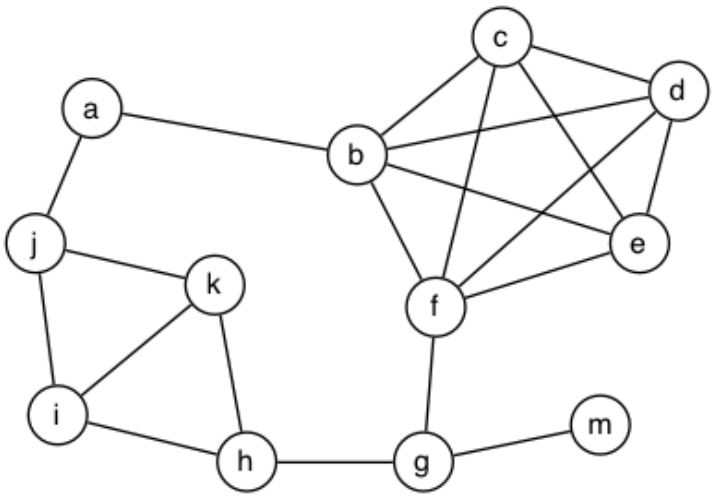
Week 5 Problem Set
Graph Data Structures and Graph Search

1. (Graph fundamentals)

   For the graph



   give examples of the smallest (but not of size/length 0) and largest of each of the following:

   a. path
   b. cycle
   c. spanning tree
   d. vertex degree
   e. clique

2. (Graph properties)

   a. Write pseudocode for computing

      - the minimum and maximum vertex degree
      - all 3-cliques (i.e. cliques of 3 nodes, "triangles")

      of a graph *g* with *n* vertices.

      Your methods should be representation-independent; the only function you should use is to check if two vertices $v,w \in \{0,\ldots n-1\}$ are adjacent in *g*.

   b. Determine the asymptotic complexity of your two algorithms. Assume that the adjacency check is performed in constant time, $O(1)$.

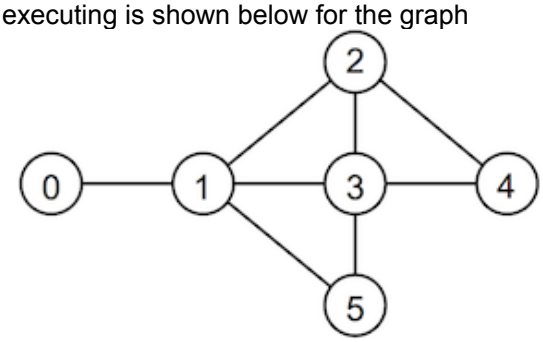
   c. Implement your algorithms in a program `graphAnalyser.c` that

      1. builds a graph from user input:
         - first, the user is prompted for the number of vertices
         - then, the user is repeatedly asked to input an egde by entering a "from" vertex followed by a "to" vertex
         - until any non-numeric character(s) are entered
      2. computes and outputs the minimum and maximum degree of vertices in the graph
      3. prints all vertices of minimum degree in ascending order, followed by all vertices of maximum degree in ascending order
      4. displays all 3-cliques of the graph in ascending order.

      Your program should use the Graph ADT from the lecture (`Graph.h` and `Graph.c`). These files should not be changed.

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

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




```
prompt$ ./graphAnalyser
Enter the number of vertices: 6
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): 2
Enter an edge (from): 1
Enter an edge (to): 3
Enter an edge (from): 3
Enter an edge (to): 4
Enter an edge (from): 1
Enter an edge (to): 5
Enter an edge (from): 5
Enter an edge (to): 3
Enter an edge (from): 2
Enter an edge (to): 3
Enter an edge (from): done
Done.
Minimum degree: 1
Maximum degree: 4
Nodes of minimum degree:
0
Nodes of maximum degree:
1
3
Triangles:
1-2-3
1-3-5
2-3-4
```

Note that any non-numeric data can be used to 'finish' the interaction.

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

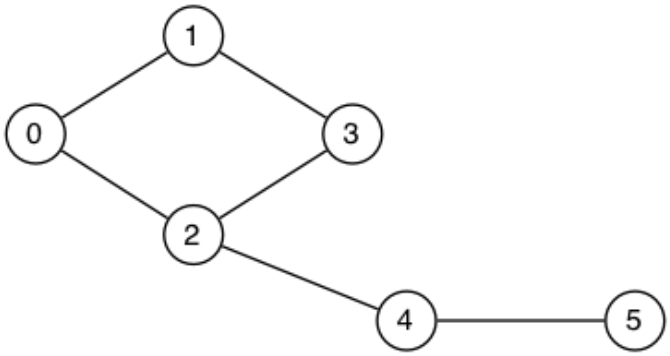
```
prompt$ 9024 dryrun graphAnalyser
```

Please ensure that your program output follows exactly the format shown in the sample interaction above. In particular, the vertices of minimum and maximum degree and the 3-cliques should be printed in ascending order.

3. (Graph representations)

Show how the following graph would be represented by

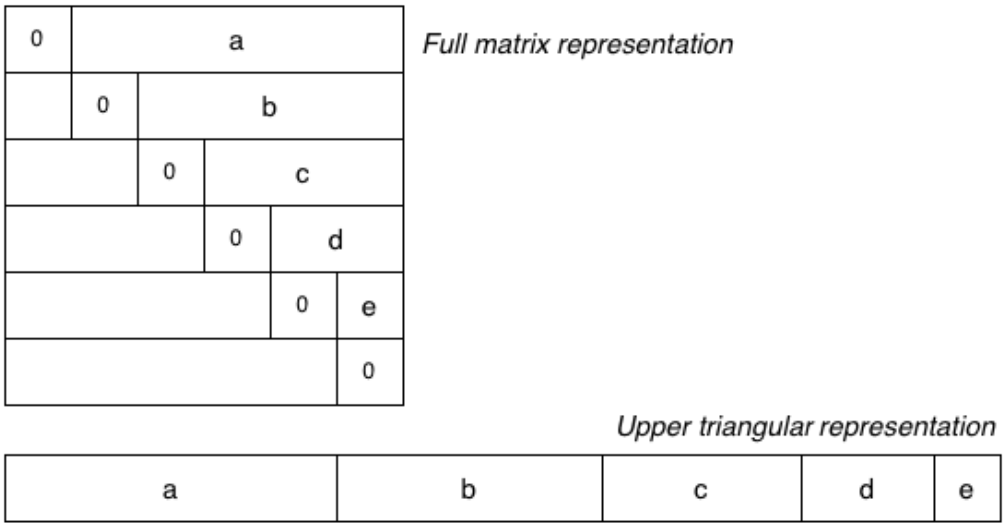
i. an adjacency matrix representation (V×V matrix with each edge represented twice)
ii. an adjacency list representation (where each edge appears in two lists, one for *v* and one for *w*)



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 roughly the V:E ratio at which it is more storage efficient to use an adjacency matrix representation vs the adjacency list representation.

   For the purposes of the analysis, ignore the cost of storing the `GraphRep` structure. Assume that: each pointer is 8 bytes long, a `Vertex` value is 4 bytes, a linked-list *node* is 16 bytes long and that 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 usage 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:



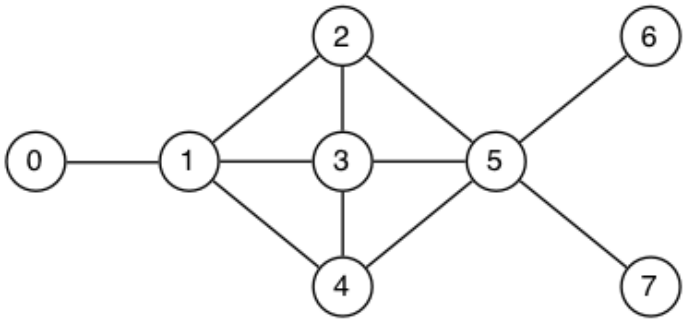
   The *V*×*V* matrix has been replaced by a single 1-dimensional array `g.edges[]` containing just the "useful" parts of the matrix.

   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`.

4. (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



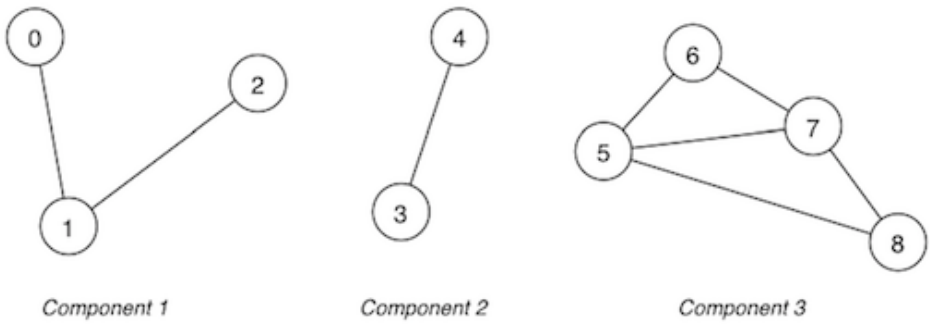
Assume the use of a stack for depth-first search (DFS) and a queue for breadth-first search (BFS), respectively. Show the state of the stack or queue explicitly in each step. When choosing which neighbour to visit next, always choose the smallest unvisited neighbour.

5. (Cycle check)

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

   b. Write a C program `cycleCheck.c` that implements your solution to check whether a graph has a cycle. The graph should be built from user input in the same way as in exercise 2. Your program should use the Graph ADT from the lecture (`Graph.h` and `Graph.c`). These files should not be changed.

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




```
prompt$ ./cycleCheck
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
Done.
The graph has a cycle.
```

      If the graph has no cycle, then the output should be:

```
prompt$ ./cycleCheck
Enter the number of vertices: 3
```

```
Enter an edge (from): 0
Enter an edge (to): 1
Enter an edge (from): #
Done.
The graph is acyclic.
```

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

To test your program you can execute the `dryrun` *program that corresponds to this exercise. It expects to find a program named* `cycleCheck.c` *in the current directory. You can use autotest as follows:*

```
prompt$ 9024 dryrun cycleCheck
```

Note: Please ensure that your output follows exactly the format shown above.

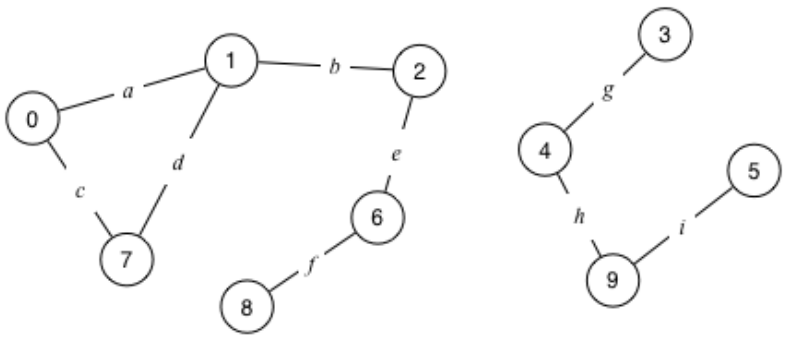
6. (Connected components)

   a. 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:



Component 0 *Component 1*

   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

   b. 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
```

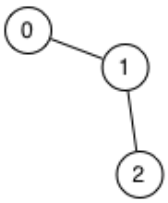
```
Output new empty graph

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
```

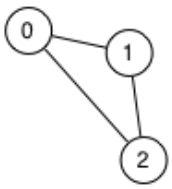
Modify the pseudocode for edge insertion and edge removal from the lecture to maintain the two new fields.

7. (Hamiltonian/Euler paths and circuits)

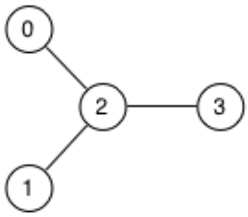
   a. Identify any Hamiltonian/Euler paths/circuits in the following graphs:



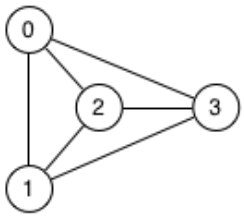
Graph 1



Graph 2

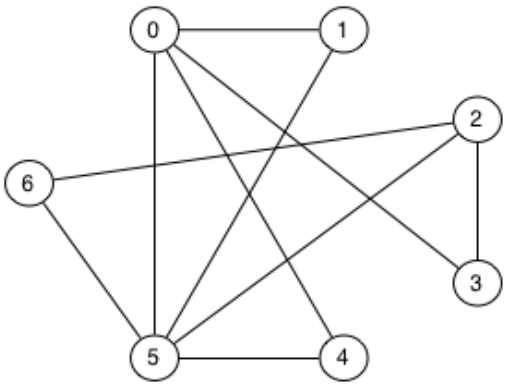


Graph 3



Graph 4

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



8. **Challenge Exercise**

   Extend your program from exercise 2 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.

Assessment

After you've solved the exercises, go to COMP9024 19T3 Quiz Week 5 to answer 5 quiz questions on this week's problem set (Exercises 1-7 only) and lecture.

The quiz is worth 2 marks.

The deadline for submitting your quiz answers is **Monday, 21 October 11:00:00am**.

Please continue to respect the **quiz rules**:

Do …

- use your own best judgement to understand & solve a question
- discuss quizzes on the forum only **after** the deadline on Monday

Do not …

- post specific questions about the quiz **before** the Monday deadline
- agonise too much about a question that you find too difficult