Assignment #6, CSE250

Due Date: Thur. May 5, 2011, 9:00 - 10:00am, 232 Bell Hall

THERE WILL BE NO EXTENSION

UNSUPPORTED SOLUTIONS RECEIVE NO CREDIT.

Total points: 23

1. (2+2= 4 pts) Consider the graph shown in Figure 1.

(a) Represent the graph by using the adjacency matrix representation.

(b) Represent the graph by using the adjacency list representation. (The vertices in each adjacency list should appear in numerical order).

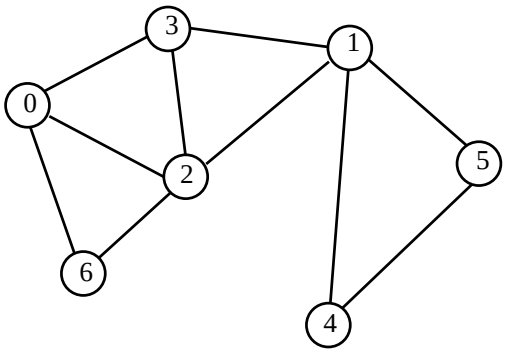


Figure 1:

Solution:

(a)

	0	1	2	3	4	5	6
0			1	1			1
1			1	1	1	1	
2	1	1		1			1
3	1	1	1				
4		1				1	
5		1			1		
6	1		1				

Figure 2: Adjacency matrix

(b)

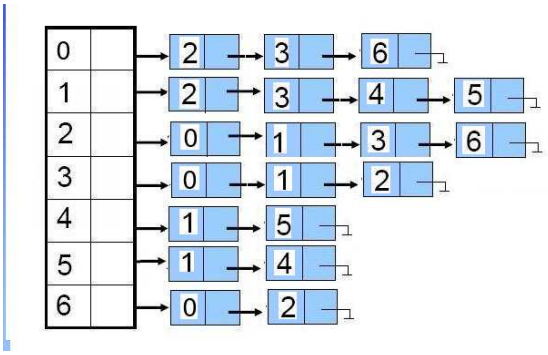


Figure 3: Adjacency list

2. (4 pts) Run BFS (bread-first search) algorithm on the graph shown in Fig 1, using vertex 0 as the starting vertex. (Assuming the adjacency lists are arranged in numerical order).

You should use a table similar to the Table 12.4 (page 717) to show the progress of the algorithm. Also show the BFS tree constructed by the algorithm.

Solution: (1) Bread-first search table

vertex being visited	queue contents after visit	visit sequence
0	2, 3, 6	0
2	3, 6, 1	0 2
3	6, 1	0 2 3
6	1	0 2 3 6
1	4, 5	0 2 3 6 1
4	5	0 2 3 6 1 4
5	NIL	0 2 3 6 1 4 5

(2)

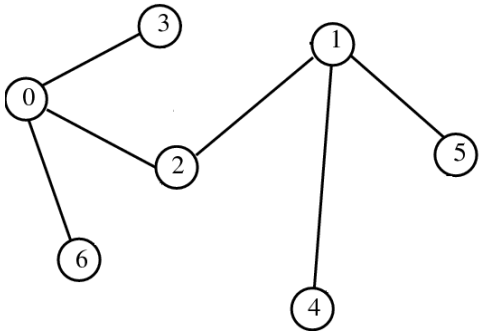


Figure 4: BSF Tree

3. (5 pts) Run Dijkstra's algorithm on the directed graph in Figure 5, using the vertex a as the starting vertex. Show the $d[*]$ and the $p[*]$ values after each iteration of the **while** loop. Also show the shortest path tree constructed by the algorithm.

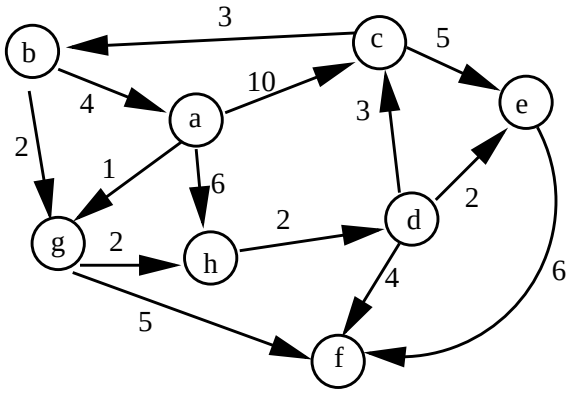


Figure 5: Graph for Dijkstra's Algorithm

Solution:

Table 1: Initially starting: $S = \{a\}$

v	d[*]	p[*]
b	∞	0
c	10	a
d	∞	0
e	∞	0
f	∞	0
g	1	a
h	6	a

Table 2: After iteration 1: $S = \{a, g\}$

v	d[*]	p[*]
b	∞	0
c	10	a
d	∞	0
e	∞	0
f	6	g
g	1	a
h	3	g

Table 3: After iteration 2: $S = \{a, g, h\}$

v	d[*]	p[*]
b	∞	0
c	10	a
d	5	h
e	∞	0
f	6	g
g	1	a
h	3	g

Table 4: After iteration 3: $S = \{a, g, h, d\}$

v	d[*]	p[*]
b	∞	0
c	8	d
d	5	h
e	7	d
f	6	g
g	1	a
h	3	g

Table 5: After iteration 4: $S = \{a, g, h, d, f\}$

v	d[*]	p[*]
b	∞	0
c	8	d
d	5	h
e	7	d
f	6	g
g	1	a
h	3	g

Table 6: After iteration 5: $S = \{a, g, h, d, f, e\}$

v	d[*]	p[*]
b	∞	0
c	8	d
d	5	h
e	7	d
f	6	g
g	1	a
h	3	g

Table 7: After iteration 6: $S = \{a, g, h, d, f, e, c\}$

v	d[*]	p[*]
b	11	c
c	8	d
d	5	h
e	7	d
f	6	g
g	1	a
h	3	g

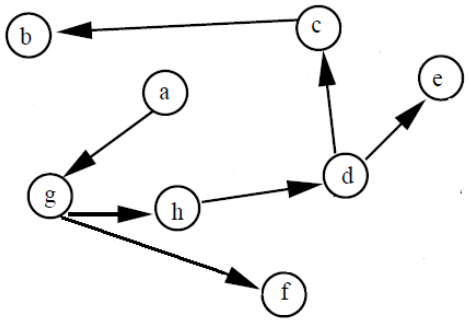
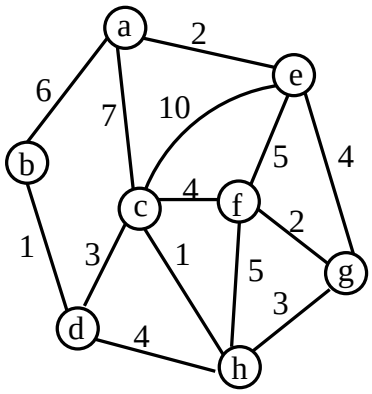


Figure 6: Shortest Path Tree

4. (5 pts) Run Prim's algorithm on the graph $G = (V, E)$ shown in Figure 7. (The integer near an edge is its weight) to compute a minimum spanning tree T of G starting from the vertex a. Use the style shown in Figure 12.28 (page 740) to show the progress of the algorithm after each step.



Prim's algorithm

Figure 7:

Solution:
The order in which the edges are chosen by the algorithm is:
$(a, e), (e, g), (g, f), (g, h), (h, c), (c, d), (d, b)$
The tree is shown in Figure 8

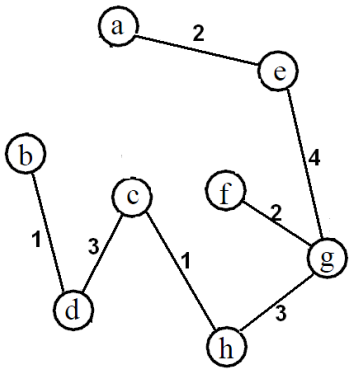


Figure 8: Minimum Spanning Tree

5. (2 pts) In both Prim's and Dijkstra algorithms, a *priority queue* data structure is used. Briefly explain why.

Solution:

A Priority Queue removes the elements based on their weights rather than the FIFO order. In both Prim's and Dijkstra algorithms, we have weights (d[*] values) associated with the vertices. Both algorithms go through a loop. Each iteration of the loop selects the unfinished vertex with the smallest d[*] value as the next vertex to be processed. Without a priority queue, the algorithms must look at every unfinished vertex, which takes $O(n)$ time, hence the total run time would be at least $O(n^2)$. By using a priority queue Q (implemented as a min heap), we can insert the vertices into Q using d[*] as key value. Then selecting the next vertex to process can be done by simply calling the delete-min() function of Q, which only takes $O(\log n)$ time. Hence the total run time of the algorithm are reduced to $O(m \log n)$.

6. (3 pts) We have two graphs $G = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Suppose that $n = |V_1| = |V_2|$, $|E_1| = 4n$ and $|E_2| = n^2/4$. We want to ran certain algorithms on G_1 and G_2. The algorithm will call the function `get_edge(i,j)` (if $\{i, j\}$ is an edge of the graph, the function retrieves the information associated with the edge, if not, the function returns nil), and the function *neighbor-listing(i)* (which returns a list of the neighbors of the vertex i). We need to decide which graph representation (adjacency list or adjacency matrix) should be used.

1. If the space requirement is our main concern, which representation should be used for G_1? and G_2? Why?

2. Suppose that the algorithm will call the function *find-edge(i,j)* $O(n)$ times, and the function *neighbor-listing(i)* $O(\log n)$ times. If the run time is our main concern, which representation should be used for G_1? and G_2? Why?

3. Suppose that the algorithm will call the function *find-edge(i,j)* $O(1)$ times, and the function *neighbor-listing(i)* $O(n)$ times. If the run time is our main concern, which representation should be used for G_1? and G_2? Why?

Solution:

(1) Adjacency list representation should be used for G_1 if space requirement is our main concern. This is because adjacency list requires $O(|V| + |E|)$ space as compared to $O(|V|^2)$. When $|E_1| = 4n, O(|V + E|) = O(n + 4n)$ is far less than $O(n^2)$.

For G_2, we should still use list representation (although saving is less significant in this case.) Since each edge appears in adjacency lists twice, there are $2|E_2|$ entries in all lists. Thus the total space needed for adjacency list representation is $n + 2m = n + n^2/2$. Compared to the n^2 spaced needed by adjacency matrix, we still save about half of the space.

(2) Adjacency matrix should be used for both the graphs in this case. find-edge(i,j) takes $O(1)$ time in adjacency matrix representation as compared to $O(deg(i))$ in adjacency list representation. In this problem find-edge(i,j) is called more number of times, so adjacency matrix should be used.

(3) Adjacency list should be used for both the graphs in this case. neighbor-listing(i) takes $O(deg(i))$ time in adjacency list representation as compared to $O(n)$ in case of adjacency matrix representation. In this problem neighbor-listing(i) is called more number of times, so adjacency list should be used.