Sau đây là **thuật toán DFS dựa trên Kosaraju dựa trên hai đường dẫn DFS** của đồ thị:   
**1)** Khởi tạo tất cả các đỉnh không được truy cập.

**2)** Thực hiện traversal DFS của đồ thị bắt đầu từ bất kỳ đỉnh nào tùy ý v. Nếu traversal DFS không truy cập vào tất cả các đỉnh, sau đó trả về false.

**3)** Ngược lại tất cả các cung (hoặc tìm transpose hoặc đảo ngược của đồ thị)

**4)** Đánh dấu tất cả các đỉnh là không được truy cập trong đồ thị đảo ngược.

**5)** Thực hiện traversal DFS của đồ thị đảo ngược bắt đầu từ cùng một vertex v (Tương tự như bước 2). Nếu traversal DFS không truy cập vào tất cả các đỉnh, sau đó trả về false. Nếu không trở lại đúng.

Ý tưởng là, nếu mỗi nút có thể đạt được từ một đỉnh v, và mỗi nút có thể đạt v, sau đó đồ thị được kết nối mạnh mẽ. Trong bước 2, chúng ta kiểm tra xem tất cả các đỉnh có thể truy cập từ v. Ở bước 4, chúng ta kiểm tra xem tất cả các đỉnh có thể đạt v (Trong đồ thị đảo ngược, nếu tất cả các đỉnh đều có thể đến được từ v, thì tất cả các đỉnh có thể đạt v trong đồ thị ban đầu).

Sau đây là việc thực hiện các thuật toán trên.

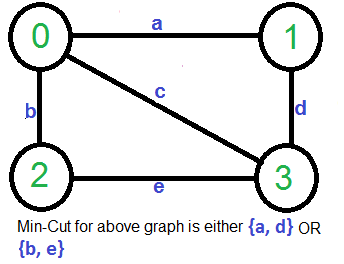
* C ++
* Java
* Python

|  |
| --- |
| // C++ program to check if a given directed graph is strongly  // connected or not  #include <iostream>  #include <list>  #include <stack>  using namespace std;    class Graph  {      int V;    // No. of vertices      list<int> \*adj;    // An array of adjacency lists        // A recursive function to print DFS starting from v      void DFSUtil(int v, bool visited[]);  public:      // Constructor and Destructor      Graph(int V) { this->V = V;  adj = new list<int>[V];}      ~Graph() { delete [] adj; }        // Method to add an edge      void addEdge(int v, int w);        // The main function that returns true if the graph is strongly      // connected, otherwise false      bool isSC();        // Function that returns reverse (or transpose) of this graph      Graph getTranspose();  };    // A recursive function to print DFS starting from v  void Graph::DFSUtil(int v, bool visited[])  {      // Mark the current node as visited and print it      visited[v] = true;        // Recur for all the vertices adjacent to this vertex      list<int>::iterator i;      for (i = adj[v].begin(); i != adj[v].end(); ++i)          if (!visited[\*i])              DFSUtil(\*i, visited);  }    // Function that returns reverse (or transpose) of this graph  Graph Graph::getTranspose()  {      Graph g(V);      for (int v = 0; v < V; v++)      {          // Recur for all the vertices adjacent to this vertex          list<int>::iterator i;          for(i = adj[v].begin(); i != adj[v].end(); ++i)          {              g.adj[\*i].push\_back(v);          }      }      return g;  }    void Graph::addEdge(int v, int w)  {      adj[v].push\_back(w); // Add w to v’s list.  }    // The main function that returns true if graph is strongly connected  bool Graph::isSC()  {      // St1p 1: Mark all the vertices as not visited (For first DFS)      bool visited[V];      for (int i = 0; i < V; i++)          visited[i] = false;        // Step 2: Do DFS traversal starting from first vertex.      DFSUtil(0, visited);         // If DFS traversal doesn’t visit all vertices, then return false.      for (int i = 0; i < V; i++)          if (visited[i] == false)               return false;        // Step 3: Create a reversed graph      Graph gr = getTranspose();        // Step 4: Mark all the vertices as not visited (For second DFS)      for(int i = 0; i < V; i++)          visited[i] = false;        // Step 5: Do DFS for reversed graph starting from first vertex.      // Staring Vertex must be same starting point of first DFS      gr.DFSUtil(0, visited);        // If all vertices are not visited in second DFS, then      // return false      for (int i = 0; i < V; i++)          if (visited[i] == false)               return false;        return true;  }    // Driver program to test above functions  int main()  {      // Create graphs given in the above diagrams      Graph g1(5);      g1.addEdge(0, 1);      g1.addEdge(1, 2);      g1.addEdge(2, 3);      g1.addEdge(3, 0);      g1.addEdge(2, 4);      g1.addEdge(4, 2);      g1.isSC()? cout << "Yes\n" : cout << "No\n";        Graph g2(4);      g2.addEdge(0, 1);      g2.addEdge(1, 2);      g2.addEdge(2, 3);      g2.isSC()? cout << "Yes\n" : cout << "No\n";        return 0;  } |

**Thời gian phức tạp:** Thời gian phức tạp của việc thực hiện trên là sane như [sâu Tìm kiếm đầu tiên](http://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)đó là O (V + E) nếu đồ thị được đại diện bằng cách sử dụng đại diện danh sách kề.

Thuật toán Karger cho phép cắt nhỏ nhất | Tập 1 (Giới thiệu và Thực hiện)

Với đồ thị vô hướng và không trọng tải, tìm cắt nhỏ nhất (số cạnh nhỏ nhất ngắt kết nối đồ thị thành hai thành phần).  
Đồ thị đầu vào có thể có cạnh song song.

Ví dụ xem xét ví dụ sau, cắt nhỏ nhất có 2 cạnh.  
[](http://www.geeksforgeeks.org/wp-content/uploads/Kargerfirst.png)

Một giải pháp đơn giản sử dụng [thuật toán cắt s-t](http://www.geeksforgeeks.org/minimum-cut-in-a-directed-graph/)[dựa trên dòng chảy Max-Flow](http://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/) để tìm cắt giảm tối thiểu. Hãy xem xét mỗi cặp đỉnh như là nguồn 'và chìm' t ', và gọi thuật toán cắt tối thiểu để tìm st cắt. Trả lại tối thiểu tất cả các vết cắt. Độ phức tạp thời gian tốt nhất có thể của thuật toán này là O (V 5 ) cho một đồ thị. [Làm sao? có tổng thể V 2 cặp và thuật toán cắt st cho một cặp mất O (V \* E) Thời gian và E = O (V 2 )].

Dưới đây là thuật toán đơn giản của Karger cho mục đích này. Dưới thuật toán của Karger có thể được thực hiện trong thời gian O (E) = O (V 2 ).

1) Khởi tạo CG đồ thị theo hợp đồng như bản sao của biểu đồ gốc

2) Trong khi có hơn 2 đỉnh.

a) Chọn một cạnh ngẫu nhiên (u, v) trong đồ thị hợp đồng.

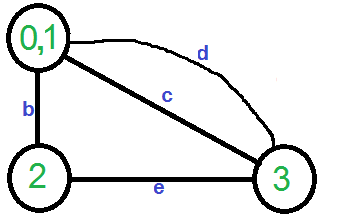
b) Hợp nhất (hoặc hợp đồng) u và v thành một đỉnh duy nhất (cập nhật

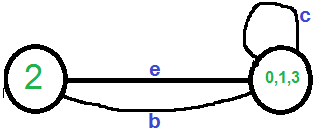
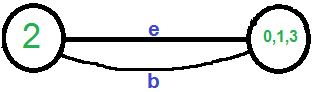
đồ thị hợp đồng).

c) Xoá vòng lặp tự

3) Quay trở lại đại diện bởi hai đỉnh.

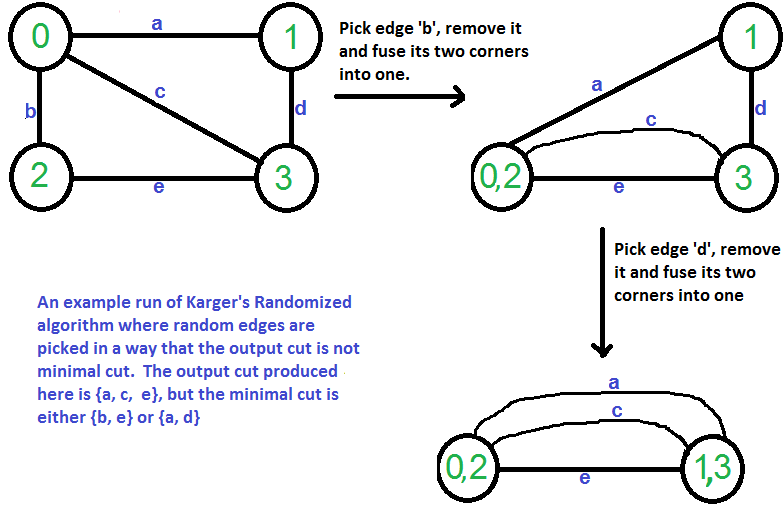
Hãy để chúng tôi hiểu thuật toán trên theo ví dụ được đưa ra.

Cho phép đỉnh đầu ngẫu nhiên được chọn ngẫu nhiên là ' **a** ' kết nối các đỉnh 0 và 1. Chúng ta loại bỏ cạnh này và ký hiệu đồ thị (kết hợp các đỉnh 0 và 1). Chúng ta có được đồ thị sau đây.  
[](http://www.geeksforgeeks.org/wp-content/uploads/Karger21.png)

Hãy để cạnh chọn ngẫu nhiên tiếp theo là 'd'. Chúng tôi loại bỏ cạnh này và kết hợp các đỉnh (0,1) và 3. Chúng ta cần loại bỏ các vòng lặp tự trong đồ thị. Vì vậy, chúng tôi loại bỏ cạnh 'c'  
[](http://www.geeksforgeeks.org/wp-content/uploads/Karger3.png)  
  
[](http://www.geeksforgeeks.org/wp-content/uploads/Karger4.png)

Bây giờ đồ thị có hai đỉnh, vì vậy chúng ta dừng lại. Số cạnh của đồ thị kết quả là sự cắt giảm được tạo ra theo thuật toán của Karger.

***Thuật toán của Karger là một thuật toán***[***Monte Carlo***](http://en.wikipedia.org/wiki/Monte_Carlo_algorithm)***và được cắt bởi nó có thể không phải là tối thiểu.***Ví dụ, biểu đồ sau đây cho thấy rằng một trật tự khác nhau của việc chọn cạnh ngẫu nhiên tạo ra một min-cut của kích thước 3.

[](http://www.geeksforgeeks.org/wp-content/uploads/Karger11.png)

Dưới đây là C + + thực hiện các thuật toán trên. Biểu đồ đầu vào được biểu diễn dưới dạng tập hợp các cạnh và [cấu trúc dữ liệu union-find](http://www.geeksforgeeks.org/union-find-algorithm-set-2-union-by-rank/) được sử dụng để theo dõi các thành phần.

|  |
| --- |
| // Karger's algorithm to find Minimum Cut in an  // undirected, unweighted and connected graph.  #include <stdio.h>  #include <stdlib.h>  #include <time.h>    // a structure to represent a unweighted edge in graph  struct Edge  {      int src, dest;  };    // a structure to represent a connected, undirected  // and unweighted graph as a collection of edges.  struct Graph  {      // V-> Number of vertices, E-> Number of edges      int V, E;        // graph is represented as an array of edges.      // Since the graph is undirected, the edge      // from src to dest is also edge from dest      // to src. Both are counted as 1 edge here.      Edge\* edge;  };    // A structure to represent a subset for union-find  struct subset  {      int parent;      int rank;  };    // Function prototypes for union-find (These functions are defined  // after kargerMinCut() )  int find(struct subset subsets[], int i);  void Union(struct subset subsets[], int x, int y);    // A very basic implementation of Karger's randomized  // algorithm for finding the minimum cut. Please note  // that Karger's algorithm is a Monte Carlo Randomized algo  // and the cut returned by the algorithm may not be  // minimum always  int kargerMinCut(struct Graph\* graph)  {      // Get data of given graph      int V = graph->V, E = graph->E;      Edge \*edge = graph->edge;        // Allocate memory for creating V subsets.      struct subset \*subsets = new subset[V];        // Create V subsets with single elements      for (int v = 0; v < V; ++v)      {          subsets[v].parent = v;          subsets[v].rank = 0;      }        // Initially there are V vertices in      // contracted graph      int vertices = V;        // Keep contracting vertices until there are      // 2 vertices.      while (vertices > 2)      {         // Pick a random edge         int i = rand() % E;           // Find vertices (or sets) of two corners         // of current edge         int subset1 = find(subsets, edge[i].src);         int subset2 = find(subsets, edge[i].dest);           // If two corners belong to same subset,         // then no point considering this edge         if (subset1 == subset2)           continue;           // Else contract the edge (or combine the         // corners of edge into one vertex)         else         {            printf("Contracting edge %d-%d\n",                   edge[i].src, edge[i].dest);            vertices--;            Union(subsets, subset1, subset2);         }      }        // Now we have two vertices (or subsets) left in      // the contracted graph, so count the edges between      // two components and return the count.      int cutedges = 0;      for (int i=0; i<E; i++)      {          int subset1 = find(subsets, edge[i].src);          int subset2 = find(subsets, edge[i].dest);          if (subset1 != subset2)            cutedges++;      }        return cutedges;  }    // A utility function to find set of an element i  // (uses path compression technique)  int find(struct subset subsets[], int i)  {      // find root and make root as parent of i      // (path compression)      if (subsets[i].parent != i)        subsets[i].parent =               find(subsets, subsets[i].parent);        return subsets[i].parent;  }    // A function that does union of two sets of x and y  // (uses union by rank)  void Union(struct subset subsets[], int x, int y)  {      int xroot = find(subsets, x);      int yroot = find(subsets, y);        // Attach smaller rank tree under root of high      // rank tree (Union by Rank)      if (subsets[xroot].rank < subsets[yroot].rank)          subsets[xroot].parent = yroot;      else if (subsets[xroot].rank > subsets[yroot].rank)          subsets[yroot].parent = xroot;        // If ranks are same, then make one as root and      // increment its rank by one      else      {          subsets[yroot].parent = xroot;          subsets[xroot].rank++;      }  }    // Creates a graph with V vertices and E edges  struct Graph\* createGraph(int V, int E)  {      Graph\* graph = new Graph;      graph->V = V;      graph->E = E;      graph->edge = new Edge[E];      return graph;  }    // Driver program to test above functions  int main()  {      /\* Let us create following unweighted graph          0------1          | \    |          |   \  |          |     \|          2------3   \*/      int V = 4;  // Number of vertices in graph      int E = 5;  // Number of edges in graph      struct Graph\* graph = createGraph(V, E);        // add edge 0-1      graph->edge[0].src = 0;      graph->edge[0].dest = 1;        // add edge 0-2      graph->edge[1].src = 0;      graph->edge[1].dest = 2;        // add edge 0-3      graph->edge[2].src = 0;      graph->edge[2].dest = 3;        // add edge 1-3      graph->edge[3].src = 1;      graph->edge[3].dest = 3;        // add edge 2-3      graph->edge[4].src = 2;      graph->edge[4].dest = 3;        // Use a different seed value for every run.      srand(time(NULL));        printf("\nCut found by Karger's randomized algo is %d\n",             kargerMinCut(graph));        return 0;  } |