CMC UNIVERSITY

CHƯƠNG 5. ĐỒ THỊ

CMC UNIVERSITY

Nội dung

- ✓ Các khái niệm cơ bản
- ✓ Biểu diễn đồ thị
- ✓ Duyệt đồ thị
- ✓ Đường đi ngắn nhất trên đồ thị
- ✓ Cây khung cực tiểu

CMC UNIVERSITY

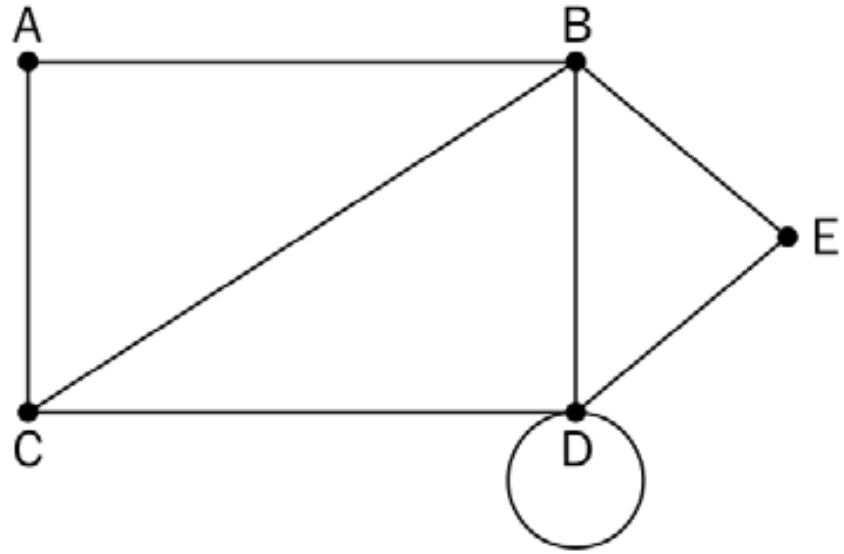
CÁC KHÁI NIỆM CƠ BẢN

Đồ thị

- **Đồ thị** gồm một tập hợp hữu hạn các đỉnh (còn gọi là nút) và các cạnh, trong đó các cạnh là liên kết giữa các đỉnh và mỗi cạnh trong đồ thị nối hai đỉnh phân biệt
- Đồ thị là một biểu diễn toán học hình thức của một mạng, tức là đồ thị G = (V, E) là một cặp có thứ tự của tập đỉnh V và tập cạnh E.

Ví dụ 1: Đồ thị G = (V, E)

- V = {A, B, C, D, E}
- E = {{A, B}, {A, C}, {B, C}, {B, D}, {C, D}, {D, D}, {B, E}, {D, E}}



Hình 1: Ví dụ đồ thị

Đồ thị

- **Nút (đỉnh)**: Một điểm hay nút trong đồ thị được gọi là đỉnh. Trong hình 1, các đỉnh hoặc nút là A, B, C, D, và E được biểu diễn bởi dấu chấm.
- **Cạnh**: là liên kết giữa hai đỉnh. Ví dụ, liên kết nối A và B là một cạnh.
- **Khuyên**: Khi một cạnh liên kết một nút với chính nó được gọi là khuyên, ví dụ nút D có khuyên.
- **Bậc của đỉnh/nút**: Tổng số các cạnh liên kết với một đỉnh được gọi là bậc của đỉnh đó. Ví dụ, bậc của đỉnh B trong đồ thị hình 1 là 4, kí hiệu deg(B).
- **Liền kề**: Nếu giữa 2 đỉnh có một cạnh nối thì hai đỉnh được gọi là liền kề. Ví dụ, C liền kề với đỉnh A vì giữa chúng có cạnh nối.

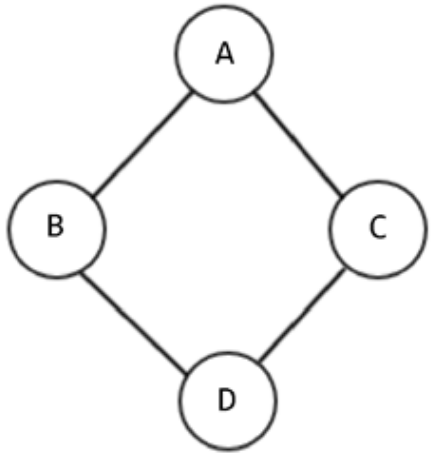
CMC UNIVERSITY

Đồ thị

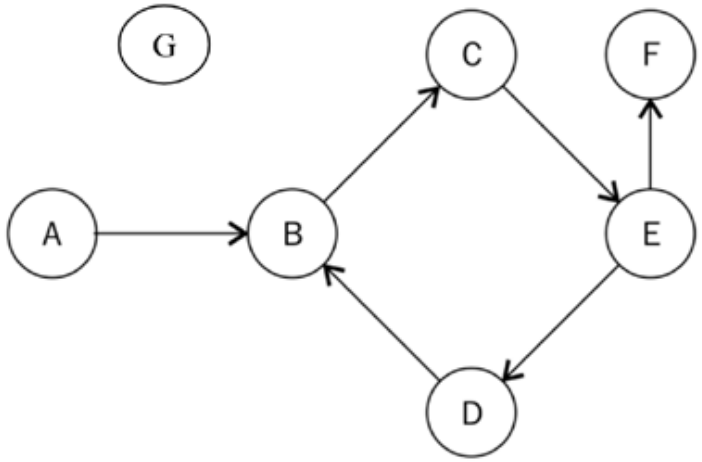
- **Đường đi**: Một chuỗi các đỉnh và cạnh giữa hai đỉnh bất kỳ biểu thị một đường đi. Ví dụ, CABE là đường đi từ đỉnh C đến đỉnh E.
- **Đường đi đơn**: đường đi mà mọi đỉnh, trừ đỉnh đầu tiên và đỉnh cuối cùng, đều khác nhau.
- **Độ dài đường đi**: số cạnh trên đường đi.
- **Đỉnh lá**: Một đỉnh là lá nếu bậc của đỉnh đúng bằng 1.

Đồ thị vô hướng và có hướng

- Nếu các cạnh nối trong đồ thị là vô hướng thì đồ thị đó được gọi là đồ thị vô hướng (**undirected graph**)và nếu các cạnh nối trong đồ thị là có hướng thì nó được gọi là đồ thị có hướng (**directed graph**).
- Đồ thị vô hướng chỉ đơn giản biểu diễn các cạnh dưới dạng các đường nối giữa các nút, không có thông tin bổ sung nào về mối quan hệ giữa các nút. Ví dụ, hình 2 minh họa một đồ thị vô hướng gồm bốn nút A, B, C và D, được kết nối bằng các cạnh:



Hình 2 Ví dụ đồ thị vô hướng



Hình 3. Ví dụ đồ thị có hướng

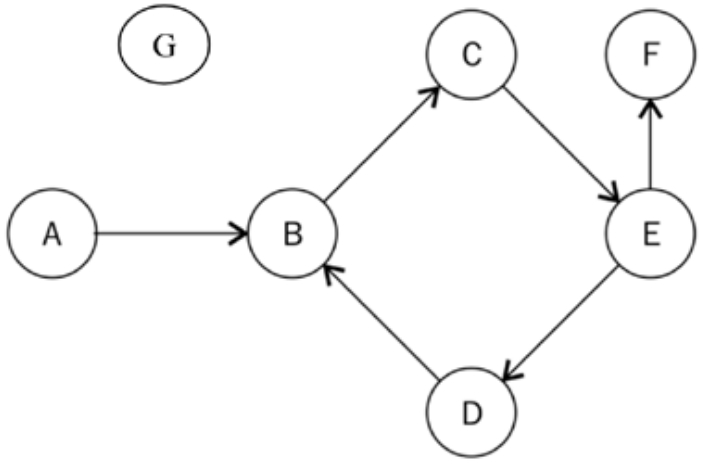
Đồ thị vô hướng và có hướng

Mũi tên trên cạnh xác định hướng của cạnh đó. Đồ thị hình 3 chỉ ra rằng có hướng từ A đến B, nhưng không có hướng từ B đến A. Trong đồ thị có hướng, mỗi đỉnh có một ***bậc trong*** (*indegree)* và một ***bậc ngoài*** (*outdegree).*

- **Bậc trong**: Tổng số cạnh đi vào đỉnh đó. Ví dụ, trong hình 3, đỉnh E có bậc trong là 1, vì cạnh CE đi vào đỉnh E.
- **Bậc ngoài**: Tổng số cạnh đi ra từ đỉnh đó. Ví dụ, trong hình 3, đỉnh E có bậc ngoài là 2, vì có 2 cạnh EF và ED đi ra khỏi nút đó.

Đồ thị vô hướng và có hướng

- **Đỉnh cô lập**: Một đỉnh được gọi là cô lập khi nó có bậc bằng 0, ví dụ đỉnh G ở hình 3
- **Đỉnh nguồn**: Một đỉnh được gọi là đỉnh nguồn nếu nó có bậc trong bằng 0. Ví dụ, đỉnh A ở hình 3.
- **Đỉnh chìm (sink vertex)**: Một đỉnh là đỉnh chìm nếu nó có bậc ngoài bằng 0. Ví dụ, đỉnh F ở hình 3.

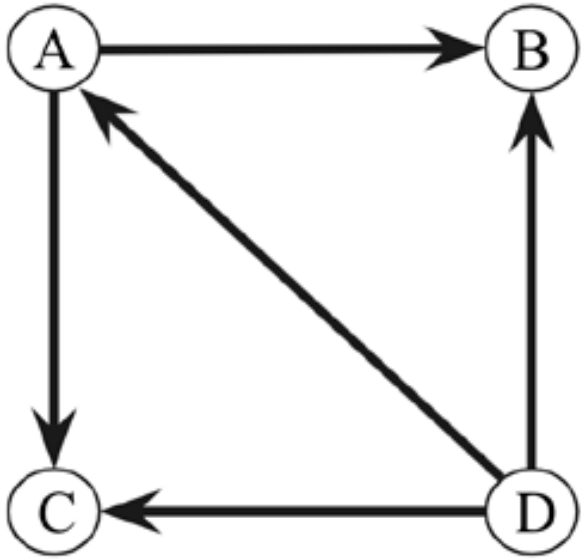


Hình 3. Ví dụ đồ thị có hướng

Đồ thị vòng có hướng

- **Đồ thị vòng có hướng** (directed acyclic graph - DAG) là một đồ thị có hướng không có chu trình; trong DAG tất cả các cạnh có hướng từ nút này đến nút khác sao cho chuỗi các cạnh không tạo thành một vòng khép kín.
- **Chu trình** (cycle) trong một đồ thị được hình thành khi nút bắt đầu của cạnh đầu tiên bằng nút kết thúc của cạnh cuối cùng trong chuỗi các cạnh

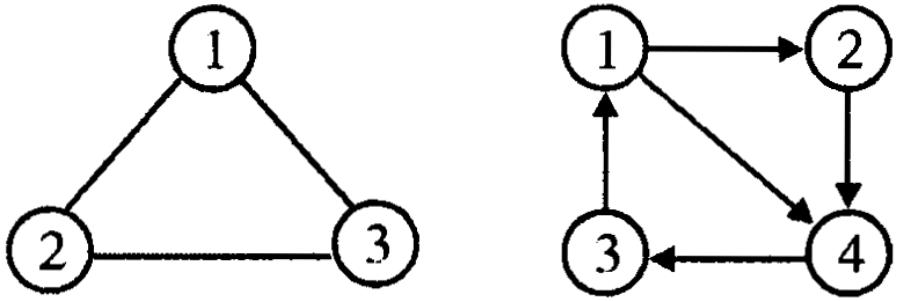
Trong DAG, nếu ta bắt đầu trên đường đi bất kỳ từ một nút đã cho, ta sẽ không tìm thấy đường đi kết thúc trên cùng một nút. DAG có nhiều ứng dụng như lập kế hoạch công việc, đồ thị trích dẫn, và nén dữ liệu.



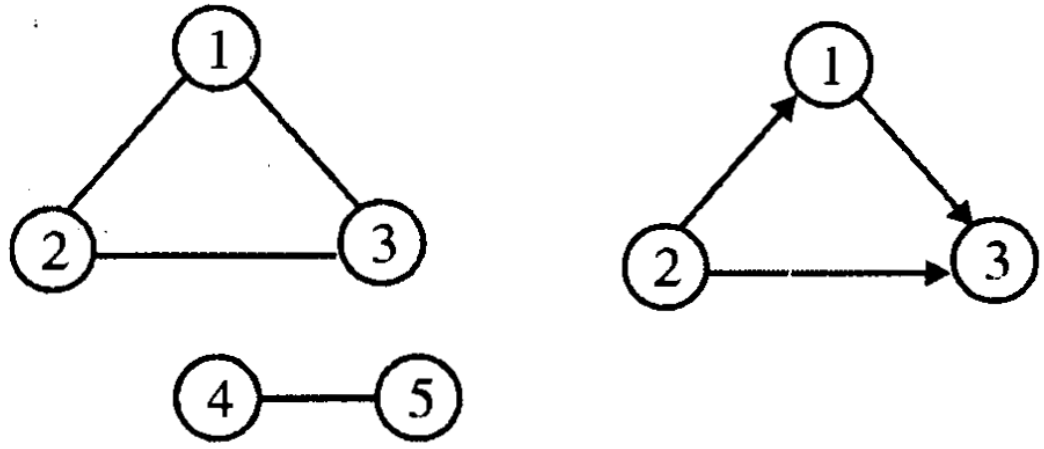
Hình 4. Ví dụ đồ thị vòng có hướng

Đồ thị liên thông

- **Liên thông** (connected): Hai đỉnh v_i và v_j được gọi là liên thông nếu có đường đi từ v_i đến v_j (với đồ thị vô hướng, cũng có đường đi từ v_j đến v_i).
- G được gọi là **đồ thị liên thông** nếu mọi cặp đỉnh phân biệt v_i và v_j trong V(G) có đường đi từ v_i đến v_j .



Hình a) Đồ thị liên thông

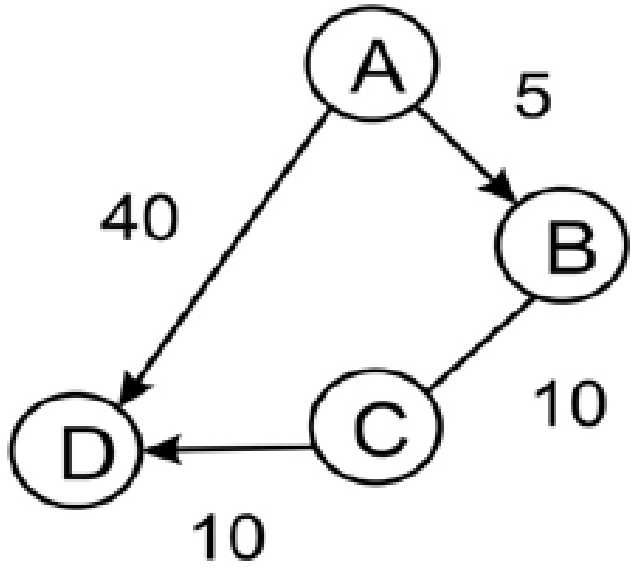


Hình b) Đồ thị không liên thông

Đồ thị có trọng số

- **Đồ thị có trọng số** (weighted graph) là đồ thị có trọng số liên kết với các cạnh trong đồ thị. Đồ thị có trọng số có thể là đồ thị vô hướng hoặc có hướng. Trọng số có thể là khoảng cách hoặc chi phí giữa 2 nút, phụ thuộc vào mục đích của đồ thị:

Đồ thị có trọng số trong Hình 5 chỉ ra các cách khác nhau để đi từ nút A đến nút D.
Có hai đường đi: A-D (tổng trọng số 40) và A-B-C-D (tổng trọng số 25)

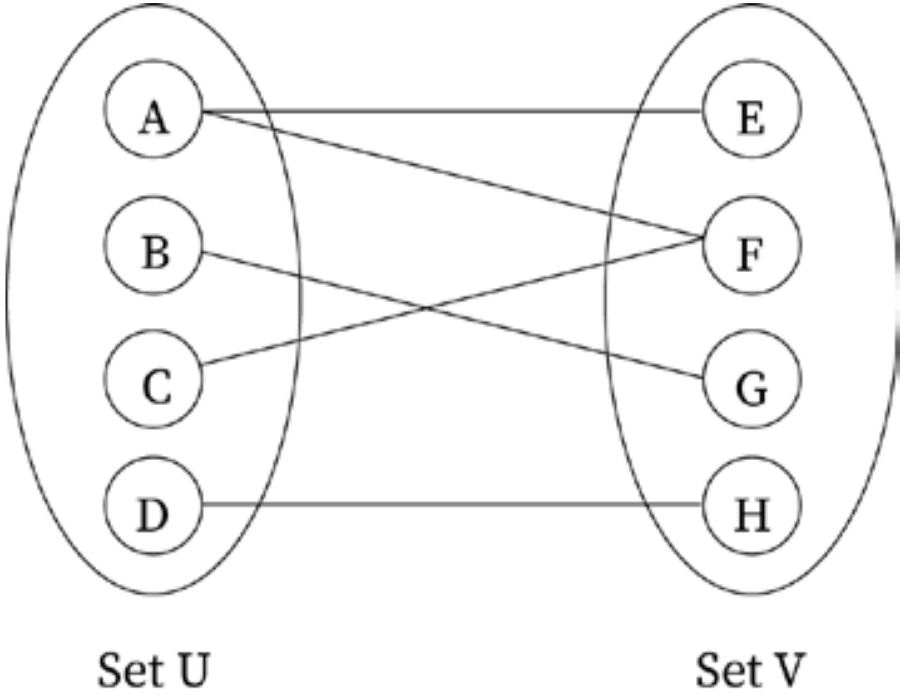


Hình 5. Ví dụ đồ thị có trọng số

Đồ thị hai phía

- **Đồ thị hai phía** (bipartite graph - bigraph) là một đồ thị đặc biệt, trong đó tất cả các đỉnh của đồ thị được chia thành hai tập hợp sao cho các cạnh nối các đỉnh của tập này đến các đỉnh của tập kia.

Đồ thị hai phía trong Hình 6: các nút được chia thành 2 tập U và V, sao cho mỗi cạnh có 1 nút kết thúc trong U và nút kết thúc khác trong V (ví dụ, cạnh (A, E)).
Trong đồ thị hai phía, không có cạnh nối 2 nút trong cùng một tập nút.



Hình 6. Ví dụ đồ thị hai phía

Đồ thị hai phía hữu ích khi cần mô hình hóa mối quan hệ giữa hai lớp đối tượng khác nhau, ví dụ: đồ thị ứng viên và công việc, đồ thị cầu thủ và câu lạc bộ, ...

CMC UNIVERSITY

BIỂU DIỄN ĐỒ THỊ

CMC UNIVERSITY

Biểu diễn đồ thị

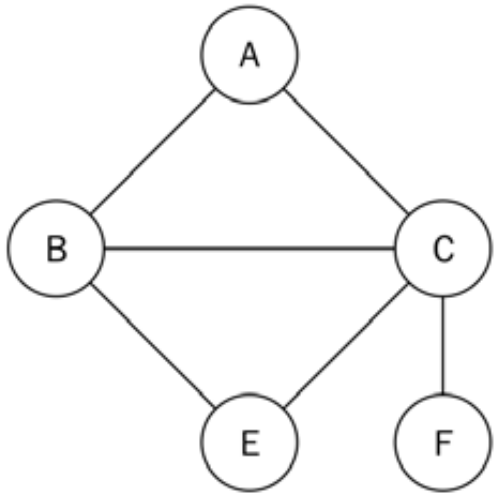
Kỹ thuật biểu diễn đồ thị là cách lưu trữ đồ thị trong bộ nhớ, nghĩa là cách lưu trữ các đỉnh, cạnh và trọng số (nếu đồ thị là đồ thị có trọng số).

Đồ thị có thể được biểu diễn bởi: danh sách kề (adjacency list), và ma trận kề (adjacency matrix)

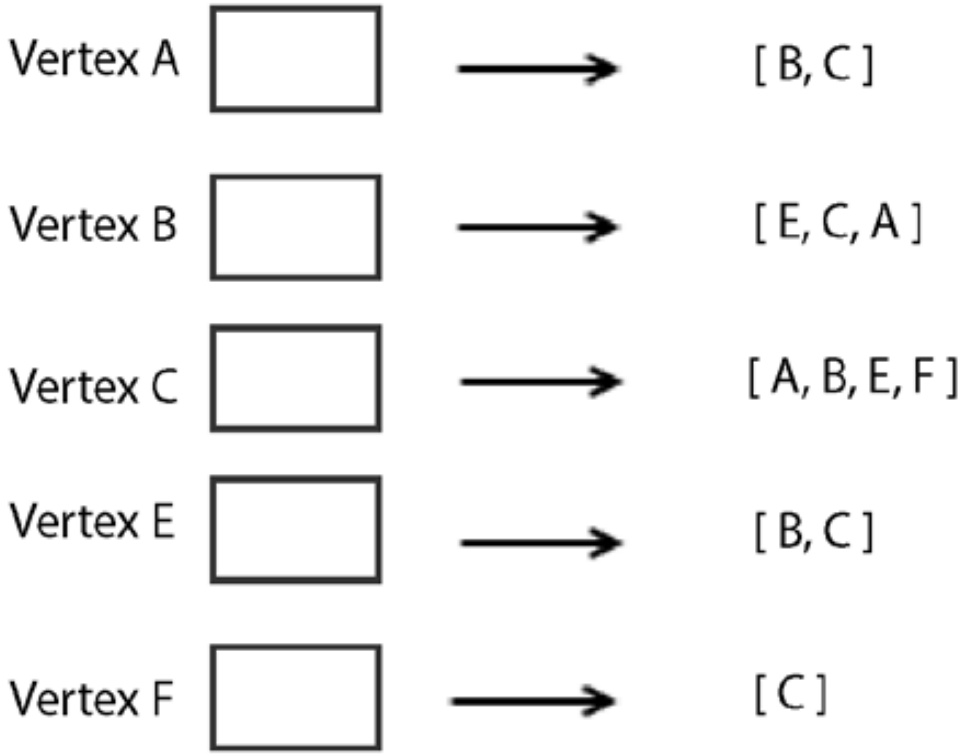
- Biểu diễn danh sách kề dựa trên danh sách liên kết. Đồ thị được biểu diễn bằng danh sách các láng giềng (còn được gọi là nút liền kề) cho mọi đỉnh (hoặc nút) của đồ thị.
- Trong biểu diễn đồ thị bằng ma trận kề, sử dụng ma trận biểu diễn mỗi nút liền kề với nút khác trong đồ thị; tức là, ma trận kề có thông tin về mọi cạnh trong đồ thị, biểu diễn bởi các ô của ma trận.

Danh sách kề

- Tất cả các nút nối trực tiếp với nút x được liệt kê trong danh sách các nút liền kề của nó. Đồ thị được biểu diễn bằng danh sách liền kề cho tất cả các nút của đồ thị.
- Hai nút A và B trong đồ thị hình 7 là liền kề nếu có cạnh nối trực tiếp giữa chúng:



Hình 7. Ví dụ đồ thị 5 nút



Hình 8. Danh sách kề cho đồ thị hình 7

Danh sách kề

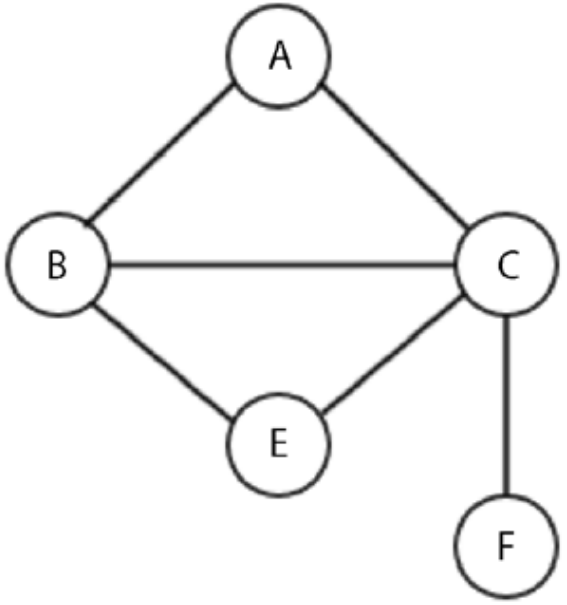
- Danh sách liên kết có thể được dung để thực hiện danh sách kề. Để biểu diễn đồ thị, cần số danh sách liên kết bằng với số nút của đồ thị. Tại mỗi chỉ số nút, là danh sách các nút kề với nó được lưu trữ. Ví dụ, danh sách kề trong hình 8 biểu diễn đồ thị hình 7

- Danh sách kề là một kỹ thuật biểu diễn đồ thị thích hợp hơn khi đồ thị thưa và các nút trong đồ thị cần thêm hoặc xóa thường xuyên. Rất khó để kiểm tra xem một cạnh đã cho có xuất hiện trong đồ thị hay không khi sử dụng kỹ thuật này.

```
public class Main {
    public static void main(String args[]) {
        // Khởi tạo đồ thị với 5 đỉnh
        char[] vertices = {'A', 'B', 'C', 'E', 'F'};
        Graph graph = new Graph(5, vertices);
        // Thêm các cạnh vào đồ thị
        graph.addEdge('A', 'B');
        graph.addEdge('A', 'C');
        graph.addEdge('B', 'E');
        graph.addEdge('B', 'C');
        graph.addEdge('B', 'A');
        graph.addEdge('C', 'A');
        graph.addEdge('C', 'B');
        graph.addEdge('C', 'E');
        graph.addEdge('C', 'F');
        graph.addEdge('E', 'B');
        graph.addEdge('E', 'C');
        graph.addEdge('F', 'C');
        // Hiển thị danh sách kề của đồ thị
        graph.display();
    }
}
```

Ma trận kề

- Đồ thị được biểu diễn bằng cách hiển thị các nút và các liên kết của chúng qua các cạnh. Ma trận (V x V) được sử dụng để biểu diễn đồ thị, trong đó mỗi ô biểu thị một cạnh trong đồ thị.
- Ma trận kề là một mảng 2 chiều, các ô của ma trân có giá trị 1 hoặc 0, phụ thuộc vào liệu 2 nút có được nối bởi 1 cạnh hay không. Ví dụ về đồ thị và ma trận kề tương ứng của nó trong hình 9.



Adjacency Matrix

	A	B	C	E	F
A	0	1	1	0	0
B	1	0	1	1	0
C	1	1	0	1	1
E	0	1	1	0	0
F	0	0	1	0	0

Hình 9. Ma trận kề của đồ thị đã cho

CMC UNIVERSITY

Ma trận kề

```java
import java.util.*;

// Định nghĩa lớp đại diện cho đồ thị
class Graph {
    int V; // Số đỉnh của đồ thị
    char[] vertices; // Danh sách nhãn đỉnh
    int[][] adjacencyMatrix; // Ma trận kề

    // Khởi tạo đồ thị với số đỉnh V và danh sách nhãn đỉnh
    public Graph(int V, char[] vertices) {
        this.V = V;
        this.vertices = vertices;
        adjacencyMatrix = new int[V][V];
    }
    // Thêm một cạnh vào đồ thị
    void addEdge(char src, char dest) {
        int srcIndex = getIndex(src);
        int destIndex = getIndex(dest);
        adjacencyMatrix[srcIndex][destIndex] = 1;
    }
```

CMC UNIVERSITY

Ma trận kề

```
33      // Hiển thị ma trận kề của đồ thị
34      void display() {
35          System.out.print("  ");
36          for (char vertex : vertices) {
37              System.out.print(vertex + " ");
38          }
39          System.out.println();
40          for (int i = 0; i < V; ++i) {
41              System.out.print(vertices[i] + " ");
42              for (int j = 0; j < V; ++j) {
43                  System.out.print(adjacencyMatrix[i][j] + " ");
44              }
45              System.out.println();
46          }
47      }
```

CMC UNIVERSITY

Ma trận kề

```java
public class Main {
    public static void main(String args[]) {
        // Khởi tạo đồ thị với 5 đỉnh và danh sách nhãn đỉnh
        char[] vertices = {'A', 'B', 'C', 'E', 'F'};
        Graph graph = new Graph(5, vertices);

        // Thêm các cạnh vào đồ thị
        graph.addEdge('A', 'B');
        graph.addEdge('A', 'C');
        graph.addEdge('B', 'E');
        graph.addEdge('B', 'C');
        graph.addEdge('B', 'A');
        graph.addEdge('C', 'A');
        graph.addEdge('C', 'B');
        graph.addEdge('C', 'E');
        graph.addEdge('C', 'F');
        graph.addEdge('E', 'B');
        graph.addEdge('E', 'C');
        graph.addEdge('F', 'C');

        // Hiển thị ma trận kề của đồ thị
        graph.display();
    }
}
```

	A	B	C	E	F
A	0	1	1	0	0
B	1	0	1	1	0
C	1	1	0	1	1
E	0	1	1	0	0
F	0	0	1	0	0

Ma trận kề

- Mảng adjacencyMatrix có các dòng và cột, bắt đầu từ A đến tất cả các đỉnh khác có chỉ số từ 0 đến 5. Vòng lặp for lặp qua danh sách các bộ dữ liệu và sử dụng phương thức chỉ mục để lấy các chỉ mục lưu trữ các cạnh tương ứng.
- Output của đoạn code trước là ma trận kề cho cùng đồ thị trong hình 9. Ma trận kề được tạo có dạng như sau:

  (A, A): 0 biểu diễn sự vắng mặt của cạnh giữa A và A;

  (C, B): 1 biểu thị cạnh giữa đỉnh C đình B trong đồ thị.

- Việc sử dụng ma trận kề biểu diễn đồ thị là phù hợp khi phải thường xuyên tra cứu, kiểm tra sự có mặt hay vắng mặt của một cạnh giữa hai nút của đồ thị, chẳng hạn: tạo bảng định tuyến trong mạng, tìm kiếm tuyến đường trong các ứng dụng giao thông công cộng và hệ thống định vị, v.v.
- Ma trận kề không phù hợp khi các nút thường xuyên được thêm hoặc xóa trong đồ thị, trong những tình huống đó, danh sách kề là một kỹ thuật tốt hơn

CMC UNIVERSITY

DUYỆT ĐỒ THỊ

CMC UNIVERSITY

Duyệt đồ thị

- Duyệt đồ thị có nghĩa là thăm tất cả các đỉnh của đồ thị trong khi lưu vết những đỉnh nào đã được thăm và những đỉnh nào chưa được thăm.
- Một thuật toán duyệt đồ thị hiệu quả nếu nó duyệt tất cả các đỉnh của đồ thị với thời gian ít nhất có thể.
- Duyệt đồ thị, cũng như thuật toán tìm kiến trên đồ thị, khá tương tự với thuật toán duyệt cây như các thuật toán duyệt theo thứ tự trước, thứ tự giữa, thứ tự sau và thứ tự mức. Trong thuật toán tìm kiếm trên đồ thị, ta bắt đầu với 1 nút và duyệt qua các cạnh đến tất cả các nút khác của đồ thị.
- Một chiến lược duyệt đồ thị phổ biến là theo một đường đi cho đến khi gặp một ngõ cụt, sau đó duyệt ngược lại cho đến khi gặp một đỉnh mà có một đường đi thay thế. Có thể lặp lại việc di chuyển từ đỉnh này sang đỉnh khác để duyệt qua toàn bộ hoặc một phần của đồ thị.
- Duyệt đồ thị có nhiều ứng dụng thực tế như tìm đường đi ngắn nhất từ thành phố này sang thành phố khác trong mạng lưới các thành phố.

CMC UNIVERSITY

Duyệt theo chiều rộng

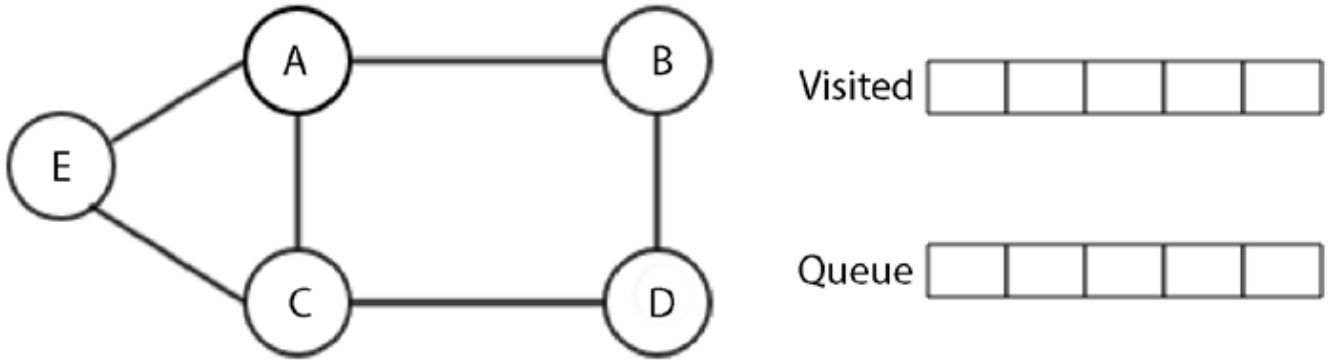
- **Tìm kiếm theo chiều rộng** (Breadth-first search - BFS) thực hiện tương tự như thuật toán duyệt theo thứ tự mức trong cấu trúc cây.
- Thuật toán BFS: bắt đầu bằng thăm nút gốc tại mức 0, sau đó đến tất cả các nút trực tiếp nối với nút gốc tại mức 1. Tiếp theo, các nút ở mức 2 được thăm tiếp theo. Tương tự, tất cả các nút trong đồ thị được duyệt theo từng mức cho đến khi tất cả các nút được thăm.
- Cấu trúc hàng đợi được sử dụng để lưu thông tin các đỉnh đã được thăm trong đồ thị:

- Đầu tiên, ta truy cập nút bắt đầu và sau đó truy cập tất cả các đỉnh liền kề của nó. Đầu tiên chúng ta lần lượt thăm các đỉnh liền kề này, đồng thời thêm các đỉnh liền kề của chúng vào danh sách các đỉnh sẽ được thăm..
- Làm theo qui trình này cho đến khi thăm tất cả các đỉnh của đồ thị, đảm bảo rằng không có đỉnh nào được thăm hai lần.

Duyệt theo chiều rộng

Ví dụ 1: Hình 10 biểu diễn đồ thị với 5 nút ở bên trái, và bên phải là một cấu trúc dữ liệu hàng đợi lưu trữ các đỉnh đã được thăm.

- Bắt đầu thăm nút đầu tiên A, sau đó them các đỉnh kề của nó B, C, và E, vào hàng đợi. Có nhiều cách bổ sung các nút này vào hàng đợi: BCE, CEB, CBE, BEC, hoặc ECB, mỗi cách này sẽ đưa ra các kết quả duyệt đồ thị khác nhau.

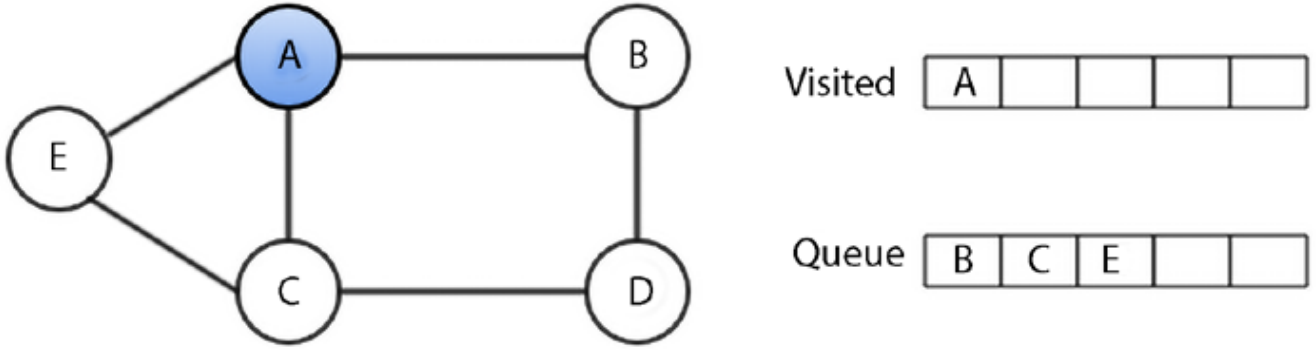


Hình 10. Đồ thị ví dụ

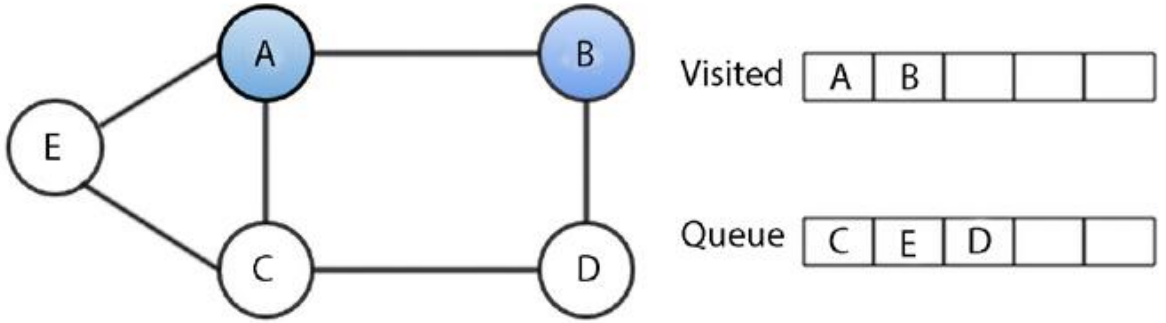
Duyệt theo chiều rộng

- Để đơn giản, ta thêm các nút theo thứ tự bảng chữ cái vào hàng đợi, tức là, BCE. Nút A được thăm như được chỉ ra ở Hình 11.

- Tiếp theo, thăm đỉnh kề đầu tiên, B, và bổ sung đỉnh kề của B mà chưa được bổ sung vào hàng đợi hoặc chưa được thăm. Trong trường hợp này, bổ sung đỉnh D (vì B có 2 đỉnh kề A và D, nhưng A đã được thăm) vào hàng đợi, như hình 12

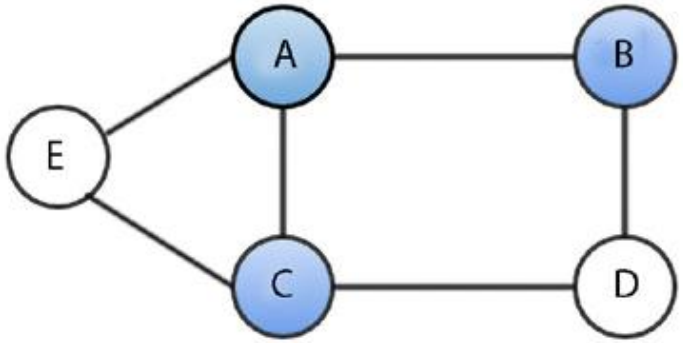


Hình 11. Nút A đã được thăm theo duyệt theo chiều rộng



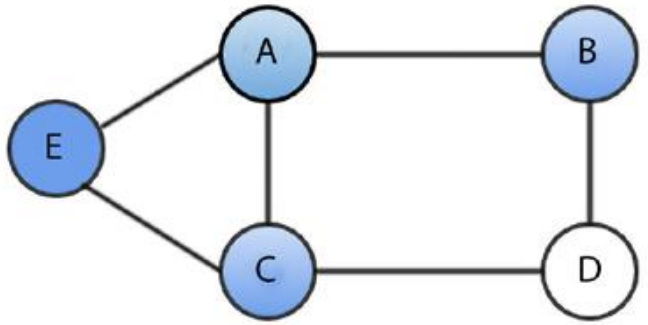
Hình 12. Nút B đã được thăm theo duyệt theo chiều rộng

Duyệt theo chiều rộng



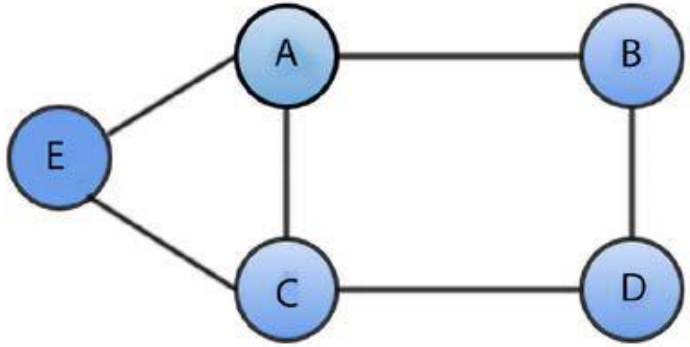
Visited | A | B | C | | |

Queue | E | D | | | |



Visited | A | B | C | E | |

Queue | D | | | | |



Visited | A | B | C | E | D |

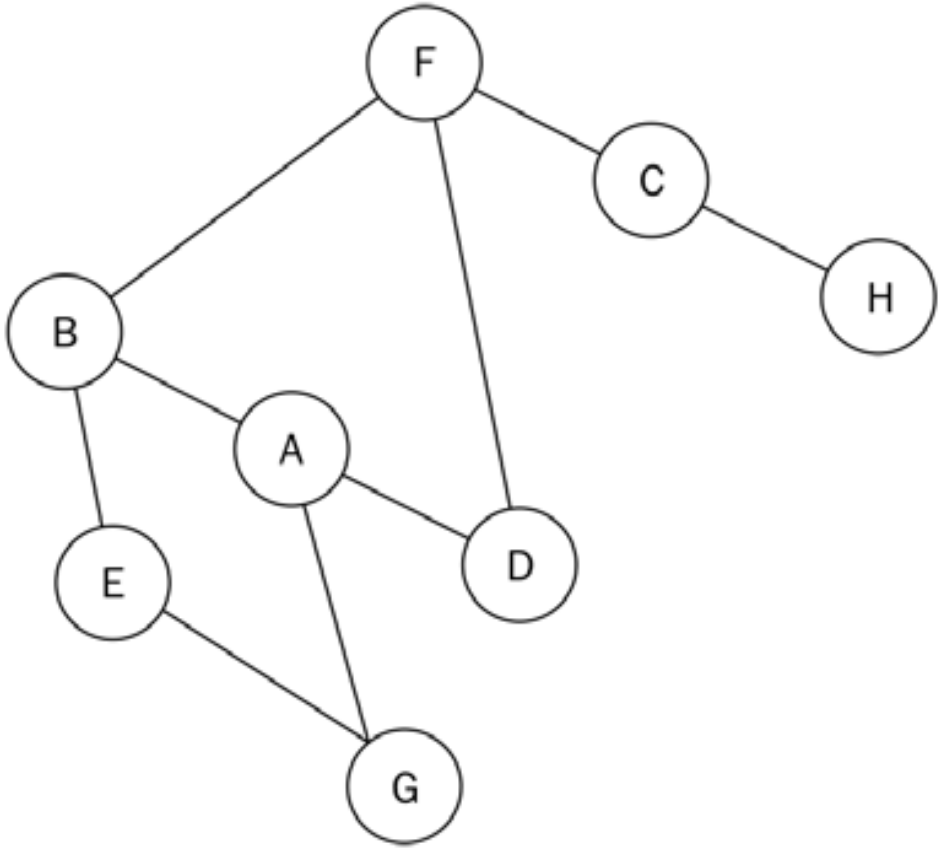
Queue | | | | | |

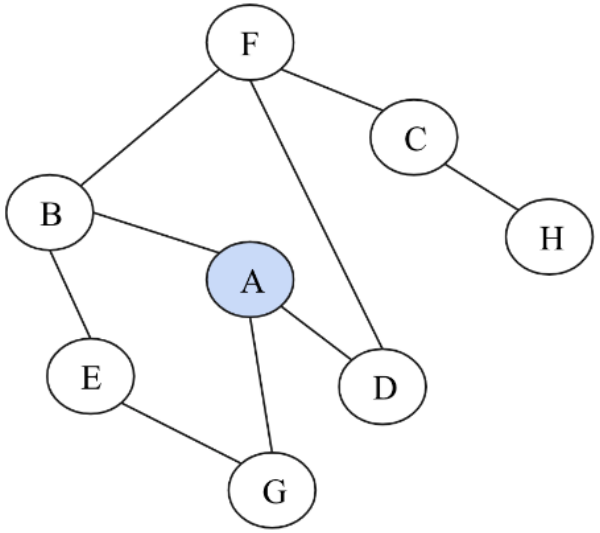
The BFS algorithm for traversing the preceding graph visits the vertices in the order of A-B-C-E-D.

Duyệt theo chiều rộng

Ví dụ 2: cho đồ thị như hình sau

Duyệt đồ thị theo chiều rộng bắt đầu từ A

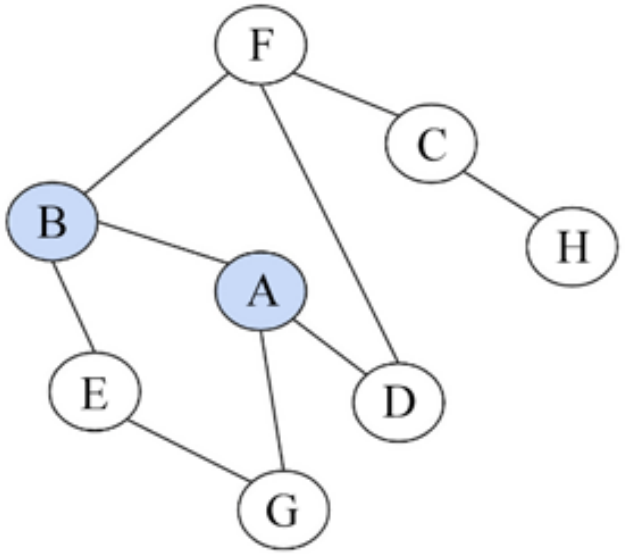




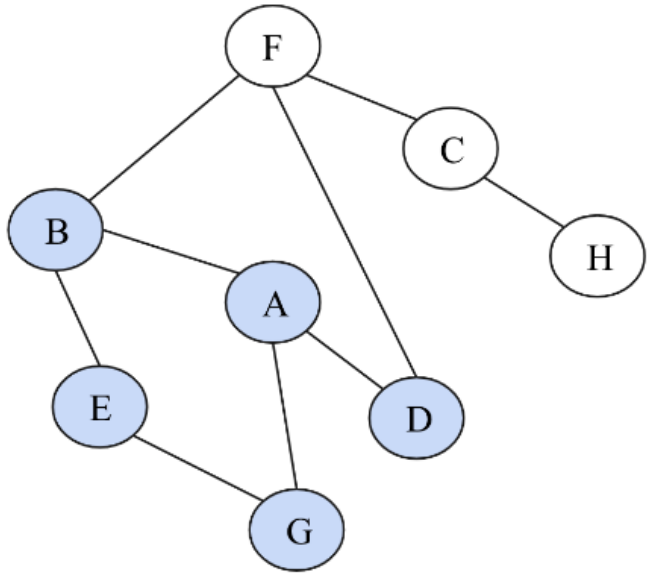
Visited	A						
Queue	B	D	G				

CMC UNIVERSITY

Duyệt theo chiều rộng



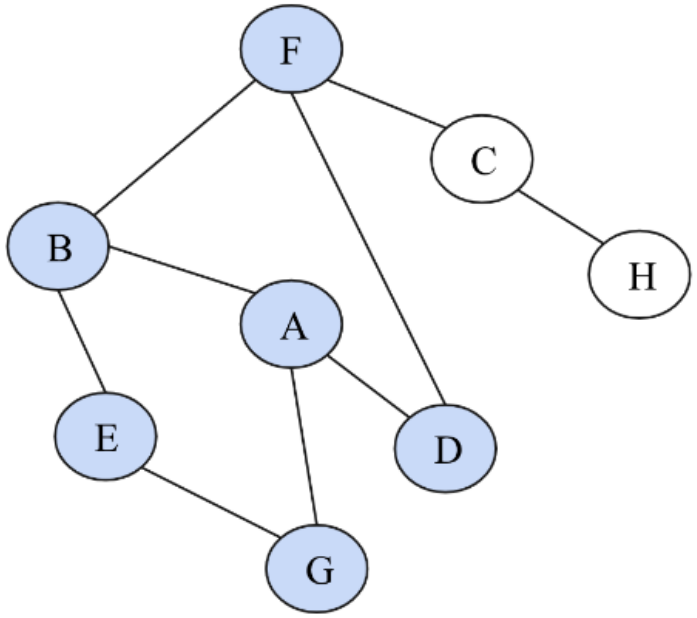
Visited	A	B					
Queue	D	G	E	F			



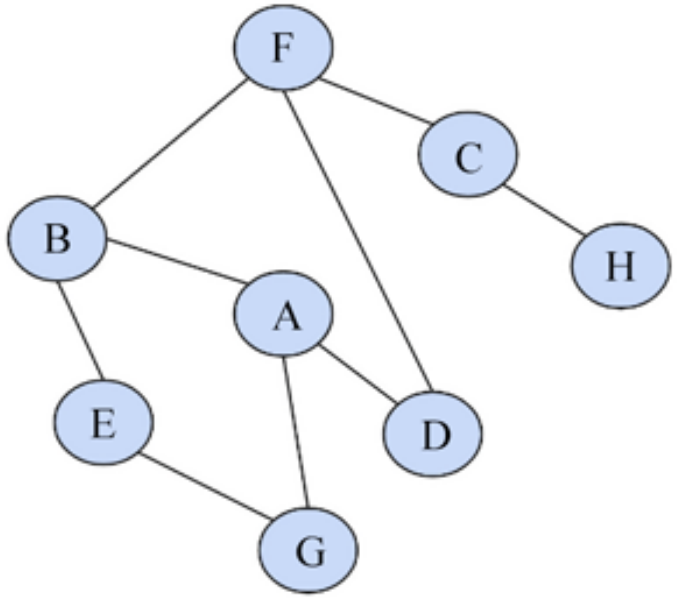
Visited	A	B	D	G	E			
Queue	F							

CMC UNIVERSITY

Duyệt theo chiều rộng



Visited	A	B	D	G	E	F	
Queue	C						



Visited	A	B	D	G	E	F	C	H
Queue								

A, B, D, G, E, F, C, H

CMC UNIVERSITY

Duyệt theo chiều rộng

Cài đặt thuật toán BFS bằng Java:

```java
import java.util.*;

public class BFS1 {
    private Map<Character, List<Character>> adjacencyList; // Danh sách kề của đồ thị

    public BFS1() {
        adjacencyList = new HashMap<>();    // Khởi tạo danh sách kề
    }

    // Thêm cạnh vào đồ thị
    public void addEdge(char src, char dest) {
        adjacencyList.computeIfAbsent(src, k -> new ArrayList<>()).add(dest);
    }

```

CMC UNIVERSITY

Duyệt theo chiều rộng

```java
// Thuật toán BFS bắt đầu từ đỉnh source
public void bfs1(char source) {
    Set<Character> visited = new HashSet<>(); // Đánh dấu các đỉnh đã được thăm
    Queue<Character> queue = new LinkedList<>(); // Hàng đợi cho BFS

    visited.add(source);     // Đánh dấu đỉnh nguồn đã thăm và đưa vào hàng đợi
    queue.add(source);

    while (!queue.isEmpty()) {
        // Lấy đỉnh ra khỏi hàng đợi và in ra
        char vertex = queue.poll();
        System.out.print(vertex + " ");

        // Lấy tất cả các đỉnh kề của đỉnh đã lấy ra
        List<Character> neighbors = adjacencyList.get(vertex);
        if (neighbors != null) {
            Collections.sort(neighbors); // Sắp xếp theo thứ tự bảng chữ cái
            for (char neighbor : neighbors) {
                if (!visited.contains(neighbor)) {
                    visited.add(neighbor); // Đánh dấu đỉnh kề là đã thăm
                    queue.add(neighbor); // Đưa đỉnh kề vào hàng đợi
                }
            }
        }
    }
}
```

CMC UNIVERSITY

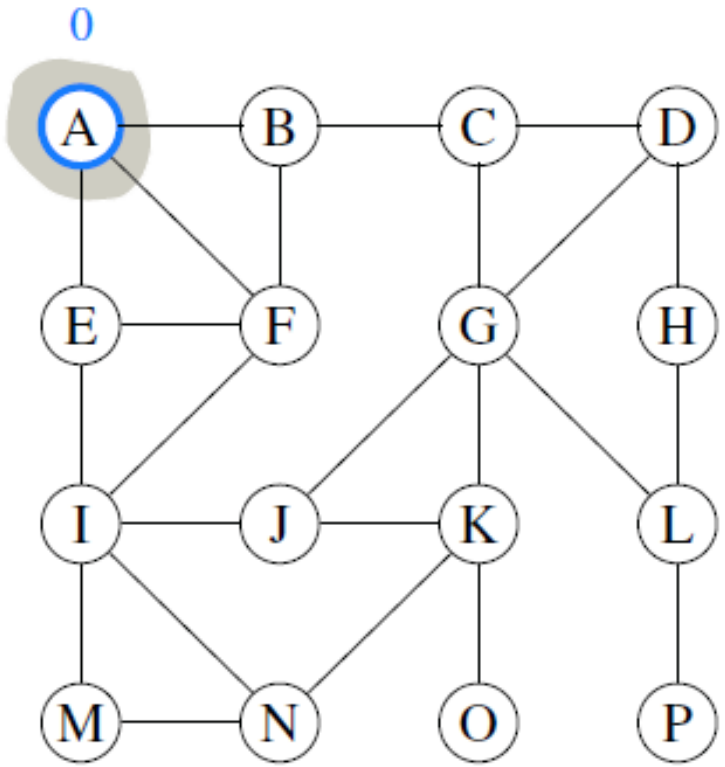
Duyệt theo chiều rộng

```
42      public static void main(String[] args) {
43          BFS1 graph = new BFS1();
    Run | Debug
44
45          graph.addEdge('A', 'B');
46          graph.addEdge('A', 'G');
47          graph.addEdge(src:'A', dest:'D');
48          graph.addEdge(src:'B', dest:'A');
49          graph.addEdge(src:'B', dest:'F');
50          graph.addEdge(src:'B', dest:'E');
51          graph.addEdge(src:'C', dest:'F');
52          graph.addEdge(src:'C', dest:'H');
53          graph.addEdge(src:'D', dest:'F');
54          graph.addEdge(src:'D', dest:'A');
55          graph.addEdge(src:'E', dest:'B');
56          graph.addEdge(src:'E', dest:'G');
57          graph.addEdge(src:'F', dest:'B');
58          graph.addEdge(src:'F', dest:'D');
59          graph.addEdge(src:'F', dest:'C');
60          graph.addEdge(src:'G', dest:'A');
61          graph.addEdge(src:'G', dest:'E');
62          graph.addEdge(src:'H', dest:'C');
63  src:dest:
64          System.out.prisrc:ntln(dest:"BFS starting from vertex A:");
65          graph.bfs1('A');
66      }
67  }
```

```
BFS starting from vertex A:
A B D G E F C H
```

Duyệt theo chiều rộng

- ***Ví dụ 3***: Duyệt đồ thị sau theo chiều rộng với đỉnh bắt đầu là A



CMC UNIVERSITY

Duyệt theo chiều rộng

G được biểu diễn bởi danh sách kề

- Trong trường hợp xấu nhất, mỗi nút và cạnh sẽ cần được duyệt qua, và do đó mỗi nút sẽ được enqueued và dequeued ít nhất một lần.
- Thời gian thực hiện cho mỗi hoạt động enqueue và dequeue là $O(1)$, do vậy tổng thời gian cho nhiệm vụ này là $O(|V|)$.
- Thời gian quét danh sách kề cho mọi đỉnh là $O(|E|)$.
- Tổng độ phức tạp thời gian của thuật toán BFS là $O(|V| + |E|)$, trong đó $|V|$ là số đỉnh, $|E|$ là số cạnh của đồ thị.

G được biểu diễn bởi ma trận kề: độ phức tạp thời gian $O(|V|^2)$

CMC UNIVERSITY

Duyệt theo chiều rộng

- Thuật toán BFS rất hữu ích để xây dựng đường đi ngắn nhất trong đồ thị với số lần lặp tối thiểu.
- BFS có thể được sử dụng để tạo trình thu thập dữ liệu web hiệu quả, nó có thể duy trì nhiều cấp độ chỉ mục cho các công cụ tìm kiếm và có thể duy trì danh sách các trang web đã đóng từ một trang web nguồn.
- BFS cũng hữu ích cho các hệ thống định vị trong đó có thể dễ dàng truy xuất các vị trí lân cận của các vị trí dựa vào đồ thị.

CMC UNIVERSITY

Duyệt theo chiều sâu

- Duyệt theo chiều sâu (depth-first search - DFS) trên đồ thị tương tự với thuật toán duyệt theo thứ tự trước trên cấu trúc dữ liệu cây.
- Trong thuật toán DFS, ta duyệt cây theo độ sâu của bất kỳ đường đi cụ thể nào trong đồ thị. Như vậy, các nút con được truy cập đầu tiên trước các nút anh chị em.

Thuật toán:

- Đầu tiên, thăm đỉnh xuất phát v,
- Sau đó, xem tất cả các nút kề với nút hiện tại v, thăm một trong các nút kề w:

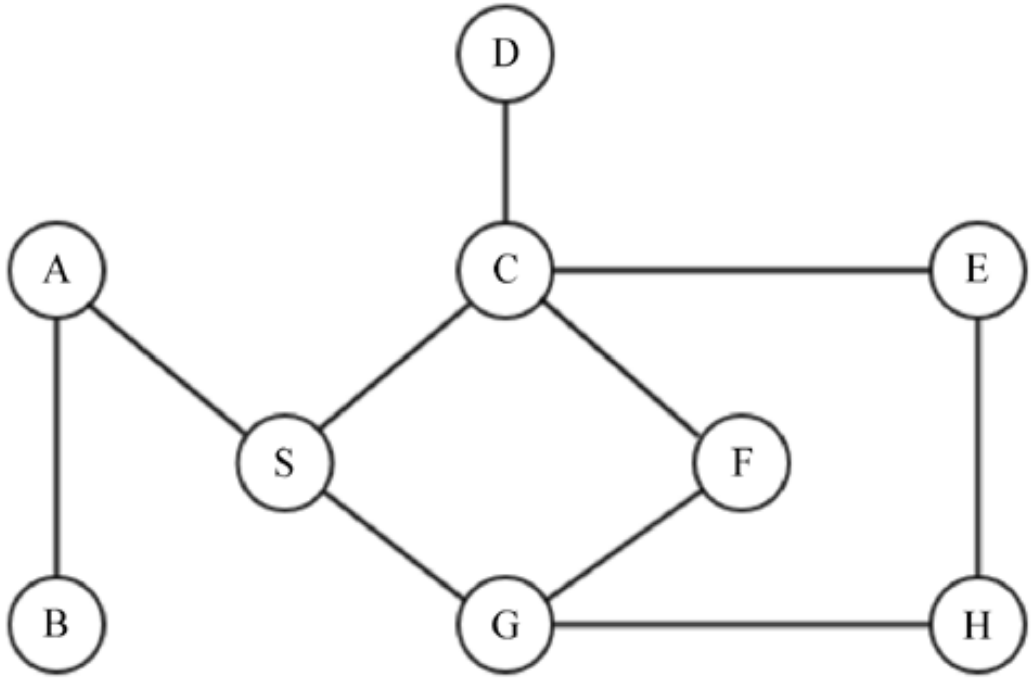
- Nếu w đã được thăm, quay lại nút hiện tại v.
- Nếu w chưa được thăm, đi đến nút w đó và tiếp tục tìm kiếm theo chiều sâu từ nút w.

- Tiếp tục quá trình trên, phép tìm kiếm kết thúc khi không còn một đỉnh nào chưa được thăm mà vẫn có thể với tới được từ nút đã được thăm.

Duyệt theo chiều sâu

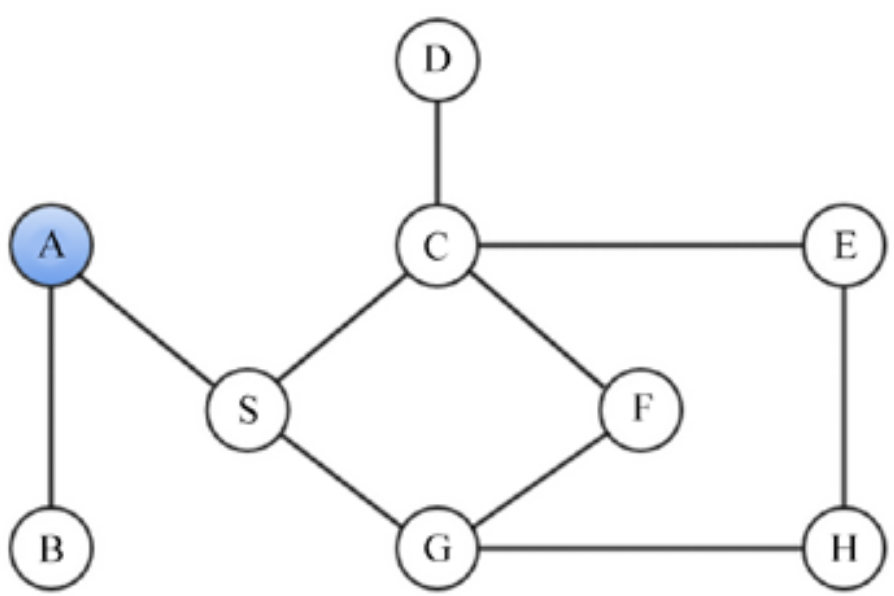
Ví dụ 4: cho đồ thị sau

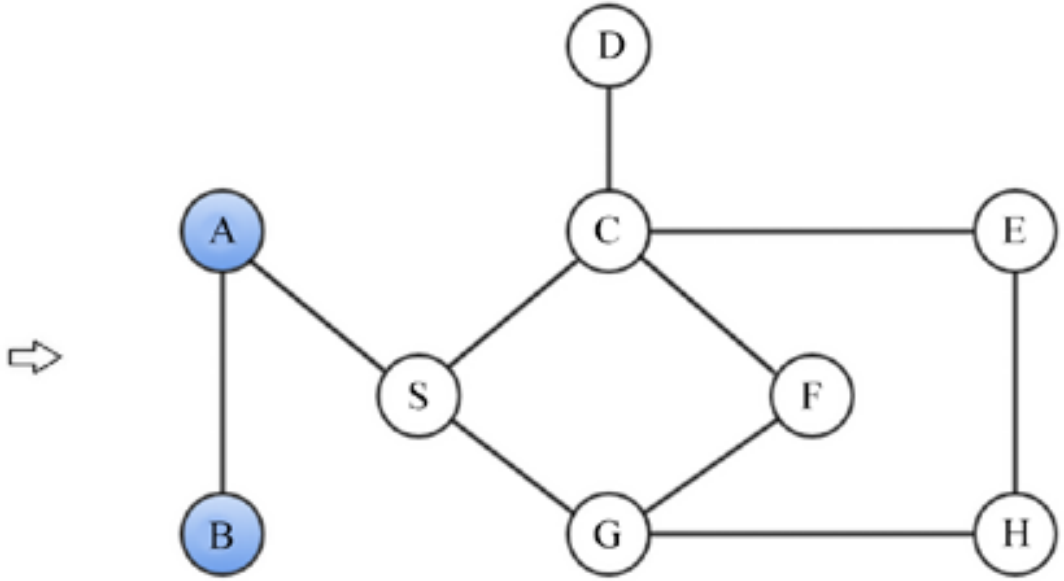
- Bắt đầu thăm nút A, và sau đó xét các láng giềng của A, tiếp theo là láng giềng của láng giềng đó, v.v.
- Thăm một trong các láng giềng của A là B (trong ví dụ này, sắp xếp các nhãn đỉnh theo thứ tự bảng chữ cái)

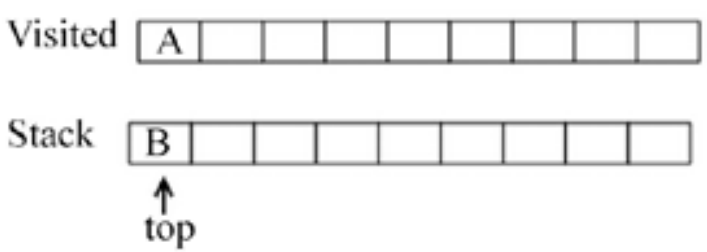


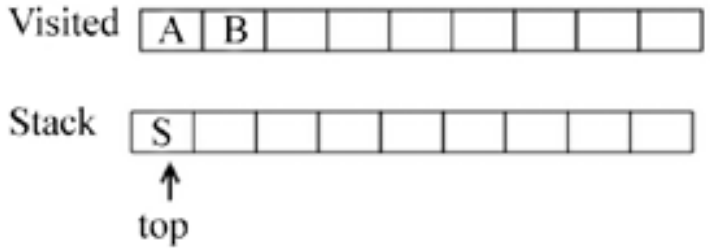
Duyệt theo chiều sâu

Ví dụ 4:



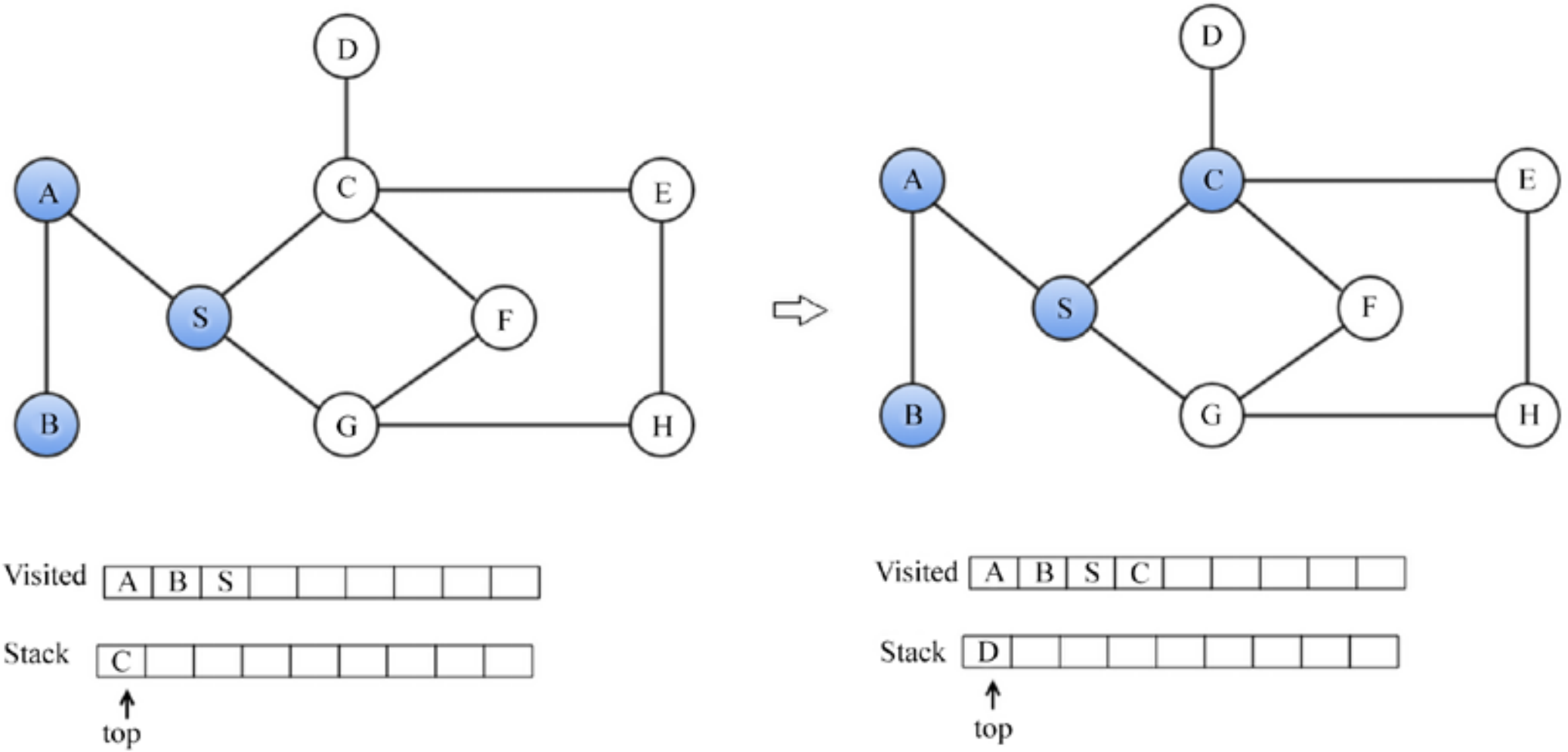






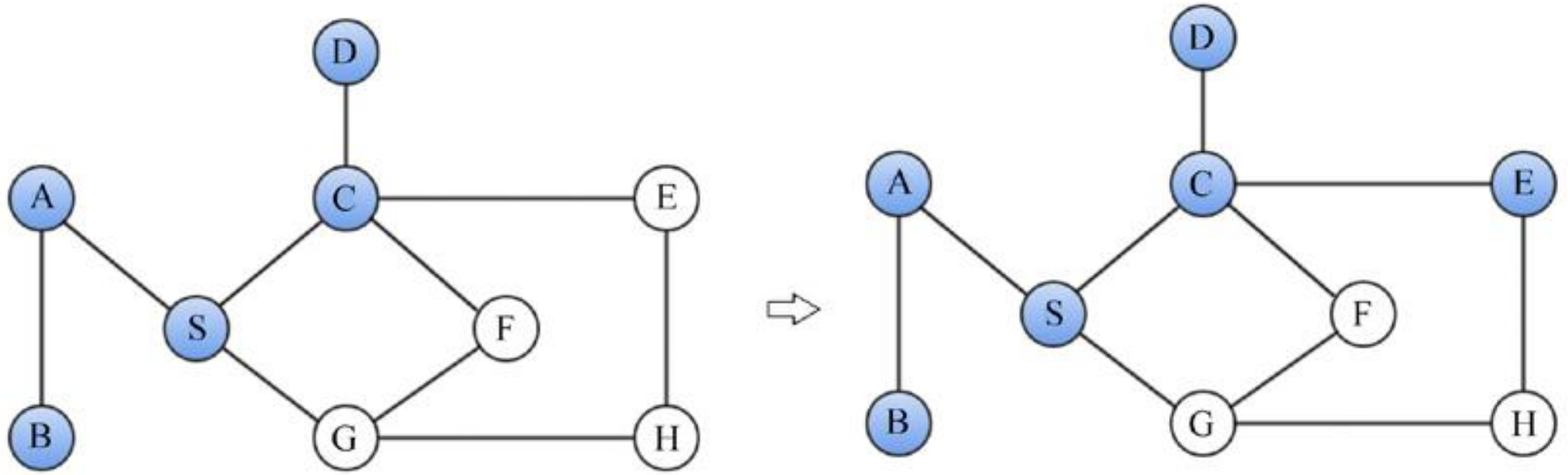
Duyệt theo chiều sâu

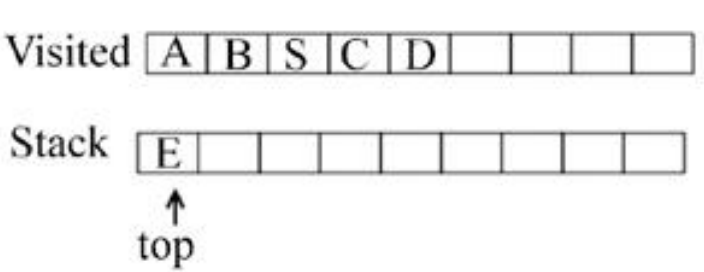
Ví dụ 4:

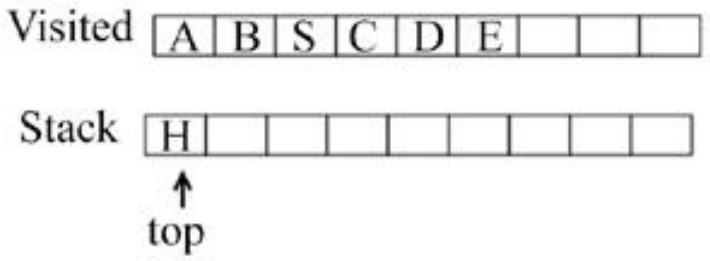


Duyệt theo chiều sâu

Ví dụ 4:

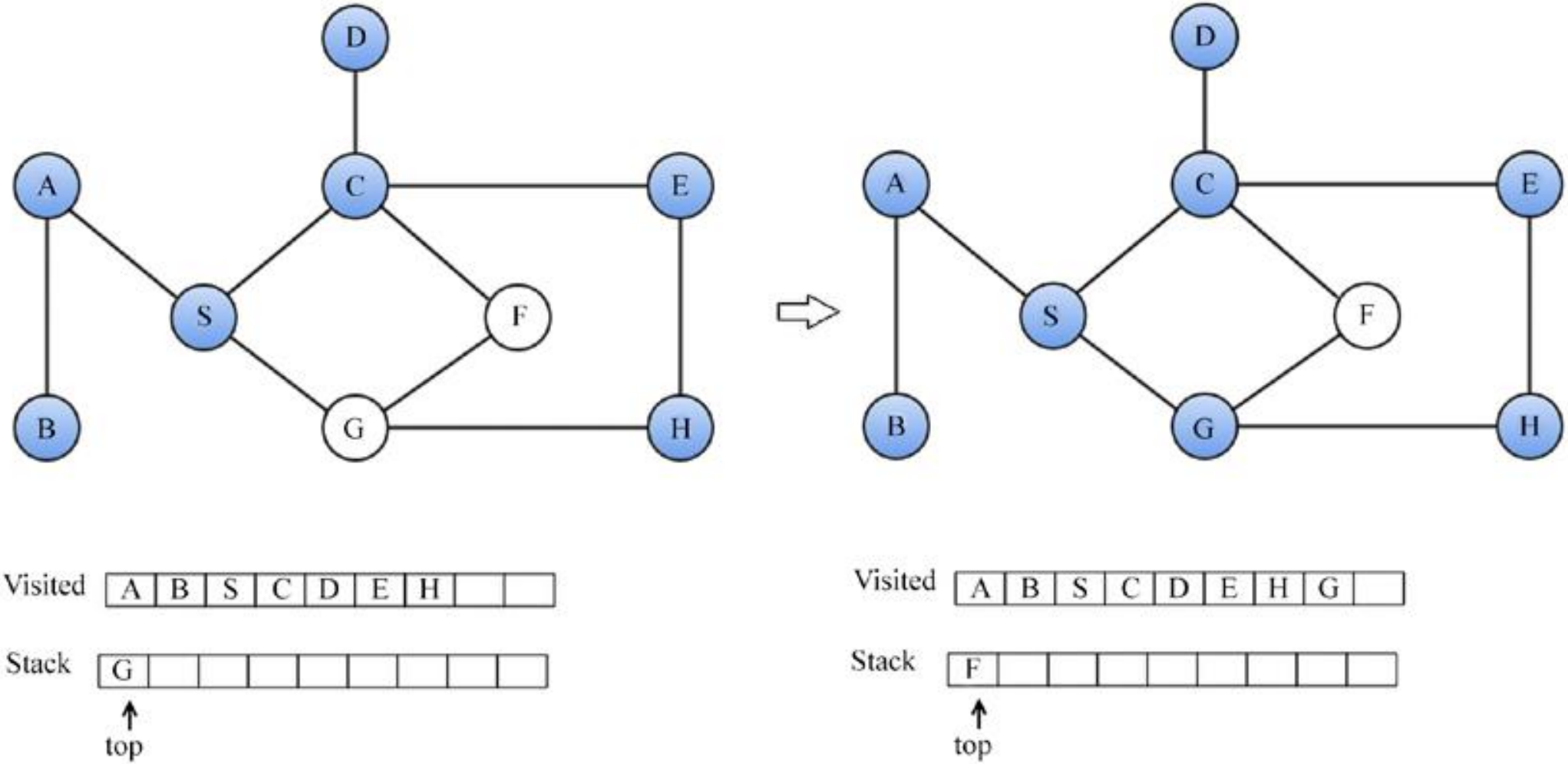






Duyệt theo chiều sâu

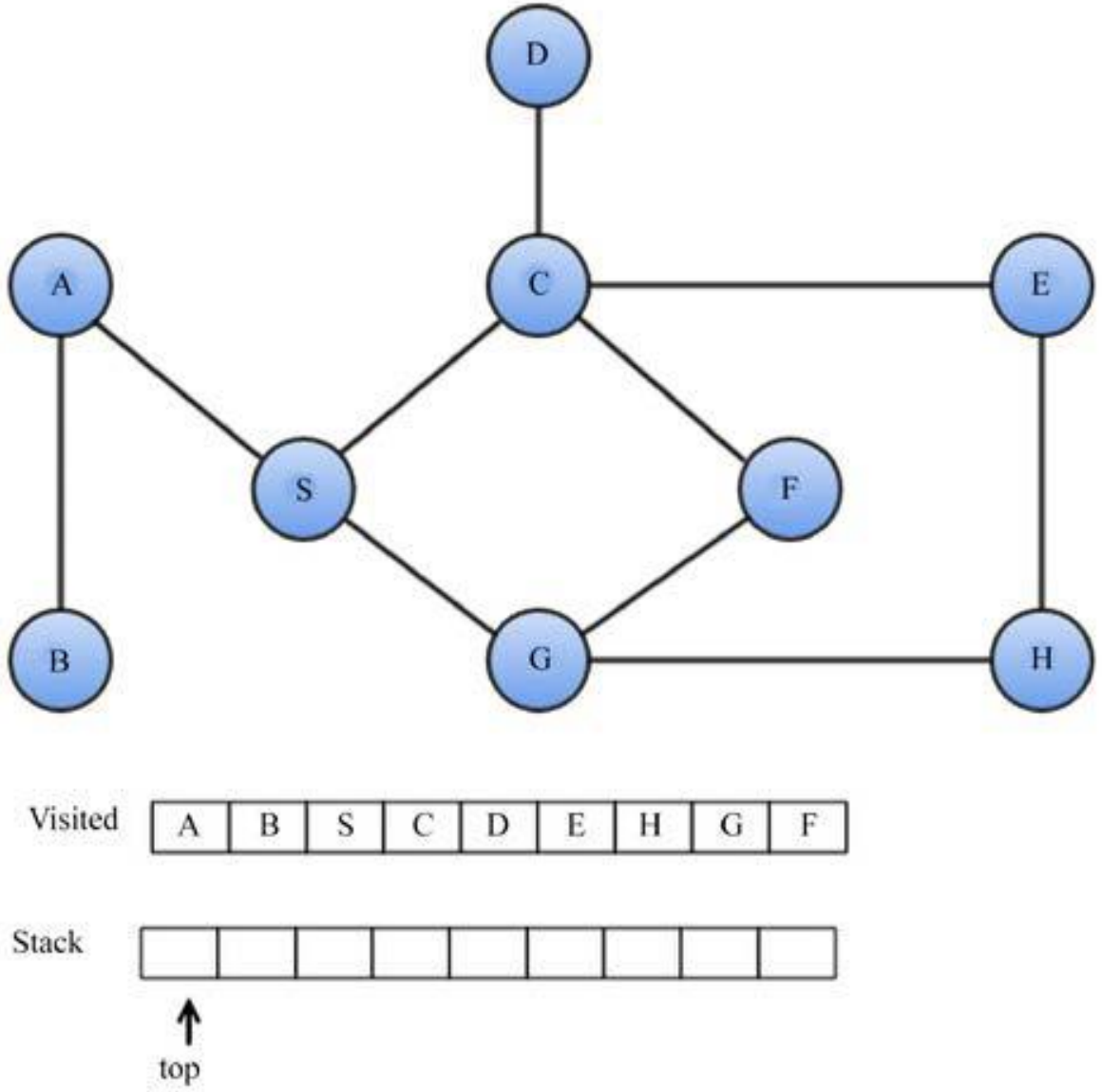
Ví dụ 4:



Duyệt theo chiều sâu

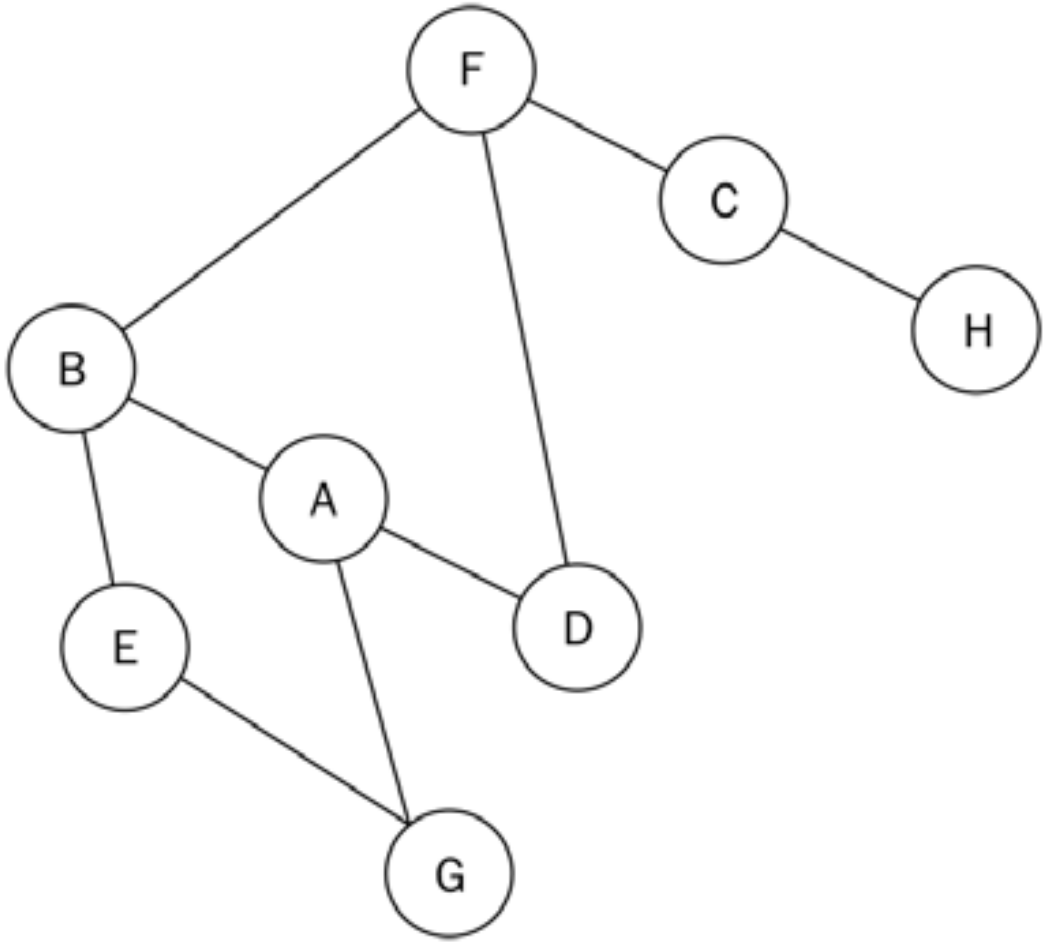
Ví dụ 4:

Output của DFS là A-B-S-C-D-E-H-G-F.



Duyệt theo chiều sâu

Ví dụ 5: Cho đồ thị sau, thực hiện duyệt theo chiều sâu với đỉnh bắt đầu A.
Output của DFS là:
A-B-E-G-F-C-H-D.



CMC UNIVERSITY

Duyệt theo chiều sâu

Cài đặt thuật toán DFS bằng Java (minh họa ví dụ 5):

```java
import java.util.*;

public class DFS1 {
    private Map<Character, List<Character>> adjacencyList; // Danh sách kề của đồ thị

    public DFS1() {
        adjacencyList = new HashMap<>();    // Khởi tạo danh sách kề
    }

    public void addEdge(char src, char dest) {      // Thêm cạnh vào đồ thị
        adjacencyList.computeIfAbsent(src, k -> new ArrayList<>()).add(dest);
    }

    public void dfs1(char source) {         // Thuật toán DFS bắt đầu từ đỉnh source
        Set<Character> visited = new HashSet<>(); // Đánh dấu các đỉnh đã được thăm
        dfsUtil(source, visited);
    }
```

CMC UNIVERSITY

Duyệt theo chiều sâu

```
18
19      // Hàm đệ quy thực hiện DFS
20      private void dfsUtil(char vertex, Set<Character> visited) {
21          visited.add(vertex);
22          System.out.print(vertex + " ");
23
24          List<Character> neighbors = adjacencyList.get(vertex);
25          if (neighbors != null) {
26              Collections.sort(neighbors); // Sắp xếp theo thứ tự bảng chữ cái
27              for (char neighbor : neighbors) {
28                  if (!visited.contains(neighbor)) {
29                      dfsUtil(neighbor, visited);
30                  }
31              }
32          }
33      }
34
```

CMC UNIVERSITY

Duyệt theo chiều sâu

```
Run | Debug
35      public static void main(String[] args) {
36          DFS1 graph = new DFS1();
37
38          graph.addEdge(src:'A', dest:'B');
39          graph.addEdge(src:'A', dest:'G');
40          graph.addEdge(src:'A', dest:'D');
41          graph.addEdge(src:'B', dest:'A');
42          graph.addEdge(src:'B', dest:'F');
43          graph.addEdge(src:'B', dest:'E');
44          graph.addEdge(src:'C', dest:'F');
45          graph.addEdge(src:'C', dest:'H');
46          graph.addEdge(src:'D', dest:'F');
47          graph.addEdge(src:'D', dest:'A');
48          graph.addEdge(src:'E', dest:'B');
49          graph.addEdge(src:'E', dest:'G');
50          graph.addEdge(src:'F', dest:'B');
51          graph.addEdge(src:'F', dest:'D');
52          graph.addEdge(src:'F', dest:'C');
53          graph.addEdge(src:'G', dest:'A');
54          graph.addEdge(src:'G', dest:'E');
55          graph.addEdge(src:'H', dest:'C');
56
57          System.out.println(x:"DFS starting from vertex A:");
58          graph.dfs1(source:'A');
59      }
60  }
```

```
DFS starting from vertex A:
A B E G F C H D
```

CMC UNIVERSITY

Duyệt theo chiều sâu

- **G được biểu diễn bởi danh sách kề**: Độ phức tạp thời gian của DFS là $O(|V|+|E|)$

Vì đỉnh w lân cận của v được xác định bằng cách dựa vào danh sách móc nối ứng với v. DFS chỉ xem xét mỗi nút trong một danh sách lân cận nhiều nhất một lần, mà có $2|E|$ nút danh sách (ứng với $|E|$ cạnh).

- **G được biểu diễn bởi ma trận kề**: Độ phức tạp thời gian của DFS là $O(|V|^2)$

Vì thời gian để xác định mọi đỉnh lân cận của v là $O(|V|)$. Tối đa có $|V|$ đỉnh được thăm nên thời gian tìm kiếm tổng quát là $O(|V|^2)$

- DFS có thể được áp dụng để giải bài toán mê cung, tìm các thành phần liên thông,xác định đồ thị có liên thông hay không, phát hiện chu trình trong đồ thị, tìm các cầu trong đồ thị, …

CMC UNIVERSITY

ĐƯỜNG ĐI NGẮN NHẤT

Đường đi ngắn nhất

Đường đi ngắn nhất từ nút nguồn đến nút đích:

- Đường đi có số lượng cạnh ít nhất từ nút nguồn đến nút đích, với đồ thị không có trọng số,
- Đường đi có giá trị tổng trọng số thấp nhất đi qua tập cạnh của đường đi, đối với đồ thị có trọng số.

Thông thường chúng ta cần sử dụng đồ thị để tìm đường đi giữa hai nút. Đôi khi, cần phải tìm tất cả các đường đi giữa các nút và trong một số trường hợp, chúng ta có thể cần tìm đường đi ngắn nhất giữa các nút. Ví dụ, trong các ứng dụng định tuyến, chúng ta thường sử dụng các thuật toán khác nhau để xác định đường đi ngắn nhất từ nút nguồn đến nút đích.

CMC UNIVERSITY

Đường đi ngắn nhất

Đường đi ngắn nhất trên đồ thị có trọng số

Cho đồ thị có trọng số G. ***Độ dài*** (hoặc trọng số) của đường đi là tổng trọng số của các cạnh của P. Tức là, nếu P = $((v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k))$, thì độ dài của P, kí hiệu w(P), được định nghĩa:

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

Khoảng cách từ đỉnh u đến đỉnh v trong G, kí hiệu d(u,v), là độ dài của đường đi có độ dài nhỏ nhất (cũng gọi là ***đường đi ngắn nhất***) từ u đến v, nếu đường đi tồn tại.

CMC UNIVERSITY

Thuật toán Dijkstra

- Thuật toán Dijkstra khá tương đồng với thuật toán tìm kiếm theo chiều sâu. Thuật toán xử lý như tìm kiếm theo chiều sâu, nhưng bắt đầu với nút nguồn duy nhất s và thăm mọi nút trong đồ thị.

- Mỗi đỉnh v trong đồ thị được gán một giá trị là tổng trọng số các cạnh trên đường đi từ nguồn v. Ban đầu đỉnh nguồn được gán giá trị 0. Tất cả các đỉnh khác ban đầu được gán giá trị vô cùng lớn.

(Giá trị vô cùng lớn được xem là giá trị lớn hơn tổng trọng số của tất cả các cạnh trong đồ thị)

Thuật toán Dijkstra

- Thuật toán Dijkstra duy trì hai tập hợp:

+ *Tập chưa được thăm* (*unvisited set)*. Đây là tập hợp đỉnh cần được xem xét khi tìm đường đi có chi phí tối thiểu. Tập hợp chưa được thăm có cùng mục đích như stack khi thực hiện tìm kiếm theo chiều sâu trên đồ thị.

+ *Tập được thăm* (*visited set)* chứa tất cả các đỉnh đã được tính toán đường đi với chi phí tối thiểu. Nó có cùng mục đích như tập đỉnh được thăm trong tìm kiếm theo chiều sâu trên đồ thị.

CMC UNIVERSITY

Thuật toán Dijkstra

1. Tìm đỉnh u (gọi là đỉnh hiện tại) trong tập các đỉnh chưa được thăm sao cho đường đi đến u có chi phí thấp nhất, và xóa u khỏi tập đỉnh chưa được thăm.
2. Thêm đỉnh hiện tại u vào tập đỉnh đã thăm
3. Đối với mọi đỉnh liền kề với đỉnh hiện tại, kiểm tra xem đỉnh kề này có thuộc tập đỉnh đã thăm hay không. Nếu đỉnh kề nằm trong tập đã thăm, thì ta biết chi phí tối thiểu của đường đi đến đỉnh này từ nguồn, vì vậy không làm gì cả.
4. Nếu đỉnh kề v không có trong tập đã thăm, tính chi phí mới của đường đi đến đỉnh kề bằng cách duyệt qua cạnh e từ đỉnh hiện tại đến đỉnh kề. Có thể tính chi phí mới bằng cách cộng chi phí đến đỉnh hiện tại và trọng số của e. Nếu chi phí mới này tốt hơn chi phí hiện tại thì cập nhật chi phí mới này và nhớ rằng đỉnh hiện tại là đỉnh trước của đỉnh kề. Ngoài ra, thêm đỉnh kề vào tập hợp chưa được thăm.

Thuật toán Dijkstra

Ví dụ 1: Cho đồ thị như bên dưới

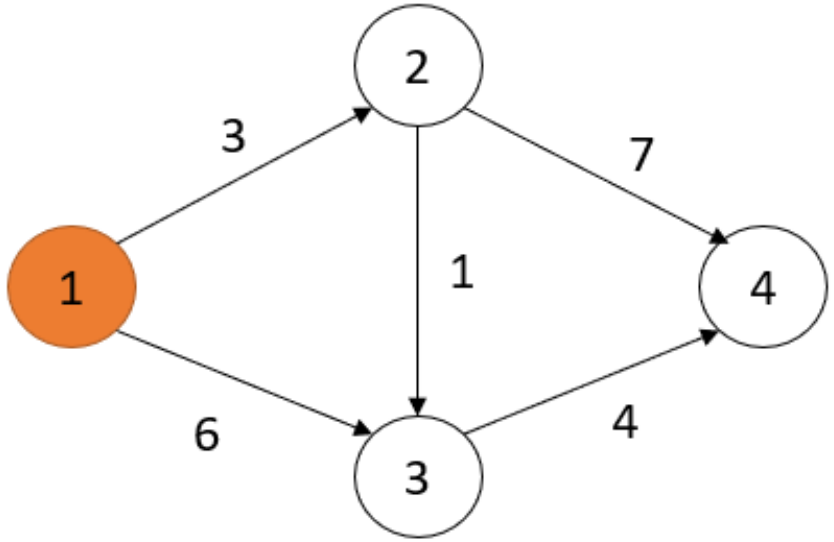
Ta cần khởi tạo hai mảng như sau:

- dist[u]: độ dài đường đi ngắn nhất từ đỉnh nguồn s đến đỉnh u.

Khởi tạo dist[u] = $\propto$ với mọi u, riêng dist[s] = 0

- visited[u]: đánh dấu khi đỉnh u được xem xét hoặc không. Khởi tạo các phần tử trong mảng có giá trị false

dist = [0, $\propto$, $\propto$, $\propto$], visited = [0, 0, 0, 0]



Thuật toán Dijkstra

Ví dụ 1:

dist = [0, ∝, ∝, ∝], visited = [0, 0, 0, 0]

1) u = 1, v = 2, dist[2] = dist[1] + (1, 2) = 3 < dist[2] = ∝, dist[2] = 3,
   u = 1, v = 3, dist[3] = dist[1] + (1, 3) = 6 < dist[3] = ∝, dist[3] = 6

   dist = [0, 3, 6, ∝], visited = [1, 0, 0, 0]

2) u = 2, v = 3, dist[3] = dist[2] + (2, 3) = 4 < dist[3] = 6, dist[3] = 4,
   u = 2, v = 4, dist[4] = dist[2] + (2, 4) = 10 < dist[4] = ∝, dist[4] = 10

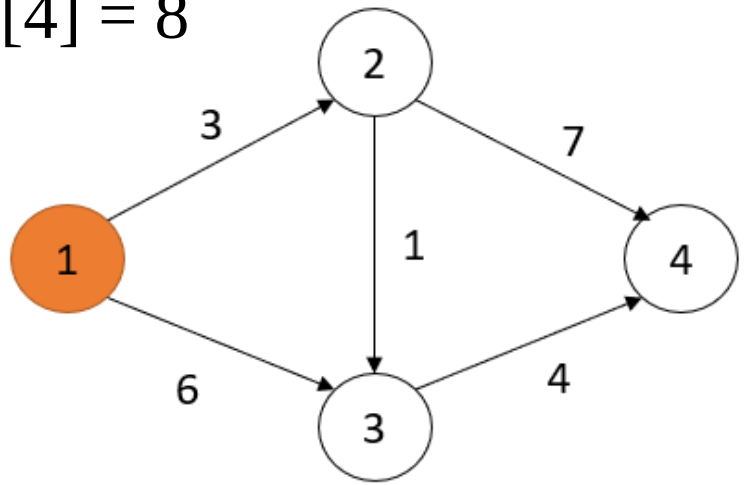
   dist = [0, 3, 4, 10], visited = [1, 1, 0, 0]

3) u = 3, v = 4, dist[4] = dist[3] + (3, 4) = 8 < dist[4] = 10, dist[4] = 8

   dist = [0, 3, 4, 8], visited = [1, 1, 1, 0]

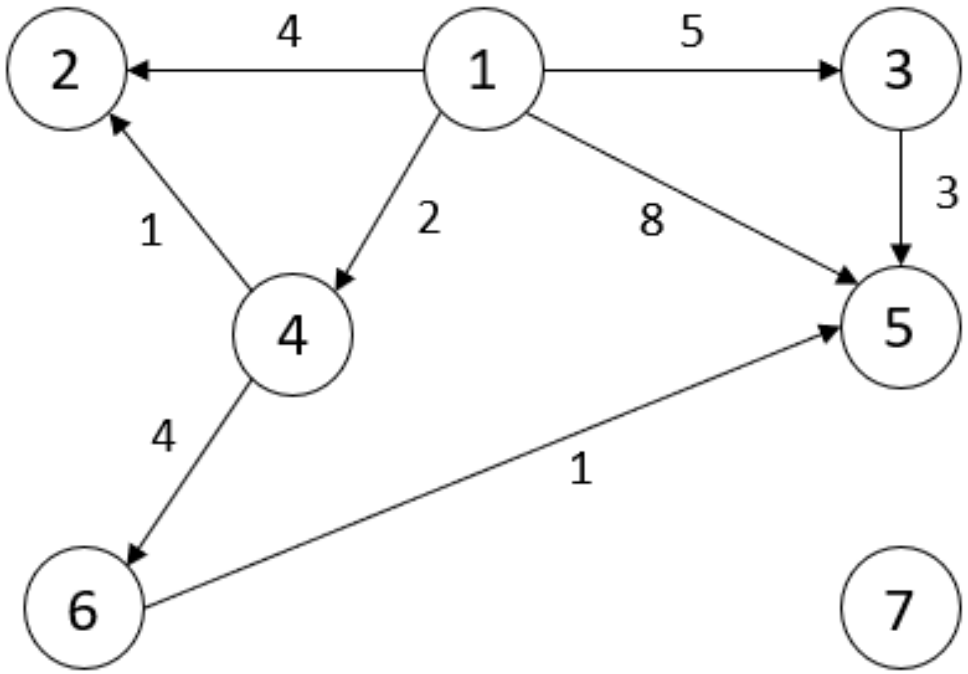
4) u = 4,

   dist = [0, 3, 4, 8], visited = [1, 1, 1, 1]



Thuật toán Dijkstra

Ví dụ 2:



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

CMC UNIVERSITY

Thuật toán Dijkstra

```java
import java.util.*;

public class DijkstraAlgorithm{
    private int V; // Số đỉnh trong đồ thị
    private List<List<Node>> adj; // Danh sách kề

    // Lớp Node đại diện cho một đỉnh trong đồ thị với trọng số
    class Node {
        int vertex;
        int weight;
        Node(int v, int w) {
            vertex = v;
            weight = w;
        }
    }
    public DijkstraAlgorithm(int vertices) {
        V = vertices;
        adj = new ArrayList<>(V);
        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<>());
        }
    }
    // Thêm cạnh vào đồ thị
    public void addEdge(int u, int v, int weight) {
        adj.get(u).add(new Node(v, weight));
        adj.get(v).add(new Node(u, weight)); // Nếu đồ thị vô hướng
    }
```

CMC UNIVERSITY

Thuật toán Dijkstra

```
28      // Hàm tìm đỉnh chưa được thăm có đường đi nhỏ nhất
29      private int minDistance(int[] dist, boolean[] visited) {
30          int min = Integer.MAX_VALUE, minIndex = -1;
31          for (int v = 0; v < V; v++) {
32              if (!visited[v] && dist[v] <= min) {
33                  min = dist[v];
34                  minIndex = v;
35              }
36          }
37          return minIndex;
38      }
39      // Hàm in kết quả
40      private void printSolution(int[] dist) {
41          System.out.println(x:"Đỉnh \t Khoảng cách từ nguồn");
42          for (int i = 0; i < V; i++) {
43              System.out.println((char)('A' + i) + " \t " + dist[i]);
44          }
45      }
```

CMC UNIVERSITY

Thuật toán Dijkstra

```java
// Hàm tìm đường đi ngắn nhất từ nguồn s đến tất cả các đỉnh
public void dijkstra(int s) {
    int[] dist = new int[V]; // Khoảng cách ngắn nhất từ nguồn đến đỉnh i
    boolean[] visited = new boolean[V]; // Đánh dấu đỉnh đã được thăm

    Arrays.fill(dist, Integer.MAX_VALUE);  // Khởi tạo khoảng cách các đỉnh với giá trị vô cùng lớn
    Arrays.fill(visited, val:false);       // đánh dấu tất cả các đỉnh chưa được thăm
    dist[s] = 0;   // Khoảng cách từ nguồn đến chính nó luôn là 0

    // Tìm đường đi ngắn nhất cho tất cả các đỉnh
    for (int count = 0; count < V; count++) {
        // Chọn đỉnh có khoảng cách nhỏ nhất từ tập các đỉnh chưa được thăm
        int u = minDistance(dist, visited);
        visited[u] = true;   // Đánh dấu đỉnh này là đã thăm

        // Cập nhật khoảng cách của các đỉnh kề với đỉnh vừa chọn
        for (Node neighbor : adj.get(u)) {
            // Cập nhật khoảng cách nếu neighbor chưa được thăm, có cạnh nối từ u đến neighbor.vertex,
            // và tổng khoảng cách từ nguồn đến u và từ u đến neighbor.vertex nhỏ hơn khoảng cách hiện tại của neighbor.vertex
            if (!visited[neighbor.vertex] && dist[u] != Integer.MAX_VALUE && dist[u] + neighbor.weight < dist[neighbor.vertex]) {
                dist[neighbor.vertex] = dist[u] + neighbor.weight;
            }
        }
    }
    printSolution(dist);      // In kết quả
}
```

CMC UNIVERSITY

Thuật toán Dijkstra

```
Run | Debug
public static void main(String[] args) {
    int V = 7; // Số đỉnh trong đồ thị
    DijkstraAlgorithm dijkstra = new DijkstraAlgorithm(V);

    // Thêm cạnh vào đồ thị
    dijkstra.addEdge(u:0, v:1, weight:4);
    dijkstra.addEdge(u:0, v:2, weight:5);
    dijkstra.addEdge(u:0, v:4, weight:8);
    dijkstra.addEdge(u:0, v:3, weight:2);
    dijkstra.addEdge(u:1, v:2, weight:3);
    dijkstra.addEdge(u:1, v:3, weight:1);
    dijkstra.addEdge(u:2, v:4, weight:3);
    dijkstra.addEdge(u:3, v:5, weight:4);
    dijkstra.addEdge(u:4, v:5, weight:1);

    // Đỉnh xuất phát là 0 (đánh số từ 0)
    dijkstra.dijkstra(s:0);
}
}
```

Đỉnh	Khoảng cách từ nguồn
A	0
B	3
C	5
D	2
E	7
F	6
G	2147483647

The complexity of Dijkstra's Algorithm is $O(n^2)$, using a priority queue, Dijkstra's Algorithm will run in O(nlogn) time

CMC UNIVERSITY

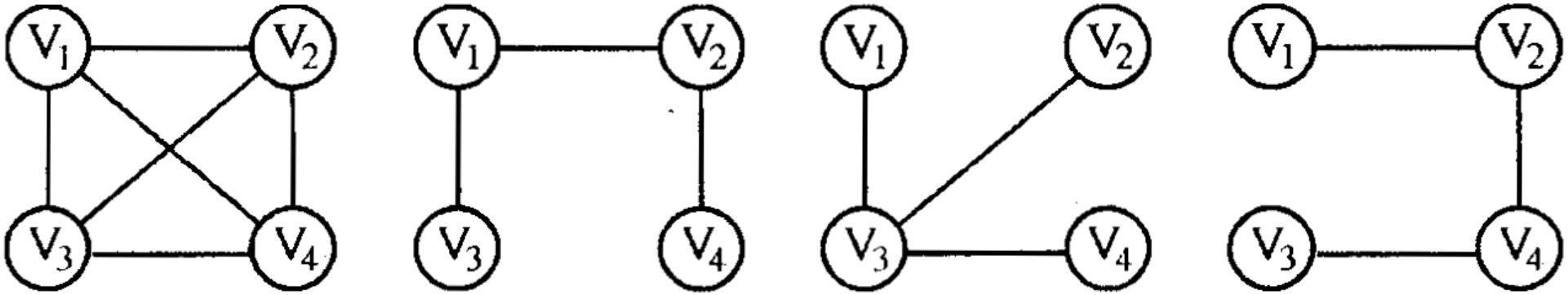
CÂY KHUNG CỰC TIỂU

Cây khung

Nếu đồ thị G liên thông thì một phép tìm kiếm theo chiều rộng hoặc theo chiều sâu, xuất phát từ một đỉnh bất kỳ, cũng cho phép tìm được mọi đỉnh của G. Trong trường hợp này, các cạnh của G được chia thành hai tập hợp:

Tập T bao gồm tất cả các cạnh được dùng tới hoặc được duyệt trong tìm kiếm, và tập B bao gồm các cạnh còn lại.

Tất cả các cạnh trong T cùng với các đỉnh tương ứng sẽ tạo thành một cây bao gồm mọi đỉnh của G. Một cây như vậy gọi là cây khung của G.

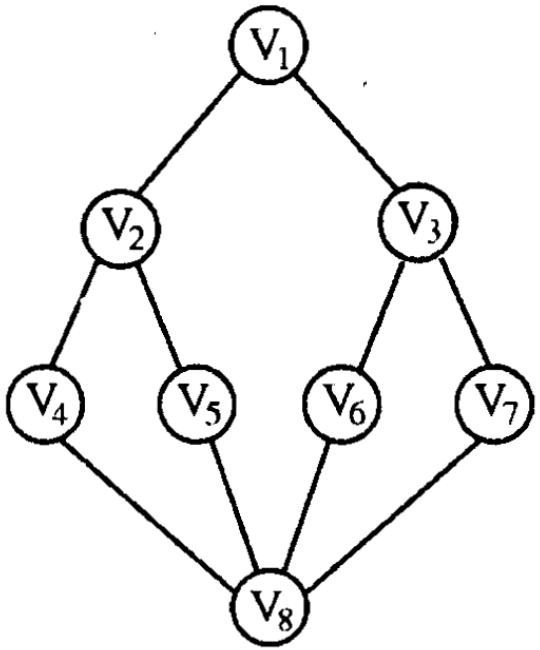


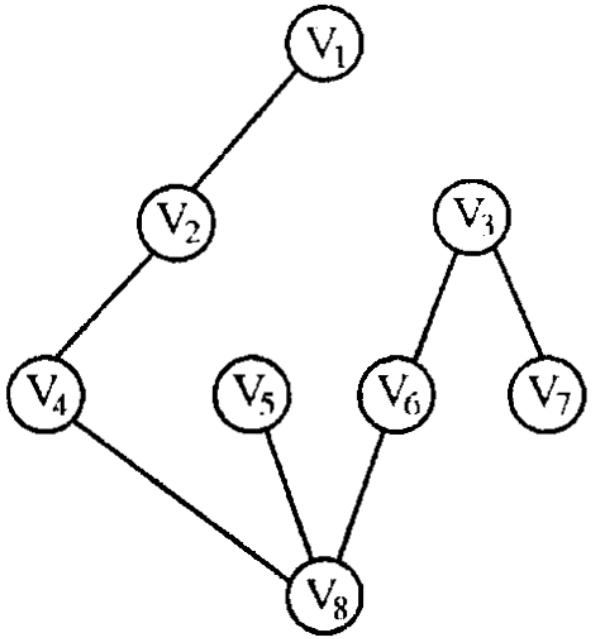
Cây khung

Tùy theo phương pháp DFS hoặc BFS được sử dụng mà cây khung tương ứng sẽ được gọi là cây khung theo chiều sâu hoặc cây khung theo chiều rộng.

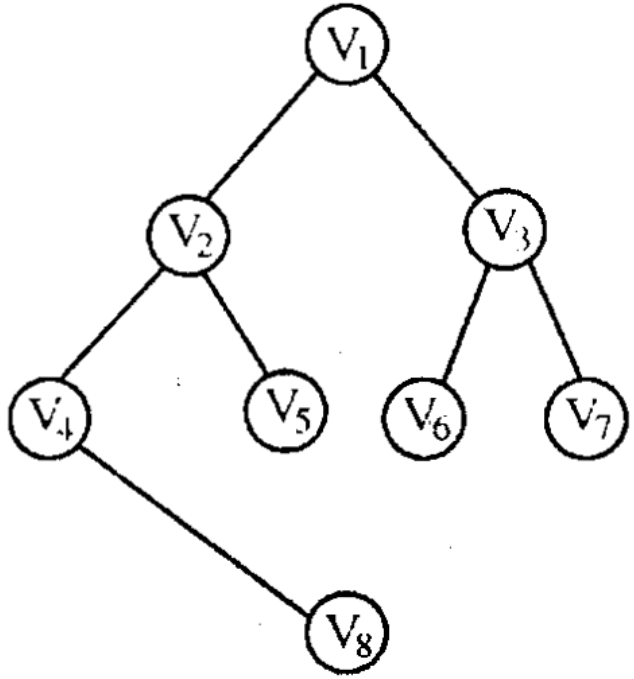
a) Cây khung DFS (V_1)

b) Cây khung BFS (V_1)





a)

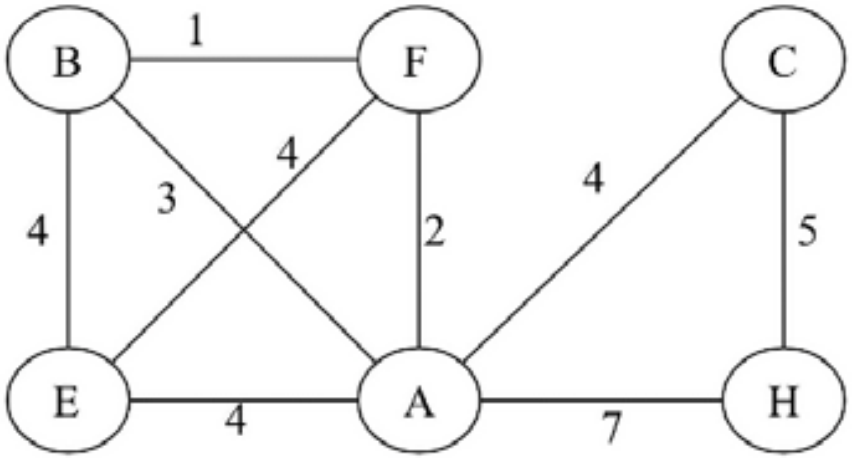


b)

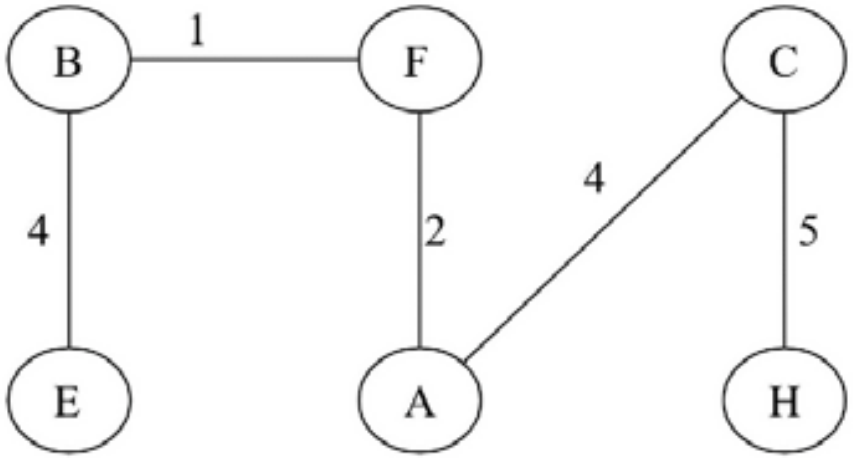
Cây khung cực tiểu

Cây khung cực tiểu (Minimum Spanning Tree - MST) là một cây khung của đồ thị liên thông có trọng số cạnh, với tổng các trọng số cạnh thấp nhất có thể và không có chu trình.

Cho đồ thị liên thông G, trong đó G = (V, E) với các trọng số cạnh giá trị thực, một MST là một đồ thị con với tập con các cạnh $T \subseteq E$ sao cho tổng trọng số các cạnh là tối thiểu và không có chu trình.



A graph



A minimum spanning tree

Cây khung cực tiểu

- Có nhiều cây khung có thể kết nối tất cả các nút của đồ thị mà không có bất kỳ chu trình nào, nhưng cây khung có trọng số tối thiểu là cây khung có tổng trọng số cạnh thấp nhất (còn gọi là chi phí) trong số tất cả các cây khung.

- MST có nhiều ứng dụng thế giới thực. Chẳng hạn, MST được dùng phổ biến trong thiết kế mạng đường bộ, cáp thủy lực, mạng cáp điện, và bài toán phân tích cụm.

CMC UNIVERSITY

Thuật toán Kruskal

- Thuật toán **Kruskal** được sử dụng phổ biến để tìm cây khung cực tiểu từ đồ thị vô hướng, liên thông và có trọng số.
- Dựa trên cách tiếp cận tham lam: trước tiên, tìm cạnh có trọng số thấp nhất và thêm nó vào cây, sau đó trong mỗi lần lặp, thêm cạnh có trọng số thấp nhất vào cây khung để chúng không tạo thành một chu trình..
- Trong thuật toán này, ban đầu, coi tất cả các đỉnh của đồ thị là một cây riêng biệt, sau đó trong mỗi lần lặp, chọn cạnh có trọng số thấp nhất sao cho nó không tạo thành một chu trình. Những cây riêng biệt này được kết hợp lại và phát triển thành cây khung. Quá trình này được lặp lại cho đến khi tất cả các đỉnh được xử lý.

Thuật toán Kruskal

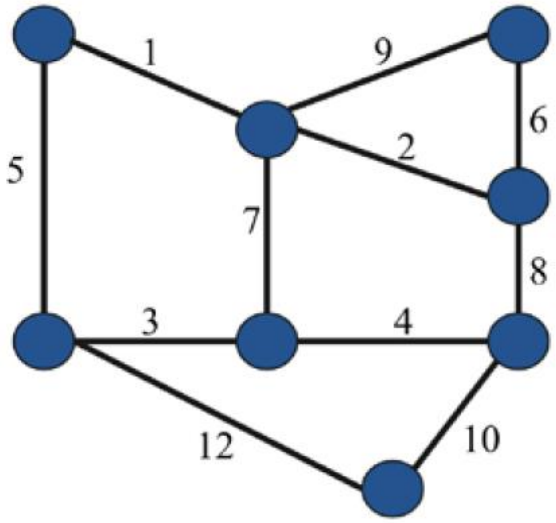
Thuật toán thực hiện như sau:

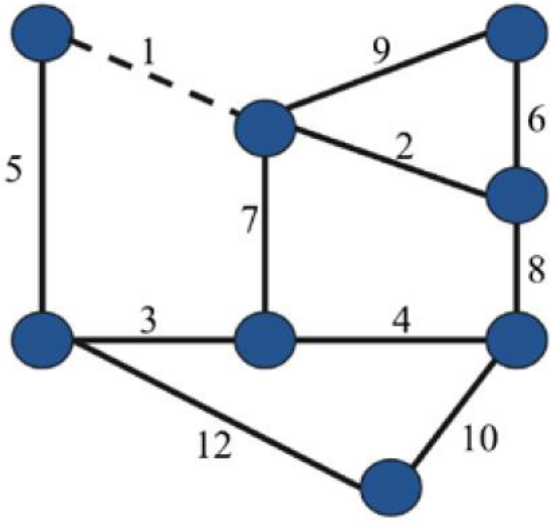
1. Khởi tạo một MST rỗng (M) không có cạnh nào

2. Sắp xếp tất cả các cạnh theo trọng số của chúng

3. Với mỗi cạnh trong danh sách đã sắp xếp, bổ sung từng cạnh vào MST (M) sao cho nó không tạo thành chu trình.

Ví dụ 1:

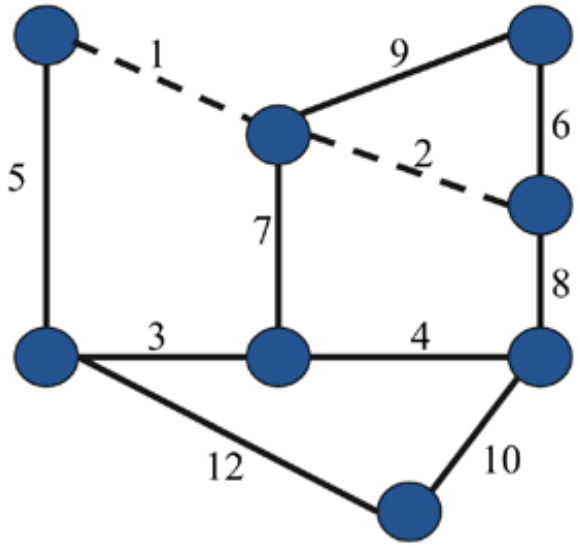


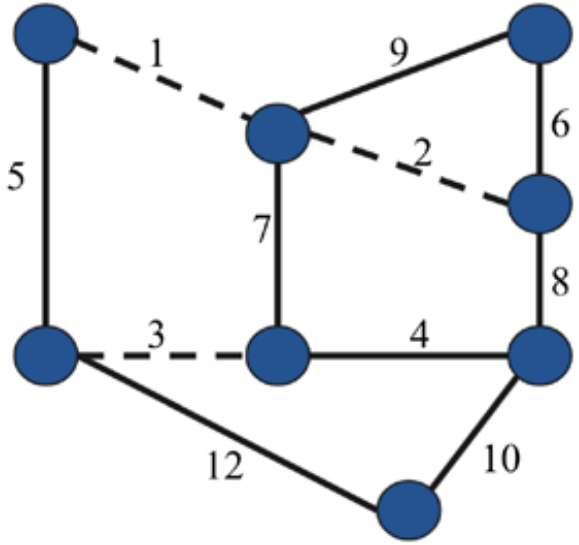


Chọn cạnh đầu tiên có trọng số thấp nhất trong cây khung

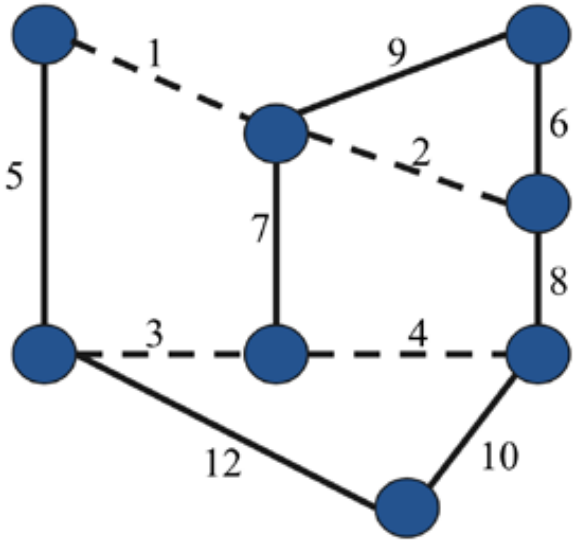
CMC UNIVERSITY

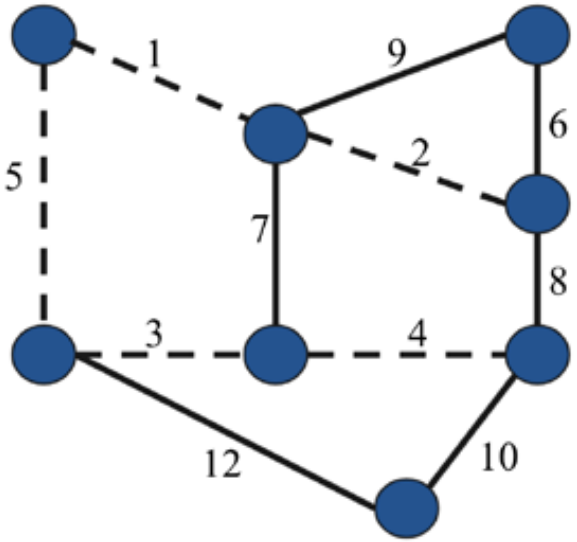
Thuật toán Kruskal





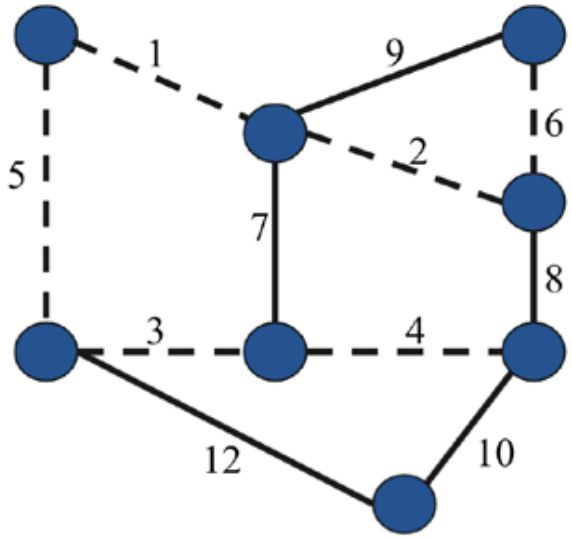
Chọn các cạnh có trọng số 2 và 3 trong cây khung

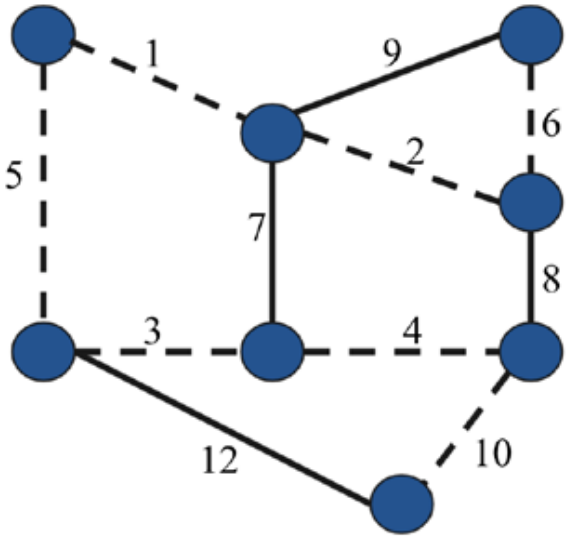




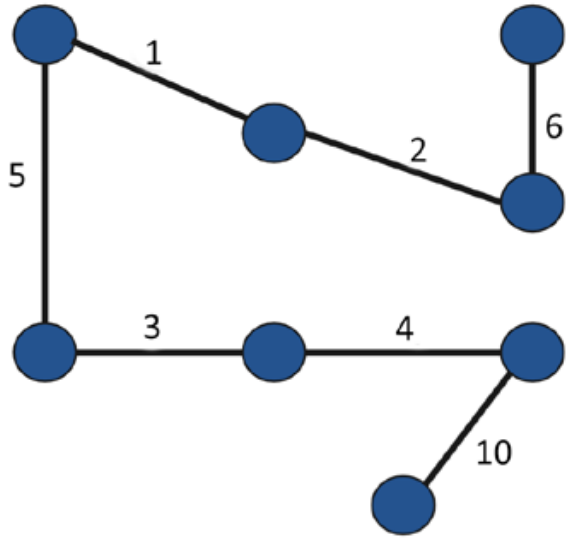
Chọn các cạnh có trọng số 4 và 5 trong cây khung

Thuật toán Kruskal





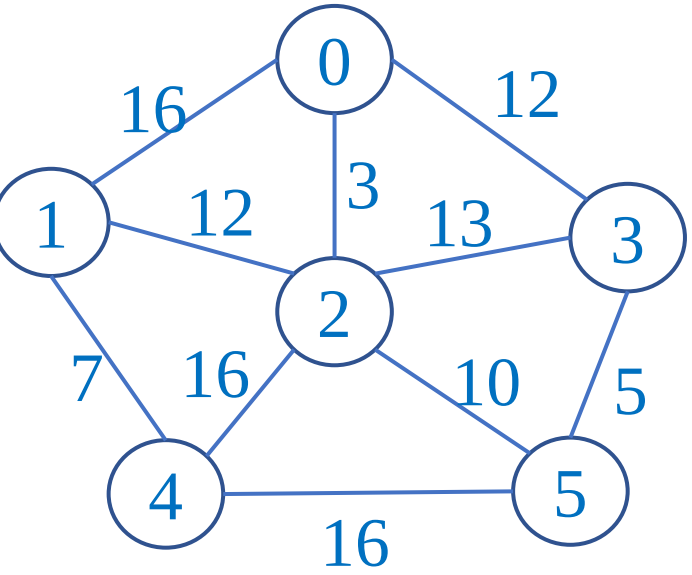
Chọn các cạnh có trọng số 6 và 10 trong cây khung

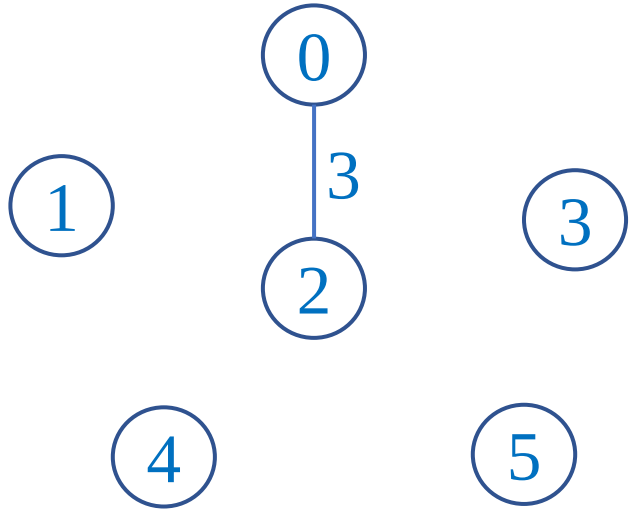


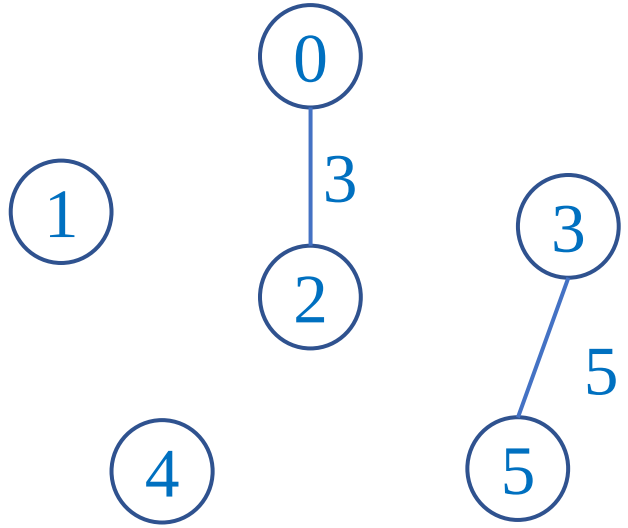
Cây khung cuối cùng được tạo sử dụng thuật toán Kruskal

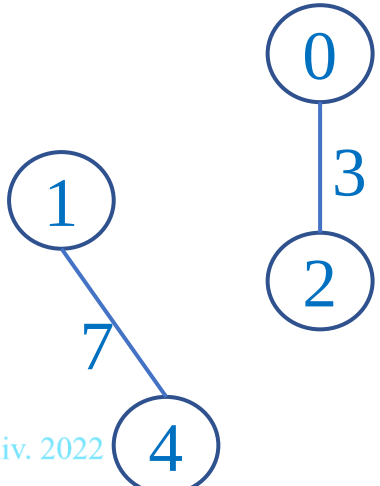
Thuật toán Kruskal

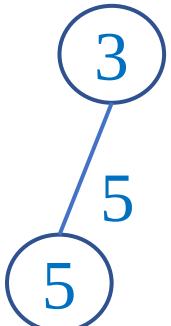
Ví dụ 2

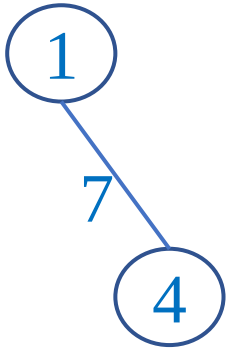


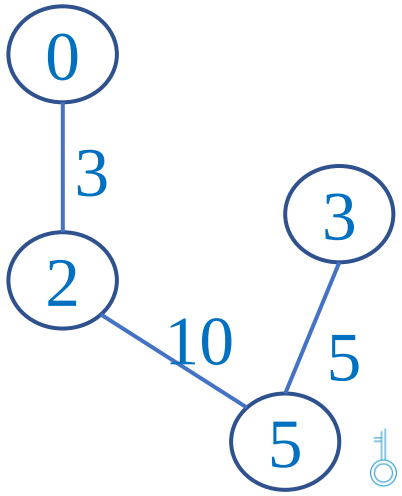


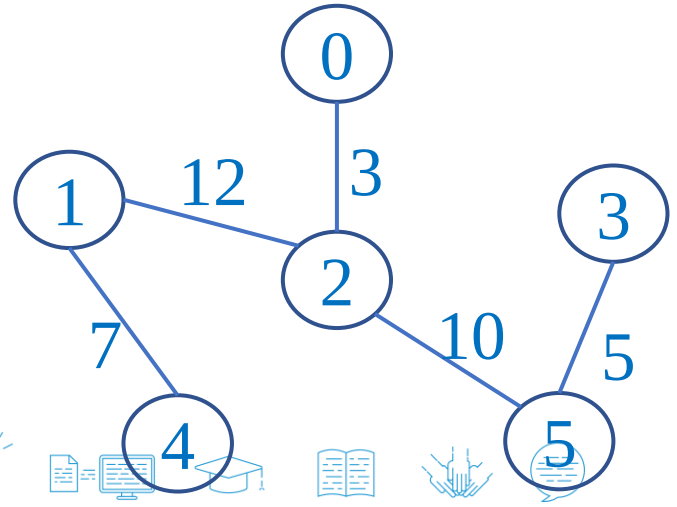












CMC UNIVERSITY

Thuật toán Kruskal

```java
import java.util.*;

class Edge implements Comparable<Edge> {     // Lớp biểu diễn các cạnh của đồ thị
    int src, dest, weight;
    public Edge(int src, int dest, int weight) {
        this.src = src;
        this.dest = dest;
        this.weight = weight;
    }
    @Override
    public int compareTo(Edge other) {       // So sánh các cạnh qua trọng số, để sx các cạnh
        return this.weight - other.weight;
    }
}
```

CMC UNIVERSITY

Thuật toán Kruskal

```
16  class UnionFind {        // Lớp quản lý các tập hợp rời rạc
17      private int[] parent, rank;
18      public UnionFind(int size) {
19          parent = new int[size];
20          rank = new int[size];
21          for (int i = 0; i < size; i++) {
22              parent[i] = i;
23              rank[i] = 0;
24          }
25      }
26
27      public int find(int node) {      // Tìm gốc của cây chứa node
28          if (parent[node] != node) {
29              parent[node] = find(parent[node]); // Path compression giảm độ sâu cây
30          }
31          return parent[node];
32      }
```

CMC UNIVERSITY

Thuật toán Kruskal

```
33      //Hợp nhất 2 cây chứa node1 và node2 và giữ cây cân bằng
34      public void union(int node1, int node2) {
35          int root1 = find(node1);
36          int root2 = find(node2);
37          if (root1 != root2) {
38              if (rank[root1] < rank[root2]) {
39                  parent[root1] = root2;
40              } else if (rank[root1] > rank[root2]) {
41                  parent[root2] = root1;
42              } else {
43                  parent[root2] = root1;
44                  rank[root1]++;
45              }
46          }
47      }
48  }
```

CMC UNIVERSITY

Thuật toán Kruskal

```
50  public class KruskalAlgorithm {      // Lớp để thực hiện thuật toán Kruskal
51      private int vertices;
52      private List<Edge> edges;
53
54      public KruskalAlgorithm(int vertices) {
55          this.vertices = vertices;
56          this.edges = new ArrayList<>();
57      }
58
59      public void addEdge(int src, int dest, int weight) {
60          edges.add(new Edge(src, dest, weight));
61      }
```

CMC UNIVERSITY

Thuật toán Kruskal

```
63      public List<Edge> kruskalMST() {
64          List<Edge> result = new ArrayList<>();
65          Collections.sort(edges);
66
67          UnionFind uf = new UnionFind(vertices);
68
69          for (Edge edge : edges) {
70              int rootSrc = uf.find(edge.src);
71              int rootDest = uf.find(edge.dest);
72
73              if (rootSrc != rootDest) {
74                  result.add(edge);
75                  uf.union(rootSrc, rootDest);
76              }
77          }
78          return result;
79      }
```

CMC UNIVERSITY

Thuật toán Kruskal

```
Run | Debug
81      public static void main(String[] args) {
82          KruskalAlgorithm graph = new KruskalAlgorithm(vertices:6);
83
84          graph.addEdge(src:0, dest:1, weight:16);
85          graph.addEdge(src:0, dest:2, weight:3);
86          graph.addEdge(src:0, dest:3, weight:12);
87          graph.addEdge(src:1, dest:2, weight:12);
88          graph.addEdge(src:1, dest:4, weight:7);
89          graph.addEdge(src:2, dest:3, weight:13);
90          graph.addEdge(src:2, dest:4, weight:16);
91          graph.addEdge(src:2, dest:5, weight:10);
92          graph.addEdge(src:3, dest:5, weight:5);
93          graph.addEdge(src:4, dest:5, weight:16);
94
95          List<Edge> mst = graph.kruskalMST();
96          System.out.println(x:"Edges in the Minimum Spanning Tree:");
97          for (Edge edge : mst) {
98              System.out.println(edge.src + " - " + edge.dest + ": " + edge.weight);
99          }
100     }
101 }
```

```
Edges in the Minimum Spanning Tree:
0 - 2: 3
3 - 5: 5
1 - 4: 7
2 - 5: 10
1 - 2: 12
```

CMC UNIVERSITY

Thuật toán Kruskal

Thuật toán Kruskal có nhiều ứng dụng thế giới thực, như giải bài toán người du lịch (the traveling salesman problem - TSP), trong đó xuất phát từ một thành phố, ta phải thăm tất cả các thành phố khác trong mạng với tổng chi phí nhỏ nhất và chỉ thăm mỗi thành phố đúng một lần.

Các ứng dụng khác như mạng truyền hình, mạng LAN, và lưới điện.

Độ phức tạp của thuật toán Kruskal là O (E log (E)) hoặc O (E log(V)), trong đó E là số cạnh và V là số đỉnh của đồ thị.

CMC UNIVERSITY

Thuật toán Prim

Thuật toán Prim cũng dựa trên cách tiếp cận tham lam để tìm cây khung có chi phí tối thiểu.

Thuật toán Prim rất giống với thuật toán Dijkstra tìm đường đi ngắn nhất trên đồ thị.

- Bắt đầu với một đỉnh tùy ý s làm đỉnh xuất phát, sau đó kiểm tra các cạnh đi ra từ s và lấy cạnh có chi phí (hoặc trọng số) nhỏ nhất (s, x).
- Tiếp theo, trong số các cạnh kề với hai đỉnh s hoặc x, tìm cạnh có trọng số nhỏ nhất, cạnh này dẫn đến đỉnh thứ 3 y, ta thu được cây bộ phận gồm 3 đỉnh và 2 cạnh.
- Quá trình này tiếp tục đến khi thu được cây n đỉnh và n – 1 cạnh, là cây khung nhỏ nhất cần tìm.

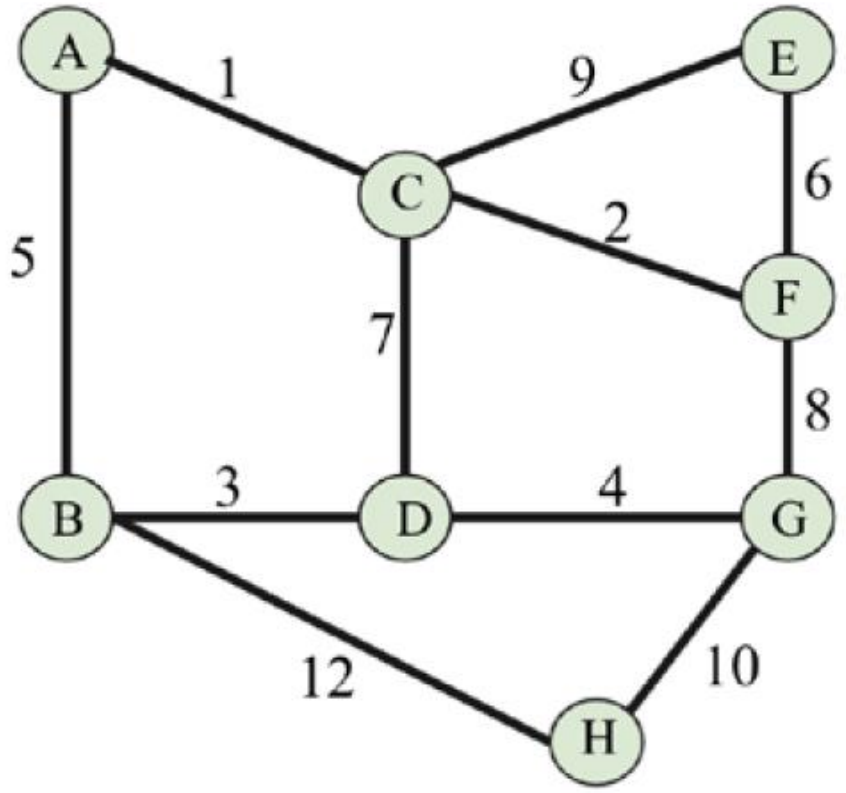
Thuật toán Prim

Thuật toán thực hiện như sau:

1. Tạo một danh sách chứa tất cả các cạnh và trọng số của chúng
2. Lấy từng cạnh có trọng số nhỏ nhất từ danh sách và phát triển cây theo cách không tạo thành chu trình

3. Lặp lại bước 2 đến khi tất cả các đỉnh được thăm.

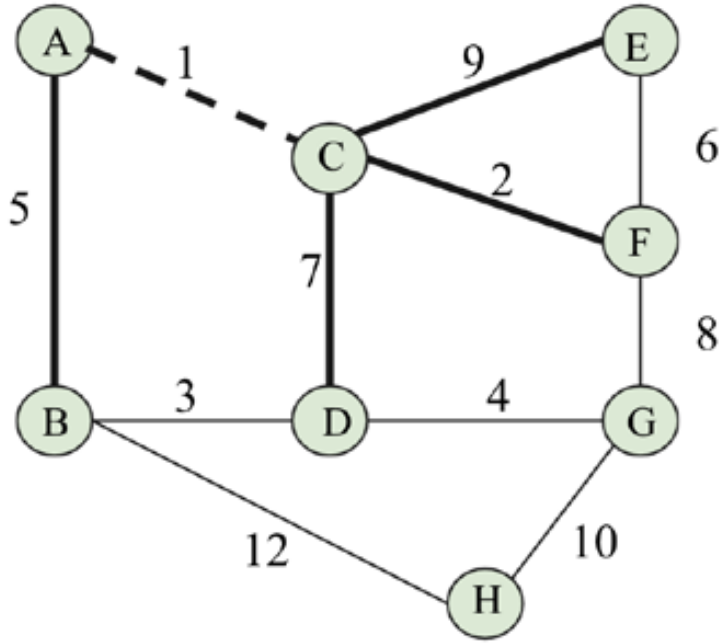
Ví dụ 3:
Cho đồ thị như hình bên, với đỉnh xuất phát A.
- Xét các cạnh đi ra từ A là AB and AC;
chọn AC vì nó có trọng số nhỉ nhất là 1,

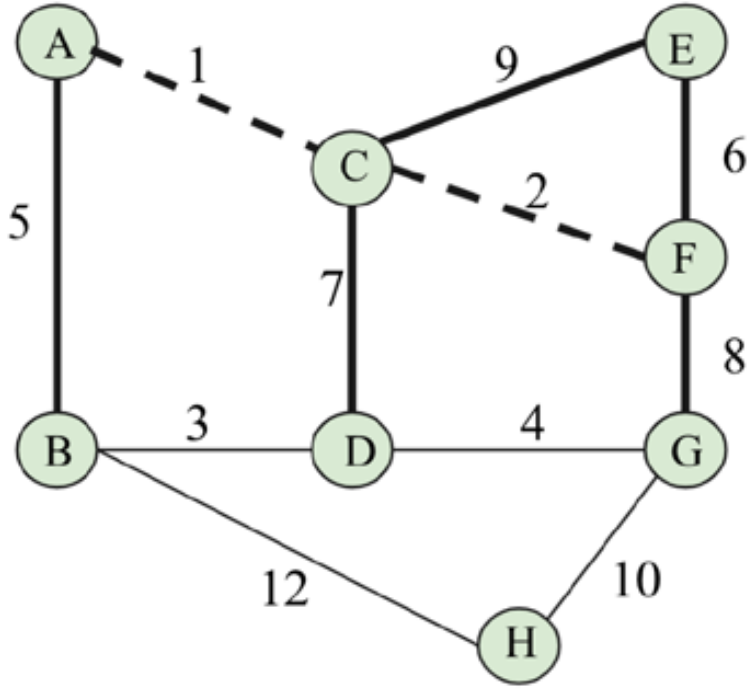


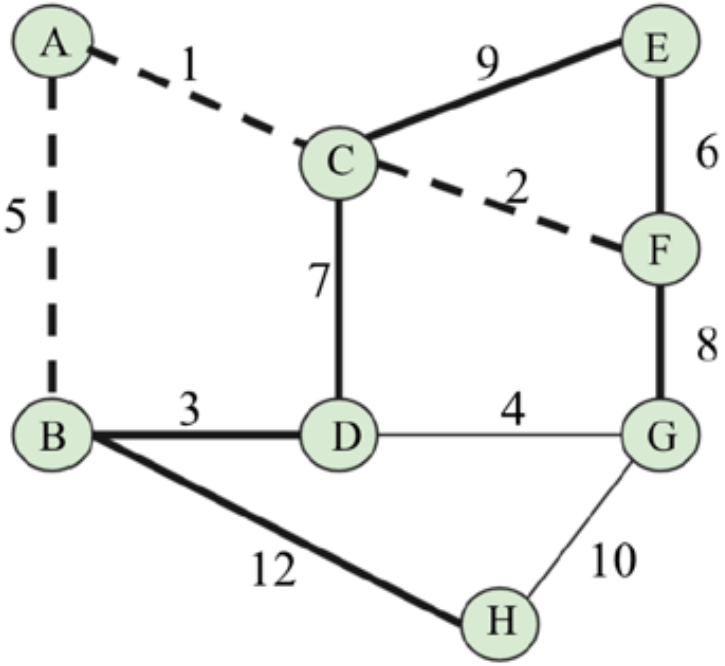
Thuật toán Prim

Ví dụ 3:

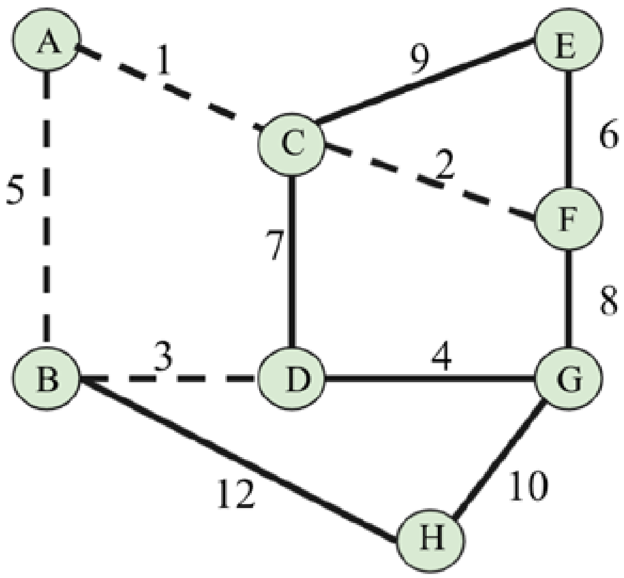
- Tiếp theo, xét các cạnh đi ra từ cạnh AC có trọng số nhỏ nhất, gồm AB, CD, CE, CF, Cạnh CF được chọn vì có trọng số nhỏ nhất là 2.
- Tương tự như vậy, tiếp tục phát triển cây, tiếp theo chọn cạnh có trọng số thấp nhất, tức là AB, BD, DG, GH.

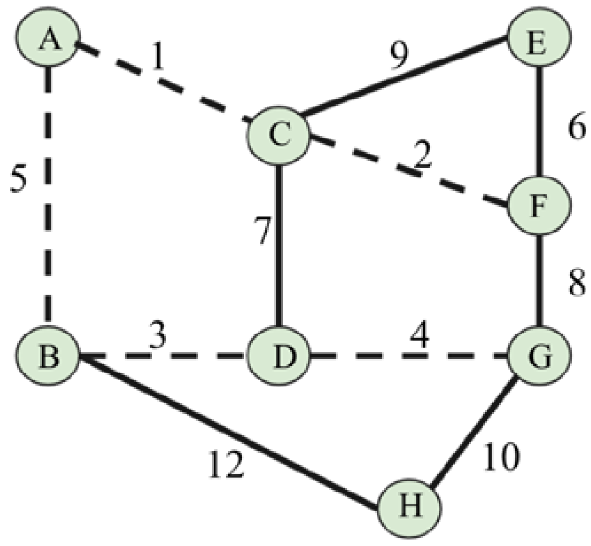


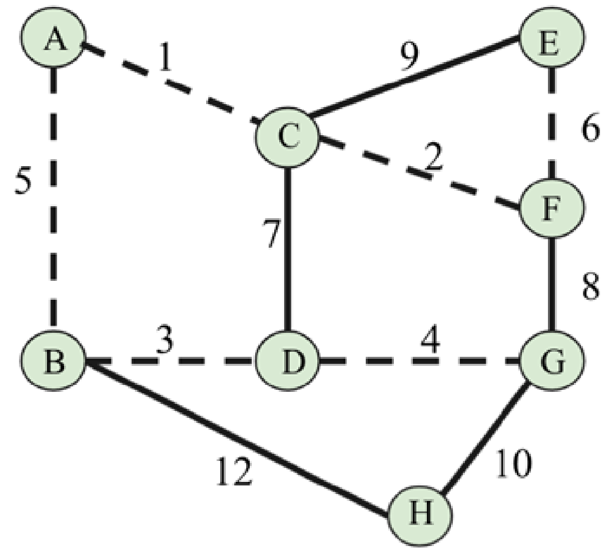


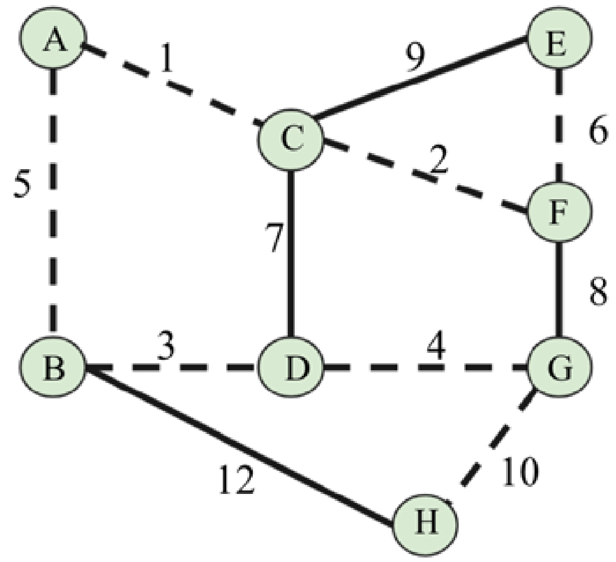


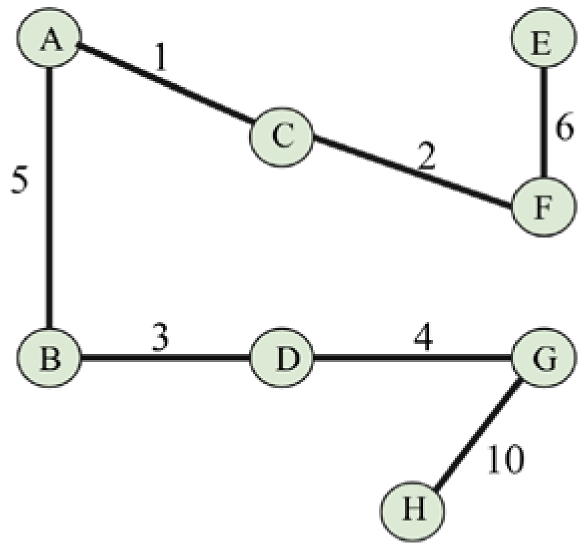
Thuật toán Prim











CMC UNIVERSITY

Thuật toán Prim

- Thuật toán Prim cũng có nhiều ứng dụng trong thế giới thực. Các ứng dụng của thuật toán Kruskal, cũng có thể sử dụng bởi thuật toán Prim. Các ứng dụng khác bao gồm mạng lưới đường bộ, phát triển trò chơi, v.v.

- Vì thuật toán Kruskal và Prim đều cùng mục đích tìm MST, vậy nên sử dụng thuật toán nào ? Nói chung, nó phụ thuộc vào cấu trúc của đồ thị. Với đồ thị n đỉnh và m cạnh, độ phức tạp thời gian trong trường hợp tồi nhất của thuật toán Kruskal là O(m logn), và của thuật toán Prim là O(m + nlogn). Do vậy, thuật toán Prim hoạt động tốt hơn khi đồ thị dày, trong khi thuật toán Kruskal tốt hơn khi đồ thị thưa.

CMC UNIVERSITY

Exercises

1. Let G be an undirected graph whose vertices are the integers 1 through 8, and let the adjacent vertices of each vertex be given by the table below:

vertex	adjacent vertices
1	(2, 3, 4)
2	(1, 3, 4)
3	(1, 2, 4)
4	(1, 2, 3, 6)
5	(6, 7, 8)
6	(4, 5, 7)
7	(5, 6, 8)
8	(5, 7)

Assume that, in a traversal of G, the adjacent vertices of a given vertex are returned in the same order as they are listed in the table above.

a. Draw G.

b. Give the sequence of vertices of G visited using a DFS traversal starting at vertex 1.

c. Give the sequence of vertices visited using a BFS traversal starting at vertex 1.

CMC UNIVERSITY

Exercises

2. Bob loves foreign languages and wants to plan his course schedule for the following years. He is interested in the following nine language courses: LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, and LA169. The course prerequisites are::

- LA15: (none)
- LA16: LA15
- LA22: (none)
- LA31: LA15
- LA32: LA16, LA31
- LA126: LA22, LA32
- LA127: LA16
- LA141: LA22, LA16
- LA169: LA32

In what order can Bob take these courses, respecting the prerequisites?

CMC UNIVERSITY

Exercises

3. There are eight small islands in a lake, and the state wants to build seven bridges to connect them so that each island can be reached from any other one via one or more bridges. The cost of constructing a bridge is proportional to its length. The distances between pairs of islands are given in the following table.

	1	2	3	4	5	6	7	8
1	-	240	210	340	280	200	345	120
2	-	-	265	175	215	180	185	155
3	-	-	-	260	115	350	435	195
4	-	-	-	-	160	330	295	230
5	-	-	-	-	-	360	400	170
6	-	-	-	-	-	-	175	205
7	-	-	-	-	-	-	-	305
8	-	-	-	-	-	-	-	-

Find which bridges to build to minimize the total construction cost?


CMC UNIVERSITY
THANK YOU