Lab Final

Michael Morikawa

June 11, 2020

Source Code

SimpleAdjMatrixGraph.hpp

```
#pragma once
#include <vector>
#include <list>
#include <iomanip>
#include <iostream>

class AdjacencyMatrixGraph
{
private:
    typedef std::vector<std::vector<bool>> Matrix;
    Matrix AdjMatrix;

public:
    AdjacencyMatrixGraph(int vertices = 0);
    void insertEdge(int src, int dest);
    void insertVertex();
    bool isAdjacent(int src, int dest);
    int inDegree(int vertex);
    void print();
    std::list<int> outgoingEdges(int u);
    int size();
};

AdjacencyMatrixGraph::AdjacencyMatrixGraph(int vertices)
    : AdjMatrix(vertices, std::vector<bool>(vertices, false)){};

void AdjacencyMatrixGraph::insertVertex()
{
    AdjMatrix.push_back(std::vector<bool>(AdjMatrix.size() + 1, false));
}

void AdjacencyMatrixGraph::insertEdge(int src, int dest)
{
    AdjMatrix[src][dest] = true;
}

std::list<int> AdjacencyMatrixGraph::outgoingEdges(int u)
{
    std::list<int> output;
```

```
    for (int w = 0; w < AdjMatrix[u].size(); w++)
    {
        if (AdjMatrix[u][w])
        {
            output.push_back(w);
        }
    }
    return output;
}

bool AdjacencyMatrixGraph::isAdjacent(int src, int dest)
{
    return AdjMatrix[src][dest];
}

int AdjacencyMatrixGraph::inDegree(int vertex)
{
    int inCounter = 0;
    for (auto row : AdjMatrix)
    {
        if (row[vertex])
        {
            inCounter++;
        }
    }
    return inCounter;
}
int AdjacencyMatrixGraph::size()
{
    return AdjMatrix.size();
}
void AdjacencyMatrixGraph::print()
{
    std::cout << std::setw(10) << "Vertex"
              << "Edges";
    for (int i = 0; i < AdjMatrix.size(); i++)
    {
        std::cout << '\n'
                  << std::setw(10)
                  << i;

        for (int j = 0; j < AdjMatrix[i].size(); j++)
        {
            if (AdjMatrix[i][j])
            {
                std::cout << j << ' ';
            }
        }
    }
    std::cout << '\n';
}
```

main.cpp

```cpp
#include <stack>
#include <vector>
#include <map>
#include <iostream>
#include <sstream>
#include <algorithm>
#include "SimpleAdjMatrixGraph.hpp"

std::vector<int> TopologicalSort(AdjacencyMatrixGraph &G);

void allLowerCase(std::string &word);

void processString(std::string input, std::map<std::string, int> &wordMap);

void top3Words(const std::map<std::string, int> &wordMap);

int main()
{
    AdjacencyMatrixGraph DAG(6);
    DAG.insertEdge(0, 2);
    DAG.insertEdge(0, 3);
    DAG.insertEdge(1, 3);
    DAG.insertEdge(3, 4);
    DAG.insertEdge(2, 4);
    DAG.insertEdge(4, 5);
    std::map<std::string, int> wordMap;
    std::string testString = "This is a test and this is another test test test test it is";

    processString(testString, wordMap);
    std::cout << "Top 3 words in \"" << testString << "\"\n\n";
    top3Words(wordMap);

    std::cout << "\nGraph:\n";
    DAG.print();
    std::cout << "\nTopological Sort: ";
    for (int v : TopologicalSort(DAG))
    {
        std::cout << v << " ";
    }
    std::cout << '\n';
}

std::vector<int> TopologicalSort(AdjacencyMatrixGraph &G)
{
    std::stack<int> S;
    std::vector<int> sorted;
    std::vector<int> incounter(G.size(), 0);
    for (int i = 0; i < G.size(); i++)
    {
        incounter[i] = G.inDegree(i);
        if (G.inDegree(i) == 0)
        {
```

```
            S.push(i);
        }
    }
    int i = 1;
    int u;
    while (!S.empty())
    {
        u = S.top();
        sorted.push_back(u);
        S.pop();
        i++;
        for (auto w : G.outgoingEdges(u))
        {
            incounter[w]--;
            if (incounter[w] == 0)
            {
                S.push(w);
            }
        }
    }
    return sorted;
}

void allLowerCase(std::string &word)
{
    for (unsigned int i = 0; i < word.size(); i++)
    {
        word[i] = std::tolower(word[i]);
    }
}

void processString(std::string input, std::map<std::string, int> &wordMap)
{
    std::stringstream ss(input);
    std::string word;
    while (ss >> word)
    {
        allLowerCase(word);
        if (wordMap.find(word) == wordMap.end())
        {
            wordMap[word] = 1;
        }
        else
        {
            wordMap[word]++;
        }
    }
}

void top3Words(const std::map<std::string, int> &wordMap)
{
    int max1 = 0, max2 = 0, max3 = 0;
    int count;
    std::string word, s1, s2, s3;
```

```
    for (auto p : wordMap)
    {
        count = p.second;
        word = p.first;
        if (count > max1)
        {
            max3 = max2;
            max2 = max1;
            max1 = count;
            s3 = s2;
            s2 = s1;
            s1 = word;
        }
        else if (count > max2)
        {
            max3 = max2;
            max2 = count;
            s3 = s2;
            s2 = word;
        }
        else if (count > max3)
        {
            max3 = count;
            s3 = word;
        }
    }
    std::cout << std::left << std::setw(15) << "Word"
              << "Count\n"
              << std::setw(15) << s1 << max1 << '\n'
              << std::setw(15) << s2 << max2 << '\n'
              << std::setw(15) << s3 << max3 << '\n';
}
```

Output

```
Top 3 words in "This is a test and this is another test test test test it is"

Word           Count
test           5
is             3
this           2

Graph:
Vertex    Edges
0         2 3
1         3
2         4
3         4
4         5
5

Topological Sort: 1 0 3 2 4 5
```