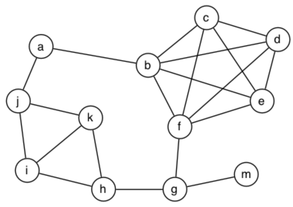
Week 4 Problem Set

Graph Data Structures and Graph Search

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

1. (Graph properties)

   For the graph



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

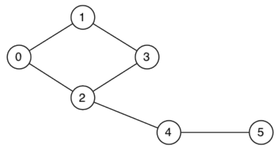
   a. simple path
   b. cycle
   c. spanning tree
   d. vertex degree
   e. clique

[show answer]

2. (Graph representations)

   a. Show how the following graph would be represented by

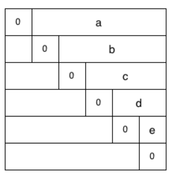
      i. an adjacency matrix representation (V×V matrix with each edge represented twice)
      ii. an adjacency list representation (where each edge appears in two lists, one for *v* and one for *w*)



   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:



Full matrix representation

Upper triangular representation

a	b	c	d	e

      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`.

[show answer]

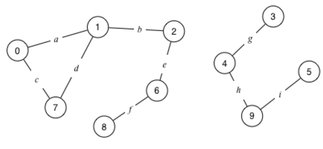
3. (Connected components)

   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:



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:

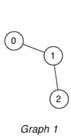
```
newGraph(V):
|  Input  number of nodes V
|  Output new empty graph
|
|  g.nV=V, g.nE=0, g.nC=V
|  allocate memory for g.edges[][]
|  for all i=0..V-1 do
|     g.cc[i]=i
|     for all j=0..V-1 do
|        g.edges[i][j]=0
|     end for
|  end for
|  return g
```

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

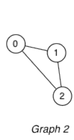
[show answer]

4. (Hamiltonian/Euler paths and circuits)

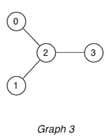
   a. Identify any Hamiltonian/Euler paths/circuits in the following graphs:



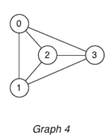
Graph 1



Graph 2

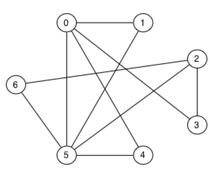


Graph 3



Graph 4

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



[show answer]

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.

[show answer]