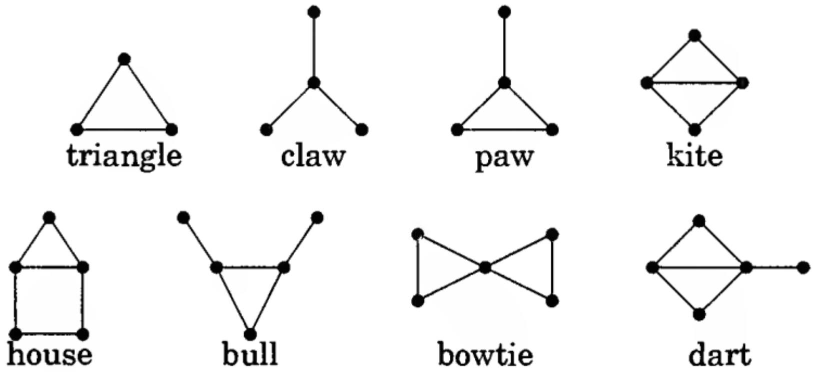
Dễ thấy, một đồ thị đầy đủ luôn có thể được phân rã thành G và $\overline{G}$. Sau đây là định nghĩa về phép phân rã trong đồ thị.

Định nghĩa 1.1.15. ***Phân rã*** *hay* ***phân hoạch*** *của một đồ thị là danh sách các đồ thị con của nó sao cho mỗi cạnh trong đồ thị chỉ xuất hiện duy nhất trong một đồ thị con.*



Hình trên là đồ thị đầy đủ với 5 đỉnh và được phân tách thành 2 chu trình. Chu trình bên ngoài được thể hiện bằng các cạnh đậm, chu trình còn lại là các cạnh nhạt hơn.

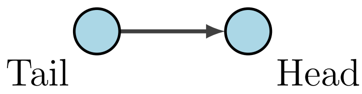
Để dễ dàng phân loại, trong lý thuyết đồ thị người ta đặt tên cho một số đồ thị đặc biệt sau.



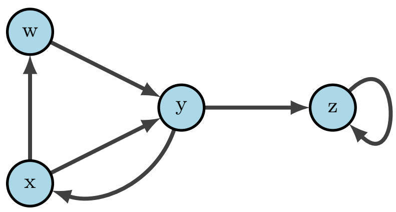
1.1.4 Đồ thị có hướng

Định nghĩa 1.1.16. *Một đồ thị G được gọi là có hướng nếu nó gồm* ***tập đỉnh*** *$V(G)$,* ***tập cạnh*** *$E(G)$, và* ***một hàm*** *xác định các cặp đỉnh thuộc $V(G)$ (có thứ tự) cho mỗi cạnh thuộc $E(G)$.*

Đỉnh đầu tiên trong cặp có thứ tự đó được gọi là **đỉnh đuôi**, đỉnh còn lại là **đỉnh đầu**. Một cạnh sẽ được định hướng từ đuôi đến đầu.



Định nghĩa 1.1.17. ***Khuyên*** *là một cạnh mà đỉnh đuôi trùng với đỉnh đầu.* ***Đa cạnh*** *là các cạnh có cùng cặp đỉnh đuôi - đỉnh đầu. Một* ***đơn đồ thị có hướng*** *sẽ không chứa đa cạnh.*



Đồ thị trên không phải đơn đồ thị có hướng do có đa cạnh xy và yx. Nếu bỏ 1 trong 2 (hoặc cả 2) thì nó sẽ trở thành đơn đồ thị có hướng.

Chú ý 1.1.2. Trong một đồ thị có hướng, ngoài việc trên các cạnh xác định một thứ tự giữa 2 đỉnh thì các định nghĩa về **walk**, **trail**, **path**, ... đều tương tự như đôi với đồ thị vô hướng.

Định nghĩa 1.1.18. *Xét một đồ thị có hướng G. Nếu như ta bỏ sự định hướng trên các cạnh mà đồ thị vẫn liên thông thì ta gọi G là* ***liên thông yếu****. G là* ***liên thông mạnh*** *nếu như với mọi cặp đỉnh thuộc $V(G)$ luôn luôn tồn tại một đường đi giữa chúng.*

Dễ thấy, đồ thị ở định nghĩa (1.1.17) là một đồ thị liên thông mạnh.

Định nghĩa 1.1.19. *Cho v là một đỉnh trong một đồ thị có hướng. **Bán bậc ra** $d^+(v)$ là số cạnh có v là đỉnh đuôi. **Bán bậc vào** $d^-(v)$ là số cạnh có v là đỉnh đầu.*

Ngoài ra, trong đồ thị có hướng ta cũng có bán bậc vào nhỏ nhất/lớn nhất, ký hiệu $\delta^-(G)$ và $\Delta^-(G)$. Đối với bán bậc ra là $\delta^+(G)$ và $\Delta^+(G)$.

Định lý 1.1.4. Trong đồ thị có hướng G,

$$\sum_{v\in V(G)} d^+(v) = e(G) = \sum_{v\in V(G)} d^-(v)$$

Định nghĩa 1.1.20. *Một **Eulerian trail** trong một đồ thị có hướng là một trail chứa tất cả các cạnh. Một **chu trình Euler** là một trail đóng chứa tất cả các cạnh.*

Một đồ thị có hướng sẽ được gọi là **Eulerian** nếu như nó có một chu trình Euler.

Định lý 1.1.5. Đồ thị có hướng G là Eulerian khi và chỉ khi $d^+(v) = d^-(v)$ với mọi đỉnh v và đồ thị vô hướng ẩn sau nó có nhiều nhất một thành phần liên thông không tầm thường (có nhiều hơn 1 đỉnh).

Chứng minh. Điều kiện cần: Giả sử G là Eulerian. Tất cả các cạnh của đồ thị G sẽ nằm trên một chu trình Euler. Do đó, tất cả các cạnh là trên cùng một thành phần liên thông. Các thành phần khác nếu có sẽ không có cạnh và chúng sẽ là các đỉnh cô lập. Với bất kỳ đỉnh v thuộc vào thành phần không tầm thường, số cạnh rời v là bằng với số cạnh đi vào nó.
Điều kiện đủ: Ta chứng minh bằng cách quy nạp theo số cạnh m.

Nếu $m = 0$, chu trình Euler là rỗng và điều kiện được thỏa mãn.

Giả sử rằng điều kiện đủ sẽ được thỏa mãn với các đồ thị có nhiều nhất m cạnh. Ta đi chứng minh rằng nó cũng đúng với $m + 1$ cạnh.

Xét một đồ thị với $m + 1$ cạnh và có nhiều nhất một thành phần liên thông không tầm thường H với $d^+(v) = d^-(v) \geq 1$ với mọi $v \in V(H)$. Dễ thấy nó chứa một chu trình C. Xóa tất cả các cạnh trên C trong đồ thị G dẫn đến việc H có thể bị phân rã thành nhiều thành phần liên thông $H_1, H_2, \dots, H_r$. Rõ ràng $d^+(v) = d^-(v)$ vẫn đúng.
Mỗi thành phần H_i sẽ có nhiều nhất m cạnh. Bằng giả thuyết quy nạp, H_i có một chu trình Euler C_i. Do G chỉ có một thành phần liên thông không

tầm thường nên chu trình C phải giao với mọi thành phần H_i. Chọn một đỉnh $v_i \in V(C) \cap C(H_i)$. Các đỉnh $v_1, \ldots, v_r$ chia chu trình C thành r đường (ví dụ $v_1P_1v_2, v_2P_2v_3, \ldots, v_rP_rv_1$). Sắp xếp các chu trình C_i sao cho đỉnh bắt đầu và đỉnh kết thúc là v_i. Như vậy, ta đã xây dựng một chu trình Euler như sau

$$C_1P_1C_2P_2 \ldots C_rP_rv_1$$

và nó chứa tất cả các cạnh của G. ■

1.2 Các cách biểu diễn đồ thị

Có rất nhiều cấu trúc dữ liệu có thể được sử dụng để biểu diễn đồ thị. Mỗi cách đều có ưu và nhược điểm riêng. Phần này sẽ thảo luận về một số cách biểu diễn động và tĩnh của đồ thị được tham khảo trong [2].

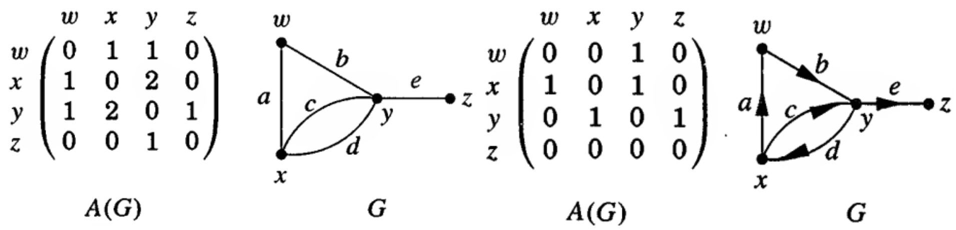
1.2.1 Biểu diễn tĩnh

Ma trận kề

Ma trận kề A của một đồ thị $G(V, E)$ là một ma trận cỡ $|V| \times |V|$ được xác định bởi:

$$A(i,j) = \begin{cases} 1, & \text{nếu } (i,j) \in E(G) \\ 0, & \text{nếu } (i,j) \notin E(G) \end{cases}$$

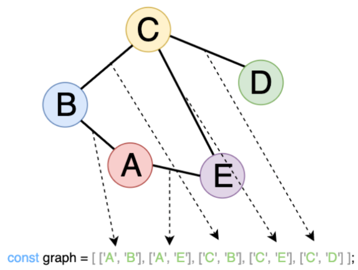
Cách biểu diễn trên có thể được dùng cho cả đồ thị vô hướng và có hướng. Tuy nhiên, nếu trong đồ thị vô hướng ma trận kề là đối xứng thì trong đồ thị có hướng điều này lại không đúng.



Hình 1.6: Ma trận kề của đồ thị vô hướng và có hướng (tham khảo trong [1])

Danh sách cạnh

Cách biểu diễn này chỉ đơn giản là một danh sách các cặp đỉnh không thứ tự (vô hướng) hoặc có thứ tự (có hướng).

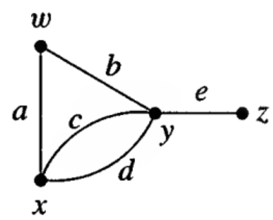


Hình 1.7: Danh sách cạnh của một đồ thị vô hướng

Ma trận liên thuộc

Đây là một dạng biểu diễn ít được sử dụng, nhưng nó vẫn có vai trò quan trọng trong lý thuyết. Một **ma trận liên thuộc** $M(G)$ là ma trận cỡ $n \times m$ (các hàng biểu diễn các đỉnh - các cột biểu diễn các cạnh), trong đó

$$M(i,j) = \begin{cases} 1, & \text{nếu } v_i \text{ liên thuộc với } e_j \\ 0, & \text{nếu ngược lại} \end{cases}$$

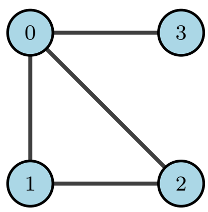


$$\begin{array}{c} \\ w \\ x \\ y \\ z \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

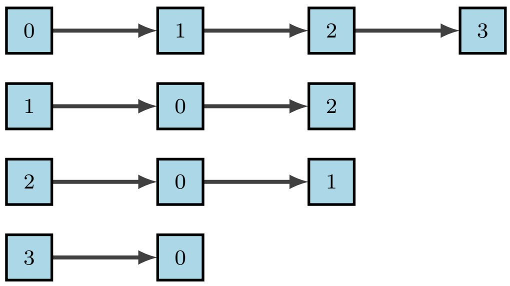
G $M(G)$

1.2.2 Biểu diễn động - danh sách kề

Một trong những cách biểu diễn phổ biến nhất là danh sách kề, ở cấu trúc dữ liệu này ta biểu diễn đồ thị thành một mảng các danh sách liên kết.



Với đồ thị trên, ta có thể biểu diễn dưới dạng mảng như dưới đây.

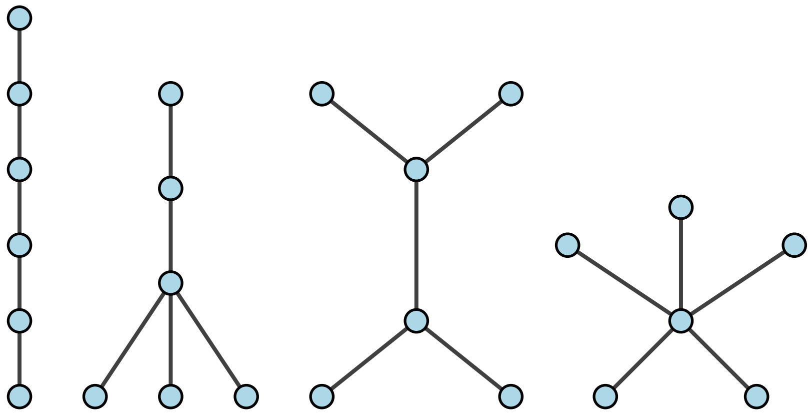


1.3 Đồ thị dạng cây

1.3.1 Cây và các tính chất cơ bản

Định nghĩa 1.3.1. ***Cây*** *được định nghĩa là một đồ thị liên thông nhưng không có chu trình. Trong cây, tồn tại những đỉnh là* ***lá*** *nếu bậc của chúng bằng 1.*

Nếu một đồ thị con bao trùm của một đồ thị là một cây, ta gọi nó là ***cây bao trùm****.*



Hình 1.8: Các cây đồ thị với 6 đỉnh

Sau đây là một số đặc trưng nhận dạng một cây đồ thị.

Định lý 1.3.1. Cho một đồ thị G với n đỉnh $(n > 1)$, những tính chất đây là tương đương và là dấu hiệu nhận biết một đồ thị có là cây:

1. G là liên thông và không tồn tại chu trình.

2. G là liên thông và có $n - 1$ cạnh.

3. G có $n - 1$ cạnh và không chứa chu trình.

4. Với mọi u, $v \in V(G)$, G có chính xác một $u, v - path$.

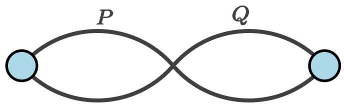
Chứng minh.

1⇒{**2, 3**}: Chúng ta sẽ sử dụng quy nạp trên n. Với $n = 1$, hiển nhiên nó thỏa mãn {**2, 3**}. Với $n > 1$, ta giả định các kết quả sẽ đúng với đồ thị cùng tính chất nhưng chỉ có $n-1$ đỉnh. Xét đồ thị G thỏa mãn **1**, nếu đặt $G' = G - v$ (v là lá) thì G' cũng liên thông và không chứa chu trình. Áp dụng giả thuyết quy nạp vào G' sẽ có $e(G') = n - 2$. Do chỉ một cạnh là là liên thuộc với v nên ta có $e(G) = n - 1$.

2⇒{**1, 3**}: Giả sử G chứa chu trình. Ta lần lượt cạnh từ các chu trình sao cho G trở thành G' với không chu trình. Do cầu không tồn tại trong một chu trình nên G' là liên thông. Do G' là liên thông và không có chu trình nên $e(G') = n - 1$. Mà $e(G) = n - 1$ nên thực chất ta không xóa được cạnh nào nên $G = G'$.

3⇒{**1, 2**}: Cho $G_1, \ldots, G_k$ là các thành phần liên thông của G. Do mỗi đỉnh chỉ xuất hiện trong 1 thành phần nên $\sum_i n(G_i) = n$. Mặt khác, G không chứa chu trình nên mỗi thành phần sẽ thỏa mãn tính chất **1**. Do đó $e(G_i) = n(G_i) - 1$. Cộng tổng theo i cho ta $e(G) = \sum_i [n(G_i) - 1] = n - k$. Theo giả thuyết, $e(G) = n - 1$ nên $k = 1$ và G là liên thông.

1⇒**4**: Do G liên thông nên mỗi luôn tồn tại đường đi giữa 2 đỉnh bất kỳ. Nếu có nhiều hơn một đường đi giữa 2 đỉnh, ta chọn 2 đường đi phân biệt P, Q với độ dài ngắn nhất mà có cùng đỉnh đầu cuối (như hình dưới).



Với cách chọn như vậy, dễ thấy rằng không có đỉnh trong nào của P hoặc Q thuộc về một đường đi nào khác. Tức là, $P \cup Q$ là một chu trình, điều này trái với giả thuyết của **1**.

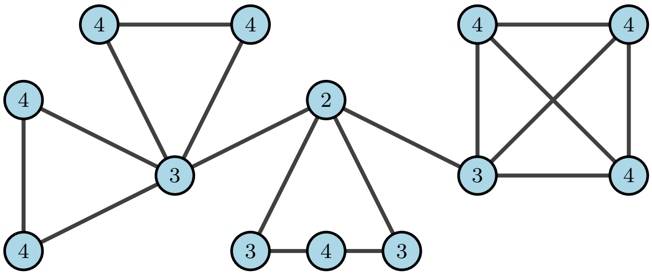
4⇒**1**: Nếu tồn tại một đường đi u, v với mọi $u, v \in V(G)$ thì G liên thông. Nếu G chứa chu trình C thì sẽ có 2 đường đi u, v với $u, v \in V(C)$. Như vậy, G sẽ không chứa chu trình. ■

1.3.2 Các khái niệm về khoảng cách trên cây và đồ thị

Định nghĩa 1.3.2. *Nếu G có đường đi giữa u, v thì **khoảng cách** từ u đến v (ký hiệu $d_G(u, v)$) là độ dài ngắn nhất của trong các đường đi đó. Nếu không tồn tại đường đi như vậy thì $d_G(u, v) = \infty$. **Đường kính** của G là $\max_{u,v \in V(G)} d(u, v)$ (ký hiệu $diam(G)$).*

Độ lệch tâm *(eccentricity) của một đỉnh u, ký hiệu* $\epsilon(u)$*, chính bằng* $\max_{v\in V(G)} d(u,v)$. ***Bán kính*** *của G, ký hiệu* $rad(G)$*, bằng* $\min_{u\in V(G)} \epsilon(u)$.

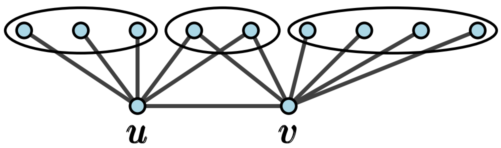
Ví dụ 1.3.1. Xét đồ thị sau đây:



Các nhãn trên đỉnh tương ứng của đồ thị biểu diễn độ lệch của nó so với tâm (*eccentricity*). Bán kính của đồ thị bằng 2, đường kính bằng 4 và độ dài của *path* dài nhất bằng 7.

Định lý 1.3.2. Cho đồ thị đơn G, nếu $diam(G) \geq 3 \Rightarrow diam(\overline{G}) \leq 3$.

Chứng minh. Nếu $diam(G) > 2$, thì tồn tại những đỉnh không kề u, $v \in V(G)$ mà không có chung hàng xóm. Do đó, mọi $x \in V(G) - \{u, v\}$ nhận ít nhất một phần tử thuộc $\{u, v\}$ không là hàng xóm. Điều này dẫn tới x kề với ít nhất một trong $\{u, v\}$ trong $\overline{G}$. Do bởi $uv \in E(\overline{G})$ nên với mỗi cặp x, y có x, y-path với độ dài lớn nhất bằng 3 trong $\overline{G}$ đi qua $\{u, v\}$. Như vậy, $diam(G) \leq 3$.



■

Định nghĩa 1.3.3. ***Tâm*** *(center) của một đồ thị G là đồ thị con cảm sinh bởi các đỉnh có độ lệch tâm nhỏ nhất.*

Chú ý 1.3.1. Ta có một vài nhận xét về tâm của một đồ thị

- Tâm của một đồ thị là chính nó khi và chỉ khi bán kính và đường kính của nó là bằng nhau.
- Tâm của một cây là một đỉnh hoặc một cạnh.

1.3.3 Cây bao trùm nhỏ nhất trên đồ thị

Định lý 1.3.3 (Công thức Cayley). *Cho một tập $S \subseteq \mathbb{N}$ với lực lượng là n, sẽ có n^{n-2} cây với tập đỉnh S.*

Theo như định lý trên, ta biết được rằng trong một đồ thị sẽ tồn tại rất nhiều cây con bao trùm. Bài toán cây bao trùm nhỏ nhất tức là trong số các cây như vậy, ta tìm cây có tổng trọng số nhỏ nhất.

Để cho đơn giản, phần này chúng em chỉ xét đến cây bao trùm nhỏ nhất trong đồ thị vô hướng có trọng số. Dưới đây là một vài tính chất của cây bao trùm nhỏ nhất trong đồ thị vô hướng có trọng số. Phần chứng minh có thể được tìm thấy ở trang web: `https://vnoi.info/wiki/algo/graph-theory/minimum-spanning-tree.md`

1. **Tính chất chu trình**: Trong một chu trình C bất kỳ, nếu e là cạnh có trọng số lớn nhất (không có cạnh nào lớn bằng nó) thì e không thể nằm trên bất kỳ cây bao trùm nhỏ nhất nào.

2. **Đường đi hẹp nhất**: Xét 2 đỉnh u, v bất kỳ trong đồ thị. Nếu w là trọng số của cạnh lớn nhất trên đường đi từ u đến v trên cây bao trùm nhỏ nhất của đồ thị thì sẽ không thể tìm được đường đi nào từ u đến v trên đồ thị ban đầu chỉ đi qua những cạnh có trọng số nhỏ hơn v.

3. **Tính duy nhất**: Nếu tất cả các cạnh đều có trọng số khác nhau thì chỉ có duy nhất một cây bao trùm nhỏ nhất.

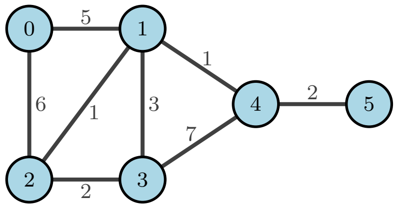
4. **Tính chất cạnh nhỏ nhất**: Nếu e là cạnh có trọng số nhỏ nhất của đồ thị, và không có cạnh nào có trọng số bằng e thì e nằm trên mọi cây bao trùm nhỏ nhất của đồ thị.

Phần này, chúng em sẽ trình bày về 2 thuật toán cơ bản để tìm kiếm cây bao trùm nhỏ nhất trong đồ thị vô hướng có trọng số.

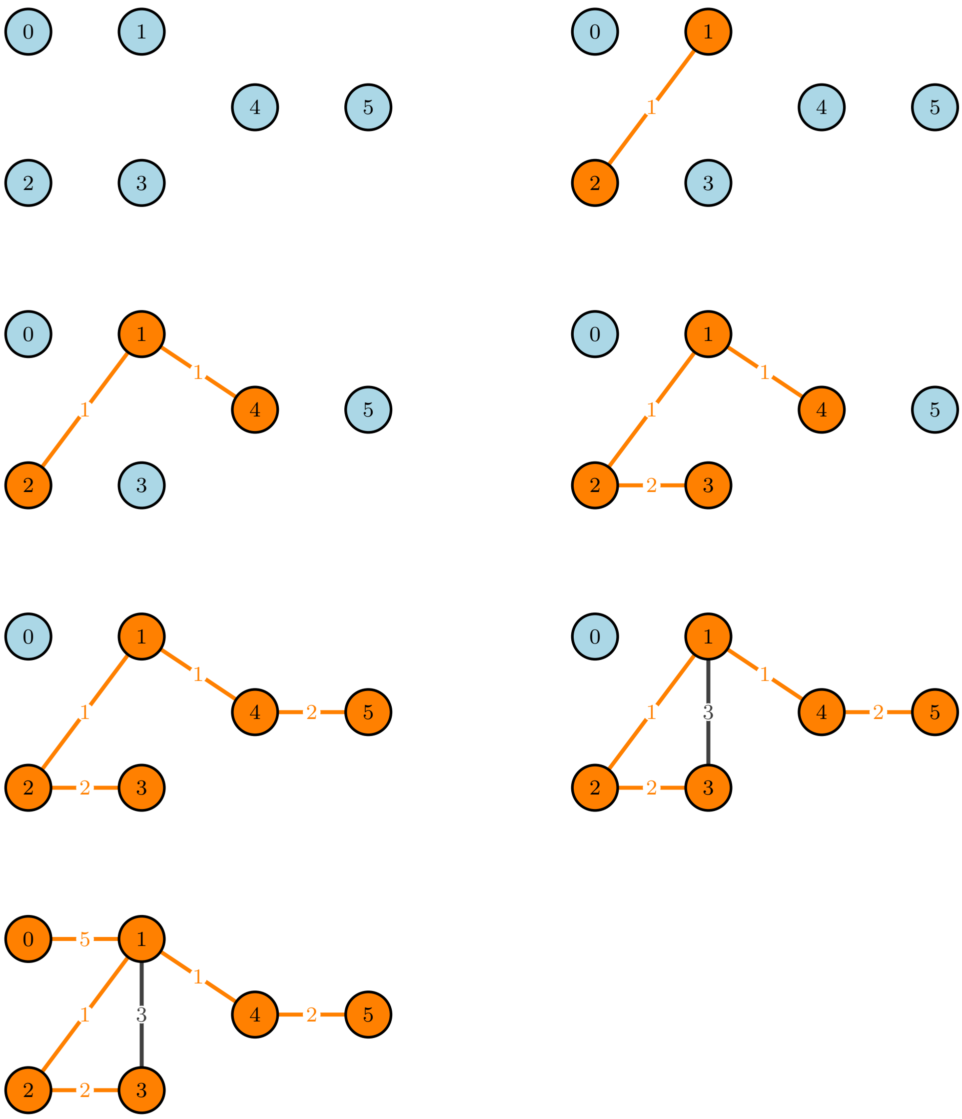
Thuật toán Kruskal

Thuật toán 1.3.1 (Thuật toán Kruskal).
Đầu vào: Một đồ thị liên thông có trọng số.
Ý tưởng: Ban đầu cây bao trùm nhỏ nhất là một rừng với các đỉnh là các cây. Lần lượt thêm các cạnh với trọng số không giảm và liên kết các cây lại để cuối cùng tạo thành một cây duy nhất.
Khởi tạo: $E(H) = \emptyset$.

Bước lặp: Nếu cạnh có trọng số nhỏ nhất tiếp theo nối 2 thành phần của H thì thêm nó vào $E(H)$, ngược lại sẽ bỏ qua. Thuật toán kết thúc khi H là liên thông.
Độ phức tạp: Nếu không xét đến độ phức tạp khi phải sắp xếp lại các cạnh thì thuật toán duyệt qua $|E|$ cạnh, mỗi cạnh ta kiểm tra xem nó có nối được 2 thành phần lại không mất $O(log(V))$. Vậy thuật toán có độ phức tạp là $O(Elog(V))$.

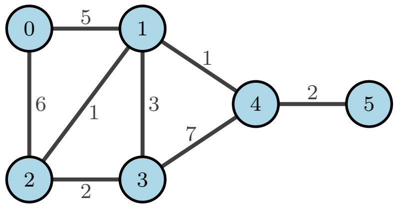


Chạy thuật toán Kruskal trên đồ thị trên (các bước thực hiện từ trái sang phải, trên xuống dưới), ta được:

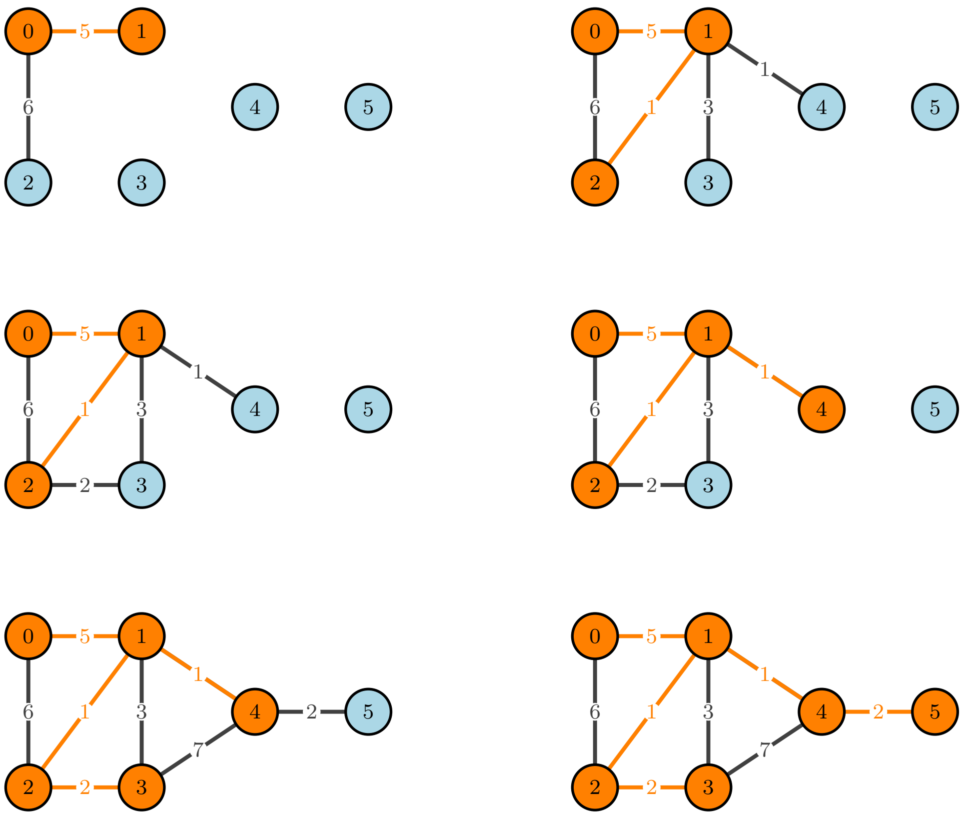


Thuật toán Prim

Thuật toán 1.3.2 (Thuật toán Prim)**.**
Đầu vào: Một đồ thị G liên thông có trọng số và một đỉnh x bất kỳ trong đó.
Ý tưởng: Xuất phát từ một cây chỉ chứa một đỉnh, sau mỗi bước ta thêm một cạnh mới vào cây cho tới khi bao trùm được hết các đỉnh của đồ thị.
Khởi tạo: $V(T) = \{x\}$, $E(T) = \emptyset$.
Bước lặp: Nếu $V(T) \neq V(G)$ thì ta chọn cạnh uv có trọng số nhỏ nhất sao cho $u \in V(T)$ và $v \in V(G)$. Ta sẽ thêm cạnh này vào $E(T)$ và thêm đỉnh v vào $V(T)$.
Độ phức tạp: $O(V^2)$ do ta phải duyệt qua mọi đỉnh của đồ thị ban đầu và trong mỗi đỉnh đó ta lại duyệt qua các đỉnh kề với nó để cập nhật trọng số.



Tiếp tục xét ví dụ trên, nhưng ta chạy với thuật toán prim với đỉnh xuất phát là đỉnh `0`, các bước được thực hiện như dưới đây (trái sang phải, trên xuống dưới):



1.4 Đường đi ngắn nhất

1.4.1 Thuật toán BFS

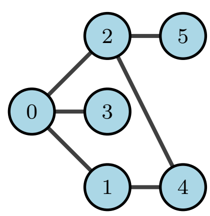
Breadth-First Search là một trong những thuật toán tìm kiếm cơ bản và quan trọng bậc nhất trên đồ thị. Ý tưởng của nó là xuất phát từ một đỉnh bất kì và lan dần ra các điểm gần nó nhất.

Thuật toán BFS có thể được ứng dụng để giải quyết nhiều bài toán, trong đó có bài toán tìm đường đi ngắn nhất. Xuất phát từ một đỉnh, thuật toán sẽ tìm đường đi ngắn nhất tới tất cả các đỉnh còn lại. Thuật toán có thể hoạt động trên đồ thị vô hướng hoặc có hướng, tuy nhiên lại chỉ có thể xét trên đồ thị không có trọng số (nếu có trọng số thì tất cả phải bằng nhau).

Thuật toán 1.4.1 (Breadth-First Search)**.**
Đầu vào: Một đồ thị (vô hướng hoặc có hướng) không có trọng số và một đỉnh xuất phát u.
Khởi tạo: $R = \{u\}$, $S = \emptyset$, $d(u, u) = 0$.
Bước lặp: Nếu $R \neq \emptyset$,

- Lấy đỉnh v đầu tiên ra khỏi R.
- Xét tất cả các đỉnh kề z với v mà không thuộc $S \cup R$,
  - Chèn các đỉnh đó vào cuối R
  - Gán cho khoảng cách $d(u, z) = d(u, v) + 1$
  - Chuyển v từ R vào S.

Bước cuối: Truy vết đường đi dựa vào $d(u, z)$.



Xét đỉnh 0 là đỉnh xuất phát, thực hiện thuật toán BFS như sau:

Trong bảng dưới, các ô màu xanh giúp ta truy vết lại đường đi từ đỉnh xuất phát đến một đỉnh nào đó trong đồ thị. Ví dụ: đường đi ngắn nhất từ đỉnh 0 đến đỉnh 5 là 2 và ta truy vết đường đi như sau: Từ hàng đã cập nhật khoảng cách của đỉnh 5 (2) gióng lên tìm hàng chứa ∞ đầu tiên, sau đó di chuyển sang ngang tìm ô màu xanh và đi ngược lại theo các ô màu xanh lên trên cùng.

	0	1	2	3	4	5
$R_0 = \{0\}$ $S_0 = \{\}$	0	∞	∞	∞	∞	∞
$R_1 = \{1, 2, 3\}$ $S_1 = \{0\}$	0	1	1	1	∞	∞
$R_2 = \{2, 3, 4\}$ $S_2 = \{0, 1\}$	0	1	1	1	2	∞
$R_3 = \{3, 4, 5\}$ $S_3 = \{0, 1, 2\}$	0	1	1	1	2	2

1.4.2 Thuật toán Dijkstra

Thuật toán Dijkstra là một thuật toán phổ biến và có tính ứng dụng cao. Nó có nhiều biến thể, trong đó thuật toán gốc của nó là đi tìm path ngắn nhất giữa 2 đỉnh. Tuy nhiên, trong một biến thể được biết đến nhiều hơn, thuật toán cố định một đỉnh (đỉnh xuất phát) và tìm path ngắn nhất tới tất cả các đỉnh còn lại - giống với BFS.

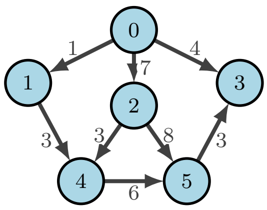
Một ưu điểm khiến cho Dijkstra vượt trội hơn so với BFS là nó được ứng dụng trên một lớp đồ thị rộng hơn (có trọng số - nhưng không âm).

Ý tưởng của thuật toán là duy trì tập các đỉnh S mà ta đã biết chắc chắn đường đi ngắn nhất xuất phát từ đỉnh u. Qua các bước lặp, ta nới rộng tập S này cho đến khi nó chứa mọi đỉnh thuộc đồ thị.

Thuật toán 1.4.2 (Thuật toán Dijkstra)**.**
Đầu vào: Một đồ thị (có hướng hoặc vô hướng) với các trọng số không âm và một đỉnh xuất phát u. Trọng số trên cạnh xy là $w(xy)$, và nếu không tồn tại cạnh giữa x và y thì $w(xy) = \infty$.
Khởi tạo: $S = \{u\}$, $t(u) = 0$, $t(z) = w(uz)$ với $u \neq z$.
Bước lặp:

- Chọn một đỉnh $v \notin S$ sao cho $t(v) = \min_{z \notin S} t(z)$. Sau đó thêm v vào S.
- Tìm kiếm các cạnh xuất phát từ v để cập nhật các khoảng cách t. Với mỗi cạnh vz với $z \notin S$, ta cập nhật $t(z)$ bằng $\min\{t(z), t(v) + w(vz)\}$.

Bước cuối: Đặt $d(u, v) = t(v)$ với mọi v.



Áp dụng thuật toán Dijkstra với đồ thị trên, ta có:

	0	1	2	3	4	5
$S_0 = \{0\}$	0	1	7	4	∞	∞
$S_1 = \{0, 1\}$	0	1	7	4	4	∞
$S_2 = \{0, 1, 3\}$	0	1	7	4	4	∞
$S_3 = \{0, 1, 3, 4\}$	0	1	7	4	4	10
$S_4 = \{0, 1, 3, 4, 2\}$	0	1	7	4	4	10

Bảng trên cho ta giá trị của các đường đi ngắn nhất xuất phát từ đỉnh 0. Để truy vết các đường đi này, ta có thể dùng một mảng `trace`.

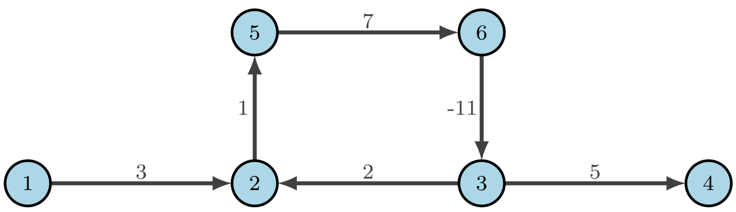
1.4.3 Thuật toán Bellman-Ford

Giống như một bản nâng cấp hơn của BFS và Dijikstra, thuật toán của Bellman-Ford giúp ta xử lý với lớp đồ thị có tồn tại các trọng số âm.

Trước khi đi đến phần ý tưởng và thuật toán, cần nhắc đến một khái niệm đặc biệt tồn tại trong đồ thị có trọng số âm.

Định nghĩa 1.4.1 (Chu trình âm)**.** *Xét một chu trình trong một đồ thị G được gọi là chu trình âm nếu tổng các trọng số trên cạnh của chu trình đó âm.*

Chú ý 1.4.1. *Nếu trên một đường đi từ u đến v chứa một chu trình âm thì độ dài đường đi ngắn nhất là $-\infty$. Sự xuất hiện của chu trình âm sẽ khiến một số cặp đỉnh không tồn tại đường đi ngắn nhất.*



Trong đồ thị trên, để đi từ đỉnh **1** đến đỉnh **4** ta có thể đi vô hạn qua chu trình âm **2, 5, 6, 3** rồi mới đi đến **4**. Điều này dẫn tới không tồn tại đường đi ngắn nhất từ **1** đến **4**.

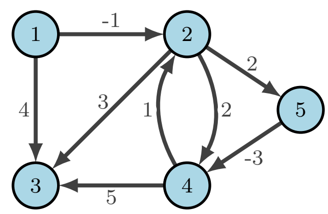
Ý tưởng của thuật toán Bellman-Ford, là ta sẽ lặp qua từng đỉnh của đồ thị, mỗi lần lặp sẽ duyệt qua toàn bộ cạnh (u, v) và so sánh đường đi $S \to v$ đã tìm được với đường đi $S \to u \to v$.

Thuật toán 1.4.3 (Thuật toán Bellman-Ford)**.**
Đầu vào: Một đồ thị hoặc có hướng cho phép có trọng số âm (hoặc vô hướng

nhưng không tồn tại trọng số âm) và một đỉnh xuất phát s.
Khởi tạo: Đặt d là mảng 1 chiều thể hiện đường đi ngắn nhất. Trong đó, $d(u) = \infty$ với mọi $u \neq s$ và $d(s) = 0$.
Bước lặp: Lặp $|V| - 1$ lần. Trong đó, $|V|$ là số đỉnh của đồ thị đã cho. Xét lần lượt các cạnh uv (theo thứ tự bất kỳ):

- Tính giá trị $curr = d(u) + w(u, v)$
- Nếu $curr < d(v)$ thì cập nhật $d(v) = curr$

Bước cuối: Lặp giống như trên với $|V|$ vòng lặp. Nếu mảng d được cập nhật ở bất cứ đỉnh nào thì không tồn tại đường đi ngắn nhất tới đỉnh đó.



Sử dụng thuật toán Bellman-Ford với đồ thị trên với các cạnh được xét theo thứ tự như sau $(25, 42, 24, 12, 13, 43, 23, 54)$, ta được:

1	2	3	4	5
0	∞	∞	∞	∞
0	-1	2	∞	∞
0	-1	2	-2	1
0	-1	2	-2	1
0	-1	2	-2	1

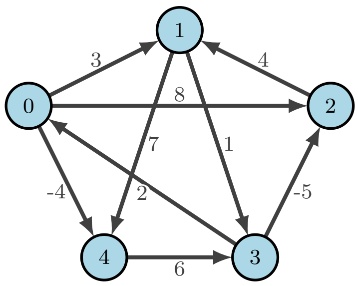
Hàng đầu tiên ở bảng trên là số thứ tự của các cạnh. Hàng thứ 2 là bước khởi tạo và 4 hàng sau đó tương ứng với 4 lần lặp của thuật toán (đồ thị có 5 đỉnh). Dễ thấy từ lần lặp thứ 2, thuật toán không thể cập nhật thêm được nữa. Từ đây, ta có các chi phí của các đường đi xuất phát từ đỉnh 1 đến các đỉnh còn lại.

1.4.4 Thuật toán Floyd-Warshall

Thuật toán Floyd-Warshall được sử dụng để giải quyết bài toán đường đi giữa mọi cặp đỉnh, trong đó đồ thị (vô hướng hoặc có hướng) có thể tồn tại các cạnh với trọng số âm. Giống với Bellman-Ford, nếu tồn tại các chu trình âm thì sẽ có những đỉnh không tồn tại đường đi ngắn nhất.

Ý tưởng của bài toán có hơi hướng cấu trúc đệ quy. Giả sử như có một đường đi ngắn nhất từ **0**→ **4** như sau: **0**→**1**→**2**→**3**→**4**. Như vậy, ta có thể đi tìm đường đi ngắn nhất từ **0**→**2** cộng với đường đi ngắn nhất từ **2**→**4**. Cứ tiếp như vậy, ta sẽ đi tìm các đường đi ngắn nhất khác đơn giản hơn và tối ưu hơn.

Thuật toán 1.4.4 (Thuật toán Floyd-Warshall)**.**
Đầu vào: Một đồ thị có hướng cho phép trọng số âm hoặc vô hướng nhưng không tồn tại trọng số âm.
Khởi tạo: Ma trận kề W. Ma trận chứa giá trị đường đi ngắn nhất D. Ban đầu, $D = W$. Chú ý rằng nếu không tồn tại cạnh giữa u và v thì $W[u, v] = \infty$.
Bước lặp: Thuật toán gồm 3 vòng lặp lồng nhau tương ứng với 3 chỉ số u, v, k. Ta xét mọi đỉnh trong đồ thị như một đỉnh trung gian trong vòng lặp k. Hai vòng lặp u, v có ý nghĩa là chèn đỉnh k vào đường đi giữa 2 đỉnh này.
Bên trong vòng lặp sẽ kiểm tra nếu $D[u, v] > D[u, k] + D[k, v]$ thì ta cập nhật $D[u, v] = D[u, k] + D[k, v]$.
Bước cuối: Để truy vết lại đường đi, ta có thể thêm một mảng *trace*. Nếu điều kiện ở bước lặp được thỏa mãn thì $trace[u, v] = trace[k, v]$.



Áp dụng thuật toán Floyd-Warshall với đồ thị trên, đầu tiên ta cần chuẩn bị ma trận kề $W = D^0$.

	0	1	2	3	4
0	0	3	8	∞	-4
1	∞	0	∞	1	7
2	∞	4	0	∞	∞
3	2	∞	-5	0	∞
4	∞	∞	∞	6	0

Chọn đỉnh 0 là đỉnh chính giữa, ta có D^1:

	0	1	2	3	4
0	0	3	8	∞	-4
1	∞	0	∞	1	7
2	∞	4	0	∞	∞
3	2	5	-5	0	-2
4	∞	∞	∞	6	0

Chọn đỉnh 1 là đỉnh chính giữa, ta có D^2:

	0	1	2	3	4
0	0	3	8	4	-4
1	∞	0	∞	1	7
2	∞	4	0	5	11
3	2	5	-5	0	-2
4	∞	∞	∞	6	0

Chọn đỉnh 2 là đỉnh chính giữa, ta có D^3:

	0	1	2	3	4
0	0	3	8	4	-4
1	∞	0	∞	1	7
2	∞	4	0	5	11
3	2	-1	-5	0	-2
4	∞	∞	∞	6	0

Chọn đỉnh 3 là đỉnh chính giữa, ta có D^4:

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

Chọn đỉnh 4 là đỉnh chính giữa, ta có D^5:

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

1.4.5 So sánh độ phức tạp giữa các thuật toán

Xét đồ thị G gồm tập đỉnh V, tập cạnh E.

	Breadth-First Search	**Dijkstra**	**Bellman-Ford**	**Floyd-Warshall**
Bài toán	Đường đi ngắn nhất một nguồn	Đường đi ngắn nhất một nguồn	Đường đi ngắn nhất một nguồn	Đường đi ngắn nhất mọi cặp đỉnh
Độ phức tạp	$O(V+E)$	$O((E+V)logV)$	$O(VE)$	$O(V^3)$
Cho phép trọng số âm	Không	Không	Có	Có

Các thuật toán trên tùy vào mục đích sử dụng cũng như cách lựa chọn cấu trúc dữ liệu phù hợp, ta có thể giảm độ phức tạp của nó xuống. Đối với đồ thị không có trọng số thì nên sử dụng BFS. Đối với đồ thị cỡ vừa và lớn thì có thể sử dụng Dijkstra hoặc Bellman-Ford để cho hiệu năng tốt. Thuật toán Floyd-Warshall hoạt động tốt nhất trên các đồ thị nhỏ (khoảng vài trăm đỉnh).

1.5 Mạng lưới và tính liên thông

Định nghĩa 1.5.1 (Mạng lưới)**.** Một **mạng (network)** là một đồ thị có hướng với **sức chứa (capacity)** $c(e)$ không âm trên mỗi cạnh e và 2 điểm phân biệt: **nguồn s (source)** và **đích t (sink)**.

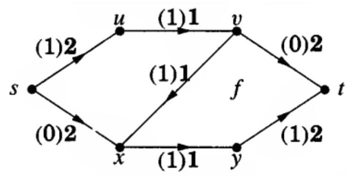
Thực tế cho thấy rất nhiều bài toán quan trọng đều có thể được mô hình hóa dưới dạng một mạng. Ví dụ như các tuyến đường giao thông, các đường ống nước, các đường dẫn dữ liệu trong máy tính hay các dòng điện trong một mạng lưới điện, ...

1.5.1 Luồng trong mạng

Định nghĩa 1.5.2 (Luồng)**.** Một **luồng (flow)** f có thể hiểu như một hàm số gán một giá trị $f(e)$ cho mỗi cạnh e. Trong đó,

$f^+(v)$ là tổng luồng trên các cạnh rời khỏi v
$f^-(v)$ là tổng luồng trên các cạnh đi vào v

Một luồng được gọi là **khả thi (feasible)** nếu nó thỏa mãn **điều kiện về sức chứa** $0 \leq f(e) \leq c(e)$ cho mỗi cạnh và **điều kiện bảo toàn** $f^+(v) = f^-(v)$ tại mỗi nút $v \notin \{s,t\}$.



Hình 1.9: Ví dụ về một luồng khả thi
(sức chứa được thể hiện bởi số in đậm và luồng giá trị là các số trong ngoặc)

Định nghĩa 1.5.3. Giá trị của một luồng (**val**(f)) là giá trị trên các cạnh chảy vào điểm đích $f^-(t) - f^+(t)$.

Ví dụ 1.5.1. Ở hình (1.9), ta có giá trị của luồng là $\text{val}(f) = f^-(t) - f^+(t) = 0 + 1 - 0 = 1$.

1.5.2 Luồng cực đại

Như ở trong định nghĩa (1.5.3), ta biết rằng mỗi luồng đều xác định một giá trị nào đó. Luồng cực đại là luồng (khả thi) làm cho giá trị đó đạt lớn nhất.

Định nghĩa 1.5.4 (Augmenting path)**.** Cho f là một luồng khả thi trên một mạng N, một **đường tăng luồng** là một path từ điểm nguồn tới điểm đích P trong đồ thị G (ẩn dưới mạng N) sao cho với mỗi $e \in E(P)$:

- nếu P đi cùng chiều với e, thì $f(e) < c(e)$.
- nếu P đi ngược chiều với e, thì $f(e) > 0$.

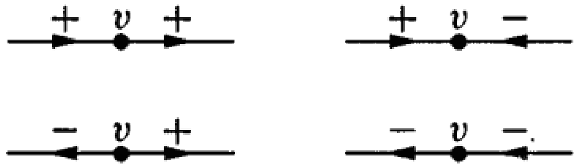
Đặt $\varepsilon(e) = f(e) - c(e)$ khi e cùng chiều với P, hoặc $\varepsilon(e) = f(e)$ nếu e ngược chiều với P. **Dung sai (tolerance)** trên P sẽ bằng $\min_{e \in E(P)} \varepsilon(e)$.

Định lý 1.5.1. Nếu P là một đường tăng luồng với dung sai z, thì việc thay đổi luồng bằng cách cộng z trên các cạnh cùng chiều với P và $-z$ trên các cạnh ngược chiều với P sẽ sinh ra một luồng mới f' (khả thi) với $\text{val}(f') = \text{val}(f) + z$.

Chứng minh. Do định nghĩa của dung sai luôn đảm bảo $0 \le f'(e) \le c(e)$ với mỗi cạnh e, nên điều kiện về sức chứa được thỏa mãn. Với điều kiện về bảo toàn trên luồng, ta chỉ cần kiểm các đỉnh trong v.

Các cạnh của P đi qua đỉnh trong của P nằm trong 1 trong 4 trường hợp sau.

Ở bất kì trường hợp nào thì sự thay đổi trên các luồng ra cũng đều tương tự với sự thay đổi trên các luồng vào. ■



Bài toán **luồng cực đại (max flow)** là một bài toán đối ngẫu với bài toán **lát cắt hẹp nhất (min cut)**. Mối liên hệ giữa hai bài toán này có thể được tìm hiểu thêm ở [1].

Dưới đây là một thuật toán do Ford-Fulkerson đề xuất, ý tưởng của nó là tìm kiếm một đường tăng luồng để làm tăng giá trị của luồng. Nếu không thể tìm được một đường nào như vậy, thì nó sẽ tìm ra một tập *min cut* ở trên với với giá trị bằng với giá trị lớn nhất của luồng (**Max flow-min cut** [1]).

Thuật toán 1.5.1 (Thuật toán Ford-Fulkerson)**.**
Đầu vào: Một luồng khả thi f trong một mạng.
Đầu ra: Một đường tăng luồng của f hoặc một tập với lực lượng bằng val(f).
Ý tưởng: Tìm những nút có thể tới được từ nút s bằng những path có dung sai dương. Nếu có thể đi đến nút t thì nó tạo thành một đường tăng luồng. Trong quá trình tìm kiếm, R là tập các nút được đánh dấu *Reached*, và S là tập con của R được đánh dấu *Searched*.
Khởi tạo: $R = \{s\}$, $S =$.
Vòng lặp: Chọn $v \in R - S$.

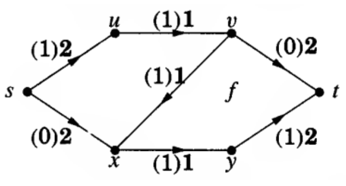
Với mỗi cạnh vw có $f(vw) < c(vw)$ và $w \notin R$ thì ta thêm w vào R.

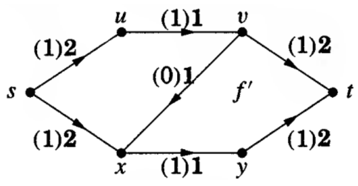
Với mỗi cạnh uv có $f(uv) > 0$ và $u \notin R$ thì thêm u vào R.

Đánh nhãn mọi đỉnh thêm vào R là *Reached*, và ghi lại đỉnh v là đỉnh xuất phát. Sau khi xem xết tất cả cạnh liên thuộc với v, ta thêm v vào S.

Nếu điểm đích t là nằm trong R, thì ta truy vết đường mà đi đến t để trả về một đường tăng luồng f và kết thúc. Nếu $R = S$, thì trả về một tập *min cut*. Nếu không có cái nào thỏa mãn, thì ta lặp lại thuật toán.

Ví dụ 1.5.2. Chúng ta sẽ áp dụng thuật toán Ford-Fulkerson với ví dụ trong hình (1.9).





Đầu tiên, tìm kiếm từ s các sức chứa còn dư tới u và tới x, đánh nhãn chúng là *reached*. Khi đó, u, $v \in R - S$. Do không còn sức chứa trên cạnh uv và xy, nên ta không thể dùng các cạnh này. Tuy nhiên, trên cạnh vx có luồng giá trị bằng $1 > 0$ nên ta đánh dấu v từ nút x. Do đó, v là phần tử duy nhất trong $R - S$ và tìm kiếm từ đỉnh này giúp ta tới được t. Truy vết lại các đỉnh trước, ta được một đường tăng luồng s, x, v, t.

Dung sai trên s, x, v, t là 1, nên luồng mới có giá trị tăng lên 1 $(= 2)$ được thể hiện ở hình bên phải. Nếu chạy thuật toán lần nữa, ta sẽ thấy rằng $R = S = \{s, u, x\}$ và ta dừng thuật toán tại đây với kết luận rằng luồng đã đạt cực đại với giá trị bằng 2.

Chương 2

Trực quan hóa đồ thị với NetworkX API

2.1 Giới thiệu về NetworkX API

NetworkX là một python package được thiết kế để tạo ra, thao tác và nghiên cứu về cấu trúc của dữ liệu dạng đồ thị. Đây là một công cụ hỗ trợ mạnh mẽ để giúp những người nghiên cứu về lý thuyết đồ thị dễ dàng hiện thực hóa các ý tưởng của họ.

Để làm quen với networkX, chúng em sẽ sử dụng bộ dữ liệu do Konect Seventh Graders cung cấp. Đây là dữ liệu dạng đồ thị có hướng chứa các đánh giá lẫn nhau của 29 học sinh khối 7. Các đỉnh đại diện cho các học sinh. Các cạnh có hướng và được đánh trọng số (1 đến 3) thể hiện rằng một học sinh đánh giá một học sinh khác như thế nào.

2.1.1 Mô hình dữ liệu

Trong NetworkX, dữ liệu dạng đồ thị được lưu trữ dưới một cấu trúc kiểu từ điển **Graph**.

```
G = nx.Graph() # or nx.DiGraph()
```

Các đỉnh có thể được truy xuất thông qua thuộc tính **nodes** của **G**. Tương tự với các cạnh được truy xuất thông qua thuộc tính **edges**, tuy nhiên ta cần truyền vào 2 đỉnh phân biệt.

```
# accessing to node 1
G.nodes[node1]
# accessing to edge between node 1 and node 2
G.edges[node1, node2]
```

Do **Graph** được cài đặt dưới dạng một từ điển, nên bất kỳ một **hashable object** nào cũng có thể là một đỉnh (**string** hoặc **tuples**, nhưng không phải **list** hay **sets**)

Sau khi khởi tạo **Graph**, ta cần nạp dữ liệu vào trong đối tượng này bằng phương thức sau:

```
G.add_edge(u, v, **attr)
# u: first node
# v: second node
# **attr: attributes of edge
```

Đối với bộ dữ liệu *senventh grades* như đã nói ở trên, chúng em đã viết một hàm để đưa dữ liệu vào trong đối tượng **Graph**.

```
G = load_data.load_seventh_grader_network()
```

2.1.2 Các thống kê cơ bản của một đồ thị

Khi đã xây dựng được dữ liệu đồ thị, một trong những điều tiên ta cần làm là kiểm tra các thống kê cơ bản của nó, chẳng hạn như: số lượng các đỉnh, số lượng các cạnh, ...

Truy vấn loại đồ thị

Trong trường hợp ta không rõ đồ thị chúng ta đang làm việc là vô hương hay có hướng thì có thể sử dụng câu lệnh sau để kiểm tra

```
type(G)
# return: networkx.classes.digraph.DiGraph
```

Truy vấn thông tin trên đỉnh

Để truy vấn tập hợp đỉnh ta dùng câu lệnh sau

```
G.nodes()
# return: NodeView((1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
   16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29))
```

Câu lệnh trên trả về một *"view"* của các đỉnh. Ngoài ra, kiểu dữ liệu trả về của câu lệnh trên là kiểu **iterable** nên ta có thể dùng câu lệnh

```
len(G.nodes())
# return: 29
```

để đếm số đỉnh trong đồ thị.

Nếu trên đồ thị có gắn *metadata*, thì ta cũng có thể xem được chúng với câu lệnh

```
list(G.nodes(data=True))[0:5] # show 5 nodes
# return: [(1, {'gender': 'male'}),
#          (2, {'gender': 'male'}),
#          (3, {'gender': 'male'}),
#          (4, {'gender': 'male'}),
#          (5, {'gender': 'male'})]
```

Câu lệnh trên trả về kiểu **NodeDataView** - một dạng của từ điển. Dựa vào tính chất của từ điển, ta có dòng duyệt qua từng cặp key - value của nó. Sau đây là hàm đếm số học sinh nam và nữ.

```
def node_metadata(G):
    from collections import Counter

    mf_counts = Counter([d["gender"] for n, d in G.nodes(data=True)])
    return mf_counts

print(node_metadata(G)
# return: Counter({'female': 17, 'male': 12})
```

Truy vấn thông tin trên cạnh

Câu lệnh sau sẽ giúp chúng ta lấy được một danh sách các cạnh trên một đồ thị

```
list(G.edges())[0:5] # show 5 edges
# return: [(1, 2), (1, 3), (1, 4), (1, 5), (1, 6)]
```

Tương tự như trên đỉnh, phương thức **G.edges()** trả về một **EdgeView** mà có thể duyệt qua được. Do vậy, ta có thể đếm được số cạnh bằng câu lệnh sau

```
len(G.edges())
# return: 376
```

Do trên cạnh cũng có thể tồn tại metadata nên bằng cách truyền tham số **data=True**, ta sẽ thu được một **EdgeDataView**

```
list(G.edges(data=True))[0:5]
# return: [(1, 2, {'count': 1}),
#          (1, 3, {'count': 1}),
#          (1, 4, {'count': 2}),
#          (1, 5, {'count': 2}),
```

```
#          (1, 6, {'count': 3})]
```

Thêm vào đó, nếu muốn truy xuất một cạnh bất kì ta có thể làm như sau

```
G.edges[15, 10] # edge between vertex 15 and 10
# return: {'count': 2}

G.edges[15, 16] # edge between vertex 15 and 16
# get an error because this edge doesn't exist.
```

Do **EdgeDataView** có dạng một từ điển nên ta có thể duyệt qua các cặp key-value của nó. Đoạn code sau đây thống kê số lần nhiều nhất mà một học sinh đánh giá học sinh khác

```
def edge_metadata(G):
    counts = [d["count"] for n1, n2, d in G.edges(data=True)]
    max_count = max(counts)
    return max_count

print(edge_metadata(G))
# return: 3
```

2.1.3 Thao tác trên đồ thị

Thêm đỉnh

```
G.add_node(node, node_data1=some_value, node_data2=some_value)
```

Giả sử trong bộ dữ liệu bị thiếu mất 2 học sinh (số 30 và số 31). Ta cần thêm vào bộ dữ liệu số 30 (nam) và số 31 (nữ).

```
def add_students(G):
    G = G.copy()
    G.add_node(30, gender="male")
    G.add_node(31, gender="female")
    return G
```

Thêm cạnh

```
G.add_edge(node1, node2, edge_data1=some_value, edge_data2=some_value)
```

Giữa 2 học sinh 30 và 31 ta vừa thêm, giả sử họ rất thân với nhau, nên ta thêm một cạnh giữa node 30 và node 31.

```python
def add_student_rating(G):
    G = G.copy()
    G.add_edge(30, 31, count=3)
    G.add_edge(31, 30, count=3)
    return G
```

Khai phá dữ liệu

Giả sử trong số các học sinh trong bộ dữ liệu, có bạn rất yêu mến một bạn học sinh khác nhưng lại không được đáp lại. Nhiệm vụ ở đây là tìm ra các học sinh có một tình bạn "đơn phương" như vậy

```python
def unrequitted_friendships(G):
    losers = []
    for n1, n2 in G.edges():
        if not G.has_edge(n1, n2):
            losers.append((n1, n2))
    return losers
```

Tuy nhiên, có thể thấy rằng trong mạng lưới học sinh này không có bất kì ai như vậy.

2.2 Trực quan hóa đồ thị

Ai đó đã từng nói:

"A picture is worth a thousand words"

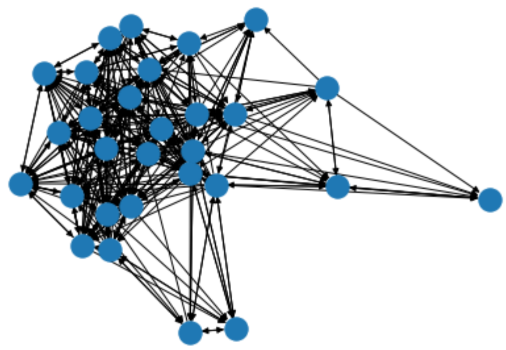
Đặc biệt trong lĩnh vực lý thuyết đồ thị, câu nói trên càng được khẳng định tính đúng đắn của nó. Việc sử dụng các hình ảnh hợp lý, có thể giúp tìm ra cấu trúc ẩn sau đồ thị, cách mà các thành phần trong đồ thị giao tiếp với nhau hay đôi khi là dễ dàng chứng minh một mệnh đề hoặc định lý nào đó.

Vẫn sử dụng bộ dữ liệu *seventh graders* ở phần (2.1). Dưới đây là một số các cách để biểu diễn mạng lưới này lên máy tính thông qua networkX.

2.2.1 Hairball

Biểu đồ thể hiện mối liên kết giữa các đỉnh qua cạnh (có hướng hoặc vô hướng) là một trong những biểu đồ phổ biến nhất. Các đỉnh thường được biểu diễn dưới dạng các hình tròn và với cạnh thì là đoạn thẳng (có hướng hoặc vô hướng) nối giữa 2 đỉnh có liên kết với nhau.

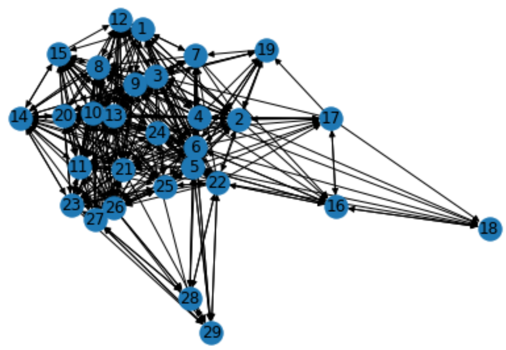
```
nx.draw(G)
```



NetworkX vẽ đồ thị dựa trên cơ chế chọn một đỉnh đầu tiên và vẽ các đỉnh, liên kết còn lại xung quanh đó. Do việc chọn đỉnh là ngẫu nhiên, nên ta thường sẽ không nhận được cùng một hình ở mỗi lần chạy. Tuy nhiên, có thể thấy rằng các đỉnh có xu hướng liên kết mạnh hơn thì sẽ được đặt gần nhau và tạo thành một cụm.

Ngoài ra, nếu đồ thị và các nhãn đủ nhỏ, ta có thể dùng câu lệnh sau để hiển thị một số thông tin lên đồ thị.

```
nx.draw(G, with_labels=True)
```

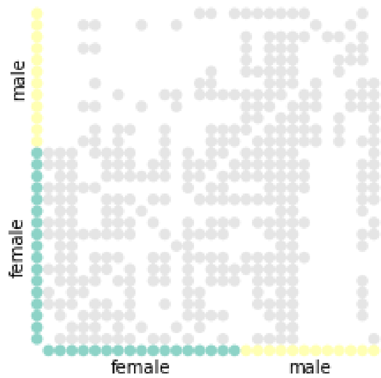


Tuy nhiên, với một đồ thị cỡ lớn thì cách biểu diễn này không thực sự là một sự lựa chọn tốt.

2.2.2 Matrix plot

Như trong chương 1 đã thảo luận, một trong những cách mà máy tính định nghĩa được một đồ thị là qua dạng ma trận.

```
nv.matrix(G, group_by="gender", node_color_by="gender")

from nxviz import annotate
annotate.matrix_group(G, group_by="gender")
```



Đối với bộ dữ liệu này, các đỉnh nằm trên các trục Ox, Oy là các học sinh và được nhóm theo giới tính bởi 2 màu (xanh là nữ, vàng là nam). Các hình tròn màu xám biểu thị giữa 2 học sinh có mối liên hệ với nhau (thông qua *đánh giá*). Ngoài ra, từ ma trận trên có thể thấy rằng:

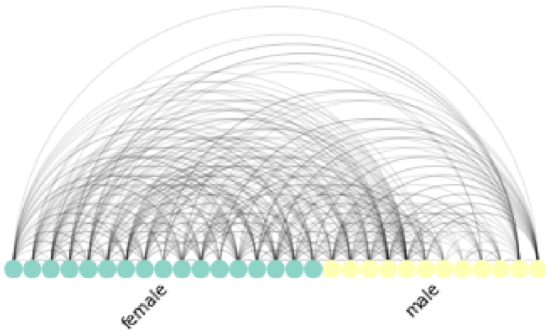
- không có học sinh nào tự bầu chọn cho chính mình (không có chấm nào trên đường chéo).
- ma trận trên khồng đối xứng do là đây là đồ thị có hướng

Cách biểu diễn trên cũng có một hạn chế là ta không thể biết được mức độ đánh giá của một học sinh này đối với một học sinh khác là như thế nào.

2.2.3 Arc plot

Arc plot là một dạng biểu diễn đồ thị đặc biệt. Trong đó, các đỉnh được biểu diễn bằng các thực thể hình tròn và các cạnh nối với nhau thể hiện mối quan hệ giữa 2 thực thể. Các đỉnh (hay thực thể) được sắp xếp trên một trục duy nhất và các liên kết thể hiện bởi các cạnh dạng vòng cung.

```
nv.arc(G, node_color_by="gender", group_by="gender")
annotate.arc_group(G, group_by="gender")
```

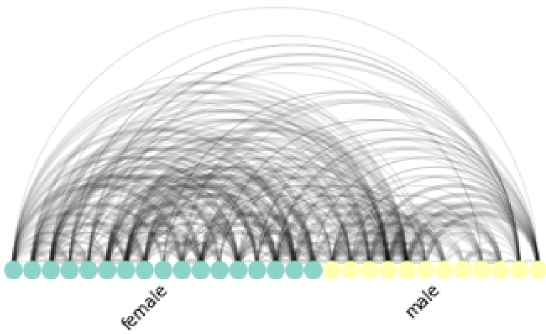


Kiểu biểu diễn arc plot có 2 điểm mạnh sau

- Nếu các đỉnh được sắp xếp tốt, arc plot có thể chỉ rõ các cụm hoặc các cầu.
- Việc gán nhãn các đỉnh là dễ dàng hơn so với những kiểu biểu diễn thông thường như hairball.

Một biến thể khác của arc plot là ta có thể làm cho các cạnh của nó lớn hơn nếu trong đồ thị là có trọng số.

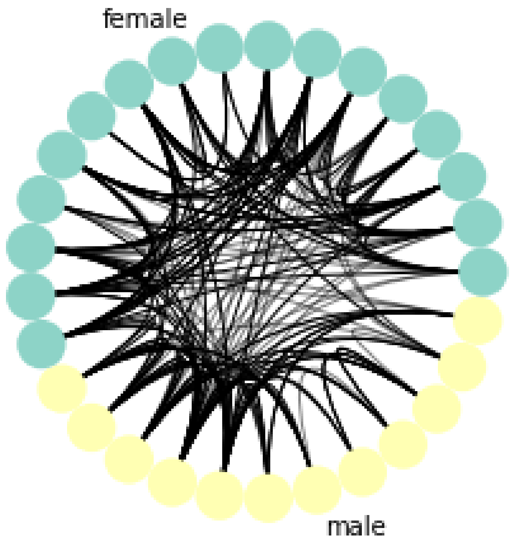
```
nv.arc(G, node_color_by="gender", group_by="gender", edge_lw_by="count")
annotate.arc_group(G, group_by="gender")
```



2.2.4 Circos plot

Circos là một sự cải tiến từ arc plot. Trong kiểu biểu diễn này, các đỉnh tạo với nhau thành một vòng tròn bao quanh các liên kết giữa các đỉnh.

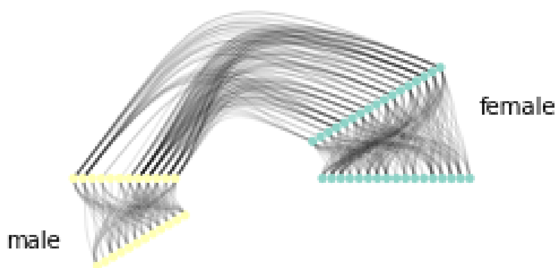
```
nv.circos(G, group_by="gender", node_color_by="gender",
    edge_alpha_by="count")
annotate.circos_group(G, group_by="gender")
```



Cách biểu diễn dạng circos được sử dụng khá phổ biến và rộng rãi do nó đẹp và nó là lý tưởng để nhận biết mối quan hệ giữa các thực thể và cấu trúc ẩn sau của chúng.

2.2.5 Hive plot

```
nv.hive(G, group_by="gender", node_color_by="gender")
annotate.hive_group(G, group_by="gender")
```



Như trên hình có thể thấy, hive plot nhóm các đỉnh về 2 hoặc 3 trục đồng tâm. Trong trường hợp trên, các học sinh nam được đặt trên một trục và các bạn nữ ở trục còn lại.

Trước tiên, các cạnh được xây dựng dựa trên liên kết giữa các học sinh của 2 nhóm khác nhau. Sau đó, tiếp tục xây dựng các cạnh bên trong 1 nhóm bằng cách nhân bản các trục nam và nữ. Như trên hình, ta thấy *male* có 2 nhóm với cùng số thực thể và *female* cũng vậy.

Chương 3

Ứng dụng trong mạng lưới giao thông của Uber

Uber Technologies, Inc. (Uber) là một công ty cung cấp dịch vụ vận chuyển của Mỹ và có mặt trên toàn cầu. Như rất nhiều công ty lớn khác, để phục vụ cho mục đích cộng đồng, Uber cung cấp một lượng lớn dữ liệu vận chuyển. Trong bài báo cáo này, chúng em sử dụng tập dữ liệu do Uber cung cấp trên thành phố San Francisco của Mỹ trong quý 4 năm 2017 để làm quen với mô hình đồ thị trong thực tế.

Chuẩn bị dữ liệu:

1. Dữ liệu do Uber cung cấp có thể được tìm thấy tại *"Uber Movement"*.
2. Tìm thành phố San Francisco, chọn travel times và nhấn download.
3. Nếu download thành công, tên file sẽ có dạng san_francisco-censustracts-2017-4-All-MonthlyAggregate.csv
4. Ngoài ra, ta cũng cần download file Geo Boundaries từ trang web đó.

Mô rả dữ liệu: Nhìn chung, dữ liệu chúng ta có được chia làm 2 file: *Travel times* và *Geo boundaries*.

- File thứ nhất *Travel times* gồm các thống kê của Uber về thời gian di chuyển giữa các cặp vị trí (points) trong khu vực San Francisco. Các vị trí trên bản đồ được biểu diễn bởi các số ID duy nhất.
- File thứ hai *Geo boundaries* cho biết mối liên hệ giữa các ID và khu vực trên San Francisco. File này chứa kinh độ và vĩ độ của các đỉnh trên mỗi khu vực (hình đa giác). Ví dụ, nếu một khu vực được biểu diễn bởi một đa giác gồm 5 đỉnh thì dữ liệu của ta sẽ có dạng một ma trận cỡ 5×2 với các hàng là chỉ số của các đỉnh và 2 cột là vị trí vĩ độ và kinh độ.

3.1 Xây dựng mô hình đồ thị

Sử dụng thư viện *pandas*, đọc 2 file mà ta đã tải về vào trong 2 dataframe *data* và *data_json*.

```
data = pd.read_csv(
    datasets /
        "uber/san_francisco-censustracts-2017-4-All-MonthlyAggregate.csv",
    header=None,
    sep=","
)
json_data = pd.read_json(datasets /
    "uber/san_francisco_censustracts.json")
```

Mặc dù có nhiều cách di chuyển từ địa điểm A đến B và từ B về A, nhưng do chỉ quan tâm đến thời gian trung bình nên chúng em sẽ áp dụng mô hình đồ thị vô hướng.

```
G = nx.Graph()
```

Các đỉnh trên đồ thị tương ứng là các địa điểm trên bản đồ. Ngoài ra, trên mỗi đỉnh còn có 2 thuộc tính.

1. Display name: địa chỉ con đường qua khu vực đó.
2. Location: trung bình tọa độ các đỉnh của đa giác (vector 2 chiều).

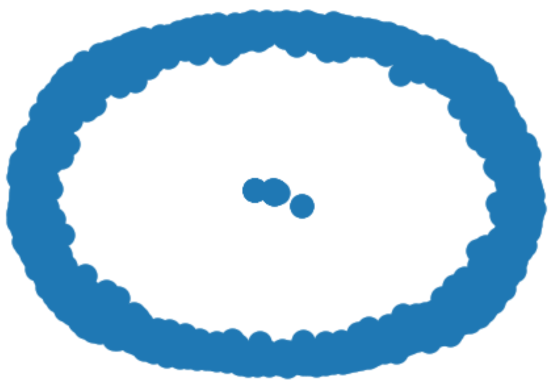
```
node_dict = {}
for node_info in json_data["features"]:
    node_id = int(node_info["properties"]["MOVEMENT_ID"])
    if node_id in node_dict:
        pass
    else:
        dis_name = node_info["properties"]["DISPLAY_NAME"]
        location =
            np.mean(np.array(node_info["geometry"]["coordinates"][0][0]),
            axis=0)
        node_dict[node_id] = {"DISPLAY_NAME": dis_name, "Location":
            location}
        G.add_node(node_id, name=dis_name, Location=location)
```

Chúng em xử lý các cạnh bằng cách xét 2 điểm kề nhau, tính tổng trọng số trên mọi con đường qua 2 điểm đó và chia cho số đường.

```
data_row = data.shape[0]
edge_dic = {}
```

```
    for idx in range(data_row):
        if int(data['month'][idx]) != 12:
            continue
        edge_w = float(data["mean_travel_time"][idx])
        source_id = int(data["sourceid"][idx])
        destin_id = int(data['dstid'][idx])
        if (source_id, destin_id) in edge_dic:
            edge_dic[(source_id, destin_id)][0] += edge_w
            edge_dic[(source_id, destin_id)][1] += 1
        elif (destin_id, source_id) in edge_dic:
            edge_dic[(destin_id, source_id)][0] += edge_w
            edge_dic[(destin_id, source_id)][1] += 1
        else:
            edge_dic[(source_id, destin_id)] = [edge_w, 1]

    # add edges to graph
    for key, item in edge_dic.items():
        w = item[0]/item[1]
        G.add_edge(key[0], key[1], weight=w)
```

Đồ thị lúc này có dạng như sau:



Do chỉ quan tâm đến các thành phần liên thông, nên chúng em sẽ loại bỏ các điểm cô lập và chỉ giữ lại thành phần lớn nhất

```
Gcc = max((G.subgraph(c) for c in nx.connected_components(G)), key=len)

print("Number of nodes are:", nx.number_of_nodes(Gcc)) # 1898
print("Number of edges are:", nx.number_of_edges(Gcc)) # 321703
```

Sau khi đã tiền xử lý xong, ta có thể thấy **Gcc** là một đồ thị liên thông với 1898 đỉnh và 321703 cạnh. Đây cũng là đồ thị mà chúng em tập trung phân tích ở những phần sau.

3.2 Bài toán người du lịch

Do thuộc lớp bài toán NP-C nên việc giải quyết bài toán người du lịch theo những thuật toán truyền thống là rất khó khăn. Vậy nên, ở đây chúng em xin đề xuất thuật toán *2-approximation*[4] thuộc vào lớp các phương pháp xấp xỉ để giải bài toán người du lịch.

Bước 1: Trước tiên, để có thể áp dụng thuật toán *2-approximation* ta cần kiểm tra giả thuyết bất đẳng thức tam giác trên **Gcc**. Việc kiểm tra tất cả các tam giác trong đồ thị là không cần thiết, vậy nên chúng em chỉ lấy mẫu khoảng 1000 tam giác. Nếu số lượng tam giác thỏa mãn đủ lớn, ta sẽ ngầm hiểu rằng kết quả sinh ra bởi thuật toán này là một xấp xỉ tốt.

```python
import random

bound = max(node_dict.keys())
sample_num = 0
triangle_num = 0
used = set()

while sample_num < 1000:
    v1 = random.randint(1, bound)
    v2 = random.randint(1, bound)
    v3 = random.randint(1, bound)

    if (v1 != v2 != v3) \
        and (v1, v2) in edge_dict and (v2, v3) in edge_dict and (v3, v1)
            in edge_dict and \
            (v1, v2, v3) not in used:
        used.add((v1, v2, v3))
        sample_num += 1

        d12 = edge_dict[(v1, v2)][0] / edge_dict[(v1, v2)][1]
        d23 = edge_dict[(v2, v3)][0] / edge_dict[(v2, v3)][1]
        d31 = edge_dict[(v3, v1)][0] / edge_dict[(v3, v1)][1]

        if (d12 + d23 > d31) and (d23 + d31 > d12) and (d12 + d31 > d23):
            triangle_num += 1

print("The percentage of triangles in the graph is",
    triangle_num/sample_num * 100)
# The percentage of triangles in the graph is 93.2
```

Bước 2: Từ đồ thị **Gcc**, ta sẽ xây dựng một cây con bao trùm nhỏ nhất. Có một vài thuật toán để xây dựng cây con bao trùm nhỏ nhất, tuy nhiên chúng em sẽ áp dụng thuật toán của Kruskal do tác giả của cuốn [1] đề xuất.

```
MST = nx.minimum_spanning_tree(Gcc, algorithm='kruskal')
```

Với **MST** đã xây dựng ở trên, ta ghi lại một số địa chỉ kết nối với nhau thông qua các cạnh trên cây con bao trùm nhỏ nhất.

```
for point in sorted(list(nx.edges(MST))[0:5]):
    print(node_dict[point[0]]["DISPLAY_NAME"], "----",
        node_dict[point[1]]["DISPLAY_NAME"])
# 400 Northumberland Avenue, Redwood Oaks, Redwood City ---- 1500 Oxford
    Street, Palm Park, Redwood City
# 400 Northumberland Avenue, Redwood Oaks, Redwood City ---- 100 Fifth
    Avenue, South Fair Oaks, Redwood City
# 18300 Sutter Boulevard, Morgan Hill ---- 17300 Lotus Way, Morgan Hill
# 18300 Sutter Boulevard, Morgan Hill ---- 1900 Alpet Drive, Morgan Hill
# 3200 Huntsman Drive, Rosemont Park, Sacramento ---- 8900 Cal Center
    Drive, Sacramento
```

Dựa vào google map, có thể thấy các kết quả trên khá đáng tin. Với 2 địa chỉ gần nhau thì sẽ có thời gian di chuyển nhỏ tương đương với trọng số trên cạnh cũng nhỏ. Đây chính xác là một trong những tính chất của cây con bao trùm nhỏ nhất.

Bước 3: Xây dựng một đồ thị đa cạnh từ **Gcc** bằng cách gấp đôi số cạnh trên mọi $uv \in E(\textbf{Gcc})$

```
Multi_G = nx.MultiDiGraph()
Multi_G.add_nodes_from(MST)
path = []

for edge in MST.edges:
    w = MST.edges[edge]['weight']
    Multi_G.add_edge(edge[0], edge[1], weight=w)
    Multi_G.add_edge(edge[1], edge[0], weight=w)
```

Bước 4: Tìm một chu trình Eulerian trong **Multi_G**. Từ đó xây dựng phương án di chuyển cho bài toán người du lịch.

```
Euler_circle = nx.eulerian_circuit(multi_G)

travel_length = 0
idx = 0

for edge in Euler_circle:
    if idx == 0:
        start = edge[0]
    if edge[1] == start:
```

```
            print(edge[0])
            dst = edge[0]
            path.append(dst)
            print('-'*30)
            break
        travel_length += multi_G.edges[edge[0], edge[1], 0]['weight']
        path.append(edge[0])
        print(edge[0], "->", end=' ')
        idx += 1

    print("Approximate TSP cost is", travel_length)
    # return: Approximate TSP cost is 440276.2400000001
```

Bước 5: Đánh giá tính hiệu quả của thuật toán. Gọi ρ là hiệu suất về mặt chi phí của phương án xấp xỉ so với phương án tối ưu. Ta có:

$$\rho = \frac{\text{Chi phí của phương án xấp xỉ}}{\text{Chi phí của phương án tối ưu}}$$

Mặt khác,

$$\begin{aligned}\text{chi phí của phương án xấp xỉ} &\le \text{chi phí trên chu trình Euler}\\ &\le 2\times\text{chi phí trên cây bao trùm nhỏ nhất}\\ &\le 2\times\text{chi phí trên phương án tối ưu nhất}\end{aligned}$$

Do vậy,

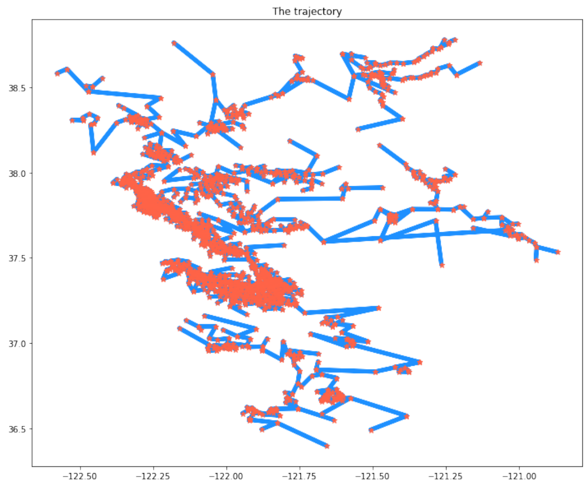
$$\rho \le \frac{\text{chi phí trên chu trình Euler}}{\text{chi phí trên cây bao trùm nhỏ nhất}}$$

```
opt_length = 0

for edge in MST.edges:
    opt_length += MST.edges[edge]['weight']

print("The upper bound on the empirical performance of the approximation
    algorithm is", travel_length/opt_length)
# The upper bound on the empirical performance of the approximation
    algorithm is 1.5217849499512972
```

Bước 6: Trực quan hóa phương án di chuyển xấp xỉ tối ưu của bài toán người du lịch.

```
X = [] # latitude
Y = [] # longitude

for node in path:
    X.append(node_dict[node]['Location'][0])
```

```
        Y.append(node_dict[node]['Location'][1])

    plt.figure(figsize=(12, 10))
    plt.plot(X, Y, linewidth=5, color='dodgerblue')
    plt.plot(X, Y, '*', markersize=7, color='tomato')
    plt.title('The trajectory')
```

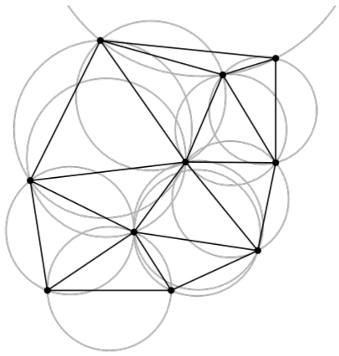


3.3 Khái quát hóa bản đồ giao thông

Khái quát hóa bản đồ là quá trình tạo nên một bản đồ dễ đọc từ một bộ dữ liệu địa lý chi tiết. Điều này đạt được bằng cách loại bỏ một số chi tiết và chỉ giữ lại những thứ quan trọng. Ví dụ như loại bỏ một số con đường (cạnh), khu vực; kết hợp một số khu vực lân cận; phóng đại các đối tượng địa lý; giảm kích thước các con đường, khu vực; dịch chuyển để đảm bảo khoảng cách giữa các đối tượng là phù hợp. Các phép toán trên có thể được tham khảo trong [5], và sâu hơn trong một mạng lưới giao thông thực sự [7].

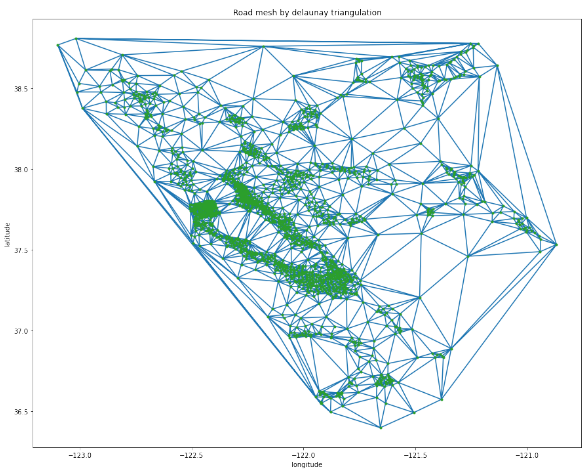
Một trong những phương thức được sử dụng để khái quát hóa bản đồ là dùng *Delaunay Triangulation* (thuật toán về nó được thảo luận trong [6]).

Định nghĩa 3.3.1 (Tam giác phân Delaunay)**.** *Cho một tập P các điểm rời rạc, và một tập cạnh nối các điểm đó thành một tập các tam giác. Tập tam giác đó được gọi là tam giác phân Delaunay nếu không tồn tại bất kì điểm nào nằm trong 1 đường tròn ngoại tiếp tam giác bất kỳ.*



Do cài đặt thuật toán khá khó khăn nên chúng em sẽ sử dụng gói có sẵn trong python.

```
from scipy.spatial import Delaunay

nums_of_area = len(node_dict)
vertices = nx.nodes(Gcc)
mean_coordinates = np.array([elm['Location'] for elm in
    list(node_dict.values())])
mean_coordinates_gcc = mean_coordinates[[x-1 for x in vertices], :]
triangulation = Delaunay(mean_coordinates_gcc)

plt.figure(figsize=(15, 12))
plt.triplot(mean_coordinates_gcc[:, 0], mean_coordinates_gcc[:, 1],
    triangulation.simplices)
plt.plot(mean_coordinates_gcc[:, 0], mean_coordinates_gcc[:, 1], '.')
plt.xlabel('longitude')
plt.ylabel('latitude')
plt.title('Road mesh by delaunay triangulation')
```



Gọi đồ thị sau khi áp dụng thuật toán tam giác phân Delaunay là G_Δ. Các đỉnh được thể hiện bởi các chấm màu xanh lá cây và các cạnh là những đoạn thẳng màu xanh nước biển. Các tam giác trong G_Δ cũng chính là số lượng những tam giác thỏa mãn bất đẳng thức tam giác mà ta nhắc đến trong bài toán người du lịch.

```
G_delta = nx.Graph()

used_nodes = triangulation.simplices
edges_delta = set()

for row in range(used_nodes.shape[0]):
    for idx in range(3):
        node1 = list(vertices)[used_nodes[row][idx]]
        node2 = list(vertices)[used_nodes[row][(idx+1) % 3]]
        if (node1, node2) not in edges_delta and (node2, node1) not in
            edges_delta:
            edges_delta.add((node1, node2))
            G_delta.add_edge(node1, node2)

print("Number of nodes in G_delta is", nx.number_of_nodes(G_delta))
print("Number of edges in G_delta is", nx.number_of_edges(G_delta))
# Number of nodes in G_delta is 1898
# Number of edges in G_delta is 5680
```

Sau khi có được G_Δ, ở các phần sau khi thực hiện tính toán về luồng ta sẽ sử dụng nó thay cho Gcc.

3.4 Lưu lượng giao thông

Giả sử ta có các thông tin sau

- Mỗi một độ trong kinh độ và vĩ độ tương đương với 69 dặm Anh.
- Chiều dài của mọi chiếc xe là 5 mét $\approx$ 0.003 dặm.
- Mỗi xe có một khoảng cách an toàn tối thiểu là 2 giây đối với chiếc xe kề với nó.
- Mọi con đường đều có 2 làn.

Với giả thuyết rằng không có tắc đường, và xem xét luồng giao thông mà ta tính toán được như sức chứa cực đại trên con đường đó.

Câu hỏi: Làm thế nào để tính toán được lưu lượng xe trên một giờ trên mỗi con đường?

Dựa vào các giả thuyết trên, ta có được các hằng số sau

```
miles_per_degree = 69
car_length = 0.003
safe_time = 2/3600
lanes = 2
```

Ngoài ra, ta cũng khai báo thêm 2 biến để lưu lại kết quả tính được.

```
traffic_flow = {}
fake_flow = []
```

Sở dĩ, ta có biến `fake_flow` là do chúng em đang xét trên đồ thị G_Δ nên giữa 2 địa điểm có thể tồn tại một con đường không nằm trong tập dữ liệu.

Để tính toán lưu lượng giao thông trên mọi con đường trong G_Δ, ta cần duyệt qua mọi cạnh của nó. Nếu có một cạnh giữa 2 đỉnh u và v trong G_Δ. ta sẽ đặt `mean_time` là thời gian di chuyển trung bình giữa 2 điểm (đơn vị theo giây). Nếu không tồn tại một cạnh như vậy, đơn giản là ta đặt `mean_time` bằng một số rất lớn giữa 2 điểm đó. Sau đó, sử dụng các công thức sau để tính toán lưu lượng giao thông trên mỗi con đường.

- Khoảng cách giữa các đỉnh $= \sqrt{(u_x - v_x)^2 + (u_y - v_y)^2}$
- tốc độ của xe trên 1 đoạn đường $= \frac{\text{khoảng cách trên đoạn đường đó}}{\text{thời gian di chuyển trung bình trên đoạn đường đó}}$.
- khoảng cách giữa 2 xe $=$ tốc độ của xe $\times$ khoảng cách an toàn.
- Lưu lượng xe $= \frac{\text{tốc độ xe}}{(\text{chiều dài xe} + \text{khoảng cách giữa 2 xe})} \times$ số làn.

Sau khi biết được lưu lượng xe cộ trên từng con đường trong G_Δ, ta thêm các giá trị đó vào các cạnh như là một thuộc tính trên cạnh đó (tương đương với sức chứa của con đường).

```
edges_delta = nx.edges(G_delta)
for edge in list(edges_delta):
    u = edge[0]
    v = edge[1]

    mean_time = 1e8   # threshold
    if (u, v) in edge_dict:
        mean_time = edge_dict[(u,v)][0] / edge_dict[(u,v)][1]
    elif (v, u) in edge_dict:
        mean_time = edge_dict[(v,u)][0] / edge_dict[(v,u)][1]

    miles = np.sqrt(
        np.square(abs(node_dict[u]['Location'][0] -
            node_dict[v]['Location'][0]) * miles_per_degree) +
```

```
            np.square(abs(node_dict[u]['Location'][1] -
                node_dict[v]['Location'][1]) * miles_per_degree)
        )

        car_speed = miles / (mean_time/3600)
        safety_distance = car_speed * safe_time
        car_flow = (car_speed / (car_length + safety_distance)) * lanes

        G_delta.add_edge(u, v, capacity=car_flow)
        traffic_flow[(u,v)] = car_flow
        if mean_time == 1e8:
            fake_flow.append(car_flow)
```

Ta có thể kiểm tra một vài kết quả qua đoạn code dưới đây.

```
from itertools import islice

def take(n, iterable):
    return list(islice(iterable, n))

n_roads = take(5, traffic_flow.items())
for i in range(len(n_roads)):
    print(f"{n_roads[i][0]}: {n_roads[i][1]}")

# (1444, 2017): 0.5261317608739441
# (1444, 583): 1.3213564449322088
# (1444, 1231): 1.1921773439325514
# (1444, 793): 0.954980147325594
# (1444, 38): 3324.9516604562614
```

Ý nghĩa của kết quả trên, lấy ví dụ trên đoạn đường giữa điểm 1444 và điểm 38, tức là trung bình trong một giờ có khoảng 3325 xe lưu hành trên đoạn đường đó.

Ở 4 kết quả đầu tiên có thể thấy rằng số lượng xe lưu thông trên đường chỉ khoảng 1 đến 2 xe trên một giờ. Và như đã nói ở trên, chúng ta đang làm việc với G_Δ, nên có thể sẽ tồn tại những cung đường ảo. Ta nên đặt một giá trị giới hạn để tách biệt những con đường này.

```
fake_flow_threshold = max(fake_flow)
print("The fake flow are all under", fake_flow_threshold) # 3.5027891
print("Number of fake flows are:", len(fake_flow)) # 392
```

Vậy tất cả những con đường mà có lưu lượng giao thông dưới 3.5027891 đều là những con đường ảo và có tổng cộng tất cả 392 con đường như vậy.

3.5 Bài toán về luồng cực đại

Xem xét 2 địa điểm sau:

- Địa điểm nguồn: 100 Campus Drive, Stanford
- Địa điểm đích: 700 Meder Street, Santa Cruz

Tính số lượng xe cộ tối đa có thể đi trong một giờ từ Stanford tới UCSC. Đồng thời, tính số lượng các path phân biệt (theo cạnh) giữa 2 địa điểm trên.

Để trả lời câu hỏi thứ nhất, trước tiên chúng em sẽ xác định các nút đại diện cho Stanford và UCSC.

```python
Stanford_ID = 0
UCSC_ID = 0

for key in node_dict:
    if node_dict[key]['DISPLAY_NAME'] == '100 Campus Drive, Stanford':
        Stanford_ID = key
        print("Stanford node is", Stanford_ID)
    elif node_dict[key]['DISPLAY_NAME'] == '700 Meder Street, Santa Cruz':
        UCSC_ID = key
        print("UCSC node is", UCSC_ID)

# Stanford node is 2607
# UCSC node is 1968
```

Tiếp theo, chúng em sẽ sử dụng một hàm có sẵn của *networkx* để tìm luồng cực đại.

```python
flow_value, flow_dict = nx.maximum_flow(G_delta, UCSC_ID, Stanford_ID)
print("The maximum flow between Stanford and UCSC is", flow_value)
# The maximum flow between Stanford and UCSC is 14866.477294089982
```

Hàm `nx.maximum_flow` ở trên trả về một biến giá trị (giá trị của luồng cực đại) và một từ điển chứa giá trị của luồng cực đại đó gán cho mỗi cạnh.

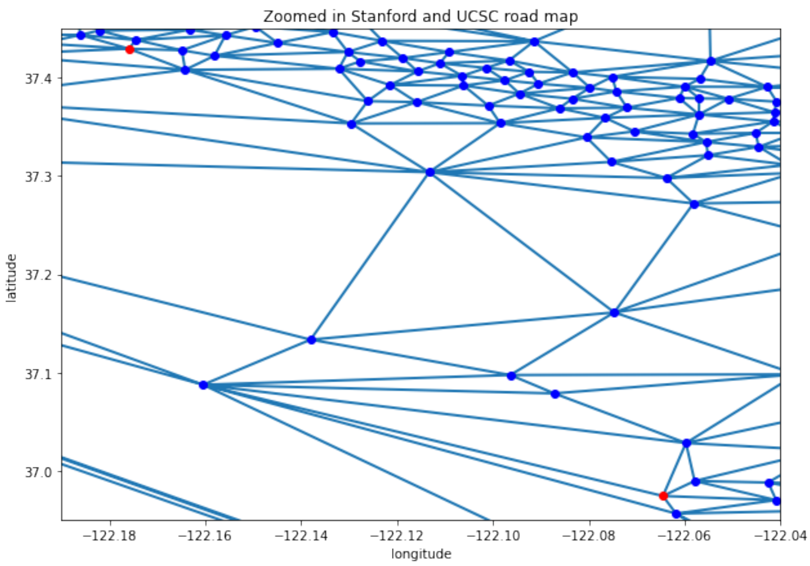
Tương tự, để tìm các path phân biệt (theo cạnh), chúng em dùng hàm `nx.edge_disjoint_paths`.

```python
edge_disjoint_paths = nx.edge_disjoint_paths(G_delta, UCSC_ID, Stanford_ID)

count = 0
for path in edge_disjoint_paths:
    count += 1
```

```
        print(path)
    print("Number of edge-disjoint path is", count)

    # [1968, 2241, 1980, 2242, 744, 1869, 1363, 2240, 2607]
    # [1968, 1424, 1431, 1980, 1763, 1762, 1209, 1733, 1725, 2607]
    # [1968, 1431, 1989, 938, 744, 1737, 1736, 1726, 2607]
    # [1968, 748, 2241, 1171, 1955, 1763, 1737, 1363, 2607]
    # [1968, 1980, 1955, 2458, 1210, 1762, 1736, 2607]
    # Number of edge-disjoint path is 5
```

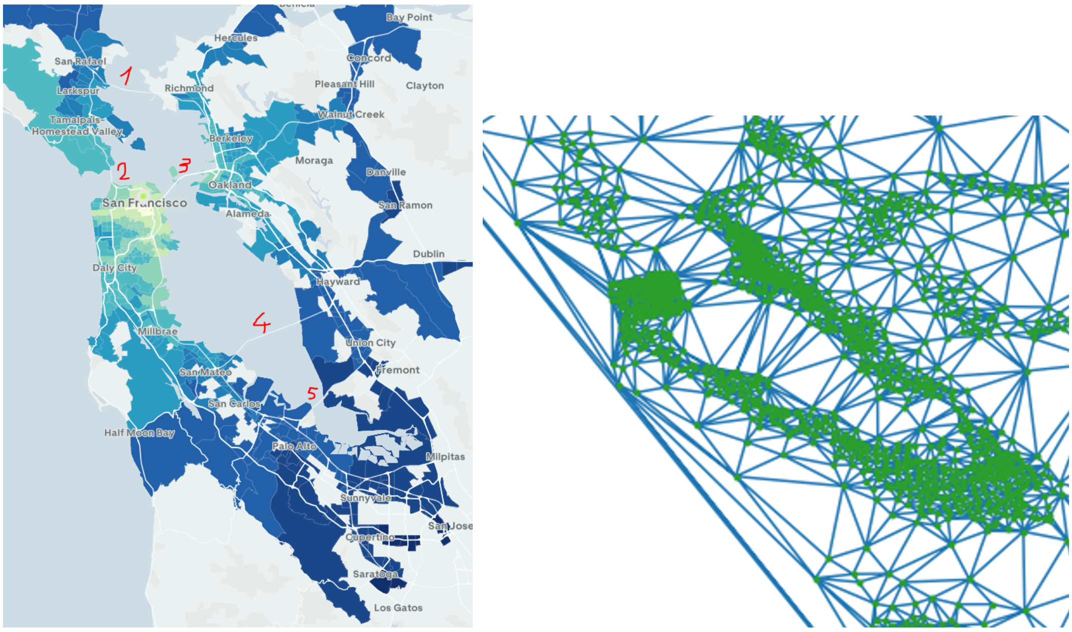
Để dễ dàng hình dung và tưởng tượng, ta có thể phóng to bản đồ khu vực Stanford và UCSC.

```
plt.figure(figsize=(10, 7))
plt.plot(Stanford_loc[0], Stanford_loc[1], 'o', color='red')
plt.plot(UCSC_loc[0], UCSC_loc[1], 'o', color='red')

for node in node_dict:
    if node != Stanford_ID and node != UCSC_ID:
        plt.plot(node_dict[node]['Location'][0],
            node_dict[node]['Location'][1], 'o', color='blue')

plt.triplot(mean_coordinates_gcc[:, 0], mean_coordinates_gcc[:, 1],
    triangulation.simplices)
plt.xlim(-122.19, -122.04)
plt.ylim(36.95, 37.45)

plt.xlabel('longitude')
plt.ylabel('latitude')
plt.title('Zoomed in Stanford')
plt.show()
```



3.6 Tối ưu mạng giao thông

Như chúng em đã nhắc đến ở các phần trước, trong đồ thị G_Δ có một số những con đường không thật sự tồn tại. Ví dụ như ở hình bên trái dưới đây, ta thấy rằng chỉ có 5 cây cùng bắc qua vùng vịnh San Francisco, nhưng ở trong đồ thị G_Δ lại có rất nhiều cây cầu như vậy.



Hình 3.1: Những cây cầu ở San Francisco

Việc cần làm trong phần này là cắt giảm số lượng các cạnh ảo nhằm tối ưu đồ thị. Ở trong phần 3.4, ta biết được rằng một số lượng lớn các cạnh ảo đều có lưu lượng xe cộ trên giờ rơi vào khoảng dưới 3.5. Vậy nên ta có thể đặt ngưỡng phát hiện cạnh giả là 5 và từ đó xây dựng lên $\tilde{G}_\Delta$.

```
G_tilde = nx.Graph()
G_tilde.add_nodes_from(G_delta)

fake_flow_threshold = 5
for edge in G_delta.edges():
    capacity = G_delta.edges[edge]['capacity']
    if capacity <= fake_flow_threshold:
        continue
    G_tilde.add_edge(edge[0], edge[1], capacity=capacity)

print("Number of edges in trimmed graph is", nx.number_of_edges(G_tilde))
# Number of edges in trimmed graph is 5288
```

Để chắc chắn rằng ta không loại bỏ một số cạnh quan trọng, ta xét tọa độ của các cây cầu bắc qua vịnh San Francisco sau đây:

- Golden Gate Bridge: [[-122.475, 37.806], [-122.479, 37.83]]
- Richmond, San Rafael Bridge: [[-122.501, 37.956], [-122.387, 37.93]]
- San Mateo Bridge: [[-122.273, 37.563], [-122.122, 37.627]]
- Dambarton Bridge: [[-122.142, 37.486], [-122.067, 37.54]]
- San Francisco - Oakland Bay Bridge: [[-122.388, 37.788], [-122.302, 37.825]]

Chúng em sẽ xác định vị trí các cây cầu này trên $\tilde{G}_{\Delta}$ và kiểm tra xem chúng có bị loại bỏ hay không.

Đàu tiên là khởi tạo các biến lưu tọa độ các cây cầu đã nói ở trên.

```
GGB   = np.array([[-122.475, 37.806], [-122.479, 37.83]])
RSRB  = np.array([[-122.501, 37.956], [-122.387, 37.93]])
SMB   = np.array([[-122.273, 37.563], [-122.122, 37.627]])
DB    = np.array([[-122.142, 37.486], [-122.067, 37.54]])
SFOBB = np.array([[-122.388, 37.788], [-122.302, 37.825]])

bridges = [GGB, RSRB, SMB, DB, SFOBB]
bridge_on_map = []
```

Các cây cầu trên được xác định bởi 2 cặp tọa độ (chân cầu). Tuy nhiên, do các tọa độ đó được ước lượng trên bản đồ thật, nên ta cần xác định các điểm gần với các tọa độ đó để xét làm chân cầu. Dưới đây là một hàm để xác định đỉnh gần nhất đối với đỉnh `target` trong một danh sách sách các đỉnh gần nó.

```
def get_nearest_node(target, node_list):
    distances = []
    for i in range(len(node_list)):
        x = node_dict[node_list[i]]['Location'][0]
        y = node_dict[node_list[i]]['Location'][1]
        dist = (target[0]-x)**2 + (target[1]-y)**2
        distances.append(dist)
    return node_list[np.argsort(distances)[0]]
```

Sau đó, duyệt qua danh sách `bridges` đã khai báo ở trên để xác định tọa độ các cây cầu trong $\tilde{G}_{\Delta}$.

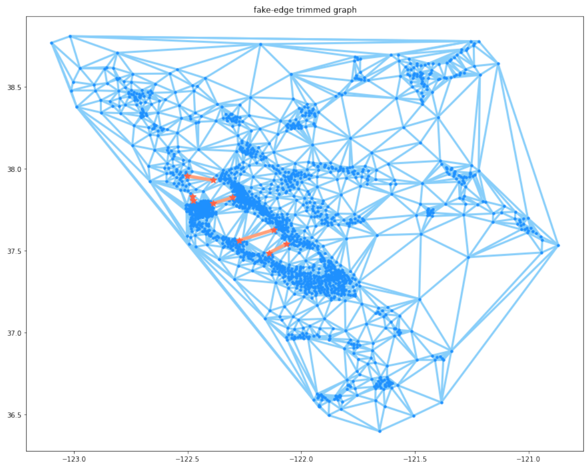
```
for i in range(len(bridges)):
    road = bridges[i]
    near_node1 = []
```

```
    near_node2 = []
    for node in node_dict:
        x = node_dict[node]['Location'][0]
        y = node_dict[node]['Location'][1]
        if round(road[0, 0], 1) == round(x, 1) and round(road[0, 1], 1)
            == round(y, 1):
            near_node1.append(node)
        if round(road[1, 0], 1) == round(x, 1) and round(road[1, 1], 1)
            == round(y, 1):
            near_node2.append(node)
    node1 = get_nearest_node(road[0, :], near_node1)
    node2 = get_nearest_node(road[1, :], near_node2)
    bridge_on_map.append((node1, node2))
```

Cuối cùng là vẽ đồ thị $\tilde{G}_\Delta$.

```
plt.figure(figsize=(15, 12))

for edge in G_delta.edges:
    x = [node_dict[edge[0]]['Location'][0],
        node_dict[edge[1]]['Location'][0]]
    y = [node_dict[edge[0]]['Location'][1],
        node_dict[edge[1]]['Location'][1]]
    plt.plot(x, y, linewidth=3, color='lightskyblue')

for node in G_delta.nodes:
    plt.plot(node_dict[node]['Location'][0],
        node_dict[node]['Location'][1], '.', markersize=7,
        color='dodgerblue')

for road in bridges:
    x = [road[0, 0], road[1, 0]]
    y = [road[0, 1], road[1, 1]]
    plt.plot(x, y, linewidth=5, color='lightsalmon')
    plt.plot(x, y, '*', markersize=10, color='tomato')

plt.title("fake-edge trimmed graph")
plt.show()
```

Trong đồ thị của $\tilde{G}_\Delta$, phân biệt các đỉnh, cạnh màu xanh (thông thường) với các đỉnh, cạnh màu đỏ (các cây cầu).



Có thể thấy rằng, các cây cầu này sau khi qua quá trình tối ưu vẫn được giữ nguyên.

Ngoài ra, ta có thể lặp lại bài toán luồng cực đại đã thực hiện ở phần 3.5 để xem liệu có thay đổi nào sau khi thực hiện tối ưu bản đồ không.

```
flow_value, flow_dict = nx.maximum_flow(G_tilde, UCSC_ID, Stanford_ID)
print("Max flow is", flow_value)
# Max flow is 14866.477294089984
```

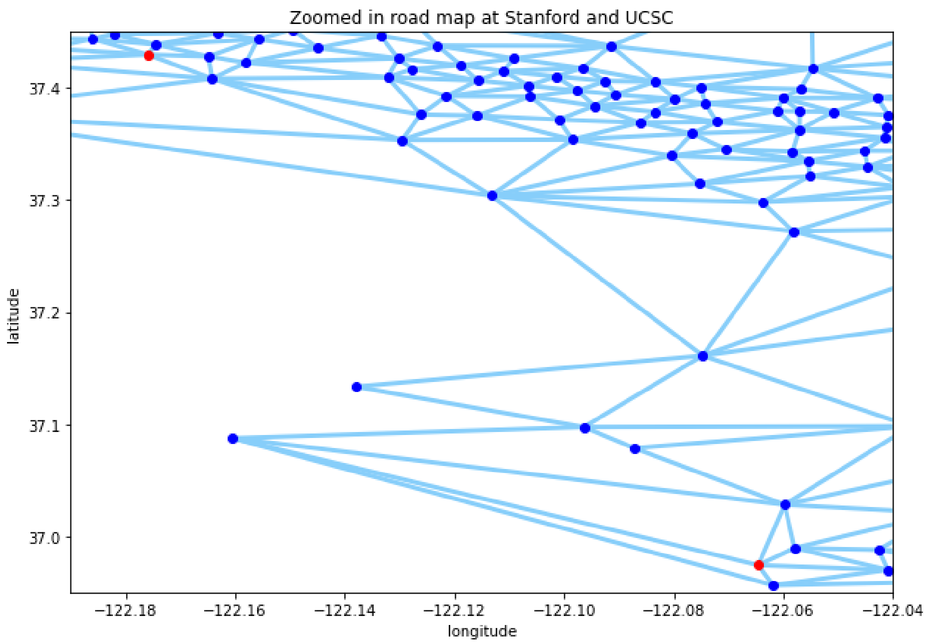
Có thể rằng giá trị của luồng cực đại không thay đổi gì.

```
edge_disjoint_paths = nx.edge_disjoint_paths(G_tilde, UCSC_ID,
    Stanford_ID)
count = 0
for i in edge_disjoint_paths:
    count += 1
    print(i)

print("The number of edge-disjoint paths are:", count)
# [1968, 2241, 1980, 2528, 2242, 938, 1947, 2656, 829, 931, 2222, 2223,
    1357, 1853, 1848, 1849, 1850, 1358, 1852, 743, 2378, 2607]
# [1968, 1424, 239, 1432, 1989, 1934, 2596, 1936, 725, 2656, 1948, 2490,
    2223, 1853, 1849, 1361, 1363, 2240, 2607]
# [1968, 1431, 1981, 2528, 938, 2221, 725, 1947, 1948, 744, 1853, 1869,
    1363, 2607]
# [1968, 748, 1424, 1431, 1989, 938, 744, 1869, 1737, 1736, 1726, 2607]
# [1968, 1980, 1431, 1432, 2232, 1934, 2221, 1947, 744, 1737, 1363,
    1736, 2607]
# The number of edge-disjoint paths are: 5
```

Đối với các path phân biệt (theo cạnh) từ UCSC đến Stanford có một chút sự khác biệt khi mà các path này dường như sẽ dài hơn so với trước khi tối ưu. Điều này là dễ hiểu bởi quá trình tối ưu là quá trình loại bỏ các cạnh giả. Tuy nhiên thì số lượng các đường như vậy vẫn không thay đổi là 5.

Hiển nhiên, nếu xét vùng bản đồ chứa UCSC và Stanford ta sẽ thấy nó thưa hơn so với lúc chưa tối ưu.

```python
plt.figure(figsize=(10, 7))
for edge in G_tilde.edges:
    x = [node_dict[edge[0]]['Location'][0],
        node_dict[edge[1]]['Location'][0]]
    y = [node_dict[edge[0]]['Location'][1],
        node_dict[edge[1]]['Location'][1]]
    plt.plot(x, y, linewidth=3, color='lightskyblue')

plt.plot(Stanford_loc[0], Stanford_loc[1], 'o', color='red')
plt.plot(UCSC_loc[0], UCSC_loc[1], 'o', color='red')

for node in node_dict:
    if node != Stanford_ID and node != UCSC_ID:
        plt.plot(node_dict[node]['Location'][0],
            node_dict[node]['Location'][1], 'o', color='blue')
plt.xlim(-122.19, -122.04)
plt.ylim(36.95, 37.45)

plt.xlabel('longitude')
plt.ylabel('latitude')
plt.title('Zoomed in road map at Stanford and UCSC')
plt.show()
```



Kết luận

Đồ án đã đạt được mục tiêu đề ra

Đồ án đã tìm hiểu về những kiến thức cơ bản, cốt lõi trong lĩnh vực *lý thuyết đồ thị*, một số mệnh đề, kết quả quan trọng được nêu ra và được chứng minh. Ngoài ra, đồ án cũng đã ứng dụng được để giải quyết một số bài toán thực tiễn quan trọng trên mạng giao thông.

Kỹ năng đạt được thông qua đồ án

1. Tìm kiếm, đọc tài liệu chuyên ngành (tiếng Anh) liên quan đến nội dung đồ án.
2. Tổng hợp kiến thức, tư duy, suy luận giải quyết bài toán.
3. Biết cách trình bày một báo cáo khoa học.
4. Hình thành tư duy phản biện, có khả năng kiểm tra tính đúng sai của tri thức trong nhiều tài liệu chuyên ngành.
5. Nâng cao khả năng lập trình, giải quyết bài toán với dữ liệu thực tế.

Hạn chế trong bài báo cáo

1. Một số thuật toán trong phần cơ sở lý thuyết chưa có ví dụ cụ thể (các thuật toán đường đi ngắn nhât, ...)
2. Các hình vẽ còn sao chép từ các nguồn trên mạng.
3. Một số bài toán trong phần ứng dụng chưa nói rõ về mặt lý thuyêt mà chỉ trích dẫn ở tài liệu tham khảo.
4. Phần code khó đọc.

Hướng phát triển đồ án trong tương lai

1. Tự triển khai, vẽ đồ thị minh họa các ví dụ, thuật toán.
2. Nghiên cứu sâu hơn về những bài toán có đề cập trong đồ án.
3. Tự cài đặt các thuật toán mà không dùng các gói có sẵn.
4. Tìm hiểu về các bài toán phức tạp hơn được ứng dụng trong mạng giao thông như dự đoán luồng, phân loại khu vực, ...

Tài liệu tham khảo

[1] West, D. B. (2000). *Introduction to Graph Theory*. Prentice Hall. ISBN: 0130144002.

[2] McHugh, J. A. (1990). *Algorithmic graph theory*. Prentice Hall. ISBN: 978-0-13-019092-5

[3] Bondy, J. A., Murty, U. S. R. (1976). *Graph Theory with Applications*. New York: Elsevier.

[4] Papadimitriou, C. H., & Steiglitz, K. (1998). *Combinatorial optimization: algorithms and complexity*. Courier Corporation.

[5] Poorten, P. V. D., & Jones, C. B. (2002). *Characterisation and generalisation of cartographic lines using Delaunay triangulation*. International Journal of Geographical Information Science, 16(8), 773-794.

[6] Shewchuk, J. R. (1997). *Delaunay refinement mesh generation*. Carnegie Mellon University.

[7] Zhang, C., Li, Y., Xiang, L., Jiao, F., Wu, C., & Li, S. (2021). *Generating road networks for old downtown areas based on crowd-sourced vehicle trajectories*. Sensors, 21(1), 235.