Name: Suhail Basalama

Homework 8

15.2-2

```
Matrix MATRIX_MULTIPY(A,B)
{
    if (A.columns != B.rows) error "incompatibl dimensions"
    else
        let C be a new A.rows X B.columns Matrix
    for i = 1 to A.rows
        for j=1 to B.columns
            cij=0
            for k=1 to A.columns
                cij = cij + aik.bkj
    return C
}

//First Solution
Matrix MATRIX_CHAIN_MULTIPLY(Matrix[] A, int[][] s, int i, int j)
{
    int k = s[i][j];
    if (i==j) return A[i];
    Matrix A = MATRIX_CHAIN_MULTIPLY(A,s,i,k);
    Matrix B = MATRIX_CHAIN_MULTIPLY(chain,s,k+1,j);
    return MATRIX_MULTIPLY(A,B);

}
//Second Solution (just in case)
Matrix MATRIX_CHAIN_MULTIPLY(Matrix[] A, int[][] s, int i, int j)
{
    int k = s[i][j];
    if (i==j) return A[i];
    if (i+1==j) return MATRIX_MULTIPLY(A[i], A[j]);
    Matrix A = MATRIX_CHAIN_MULTIPLY(A,s,i,k);
    Matrix B = MATRIX_CHAIN_MULTIPLY(chain,s,k+1,j);
    return MATRIX_MULTIPLY(A,B);

}
```

22.2-9

We can use either DFS or BFS since they both have *O(V+E)* run times. However, DFS is more suitable for this problem.

First, we can use DFS which traverses any edge twice at most (once in each direction: once in exploration, and once in backing up). Since DFS explores every vertex and G is undirected connected graph, there must be a path between any two vertices. So, in our algorithm, we add a variable to count the number of times a path is traversed, and we should never go through a path that has a count of two. The algorithm must take unexplored paths only.

Second, we can use BFS while restricting to the edges between v and $v.\pi$ for every v. To prevent double counting edges, we fix any ordering $\leq$ on the vertices. We construct the sequence of steps by calling the function `Build-Path(start)`
where $start$ was the root used for the BFS.

```
Build-Path(u)
    for each v ∈ Adj[u] but not in the tree such that u ≤ v
        go to v and back to u
    for each v ∈ Adj[u] but not equal to u.π
        go to v
        perform the path proscribed by Build-Path(v)
    go to u.π
```

To get out of the maze, we can put pennies in every path we travel. When we reach a dead-end, we should back up and put pennies on the way back. If faced with a branch, we should always go through unvisited paths and should never go through a path with two pennies. Doing this for every path of the maze, we should find a way out.

22.3-1

Directed

j\i	White	Gray	Black
White	Yes: All Kinds	Yes: Tree, Forward	No
Gray	Yes: Back, Cross	Yes: Tree, Back, Forward	Yes: Back
Black	Yes: Cross	Yes: Tree, Back, Cross	Yes: All kinds

Undirected

j\i	White	Gray	Black
White	Yes: Tree, Back	Yes: Tree, Back	No
Gray	Yes: Tree, Back	Yes: Tree, Back	Yes: Tree, Back
Black	No	Yes: Tree, Back	Yes: Tree, Back

22.3-7

```
time //global variable

DFS(G)

    for each vertex u in G.V
        u.color = WHITE
        u.π     = NIL

    time = 0

    for each vertex s in G.V
        if s.color == WHITE
            DFS_VISIT(G,s)

DFS_VISIT(G,s)

    new stack
    stack.push(s)

    while stack != EMPTY
        u = stack.pop()
        if u.color == WHITE
            time++
            u.d = time
            u.color = GRAY
            stack.push(u)
            for each v in G.Adj[u]
                if v.color == WHITE
                    v.π = u
                    stack.push(v)
        else if u.color == GRAY
            time++
            u.f = time
            u.color = BLACK
```

22.3-8

Running the DFS on the following graph starting at s, we get the following values for u and v.

$u.d < v.d$, and there is a path from $u\ to\ v$, but v is not a descendant of u.