Mục lục

[**I.** **Duyệt đồ thị với thuật toán DFS, BFS theo 2 cách:** 1](#_Toc135205619)

[**1.** **Thuật toán DFS.** 1](#_Toc135205620)

[**a)** **Dùng đệ qui.** 1](#_Toc135205621)

[**b)** **Không dùng đệ qui.** 1](#_Toc135205622)

[**2.** **Thuật toán BFS.** 2](#_Toc135205623)

[**a)** **Dùng đệ qui.** 2](#_Toc135205624)

[**b)** **Không dùng đệ qui.** 2](#_Toc135205625)

[**3.** **So sánh và nhận định trường hợp áp dụng cho riêng từng cách.** 3](#_Toc135205626)

[**II.** **Kiểm tra 1 đồ thị vô hướng có thỏa mãn là đồ thị Euler hay nửa Euler** 6](#_Toc135205627)

[**III.** **Kiểm tra 1 đồ thị có hướng có thỏa mãn là đồ thị Euler hay nửa Euler** 6](#_Toc135205628)

[**IV.** **Xây dựng cây khung đồ thị với : 1/DFS, 2/BFS** 7](#_Toc135205629)

[**1)** **Xây dựng cây khung đồ thị bằng thuật toán DFS** 8](#_Toc135205630)

[**2)** **Xây dựng cây khung đồ thị bằng thuật toán BFS** 9](#_Toc135205631)

[**V.** **Xây dựng cây khung nhỏ nhất cho đồ thị với: 1/ Kruskal ; 2/PRIM** 9](#_Toc135205632)

[**1)** **Xây dựng cây khung nhỏ nhất cho đồ thị với thuật toán Kruskal** 9](#_Toc135205633)

[**2)** **Xây dựng cây khung nhỏ nhất cho đồ thị với thuật toán PRIM** 10](#_Toc135205634)

[**VI.** **Tìm đường đi ngắn nhất trên đồ thị với : 1/ Dijkstra; 2/ Bellman-Ford ; 3/ Floyd** 11](#_Toc135205635)

[**1)** **Tìm đường đi ngắn nhất trên đồ thị với Dijkstra.** 12](#_Toc135205636)

[**2)** **Tìm đường đi ngắn nhất trên đồ thị với bellman-Ford.** 13](#_Toc135205637)

[**3)** **Tìm đường đi ngắn nhất trên đồ thị với Floyd.** 14](#_Toc135205638)

**BÀI TẬP TỔNG HỢP GRAPH**

1. **Duyệt đồ thị với thuật toán DFS, BFS theo 2 cách:**

**a/ Dùng đệ qui ;**

**b/ Không dùng đệ qui.**

1. **Thuật toán DFS.**

Tư tưởng cơ bản của thuật toán tìm kiếm theo chiều sâu là bắt đầu tại một đỉnh v0 nào đó, chọn một đỉnh u bất kỳ kề với v0 và lấy nó làm đỉnh duyệt tiếp theo. Cách duyệt tiếp theo được thực hiện tương tự như đối với đỉnh v0 với đỉnh bắt đầu là u.

Để kiểm tra việc duyệt mỗi đỉnh đúng một lần, chúng ta sử dụng một mảng chuaxet[] gồm n phần tử (tương ứng với n đỉnh), nếu đỉnh thứ u đã được duyệt, phần tử tương ứng trong mảng chuaxet[u] có giá trị FALSE. Ngược lại, nếu đỉnh chưa được duyệt, phần tử tương ứng trong mảng có giá trị TRUE.

1. **Dùng đệ qui.**

**Biểu diễn thuật toán DFS(u):**

**Thuật toán DFS (u):** //u là đỉnh bắt đầu duyệt

**Begin**

**<thăm đỉnh u>;**//duyệt đỉnh u

**chuaxet[u] := FALSE;**//xác nhận đỉnh u đã duyệt

**for each v ∈ke(u) do** //lấy mỗi đỉnh v∈Ke(u).

**if (chuaxet[v] ) then** //nếu đỉnh v chưa duyệt

**DFS(v);** //duyệt theo chiều sâu bắt đầu từ đỉnh v

**EndIf;**

**EndFor;**

**End**

**Link code full:**

1. **Không dùng đệ qui.**

**Biểu diễn thuật toán DFS(u):**

Thuật toán DFS(u):

**Begin**

**Bước 1 (Khởi tạo):**

stack = ∅; //Khởi tạo stack là ∅

Push(stack, u); //Đưa đỉnh u vào ngăn xếp

<Thăm đỉnh u>; //Duyệt đỉnh u

chuaxet[u] = False; //Xác nhận đỉnh u đã duyệt

**Bước 2 (Lặp) :**

while ( stack ≠ ∅ ) do

s = Pop(stack); //Loại đỉnh ở đầu ngăn xếp

for each t∈ Ke(s) do //Lấy mỗi đỉnh t∈Ke(s)

if ( chuaxet[t] ) then //Nếu t đúng là chưa duyệt

<Thăm đỉnh t>; // Duyệt đỉnh t

chuaxet[t] = False; // Xác nhận đỉnh t đã xét

Push(stack, s);//Đưa s vào stack

Push(stack, t); //Đưa t vào stack

break; //Chỉ lấy một đỉnh t

EndIf;

EndFor;

EndWhile;

**Bước 3 (Trả lại kết quả):**

Return(<Tập đỉnh đã duyệt>);

**End.**

**Link code full:**

1. **Thuật toán BFS.**
2. **Dùng đệ qui.**

* Không có lời giải cho bài toán này của thuật toán BFS.

1. **Không dùng đệ qui.**

Trong thủ tục này, đỉnh được nạp vào hàng đợi đầu tiên là u, các đỉnh kề với u là ( v1, v2, . . ., vk) được nạp vào hàng đợi nếu như nó chưa được xét đến. Quá trình duyệt tiếp theo được bắt đầu từ các đỉnh còn có mặt trong hàng đợi.

Để ghi nhận trạng thái duyệt các đỉnh của đồ thị, ta cũng vẫn sử dụng mảng chuaxet[] gồm n phần tử thiết lập giá trị ban đầu là TRUE. Nếu đỉnh u của đồ thị đã được duyệt, giá trị chuaxet[u] sẽ nhận giá trị FALSE. Thuật toán dừng khi hàng đợi rỗng. Hình 5.7. dưới đây mô tả chi tiết thuật toán BFS(u).

**Thuật toán BFS(u):**

**Bước 1(Khởi tạo):**

Queue = ∅; //tạo lập hàng đợi rỗng

Push(Queue,u); //đưa u vào hàng đợi

chuaxet[u] = False; //ghi nhận đỉnh u đã xét

**Bước 2 (Lặp):**

while (Queue ≠ ∅ ) do //lặp đến khi hàng đợi rỗng

s = Pop(Queue); //đưa s ra khỏi hàng đợi

<Thăm đỉnh s>;

for each t∈Ke(s) do //duyệt trên danh sách ke(s)

if ( chuaxet[t] ) then //nếu t chưa xét

Push(Queue, t); //đưa t vào hàng đợi

chuaxet[t] = False; //ghi nhận t đã xét

EndIf ;

EndFor ;

EndWhile ;

**Bước 3 (Trả lại kết quả) :**

Return(<Tập đỉnh được duyệt>) ;

**End**

**Link code full:**

1. **So sánh và nhận định trường hợp áp dụng cho riêng từng cách.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| S. No. | Parameters | BFS | DFS |  |  |  |
| 1. | Stands for | BFS stands for Breadth First Search. | DFS stands for Depth First Search. |  |  |  |
| 2. | Data Structure | BFS(Breadth First Search) Sử dụng Queue data structure để tìm ra con đường ngắn nhất. | DFS(Depth First Search) Sử dụng Stack data structure. |  |  |  |
| 3. | Định nghĩa | BFS là một cách tiếp cận trong đó chúng ta đi qua tất cả các nút trên cùng một cấp độ trước khi chuyển sang cấp độ tiếp theo. | DFS cũng là một cách tiếp cận đi ngang trong đó việc đi qua bắt đầu từ nút gốc và tiến hành qua các nút càng xa càng tốt cho đến khi chúng ta đến nút mà không có nút lân cận nào chưa được truy cập. |  |  |  |
| 4. | Technique | BFS có thể được sử dụng để tìm một đường dẫn ngắn nhất nguồn duy nhất trong đồ thị không trọng số bởi vì, trong BFS, chúng ta đạt đến một đỉnh có số cạnh tối thiểu từ đỉnh nguồn. | Trong DFS, chúng ta có thể đi qua nhiều cạnh hơn để đến đỉnh đích từ một nguồn. |  |  |  |
| 6. | Approach used | Nó hoạt động trên khái niệm FIFO (First In First Out). | Nó hoạt động trên khái niệm LIFO (Last In First Out). |  |  |  |
| 7. | Thích hợp cho | BFS phù hợp hơn để tìm kiếm các đỉnh gần nguồn nhất định. | DFS phù hợp hơn khi có các giải pháp xa nguồn. |  |  |  |
| 8. | Suitability for Decision-Trees | BFS Xem xét tất cả hàng xóm trước và do đó không phù hợp với cây ra quyết định được sử dụng trong các trò chơi hoặc câu đố. | DFS phù hợp hơn cho các vấn đề trò chơi hoặc câu đố. Chúng tôi đưa ra quyết định, và sau đó khám phá tất cả các con đường thông qua quyết định này. Và nếu quyết định này dẫn đến tình huống chiến thắng, chúng tôi dừng lại. |  |  |  |
| 9. | Time Complexity | Độ phức tạp thời gian của BFS là O (V + E) khi Danh sách kề được sử dụng  và O (V ^ 2) khi Ma trận kề được sử dụng, trong đó V là viết tắt của các đỉnh và E là viết tắt của các cạnh. | Độ phức tạp thời gian của DFS cũng là O (V + E) khi sử dụng Danh sách kề và O (V ^ 2) khi Ma trận kề được sử dụng, trong đó V là viết tắt của các đỉnh và E là viết tắt của các cạnh. |  |  |  |
| 10. | Visiting of Siblings/ Children | Các nút cùng cấp sẽ được thăm trước các nút con | Các nút con sẽ được thăm trước |  |  |  |
| 11. | Loại bỏ các nút đi qua | Các nút được đi qua nhiều lần sẽ bị xóa khỏi hàng đợi. | Các nút đã truy cập được thêm vào ngăn xếp và sau đó bị xóa khi không còn nút nào để truy cập. |  |  |  |
| 12. | Backtracking | Trong BFS không có khái niệm backtracking. | Thuật toán DFS là một thuật toán đệ quy sử dụng ý tưởng backtracking |  |  |  |
| 13. | Ứng dụng | BFS được sử dụng trong các ứng dụng khác nhau như đồ thị hai bên, đường đi ngắn nhất, v.v. | DFS được sử dụng trong các ứng dụng khác nhau như acyclic graphs and topological order etc. |  |  |  |
| 14. | Memory | BFS yêu cầu nhiều bộ nhớ hơn. | DFS yêu cầu ít bộ nhớ hơn. |  |  |  |
| 15. | Optimality | BFS là tối ưu để tìm con đường ngắn nhất. | DFS không phải là tối ưu để tìm con đường ngắn nhất. |  |  |  |
| 16. | Space complexity | Trong BFS, độ phức tạp của không gian quan trọng hơn so với độ phức tạp của thời gian. | DFS có độ phức tạp không gian ít hơn vì tại một thời điểm, nó chỉ cần lưu trữ một đường dẫn duy nhất từ gốc đến nút lá. |  |  |  |
| 17. | Speed | BFS chậm so với DFS. | DFS nhanh so với BFS. |  |  |  |
| 18, | Tapping in loops | Trong BFS, không có vấn đề bẫy vào các vòng lặp hữu hạn. | Trong DFS, chúng ta có thể bị mắc kẹt trong các vòng lặp vô hạn. |  |  |  |
| 19. | When to use? | Khi mục tiêu gần với nguồn, BFS hoạt động tốt hơn. | Khi mục tiêu ở xa nguồn, DFS được ưu tiên hơn. |  |  |  |

1. **Kiểm tra 1 đồ thị vô hướng có thỏa mãn là đồ thị Euler hay nửa Euler**

**Định lý:**

* Đồ thị vô hướng liên thông G = là Euler khi và chỉ khi tất cả các đỉnh của V đều có bậc chẵn.
* Đồ thị vô hướng liên thông G = là nửa Euler nhưng không là Euler khi và chỉ khi G có đúng hai đỉnh bậc lẻ. Đường đi Euler xuất phát tại một đỉnh bậc lẻ và kết thúc tại đỉnh bậc lẻ còn lại.

**Biểu diễn thuật toán:**

**Thuật toán Check\_Euler(G=<V, E>):**

**begin**

**for each u∈V do** //duyệt trên tập các đỉnh trong V

**if ( deg(u) mode 2 )** //nếu bậc của đỉnh là lẻ

**return false;** //trả lại giá trị false

**endif;**

**endfor;**

**return true;**//nếu tất cả các đỉnh đều có bậc chẵn

**end.**

**Link code full:**

1. **Kiểm tra 1 đồ thị có hướng có thỏa mãn là đồ thị Euler hay nửa Euler**

**Định lý:**

* Đồ thị có hướng liên thông yếu G = là Euler khi và chỉ khi mọi đỉnh của G đều có bán bậc ra bằng bán đỉnh bậc vào của đỉnh.
* Đồ thị có hướng liên thông yếu G = là nửa Euler nhưng không là Euler khi và chỉ khi tồn tại đúng hai đỉnh u, v sao cho bán bậc ra của u trừ bán đỉnh bậc vào của u bằng bán đỉnh bậc vào của v trừ bán đỉnh bậc ra của v và bằng 1. Các đỉnh khác u, v còn lại có bán bậc ra bằng bán bậc vào. Đường đi Euler trên đồ thị sẽ xuất phát tại u và kết thúc tại v.

**Biểu diễn thuật toán:**

**Thuật toán Check\_Euler(G=<V, E>):**

**begin**

**for each u∈V do** //duyệt trên tập các đỉnh trong V

**if ( deg+ (u) !=deg-(u))** //nếu bán bậc ra khác bán bậc vào của u **return false;** //trả lại giá trị false

**endif;**

**endfor;**

**return true;**//nếu tất cả các đỉnh đều có bậc chẵn

**end.**

**Link code full:**

1. **Xây dựng cây khung đồ thị với : 1/DFS, 2/BFS**

**Định nghĩa 1**. Ta gọi cây là đồ thị vô hướng liên thông không có chu trình. Đồ thị không liên thông được gọi là rừng. Như vậy, rừng là đồ thị mà mỗi thành phần liên thông của nó là một cây.

**Định lý 1.** Giả sử T= là đồ thị vô hướng n đỉnh. Khi đó những khẳng định sau là tương đương

a) T là một cây.

b) T không có chu trình và có n-1 cạnh.

c) T liên thông và có đúng n-1 cạnh.

d) T liên thông và mỗi cạnh của nó đều là cầu.

e) Giữa hai đỉnh bất kỳ của T được nối với nhau bởi đúng một đường đi đơn.

f) T không chứa chu trình nhưng hễ cứ thêm vào nó một cạnh ta thu được đúng một chu trình.

**Định nghĩa 2.** Cho G = là đồ thị vô hướng liên thông. Ta gọi đồ thị con H= là một cây khung của G nếu H là một cây và T⊆E.

1. **Xây dựng cây khung đồ thị bằng thuật toán DFS**

Khi ta thực hiện thủ tục tìm kiếm theo chiều sâu bắt đầu tại đỉnh u∈V, ta nhận được tập đỉnh v∈V có cùng thành phần liên thông với u. Nếu DFS(u) = V thì ta kết luận đồ thị liên thông và phép duyệt DFS(u) đi qua đúng n-1 cạnh. Nếu Nếu DFS(u) ≠ V thì ta kết luận đồ thị không liên thông. Chính vì vậy, ta có thể sử dụng phép duyệt DFS(u) để xây dựng cây khung của đồ thị. Trong mỗi bước của thuật toán DFS, xuất phát tại đỉnh u ta sẽ thăm được đỉnh v và ta kết nạp cạnh (u,v) vào tập cạnh của cây khung. Các bước tiếp theo được tiến hành tại đỉnh v cho đến khi kết thúc thuật toán DFS.

**Biểu diễn thuật toán:**

**Thuật toán Tree-DFS(u):**

**Begin**

**Bước 1 (Khởi tạo):**

T = ∅; //tập cạnh cây khung ban đầu.

stack = ∅; //thiết lập stack rỗng;

Push(stack, u); //đưa u vào stack;

chuaxet[u] = False;//bật trạng thái đã xét của đỉnh u

**Bước 2 (Lặp):**

while (stack≠∅ ) do { //lặp cho đến khi stack rỗng

s = Pop(stack); //lấy s ra khỏi stack

for each t∈Ke(s) do { //lặp trên danh sách Ke(s)

if (chuaxet[t] ) then { //nếu đỉnh t chuaxet

Push(stack, s);// đưa s vào stack trước

Push(stack, s);// đưa t vào stack sau

T = T∪(s,t); //kết nạp (s,t) vào cây khung chuaxet[t] = False; //ghi nhận t đã xét

break ;//chỉ lấy đỉnh đầu tiên endif ;

endfor ;

endwwhile ;

**Bước 3 (Trả lại kết quả) :**

if (| T | < n-1 ) <Đồ thị không liên thông>;

else <Ghi nhận tập cạnh T của cây khung>;

**end.**

**Link code full:**

1. **Xây dựng cây khung đồ thị bằng thuật toán BFS**

Để tìm một cây khung trên đồ thị vô hướng liên thông ta có thể sử dụng kỹ thuật tìm kiếm theo chiều rộng. Giả sử ta cần xây dựng một cây bao trùm xuất phát tại đỉnh u nào đó. Trong cả hai trường hợp, mỗi khi ta đến được đỉnh v tức (chuaxet[v] = False) từ đỉnh u thì cạnh (u,v) được kết nạp vào cây khung.

**Biểu diễn thuật toán:**

**Thuật toán Tree-BFS(u):**

**Begin**

**Bước 1 (Khởi tạo):**

T = ∅; //tập cạnh cây khung ban đầu.

Queue = ∅; //thiết lập hàng đợi ban đầu;

Push(Queue, u); //đưa u vào hàng đợi;

chuaxet[u] = False;//bật trạng thái đã xét của đỉnh u

**Bước 2 (Lặp):**

while (Queue≠∅ ) do { //lặp cho đến khi hàng đợi rỗng

s = Pop(Queue); //lấy s ra khỏi hàng đợi

for each t∈Ke(s) do { //lặp trên danh sách Ke(s)

if (chuaxet[t] ) then { //nếu đỉnh t chuaxet

Push(Queue, t);// đưa t vào hàng đợi

T = T∪(s,t); //kết nạp (s,t) vào cây khung

chuaxet[t] = False; //ghi nhận t đã xét

endif ;

endfor ;

endwhile ;

**Bước 3 (Trả lại kết quả) :**

if (| T | < n-1 ) <Đồ thị không liên thông>;

else <Ghi nhận tập cạnh T của cây khung>;

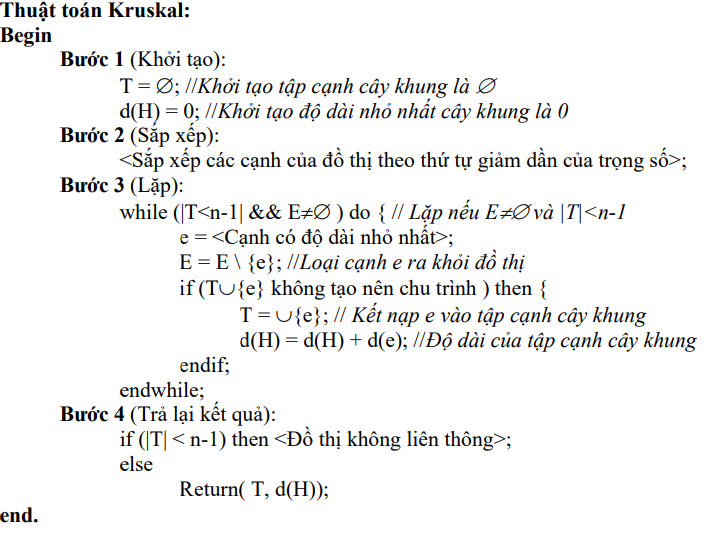
**end.**

**Link code full:**

1. **Xây dựng cây khung nhỏ nhất cho đồ thị với: 1/ Kruskal ; 2/PRIM**
2. **Xây dựng cây khung nhỏ nhất cho đồ thị với thuật toán Kruskal**

Bài toán tìm cây khung nhỏ nhất là một trong những bài toán tối ưu trên đồ thị có ứng dụng trong nhiều lĩnh vực khác nhau của thực tế. Bài toán được phát biểu như dưới sau. Cho G= là đồ thị vô hướng liên thông với tập đỉnh V = {1, 2, . . ., n } và tập cạnh E gồm m cạnh. Mỗi cạnh e của đồ thị được gán với một số không âm c(e) được gọi là độ dài cạnh. Giả sử H= là một cây khung của đồ thị G. Ta gọi độ dài c(H) của cây khung H là tổng độ dài các cạnh: ∑ ∈ = e T c(H) c(e) . Bài toán được đặt ra là, trong số các cây khung của đồ thị hãy tìm cây khung có độ dài nhỏ nhất của đồ thị.

**Biểu diễn thuật toán.**

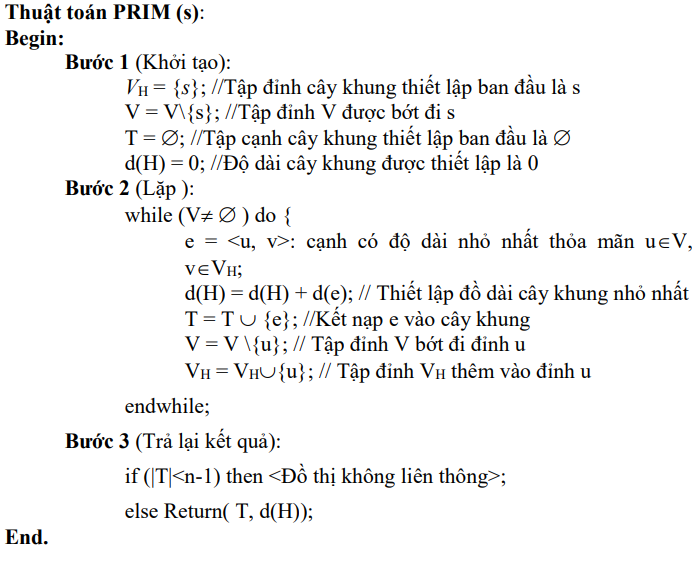


**Link code full:**

1. **Xây dựng cây khung nhỏ nhất cho đồ thị với thuật toán PRIM**

Thuật toán Kruskal làm việc kém hiệu quả đối với những đồ thị có số cạnh khoảng m=n(n-1)/2. Trong những tình huống như vậy, thuật toán Prim tỏ ra hiệu quả hơn. Thuật toán Prim còn được mang tên là người láng giềng gần nhất. Trong thuật toán này, bắt đầu tại một đỉnh tuỳ ý s của đồ thị, nối s với đỉnh y sao cho trọng số cạnh graph[s, y] là nhỏ nhất. Tiếp theo, từ đỉnh s hoặc y tìm cạnh có độ dài nhỏ nhất, điều này dẫn đến đỉnh thứ ba z và ta thu được cây bộ phận gồm 3 đỉnh 2 cạnh. Quá trình được tiếp tục cho tới khi ta nhận được cây gồm n-1 cạnh, đó chính là cây bao trùm nhỏ nhất cần tìm

**Biểu diễn thuật toán:**

****

**Link code full:**

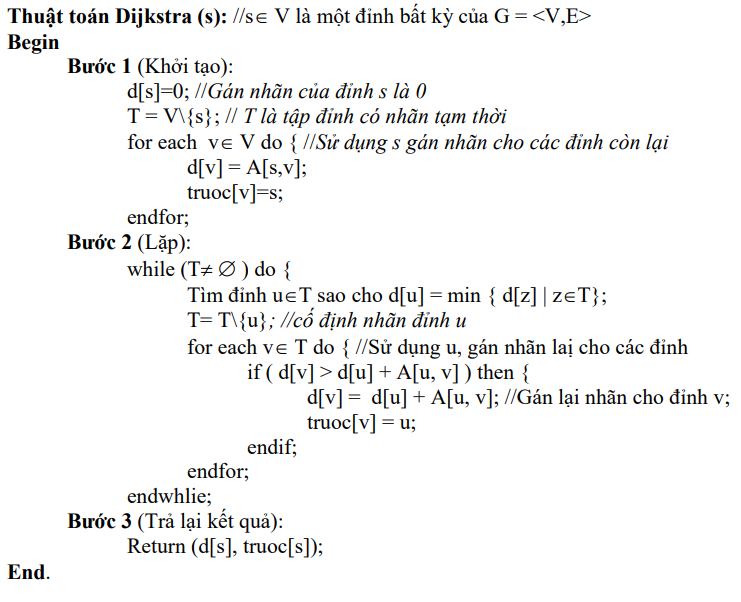
1. **Tìm đường đi ngắn nhất trên đồ thị với : 1/ Dijkstra; 2/ Bellman-Ford ; 3/ Floyd**

Bài toán tìm đường đi ngắn nhất trên đồ thị dưới dạng tổng quát có thể được phát biểu dưới dạng sau: tìm đường đi ngắn nhất từ một đỉnh xuất phát s∈V (đỉnh nguồn) đến đỉnh cuối t∈V (đỉnh đích). Đường đi như vậy được gọi là đường đi ngắn nhất từ s đến t, độ dài của đường đi d(s,t) được gọi là khoảng cách ngắn nhất từ s đến t (trong trường hợp tổng quát d(s,t) có thể âm). Nếu như không tồn tại đường đi từ s đến t thì độ dài đường đi d(s,t)=∞.

1. **Tìm đường đi ngắn nhất trên đồ thị với Dijkstra.**

Thuật toán tìm đường đi ngắn nhất từ đỉnh s đến các đỉnh còn lại được Dijkstra đề nghị áp dụng cho trường hợp đồ thị có hướng với trọng số không âm. Thuật toán được thực hiện trên cơ sở gán tạm thời cho các đỉnh. Nhãn của mỗi đỉnh cho biết cận trên của độ dài đường đi ngắn nhất tới đỉnh đó. Các nhãn này sẽ được biến đổi (tính lại) nhờ một thủ tục lặp, mà ở mỗi bước lặp một số đỉnh sẽ có nhãn không thay đổi, nhãn đó chính là độ dài đường đi ngắn nhất từ s đến đỉnh đó

**Biểu diễn thuật toán.**

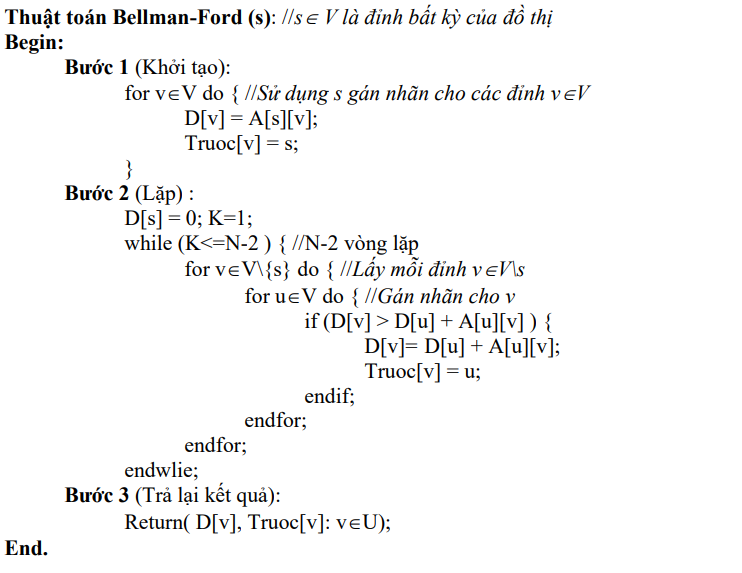
****

**Link code full:**

1. **Tìm đường đi ngắn nhất trên đồ thị với bellman-Ford.**

Thuật toán Bellman-Ford dùng để tìm đường đi ngắn nhất trên đồ thị không có chu trình âm. Do vậy, trong khi thực hiện thuật toán Bellman-Ford ta cần kiểm tra đồ thị có chu trình âm hay không. Trong trường hợp đồ thị có chu trình âm, bài toán sẽ không có lời giải. Thuật toán được thực hiện theo k = n - 2 vòng lặp trên tập đỉnh hoặc tập cạnh tùy thuộc vào dạng biểu diễn của đồ thị. Nếu đồ thị được biểu diễn dưới dạng ma trận kề, độ phức tạp thuật toán là O(V^3 ), với V là số đỉnh của đồ thị. Trong trường hợp đồ thị được biểu diễn dưới dạng danh sách cạnh, độ phức tạp thuật toán là O(V.E), với V là số đỉnh của đồ thị, E là số cạnh của đồ thị.

**Biểu diễn thuật toán.**

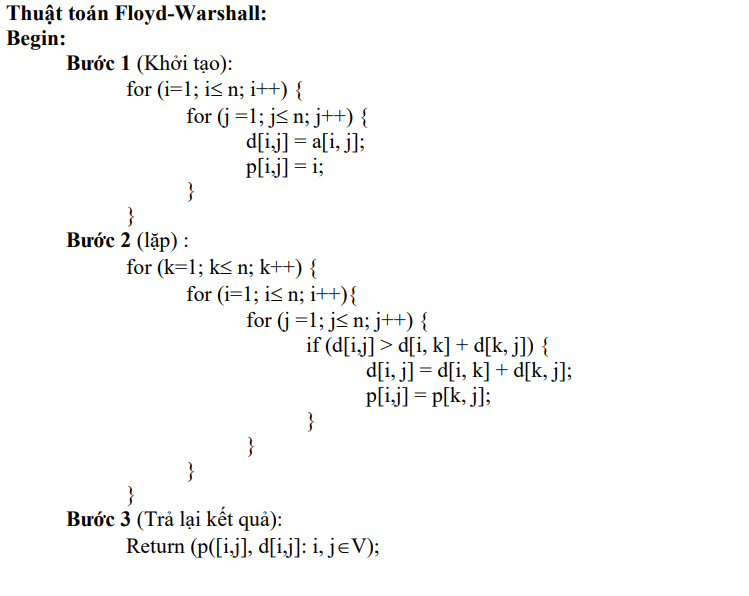
****

**Link code full:**

1. **Tìm đường đi ngắn nhất trên đồ thị với Floyd.**

Để tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh của đồ thị, chúng ta có thể sử dụng V lần thuật toán Ford\_Bellman hoặc Dijkstra (trong trường hợp trọng số không âm). Tuy nhiên, trong cả hai thuật toán được sử dụng đều có độ phức tạp tính toán lớn (chí ít là O(V^3 )). Trong trường hợp tổng quát, người ta thường dùng thuật toán Floyd.

**Biểu diễn thuật toán:**

****

**Link code full:**