VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

CẤU TRÚC DỮ LIỆU & GIẢI THUẬT (MR) (CO2003)

Bài tập lớn

Đồ thị vô hướng (Undirected Graph Data Structure)

GVHD: Lê Thành Sách
Sinh viên: Nguyễn Tuấn Minh - 2110359

Thành phố Hồ Chí Minh, Tháng 12/2022

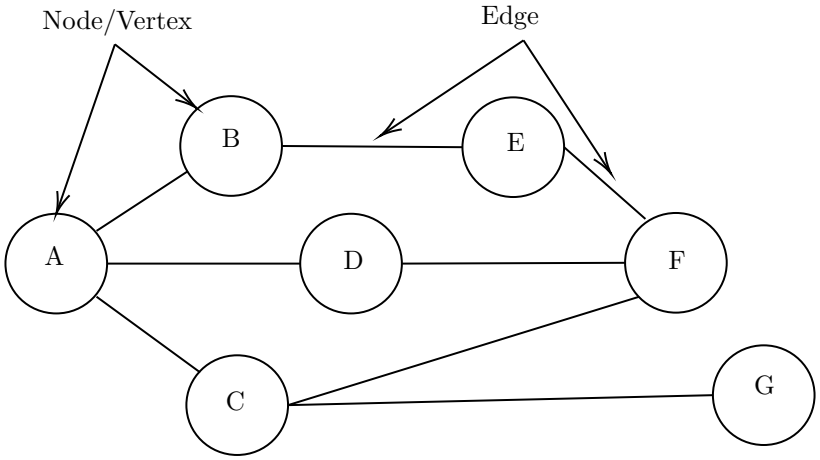
Contents

1 Cấu trúc dữ liệu Graph

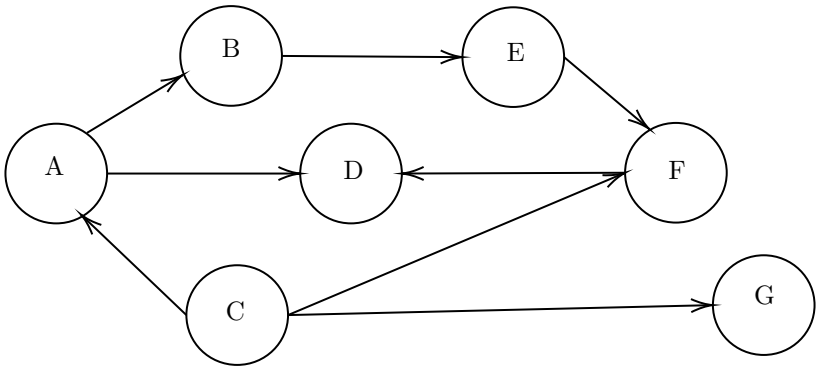
1.1 Giới thiệu về Graph

1.1.1 Định nghĩa

Đồ thị là một cấu trúc dữ liệu phi tuyến tính gồm các đỉnh (vertex/node) và các cạnh (edge). Các đỉnh cũng thường được gọi là các node và các cạnh là các đường thẳng hoặc đường cong nối hai node với nhau. Theo định nghĩa, đồ thị là một tập hợp không rỗng các đỉnh (V) và một tập hợp các cạnh (E), kí hiệu G(V, E). Các đỉnh/cạnh có thể có hoặc không có nhãn và có hoặc không có trọng số, bên cạnh đó các cạnh cũng có thể có thứ tự (đối với đồ thị có hướng) hoặc không (đối với đồ thị vô hướng)



Đồ thị vô hướng



Đồ thị có hướng

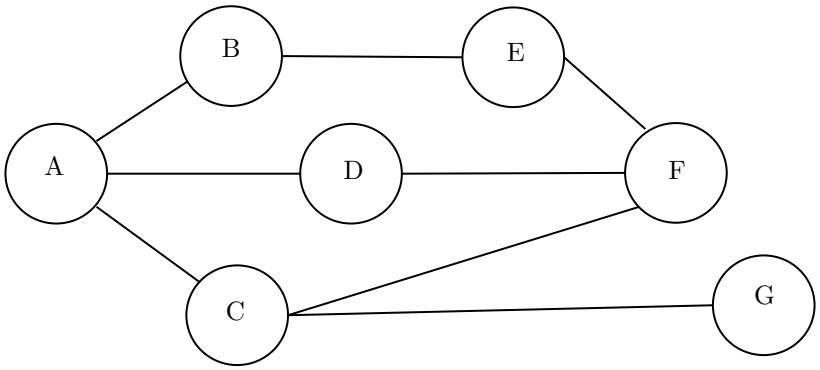
Đường đi (Path) là một chuỗi các đỉnh kề nhau

Chu trình (Cycle/Circuit) là một đường đi có ít nhất ba đỉnh mà đỉnh cuối cùng kề với đỉnh đầu tiên
Ở bài tập lớn này, ta tập trung tìm hiểu về **Đồ thị vô hướng** trong cấu trúc dữ liệu đồ thị

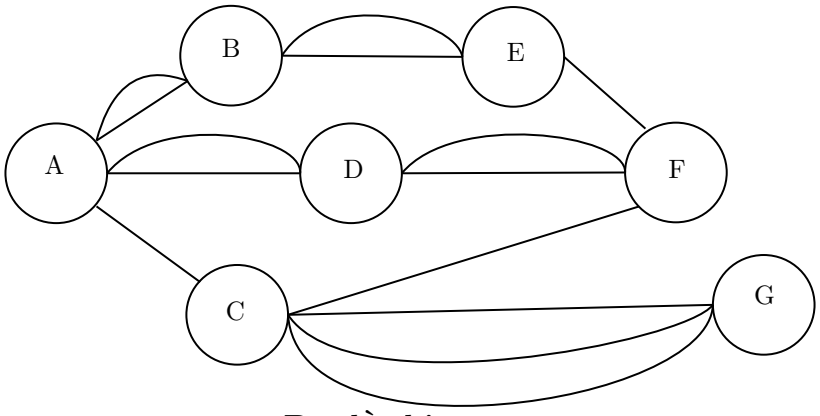
1.2 Đồ thị vô hướng (Undirected Graph)

1.2.1 Các loại đồ thị

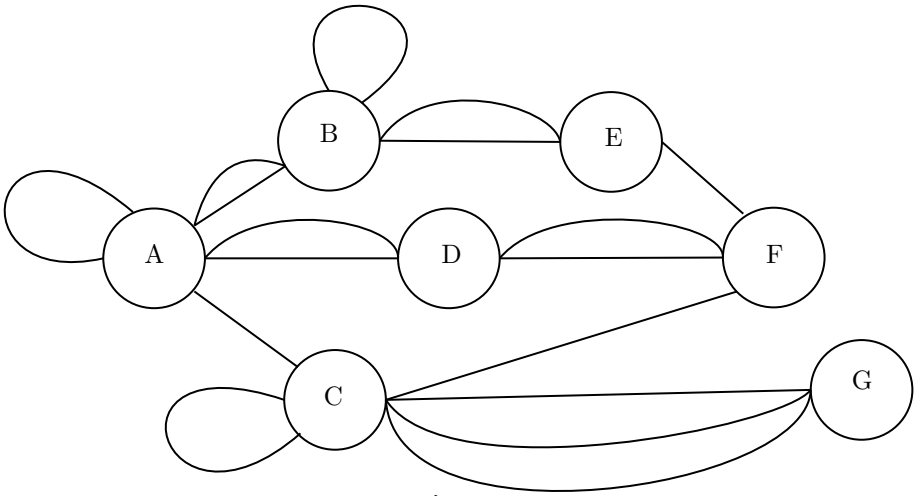
Đối với đồ thị vô hướng, dựa theo tính chất của các cạnh, ta có thể chia thành: đơn đồ thị (simple graph), đa đồ thị (multigraph) và giả đồ thị (pseudograph)



Đơn đồ thị

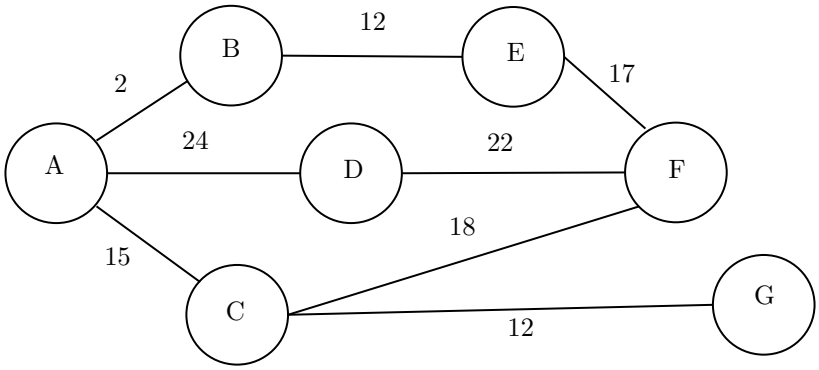


Đa đồ thị



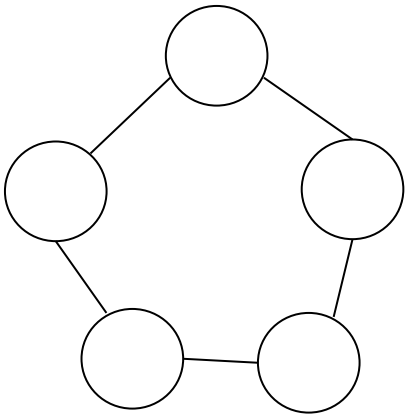
Giả đồ thị

Bên cạnh đó đồ thị còn có thể chia thành đồ thị có trọng số (weighted graph) và đồ thị không có trọng số (unweighted graph).

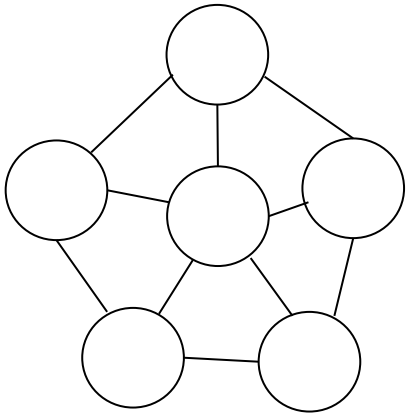


Đồ thị có trọng số

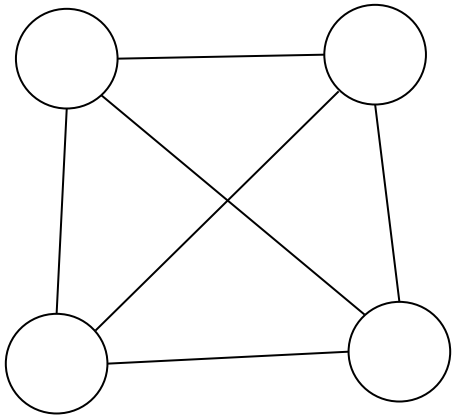
Ngoài ra, dựa vào tính chất khác của đồ thị, còn có thể thành: đồ thị vòng (cycle), đồ thị đầy đủ (complete graph), đồ thị bánh xe (wheel), khối n-chiều (n-dimensional hypercube), đồ thị rỗng (null graph), đồ thị tầm thường (trivial graph),...



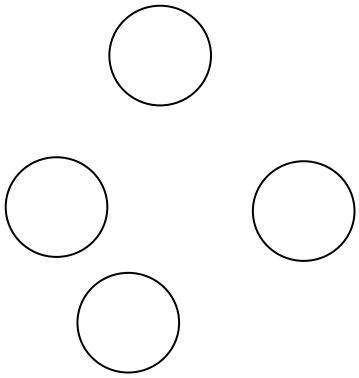
Đồ thị vòng



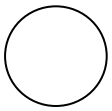
Đồ thị bánh xe



Đồ thị đầy đủ



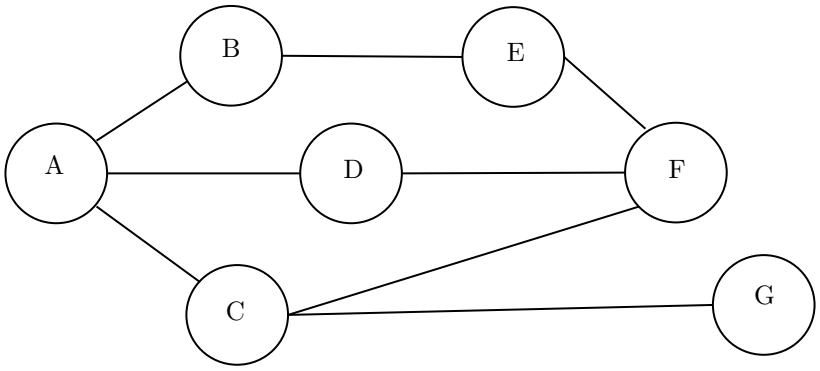
Đồ thị rỗng



Đồ thị tầm thường

1.2.2 Cách biểu diễn

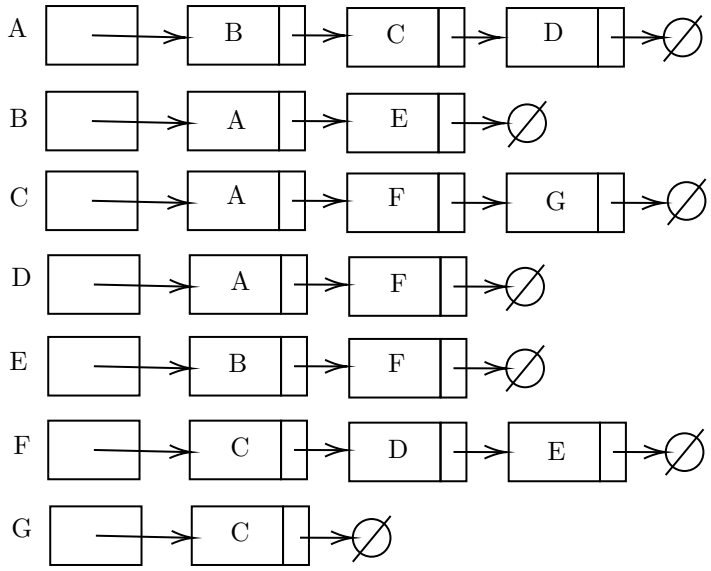
Để dễ hình dung, ta sẽ sử dụng đồ thị sau đây để làm ví dụ với các cách biểu diễn.



Danh sách kề

Danh sách kề là danh sách thể hiện các đỉnh kề của mỗi đỉnh trong đồ thị. Để hiện thực, ta cần dử dụng một mảng (array) các danh sách (list), danh sách thứ i của mảng thể hiện danh sách những đỉnh kể với đỉnh thứ i.

Đỉnh	**Đỉnh kề**
A	{B, C, D}
B	{A, E}
C	{A, F, G}
D	{A, F}
E	{B, F}
F	{C, D, E}
G	{C}



Ma trận kề

Ma trận kề là cách biểu diễn thường thấy nhất ở một đồ thị, ở cách biểu diễn này, đồ thị sẽ được biểu diễn bằng một ma trận vuông cấp n (với n là số đỉnh), nếu phần từ tại hàng i cột j có giá trị là 1 thì đỉnh thứ i và đỉnh thứ j là hai đỉnh kề nhau, ngược lại nếu giá trị đó là 0 thì hai

đỉnh không kề nhau. Để hiện thực ma trận kề, đơn giản ta chỉ cần sử dụng một mảng hai chiều. Đối với đồ thị có trọng số, nếu hai đỉnh kề nhau, giá trị tại phần tử tương ứng trong ma trận sẽ chính là trọng số đó.

$$\begin{bmatrix} & A & B & C & D & E & F & G \\ A & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ B & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ C & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ D & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ E & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ F & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ G & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Ngoài hai cách biểu diễn thường được sử dụng ở trên, còn có rất nhiều cách biểu diễn đồ thị tùy thuộc vào những trường hợp khác nhau như ma trận liên thuộc,...

1.2.3 Các thuộc tính

Class Edge dùng để biểu diễn đối tượng là các cạnh (edge), gồm 3 thuộc tính:

- VertexNode *from, *to: là các con trỏ tới đỉnh đầu và đỉnh cuối của cạnh
- float weight: là trọng số của cạnh, nếu đồ thị không có trọng số thì trọng số bằng 0

Class VertexNode dùng để biểu diễn đối tượng là các đỉnh (node/vertex), gồm 4 thuộc tính:

- T vertex: biểu diễn nhãn/giá trị của đỉnh
- int inDegree: dùng để lưu trữ số bậc trong của đỉnh hiện tại
- int outDegree: dùng để lưu trữ số bậc ngoài của đỉnh hiện tại
- DLinkedList<Edge*> adList: dùng để lưu trữ một danh sách các con trỏ trỏ vào cạnh đến hoặc đi từ đỉnh hiện tại.

Tuy nhiên, với đồ thị vô hướng, ta không phân biệt bậc trong và bậc ngoài cũng như việc một cạnh là đi tới hay đi từ đỉnh đang xét. Bậc của một đỉnh, bậc trong và bậc ngoài của đỉnh đó đều sẽ bằng nhau.

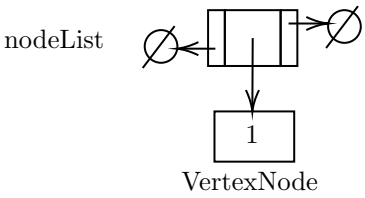
Class Iterator biểu diễn đối tượng là một Iterator, gồm duy nhất một thuộc tính:

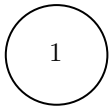
- typename DLinkedList<VertexNode*>::Iterator nodeIt: một Iterator cho một danh sách liên kết đôi với phần tử là con trỏ tới các đỉnh của đồ thị

DLinkedList<VertexNode*> nodeList là một danh sách dùng để lưu trữ con trỏ tới tất cả các đỉnh có trong đồ thị, đây cũng sẽ là thuộc tính chính để quản lý, biểu diễn và hiện thực các phương thức của đồ thị

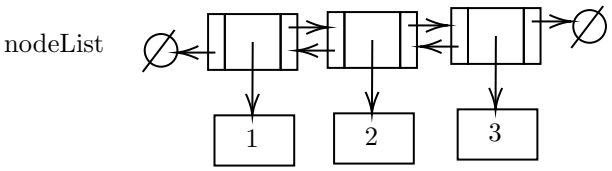
1.2.4 Các phương thức

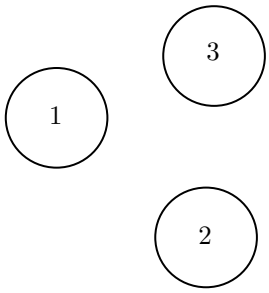
void add(T vertex): phương thức này nhận tham số là một đỉnh và thêm đỉnh đó vào đồ thị



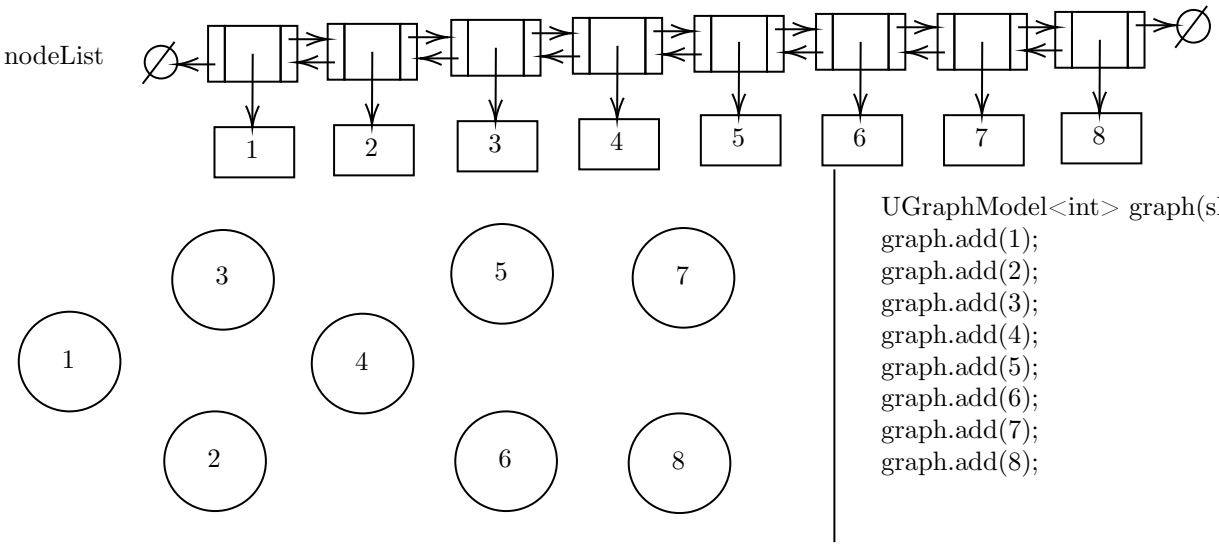



```
UGraphModel<int> graph;
graph.add(1);
```



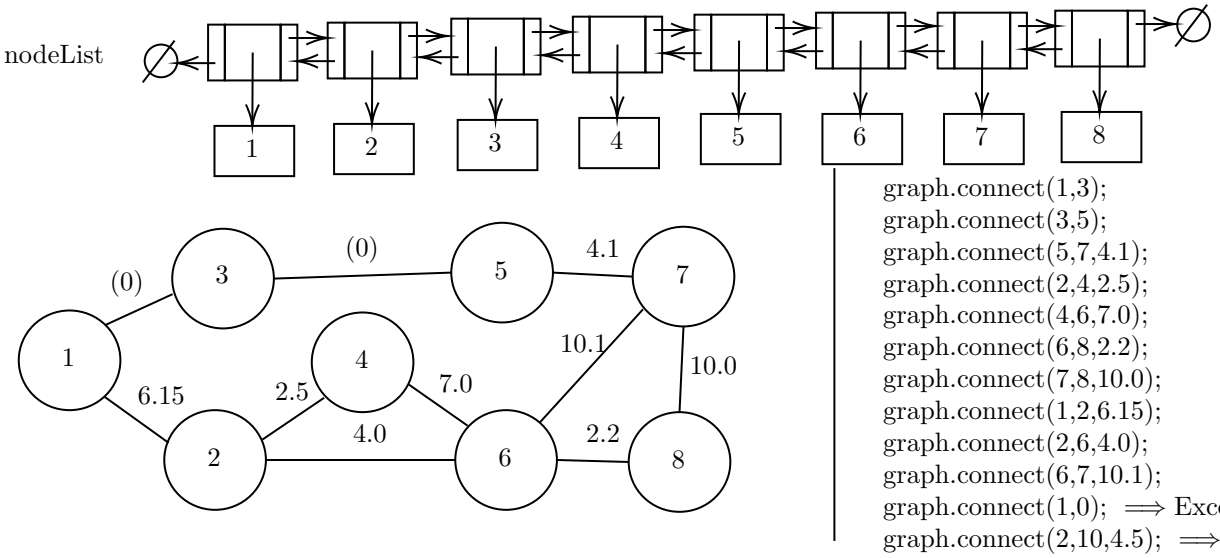



```
UGraphModel<int> graph;
graph.add(1);
graph.add(2);
graph.add(3);
```




```
UGraphModel<int> graph(sEQ,v2s);
graph.add(1);
graph.add(2);
graph.add(3);
graph.add(4);
graph.add(5);
graph.add(6);
graph.add(7);
graph.add(8);
```

void connect(T from, T to, float weight=0): phương thức này nhận vào hai tham số là hai đỉnh và một tham số mặc định thứ ba là một số thực với giá trị mặc định bằng 0 (trường hợp đồ thị không có trọng số), nếu hai đỉnh tồn tại thì hai đỉnh đó sẽ được kết nối với nhau bằng một cạnh với trọng số là số thực truyền vào, ngược lại ném ra ngoại lệ

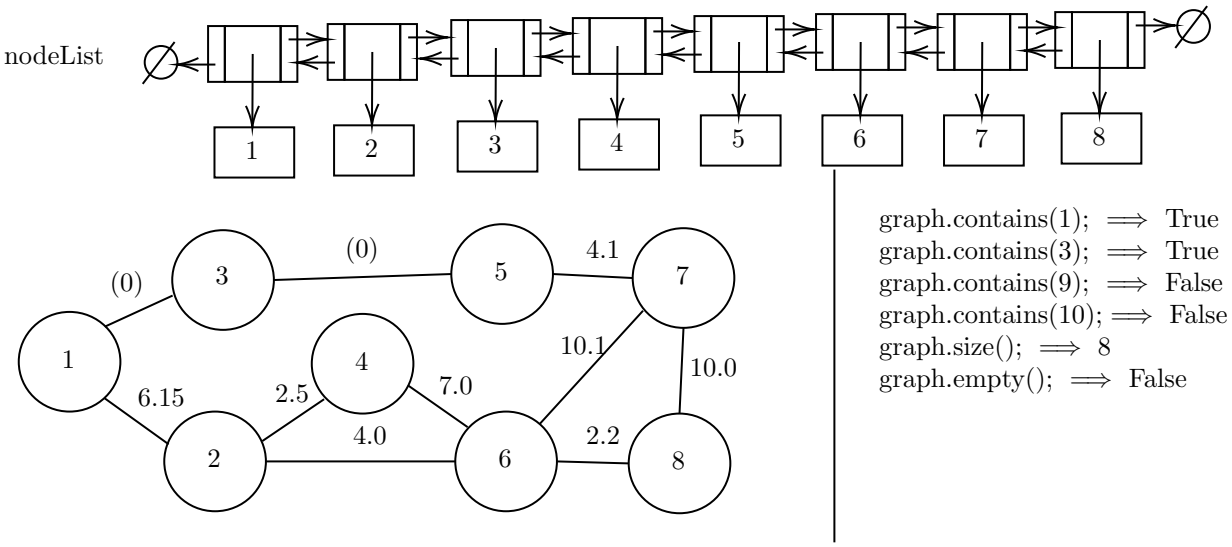



```
graph.connect(1,3);
graph.connect(3,5);
graph.connect(5,7,4.1);
graph.connect(2,4,2.5);
graph.connect(4,6,7.0);
graph.connect(6,8,2.2);
graph.connect(7,8,10.0);
graph.connect(1,2,6.15);
graph.connect(2,6,4.0);
graph.connect(6,7,10.1);
graph.connect(1,0); ⟹ Exception
graph.connect(2,10,4.5); ⟹ Exception
```

bool contains(T vertex): phương thức này nhận tham số là một đỉnh, nếu đỉnh tồn tại thì trả về **True**, ngược lại trả về **False**

int size(): phương thức này trả về số lượng đỉnh có trong đồ thị, cũng chính là số lượng phần tử có trong danh sách **DLinkedList<VertexNode*> nodeList**

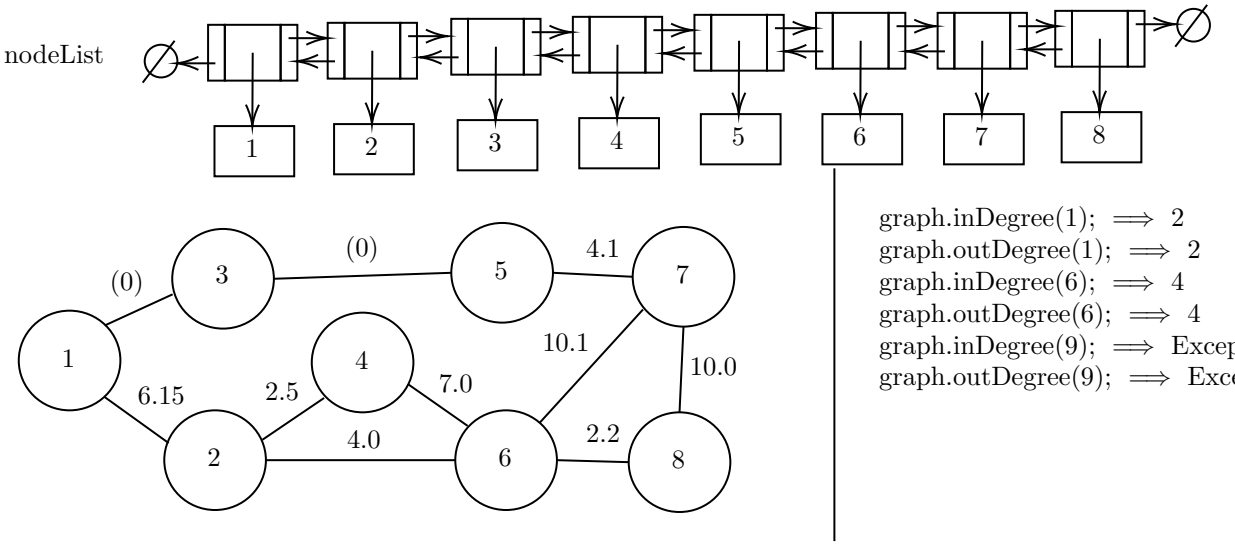
bool empty(): phương thức này trả về **True** (đồ thị đang xét rỗng) nếu đồ thị có số lượng đỉnh bằng 0, ngược lại trả về **False**



graph.contains(1); $\Longrightarrow$ True
graph.contains(3); $\Longrightarrow$ True
graph.contains(9); $\Longrightarrow$ False
graph.contains(10); $\Longrightarrow$ False
graph.size(); $\Longrightarrow$ 8
graph.empty(); $\Longrightarrow$ False

int inDegree(T vertex): phương thức này nhận tham số là một đỉnh, tìm kiếm đỉnh đó trong đồ thị, nếu đỉnh tồn tại thì trả về số bậc trong của đỉnh đó, ngược lại sẽ ném ra ngoại lệ

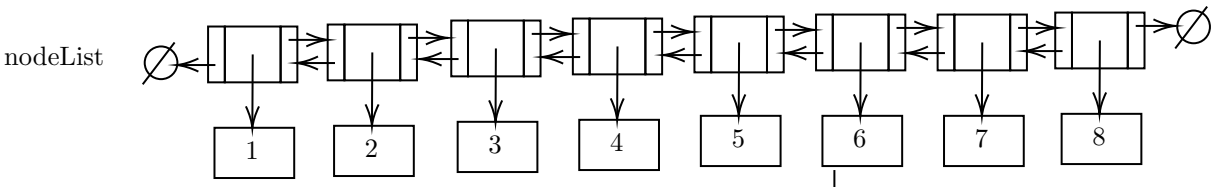
int outDegree(T vertex): phương thức này nhận tham số là một đỉnh, tìm kiếm đỉnh đó trong đồ thị, nếu đỉnh tồn tại thì trả về số bậc ngoài của đỉnh đó, ngược lại sẽ ném ra ngoại lệ
***Đối với độ thị vô hướng hai phương thức **int inDegree(T vertex)** và **int outDegree(T vertex)** đều sẽ trả về kết quả như nhau vì ta không phân biệt bậc ngoài và bậc trong của một đỉnh

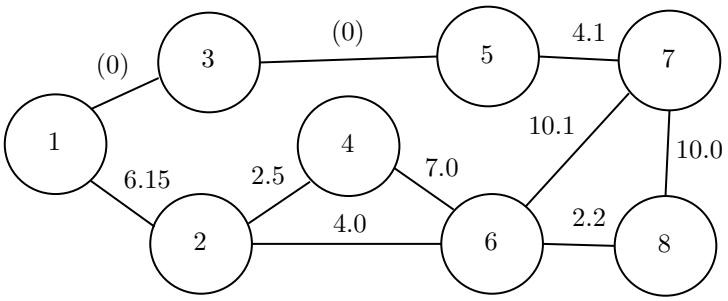


graph.inDegree(1); $\Longrightarrow$ 2
graph.outDegree(1); $\Longrightarrow$ 2
graph.inDegree(6); $\Longrightarrow$ 4
graph.outDegree(6); $\Longrightarrow$ 4
graph.inDegree(9); $\Longrightarrow$ Exception
graph.outDegree(9); $\Longrightarrow$ Exception

DLinkedList<T> getOutwardEdges(T from): phương thức này nhận vào tham số là một đỉnh, nếu đỉnh này tồn tại thì trả về một danh sách các đỉnh có thể đi tới từ đỉnh đã cho. Nếu đỉnh đã cho không tồn tại, ném ra ngoại lệ

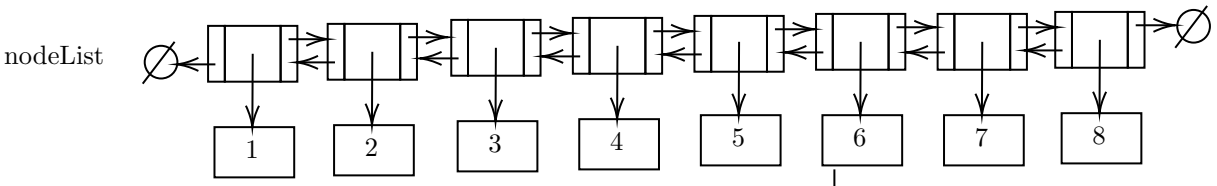
DLinkedList<T> getInwardEdges(T to): phương thức này nhận vào tham số là một đỉnh, nếu đỉnh này tồn tại thì trả về một danh sách các đỉnh có thể đi tới đỉnh đã cho. Nếu đỉnh đã cho không tồn tại, ném ra ngoại lệ
***Vì ta chỉ đang xét đồ thị vô hướng nên cả hai phương thức **DLinkedList<T> getOutwardEdges(T from)** và **DLinkedList<T> getInwardEdges(T to)** đều sẽ trả về kết quả như nhau là một danh sách các đỉnh kề với đỉnh đang xét

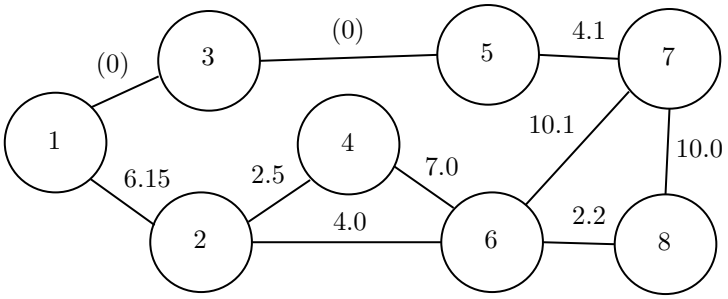




graph.getOutwardEdge(1); $\Longrightarrow$ 2-3
graph.getInwardEdge(1); $\Longrightarrow$ 2-3
graph.getOutwardEdge(6); $\Longrightarrow$ 2-4-7-8
graph.getInwardEdge(6); $\Longrightarrow$ 2-4-7-8
graph.getOutwardEdge(10); $\Longrightarrow$ Exception
graph.getInwardEdge(10); $\Longrightarrow$ Exception

float weight(T from, T to): phương thức này nhận vào tham số là hai đỉnh, nếu hai đỉnh tồn tại và giữa hai đỉnh đó có cạnh, trả về trọng số của cạnh đó, ngược lại ném ra ngoại lệ

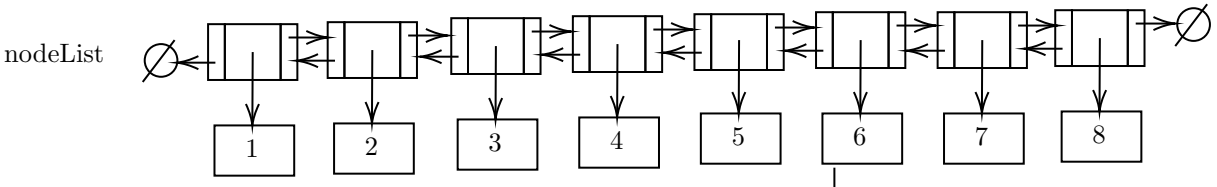


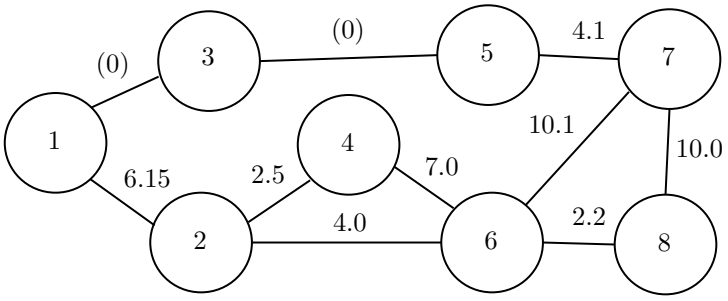


graph.weight(1,2); $\Longrightarrow$ 6.15
graph.weight(1,3); $\Longrightarrow$ 0
graph.weight(6,7); $\Longrightarrow$ 10.1
graph.weight(1,4); $\Longrightarrow$ Exception
graph.weight(1,10); $\Longrightarrow$ Exception
graph.weight(9,10); $\Longrightarrow$ Exception

bool connected(T from, T to): phương thức này nhận vào tham số là hai đỉnh, nếu hai đỉnh tồn tại và giữa hai đỉnh đó có cạnh, trả về **True**, nếu không có cạnh trả về **False**, ngược lại nếu một trong hai hoặc cả hai đỉnh đều không tồn tại thì ném ra ngoại lệ

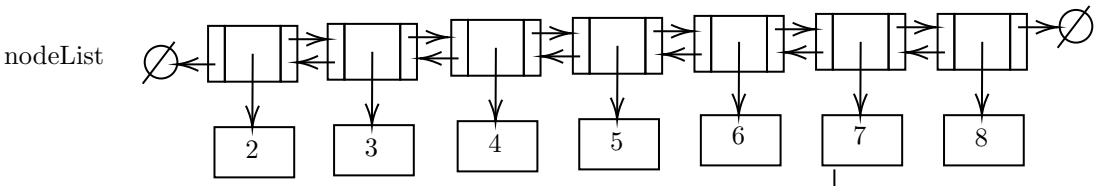
DLinkedList<T> vertices(): phương thức này sẽ trả về một danh sách các đỉnh của đồ thị

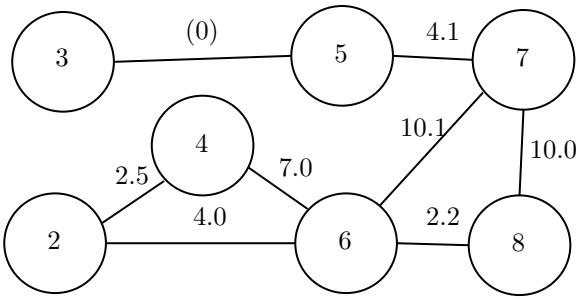




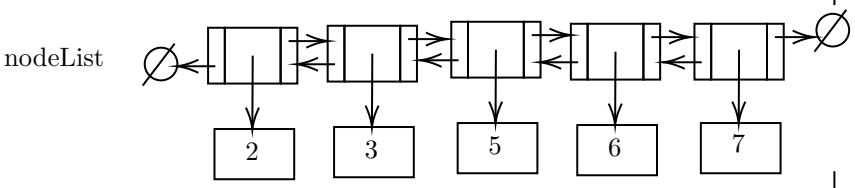
graph.connected(1,2); $\Longrightarrow$ True
graph.connected(1,3); $\Longrightarrow$ True
graph.connected(6,7); $\Longrightarrow$ True
graph.connected(1,4); $\Longrightarrow$ False
graph.connected(1,10); $\Longrightarrow$ Exception
graph.connected(9,10); $\Longrightarrow$ Exception
graph.vertices(); $\Longrightarrow$ 1-2-3-4-5-6-7-8

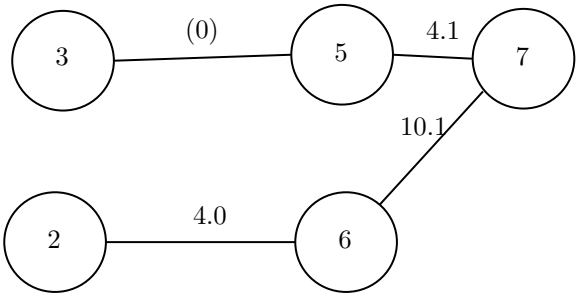
void remove(T vertex): phương thức này nhận vào tham số là một đỉnh, nếu đỉnh này tồn tại, xóa đi tất cả các cạnh từ đỉnh đó và xóa đỉnh đó khỏi đồ thị. Ngược lại ném ra một ngoại lệ





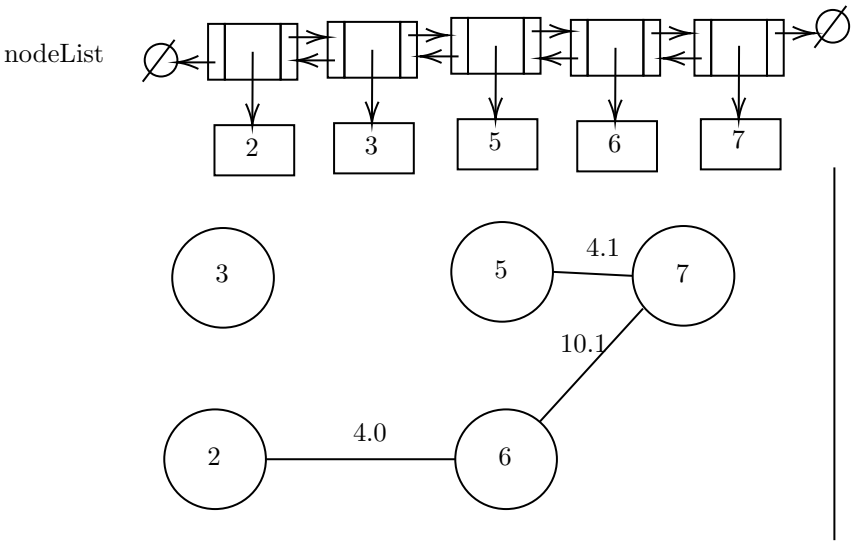
graph.remove(1);
graph.size(); $\Longrightarrow$ 7
graph.contains(1); $\Longrightarrow$ False
graph.vertices(); $\Longrightarrow$ 2-3-4-5-6-7-8
graph.remove(1); $\Longrightarrow$ Exception
graph.remove(9); $\Longrightarrow$ Exception





graph.remove(4);
graph.remove(8);
graph.size(); $\Longrightarrow$ 5
graph.contains(4); $\Longrightarrow$ False
graph.vertices(); $\Longrightarrow$ 2-3-5-6-7

void disconnect(T from, T to): phương thức này nhận vào hai tham số là hai đỉnh, nếu hai đỉnh tồn tại và hai đỉnh đó được kết nối bằng một cạnh thì cạnh đó sẽ được xóa đi, ngược lại ném ra ngoại lệ



graph.disconnect(3,5);
graph.disconncet(3,5); $\Longrightarrow$ Exception
graph.disconncet(3,7); $\Longrightarrow$ Exception
graph.disconncet(1,5); $\Longrightarrow$ Exception
graph.disconncet(1,4); $\Longrightarrow$ Exception

void clear(): phương thức này xóa đi các phần tử trong danh sách **DLinkedList<VertexNode*> nodeList** đến khi nào danh sách đó rỗng hay nói cách khác là xóa đồ thị

graph.clear();
graph.empty(); $\Longrightarrow$ True

string toString(): phương thức này trả về biểu diễn đồ thị bằng một chuỗi tùy theo cài đặt của người sử dụng

1.3 Các vấn đề trong bài toán về đồ thị

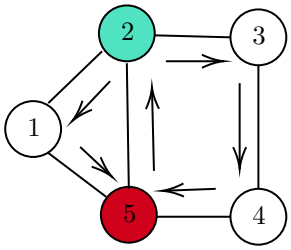
1.3.1 Thuật toán tìm đường đi/chu trình Euler

Đường đi Euler là đường đi mà đi qua tất cả các cạnh của đồ thị duy nhất một lần

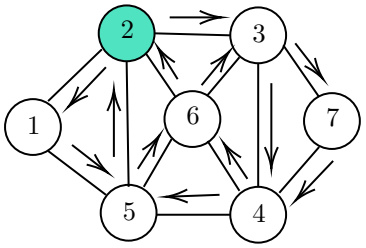
Chu trình Euler là đường đi Euler có đỉnh đầu và đỉnh cuối trùng nhau

Để dễ hình dung, việc tìm đường đi/chu trình Euler tương tự với việc tìm cách vẽ tất cả các cạnh của đồ thị chỉ với một nét bút. Một đồ thị có thể có nhiều đường đi/chu trình Euler

Một đồ thị được gọi là **Đồ thị Euler** nếu đồ thị đó có chu trình Euler, và **Đồ thị bán Euler** nếu đồ thị đó có đường đi Euler



Đường đi Euler: 2-3-4-5-2-1-5



Chu trình Euler: 2-1-5-2-3-4-5-6-3-7-4-6-2

Để một đồ thị tồn đường đi/chu trình Euler, đồ thị đó phải thỏa mãn hai điều kiện sau đây

1. Tất cả những đỉnh có bậc khác 0 đều liên thông với nhau
2. Tất cả các đỉnh đều có bậc chẵn đối với chu trình Euler, có 0 hoặc chỉ 2 đỉnh có bậc lẻ đối với đường đi Euler

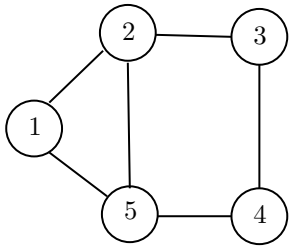
Để xác định đường đi/chu trình Euler, có thể sử dụng thuật toán sau:

1.3.1.a Thuật toán Fleury

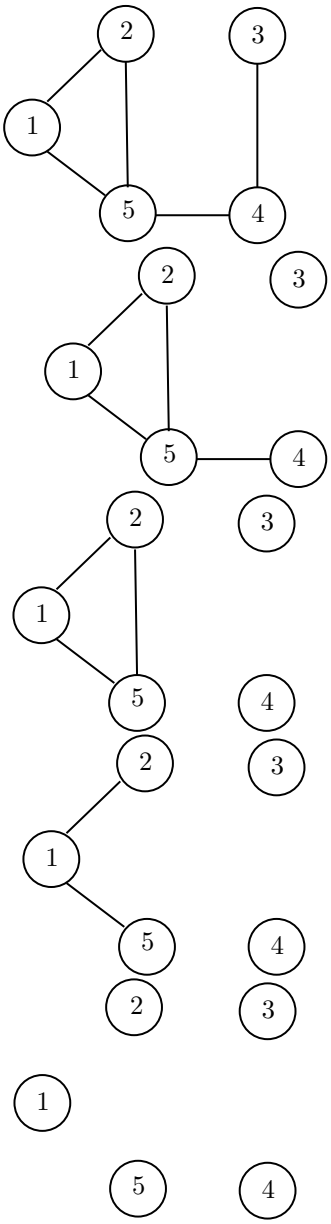
Các bước hiện thực Thuật toán Fleury:

1. Đảm bảo tồn tại có đường đi/chu trình Euler
2. Nếu không có đỉnh bậc lẻ, bắt đầu từ bất kì đâu. Nếu có hai đỉnh có bậc lẻ, bắt đầu từ một trong hai đỉnh đó.
3. Đi theo một cạnh nào đó tại mỗi đỉnh, nếu phải chọn giữa cạnh cầu/cạnh cắt (bridge/cut-edge) và không tạo ra cầu (non-brigde), luôn chọn cạnh không tạo ra cầu. (**Cầu/cạnh cắt (Bridge/cut-edge)** là cạnh khi bỏ đi sẽ làm tăng số thành phần liên thông trong đồ thị)
4. Dừng lại khi đã đi qua tất cả các cạnh

Ví dụ minh họa:
Ta sẽ xem xét đồ thị sau:



Dễ thấy đồ thị bên tồn tại đường đi Euler với hai đỉnh 2 và 5 có bậc lẻ, các đỉnh còn lại bậc chẵn, ta chọn đỉnh 2 làm đỉnh bắt đầu



Từ đỉnh 2, ta có 3 lựa chọn: cạnh (2,3), cạnh (2,1) và cạnh (2,5). Chọn cạnh (2,3).

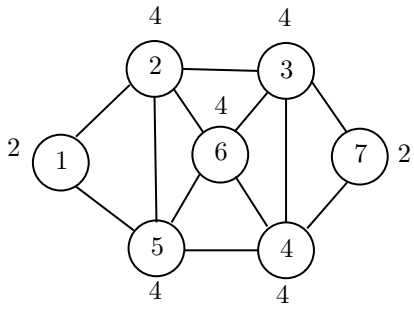
Từ đỉnh 3, ta chỉ có một lựa chọn duy nhất là cạnh (3,4).

Từ đỉnh 4, ta cũng chỉ có một lựa chọn duy nhất là cạnh (4,5).

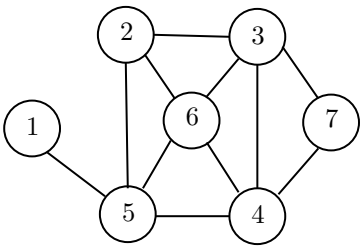
Từ đỉnh 5, ta có 2 lựa chọn là: cạnh (5,2) và cạnh (5,1). Chọn cạnh (5,2)

Từ đỉnh đây trở đi, mỗi bước ta cũng sẽ chỉ có một lựa chọn, lần lượt là cạnh (2,1) và (1,5). Đường đi Euler của đồ thị này là 2-3-4-5-2-1-5.

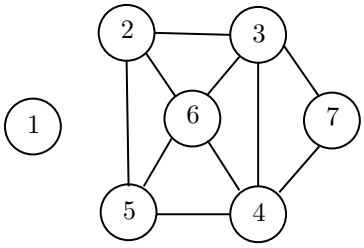
Tiếp tục với đồ thị sau:



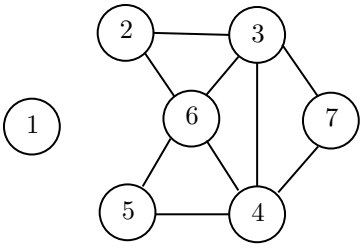
Đồ thị bên có tất cả các đỉnh đều có bậc chẵn, ta có thể bắt đầu từ bất kỳ đỉnh nào.



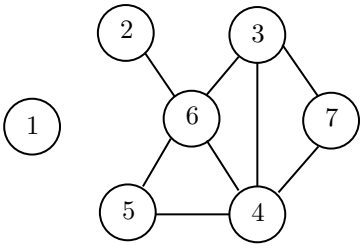
Chọn đỉnh 2 làm đỉnh bắt đầu. Tại đây ta có 4 lựa chọn, chọn cạnh (2,1).



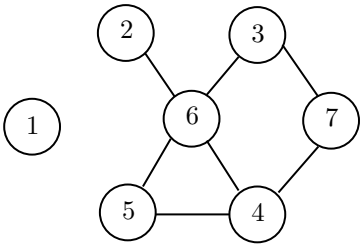
Từ đỉnh 1 ta chỉ có một lựa chọn là cạnh (1,5).



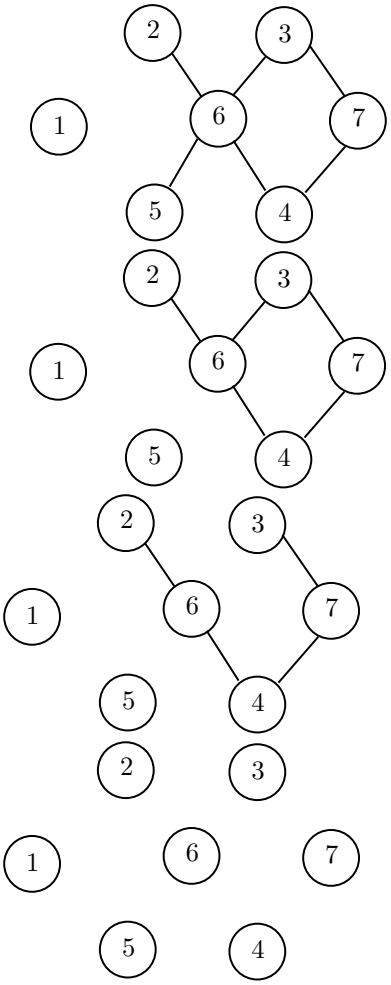
Từ đỉnh 5 ta có 3 lựa chọn, chọn cạnh (2,5).



Từ đỉnh 2 ta cũng có 3 lựa chọn, chọn cạnh (2,3).



Từ đỉnh 3 ta cũng có 3 lựa chọn, chọn cạnh (3,4).



Từ đỉnh 4 ta có 2 lựa chọn, chọn cạnh (4,5).

Từ đỉnh 5 ta chỉ có 1 lựa chọn là cạnh (5,6).

Từ đỉnh 6 ta có 3 lựa chọn, tuy nhiên cạnh (6,2) là cạnh cắt (cut-edge) nên ta sẽ chọn 1 trong hai cạnh còn lại. Chọn cạnh (6,3).

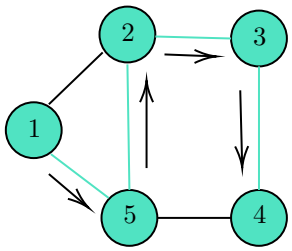
Từ đây, tại mỗi bước ta chỉ có một lựa chọn, lần lượt là các cạnh (3,7)-(7,4)-(4.6)-(6.2). Vậy chu trình Euler của đồ thị là 2-1-5-2-3-4-5-6-3-7-4-6-2

1.3.2 Thuật toán tìm đường đi/chu trình Hamilton

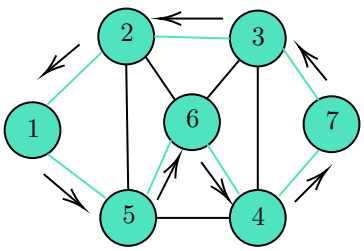
Đường đi Hamilton là đường đi mà đi qua tất cả các đỉnh của đồ thị duy nhất một lần

Chu trình Hamilton là đường đi Hamilton có đỉnh đầu và cuối trùng nhau

Một đồ thị có thể có nhiều đường đi/chu trình Hamilton.



Đường đi Hamilon: 1-5-2-3-4



Chu trình Hamilon: 1-5-6-4-7-3-2-1

Ở phần này, ta chỉ tìm hiểu định nghĩa về đường đi/chu trình Hamilton vì: Khác với việc tím ra liệu một đồ thị có đường đi/chu trình Halmiton hay không và tìm ra nó có thể được giải quyết trong thời gian đa thức, để xác định sự tồn tại của đường đi/chu trình Hamilton có trong đồ thị là một vấn đề NP và khá phức tạp khi phải kết hợp nhiều giải thuật khác nhau.

1.3.3 Thuật toán duyệt đồ thị

Đối với thao tác duyệt đồ thị, hai thuật toán thường được dùng nhất là BFS và DFS

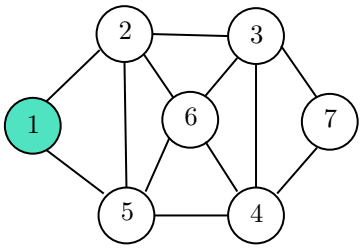
1.3.3.a BFS (Breadth First Search)

Ý tưởng: BFS là thuật toán duyệt đồ thị theo chiều rộng, thuật toán được mô tả đơn giản bằng việc: chọn một đỉnh bắt đầu, duyệt qua tất cả các đỉnh kề với nó, sau đó duyệt tiếp các đỉnh kề các đỉnh vừa duyệt, cứ tiếp tục làm vậy đến khi nào tất cả các đỉnh đều được duyệt qua. Thuật toán BFS có thể hiểu đơn giản việc duyệt như một ngọn lửa bắt đầu từ điểm được chọn, sau đó lan dần ra cho các đỉnh kề ở mỗi bước. Để hiện thực thuật toán BFS, cần sử dụng thêm cấu trúc dữ liệu Queue (hàng đợi) để hỗ trợ. Các bước hiện thực BFS:

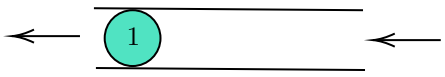
1. Thêm đỉnh bắt đầu vào hàng đợi
2. Khởi tạo mảng đánh dấu các phần tử đã được duyệt và đánh dầu phần tử đầu tiên đã được duyệt
3. Lặp lại các bước sau đến khi nào hàng đợi rỗng
    - Loại bỏ đỉnh đầu tiên trong hàng đợi
    - Thêm tất cả những đỉnh kề với đỉnh đầu tiên và chưa được duyệt vào hàng đợi và duyệt qua các đỉnh đó

Độ phức tạp về thời gian: O(V+E) đối với cách biểu diễn đồ thị bằng danh sách kề, O(V^2) đối với ma trận kề

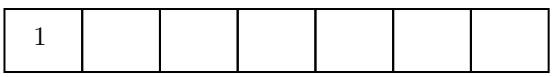
Ví dụ minh họa:



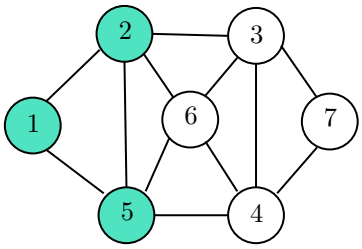
Chọn đỉnh 1 làm đỉnh bắt đầu, đánh dấu đã duyệt và thêm đỉnh 1 vào hàng đợi.



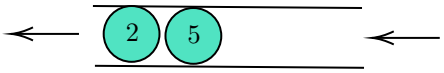
Hàng đợi



Thứ tự duyệt



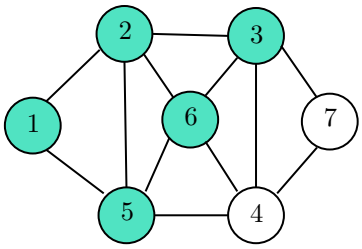
Loại đỉnh 1 ra khỏi hàng đợi. thêm các đỉnh 2,5 kề với đỉnh 1 vào hàng đợi và duyệt các đỉnh đó.



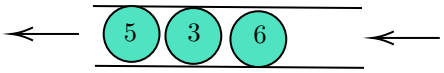
Hàng đợi

1	2	5				

Thứ tự duyệt



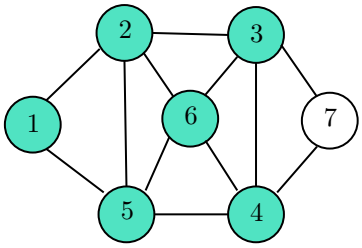
Loại đỉnh 2 ra khỏi hàng đợi, thêm các đỉnh 3,6 vào hàng đợi và duyệt các đỉnh đó.



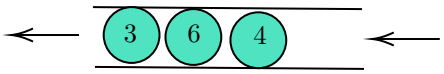
Hàng đợi

1	2	5	3	6		

Thứ tự duyệt



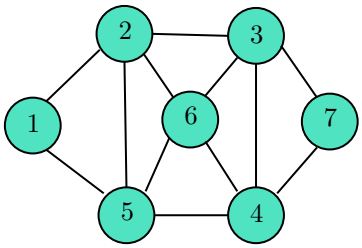
Loại đỉnh 5 ra khỏi hàng đợi, thêm đỉnh 4 và hàng đợi và duyệt đỉnh đó.



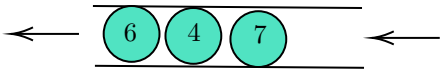
Hàng đợi

1	2	5	3	6	4	

Thứ tự duyệt



Loại đỉnh 3 ra khỏi hàng đợi, thêm đỉnh 7 và hàng đợi và duyệt đỉnh đó. Tới đây, vì tất cả các đỉnh đã được duyệt nên hàng đợi sẽ từ từ trở thành rỗng vì không có đỉnh nào có thể được thêm vào hàng đợi nữa, ta có thứ tự duyệt theo chiều rộng.



Hàng đợi

1	2	5	3	6	4	7

Thứ tự duyệt

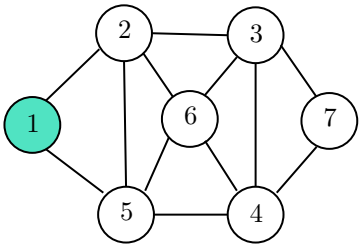
1.3.3.b DFS (Depth First Search)

Ý tưởng: DFS là thuật toán duyệt đồ thị theo chiều sâu, thuật toán đươc mô tả đơn giản bằng việc: chọn một đỉnh bắt đầu, duyệt theo một nhánh sâu nhất có thể, sau đó quay lui và tiếp tục duyệt sâu nhất có thể với mỗi nhánh chưa được duyệt cho đến khi tất cả các đỉnh đã được duyệt. Thuật toán DFS thường được sử dụng để phát hiện chu trình trong đồ thị: nếu tại mỗi nhánh ta bắt gặp một đỉnh đã được duyệt trước đó trước đó thì đồ thị sẽ có chu trình.
Các bước hiện thực DFS:

1. Khởi tạo mảng đánh dấu các phần tử đã được duyệt
2. Tạo hàm đệ quy nhận tham số đầu vào là thứ tự của đỉnh và mảng đánh dấu
3. Chọn một đỉnh bắt đầu và đánh dấu đỉnh đó đã được duyệt
4. Duyệt tất cả các đỉnh kề chưa được đánh dấu và gọi đệ quy với thứ tự của đỉnh kề

Độ phức tạp về thời gian: O(V+E) đối với cách biểu diễn đồ thị bằng danh sách kề, O(V^2) đối với ma trận kề

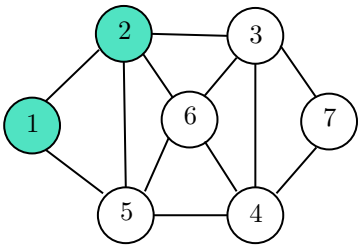
Ví dụ minh họa:



Chọn đỉnh 1 làm đỉnh bắt đầu, đánh dấu đã duyệt.

1						

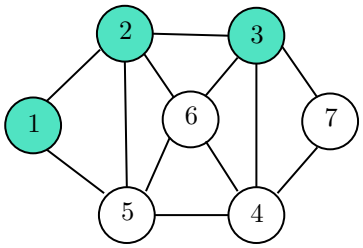
Thứ tự duyệt



Chọn đỉnh bất kì kề với đỉnh 1 mà chưa được duyệt, ở đây có 2 đỉnh là đỉnh 2 và đỉnh 5, ta có thể chọn bất kỳ đỉnh nào. Ở ví dụ này, ta sẽ chọn đỉnh nhỏ hơn trước . Vậy ta chọn đỉnh 2 trước và đỉnh 5 sau. Ở đây ta cần gọi hàm đệ quy với đỉnh 2 và đỉnh 5.

1	2					

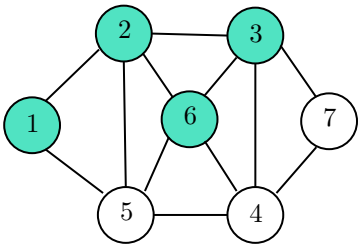
Thứ tự duyệt



Tương tự với đỉnh 2, ta có thể chọn bất kỳ đỉnh kề nào chưa được duyệt, vậy theo thứ tự ta chọn đỉnh 3 trước và đỉnh 6 sau. Tương tự ta cũng gọi hàm đệ quy với đỉnh 3 và đỉnh 6.

1	2	3				

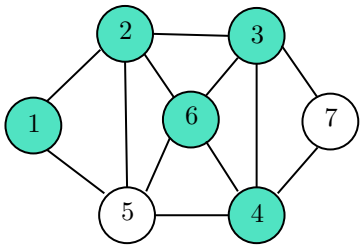
Thứ tự duyệt



Tiếp tục với đỉnh 3, ta chọn đỉnh 6 trước và đỉnh 7 sau. Gọi hàm đệ quy với hai đỉnh này.

1	2	3	6			

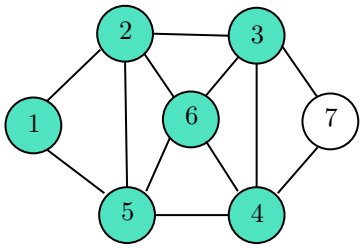
Thứ tự duyệt



Tiếp tục với đỉnh 6, ta chọn đỉnh 4 trước và đỉnh 5 sau và gọi hàm đệ quy.

1	2	3	6	4		

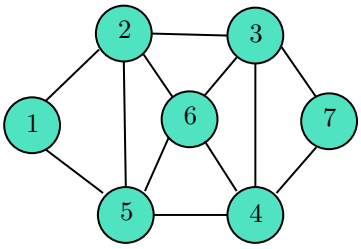
Thứ tự duyệt



Tiếp tục với đỉnh 4, ta chọn đỉnh 5 trước và đỉnh 7 sau và gọi hàm đệ quy.

1	2	3	6	4	5	

Thứ tự duyệt



Vì không còn đỉnh kề nào với đỉnh 5 chưa duyệt nên ta quay lại duyệt đỉnh 7. Sau khi đỉnh 7 được duyệt thì không còn đỉnh nào chưa được duyệt nên quay lui lên các đỉnh đã được gọi đệ quy còn lại ở các bước trên cũng sẽ không có chuyện gì xảy ra, ta có thứ tự duyệt theo chiều sâu.

1	2	3	6	4	5	7

Thứ tự duyệt

Đối với các thuật toán duyệt đồ thị như BFS và DFS, thứ tự duyệt có thể thay đổi tùy vào cách chọn đỉnh nào trước trong những đỉnh có vai trò tương tự nhau.

1.3.4 Thuật toán tìm đường đi ngắn nhất (shortest path)

Đối với đồ thị vô hướng có trọng số, có một vấn đề được đặt ra là làm thể nào để tìm được đường đi ngắn nhất từ một đỉnh cho trước. Để giải quyết vấn đề này, ta có thể sử dụng một số thuật toán như sau:

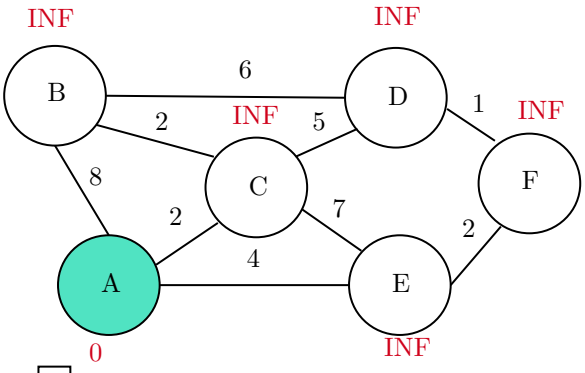
1.3.4.a Thuật toán Dijkstra

Các bước hiện thực thuật toán Dijkstra:

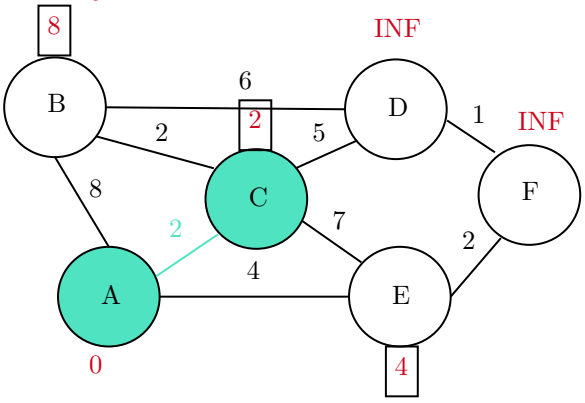
1. Khởi tạo một tập cây có đường đi ngắn nhất (shortest path tree set) sptSet để lưu trữ các đỉnh đã được thêm vào cây có đường đi ngắn nhất mà đường đi ngắn nhất tính từ nguồn tới đỉnh đó đã được tính toán và hoàn thành.
2. Khởi tạo một mảng chứa giá trị khoảng cách cho các đỉnh. Gán giá trị khoảng cách khởi tạo cho đỉnh nguồn là 0 và cho tất cả các đỉnh còn lại là ∞
3. Lặp lại các bước sau cho đến khi tất cả các đỉnh đều có trong tập cây có đường đi ngắn nhất.
    - Chọn đỉnh **u** có giá trị khoảng cách là nhỏ nhất trong tất cả các đỉnh chưa được thêm vào tập cây con có đường đi ngắn nhất
    - Thêm đỉnh đó vào cây có đường đi ngắn nhất
    - Cập nhật giá trị khoảng cách cho tất cả các đỉnh kề với đỉnh **u**: nếu tổng của giá trị khoảng cách của đỉnh u và trọng số của canh (u,v) với v là đỉnh kề với u nhỏ hơn giá trị khoảng cách của đỉnh u, cập nhật tổng giá trị trên cho giá trị khoảng cách của đỉnh v

Độ phức tạp về thời gian: O(V^2), tuy nhiên nếu hiện thực bằng việc sử dụng cấu trúc dữ liệu Heap hay còn gọi là hàng đợi ưu tiên có thể giảm độ phức tạp xuống O($E * log(V)$)

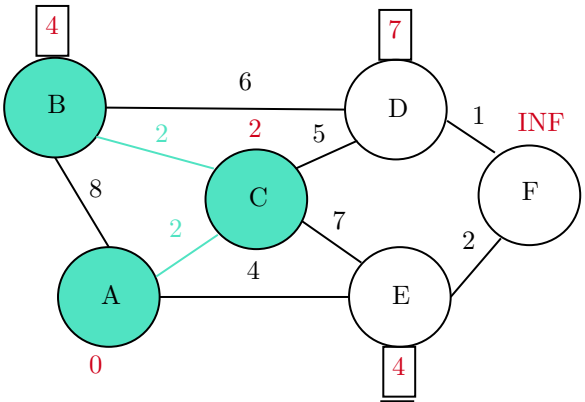
Ví dụ minh họa:



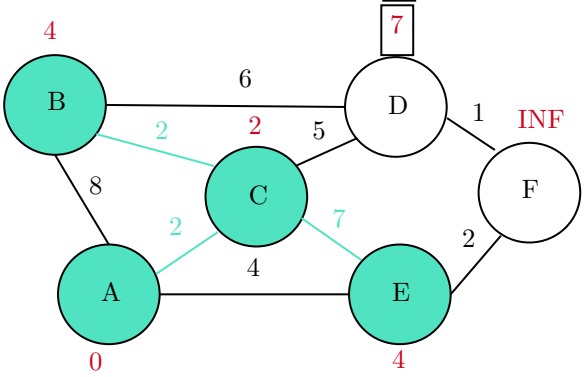
Chọn đỉnh A làm đỉnh bắt đầu và khởi tạo giá trị khoảng cách cho đỉnh A là 0, các đỉnh còn lại là INF (vô cực). Thêm đỉnh A vào cây có đường đi ngắn nhất.



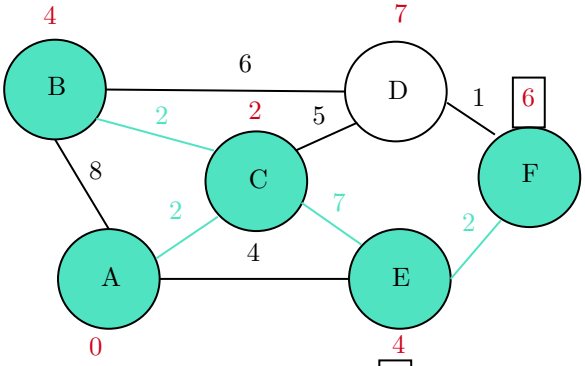
Cập nhật giá trị khoảng cách cho các đỉnh kề với đỉnh A gồm có: đỉnh B, đỉnh C và đỉnh E. Sau khi cập nhật, đỉnh C có giá trị khoảng cách nhỏ nhất. Thêm đỉnh C vào cây có đường đi ngắn nhất.



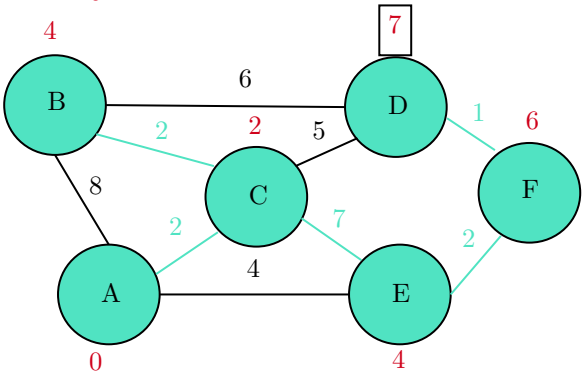
Cập nhật giá trị khoảng cách cho các đỉnh kề với đỉnh C mà chưa thuộc cây có đường đi ngắn nhất: gồm đỉnh B, đỉnh D và đỉnh E. Sau khi cập nhật, đỉnh B và đỉnh E cùng có giá trị khoảng cách nhỏ nhất. Ta chọn đỉnh B và thêm vào cây có đường đi ngắn nhất.



Cập nhật giá trị khoảng cách cho các đỉnh kề với đỉnh B mà chưa thuộc cây có đường đi ngắn nhất, ở đây ta chỉ có đỉnh D. Sau khi cập nhật, đỉnh E có giá trị khoảng cách nhỏ nhất. Thêm đỉnh E vào cây có đường đi ngắn nhất.



Cập nhật giá trị khoảng cách cho các đỉnh kề với đỉnh E mà chưa thuộc cây có đường đi ngắn nhất, ở đây ta chỉ có đỉnh F. Sau khi cập nhật, đỉnh F có giá trị khoảng cách nhỏ nhất. Thêm đỉnh F vào cây có đường đi ngắn nhất.



Cập nhật giá trị khoảng cách cho các đỉnh kề với đỉnh F mà chưa thuộc cây có đường đi ngắn nhất, ở đây ta chỉ còn đỉnh D. Thêm đỉnh D vào cây có đường đi ngắn nhất. Khi này tất cả các đỉnh đã được thêm vào. Ta có được đường đi ngắn nhất từ đỉnh A tới tất cả các đỉnh còn lại.

Thuật toán Dijkstra giúp tìm ra đường đi ngắn nhất từ một đỉnh nguồn tới tất cả các đỉnh còn lại với độ phức tạp về thời gian tương đối thấp với $O(V^2)$ hoặc $O(E * log(V))$. Tuy nhiên điều

kiện để thuật toán Dijkstra cho kết quả đúng là đồ thị phải có trọng số không âm.

1.3.4.b Thuật toán Bellman Ford

Các bước hiện thực thuật toán Bellman Ford:

1. Khởi tạo một mảng dis[] chứa giá trị khoảng cách cho các đỉnh. Gán giá trị khoảng cách khởi tạo cho đỉnh nguồn là 0 và cho tất cả các đỉnh còn lại là ∞
2. Lặp lại thao tác sau V-1 lần với V là số lượng đỉnh, đối với mỗi cặp cạnh (u,v):
   - Nếu dis[v] > dis[u] + trọng số cạnh nối u và v, cập nhật giá trị dis[v] = dis[u] + trọng số cạnh nối u và v
3. Lặp lại thao tác trên một lần nữa với mỗi cặp cạnh (u,v), nếu dis[v] > dis[u] + trọng số cạnh nối u và v $\implies$ đồ thị có chu trình âm. Điều này có nghĩa là số bước tối đa để xác định được đường đi ngắn nhất từ một đỉnh tới tất cả các đỉnh còn lại là V-1 lần, nếu ở bước thứ V đường đi ngắn nhất từ nguồn tới một đỉnh nào đó tiếp tục giảm xuống đồng nghĩa với việc tồn tại chu trình âm

Độ phức tạp về thời gian: vì ta lặp lại V-1 lần việc duyệt qua E cạnh nên độ phức tạp của thuật toán này là $O((V-1)*E)$ hay $O(V*E)$
Thuật toán Bellman-Ford có thể hoạt động với trọng số âm và có thể phát hiện cả chu trình âm, đây chính là một ưu điểm của thuật toán này so với thuật toán Dijkstra, tuy nhiên điều này chỉ đúng khi đồ thị đang xét là có hướng. Đối với đồ thị vô hướng, một cạnh (u,v) sẽ được xem như có hướng theo cả hai chiều (u,v) và (v,u), chính vì vậy chỉ cần cạnh mang trọng số âm thì sẽ tạo thành chu trình âm khi sử dụng thuật toán này. Vậy nên, trong trường hợp này, khi ta chỉ xét đồ thị vô hướng, điều kiện luôn cần thiết là đồ thị phải có trọng số không âm, nếu không việc tìm đường đi ngắn nhất sẽ không còn ý nghĩa vì ta có thể lặp đi lặp lại việc đi qua đi lại một cạnh có trọng số âm. Do đó, khi đã đảm bảo được điều kiện trọng số không âm, thuật toán Dijkstra được đề cập ở trên lại tỏ ra hiệu quả hơn khi độ phức tạp về thời gian được giảm xuống rất thấp.

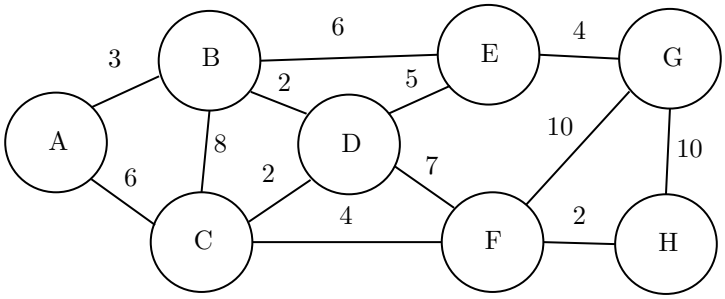
1.3.5 Thuật toán Floyd Warshall

Các bước hiện thực thuật toán Floyd Warshall:

1. Biểu diễn đồ thị dưới dạng một ma trận kề (thông thường sử dụng mảng dis[][]), nếu 2 đỉnh i và j kề nhau thì giá trị tại hàng i cột j và hàng j cột i sẽ là trọng số của cạnh nối hai đỉnh, nếu 2 đỉnh không kề nhau thì trọng số sẽ là ∞
2. Lần lượt chọn tất cả các đỉnh theo thứ tự làm đỉnh trung gian trong đường đi ngắn nhất giữa các đỉnh khác
3. Tại mỗi cặp đỉnh (i, j) có hai trường hợp có thể xảy ra
   - Nếu dis[i][j] > dis[i][k] + dis[k][j] thì k là đỉnh trung gian trong đường đi ngắn nhất giữa cặp đỉnh (i, j), thay giá trị dis[i][j] bằng giá trị dis[i][k] + dis[k][j]
   - Ngược lại, k không là đỉnh trung gian trong đường đi ngắn nhất giữa hai đỉnh (i, j), giữ nguyên giá trị dis[i][j]

Độ phức tạp về thời gian: vì ta chọn V đỉnh làm đỉnh trung gian, tại mỗi đỉnh ta phải lặp V^2 phần tử của ma trận nên độ phức tạp của thuật toán này là O(V^3)

Ví dụ minh họa:
Cho đồ thị dưới đây:



Ta có ma trận biểu diễn đồ thị trên:

$$A^0 = \begin{bmatrix} & A & B & C & D & E & F & G & H \\ A & 0 & 3 & 6 & INF & INF & INF & INF & INF \\ B & 3 & 0 & 8 & 2 & 6 & INF & INF & INF \\ C & 6 & 8 & 0 & 2 & INF & 4 & INF & INF \\ D & INF & 2 & 2 & 0 & 5 & 7 & INF & INF \\ E & INF & 6 & INF & 5 & 0 & INF & 4 & INF \\ F & INF & INF & 4 & 7 & INF & 0 & 10 & 2 \\ G & INF & INF & INF & INF & 4 & 10 & 0 & 10 \\ H & INF & INF & INF & INF & INF & 2 & 10 & 0 \end{bmatrix}$$

Lần lượt chọn các đỉnh làm đỉnh trung gian: (đỉnh trung gian được chọn sẽ có hàng và cột tương ứng được tô màu đỏ, các cập nhật về giá trị sẽ được tô màu xanh)

$$A^1 = \begin{bmatrix} & \mathbf{A} & B & C & D & E & F & G & H \\ \mathbf{A} & \mathbf{0} & \mathbf{3} & \mathbf{6} & \mathbf{INF} & \mathbf{INF} & \mathbf{INF} & \mathbf{INF} & \mathbf{INF} \\ B & \mathbf{3} & 0 & 8 & 2 & 6 & INF & INF & INF \\ C & \mathbf{6} & 8 & 0 & 2 & INF & 4 & INF & INF \\ D & \mathbf{INF} & 2 & 2 & 0 & 5 & 7 & INF & INF \\ E & \mathbf{INF} & 6 & INF & 5 & 0 & INF & 4 & INF \\ F & \mathbf{INF} & INF & 4 & 7 & INF & 0 & 10 & 2 \\ G & \mathbf{INF} & INF & INF & INF & 4 & 10 & 0 & 10 \\ H & \mathbf{INF} & INF & INF & INF & INF & 2 & 10 & 0 \end{bmatrix}$$

$$A^2 = \begin{bmatrix} & A & \mathbf{B} & C & D & E & F & G & H \\ A & 0 & \mathbf{3} & 6 & 5 & 9 & INF & INF & INF \\ \mathbf{B} & \mathbf{3} & \mathbf{0} & \mathbf{8} & \mathbf{2} & \mathbf{6} & \mathbf{INF} & \mathbf{INF} & \mathbf{INF} \\ C & 6 & \mathbf{8} & 0 & 2 & 14 & 4 & INF & INF \\ D & 5 & \mathbf{2} & 2 & 0 & 5 & 7 & INF & INF \\ E & 9 & \mathbf{6} & 14 & 5 & 0 & INF & 4 & INF \\ F & INF & \mathbf{INF} & 4 & 7 & INF & 0 & 10 & 2 \\ G & INF & \mathbf{INF} & INF & INF & 4 & 10 & 0 & 10 \\ H & INF & \mathbf{INF} & INF & INF & INF & 2 & 10 & 0 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} & A & B & \mathbf{C} & D & E & F & G & H \\ A & 0 & 3 & \mathbf{6} & 5 & 9 & 10 & INF & INF \\ B & 3 & 0 & \mathbf{8} & 2 & 6 & 12 & INF & INF \\ \mathbf{C} & \mathbf{6} & \mathbf{8} & \mathbf{0} & \mathbf{2} & \mathbf{14} & \mathbf{4} & \mathbf{INF} & \mathbf{INF} \\ D & 5 & 2 & \mathbf{2} & 0 & 5 & 6 & INF & INF \\ E & 9 & 6 & \mathbf{14} & 5 & 0 & 18 & 4 & INF \\ F & 10 & 12 & \mathbf{4} & 6 & 18 & 0 & 10 & 2 \\ G & INF & INF & \mathbf{INF} & INF & 4 & 10 & 0 & 10 \\ H & INF & INF & \mathbf{INF} & INF & INF & 2 & 10 & 0 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} & A & B & C & \mathbf{D} & E & F & G & H \\ A & 0 & 3 & 6 & \mathbf{5} & 9 & 10 & INF & INF \\ B & 3 & 0 & 4 & \mathbf{2} & 6 & 8 & INF & INF \\ C & 6 & 4 & 0 & \mathbf{2} & 7 & 4 & INF & INF \\ \mathbf{D} & \mathbf{5} & \mathbf{2} & \mathbf{2} & \mathbf{0} & \mathbf{5} & \mathbf{6} & \mathbf{INF} & \mathbf{INF} \\ E & 9 & 6 & 7 & \mathbf{5} & 0 & 11 & 4 & INF \\ F & 10 & 8 & 4 & \mathbf{6} & 11 & 0 & 10 & 2 \\ G & INF & INF & INF & \mathbf{INF} & 4 & 10 & 0 & 10 \\ H & INF & INF & INF & \mathbf{INF} & INF & 2 & 10 & 0 \end{bmatrix}$$

$$A^5 = \begin{bmatrix} & A & B & C & D & \mathbf{E} & F & G & H \\ A & 0 & 3 & 6 & 5 & \mathbf{9} & 10 & 13 & INF \\ B & 3 & 0 & 4 & 2 & \mathbf{6} & 8 & 10 & INF \\ C & 6 & 4 & 0 & 2 & \mathbf{7} & 4 & 11 & INF \\ D & 5 & 2 & 2 & 0 & \mathbf{5} & 6 & 9 & INF \\ \mathbf{E} & \mathbf{9} & \mathbf{6} & \mathbf{7} & \mathbf{5} & \mathbf{0} & \mathbf{11} & \mathbf{4} & \mathbf{INF} \\ F & 10 & 8 & 4 & 6 & \mathbf{11} & 0 & 10 & 2 \\ G & 13 & 10 & 11 & 9 & \mathbf{4} & 10 & 0 & 10 \\ H & INF & INF & INF & INF & \mathbf{INF} & 2 & 10 & 0 \end{bmatrix} \quad (1)$$

$$A^6 = \begin{bmatrix} & A & B & C & D & E & \mathbf{F} & G & H \\ A & 0 & 3 & 6 & 5 & 9 & \mathbf{10} & 13 & 12 \\ B & 3 & 0 & 4 & 2 & 6 & \mathbf{8} & 10 & 10 \\ C & 6 & 4 & 0 & 2 & 7 & \mathbf{4} & 11 & 6 \\ D & 5 & 2 & 2 & 0 & 5 & \mathbf{6} & 9 & 8 \\ E & 9 & 6 & 7 & 5 & 0 & \mathbf{11} & 4 & 13 \\ \mathbf{F} & \mathbf{10} & \mathbf{8} & \mathbf{4} & \mathbf{6} & \mathbf{11} & \mathbf{0} & \mathbf{10} & \mathbf{2} \\ G & 13 & 10 & 11 & 9 & 4 & \mathbf{10} & 0 & 10 \\ H & 12 & 10 & 6 & 8 & 13 & \mathbf{2} & 10 & 0 \end{bmatrix}$$

$$A^7 = \begin{bmatrix} & A & B & C & D & E & F & \mathbf{G} & H \\ A & 0 & 3 & 6 & 5 & 9 & 10 & \mathbf{13} & 12 \\ B & 3 & 0 & 4 & 2 & 6 & 8 & \mathbf{10} & 10 \\ C & 6 & 4 & 0 & 2 & 7 & 4 & \mathbf{11} & 6 \\ D & 5 & 2 & 2 & 0 & 5 & 6 & \mathbf{9} & 8 \\ E & 9 & 6 & 7 & 5 & 0 & 11 & \mathbf{4} & 13 \\ F & 10 & 8 & 4 & 6 & 11 & 0 & \mathbf{10} & 2 \\ \mathbf{G} & \mathbf{13} & \mathbf{10} & \mathbf{11} & \mathbf{9} & \mathbf{4} & \mathbf{10} & \mathbf{0} & \mathbf{10} \\ H & 12 & 10 & 6 & 8 & 13 & 2 & \mathbf{10} & 0 \end{bmatrix}$$

$$A^8 = \begin{bmatrix} & A & B & C & D & E & F & G & \mathbf{H} \\ A & 0 & 3 & 6 & 5 & 9 & 10 & 13 & \mathbf{12} \\ B & 3 & 0 & 4 & 2 & 6 & 8 & 10 & \mathbf{10} \\ C & 6 & 4 & 0 & 2 & 7 & 4 & 11 & \mathbf{6} \\ D & 5 & 2 & 2 & 0 & 5 & 6 & 9 & \mathbf{8} \\ E & 9 & 6 & 7 & 5 & 0 & 11 & 4 & \mathbf{13} \\ F & 10 & 8 & 4 & 6 & 11 & 0 & 10 & \mathbf{2} \\ G & 13 & 10 & 11 & 9 & 4 & 10 & 0 & \mathbf{10} \\ \mathbf{H} & \mathbf{12} & \mathbf{10} & \mathbf{6} & \mathbf{8} & \mathbf{13} & \mathbf{2} & \mathbf{10} & \mathbf{0} \end{bmatrix}$$

Ma trận A^8 là kết quả của thuật toán Floyd-Warshall, cho kết quả là đường đi ngắn nhất giữa tất cả các cặp đỉnh trong đồ thị.

Thuật toán Floyd-Warshall có thể tìm được đường đi ngắn nhất giữa tất cả các cặp đỉnh tuy nhiên phải đánh đổi lại việc độ phức tạp về thời gian khá lớn.

1.3.6 Thuật toán tìm cây khung có trọng số nhỏ nhất (minimum spanning tree)

Cây khung là một tập con của một **đồ thị liên thông** G, thỏa điều kiện tất cả các cạnh đều được kết nối với nhau, nghĩa là ta có thể duyệt đến một cạnh bất kì từ một cạnh cụ thế hoặc qua những cạnh trung gian. Bên cạnh đó, cây khung không được tồn tại chu trình. Cho một đồ thị vô hướng, liên thông và có trọng số G, cây khung là một tập con của G là một cây kết nối tất cả các đỉnh của đồ thị, một đồ thị có thể có nhiều cây khung. **Cây khung có trọng số nhỏ nhất** là cây khung có tổng trọng số của các cạnh tạo thành nhỏ hơn hoặc bằng so với bất kỳ cây khung nào của đồ thị. Ứng dụng của việc tìm cây khung nhỏ nhất có thể kể đến: tối ưu việc lắp đặt hệ thống mạng, phân tích cụm (cluster analysis),...

1.3.6.a Thuật toán Kruskal

Ý tưởng: Để tìm cây khung có trọng số nhỏ nhất với thuật toán Kruskal, đầu tiên ta sắp xếp trọng số của các cạnh thành một dãy không giảm, sau đó ta duyệt từ đầu dãy, nếu cạnh đó tạo thành chu trình thì ta xóa nó ra khỏi dãy, ngược lại nếu không tạo thành chu trình thì thêm nó vào cây khung và cũng xóa nó ra khỏi dãy. Tiếp tục lặp lại thao tác này đến khi số lượng cạnh của cây khung đã đủ (số lượng cạnh = số đỉnh - 1). Các bước hiện thực thuật toán Kruskal:

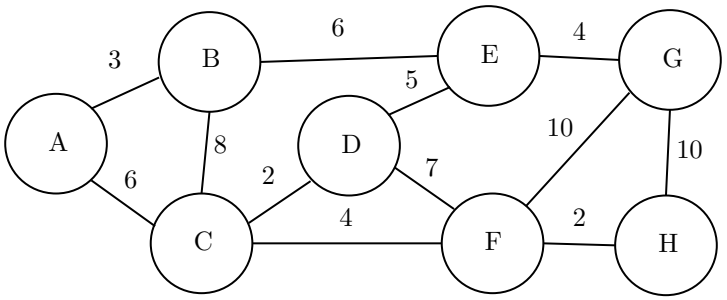
1. Sắp xếp trọng số của các cạnh thành một dãy không giảm
2. Lựa chọn lần lượt các cạnh trong dãy ở **bước 1**. Nếu cạnh đó không tạo thành chu trình, thêm vào cây khung. Ngược lại, bỏ qua cạnh đó.
3. Lặp lại **bước 2** đến khi cây khung đủ (số đỉnh - 1) cạnh

Độ phức tạp về thời gian: do độ phức tạp của thuật toán phụ thuộc vào hai phần:

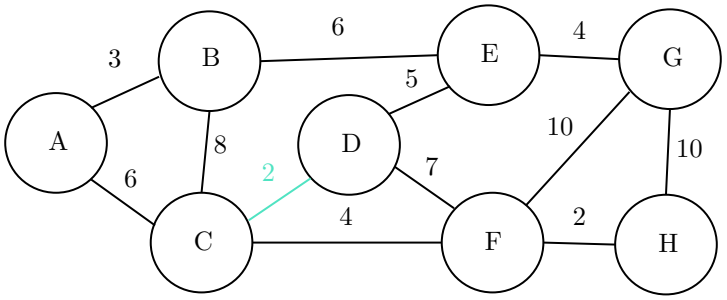
- Sắp xếp các cạnh theo trọng số không giảm, với độ phức tạp O($E * log(E)$)
- Duyệt qua tất cả các cạnh và quyết định xem có nên thêm vào cây khung hay không. Việc quyết định có độ phức tạp chỉ là O(1) nên độ phức tạp của phần này là O(E)

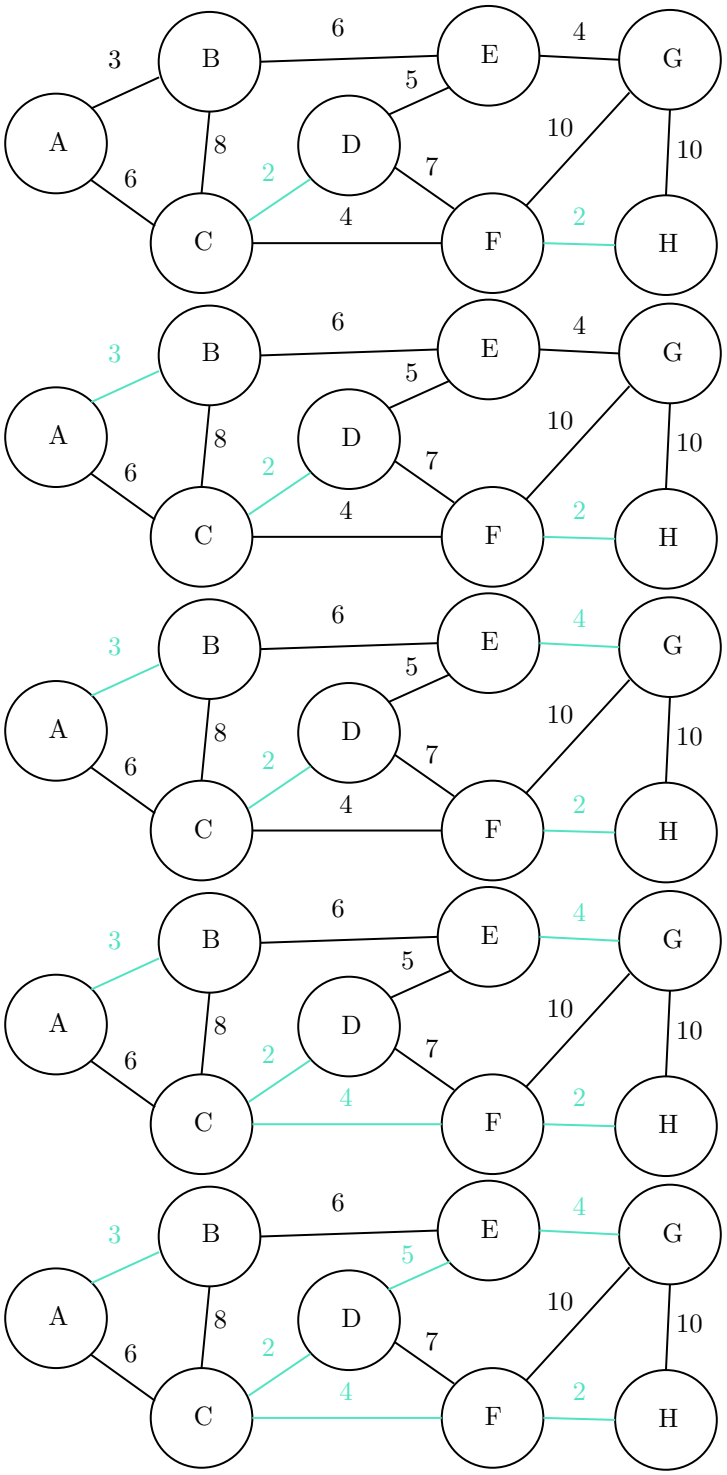
Do đó độ phức tạp của thuật toán Kruskal để tìm cây khung có trọng số nhỏ nhất là O($E*log(E)$) hay O($E * log(V)$)

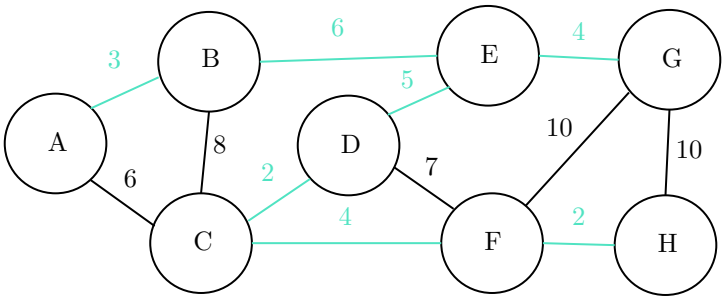
Ví dụ minh họa:



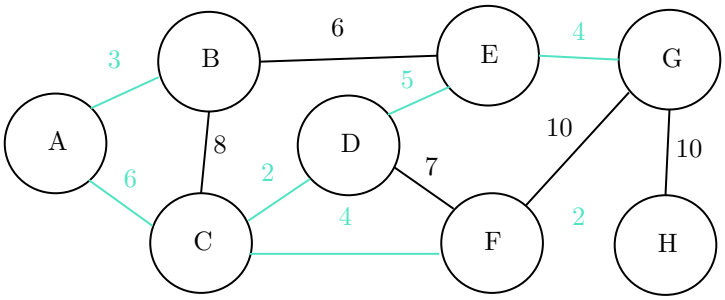
Cạnh	Trọng số	Thuộc cây khung
(C,D)	2	
(F,H)	2	
(A,B)	3	
(C,F)	4	
(E,G)	4	
(D,E)	5	
(A,C)	6	
(B,E)	6	
(D,F)	7	
(B,C)	8	
(F,G)	10	
(G,H)	10	




A
B
C
D
E
F
G
H
3
6
8
6
2
4
5
7
10
4
10
2



hoặc



Cạnh	Trọng số	Thuộc cây khung
(C,D)	2	Có
(F,H)	2	Có
(A,B)	3	Có
(C,F)	4	Có
(E,G)	4	Có
(D,E)	5	Có
(A,C)	6	hoặc cạnh (B,E)
(B,E)	6	hoặc cạnh (A,C)
(D,F)	7	Cây khung đã đủ
(B,C)	8	Cây khung đã đủ
(F,G)	10	Cây khung đã đủ
(G,H)	10	Cây khung đã đủ

1.3.6.b Thuật toán Prim

Ý tưởng: Để tìm cây khung có trọng số nhỏ nhất với thuật toán Prim, đầu tiên ta chọn và đánh dấu một đỉnh bắt đầu, sau đó đối với đỉnh kề với những đỉnh đã chọn, ta chọn và đánh dấu đỉnh nào có thể tạo thành cạnh có trọng số nhỏ nhất. Tiếp tục lặp lại thao tác này đến khi số lượng cạnh của cây khung đã đủ (số lượng cạnh = số đỉnh - 1). Gần giống với cách thức hiện thực của thuật toán Dijkstra, các bước hiện thực thuật toán Prim:

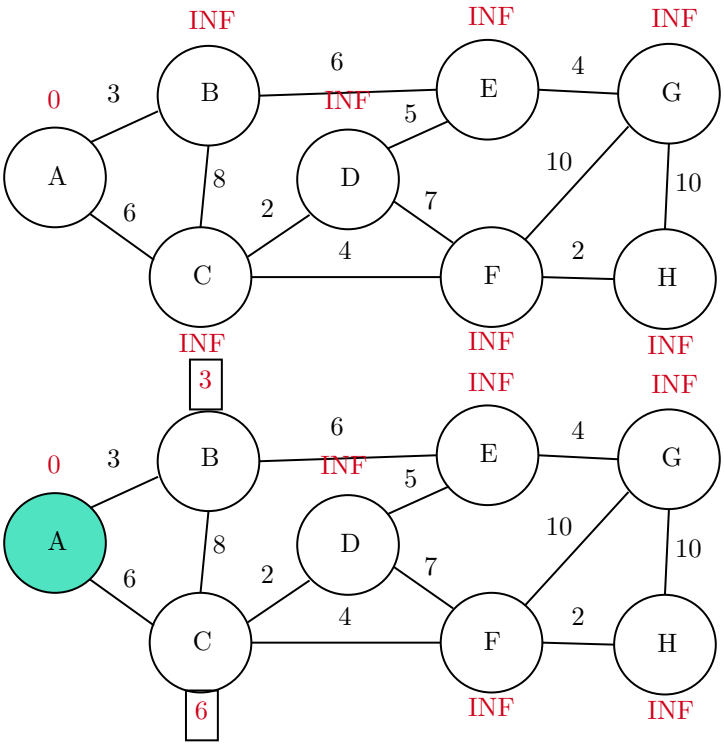
1. Khởi tạo một tập **mstSet** để theo dõi các đỉnh có trong cây khung có trọng số nhỏ nhất
2. Khởi tạo một mảng chưa giá trị cho các đỉnh. Giá trị khởi tạo cho đỉnh nguồn là 0 và cho tất cả các đỉnh còn lại là ∞

3. Lặp lại các bước sau cho đến khi tất cả các đỉnh đều có trong tập **mstSet**:
    - Chọn đỉnh u là đỉnh có giá trị nhỏ nhất và chưa nằm trong tập **mstSet**
    - Thêm đỉnh u vào tập **mstSet**
    - Cập nhật giá trị cho tất cả các đỉnh kề với đỉnh u và không thuộc tập **mstSet**: nếu trọng số cạnh (u,v) với v là đỉnh kề với u có giá trị nhỏ hơn giá trị đang có ở đỉnh v, cập nhật giá trị đó cho giá trị đang có ở đỉnh v

Độ phức tạp về thời gian: ở thuật toán Prim, độ phức tạp của **bước 3** là đáng kể hơn cả, ta xem xét ba cách hiện thực:

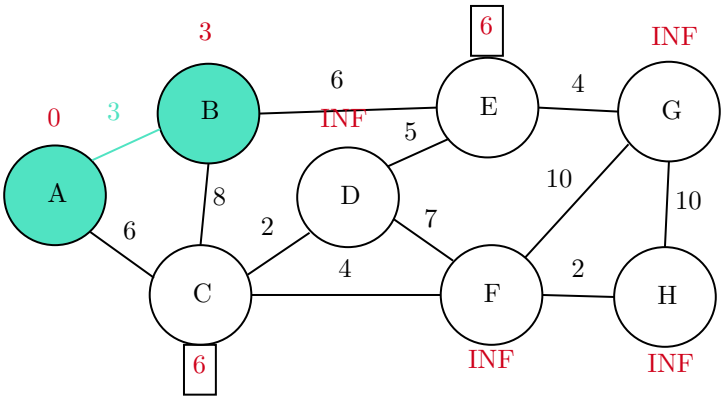
- Sử dụng ma trận kề để biểu diễn đồ thị, cách này có độ phức tạp là $O(V^2)$
- Sử dụng danh sách kề và Binary Heap (đống nhị phân), cách này có độ phức tạp là $O(E * log(V))$
- Sử dụng danh sách kề và Fibonacci Heap (đống Fibonacci), cách này có độ phức tạp là $O(E + log(V))$

Ví dụ minh họa:

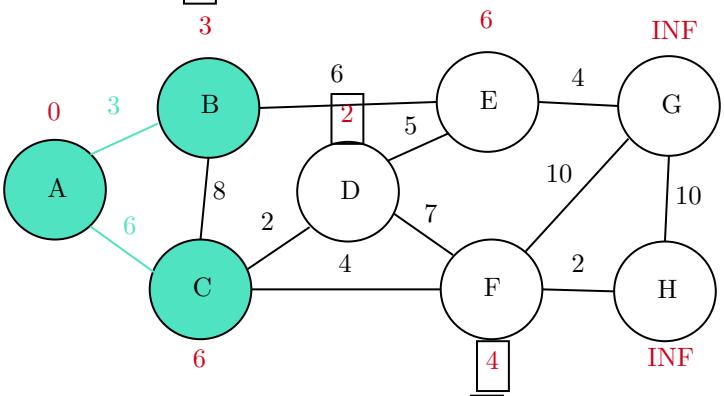


Chọn đỉnh A làm đỉnh bắt đầu, khởi tạo giá trị 0 cho đỉnh A và INF (vô cực) cho tất cả các đỉnh còn lại.

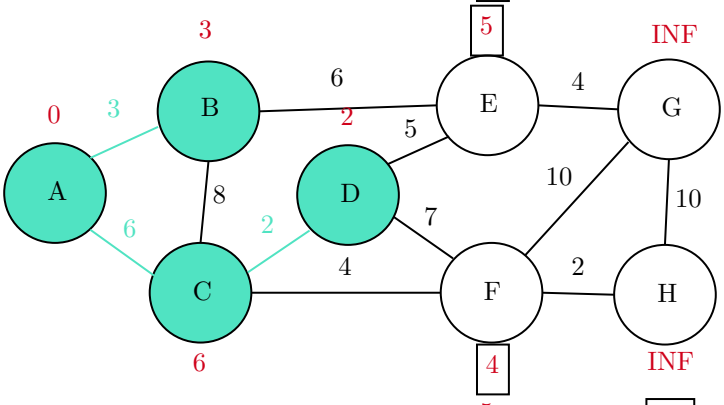
Đỉnh A có giá trị nhỏ nhất, ta thêm đỉnh A vào cây khung, cập nhật giá trị cho tất cả các đỉnh kề đỉnh A.



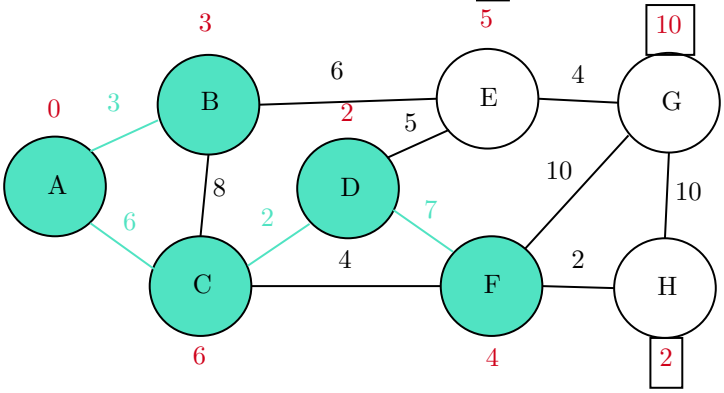
Đỉnh B có là đỉnh có giá trị nhỏ nhất mà không thuộc cây khung, ta thêm đỉnh B vào cây khung, cập nhật giá trị cho tất cả các đỉnh kề đỉnh B mà chưa thuộc cây khung.



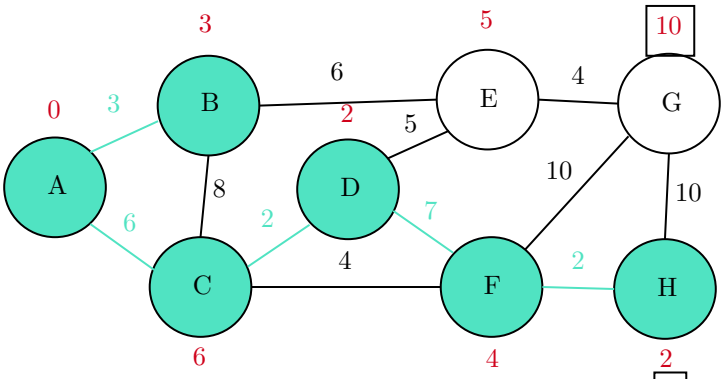
Đỉnh C và đỉnh E đều là đỉnh có giá trị nhỏ nhất mà không thuộc cây khung, ta có thể chọn bất kì đỉnh nào. Ta chọn đỉnh C, thêm đỉnh C vào cây khung, cập nhật giá trị cho tất cả các đỉnh kề đỉnh C mà chưa thuộc cây khung



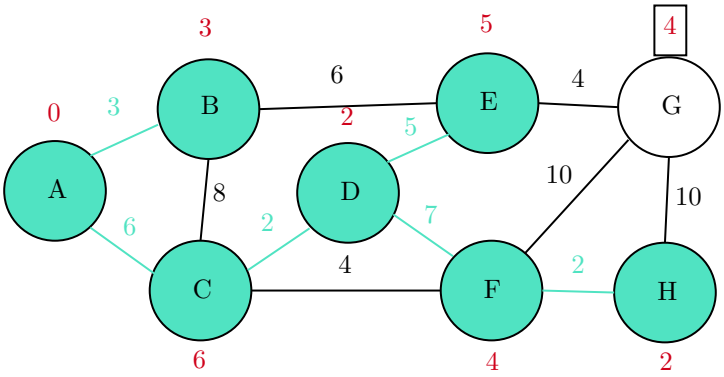
Đỉnh D có là đỉnh có giá trị nhỏ nhất mà không thuộc cây khung, ta thêm đỉnh D vào cây khung, cập nhật giá trị cho tất cả các đỉnh kề đỉnh D mà chưa thuộc cây khung.



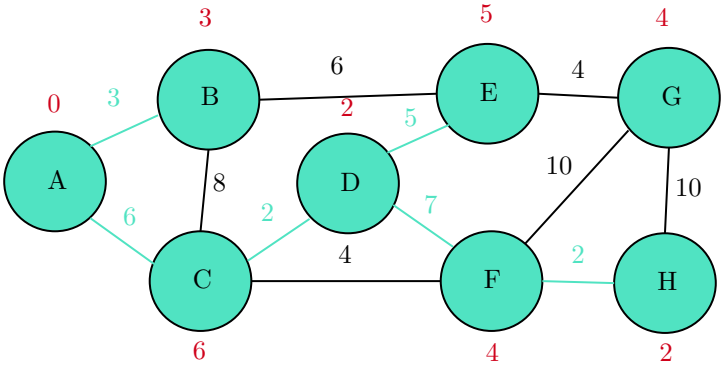
Đỉnh F có là đỉnh có giá trị nhỏ nhất mà không thuộc cây khung, ta thêm đỉnh F vào cây khung, cập nhật giá trị cho tất cả các đỉnh kề đỉnh F mà chưa thuộc cây khung.



Đỉnh H có là đỉnh có giá trị nhỏ nhất mà không thuộc cây khung, ta thêm đỉnh H vào cây khung, cập nhật giá trị cho tất cả các đỉnh kề đỉnh H mà chưa thuộc cây khung.



Đỉnh E có là đỉnh có giá trị nhỏ nhất mà không thuộc cây khung, ta thêm đỉnh E vào cây khung, cập nhật giá trị cho tất cả các đỉnh kề đỉnh E mà chưa thuộc cây khung.



Đỉnh G là đỉnh cuối cùng, ta thêm đỉnh G vào cây khung. Tại đây, tất cả các đỉnh đã được thêm vào cây khung và ta có được cây khung hoàn chỉnh.

Một đồ thị có thể tồn tại nhiều cây khung có trọng số nhỏ nhất, như ví dụ trên, ta có thể chọn cạnh (A,C) hoặc (B,E) vào cây khung tùy quy tắc do ta đặt ra, mỗi trường hợp sẽ tạo ra một cây khung có trọng số nhỏ nhất khác nhau. Tuy nhiên, khác nhau về hình dạng nhưng tổng trọng số của cây khung vẫn không thay đổi và là nhỏ nhất. Như một đồ thị cùng được sử dụng trong hai ví dụ của hai thuật toán tìm cây khung có trọng số nhỏ nhất ở trên, kết quả ta thu được là tương tự nhau.

1.3.7 Thuật toán xác định Đồ thị phân đôi (Bipartite Graph)

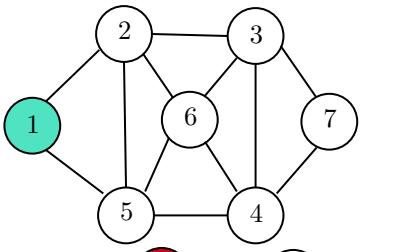
Định nghĩa: Một đơn đồ thị được gọi là đồ thị phân đôi nếu tập các đỉnh V của nó có thể được chia thành hai tập không giao nhau V_1 và V_2 sao cho bất kì một cạnh nào thuộc đồ thị đều kết nối một đỉnh thuộc V_1 và một đỉnh thuộc V_2

Giải thuật xác định đồ thị phân đôi: để xác định đồ thị có phải là đồ thị phân đôi hay không, ta cần làm những bước sau:

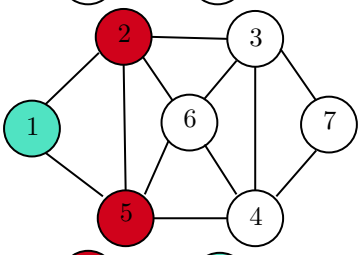
1. Đánh dấu tất cả các đỉnh là chưa duyệt
2. Bắt đầu duyệt đồ thị bằng thuật toán BFS từ một đỉnh bất kì, nếu đỉnh đó chưa được duyệt đánh dấu đỉnh đó vào tập đỉnh thứ nhất, tất cả các đỉnh kề với nó vào tập đỉnh thứ hai
3. Tiếp tục quá trình với tập đỉnh thay đổi qua mỗi đỉnh được duyệt.
4. Nếu tại một đỉnh đã nằm trong một tập đỉnh mà được đánh dấu vào tập còn lại $\implies$ đồ thị không phải là đồ thị phân đôi
5. Nếu tất cả các đỉnh đều đã được xếp vào một trong hai tập $\implies$ đồ thị là đồ thị phân đôi

Độ phức tạp về thời gian: vì cũng dựa trên các thuật toán duyệt đồ thị nên độ phức tạp của thuật toán cũng là O(V+E) với cách biểu diễn bằng danh sách kề và O(V^2) với ma trận kề

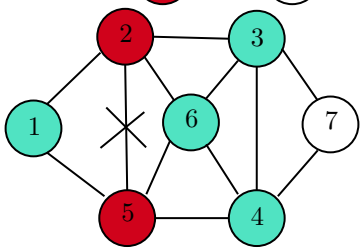
Ví dụ minh họa: Ta xét hai ví dụ sau



Chọn đỉnh 1 là đỉnh bắt đầu, đánh dấu đỉnh 1 thuộc nhóm 1 (màu xanh).

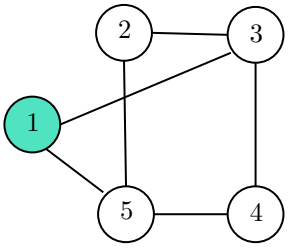


Đánh dấu các đỉnh kề với đỉnh 1 thuộc nhóm 2 (màu đỏ).

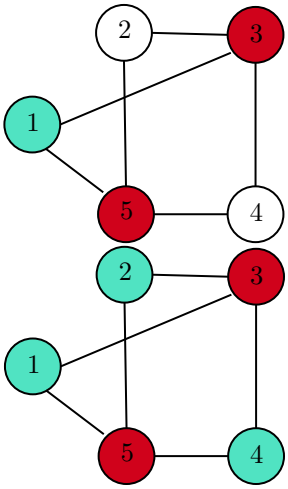


Đối với các đỉnh kề của các đỉnh thuộc nhóm 2, các đỉnh đó phải thuộc nhóm 1. Tuy nhiên đỉnh 2 và 5 kề nhau nhưng lại cùng thuộc nhóm 2, đồng nghĩa với việc đồ thị này không phải là đồ thị phân đôi.

Ví dụ tiếp theo:



Chọn đỉnh 1 là đỉnh bắt đầu, đánh dấu đỉnh 1 thuộc nhóm 1 (màu xanh).



Đánh dấu các đỉnh kề với đỉnh 1 thuộc nhóm 2 (màu đỏ).

Đánh dấu các đỉnh kề các đỉnh thuộc nhóm 2 ở bước trước thuộc về nhóm 1 (đỉnh 2,4). Các đỉnh thuộc nhóm 1 vừa được đánh dấu có các đỉnh kề đều thuộc nhóm 2. Không có đỉnh nào cùng thuộc cả 2 nhóm đồng nghĩa với việc đây là đồ thị phân đôi.

2 Các bài toán

2.1 Bài toán 1 (Transportation)

Problem. *Vấn đề giao thông là một trong những vấn đề lớn trong quy hoạch của các thành phố lớn hiện nay. Giả sử các tuyến đường trong thành phố đều được kết nối với các giao lộ, mỗi giao lộ đều có tên riêng và tùy vào lưu lượng phương tiện giao thông mà qua mỗi giao lộ sẽ cần một thời gian nhất định để di chuyển. Bên cạnh đó, thành phố sẽ đăng tải lên cổng thông tin giao thông của thành phố một danh sách các tuyến đường sẽ bị tạm dừng để sửa chữa theo các tháng trong năm. Yêu cầu dành cho sinh viên:*
a) Xây dựng cấu trúc dữ liệu phù hợp dựa trên những cấu trúc dữ liệu đã được hiện thực sẵn.
b) Viết hàm ***int minimumTime(string first, string second, int month)*** *nhận vào tham số đầu vào hai chuỗi là tên của hai giao lộ và thời điểm đang xét là một tháng trong năm, kết quả trả về là thời gian ngắn nhất để di chuyển giữa hai giao lộ.*
c) Để có thể dễ tra cứu tuyến đường tối ưu khi di chuyển trong thành phố, cổng thông tin giao thông đã cung cấp một bảng các tuyến đường ngắn nhất giữa hai điểm bất kì trong thành phố và được cập nhật liên tục qua các tháng. Viết hàm ***void allShortestPath(int month)*** *nhận tham số đấu vào là một tháng và in ra ma trận thể hiện đường đi ngắn nhất giữa tất cả các cặp đỉnh*

Solution. *File code giải bài toán trên được đính kèm có tên Ex1.cpp*

2.2 Bài toán 2 (Cut vertex)

Problem. *Một quốc gia sắp bị xâm lược có các kho lương thực ở nhiều nơi trên đất nước. Giữa các kho lương đều có những tuyến đường bí mật vận chuyển lương thực cho nhau khi có biến cố xảy ra. Biết được điều này, tướng xâm lược đã lén lấy được bản đồ kho lương, với mục đích tìm những kho lương quan trọng để tập kích và cô lập những kho lương còn lại. Nói cách khác, việc cô lập xảy ra khi một/nhiều kho lương không thể có cách nào nhận được tiếp tế từ một/nhiều kho lương khác. Mỗi kho lương sẽ gồm có tên và số lượng binh lính đang trấn giữ ở kho lương đó. Một kho lương chỉ có thể bị đánh chiếm khi có ít hơn số lượng binh lính so với quân xâm lược, và mỗi khi bị đánh chiếm thì quân xâm lược sẽ bị mất đi một số lượng binh lính tương đương với số quân trấn giữ tại kho lương đó. Mật thám của quốc gia sắp bị xâm lược cũng lén lấy được kế hoạch tấn công của quân xâm lược là một danh sách các kho lương sẽ tấn công. Các kho lương*

trong danh sách trên đều sẽ là các kho lương có khả năng cô lập một/nhiều kho lương khác. Yêu cầu dành cho sinh viên:
a) Xây dựng câu trúc dữ liệu phù hợp dựa trên những cấu trúc dữ liệu đã được hiện thực sẵn.
b) Để biết nên đánh chiếm những kho lương nào là có lợi nhất, tướng xâm lược cần biết những kho lương nào có khả năng tạo ra sự cô lập. Viết hàm ***DLinkedList<string> criticalPoint()*** *trả về một danh sách tên các kho lương trọng yếu*
c) Viết hàm ***DLinkedList*<T> defense()*** *trả về một danh sách tương tự các kho lương với số binh lính cần bổ sung để các kho lương đó không bị chiếm đóng*

Solution. *File code giải bài toán trên được đính kèm có tên Ex2.cpp*

2.3 Bài toán 3 (Min Cost to Connect All)

Problem. *Một trò chơi điện tử gồm các điểm xuất hiện ngẫu nhiên trên màn hình, để chiến thắng người chơi phải nối tất cả các điểm có trên màn hình lại với nhau bằng bút mực với số lượng mực có hạn, vậy nên người chơi phải tìm ra cách nối sao cho tổng số lực mực tiêu tốn là ít nhất. Mỗi điểm trên màn hình đều được biểu diễn bằng một tọa độ (x,y) với x, y không âm. Lượng mực tiêu tốn để nối hai điểm bất kì sẽ là khoảng cách Manhattan giữa hai điểm đó. Yêu cầu cho sinh viên:*
a) Xây dựng câu trúc dữ liệu phù hợp dựa trên những cấu trúc dữ liệu đã được hiện thực sẵn.
b) Viết hàm số ***int hasMST(int cost)*** *nhận vào một số nguyên là lượng mực tối đa cho phép, nếu tồn tại một cách nối giúp người chơi chiến thắng và sử dụng không nhiều hơn lượng mực tối đa, trả về số lượng mực chưa sử dụng khi chiến thắng trò chơi. Ngược lại trả về -1*
c) Sau khi chiến thắng trò chơi, người chơi sẽ có thể chơi thêm một trò chơi phụ: màn hình sẽ xuất hiện thêm một điểm ngẫu nhiên nữa, để chiến thắng phần chơi này người chơi cần xác định xem có tối đa bao nhiêu điểm có thể được nối với điểm vừa xuất hiện với số mực còn lại. Viết hàm ***int maximumVertex(int x, int y, int cost)*** *với (x,y) là tọa độ của đỉnh mới xuất hiện và* ***cost*** *là lượng mực còn lại.*

Solution. *File code giải bài toán trên được đính kèm có tên Ex3.cpp*

2.4 Bài toán 4 (Stable Marriage Problem)

Problem. *Cho một số lượng nam và nữ tới tuổi kết hôn như nhau. Mỗi người trong số họ đều sẽ xếp hạng độ yêu thích tất cả những người thuộc giới còn lại theo một danh sách. Nhiệm vụ là phải bắt cặp nhóm người này để ai cũng có vợ/chồng với một điều kiện là không có bất kì hai người khác giới nào cùng thích người còn lại hơn vợ/chồng của chính mình (dạng bài toán stable matching). Yêu cầu dành cho sinh viên:*
a) Xây dựng câu trúc dữ liệu phù hợp dựa trên những cấu trúc dữ liệu đã được hiện thực sẵn.
b) Viết hàm ***string fiancee(string people)*** *nhận và tên một người và trả về tên người được chọn để kết hợp với người đó.*

Solution. *File code giải bài toán trên được đính kèm có tên Ex4.cpp*

2.5 Bài toán 5 (Detonate the Maximum Bomb)

Problem. *Một trò chơi điện tử có tên "Detonate the Maximum Bomb". Trò chơi cung cấp cho người chơi một mặt phẳng tọa độ Oxy, với một số lượng quả bom có sẵn được đặt tại những vị trí khác nhau trên màn hình. Mỗi quả bom ngoài vị trí được đặt còn có bán kính nổ. Mỗi khi một quả bom nổ, nó cũng sẽ kích nổ cho tất cả các quả bom khác trong tầm. Để chiến thắng trò*

chơi, người chơi cần chọn duy nhất một quả bom để kích nổ sao cho số quả bom bị kích nổ theo là nhiều nhất. Yêu cầu dành cho sinh viên:
a) Xây dựng câu trúc dữ liệu phù hợp dựa trên những cấu trúc dữ liệu đã được hiện thực sẵn.
b) Viết hàm ***int detonatedBomb(int x, int y)*** *trả về số lượng quả bom sẽ bị nổ khi kích nổ quả bom tại tọa độ (x,y), nếu tại đó không có quả bom được đặt, trả về -1*
c) Viết hàm ***int maximumBomb()*** *trả về số lượng quả bom nhiều nhất có thể được kích nổ.*

Solution. *File code giải bài toán trên được đính kèm có tên Ex5.cpp*

2.6 Bài toán 6 (Network Delay Time)

Problem. *Một tập đoàn gồm nhiều đơn vị gồm công ty con và nhà máy cần xây dựng một hệ thống truyền thông tin giữa các đơn vị với nhau, tuy nhiên do khoảng cách địa lý nên tín hiệu truyền đi từ một đơn vị có những độ trễ khác nhau khi truyền đến với những đơn vị khác. Bên cạnh đó, một tập đoàn sẽ có thể có hoặc không có một văn phòng trung tâm - là nơi có đường truyền tin hiệu trực tiếp với tất cả các đơn vị còn lại. Yêu cầu dành cho sinh viên:*
a) Xây dựng câu trúc dữ liệu phù hợp dựa trên những cấu trúc dữ liệu đã được hiện thực sẵn.
b) Viết hàm ***bool hasCentralOffice()****, hàm này nhằm xác định xem tập đoàn có văn phòng trung tâm hay không.*
c) Viết hàm int ***int minimumDelayTime(string nameUnit)*** *với* ***nameUnit*** *là chuỗi biểu diễn tên đơn vị và hàm này sẽ trả về thời gian ngắn nhất để tín hiệu từ đơn vị đó có thể truyền tới tất cả các đơn vị còn lại*

Solution. *File code giải bài toán trên được đính kèm có tên Ex6.cpp*

2.7 Bài toán 7 (Keys Room)

Problem. *Vào mỗi cuối ngày, bác bảo vệ của một trung tâm giải trí có nhiệm vụ tắt hết điện hành lang của một mê cung giải đố gồm các căn phòng được nối với nhau bởi các hành lang. Tuy nhiên, vì sợ ma nên bác bảo về chỉ có thể đi qua các hành lang đang mở điện và sau đó tắt điện khi đi qua hành lang đó. Yêu cầu dành cho sinh viên:*
a) Xây dựng câu trúc dữ liệu phù hợp dựa trên những cấu trúc dữ liệu đã được hiện thực sẵn.
b) Viết hàm ***bool canFinish()*** *nếu có cách để bác bảo vệ có thể tắt hết đèn ở các hành lang và trở về căn phòng bắt đầu thì trả về* ***True****, ngược lại trả về* ***False***

Solution. *File code giải bài toán trên được đính kèm có tên Ex7.cpp*

2.8 Bài toán 8 (Number of provinces)

Problem. *Một đất nước gồm có nhiều huyện, các huyện có thể có hoặc không có kết nối với nhau. Hai huyện được kết nối với nhau gọi là kết nối trực tiếp, và kết nối thông qua một hoặc nhiều huyện khác là kết nối gián tiếp. Một tỉnh được định nghĩa là một tập hợp các huyện có kết nối trực tiếp hoặc gián tiếp với nhau. Bên cạnh đó, mỗi huyện sẽ có một danh sách các xã trực thuộc. Yêu cầu dành cho sinh viên:*
a) Xây dựng câu trúc dữ liệu phù hợp dựa trên những cấu trúc dữ liệu đã được hiện thực sẵn.
b) Viết hàm ***int numberOfProvinces()*** *trả về số tỉnh mà đất nước đó có*
c) Viết hàm ***int numberOfWard(string name)*** *nhận chuỗi là tên của một huyện và trả về tổng số xã mà tỉnh bao gồm huyện đó có*

Solution. *File code giải bài toán trên được đính kèm có tên Ex8.cpp*

3 Phân tích cấu trúc dữ liệu

Phần nội dung này được thực hiện bằng Google Colaboratory

References

[1] Kenneth H. Rosen *Discrete Mathematics and Its Applications*. McGraw Hill, 2011.

[2] *Graph Data Structure And Algorithms*, https://www.geeksforgeeks.org/graph-data-structure-and-algorithms, 2023