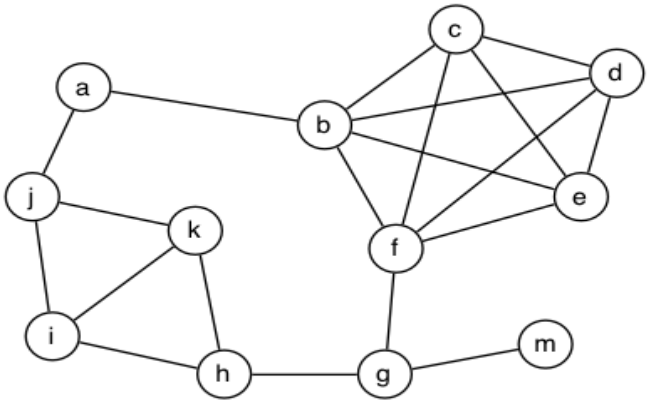
Homework Nine

Q1. For the following graph,



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

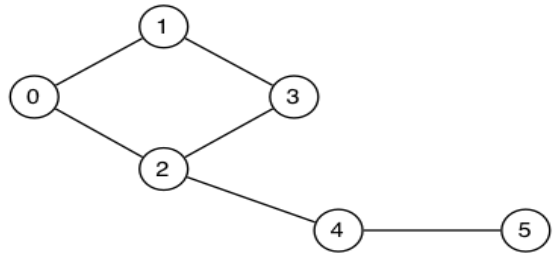
1. path
2. cycle
3. spanning tree
4. vertex degree
5. clique

Solution:

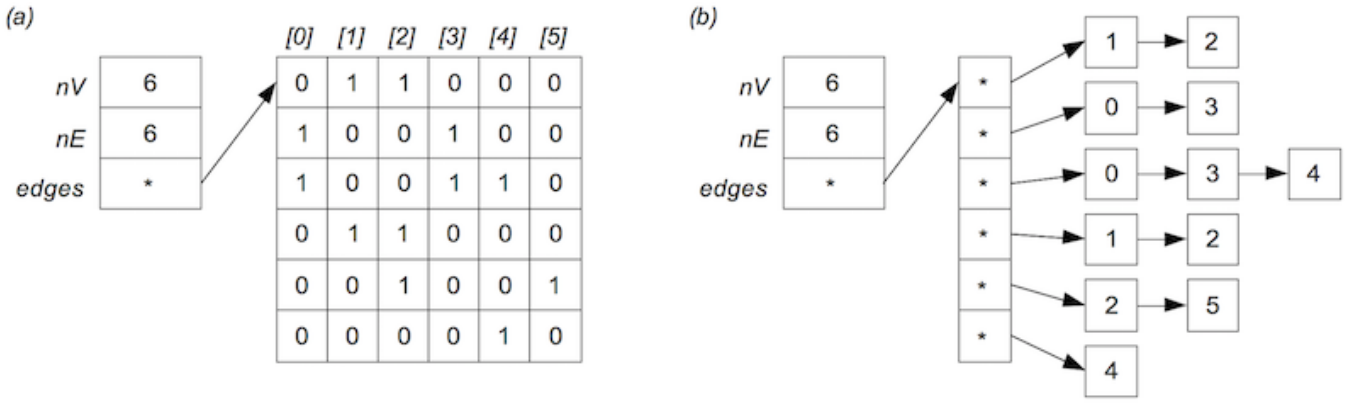
1. path
   smallest: any path with one edge (e.g. a-b or g-m)
   largest: some path including all nodes (e.g. m-g-h-k-i-j-a-b-c-d-e-f)
2. cycle
   smallest: need at least 3 nodes (e.g. i-j-k-i or h-i-k-h)
   largest: path including most nodes (e.g. g-h-k-i-j-a-b-c-d-e-f-g) (can't involve m)
3. spanning tree
   smallest: any spanning tree must include all nodes (the largest path above is an example)
   largest: same
4. vertex degree
   smallest: there is a node that has degree 1 (vertex m)
   largest: in this graph, 5 (b or f)
5. 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)

Q2. Show how the following graph would be represented by

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



Solution:



Q3. 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 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.)

Solution: 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 $V \cdot 4 + V^2$ bytes.

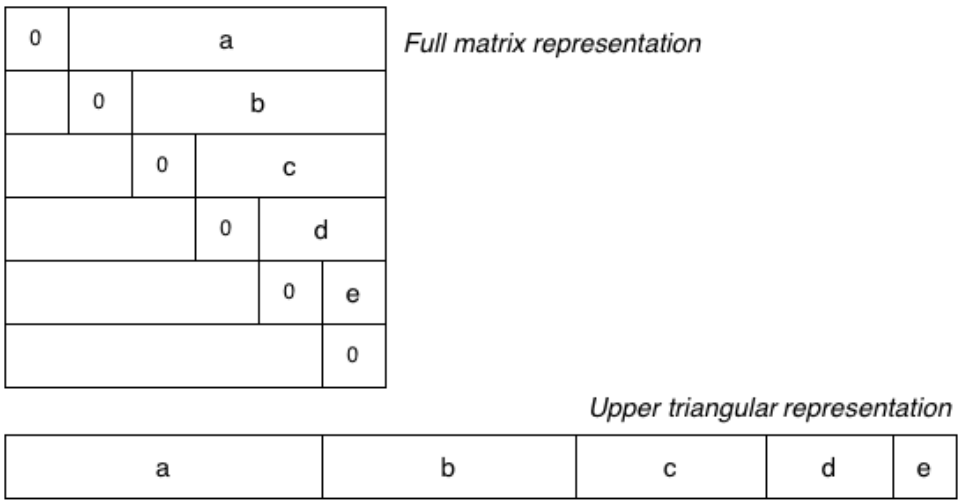
The adjacency list representation requires an array of V pointers (the start of each list), with each being 4 bytes long, and then one list node for each edge in each list. The total number of edge nodes is 2E (each edge (v,w)is stored twice, once in the list for v and once in the list for w).

Since each node requires 8 bytes (vertex+pointer), this gives a size of V·4+8·2·E. The total storage is thus V·4+16·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.

Q4. The standard adjacency matrix representation for a graph uses a full n×n 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 n×n 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.

Solution:

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

Algorithm 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 )
       swap v and w;       // to ensure v<w

    chunksize=g.nV-1;
    offset=0;
    for  i=0..v-1 do
       {
         offset=offset+chunksize;
         chunksize=chunksize-1;
       }
    offset=offset+w-v-1;
    if ( g.edges[offset]=1)
          return true;
    else   return false;

  }
```

Alternatively, you can compute the overall offset directly via the formula $(nV-1)+(nV-2)+...+(nV-v)+(w-v-1)=nvV-v(v+1)/2+w-v-1$ (assuming that v<w).

Q5. Write an algorithm in pseudocode for computing the minimum and maximum vertex degree. Your algorithm should be representation-independent; the only function you should use is to check if two vertices $v,w \in \{0,\cdots n-1\}$ are adjacent in g. Determine the time complexity of your algorithm. Assume that the adjacency check takes constant time, O(1).

Solution:

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

```
Algorithm MinMaxDegree(g,n)
Input:  graph g of n vertices
Output: minimum and maximum vertex degree in g
 {
   min=n-1;
   max=0;
   for all vertices v∈g do
     { deg[v]=0;
       for all vertices w∈g, v≠w do
           if  v and w are adjacent in g
             deg[v]=deg[v]+1;
       if ( deg[v]<min )
          min=deg[v];
       if ( deg[v]>max )
```

```
            max=deg[v];
        }
    return min,max;
}
```

Q6. Write an algorithm in pseudocode for computing all 3-cliques ("triangles") of a graph g with n vertices. Your algorithm should be representation-independent; the only function you should use is to check if two vertices v,w∈{0,⋯n-1} are adjacent in g. Determine the time complexity of your algorithm. Assume that the adjacency check takes constant time, O(1).

Solution:

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

```
Algorithm show3Cliques(g,n):
Input: graph g of n vertices numbered 0..n-1
Output: all 3-cliques
{
  for all i=0..n-3 do
      for all j=i+1..n-2 do
          if i and j are adjacent in g
              for all k=j+1..n-1 do
                  if i and k adjacent in g and j and k are adjacent in g
                     print i"-"j"-"k;
}
```