COMP9024 19T0 | Data Structures and Algorithms

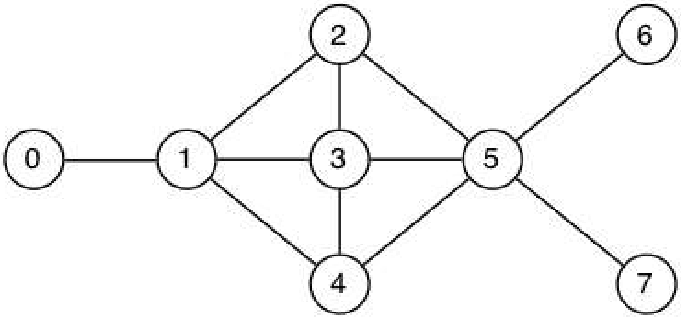
Week 04 Problem Set
Graph Traversal, Digraphs

[Show with no answers] [Show with all answers]

1. (Graph traversal: DFS and BFS)

   Show the order in which the nodes of the graph depicted below are visited by

   a. DFS starting at node 0
   b. DFS starting at node 3
   c. BFS starting at node 0
   d. BFS starting at node 3



Assume the use of a stack for depth-first search (DFS) and a queue for breadth-first search (BFS), respectively. Show the state of the stack or queue explicitly in each step. When choosing which neighbour to visit next, always choose the smallest unvisited neighbour.

[hide answer]

a. DFS starting at 0:

```
Current    Stack (top at left)
-          0
0          1
1          2 3 4
2          5 3 4
5          6 7 3 4
6          7 3 4
7          3 4
3          4
4          -
```

b. DFS starting at 3:

```
Current    Stack (top at left)
-          3
3          1 2 4 5
1          0 2 4 5
0          2 4 5
2          4 5
4          5
5          6 7
6          7
7          -
```

c. BFS starting at 0:

```
Current    Queue (front at left)
-          0
0          1
1          2 3 4
2          3 4 5
3          4 5
4          5
5          6 7
```

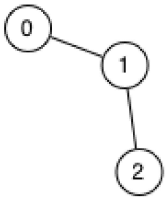
```
6           7
7           -
```

d. BFS starting at 3:

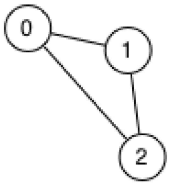
```
Current     Queue (front at left)
-           3
3           1 2 4 5
1           2 4 5 0
2           4 5 0
4           5 0
5           0 6 7
0           6 7
6           7
7           -
```

2. (Hamiltonian/Euler paths and circuits)

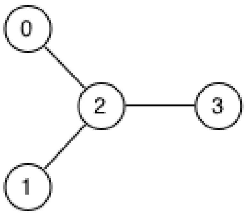
   a. Identify any Hamiltonian/Euler paths/circuits in the following graphs:



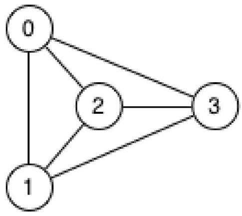
Graph 1



Graph 2

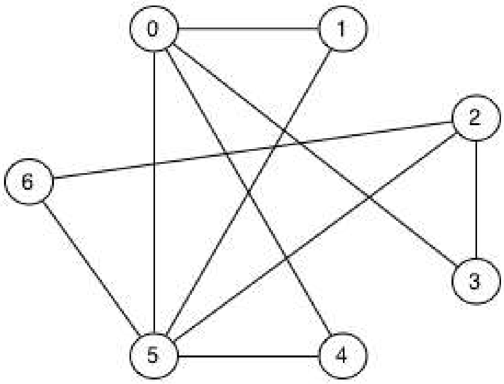


Graph 3



Graph 4

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



[show answer]

3. (Cycle check)

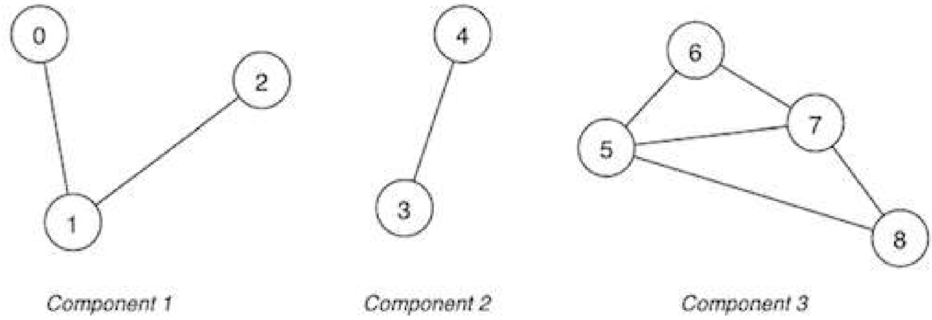
   Design an algorithm to check for the existence of a cycle in a graph using depth-first search.

   [show answer]

4. (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 5 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:

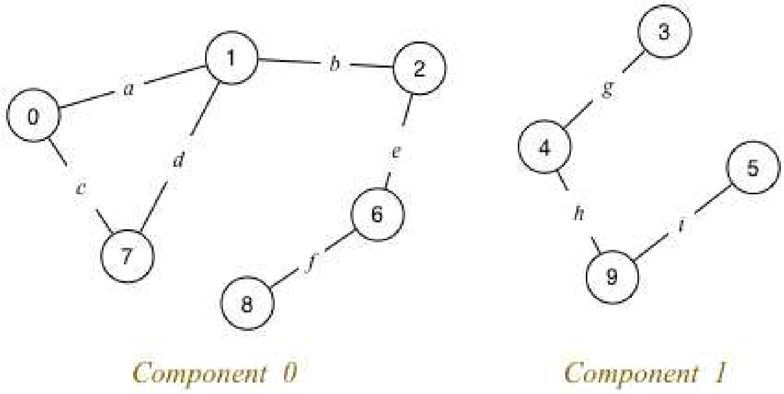
- the vertices within a component are printed in ascending order
- the components themselves are output in ascending order of their smallest node.

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 prob04
```

b. Consider the following graph with multiple components:



Assume a vertex-indexed connected components array `cc[0..nV-1]` that might form part of the `Graph` representation structure for this graph as suggested in the lecture:

```
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:

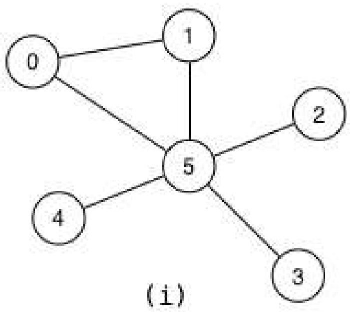
```
newGraph(V):
|  Input  number of nodes V
|  Output new empty graph
|
|  g.nV=V,  g.nE=0,  g.nC=V
|  allocate memory for g.edges[][]
|  for all i=0..V-1 do
|     g.cc[i]=i
|     for all j=0..V-1 do
|        g.edges[i][j]=0
|     end for
|  end for
|  return g
```

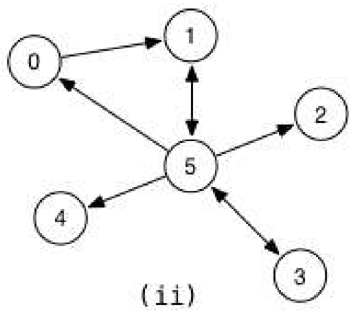
Modify the pseudo-code for edge insertion and edge removal from the lecture (week 6) to maintain the two new fields.

[show answer]

5. (Digraphs)

a. For each of the following graphs:





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

- adjacency matrix representation (assume full $V \times V$ 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.

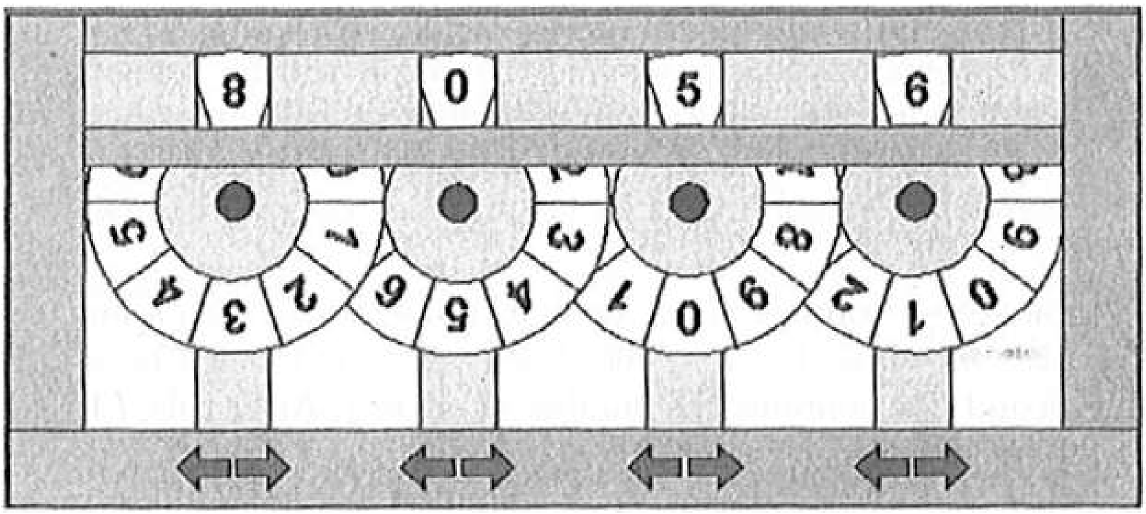
For each of the following pairs of intersections, indicate whether there is a path from the first to the second. Show a path if there is one. If there is more than one path, show two different paths.

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

[show answer]

6. **Challenge Exercise**

Consider a machine with four wheels. Each wheel has the digits 0…9 printed clockwise on it. The current *state* of the machine is given by the four topmost digits abcd, e.g. 8056 in the picture below.



Each wheel can be controlled by two buttons: Pressing the button labelled with "←" turns the corresponding wheel clockwise one digit ahead, whereas presssing "→" turns it anticlockwise one digit back.

Write a C-program to determine the *minimum* number of button presses required to transform

- a given initial state, $abcd$
- to a given goal state, $efgh$
- without passing through any of *n≥0* given "forbidden" states, $\mathrm{forbidden}[]=\{w_1x_1y_1z_1, \cdots, w_nx_ny_nz_n\}$.

For example, the state 8056 as depicted can be transformed into 0056 in 2 steps if $\mathrm{forbidden}[]=\{\}$, whereas a minimum of 4 steps is needed for the same task if $\mathrm{forbidden}[]=\{9056\}$. (Why?)

Use your program to compute the least number of button presses required to transform 8056 to 7012 if

a. there are no forbidden states
b. you are not permitted to pass through any state 7055–8055 (i.e., 7055, 7056, …, 8055 all are forbidden)
c. you are not permitted to pass through any state 0000–0999 or 7055–8055

[show answer]