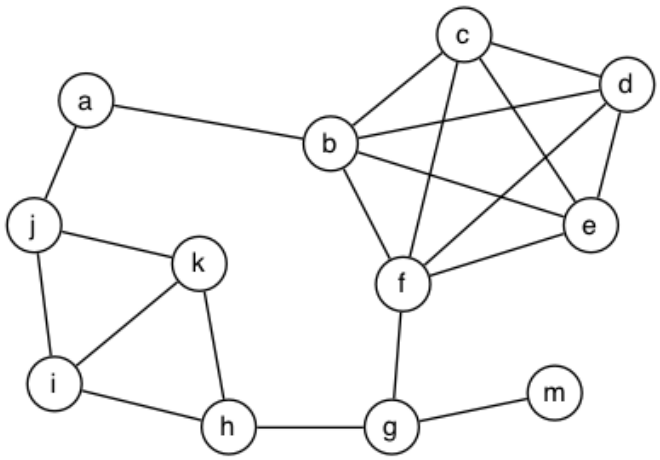
Week 06 Problem Set
Graph Data Structures and Search

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

   Answer:

   a. path
      - smallest: any path with one edge (e.g. a-b or g-m)
      - largest: some path including all nodes (e.g. m-g-h-k-i-j-a-b-c-d-e-f)

   b. cycle
      - smallest: need at least 3 nodes (e.g. i-j-k-i or h-i-k-h)
      - largest: path including most nodes (e.g. g-h-k-i-j-a-b-c-d-e-f-g) (can't involve m)

   c. spanning tree
      - smallest: any spanning tree must include all nodes (the largest path above is an example)
      - largest: same

   d. vertex degree
      - smallest: there is a node that has degree 1 (vertex m)
      - largest: in this graph, 5 (b or f)

   e. clique
      - smallest: any vertex by itself is a clique of size 1
      - largest: this graph has a clique of size 5 (nodes b,c,d,e,f)

2. (Graph properties)

   a. Write pseudocode for computing

      - the minimum and maximum vertex degree
      - all 3-cliques ("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, $O(1)$.

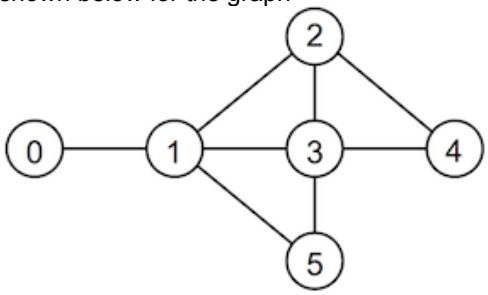
   c. Implement your algorithms in a program `graphAnalyser.c` that

      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
Finished.
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 the problem set and week. It expects to find a program named* `graphAnalyser.c` *in the current directory. You can use dryrun as follows:*

```
prompt$ ~cs9024/bin/dryrun prob06
```

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.

Answer:

The following algorithm uses two nested loops to compute the degree of each vertex. Hence its asymptotic running time is $O(n^2)$.

```
MinMaxDegree(g):
   Input  graph g
   Output minimum and maximum vertex degree in g

   min=#vertices(g)-1, max=0
   for all vertices v∈g do
      deg[v]=0
      for all vertices w∈g, v≠w do
         if v,w adjacent in g then
            deg[v]=deg[v]+1
         end if
      end for
      if deg[v]<min then
         min=deg[v]
```

```
      end if
      if deg[v]>max then
         max=deg[v]
      end if
   end for
   return min,max
```

The following algorithm uses three nested loops to print all 3-cliques in order. Hence its asymptotic running time is $O(n^3)$.

```
show3Cliques(g):
   Input graph g of n vertices numbered 0..n-1

   for all i=0..n-3 do
      for all j=i+1..n-2 do
         if i,j adjacent in g then
            for all k=j+1..n-1 do
               if i,k adjacent in g and j,k adjacent in g then
                  print i"-"j"-"k
               end if
            end for
          end if
      end for
   end for
```

Sample `graphAnalyser.c`:

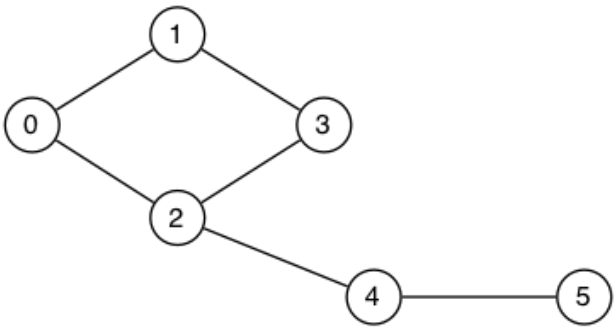
```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "Graph.h"

#define MAX_NODES 1000

// determine minimum and maximum degree of graph g with n vertices
// and output all nodes of those degrees
void MinMaxDegree(Graph g) {
   int degree[MAX_NODES];
   int nV = numOfVertices(g);
   int mindegree = nV-1;
   int maxdegree = 0;
   int v, w;
   for (v = 0; v < nV; v++) {
      degree[v] = 0;
      for (w = 0; w < nV; w++) {
         if (adjacent(g,v,w))
            degree[v]++;
      }
      if (degree[v] < mindegree)
         mindegree = degree[v];
      if (degree[v] > maxdegree)
         maxdegree = degree[v];
   }
   printf("Minimum degree: %d\n", mindegree);
   printf("Maximum degree: %d\n", maxdegree);

   printf("Nodes of minimum degree:\n");
   for (v = 0; v < nV; v++) {
      if (degree[v] == mindegree)
         printf("%d\n", v);
   }
   printf("Nodes of maximum degree:\n");
   for (v = 0; v < nV; v++) {
      if (degree[v] == maxdegree)
         printf("%d\n", v);
   }
}

// show all 3-cliques of graph g
void Show3Cliques(Graph g) {
   int i, j, k;
   int nV = numOfVertices(g);

   printf("Triangles:\n");
   for (i = 0; i < nV-2; i++)
      for (j = i+1; j < nV-1; j++)
         if (adjacent(g,i,j))
            for (k = j+1; k < nV; k++)
```

```
                if (adjacent(g,i,k) && adjacent(g,j,k))
                   printf("%d-%d-%d\n", i, j, k);
}

int main(void) {
   Edge e;
   int n;

   printf("Enter the number of vertices: ");
   scanf("%d", &n);
   Graph g = newGraph(n);

   printf("Enter an edge (from): ");
   while (scanf("%d", &e.v) == 1) {
      printf("Enter an edge (to): ");
      scanf("%d", &e.w);
      insertEdge(g, e);
      printf("Enter an edge (from): ");
   }
   printf("Finished.\n");

   MinMaxDegree(g);
   Show3Cliques(g);
   freeGraph(g);

   return 0;
}
```

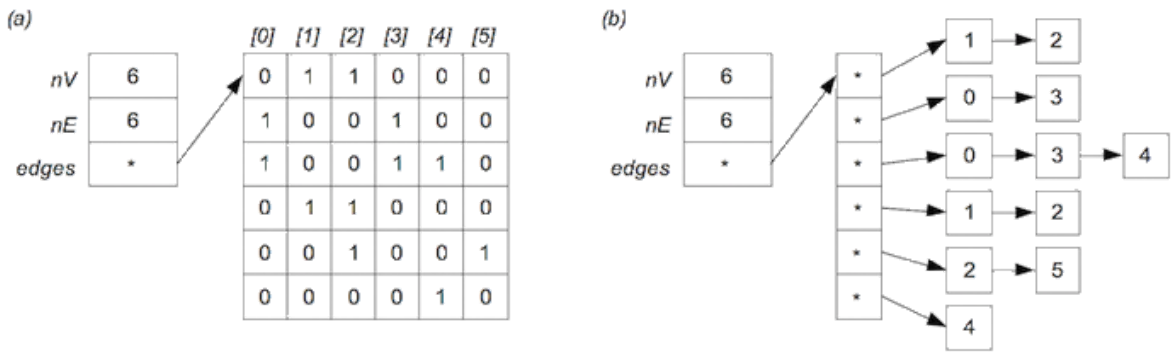
3. (Graph representations)

Show how the following graph would be represented by

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



Answer:

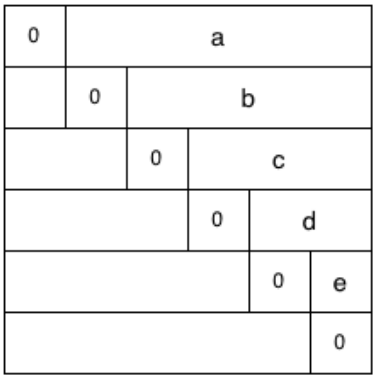


4. (Storage costs)

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 adjacenty list representation.

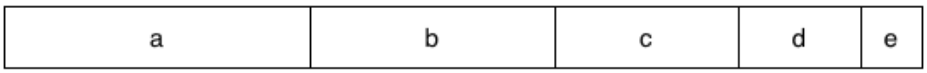
   For the purposes of the analysis, ignore the cost of storing the `GraphRep` structure. Assume that: each pointer is 4 bytes long, a `Vertex` value is 4 bytes, a linked-list *node* is 8 bytes long and that the adjacency matrix is a complete *V*×*V* matrix. Assume also that each adjacency matrix element 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:



Full matrix representation

Upper triangular representation



The $V \times V$ 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`.

Answer:

a. The adjacency matrix representation always requires a $V \times V$ 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 $V \cdot 4 + V^2$ bytes.

The adjacency list representation requires an array of V pointers (the start of each list), with each being 4 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 8 bytes (vertex+pointer), this gives a size of $V \cdot 4 + 8 \cdot 2 \cdot E$. The total storage is thus $V \cdot 4 + 16 \cdot E$.

Since both representations involve V pointers, the difference is based on V^2 vs $16E$. So, if $16E < V^2$ (or, equivalently, $E < V^2/16$), then the adjacency list representation will be more storage-efficient. Conversely, if $E > V^2/16$, then the adjacency matrix representation will be more storage-efficient.

To pick a concrete example, if $V=20$ and if we have less than 25 edges (= 20·20/16), then the adjacency list will be more storage-efficient, otherwise the adjacency matrix will be at least as storage-efficient.

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

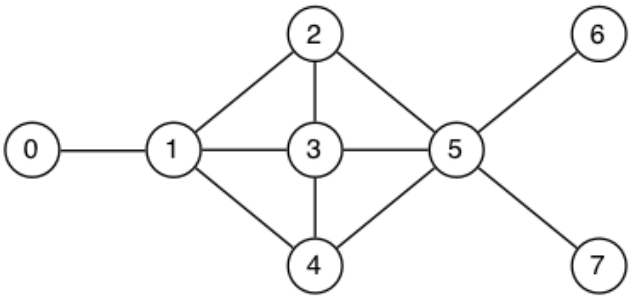
```
adjacent(g,v,w):
   Input  graph g in upper-triangle matrix representation
          v, w vertices such that v≠w
   Output true if v and w adjacent in g, false otherwise

   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 $v < w$).

5. (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.

Answer:

a. DFS starting at 0:

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

b. DFS starting at 3:

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

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          -
```

6. **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.

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.

```
int maxCliqueSize(Graph g, int v, int clique[], int k) {
//
// g         graph
// v         next vertex to consider
// clique[]  some subset of nodes 0..v-1 that forms a clique
// k         size of that clique
//
// returns size of largest complete subgraph of g that extends clique[] with nodes from v..nV-1
//
   if (v == numOfVertices(g) {                    // no more vertex to consider
      return k;
   } else {
      int k1 = maxCliqueSize(g, v+1, clique, k);   /* find largest complete subgraph that
                                                      extends clique[] without considering v */
      int i;
      for (i = 0; i < k; i++)                     // check if v can be added to clique[]:
         if (!adjacent(g, v, clique[i]))          // if v not adjacent to all nodes in clique[]
            return k1;                            // => return largest clique size without v

      clique[k] = v;                              // add v to clique[]
      int k2 = maxCliqueSize(g, v+1, clique, k+1); // find largest clique extending clique[]+v
      if (k2 > k1)
         return k2;
      else
         return k1;
}
```

To call this recursive function on a graph g with nV vertices:

```
int *clique = malloc(nV * sizeof(int));           // allocate memory for an array of vertices
int m = maxCliqueSize(g, 0, clique, 0);           // start at vertex 0 with clique of size 0
free(clique);
```

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