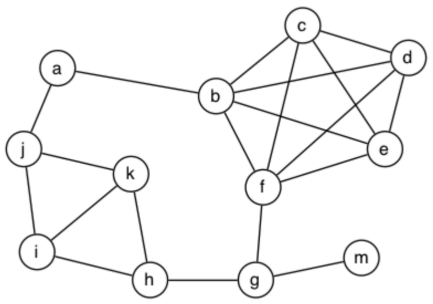
Week 4 Problem Set

Graph Data Structures and Graph Search

1. (Graph properties)

   For the graph



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

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

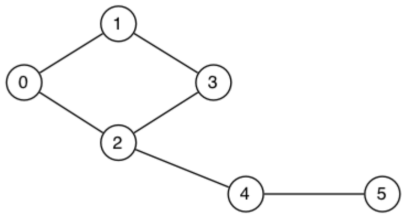
   Answer:

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

   a. Show how the following graph would be represented by

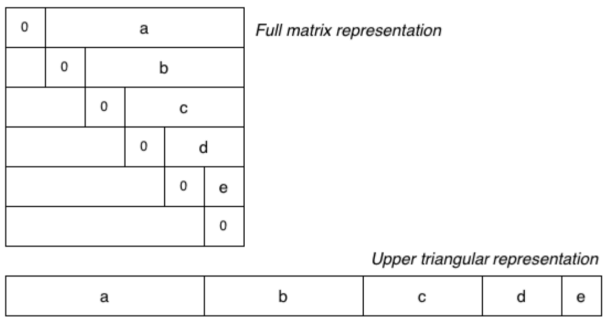
      i. an adjacency matrix representation (V×V matrix with each edge represented twice)
      ii. an adjacency list representation (where each edge appears in two lists, one for *v* and one for *w*)



   b. 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.)

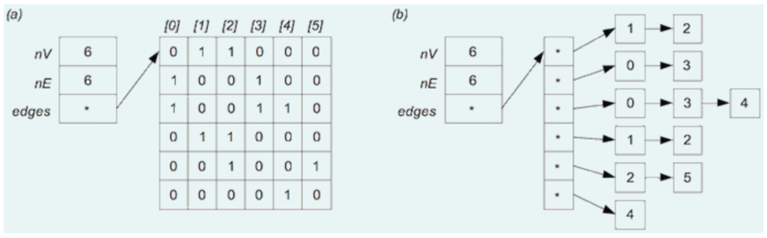
   c. The standard adjacency matrix representation for a graph uses a full *V*×*V* matrix and stores each edge twice (at [v,w] and [w,v]). This consumes a lot of space, and wastes a lot of space when the graph is sparse. One way to use less space is to store just the upper (or lower) triangular part of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

Answer:

a. 

b. 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 $V\cdot8+V^2$ 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 $V\cdot8+16\cdot2\cdot E$. The total storage is thus $V\cdot8+32\cdot E$.

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

To pick a concrete example, if *V=25* and if we have 19 or fewer edges ($25\cdot25/32 = 19.53$), then the adjacency list will be more storage-efficient, otherwise the adjacency matrix will be more storage-efficient.

c. 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
```

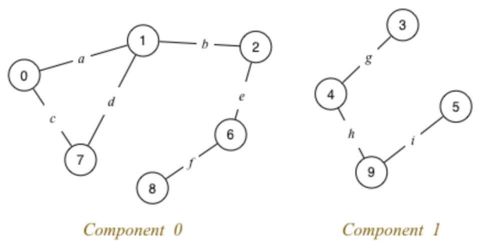
Alternatively, you can compute the overall offset directly via the formula $(nV-1)+(nV-2)+\ldots+(nV-v)+(w-v-1)=\frac{v}{2}(2\cdot nV-v-1)+(w-v-1)$ (assuming that *v < w*).

3. (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:



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.

Answer:

a. 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

b. Inserting an edge may reduce the number of connected components:

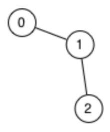
```
insertEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if g.edges[v][w]=0 then                // (v,w) not in graph
|  |  g.edges[v][w]=1, g.edges[w][v]=1   // set to true
|  |  g.nE=g.nE+1
|  |  if g.cc[v]≠g.cc[w] then            // v,w in different components?
|  |  |  c=min{g.cc[v],g.cc[w]}          // ⇒ 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:

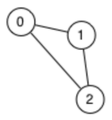
```
removeEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if g.edges[v][w]≠0 then                // (v,w) in graph
|  |  g.edges[v][w]=0, g.edges[w][v]=0   // set to false
|  |  if not hasPath(g,v,w) then         // v,w no longer connected?
|  |     dfsNewComponent(g,v,g.nC)       // ⇒ put v + connected 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
```

4. (Hamiltonian/Euler paths and circuits)

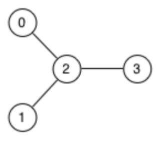
a. Identify any Hamiltonian/Euler paths/circuits in the following graphs:



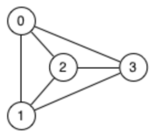
Graph 1



Graph 2

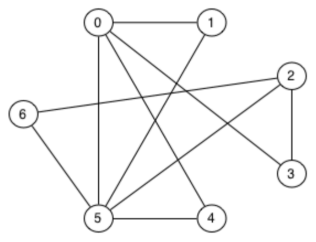


Graph 3



Graph 4

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



Answer:

a. 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.

b. 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.

5. **Challenge Exercise**

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.

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.