DAA Assignment II

1A) Do the best case analysis for Brute force string matching algorithm for the following cases:

a) Pattern is found the text

b) Pattern is not found in the text

Size of string be 'n' and pattern be 'm'.

a) Best case found = first match in string.

Let string be T = SSSSSSSSSS… (n)
and pattern P = PPPP… (m)

For best case,

SSSSSSSSSS…
PPPPP…

Number of comparisons = m
Complexity = θ(m)

b) Best case not found = mismatch on the first character.

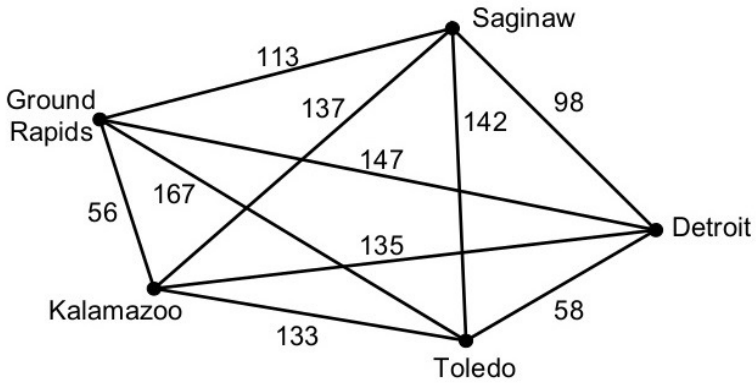
Number of comparisons = n - m
Complexity = θ(n - m) ≃ θ(n)

1B) Apply insertion sort to sort the list {E, X, A, M, P, L, E} in alphabetical order.

E | X A M P L E
E **X** | A M P L E
A E X | M P L E
A E **M** X | P L E
A E M **P** X | L E
A E **L** M P X | E
A E **E** L M P X |

E	X	A	M	P	L	E
E	**X**					
E	X	**A**				
A	E	X	**M**			
A	E	M	X	**P**		
A	E	M	P	X	**L**	
A	E	L	M	P	X	**E**
A	E	E	L	M	P	X

2) Following graph represents an instance of traveling salesman problem. Find the solution for that by exhaustive search method from the starting city Detroit.



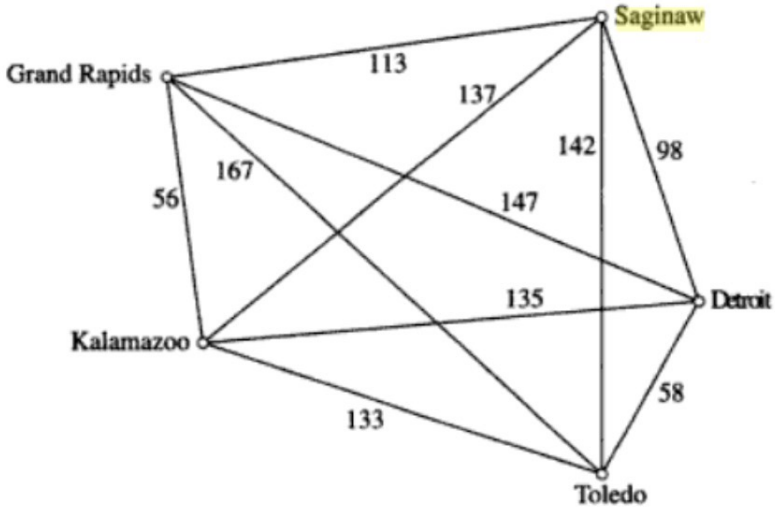
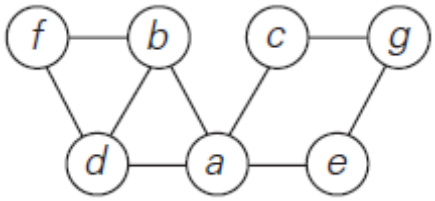


FIGURE 5 The Graph Showing the Distances between Five Cities.

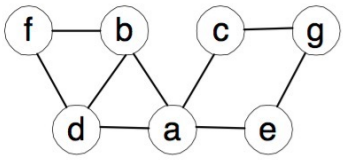
Route	*Total Distance (miles)*
Detroit–Toledo–Grand Rapids–Saginaw–Kalamazoo–Detroit	610
Detroit–Toledo–Grand Rapids–Kalamazoo–Saginaw–Detroit	516
Detroit–Toledo–Kalamazoo–Saginaw–Grand Rapids–Detroit	588
Detroit–Toledo–Kalamazoo–Grand Rapids–Saginaw–Detroit	458
Detroit–Toledo–Saginaw–Kalamazoo–Grand Rapids–Detroit	540
Detroit–Toledo–Saginaw–Grand Rapids–Kalamazoo–Detroit	504
Detroit–Saginaw–Toledo–Grand Rapids–Kalamazoo–Detroit	598
Detroit–Saginaw–Toledo–Kalamazoo–Grand Rapids–Detroit	576
Detroit–Saginaw–Kalamazoo–Toledo–Grand Rapids–Detroit	682
Detroit–Saginaw–Grand Rapids–Toledo–Kalamazoo–Detroit	646
Detroit–Grand Rapids–Saginaw–Toledo–Kalamazoo–Detroit	670
Detroit–Grand Rapids–Toledo–Saginaw–Kalamazoo–Detroit	728

3A) Consider the following graph:

Starting at vertex 'a' and resolving ties by the vertex alphabetical order, traverse the graph by depth-first search and construct the corresponding depth-first search tree. Give the order in which the vertices were reached for the first time (pushed onto the traversal stack) and the order in which the vertices became dead ends (popped off the stack).



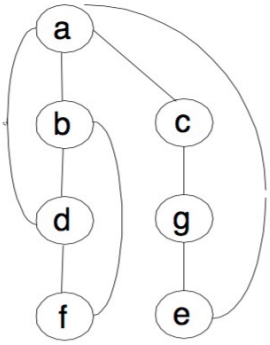
b. See below: (i) the graph; (ii) the traversal's stack (the first subscript number indicates the order in which the vertex was visited, i.e., pushed onto the stack, the second one indicates the order in which it became a dead-end, i.e., popped off the stack); (iii) the DFS tree (with the tree edges shown with solid lines and the back edges shown with dashed lines).



(i)

$f_{4,1}$ $e_{7,4}$
$d_{3,2}$ $g_{6,5}$
$b_{2,3}$ $c_{5,6}$
$a_{1,7}$

(ii)



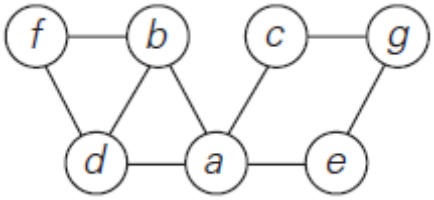
(iii)

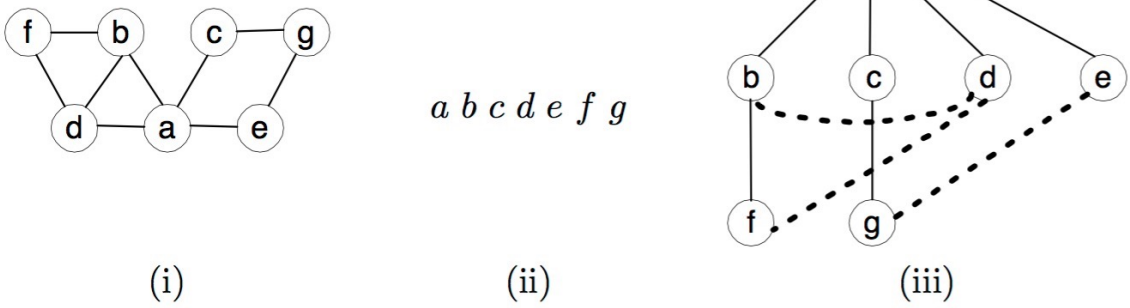
3B) Explain how to identify connected components of a graph by using a breadth-first Search.

Start a DFS (or BFS) traversal at an arbitrary vertex and mark the visited vertices with 1. By the time the traversal's stack (queue) becomes empty, all the vertices in the same connected component as the starting vertex, and only they, will have been marked with 1. If there are unvisited vertices left, restart the traversal at one of them and mark all the vertices being visited with 2, and so on until no unvisited vertices are left.

4A) Consider the following graph:

Starting at vertex 'a' and resolving ties by the vertex alphabetical order, traverse the graph by breadth-first search and construct the corresponding breadth-first search tree.





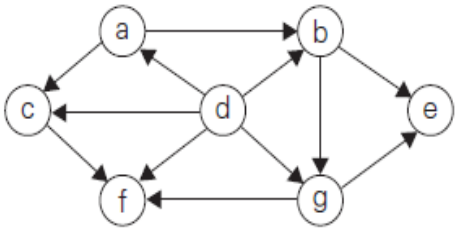
(i) the graph; (ii) the traversal's queue; (iii) the tree (the tree and cross edges are shown with solid and dotted lines, respectively).

4B) Explain how to identify connected components of a graph by using a depth-first search.

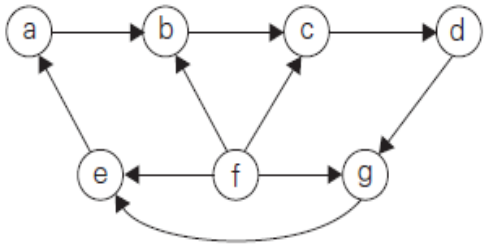
- For each vertex v in undirected graph G,
  - If v has no adjacent vertices, it is part of an undiscovered connected component. Execute DFS starting from v and mark all the vertices as adjacent to v.
  - Else, if v has an connected vertex, it is part of a connected component we've already discovered. Ignore v and move on to the next vertex.

Start a DFS (or BFS) traversal at an arbitrary vertex and mark the visited vertices with 1. By the time the traversal's stack (queue) becomes empty, all the vertices in the same connected component as the starting vertex, and only they, will have been marked with 1. If there are unvisited vertices left, restart the traversal at one of them and mark all the vertices being visited with 2, and so on until no unvisited vertices are left.

5A) Apply the DFS-based algorithm to solve the topological sorting problem for the following digraphs:

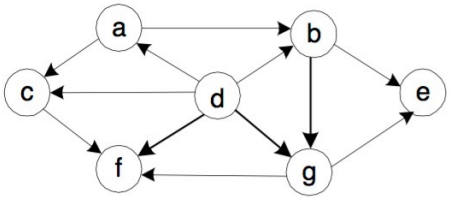


(a)



(b)

a. The digraph and the stack of its DFS traversal that starts at vertex a are given below:



f
e g
b c
a d

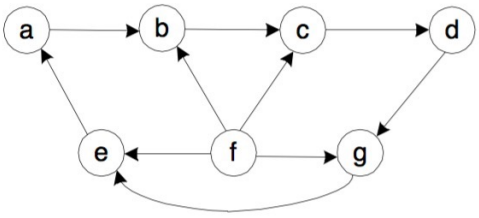
The vertices are popped off the stack in the following order:

$$e\ f\ g\ b\ c\ a\ d.$$

The topological sorting order obtained by reversing the list above is

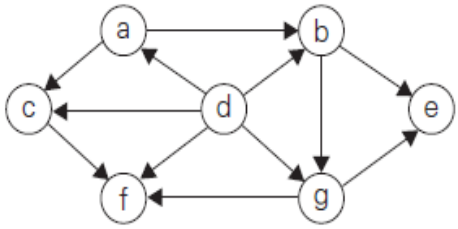
$$d\ a\ c\ b\ g\ f\ e.$$

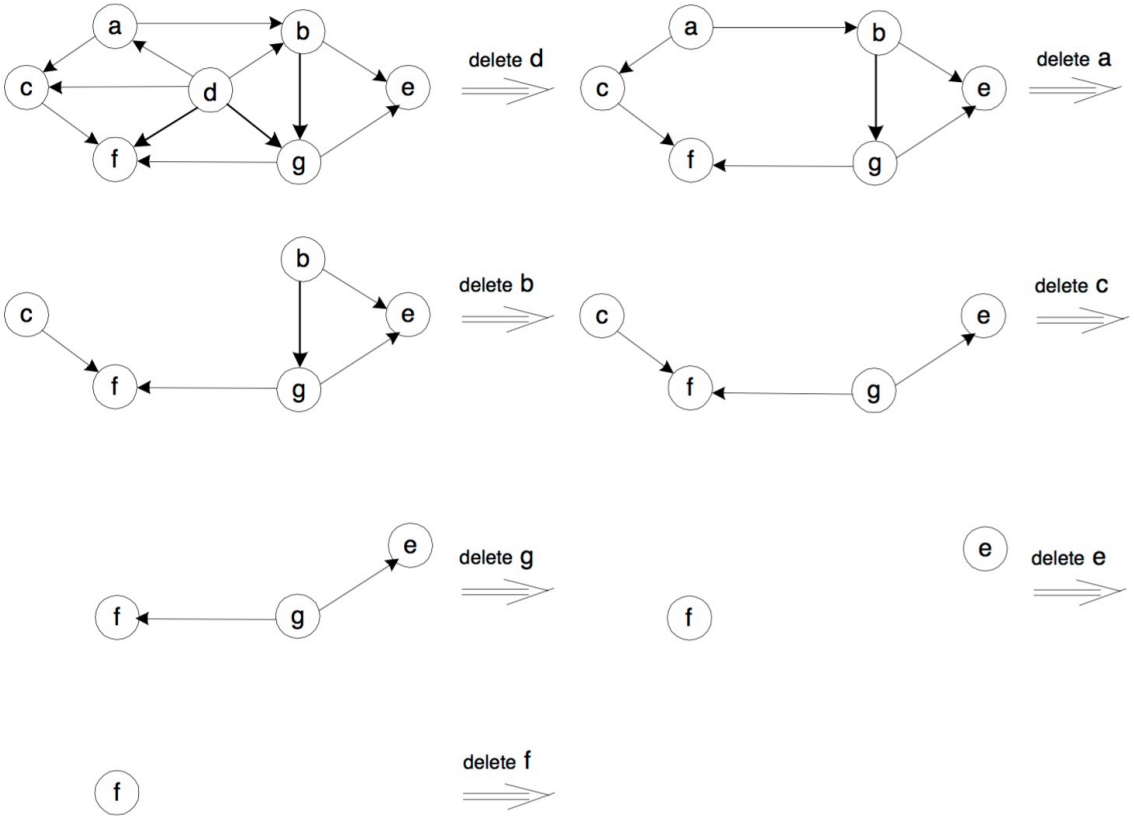
b. The digraph below is not a dag. Its DFS traversal that starts at a encounters a back edge from e to a:



e
g
d
c
b
a

5B) Apply the source-removal algorithm to solve the topological sorting problem for the following digraph:





The topological ordering obtained is $d \quad a \quad b \quad c \quad g \quad e \quad f$.

Source (Really good set of questions with solutions, absolutely recommended)