Lab 3 – Problem 6

The algorithm below implements matrix multiplication to find both the APSP matrix, as well as the shortest path between X and Y (or none, if none, or -1, if there was a negative cycle) using a second path matrix.

```
/**
 * Generates an n x n matrix containing all pairs shortest paths (APSP).
 * This is implemented as slowAPSP, meaning we don't use the cool squaring trick.
 * For small matrix sizes (n < 50) this will be fine (and as such, matrix obj is capped)
 *
 * When the matrix finds a new minimum, a path 2D array gets updated such that paths[i][j]
 * will always give you the next node to travel to such that you're on the min cost path.
 *
 * That way, travelling upon the nodes of paths[i][j] = your new node, and from that new node
 * again paths[i][j] to get a newer node, until you get to j, will get you to your destination.
 */
std::pair<matrix, std::vector<int>> GraphTraverser::allPairsShortestPath(bool printVerbose, int x, int y) {
    // Note: D^0 = identity matrix
    int n = graph.nodeCount();
    matrix W = matrix();

    vector<vector<int>> paths;
    paths.resize(n, vector<int>(n, -1)); // initialize paths with -1 as the default value
    for (int i = 0; i < graph.nodeCount(); i++) {
        for (int j : graph.getVerticesOut(i)) { // if path exists, then put J as the initial best target.
            paths[i][j] = j;
        }
    }

    for (int i = 0; i < n; i++) // fill up W matrix, aka D^1
        for (int j = 0; j < n; j++) {
            if (i == j) {
                W[i][j] = 0; // if i == j, then 0
                continue;
            }
            if (graph.isEdge(i, j)) { // if edge exists, then the cost
                W[i][j] = graph.getCost(i, j);
            } else {
                W[i][j] = COST_INFINITY; // if edge doesn't exist, then infinity
            }
        }

    auto D = matrix(W); // make a copy...
    for (int m = 2; m < n; m++) { // perform extending/matrix
multiplication...

        if (printVerbose) {  // Simply log output - nothing mysterious here
            std::cout << "D^" << m - 1 << std::endl;
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++)
                    std::cout << D[i][j] << " ";
                std::cout << std::endl;
            }
            std::cout << std::endl << std::endl;
            std::cout << "Paths Matrix" << std::endl;
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++)
                    std::cout << paths[i][j] << " ";
                std::cout << std::endl;
            }
            std::cout << std::endl;
            std::cout << "= = = = = = = = = = = = = =" << std::endl;
        } // end of logging

        D = apspExtend(D, W, paths); // D^m = extend(D^m-1, W)

    }

    if (printVerbose) { // Simply log output - nothing mysterious here
        std::cout << "D^" << n - 1 << std::endl;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++)
                std::cout << D[i][j] << " ";
            std::cout << std::endl;
        }
        std::cout << std::endl;
        std::cout << "Paths Matrix" << std::endl;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++)
                std::cout << paths[i][j] << " ";
            std::cout << std::endl;
        }
        std::cout << std::endl;
        std::cout << "= = = = = = = = = = = = = =" << std::endl;
    } // end of logging

    // negative cycles?
    for (int i = 0; i < n; i++)
        if (D[i][i] < 0)
            return {D, vector<int>(1, -1) };

    std::cout << std::endl;
    vector<int> path = apspGetPath(paths, x, y);
    return {D, path}; // return D^n-1
}

matrix GraphTraverser::apspExtend(matrix D, matrix W, vector<vector<int>>
&paths) {
    matrix C = D; // copy matrix D upon C; note a std::move() may have worked
better if implemented.
    for (int i = 0; i < graph.nodeCount(); i++)
        for (int j = 0; j < graph.nodeCount(); j++) {
            for (int k = 0; k < graph.nodeCount(); k++) {
                // we perform the matrix multiplication step here.
                if (C[i][j] > D[i][k] + W[k][j]) { // new minimum found!
                    C[i][j] = D[i][k] + W[k][j];
                    if (paths[i][k] != -1) { // needed! otherwise will fill
the matrix wrong.
                        paths[i][j] = paths[i][k]; // we update our paths
matrix accordingly.
                    }
                }
            }
        }
    return C;
}

vector<int> GraphTraverser::apspGetPath(vector<vector<int>>& paths, int x,
int y) {
    if (paths[x][y] == -1) {
        return {}; // No path exists from x to y
    } else {
        vector<int> path = {x}; // first node on path is start node
        while (x != y) {
            x = paths[x][y]; // keep searching for next node until none left
            path.push_back(x);
        }
        return path; // returned path
    }
}
```