Department of Computer Science & Engineering, Faculty of Engineering

CSC2100B/S: Data Structures
Spring 2011

Written Assignment 4
Due: *29 Apr, 2011*

Name: _____
SID: _____
(Rev. 1.0 @ 11/04)

Section I: Revision Questions (10 points)
Put down your answers to the following simple revision questions.

1. A graph with 10 vertices has at most **45** edges. [ ] 1 pt

2. If a graph has 1000 vertices and 2000 edges, it is better to use the **adjacency list** representation to store the graph. [ ] 1 pt

3. Deleting an edge in the adjacency *matrix* representation takes $O(1)$ time. [ ] 1 pt

4. Deleting an edge in the adjacency *list* representation takes $O(V)$ time. [ ] 1 pt

5. Backtracking in depth-first search is accomplished by **recursion** in common C implementations. [ ] 1 pt

6. A spanning tree of a graph with 1000 vertices contains **999** edges. [ ] 1 pt

7. Prim's algorithm is a **greedy** algorithm that keeps picking edges with minimum weights, as long as no cycle is formed. [ ] 1 pt

8. If adjacency matrix representation is used, the time complexity of Prim's algorithm is $O(V^2 \lg V)$. [ ] 1 pt

9. Bellman-Ford algorithm can be used to detect **negative-weight cycle** in graphs. [ ] 1 pt

10. Efficient implementation of Dijkstra's algorithm requires the use of **min-heap** . [ ] 1 pt

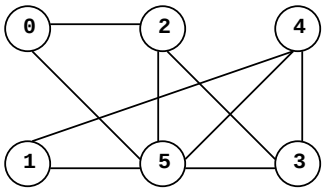
Section II: Short Questions (30 points)
Unless otherwise stated, you should answer the following questions in plain English, diagrams or math expressions instead of code snippets.

1. You are given the following set of edges of an undirected graph G:

4 pts

```
0-2 0-5 2-5 2-3 1-4 5-1 3-5 4-3 4-5
```

Draw G.
Write down any **TWO** cycles of length 4 in G.
Solution:



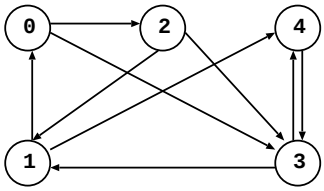
Cycles: 0-2-3-5-0, 1-4-3-5-1

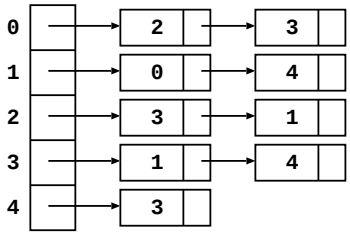
2. The following is the adjacency matrix of graph G:

4 pts

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

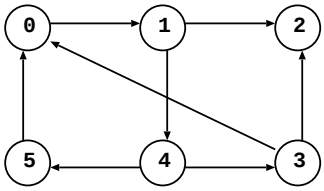
Draw G.
Put down the adjacency list representation of G.
Solution:





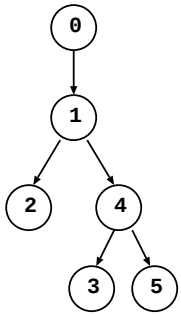
3. A directed graph G is shown below:

4 pts



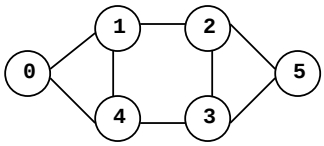
Assume that the graph is represented by *adjacency matrix*, apply depth-first search (DFS) to G: (i) write down the order in which the vertices are discovered; (ii) name all edges involved in the discovery; (iii) draw the DFS tree.

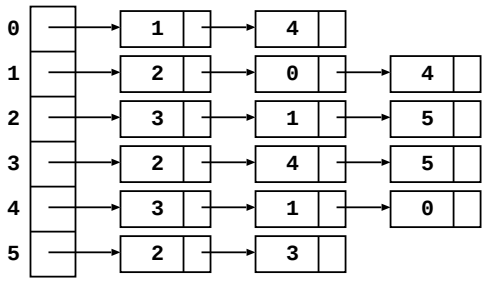
Solution: Order: 0, 1, 2, 4, 3, 5
Edges: 0-1, 1-2, 1-4, 4-3, 4-5



4. A undirected graph G with its adjacency list is shown below:

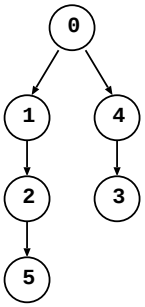
4 pts





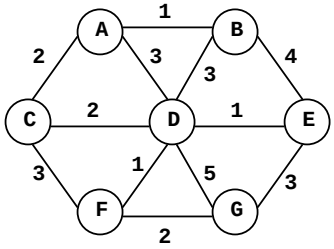
Begin with vertex 0, execute breadth-first search (BFS) on G: (i) write down the order in which the vertices are discovered; (ii) name all edges involved in the discovery; (iii) draw the BFS tree.

Solution: Order: 0, 1, 4, 2, 3, 5
Edges: 0-1, 0-4, 1-2, 4-3, 2-5



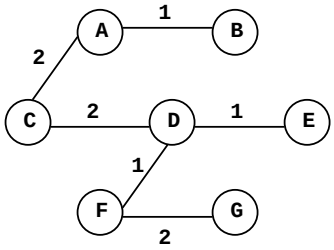
5. A weighted and undirected graph G with weights annotated along the edges is shown below:

4 pts



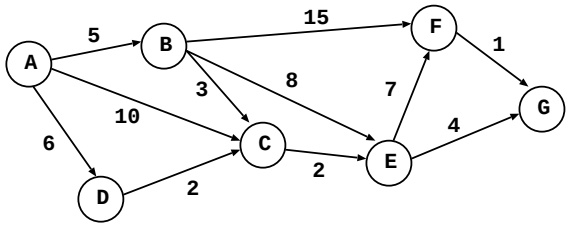
Use Prim's algorithm to the find the minimum spanning tree for G. Begin your tree construction with vertex A.
Draw the minimum spanning tree found.

Solution:



6. A weighted and directed graph G is shown below:

10 pts



We are interested to find the shortest path from vertex A to vertex G, using the algorithms we discussed in the lecture. To illustrate your working in the algorithm(s), use tables to show both $d[v]$ and $\pi[v]$. Also write down clearly the shortest path and the cost.

(a) (5 pts) Apply Bellman-Ford algorithm to find the shortest path from A to G.

Solution:

$d[v]$	A	B	C	D	E	F	G
0	-	-	-	-	-	-	-
1	0	5	10	6	-	-	-
2	0	5	8	6	10	20	-
3	0	5	8	6	10	17	14
4	0	5	8	6	10	17	14
5	0	5	8	6	10	17	14
6	0	5	8	6	10	17	14
7	0	5	8	6	10	17	14

$\pi[v]$	A	B	C	D	E	F	G
0	-	-	-	-	-	-	-
1	-	A	A	A	-	-	-
2	-	A	B	A	C	B	-
3	-	A	B	A	C	E	E
4	-	A	B	A	C	E	E
5	-	A	B	A	C	E	E
6	-	A	B	A	C	E	E
7	-	A	B	A	C	E	E

Shortest Path: $A \rightarrow B \rightarrow C \rightarrow E \rightarrow G$
Cost: $5 + 3 + 2 + 4 = 14$

(b) (5 pts) Apply Dijkstra's algorithm to find the shortest path from A to G. (Use vertex name for tie-breaking in the minimum selection, if $d[v]$'s are the same)

Solution:

min	$d[v]$	A	B	C	D	E	F	G
	0	0	-	-	-	-	-	-
A	1	0	5	10	6	-	-	-
B	2	0	5	8	6	13	20	-
D	3	0	5	8	6	13	20	-
C	4	0	5	8	6	10	20	-
E	5	0	5	8	6	10	17	14
G	6	0	5	8	6	10	17	14
F	7	0	5	8	6	10	17	14

$\pi[v]$	A	B	C	D	E	F	G
0	-	-	-	-	-	-	-
1	-	A	A	A	-	-	-
2	-	A	B	A	B	B	-
3	-	A	B	A	C	B	-
4	-	A	B	A	C	B	-
5	-	A	B	A	C	E	E
6	-	A	B	A	C	E	E
7	-	A	B	A	C	E	E

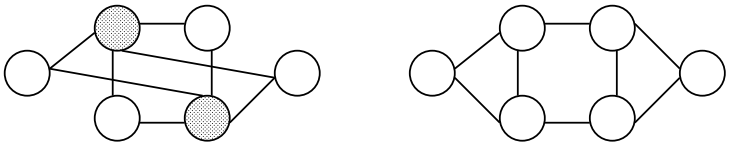
Shortest Path: $A \rightarrow B \rightarrow C \rightarrow E \rightarrow G$

Cost: $5 + 3 + 2 + 4 = 14$

Section III: Long Question / Code Study (30 points)
Unless otherwise stated, you should answer the following questions in plain English instead of code snippets.

1. A graph is two-colourable if there is a way to assign one of two colours (say black and white) to each vertex such that no edge connects two vertices of the same colour. For example, the graph on the left of the diagram below is two-colourable but that on the right is not.

15 pts



In this question, we assume a graph is represented by adjacency matrix defined in C as follows:

```
typedef struct {
    int V; /* no. of vertices */
    int E; /* no. of edges */
    int **adj; /* the adj. matrix */
} graph;
typedef struct {
    int v, w;
} edge_t;
```

(a) (5 pts) Based on DFS, design an algorithm that can check whether a graph is two-colourable or not. Write down the time complexity of your algorithm.

Solution: Allocate an $O(V)$ array to record the colours of the vertices, initially CLEAN (means not coloured). Then begin with an arbitrary vertex, colour it (arbitrarily) and start the DFS traversal. Based on the edge information, a vertex will be coloured BLACK if its predecessor is WHITE, or vice versa. If an edge is found connecting two visited vertices in the same colour, the traversal is terminated and the graph is not two-colourable.
Time complexity = $O(V + E)$

(b) (6 pts) Implement your algorithm stated in (a) using C language. Suppose that your routine is written in the C function `int is_two_colourable(graph G)` that returns 1 when it is two-colourable; 0 otherwise. You may assume the graph is undirected and the matrix is well-initialized.

Solution:

```
1  #define CLEAN -1
2  #define WHITE 0
3  #define BLACK 1
4
5  int dfs_visit(graph G, edge_t e, int *colour){
6      int i, v = e.v, w = e.w;
7      colour[w] = colour[v] == BLACK ? WHITE : BLACK;
8      for (i = 0; i < G->V; i++){
9          if (G->adj[w][i]){
10             if (colour[i] == CLEAN){
11                 if (dfs_visit(G, EDGE(w, i), colour) == 0)
12                     return 0;
13             }
14             else if (colour[i] == colour[w])
15                 return 0;
16         }
17     }
18     return 1;
19 }
```

```
1  int is_two_colourable(graph G){
2      int i, r = 1, *colour = malloc(sizeof(int) * G->V);
3      for (i = 0; i < G->V; i++)
4          colour[i] = CLEAN;
5      for (i = 0; i < G->V && r == 1; i++){
6          if (colour[i] == CLEAN){
7              colour[i] = BLACK;
8              r = dfs_visit(G, EDGE(i, i), colour);
9          }
10     }
11     free(colour);
12     return r;
13 }
```

(c) (4 pts) Can BFS be used to solve the same problem? If yes, describe briefly your algorithm; if no, explain why.

Solution: Yes. First colour the source with one of two colours. Then for all vertices that are 1 edge away, colour with another colour. Similarly, colour all vertices in distance $k+1$ with a different colour of those in distance k. Whenever an edge is found linking two vertices of the same colour, the graph is not two-colourable. The graph is two-colourable if all vertices can be coloured using above the BFS-based traversal.

2. The following code snippet shows a method to compute shortest path distance (stored in `d`) for a weighted and directed graph represented by adjacency matrix `adj`. The weights of the edges are stored in a separate array `w`. You may assume all arrays are well allocated and initialized.

15 pts

```
1  #define INF 100
2  typedef struct {
3      int V; /* no. of vertices */
4      int E; /* no. of edges */
5      int **adj; /* the adj. matrix */
6  } graph;
7  void shortest_path(graph G, int **w, int **d){
8      int i, j, k;
9      for (i = 0; i < G->V; i++)
10         for (j = 0; j < G->V; j++)
11             d[i][j] = G->adj[i][j] == 1 ? w[i][j] : INF;
12
13     for (k = 0; k < G->V; k++)
14         for (i = 0; i < G->V; i++)
15             for (j = 0; j < G->V; j++)
16                 if (d[i][j] > d[i][k] + d[k][j])
17                     d[i][j] = d[i][k] + d[k][j];
18 }
```

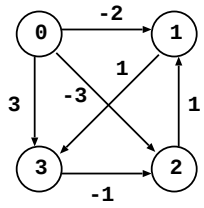
(a) (2 pts) What is usage of the value `INF`?

Solution: To represent *infinity*: indicate that no path exists between the vertices.

(b) (3 pts) Analyze the time complexity of the routine `shortest_path()` in terms of V, the number of vertices in the graph, using the O-notation.

Solution: Time for the first nested loop = $O(V^2)$
Time for the second 3-level nested loop = $O(V^3)$
Total time complexity = $O(V^3)$

(c) (4 pts) Illustrate the algorithm on the following graph:



Show your matrix `d` after every iteration of `k` loop on line 13.

Solution:

$$\begin{bmatrix} 100 & -2 & -3 & 3 \\ 100 & 98 & 97 & 1 \\ 100 & 1 & 97 & 100 \\ 100 & 98 & -1 & 100 \end{bmatrix} \Rightarrow \begin{bmatrix} 98 & -2 & -3 & -1 \\ 100 & 98 & 97 & 1 \\ 100 & 1 & 97 & 2 \\ 100 & 98 & -1 & 99 \end{bmatrix} \Rightarrow$$

$$\begin{bmatrix} 97 & -2 & -3 & -1 \\ 100 & 98 & 97 & 1 \\ 100 & 1 & 97 & 2 \\ 99 & 0 & -1 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 97 & -2 & -3 & -1 \\ 100 & 1 & 0 & 1 \\ 100 & 1 & 1 & 2 \\ 99 & 0 & -1 & 1 \end{bmatrix}$$

(d) (4 pts) Explain how to modify the algorithm so that it can check whether a directed and weighted graph contains a negative weighted cycle or not.

Solution: Initialize the path length for vertex i to itself be zero.
Apply the algorithm.
If the resultant path for (i, i) is negative, there is a negative-weighted cycle in the graph.

(e) (2 pts) Suggest two advantages of the above algorithm over Dijkstra's algorithm in finding all pair shortest paths.

Solution: The algorithm can be applied to graphs with negative weights (but not negative-weighted cycle).
The time complexity is slightly better $O(V^3) < O(VE \log V)$ for dense graphs.