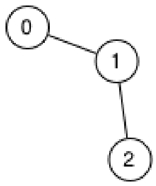
Week 07 Problem Set

Graph Traversal, Digraphs

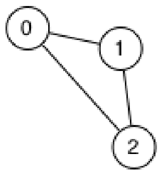
Note:
Sample solutions will be posted on Friday to help you prepare for the mid-term online test between Monday 12noon and Tuesday 12noon next week (10–11 September).

1. (Hamiltonian/Euler paths and circuits)

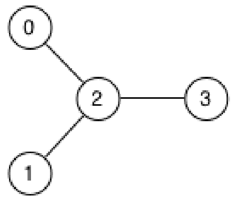
   a. Identify any Hamiltonian/Euler paths/circuits in the following graphs:



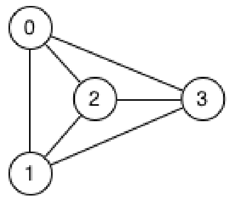
Graph 1



Graph 2

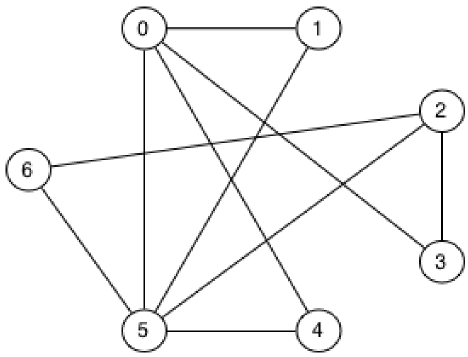


Graph 3



Graph 4

   b. Find an Euler path and an Euler circuit (if they exist) in the following graph:



Answer:

a. Graph 1: has both Euler and Hamiltonian paths (e.g. 0-1-2), but cannot have circuits as there are no cycles.

Graph 2: has both Euler paths (e.g. 0-1-2-0) and Hamiltonian paths (e.g. 0-1-2); also has both Euler and Hamiltonian circuits (e.g. 0-1-2-0).

Graph 3: has neither Euler nor Hamiltonian paths, nor Euler nor Hamiltonian circuits.

Graph 4: has Hamiltonian paths (e.g. 0-1-2-3) and Hamiltonian circuits (e.g. 0-1-2-3-0); it has neither an Euler path nor an Euler circuit.

b. An Euler path: 2-6-5-2-3-0-1-5-0-4-5

No Euler circuit since two vertices (2 and 5) have odd degree.

2. (Cycle check)

Take the "buggy" cycle check from the lecture and design a correct algorithm using depth-first search to determine if a graph has a cycle.

Answer:

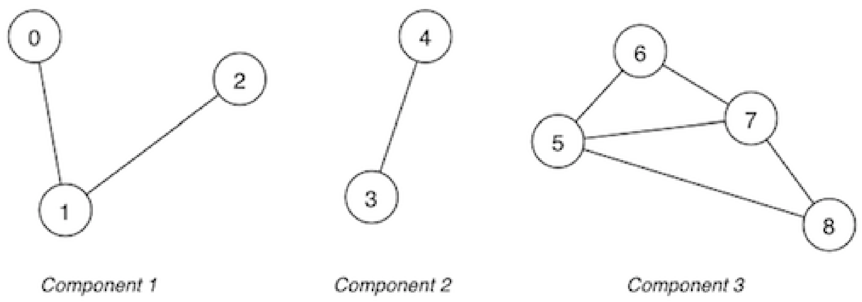
```
hasCycle(G):
|  Input  graph G
|  Output true if G has a cycle, false otherwise
|
|  mark all vertices as unvisited
|  for each vertex v∈G do             // make sure to check all connected components
|  |  if v has not been visited then
|  |     if dfsCycleCheck(G,v,v) then
|  |        return true
|  |     end if
|  |  end if
|  end for
|  return false
```

```
dfsCycleCheck(G,v,u):       // look for a cycle that does not go back directly to u
|  mark v as visited
|  for each (v,w)∈edges(G) do
|  |  if w has not been visited then
|  |     return dfsCycleCheck(G,w,v)
|  |  else if w≠u then
|  |     return true
|  |  end if
|  end for
|  return false
```

3. (Connected components)

a. Write a C program that computes the connected components in a graph. The graph should be built from user input in the same way as in exercise 2 last week (i.e., problem set week 6). Your program should use the Graph ADT (`Graph.h`, `Graph.c`) from the lecture. These files should not be changed.

An example of the program executing is shown below for the following graph:




```
prompt$ ./components
Enter the number of vertices: 9
Enter an edge (from): 0
Enter an edge (to): 1
Enter an edge (from): 1
Enter an edge (to): 2
Enter an edge (from): 4
Enter an edge (to): 3
Enter an edge (from): 6
Enter an edge (to): 5
Enter an edge (from): 6
Enter an edge (to): 7
Enter an edge (from): 5
Enter an edge (to): 7
Enter an edge (from): 5
Enter an edge (to): 8
Enter an edge (from): 7
Enter an edge (to): 8
Enter an edge (from): done
Finished.
Number of components: 3
Component 1:
0
1
2
Component 2:
3
4
Component 3:
5
6
7
8
```

Note that:

- the vertices within a component are printed in ascending order
- the components themselves are output in ascending order of their smallest node.

You may assume that a graph has a maximum of 1000 nodes.

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` *program that corresponds to the problem set and week. It expects to find a program named* `components.c` *in the current directory. You can use dryrun as follows:*

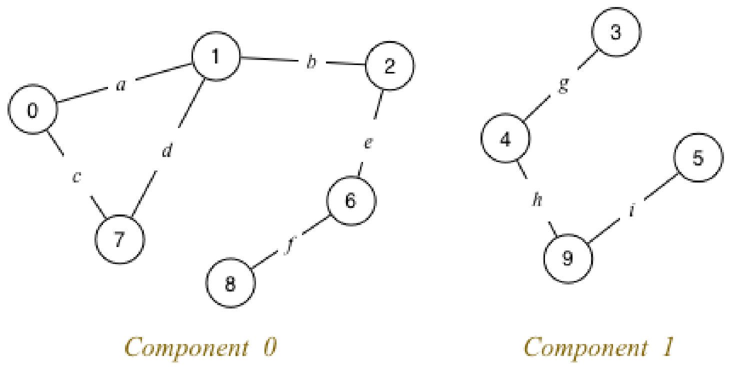
```
prompt$ ~cs9024/bin/dryrun prob07
```

b. Computing connected components can be avoided by maintaining a vertex-indexed connected components array as part of the `Graph` representation structure:

```
typedef struct GraphRep *Graph;

struct GraphRep {
   ...
   int nC;  // # connected components
   int *cc; /* which component each vertex is contained in
               i.e. array [0..nV-1] of 0..nC-1 */
   ...
}
```

Consider the following graph with multiple components:



Assume a vertex-indexed connected components array cc[0..nV-1] as introduced above:

```
nC   = 2
cc[] = {0,0,0,1,1,1,0,0,0,1}
```

Show how the `cc[]` array would change if

1. edge *d* was removed
2. edge *b* was removed

c. Consider an adjacency matrix graph representation augmented by the two fields

- `nC` (number of connected components)
- `cc[]` (connected components array)

These fields are initialised as follows:

```
newGraph(V):
|  Input  number of nodes V
|  Output new empty graph
|
|  g.nV=V, g.nE=0, g.nC=V
|  allocate memory for g.edges[][]
|  for all i=0..V-1 do
|     g.cc[i]=i
|     for all j=0..V-1 do
|        g.edges[i][j]=0
|     end for
|  end for
|  return g
```

Modify the pseudo-code for edge insertion and edge removal from the lecture (week 6) to maintain the two new fields.

Answer:

a. The following two functions together implement the algorithm from the lecture that uses the following strategy:

- pick a not-yet-visited vertex
- find all vertices reachable from that vertex
- increment the count of connected components
- repeat above until all vertices have been visited

```
#define MAX_NODES 1000
int componentOf[MAX_NODES];

void dfsComponents(Graph g, int v, int id) {
   componentOf[v] = id;
   Vertex w;
   for (w = 0; w < numOfVertices(g); w++)
```

```
      if (adjacent(g, v, w) && componentOf[w] == -1)
         dfsComponents(g, w, id);
}

// computes the connected component array
// and returns the number of connected components
int components(Graph g) {
   Vertex v;
   int nV = numOfVertices(g);
   for (v = 0; v < nV; v++)
      componentOf[v] = -1;

   int compID = 0;
   for (v = 0; v < nV; v++) {
      if (componentOf[v] == -1) {
         dfsComponents(g, v, compID);
         compID++;
      }
   }
   return compID;
}
```

Calling the function and printing the result:

```
int i, c = components(g);
printf("Number of connected components: %d\n", c);
for (i = 0; i < c; i++) {
   printf("Component %d:\n", i+1);
   Vertex v;
   for (v = 0; v < numOfVertices(g); v++)
      if (componentOf[v] == i)
         printf("%d\n", v);
}
```

b. After removing *d*, `cc[] = {0,0,0,1,1,1,0,0,0,1}` (i.e. unchanged)
After removing *b*, `cc[] = {0,0,2,1,1,1,2,0,2,1}` with `nC=3`

c. Inserting an edge may reduce the number of connected components:

```
insertEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if g.edges[v][w]=0 then                  // (v,w) not in graph
|  |  g.edges[v][w]=1, g.edges[w][v]=1      // set to true
|  |  g.nE=g.nE+1
|  |  if g.cc[v]≠g.cc[w] then               // v,w in different components?
|  |  |  c=min{g.cc[v],g.cc[w]}             // ⇒ merge components c and d
|  |  |  d=max{g.cc[v],g.cc[w]}
|  |  |  for all vertices v∈g do
|  |  |     if g.cc[v]=d then
|  |  |        g.cc[v]=c                    // move node from component d to c
|  |  |     else if g.cc[v]=g.nC-1 then
|  |  |        g.cc[v]=d                    // replace largest component ID by d
|  |  |     end if
|  |  |  end for
|  |  |  g.nC=g.nC-1
|  |  end if
|  end if
```

Removing an edge may increase the number of connected components:

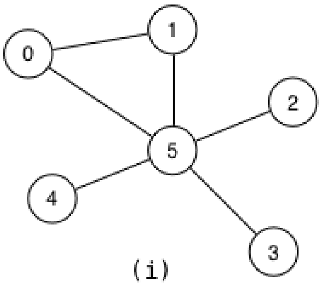
```
removeEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if g.edges[v][w]≠0 then                  // (v,w) in graph
|  |  g.edges[v][w]=0, g.edges[w][v]=0      // set to false
|  |  if not hasPath(g,v,w) then            // v,w no longer connected?
|  |     dfsNewComponent(g,v,g.nC)          // ⇒ put v + connected vertices into new component
|  |     g.nC=g.nC+1
|  |  end if
|  end if

dfsNewComponent(g,v,componentID):
|  Input graph g, vertex v, new componentID for v and connected vertices
|
|  g.cc[v]=componentID
|  for all vertices w adjacent to v do
```

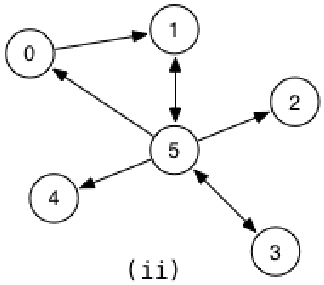
```
|     if g.cc[w]≠componentID then
|        dfsNewComponent(g,w,componentID)
|     end if
|  end if
```

4. (Digraphs)

   a. For each of the following graphs:



(i)



(ii)

Show the concrete data structures if the graph was implemented via:

- adjacency matrix representation (assume full *V*×*V* matrix)
- adjacency list representation (if non-directional, include both *(v,w)* and *(w,v)*)

   b. Consider the following map of streets in the Sydney CBD:

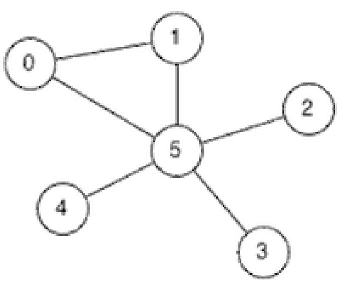
Represent this as a directed graph, where intersections are vertices and the connecting streets are edges. Ensure that the directions on the edges correctly reflect any one-way streets (this is a driving map, not a walking map). You only need to make a graph which includes the intersections marked with red letters Some things that don't show on the map: Castlereagh St is one-way heading south and Hunter St is one-way heading west.

For each of the following pairs of intersections, indicate whether there is a path from the first to the second. Show a path if there is one. If there is more than one path, show two different paths.

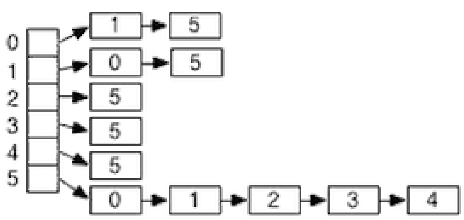
1. from intersection "D" on Margaret St to insersection "L" on Pitt St
2. from intersection "J" to the corner of Margaret St and York St (intersection "A")
3. from the intersection of Margaret St and York St ("A") to the intersection of Hunter St and Castlereagh St ("M")
4. from intersection "M" on Castlereagh St to intersection "H" on York St

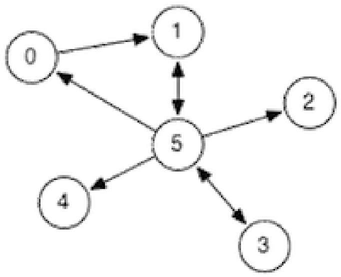
Answer:

a.

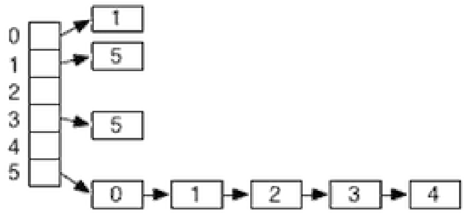


	0	1	2	3	4	5
0	0	1	0	0	0	1
1	1	0	0	0	0	1
2	0	0	0	0	0	1
3	0	0	0	0	0	1
4	0	0	0	0	0	1
5	1	1	1	1	1	0

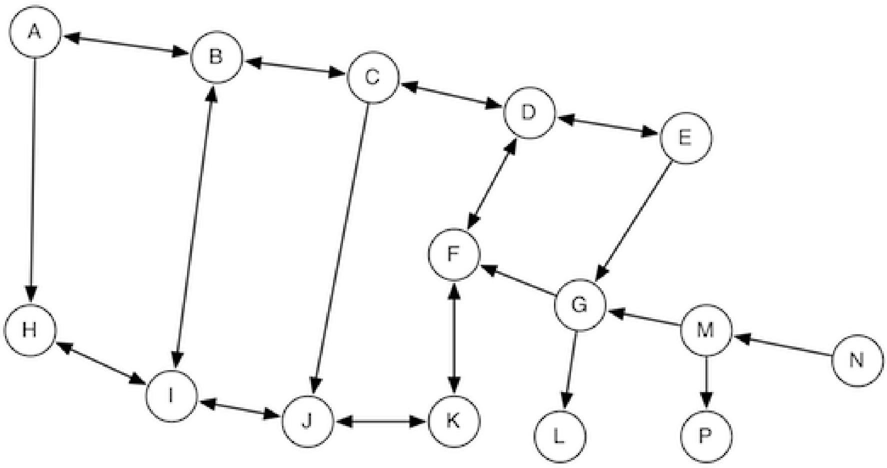




	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	0	0
3	0	0	0	0	0	1
4	0	0	0	0	0	0
5	1	1	1	1	1	0



b. The graph is as follows. Bi-directional edges are depicted as two-way arrows, rather than having two separate edges going in opposite directions.



For the paths:

1. D → E → G → L and there are no other choices that don't involve loops through D
2. J → I → B → A or J → K → F → D → C → B → A
3. You can't reach M (or N or P) from A on this graph. Real-life is different, of course.
4. M → G → F → D → C → B → A → H or M → G → F → K → J → I → H

5. (Mock test)

   Login to COMP9024 on Moodle between now and Sunday, 9 September, 12noon to do the "Mock Test" in preparation for the mid-term online test.

6. **Challenge Exercise**

   None this week — review all the course contents so far & get ready for the mid-term online test.