Congratulations! You passed!

TO PASS 80% or higher

Keep Learning

GRADE
100%

Week 3 Challenge Problem

LATEST SUBMISSION GRADE
100%

1. **Suppose you are given a undirected graph specified as a list of edges. In this challenge problem, we'll use a simplified disjoint sets data structure to count how many "connected components" the graph has, and whether each one contains a cycle or not.**

5 / 5 points

First, some background information: In an undirected graph, two vertices have connectivity if there is any path leading from one to the other using any number of edges. So, a lone vertex by itself, with no edges, is not "connected" to the other parts of the graph. A "connected component" is any subset of the graph vertices where all the vertices have paths to each other, and where that set is maximal, meaning that no reachable vertices are left out of the set. A connected component contains a "cycle" if there are two (or more) distinct paths connecting any two vertices--that means there is a closed loop somewhere.

Example: An edge label like "(0,1)" means an edge between vertex 0 and vertex 1. Suppose we have vertices numbered 0 through 8, and we have these edges:

(0,1), (1,2), (0,2), (3,4), (5,6), (6,7), (7,8)

Try drawing it on a sheet of paper. The three connected components are these sets of vertices:

{0, 1, 2}, {3, 4}, {5, 6, 7, 8}

You'll see the connected components are like islands of vertices. Here, {0, 1, 2} contains a cycle, and the other two connected components do not contain cycles. Also, notice that for a set to be a connected component, it must be maximal, meaning no vertices can be left out--and so {0, 1} is not called a connected component, because the 2 is also reachable there. (Maximal does not mean "maximum". A single, lone vertex is a connected component by itself, because the subset containing only that one vertex is maximal, considering what can be reached from it. So, the sizes of the other connected components elsewhere do not matter.)

In graph theory, it's common to say "n" for the number of vertices and "m" for the number of edges in some graph. For this problem, we'll say we have some undirected graph of some n vertices, which are arbitrarily labeled with indices from 0 through n-1. (This is reasonable because we could otherwise relabel the vertices using a hash table for lookups. Also, we won't assume that subsequent numbers are connected by edges, although that may happen in our unit tests.) Then, we'll initialize a collection of disjoint sets as n singletons (single element sets), one for each vertex; we have a DisjointSets class to represent this collection.

To create sets representing connected components, we can iterate over the graph edges: For each edge (A,B) connecting vertex A to vertex B, we can union the sets that A and B belong to, so the disjoint sets data structure now indicates now that A and B belong to the same set. Our member function for the union operation is called "dsunion" to avoid conflicting with the C++ keyword "union".

At the end of the process of calling dsunion() on every pair of vertices in the edge list, the number of disjoint sets should correspond to the number of connected components in the graph.

The disjoint sets data structure can also detect cycles. As the edges are being processed, if the edge currently being processed connects vertex A and vertex B, and both vertex A and vertex B are already in the same disjoint set, then the edge connecting vertex A and vertex B completes a cycle.

In the source code provided below, you should modify the definition of DisjointSets::dsunion (under TASK 1) and the definition of DisjointSets::count_comps (under TASK 2) according to the hints in the code comments. We'll detect cycles during the union procedure and we can count the number of components after all union operations are completed.

The starter code main() also contains an example graph with expected output. When you're ready to submit, we'll run your code through some randomized unit tests for grading.

```
#include <iostream>

// You are provided this version of a DisjointSets class.
// See below for the tasks to complete.
// (Please note: You may not edit the primary class definition here.)
class DisjointSets {
public:
  // We'll statically allocate space for at most 256 nodes.
  // (We could easily make this extensible by using STL containers
  //  instead of static arrays.)
  static constexpr int MAX_NODES = 256;

  // For a given vertex of index i, leader[i] is -1 if that vertex "leads"
  // the set, and otherwise, leader[i] is the vertex index that refers back
  // to the eventual leader, recursively. (See the function "find_leader".)
  // In this problem we'll interpret sets to represent connected components,
  // once the sets have been unioned as much as possible.
  int leader[MAX_NODES];

  // For a given vertex of index i, has_cycle[i] should be "true" if that
  // vertex is part of a connected component that has a cycle, and otherwise
  // "false". (However, this is only required to be accurate for a current
  // set leader, so that the function query_cycle can return the correct
  // value.)
  bool has_cycle[MAX_NODES];

  // The number of components found.
  int num_components;
```

```
  DisjointSets() {
    // Initialize leaders to -1
    for (int i = 0; i < MAX_NODES; i++) leader[i] = -1;
    // Initialize cycle detection to false
    for (int i = 0; i < MAX_NODES; i++) has_cycle[i] = false;
    // The components will need to be counted.
    num_components = 0;
  }

  // If the leader for vertex i is set to -1, then report vertex i as its
  // own leader. Otherwise, keep looking for the leader recursively.
  int find_leader(int i) {
    if (leader[i] < 0) return i;
    else return find_leader(leader[i]);
  }

  // query_cycle(i) returns true if vertex i is part of a connected component
  // that has a cycle. Otherwise, it returns false. This relies on the
  // has_cycle array being maintained correctly for leader vertices.
  bool query_cycle(int i) {
    int root_i = find_leader(i);
    return has_cycle[root_i];
  }

  // Please see the descriptions of the next two functions below.
  // (Do not edit these functions here; edit them below.)
  void dsunion(int i, int j);
  void count_comps(int n);
};

// TASK 1:
// dsunion performs disjoint set union. The reported leader of vertex j
// will become the leader of vertex i as well.
// Assuming it is only called once per pair of vertices i and j,
// it can detect when a set is including an edge that completes a cycle.
// This is evident when a vertex is assigned a leader that is the same
// as the one it was already assigned previously.
// Also, if you join two sets where either set already was known to
// have a cycle, then the joined set still has a cycle.
// Modify the implementation of dsunion below to properly adjust the
// has_cycle array so that query_cycle(root_j) accurately reports
// whether the connected component of root_j contains a cycle.
void DisjointSets::dsunion(int i, int j) {

  bool i_had_cycle = query_cycle(i);
  bool j_had_cycle = query_cycle(j);

  int root_i = find_leader(i);
  int root_j = find_leader(j);

  if (root_i != root_j) {

    //assign j as i's root
    leader[root_i] = root_j;
    root_i = root_j;
  }
  else {
    // A cycle is detected when dsunion is performed on an edge
    // where both vertices already report the same set leader.

    // update cycle property, make disjoint led by j has cycle
    has_cycle[ root_j ] = true;
    // TODO: Your work here! Update has_cycle accordingly.
  }

  // Also, if either one of the original sets was known to have a cycle
  // already, then the newly joined set still has a cycle.
  // TODO: Your work here!

  if( i_had_cycle || j_had_cycle )
  {
    has_cycle[ root_j ] = true;
  }

  return;
}
//end of function DisjointSets::dsunion(int i, int j)

// TASK 2:
// count_comps should count how many connected components there are in
// the graph, and it should set the num_components member variable
// to that value. The input n is the number of vertices in the graph.
// (Remember, the vertices are numbered with indices 0 through n-1.)
void DisjointSets::count_comps(int n) {

  // Insert code here to count the number of connected components
  // and store it in the "num_components" member variable.
  // Hint: If you've already performed set union on all the apparent edges,
  //  what information can you get from the leaders now?
  // TODO: Your work here!

  int num_of_connect_component = 0;

  for( int i = 0 ; i < n ; i ++ )
  {
    if( leader[i] < 0 )
    {
      num_of_connect_component++;
    }
  }

  num_components = num_of_connect_component;

  return;
}
//end of function DisjointSets::count_comps

int main() {

  constexpr int NUM_EDGES = 9;
  constexpr int NUM_VERTS = 8;

  int edges[NUM_EDGES][2] = {{0,1},{1,2},{3,4},{4,5},{5,6},{6,7},{7,3},{3,5},{4
    ,6}};

  DisjointSets d;

  // The union operations below should also maintain information
  // about whether leaders are part of connected components that
  // contain cycles. (See TASK 1 above where dsunion is defined.)
  for (int i = 0; i < NUM_EDGES; i++)
    d.dsunion(edges[i][0],edges[i][1]);

  // The count_comps call below should count the number of components.
  // (See TASK 2 above where count_comps is defined.)
  d.count_comps(NUM_VERTS);

  std::cout << "For edge list: ";
  for (int i = 0; i < NUM_EDGES; i++) {
    std::cout << "(" << edges[i][0] << ","
         << edges[i][1] << ")"
         // This avoids displaying a comma at the end of the list.
         << ((i < NUM_EDGES-1) ? "," : "\n");
  }

  std::cout << "You counted num_components: " << d.num_components << std::endl;
```

```
166
167   // The output for the above set of edges should be:
168   // You counted num_components: 2
169
170   std::cout << "Cycle reported for these vertices (if any):" << std::endl;
171   for (int i=0; i<NUM_VERTS; i++) {
172     if (d.query_cycle(i)) std::cout << i << " ";
173   }
174   std::cout << std::endl;
175
176   // The cycle detection output for the above set of edges should be:
177   // Cycle reported for these vertices (if any):
178   // 3 4 5 6 7
179
180   return 0;
181 }
182
```

Run

Reset

Correct

Passed at least 100 random tests! Last result:
For edge list: (0,1),(1,2),(2,3),(3,4),(4,2),(4,1),(5,6),(6,7),(7,8),(8,9),(9,6),(10,11),(11,12),(12,13),(13,14),(14,1(

You counted num_components: 4
Correct count is num_components: 4

Cycle query summary for these vertices (if any):
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Correct cycle query summary would be (if any):
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19