Lince Rumainum
Algorithm Analysis

HW#7

1. The efficient algorithms for computing G^2 from G for adjency-list sence the square of a directed graph $G=(V,E)$ is the graph $G^2=(V,E^2)$ for every edge, for example. (1, 2), we will have to run through all the edges from 2 in $O(n)$ time, and fill the results into an adjency matrix of G^2. Let's say that are n number of V and m number of existing edges then it will takes $O(mn)$ to create the edges and since there at most two edges between u and v then it will have to go through another $O(n)$ time to go through another adjency of u or v list which will make the run time $O(V^3)$

Fig 22.2 u = {1, 2, 3, 4, 5, 6}
v = {2, 4, 5, 6, 5, 2, 4, 6}
w = {5, 2, 4, 6, 4, 5, 2, 6}

```
for ( index = 0; index < length of u ; index ++) {
    For ( j = 0;  j < length of v ;  j++) {
        For ( k = 0 ; k < length of w ; k++) {
            newEsquare = (u, w)
        }
    }
}
```

In the case of the adjency-matrix, we can check the intermediate vertices which is n (at most) and since there n^2 pairs of vertices then the total running time will be $O(n^3)$

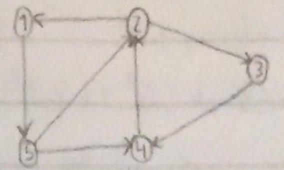
Fig 22.2 n = 6

```
For ( i = 1; i ≤ n ; index++) {
    For (j = 1; j ≤ n ; j++ ) {
        gSquare[i][j] = 0
        For (k = 1 ; k ≤ n ; k++) {
            if (g[i][k] == 1 && g[k][j] == 1) {
                gSquare[i][j] = 1
                break
            }
        }
    }
}
```

Lince Rumainum
Algorithm Analysis

HW#7

2.



$$B_{ij} = \begin{array}{c|ccccc} i \backslash j & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & -1 \\ 2 & -1 & 0 & -1 & 1 & 1 \\ 3 & 0 & 1 & 0 & -1 & 0 \\ 4 & 0 & -1 & 1 & 0 & 1 \\ 5 & 1 & -1 & 0 & -1 & 0 \end{array}$$

$$B^T = \begin{array}{c|ccccc} j \backslash i & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & -1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & -1 & -1 \\ 3 & 0 & -1 & 0 & 1 & 0 \\ 4 & 0 & 1 & -1 & 0 & -1 \\ 5 & -1 & 1 & 0 & 1 & 0 \end{array}$$

$$BB^T = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 2 & -1 & 1 & -2 & -1 \\ 2 & -1 & 4 & -1 & 0 & -2 \\ 3 & 1 & -1 & 2 & -1 & 0 \\ 4 & -2 & 0 & -1 & 3 & 1 \\ 5 & -1 & -2 & 0 & 1 & 3 \end{array}$$

Since $i = j$, the number of BB^T represent the in cost + out cost of that node

3. Using the Breadth-first trees procedures, we calculate the shortest-path distances in the tree for each vertex, V to find the largest. The running time is $O(n^2 + nm)$ where n is the number of vertices and m is the number edges in the tree.

4. If we add a new vertex, it must have traverse k-1 times, where each loop is calculated and the edge with greatest edge gets deleted from the loop. And if we add a new incident edges, then we can just add it, if it's only one. If there is more than one then k-1 edge need to be deleted.

5. Using BFS : a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p

Since the edge weight = 1 for all then there are several path that could be the shortest (edge weights = 6), one of them : $a \to c \to f \to j \to m \to o \to p$

Using Dijkstra's algorithm:

$a \to b : 0+1 = 1$

$a \to c : 0+1 = 1$

$a \to b \to d : 1+1 = 2$

Lince Rumainum
Algorithm Analysis

HW#7

a → b → e : 1+1 = 2

a → c → e : 1+1 = 2

a → c → f : 1+1 = 2

a → b → d → g : 2+1 = 3

a → b → d → h : 2+1 = 3

a → b → e → h : 2+1 = 3

a → b → e → i : 2+1 = 3

a → c → e → h : 2+1 = 3

a → c → e → i : 2+1 = 3

a → c → f → i : 2+1 = 3

a → c → f → j : 2+1 = 3

a → b → d → g → k : 3+1 = 4 → n : 4+1 = 5 → p : 5+1 = 6

a → b → d → h → k : 3+1 = 4 → n : 4+1 = 5 → p : 5+1 = 6

a → b → d → h → l : 3+1 = 4

a → b → e → h → k : 3+1 = 4 → n : 4+1 = 5 → p : 5+1 = 6

a → b → e → h → l : 3+1 = 4

a → c → e → i → l : 3+1 = 4

a → c → e → i → m : 3+1 = 4 → o : 4+1 = 5 → p : 5+1 = 6

a → c → f → i → l : 3+1 = 4

a → c → f → i → m : 3+1 = 4 → o : 4+1 = 5 → p : 5+1 = 6

a → c → f → j → m : 3+1 = 4 → o : 4+1 = 5 → p : 5+1 = 6

.... → l → n : 4+1 = 5 → p : 5+1 = 6

... → l → o : 4+1 = 5 → p : 5+1 = 6

etc.

There are a few paths also for Dijkstra's algorithm for the shortest path from a to p since all weight cost = 1. For example,

a → b → d → g → k → n → p