```cpp
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

typedef pair<int, int> ii;      // In this chapter, we will frequently use these
typedef vector<ii> vii;      // three data type shortcuts. They may look cryptic
typedef vector<int> vi;   // but shortcuts are useful in competitive programming

#define DFS_WHITE -1 // normal DFS, do not change this with other values (other
than 0), because we usually use memset with conjunction with DFS_WHITE
#define DFS_BLACK 1

vector<vii> AdjList;

void printThis(char* message) {
  printf("==================================\n");
  printf("%s\n", message);
  printf("==================================\n");
}

vi dfs_num;     // this variable has to be global, we cannot put it in recursion
int numCC;

void dfs(int u) {          // DFS for normal usage: as graph traversal algorithm
  printf(" %d", u);                                    // this vertex is visited
  dfs_num[u] = DFS_BLACK;      // important step: we mark this vertex as visited
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];                      // v is a (neighbor, weight) pair
    if (dfs_num[v.first] == DFS_WHITE)         // important check to avoid cycle
      dfs(v.first);      // recursively visits unvisited neighbors v of vertex u
} }

// note: this is not the version on implicit graph
void floodfill(int u, int color) {
  dfs_num[u] = color;                            // not just a generic DFS_BLACK
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      floodfill(v.first, color);
} }

vi topoSort;             // global vector to store the toposort in reverse order

void dfs2(int u) {    // change function name to differentiate with original dfs
  dfs_num[u] = DFS_BLACK;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      dfs2(v.first);
  }
  topoSort.push_back(u); }                   // that is, this is the only change

#define DFS_GRAY 2              // one more color for graph edges property check
vi dfs_parent;      // to differentiate real back edge versus bidirectional edge

void graphCheck(int u) {               // DFS for checking graph edge properties
  dfs_num[u] = DFS_GRAY;   // color this as DFS_GRAY (temp) instead of DFS_BLACK
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE) {     // Tree Edge, DFS_GRAY to DFS_WHITE
      dfs_parent[v.first] = u;                  // parent of this children is me
      graphCheck(v.first);
    }
    else if (dfs_num[v.first] == DFS_GRAY) {             // DFS_GRAY to DFS_GRAY
      if (v.first == dfs_parent[u])          // to differentiate these two cases
        printf(" Bidirectional (%d, %d) - (%d, %d)\n", u, v.first, v.first, u);
      else  // the most frequent application: check if the given graph is cyclic
        printf(" Back Edge (%d, %d) (Cycle)\n", u, v.first);
```

```
    }
    else if (dfs_num[v.first] == DFS_BLACK)             // DFS_GRAY to DFS_BLACK
      printf(" Forward/Cross Edge (%d, %d)\n", u, v.first);
  }
  dfs_num[u] = DFS_BLACK;     // after recursion, color this as DFS_BLACK (DONE)
}

vi dfs_low;       // additional information for articulation points/bridges/SCCs
vi articulation_vertex;
int dfsNumberCounter, dfsRoot, rootChildren;

void articulationPointAndBridge(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;      // dfs_low[u] <= dfs_num[u]
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE) {                          // a tree edge
      dfs_parent[v.first] = u;
      if (u == dfsRoot) rootChildren++;  // special case, count children of root

      articulationPointAndBridge(v.first);

      if (dfs_low[v.first] >= dfs_num[u])              // for articulation point
        articulation_vertex[u] = true;           // store this information first
      if (dfs_low[v.first] > dfs_num[u])                           // for bridge
        printf(" Edge (%d, %d) is a bridge\n", u, v.first);
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);       // update dfs_low[u]
    }
    else if (v.first != dfs_parent[u])       // a back edge and not direct cycle
      dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);       // update dfs_low[u]
} }

vi S, visited;                                    // additional global variables
int numSCC;

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;      // dfs_low[u] <= dfs_num[u]
  S.push_back(u);           // stores u in a vector based on order of visitation
  visited[u] = 1;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      tarjanSCC(v.first);
    if (visited[v.first])                                // condition for update
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
  }

  if (dfs_low[u] == dfs_num[u]) {         // if this is a root (start) of an SCC
    printf("SCC %d:", ++numSCC);            // this part is done after recursion
    while (1) {
      int v = S.back(); S.pop_back(); visited[v] = 0;
      printf(" %d", v);
      if (u == v) break;
    }
    printf("\n");
} }

int main() {
  int V, total_neighbors, id, weight;

  /*
  // Use the following input:
  // Graph in Figure 4.1
  9
  1 1 0
  3 0 0 2 0 3 0
  2 1 0 3 0
  3 1 0 2 0 4 0
  1 3 0
  0
```

```
2 7 0 8 0
1 6 0
1 6 0

// Example of directed acyclic graph in Figure 4.4 (for toposort)
8
2 1 0 2 0
2 2 0 3 0
2 3 0 5 0
1 4 0
0
0
0
1 6 0

// Example of directed graph with back edges
3
1 1 0
1 2 0
1 0 0

// Left graph in Figure 4.6/4.7/4.8
6
1 1 0
3 0 0 2 0 4 0
1 1 0
1 4 0
3 1 0 3 0 5 0
1 4 0

// Right graph in Figure 4.6/4.7/4.8
6
1 1 0
5 0 0 2 0 3 0 4 0 5 0
1 1 0
1 1 0
2 1 0 5 0
2 1 0 4 0

// Directed graph in Figure 4.9
8
1 1 0
1 3 0
1 1 0
2 2 0 4 0
1 5 0
1 7 0
1 4 0
1 6 0
*/

freopen("in_01.txt", "r", stdin);

scanf("%d", &V);
AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to AdjList
for (int i = 0; i < V; i++) {
  scanf("%d", &total_neighbors);
  for (int j = 0; j < total_neighbors; j++) {
    scanf("%d %d", &id, &weight);
    AdjList[i].push_back(ii(id, weight));
  }
}

printThis("Standard DFS Demo (the input graph must be UNDIRECTED)");
numCC = 0;
dfs_num.assign(V, DFS_WHITE);    // this sets all vertices' state to DFS_WHITE
for (int i = 0; i < V; i++)                       // for each vertex i in [0..V-1]
  if (dfs_num[i] == DFS_WHITE)            // if that vertex is not visited yet
    printf("Component %d:", ++numCC), dfs(i), printf("\n");   // 3 lines here!
```

```
  printf("There are %d connected components\n", numCC);

  printThis("Flood Fill Demo (the input graph must be UNDIRECTED)");
  numCC = 0;
  dfs_num.assign(V, DFS_WHITE);
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == DFS_WHITE)
      floodfill(i, ++numCC);
  for (int i = 0; i < V; i++)
    printf("Vertex %d has color %d\n", i, dfs_num[i]);

  // make sure that the given graph is DAG
  printThis("Topological Sort (the input graph must be DAG)");
  topoSort.clear();
  dfs_num.assign(V, DFS_WHITE);
  for (int i = 0; i < V; i++)            // this part is the same as finding CCs
    if (dfs_num[i] == DFS_WHITE)
      dfs2(i);
  reverse(topoSort.begin(), topoSort.end());                 // reverse topoSort
  for (int i = 0; i < (int)topoSort.size(); i++)       // or you can simply read
    printf(" %d", topoSort[i]);           // the content of `topoSort' backwards
  printf("\n");

  printThis("Graph Edges Property Check");
  numCC = 0;
  dfs_num.assign(V, DFS_WHITE); dfs_parent.assign(V, -1);
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == DFS_WHITE)
      printf("Component %d:\n", ++numCC), graphCheck(i);       // 2 lines in one

  printThis("Articulation Points & Bridges (the input graph must be UNDIRECTED)");
  dfsNumberCounter = 0; dfs_num.assign(V, DFS_WHITE); dfs_low.assign(V, 0);
  dfs_parent.assign(V, -1); articulation_vertex.assign(V, 0);
  printf("Bridges:\n");
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == DFS_WHITE) {
      dfsRoot = i; rootChildren = 0;
      articulationPointAndBridge(i);
      articulation_vertex[dfsRoot] = (rootChildren > 1); }       // special case
  printf("Articulation Points:\n");
  for (int i = 0; i < V; i++)
    if (articulation_vertex[i])
      printf(" Vertex %d\n", i);

  printThis("Strongly Connected Components (the input graph must be DIRECTED)");
  dfs_num.assign(V, DFS_WHITE); dfs_low.assign(V, 0); visited.assign(V, 0);
  dfsNumberCounter = numSCC = 0;
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == DFS_WHITE)
      tarjanSCC(i);

  return 0;
}
```