Questions

1. Let A be the adjacency matrix of a *weighted graph*, G, with the columns and rows labelled, in order, by the vertices of the set $V = \{v_0, v_1, v_2, v_3, v_4, v_5\}$ so the first row of the matrix represents the adjacencies of vertex v_0, the second row those of vertex v_1, etc. Note that there are some non-zero diagonals which means that there are some edges of the form (v, v). Also, an entry with weight 0 means there is no edge.

$$A = \begin{bmatrix} 0 & 1 & 3 & 0 & 0 & 2 \\ 1 & 1 & 0 & 4 & 5 & 0 \\ 3 & 0 & 0 & 5 & 4 & 0 \\ 0 & 4 & 5 & 2 & 0 & 6 \\ 0 & 5 & 4 & 0 & 0 & 2 \\ 0 & 0 & 0 & 6 & 2 & 3 \end{bmatrix}$$

   (a) Write down the adjacency lists for the graph, such that the edges in each list are ordered *by increasing weight*. (If two edges have the same weight, then order them by their index order.)

   (b) Perform a (preorder) depth first search on G, beginning at vertex v_0. Use the ordering in the adjacency list to determine the ordering of vertices visited. Give solutions both using the recursive algorithm and using a stack.

   (c) Perform a breadth first search of the graph, G, again beginning at v_0 and using the ordering in the adjacency list.

2. Suppose a graph has n vertices. We informally say that the graph is *dense* if number of edges is close to n^2 and we say the graph is *sparse* if number of edges is close to n.

   Would you use the adjacency list structure or the adjacency matrix structure in each of the following cases? Justify your choice.

   (a) The graph has 10,000 vertices and 20,000 edges and it is important to use as little space as possible.

   (b) The graph has 10,000 vertices and 20,000,000 edges, and it is important to use as little space as possible.

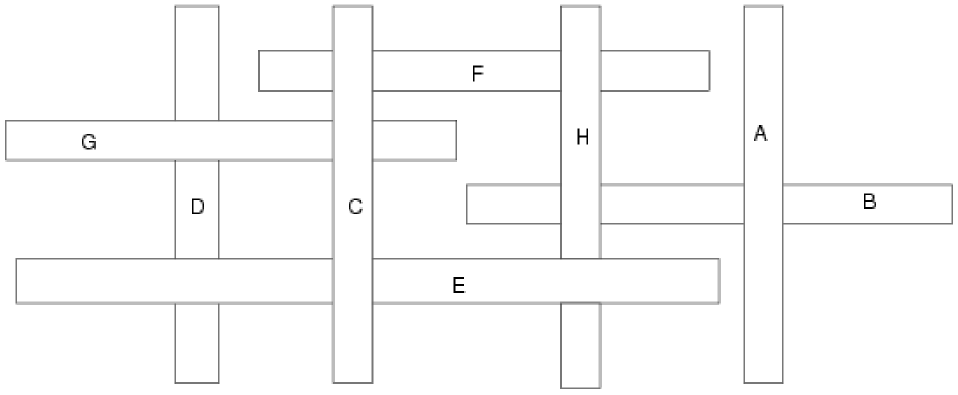
   (c) You need to answer the query `areAdjacent`(u, v) as quickly as possible, no matter how much space you use. (Two vertices u and v are adjacent if either (u, v) or (v, u) is an edge in the graph.)

   (d) You need to perform operation `insertVertex`(v).

   (e) You need to perform operation `removeVertex`(v).

3. Explain why the depth first traversal runs in worst case $O(n^2)$ time on an n-vertex graph that is represented with an adjacency matrix structure.

4. The figure below shows a set of rectangles whose overlap relationships can be represented with a directed graph. Each rectangle is represented by one vertex, and there is a directed edge whenever one rectangle overlaps another rectangle. For example, there is an edge `(A,B)` but no edge `(B,A)`.



   (a) Give the adjacency list for this graph. *The vertices must be ordered alphabetically.*

   (b) Give the ordering of vertices visited in a *breadth first* traversal of the graph, starting from vertex `C`. *Show the BFT tree.*

   NOTE: In this question and the next, there is only one answer since you must order the vertices alphabetically.

   (c) Give the ordering of vertices visited in a *recursive preorder depth first search* traversal, starting from vertex `C`. *Show the DFT tree.*

   (d) Give a *new* example having four rectangles `I,J,K,L`, such that corresponding graph contains a *cycle*. Draw the rectangles and the graph.

5. The recursive depth first graph traversal algorithm that I presented in class was *preorder* in that a vertices is visited before its adjacent vertex. (By this, I mean that if the parameter of the algorithm is v and edge w is in the adjacency list of v, then the algorithm visits v before w.)

   How could we rewrite the recursive depth first graph algorithm so that it is post-order? For example, suppose that we were to run the algorithm on a rooted tree rather than a graph; we would like the algorithm to visit the vertices in a postorder way.

   Hint: distinguish the idea of reaching a node versus "visiting" a node, by having two variables: `reached` and `visited`. You can let *reached* play the role that visited did in the lecture, and let `visited play a different role`. If you have no idea what I mean, then look at the solution.

6. [**Updated after lecture 32 in response to question**]

   Suppose that in the graph traversal using stack algorithm, we visit after popping as below.

```
graphTraversalUsingStack(v){
  initialize empty stack s
  s.push(v)
  while (s is not empty) {
     u =  s.pop()
     visit u
     for each w in u.adjList{
         if (!w.visited){
            s.push(w)
     }
}
```

   What is the problem here?

   As an example, consider a graph with four vertices `a, b, c, d` and all pairwise edges between them and in both directions. Consider how the stack evolves over time.

Answers

1. (a) $v_0 - v_1, v_5, v_2$
   $v_1 - v_0, v_1, v_3, v_4$
   $v_2 - v_0, v_4, v_3$
   $v_3 - v_3, v_1, v_2, v_5$
   $v_4 - v_5, v_2, v_1$
   $v_5 - v_4, v_5, v_3$

   (b) depth first traversal using recursion: $v_0, v_1, v_3, v_2, v_4, v_5$
   depth first traversal using stack: $v_0, v_1, v_5, v_2, v_4, v_3$

   (c) breath first traversal: $v_0, v_1, v_5, v_2, v_3, v_4$

2. (a) An adjacency list would be better. Why? The adjacency lists would have about 20,000 nodes, whereas the adjacency matrix would require $10,000 \times 10,000 = 100,000,000$ booleans. A boolean is still typically represented with a byte (and the value would be either 0 or 1), so the matrix would require more space. Even if the booleans were packed more efficiently – 8 booleans packed into a byte – it would still require over $10,000,000$ bytes.

   (b) There is no clear winner. The exact space usage of the two structures depends on the implementation details, in particular, of a node. It also depends on how the booleans are represented (see (a)).

   (c) Adjacency matrix. To answer `areAdjacent`(i,j) one need only read off matrix entries (i, j) and (j, i), a constant time i.e. O(1) operation. In an adjacency list, one would would need to search along row i to find j, and linear search is $O(n)$.

   (d) The adjacency list structure is much better for `insertVertex`. If you used a hash table to represent the vertices – each entry being a (name,vertex) pair – then the vertex could be added to the hash table in time $O(1)$. Note that if you are just adding a vertex, then the adjacency list for that vertex is empty.

   For an adjacency matrix representation of the edges, you would need to build a new (larger by 1 row and 1 column) matrix and copy all entries from old to new. This would be $O(n^2)$.

   (e) For `removeVertex()`, if you used a hash table to represent the vertices, then the vertex could be removed from the hash table in time $O(1)$, and the edges *from* the vertex could be removed in time $O(1)$ also, just by deleting the adjacency list for that vertex.

   However, you also want to remove the edges *to* the vertex. This would require checking all the other adjacency lists and in the worst case it would take time proportional to $O(n + m)$ where m is the number of edges in the graph. In the case of a dense graph m can be as large as n^2.

   If you use an adjacency matrix, and you don't want rows and columns to continue representing edges that involved deleted vertices, then you would need to remake the matrix every time you do a remove. Just like in (d), this would be expensive: $O(n^2)$.

3. A depth first traversal will consider every reachable vertex of the graph, eventually. For each vertex that it considers, it must consider the set of neighbours of the vertex – in order to do

so, it must examine the entire row of the adjacency matrix corresponding to the vertex. Thus, for each of n rows/vertices, it must consider the $n-1$ non-diagonal entries, which makes for $O(n^2)$ operations.

4.

```
a)   A - B
     B -
     C - E,F,G
     D -
     E - D,H
     F -
     G - D
     H - B,F

b)           C      ordering is  CEFGDHB
           / | \
          E  F  G
         / \
        D  H
           |
           B


c)           C      ordering is CEDHBFG
           /   \
          E     G
         / \
        D  H
          / \
         B   F

d)  Use the overlap method for the four (folding) top
    pieces of a packing box.  i.e.  (A,B), (B,C), (C,D), (D,A).

           |     |
         --|--A-----
           |     |
           D     B
           |     |
         -----C--|-
           |     |

    My apologies for the ASCII art.
```

5. We cannot just move the `v.visited = true` assignment after the `depthFirst` call because we could get stuck in an infinite loop, in the case of a cycle. The following will NOT work, for example, if the graph consists of a single cycle:

```
depthFirst(v){
   for each w such that (v,w) is in E
      if !w.visited
         depthFirst(w)
   v.visited = true
}
```

   To make a post-order traversal work, we need to distinguish between "reaching" a vertex and "visiting" a vertex. Reaching a vertex could just mean that we access the adjacency list of the vertex, whereas visiting it might mean that we do some computation on the vertex (as in our examples from trees). The following works – in the sense that it avoids the infinite loop from a cycle, and also allows us to visit a vertex after all the adjacent vertices have been visited.

```
depthFirst(v){
   v.reached = true
   for each w such that (v,w) is in E
      if !w.reached
         depthFirst(w)
   v.visited = true    //  done right before exiting
}
```

   For the above algorithm to work properly, the fields `visited` and `reached` should both be initialized to false.

6. Consider how the stack develops when we traverse from vertex `a`.
   Let the top of the stack be on the right.

```
STACK
------
a                  then pop a, visit it, and push b,c,d  giving...
b, c, d            then pop d, visit it, and push b, c  giving ...
b, c, b, c         then pop c, visit it, and push b giving ...
b, c, b, b         then pop b, visit it ....
```

   From there, we will just continue popping until the stack is empty.

   So the problem here isn't that we get into an infinite loop, but that we inelegantly push more than we need to.