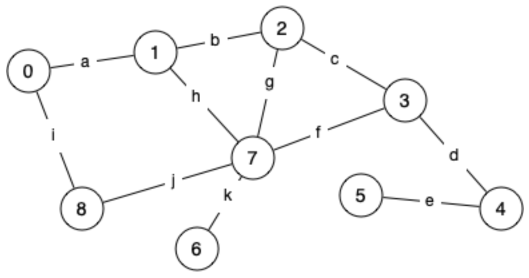
COMP9024 24T1 Data Structures and Algorithms

Week 5 Tutorial
Graph Data Structures

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

1. (Graph properties)

   Consider the following graph



   The edges are labelled simply for convenience in describing graph properties.

   a. How many edges does it have?
   b. How many *cycles* are there in the graph?
   c. How many *cliques* with at least 3 nodes are there in the graph?
   d. What is the *degree* of each vertex?
   e. How many edges in the *longest* simple path from 5 to 8?

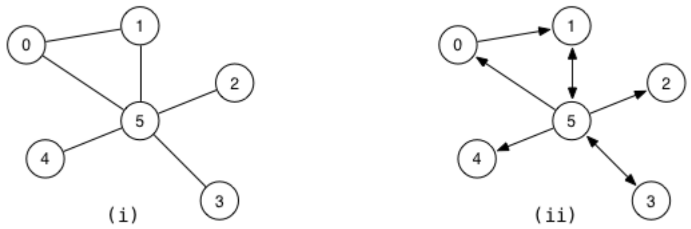
   [hide answer]

   a. 11 edges (labelled a..k)
   b. 6 cyles: 0-1-7-8-0, 1-2-7-1, 2-3-7-2, 0-1-2-7-8-0, 0-1-2-3-7-8-0, 1-2-3-7-1
   c. 2 cliques: {1, 2, 7}, {2, 3, 7}
   d. d(0) = 2, d(1) = 3, d(2) = 3, d(3) = 3, d(4) = 2, d(5) = 1, d(6) = 1, d(7) = 5, d(8) = 2
   e. 7 edges, path is 5-4-3-2-7-1-0-8 or 5-4-3-7-2-1-0-8

2. (Graph representations)

   a. How is the adjacency matrix for a directed graph different to that of an undirected graph?

   b. For each of the following graphs:



(i) (ii)

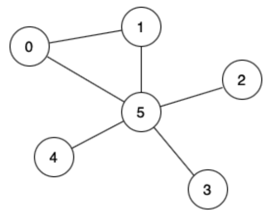
   Show the concrete data structures if the graph was implemented via:

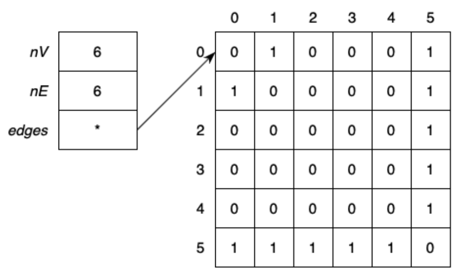
   i. an adjacency matrix representation (V×V matrix, and if non-directional, with each edge represented twice)
   ii. an adjacency list representation (if non-directional, with each edge appearing in two lists, one for *v* and one for *w*)

   [hide answer]

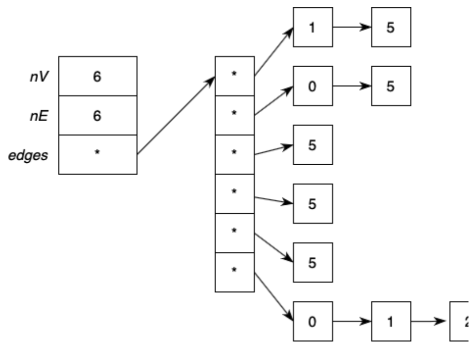
   a. The adjacency matrix for an undirected graph is symmetric and has zeroes on the leading diagonal (no self-edges). The adjacency matrix for a directed graph is typically not symmetric and can have non-zero values on the leading diagonal (self-edges are allowed).

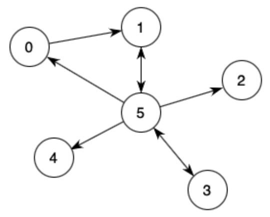
   b.

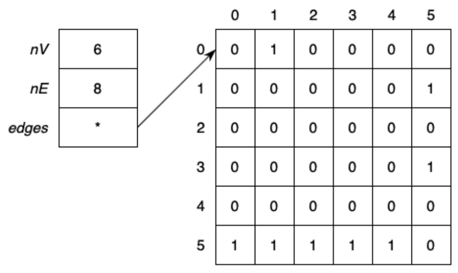




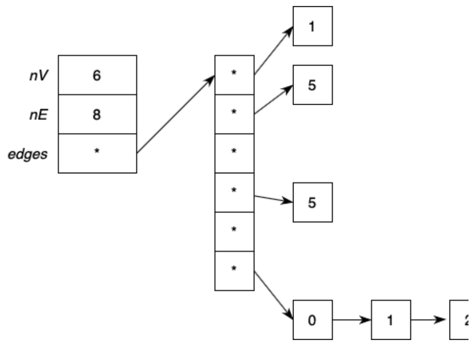
	0	1	2	3	4	5
0	0	1	0	0	0	1
1	1	0	0	0	0	1
2	0	0	0	0	0	1
3	0	0	0	0	0	1
4	0	0	0	0	0	1
5	1	1	1	1	1	0







	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



3. (Graph ADT clients)

   The lecture included a Graph ADT Client exercise to write a program that uses the graph ADT to:

   - build a graph; and
   - print all the nodes that are incident to a particular vertex in ascending order.

   Modify the code from the lecture to build the undirected graph in Exercise 2b, then print all the nodes that are incident to vertex 1 in *descending* order.

   [hide answer]

```c
#include <stdio.h>
#include "Graph.h"

#define NODES 6
#define NODE_OF_INTEREST 1

int main(void) {
   Graph g = newGraph(NODES);

   Edge e;
   e.v = 0; e.w = 1; insertEdge(g,e);
   e.v = 0; e.w = 5; insertEdge(g,e);
   e.v = 1; e.w = 5; insertEdge(g,e);
   e.v = 2; e.w = 5; insertEdge(g,e);
   e.v = 3; e.w = 5; insertEdge(g,e);
   e.v = 4; e.w = 5; insertEdge(g,e);

   int v;
   for (v = NODES-1; v >= 0; v--) {
      if (adjacent(g, v, NODE_OF_INTEREST))
         printf("%d\n", v);
   }

   freeGraph(g);
   return 0;
}
```

4. (Social graphs)

   Facebook could be considered as a giant "social graph"

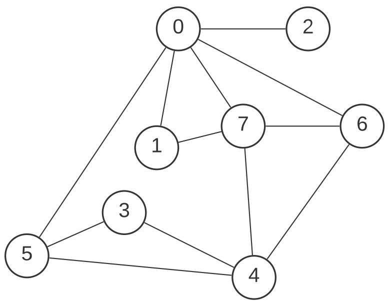
   a. What are the vertices?
   b. What are the edges?
   c. Are edges directional?
   d. What does the degree of each vertex represent?
   e. What kind of graph algorithm could suggest potential friends?

   [hide answer]

   a. Vertices are people (Facebook users)
   b. Edges are "friend" relationships
   c. No - if you are someone's friend, they are also your friend
   d. The degree of a vertex is the number of friends that the person represented by the vertex has
   e. Breadth-first search - find your immediate friends, then consider their friends, and so on

5. (Graph traversal)

   Consider the breadth-first and depth-first traversal algorithms below and the following graph:



```c
void breadthFirst(Graph g, int src) {
   int nV = numOfVertices(g);
   bool *visited = calloc(nV, sizeof(bool));
   int *pred = calloc(nV, sizeof(int));
   queue q = newQueue();

   visited[src] = true;
   QueueEnqueue(q, src);
   while (!QueueIsEmpty(q)) {
      int v = QueueDequeue(q);

      printf("%d\n", v);
      for (int w = 0; w < nV; w++) {
         if (adjacent(g, v, w) && !visited[w]) {
            visited[w] = true;
            pred[w] = v;
            QueueEnqueue(q, w);
         }
      }
   }

   free(visited);
   free(pred);
   dropQueue(q);
}

void depthFirst(Graph g, int src) {
   int nV = numOfVertices(g);
   bool *visited = calloc(nV, sizeof(bool));
   int *pred = calloc(nV, sizeof(int));
   stack s = newStack();

   StackPush(s, src);
   while (!StackIsEmpty(s)) {
      int v = StackPop(s);

      if (visited[v]) continue;
      visited[v] = true;

      printf("%d\n", v);
      for (int w = nV - 1; w >= 0; w--) {
         if (adjacent(g, v, w) && !visited[w]) {
            pred[w] = v;
            StackPush(s, w);
         }
      }
   }

   free(visited);
   free(pred);
   dropStack(s);
}
```

Trace the execution of the traversal algorithms, and show the state of the `visited` and `pred` arrays and the `Queue` (BFS) or `Stack` (DFS) at the end of each iteration, for each of the following function calls:

```
breadthFirst(g, 0);
breadthFirst(g, 3);
depthFirst(g, 0);
depthFirst(g, 3);
```

[hide answer]

For `breadthFirst(g, 0)`:

```
                    visited                        pred
#  Printed  0  1  2  3  4  5  6  7    0  1  2  3  4  5  6  7    Queue (front at left)
-------------------------------------------------------------------------------------
0     -     1  0  0  0  0  0  0  0    -  -  -  -  -  -  -  -    0
1     0     1  1  1  0  0  1  1  1    -  0  0  -  -  0  0  0    1  2  5  6  7
2     1     1  1  1  0  0  1  1  1    -  0  0  -  -  0  0  0    2  5  6  7
3     2     1  1  1  0  0  1  1  1    -  0  0  -  -  0  0  0    5  6  7
4     5     1  1  1  1  1  1  1  1    -  0  0  5  5  0  0  0    6  7  3  4
5     6     1  1  1  1  1  1  1  1    -  0  0  5  5  0  0  0    7  3  4
6     7     1  1  1  1  1  1  1  1    -  0  0  5  5  0  0  0    3  4
7     3     1  1  1  1  1  1  1  1    -  0  0  5  5  0  0  0    4
8     4     1  1  1  1  1  1  1  1    -  0  0  5  5  0  0  0
```

For `breadthFirst(g, 3)`:

```
                    visited                        pred
#  Printed  0  1  2  3  4  5  6  7    0  1  2  3  4  5  6  7    Queue (front at left)
-------------------------------------------------------------------------------------
0     -     0  0  0  1  0  0  0  0    -  -  -  -  -  -  -  -    3
1     3     0  0  0  1  1  1  0  0    -  -  -  -  3  3  -  -    4  5
2     4     0  0  0  1  1  1  1  1    -  -  -  -  3  3  4  4    5  6  7
3     5     1  0  0  1  1  1  1  1    5  -  -  -  3  3  4  4    6  7  0
4     6     1  0  0  1  1  1  1  1    5  -  -  -  3  3  4  4    7  0
5     7     1  1  0  1  1  1  1  1    5  7  -  -  3  3  4  4    0  1
6     0     1  1  1  1  1  1  1  1    5  7  0  -  3  3  4  4    1  2
7     1     1  1  1  1  1  1  1  1    5  7  0  -  3  3  4  4    2
8     2     1  1  1  1  1  1  1  1    5  7  0  -  3  3  4  4
```

For `depthFirst(g, 0)`:

```
                   visited                    pred
#  Printed  0  1  2  3  4  5  6  7    0  1  2  3  4  5  6  7    Stack (top at right)
--------------------------------------------------------------------------------------
0     -     0  0  0  0  0  0  0  0    -  -  -  -  -  -  -  -    0
1     0     1  0  0  0  0  0  0  0    -  0  0  -  -  0  0  0    7  6  5  2  1
2     1     1  1  0  0  0  0  0  0    -  0  0  -  -  0  0  1    7  6  5  2  7
3     7     1  1  0  0  0  0  0  1    -  0  0  -  7  0  7  1    7  6  5  2  6  4
4     4     1  1  0  0  1  0  0  1    -  0  0  4  7  4  4  1    7  6  5  2  6  6  5  3
5     3     1  1  0  1  1  0  0  1    -  0  0  5  7  4  4  1    7  6  5  2  6  6  5  5
6     5     1  1  0  1  1  1  0  1    -  0  0  5  7  4  4  1    7  6  5  2  6  6  5
7     6     1  1  0  1  1  1  1  1    -  0  0  5  7  4  4  1    7  6  5  2  6
8     2     1  1  1  1  1  1  1  1    -  0  0  5  7  4  4  1    7  6  5
```

For `depthFirst(g, 3)`:

```
                   visited                    pred
#  Printed  0  1  2  3  4  5  6  7    0  1  2  3  4  5  6  7    Stack (top at right)
--------------------------------------------------------------------------------------
0     -     0  0  0  0  0  0  0  0    -  -  -  -  -  -  -  -    3
1     3     0  0  0  1  0  0  0  0    -  -  -  -  3  3  -  -    5  4
2     4     0  0  0  1  1  0  0  0    -  -  -  -  3  4  4  4    5  7  6  5
3     5     0  0  0  1  1  1  0  0    5  -  -  -  3  4  4  4    5  7  6  0
4     0     1  0  0  1  1  1  0  0    5  0  0  -  3  4  0  0    5  7  6  7  6  2  1
5     1     1  1  0  1  1  1  0  0    5  0  0  -  3  4  0  1    5  7  6  7  6  2  7
6     7     1  1  0  1  1  1  0  1    5  0  0  -  3  4  7  1    5  7  6  7  6  2  6
7     6     1  1  0  1  1  1  1  1    5  0  0  -  3  4  7  1    5  7  6  7  6  2
8     2     1  1  1  1  1  1  1  1    5  0  0  -  3  4  7  1    5  7  6  7  6
```

6. (Connected components)

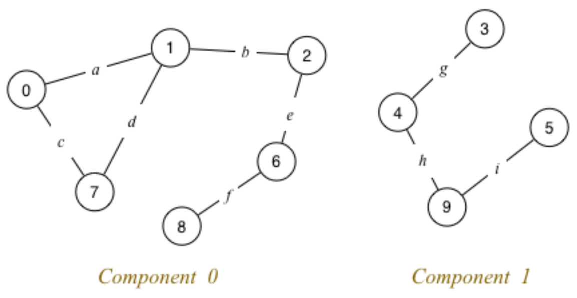
   a. What is the difference between a connected graph and a complete graph? Give simple examples of each.

   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}
```

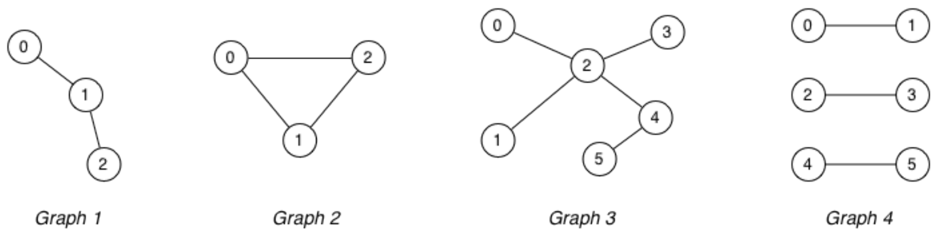
   Show how the `cc[]` array would change if

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

[hide answer]

   a. A connected graph is a graph where all vertices are reachable from all others, while a complete graph is a graph where all vertices are directly connected (by edges) to all others.

   Connected graphs have the property that there is a single connected component. Complete graphs have the property that every vertex has *V* - 1 incident edges, where *V* is the number of vertices in the graph. A graph cannot be complete if it is not connected. Some simple examples:



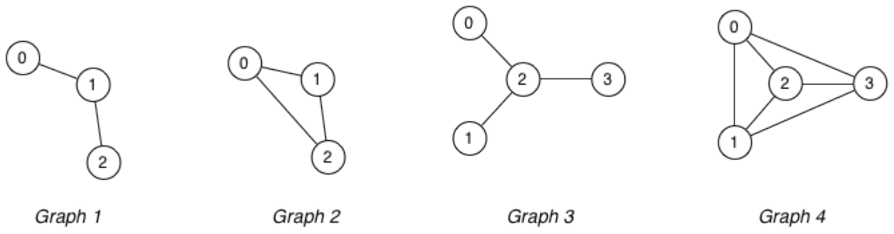
Graphs 1, 2 and 3 are connected. Graph 2 is also complete. Graph 4 is not connected, and therefore also not complete despite each of the individual c omponents being complete.

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`

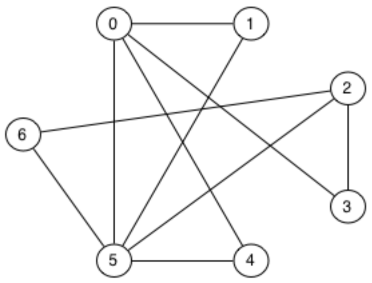
7. (Hamiltonian/Euler paths and circuits)

   a. What is the difference between a Hamiltonian path/circuit and an Euler path/circuit?

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



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



   d. Write a function to check whether a path, supplied as an `Edge` array, is an Euler path. Assume the function has the interface:

```
bool isEulerPath(Graph g, Edge e[], int nE)
```

   where `e[]` is an array of `nE` edges, in path order. You may also assume the standard Graph ADT interface from the lectures has been expanded to include a function:

```
int numOfEdges(Graph);
```

[hide answer]

a. A Hamiltonian path is a path that visits each vertex in the graph exactly once. A Hamiltonian circuit is a cycle that visits each vertex exactly once. An Euler path is a path that traverses each edge in the graph exactly once. An Euler circuit is an Euler path that ends at the same vertex where it started.

b. 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.

c. 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.

d.
```
// check whether a given path is a Euler path
bool isEulerPath(Graph g, Edge e[], int nE) {
   assert(g != NULL);

   // includes all edges
   if (numOfEdges(g) != nE) {
      return false;
   }

   // edges are actually in the graph
   for (int i = 0; i < nE; i++) {
      if (!adjacent(g, e[i].v, e[i].w)) {
         return false;
      }
   }

   // is actually a path
   for (int i = 0; i < nE - 1; i++) {
      if (e[i].w != e[i + 1].v) {
         return false;
      }
   }

   // includes edges exactly once
   for (int i = 0; i < nE; i++) {
      for (int j = i + 1; j < nE; j++) {
         if (e[i].v == e[j].v && e[i].w == e[j].w) return false;
```

```
            if (e[i].v == e[j].w && e[i].w == e[j].v) return false;
        }
    }

    return true;
}
```

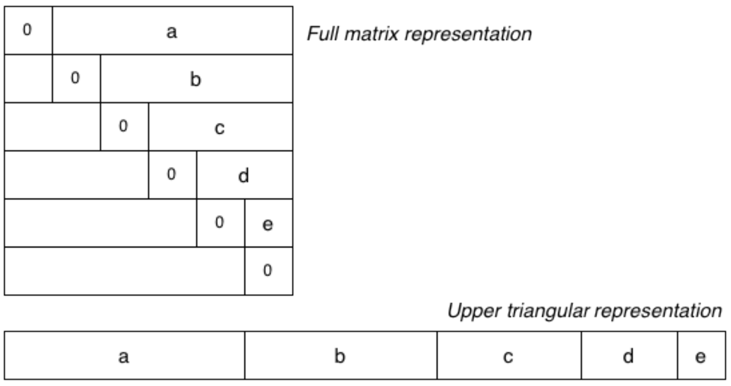
Challenge Exercises

8. (Graph representations)

   a. Consider the adjacency matrix and adjacency list representations for graphs. Analyse the storage costs for the two representations in more detail in terms of the number of vertices *V* and the number of edges *E*. Determine the E:V ratio at which it is more storage efficient to use an adjacency matrix representation vs the adjacency list representation.

      Assumptions. For the purposes of the analysis, *ignore the cost of storing the `GraphRep` structure*. Assume that: each pointer is 8 bytes long and a `Vertex` value is 8 bytes, so a linked-list *node* is 16 bytes long, and the adjacency matrix is a complete *V*×*V* matrix. Assume also that each adjacency matrix element is **1 byte** long. (*Hint:* Defining the matrix elements as 1-byte boolean values rather than 4-byte integers is a simple way to improve the space efficiency for the adjacency matrix representation.)

   b. The standard adjacency matrix representation for a graph uses a full *V*×*V* matrix and stores each edge twice (at [v,w] and [w,v]). This consumes a lot of space, and wastes a lot of space when the graph is sparse. One way to use less space is to store just the upper (or lower) triangular part of the matrix, as shown in the diagram below:



      The *V*×*V* matrix has been replaced by a single 1-dimensional array `g.edges[]` containing just the "useful" parts of the matrix.

      Accessing the elements is no longer as simple as `g.edges[v][w]`. Write pseudocode for a method to check whether two vertices `v` and `w` are adjacent under the upper-triangle matrix representation of a graph `g`.

[hide answer]

   a. The adjacency matrix representation always requires a *V*×*V* matrix, regardless of the number of edges, where each element is 1 byte long. It also requires an array of *V* pointers. This gives a fixed size of $V\cdot 8+V^2$ bytes.

      The adjacency list representation requires an array of *V* pointers (the start of each list), with each being 8 bytes long, and then one list node for each edge in each list. The total number of edge nodes is *2E* (each edge *(v,w)* is stored twice, once in the list for *v* and once in the list for *w*). Since each node requires 16 bytes (vertex+padding+pointer), this gives a size of $V\cdot 8+16\cdot 2\cdot E$. The total storage is thus $V\cdot 8+32\cdot E$.

      Since both representations involve *V* pointers, the difference is based on V^2 vs *32E*. So, if $32E < V^2$ (or, equivalently, $E:V < V/32$), then the adjacency list representation will be more storage-efficient. Conversely, if $E:V > V/32$, then the adjacency matrix representation will be more storage-efficient.

      To pick a concrete example, if $V=60$ and if we have 112 or fewer edges (112/60 = 1.867 < 60/32 = 1.875), then the adjacency list will be more storage-efficient, otherwise the adjacency matrix will be more storage-efficient.

   b. The following solution uses a loop to compute the correct index in the 1-dimensional `edges[]` array:

```
adjacent(g,v,w):
   Input  graph g in upper-triangle matrix representation
          v, w vertices such that v≠w
   Output true if v and w adjacent in g, false otherwise

   if v>w then
      swap v and w        // to ensure v<w
   end if
   chunksize=g.nV-1, offset=0
   for all i=0..v-1 do
      offset=offset+chunksize
      chunksize=chunksize-1
   end if
   offset=offset+w-v-1
   if g.edges[offset]=0 then return false
                        else return true
   end if
```

      Alternatively, you can compute the overall offset directly via the formula $(nV-1)+(nV-2)+\ldots+(nV-v)+(w-v-1) = \frac{v}{2}(2\cdot nV-v-1)+(w-v-1)$ (assuming that $v < w$).

9. (Connected components)

   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 pseudocode for edge insertion and edge removal from the lecture to maintain the two new fields.

[hide answer]

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
```

10. (Maximum clique)

Write pseudocode to compute the *largest* size of a clique in a graph. For example, if the input happens to be the complete graph K_5 but with any one edge missing, then the output should be 4.

Hint: Computing the maximum size of a clique in a graph is known to be an *NP-hard problem*. Try a generate-and-test strategy.

[hide answer]

As an NP-hard problem, no tractable algorithm for computing the maximum size of a clique in a graph is known. Here is a sample 'brute-force' algorithm that essentially generates-and-tests all possible subsets of vertices to determine the maximum size of a complete subgraph.

```
maxCliqueSize(g,v,clique,k):
|  Input  g       graph with n nodes 0..n-1
|         v       next vertex to consider
|         clique  some subset of nodes 0..v-1 that forms a clique
|         k       size of that clique
|  Output size of largest complete subgraph of g that extends clique with nodes from v..n-1
|
|  if v=n then                             // no more vertices to consider
|     return k
|  else
|  |  k1=maxCliqueSize(g,v+1,clique,k)     /* find largest complete subgraph that
|  |                                          extends clique without considering v */
|  |  for all w∈clique do                  // check if v can be added to clique:
|  |  |  if v is not adjacent to w then    // if v not adjacent to some node in clique
|  |  |     return k1                    // ⇒ return largest clique size without v
|  |  |  end if
|  |  end for
|  |  add v to clique
|  |  k2=maxCliqueSize(g,v+1,clique,k+1)  // find largest clique extending clique ∪ {v}
|  |  if k2>k1 then return k2
|  |          else return k1
|  |  end if
|  end if
```

Starting with an empty `clique`, the function call `maxCliqueSize(g,0,clique,0)` will return the maximum clique size of graph `g`.

Click here if you are interested in reading more about the computational aspects of computing cliques including some references to more sophisticated algorithms.