Congratulations! You passed!

TO PASS 80% or higher

Keep Learning

GRADE
100%

Week 4 Challenge Problem

LATEST SUBMISSION GRADE
100%

1. **We can use disjoint sets to implement a breadth first traversal. The code below implements a bfs() method that implements a breadth first traversal that also measures the distance (in number of edges) from each vertex to the vertex serving as the source of the traversal.**

   This algorithm uses disjoint sets to keep track of two sets. All of the vertices are initialized to be singletons (the only element of their set). The bfs(int i, int n, int m, int edges[][2]) method is called with the index i of the source vertex of the traversal. This vertex is assigned a distance of zero, and this vertex index i will be a member of the set of all processed vertices (vertices that the breadth first traversal has visited and measured an edge distance).

   We then iterate. (We iterate no more than m times. If the graph was, for example, a long line of edges and the start vertex was at one end, then each iteration would process only one edge. For most graphs, we will probably need fewer iterations so we have a conditional break later in the loop.) Each of these iterations adds a layer of breadth to the traversal.

   In each iteration, an inner loop cycles through all of the edges in the edge list. If any edge has one and only one vertex in the already-processed set (the same set as the start vertex i) then we add its other vertex to the current frontier set.

   After the inner loop has iterated through all of the edges, the frontier set contains all of the vertices one edge away from all of the already-processed edges. When each of these vertices is added to the frontier set, we assign its distance to be the current loop counter of the outer loop. Since all of the vertices one edge away from the already-processed set of vertices have now been found (and their distance has been recorded), these new vertices can now be added to the already-processed set with a union operation. Then the outer loop can increment the edge distance counter and the next frontier of new vertices can be found.

   In the code below, there are two conditions that need to be filled in. Each edge (say edge j) has two vertices: edge[j][0] and edge[j][1]. The condition needs to determine if one of these vertices is in the already processed set and the other is not, and if so, then the one that is not should be added to the frontier set. Your job is to figure out the condition using the variables defined, to determine if the appropriate vertex is a member (or not a member) of the already-processed set.

   5 / 5 points

```
#include <iostream>
#include <string>

// Note: You must not change the definition of DisjointSets here.
class DisjointSets {
public:
  int s[256];
  int distance[256];

  DisjointSets() {
    for (int i = 0; i < 256; i++) s[i] = distance[i] = -1;
  }

  int find(int i) { return s[i] < 0 ? i : find(s[i]); }

  void dsunion(int i, int j) {
    int root_i = find(i);
    int root_j = find(j);
    if (root_i != root_j) {
      s[root_i] = root_j;
    }
  }

  void bfs(int i, int n, int m, int edges[][2]);
};


/* Below are two conditions that need to be programmed
 * to allow this procedure to perform a breadth first
 * traversal and mark the edge distance of the graph's
 * vertices from vertex i.
 */

void DisjointSets::bfs(int i, int n, int m, int edges[][2]) {

  distance[i] = 0;

  // no need to iterate more than m times
  // but loop terminates when no new
  // vertices added to the frontier.

  for (int d = 1; d < m; d++) {

    // f is the index of the first
    // vertex added to the frontier
    int f = -1;

    // rooti is the name of the set
    // holding all of the vertices
    // that have already been assigned
    // distances

    int rooti = find(i);

    // loop through all of the edges
    // (this could be much more efficient
    // if an adjacency list was used
    // instead of a simple edge list)

    for (int j = 0; j < m; j++) {

      // root0 and root1 are the names of
      // the sets that the edge's two
      // vertices belong to
```

```
      int root0 = find(edges[j][0]);
      int root1 = find(edges[j][1]);

      if ( root0 == rooti && root1 != rooti) {

        // add the [1] vertex of the edge
        // to the frontier, either by
        // setting f to that vertex if it
        // is the first frontier vertex
        // found so far, or by unioning
        // it with the f vertex that was
        // already found.

        if (f == -1)
          f = edges[j][1];
        else
          dsunion(f,edges[j][1]);

        // set the distance of this frontier
        // vertex to d

        distance[edges[j][1]] = d;

      } else if ( root0 != rooti && root1 == rooti ) {
        if (f == -1)
          f = edges[j][0];
        else
          dsunion(f,edges[j][0]);
        distance[edges[j][0]] = d;
      }
    }

    // if no vertices added to the frontier
    // then we have run out of vertices and
    // are done, otherwise union the frontier
    // set with the set of vertices that have
    // already been processed.

    if (f == -1)
      break;
    else
      dsunion(f,i);
  }
}

int main() {

  int edges[8][2] = {{0,1},{1,2},{2,3},{3,4},{4,5},{5,6},{6,7},{7,3}};

  DisjointSets d;

  d.bfs(3,8,8,edges);

  for (int i = 0; i < 8; i++)
    std::cout << "Distance to vertex " << i
              << " is " << d.distance[i] << std::endl;

// Should print
// Distance to vertex 0 is 3
// Distance to vertex 1 is 2
// Distance to vertex 2 is 1
// Distance to vertex 3 is 0
// Distance to vertex 4 is 1
// Distance to vertex 5 is 2
// Distance to vertex 6 is 2
// Distance to vertex 7 is 1


  return 0;
}
```

Run

Reset

Correct

Correct!
Distance to vertex 0 is 3.
Distance to vertex 1 is 2.
Distance to vertex 2 is 1.
Distance to vertex 3 is 0.
Distance to vertex 4 is 1.
Distance to vertex 5 is 2.
Distance to vertex 6 is 2.
Distance to vertex 7 is 1.