Lab 10: Graph Traversals

Michael Morikawa

May 19, 2020

Lab Questions

Question 1 Does DFS or BFS guarantee to visit the vertices in a certain order? Why or why not?
DFS and BFS will of course have a different order that they visit the vertices but they should visit the vertices in same order every time. However, the order that the different searches visit the vertices depends on the implemention of the algoritms and the graph.

Question 2 What is a DAG? Give an example of a real-life DAG.
A DAG is a directed acyclic graph, which is a digraph that has no cycles in it, meaning that for every vertex in the graph there is not path that leads back to itself. An example of a DAG would be a dependency graph that could be used for scheduling. In the dependency graph the vertices would be a task, the incoming edges are from the dependent task and the outgoing edges connect to tasks that depend on that vertex.

Source Code

main.cpp

```
#include <stack>
#include "BFS.hpp"
#include "LabDFS.hpp"
#include "AdjacencyListGraph.hpp"

std::list<Vertex *> TopologicalSort(Graph *G);

int main()
{
    Graph *test = new AdjacencyListGraph();

    Vertex *A = test->insertVertex("A");
    Vertex *B = test->insertVertex("B");
    Vertex *C = test->insertVertex("C");
    Vertex *D = test->insertVertex("D");
    Vertex *E = test->insertVertex("E");

    test->insertEdge(A, B, 1);
    test->insertEdge(A, D, 2);
    test->insertEdge(B, C, 3);
    test->insertEdge(C, D, 4);
    test->insertEdge(D, E, 5);
    LabDFS depthSearch(test);
    depthSearch.initialize();
    std::cout << "Depth First Search: \n";

    depthSearch.dfsTraversal(A);

    BFS breadthSearch(test);
    breadthSearch.initialize();
    std::cout << "\nBreadth First Search:\n";
    breadthSearch.bfsTraversal(A);
    //DAG Driver

    Graph *dagTest = new AdjacencyListGraph(true);
    Vertex *a = dagTest->insertVertex("A");
    Vertex *b = dagTest->insertVertex("B");
    Vertex *c = dagTest->insertVertex("C");
    Vertex *d = dagTest->insertVertex("D");
    Vertex *e = dagTest->insertVertex("E");
    dagTest->insertEdge(b, a, 100);
    dagTest->insertEdge(b, c, 200);
    dagTest->insertEdge(c, d, 300);
    dagTest->insertEdge(a, d, 400);
    dagTest->insertEdge(d, e, 500);
    std::cout << "\nDAG Topological Ordering\n";
    std::list<Vertex *> sortedVertices = TopologicalSort(dagTest);
    int i = 1;
    for (auto v : sortedVertices)
    {
        std::cout << v->getElement() << " " << '\n';
    }
}

std::list<Vertex *> TopologicalSort(Graph *G)
{
    std::stack<Vertex *> S;
    std::list<Vertex *> sorted;
    std::map<Vertex *, int> incounter;
    for (auto u : G->getVertices())
    {
        incounter[u] = G->inDegree(u);
        if (G->inDegree(u) == 0)
        {
            S.push(u);
        }
    }
    int i = 1;
    Vertex *u;
    Vertex *w;
    while (!S.empty())
    {
        u = S.top();
        sorted.push_back(u);
        S.pop();
        i++;
        for (auto e : G->outgoingEdges(u))
        {
            w = G->opposite(u, e);
            incounter[w]--;
            if (incounter[w] == 0)
            {
                S.push(w);
            }
        }
    }
    return sorted;
}
```

DFS.h (Given)

```
#pragma once
#include "Decorator.h"
#include "Graph.h"

using namespace std;

// Created by T. Vo for CSCI 230
// Based on C++ code fragment of Goodrich book

// Make sure that Vertex and Edge is-a Decorator

class DFS
{                                          // generic DFS
protected:                          // member data
        Graph *graph;          // the graph
        Vertex *start;          // start vertex
        Object *yes, *no; // decorator values

public:                                                                          // member functions
        DFS(Graph *g);                                                  // constructor
        void initialize();                                          // initialize a new DFS
        void dfsTraversal(Vertex *v);                  // recursive DFS utility
                                                                                  // overridden functio
        virtual void startVisit(Vertex *v) {} // arrived at v
                                                                                  // discovery edge e
        virtual void traverseDiscovery(Edge *e, Vertex *from) {}
        // back edge e
        virtual void traverseBack(Edge *e, Vertex *from) {}
        virtual void finishVisit(Vertex *v) {}                  // finished with v
        virtual bool isDone() const { return false; } // finished?

        void visit(Vertex *v) { v->set("visited", yes); }
        void visit(Edge *e) { e->set("visited", yes); }
        void unvisit(Vertex *v) { v->set("visited", no); }
        void unvisit(Edge *e) { e->set("visited", no); }
        bool isVisited(Vertex *v) { return v->get("visited") == yes; }
        bool isVisited(Edge *e) { return e->get("visited") == yes; }
};

DFS::DFS(Graph *g) : graph(g), yes(new Object), no(new Object) {}
```

```
void DFS::initialize()
{
        list<Vertex *> verts = graph->getVertices();
        for (auto pv = verts.begin(); pv != verts.end(); ++pv)
                unvisit(*pv); // mark vertices unvisited

        list<Edge *> edges = graph->getEdges();
        for (auto pe = edges.begin(); pe != edges.end(); ++pe)
                unvisit(*pe); // mark edges unvisited
}

void DFS::dfsTraversal(Vertex *v)
{
        startVisit(v);
        visit(v); // visit v and mark visited
        vector<Edge *> incident = graph->outgoingEdges(v);
        auto pe = incident.begin();
        while (!isDone() && pe != incident.end())
        { // visit v's incident edges
                Edge *e = *pe++;
                if (!isVisited(e))
                {                                                                           // discover
                        visit(e);                                                   // mark it visited
                        Vertex *w = graph->opposite(v, e); // get opposing vertex
                        if (!isVisited(w))
                        {                                                         // unexplored?
                                traverseDiscovery(e, v); // let's discover it
                                if (!isDone())
                                        dfsTraversal(w); // continue traversal
                        }
                        else
                                traverseBack(e, v); // process back edge
                }
        }
        if (!isDone())
                finishVisit(v); // finished with v
}
```

LabDFS.hpp

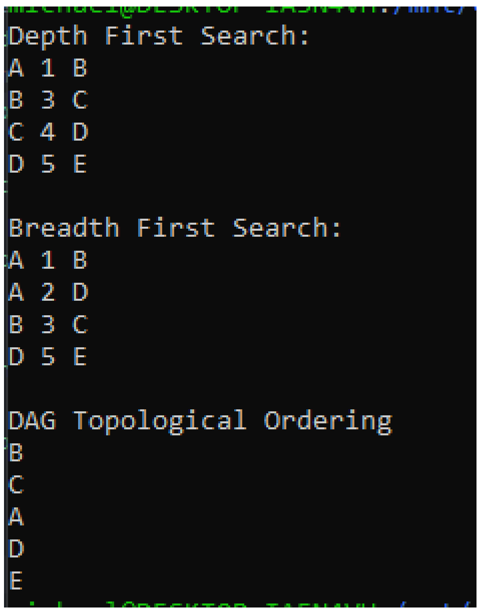
```
#pragma once
#include "DFS.h"
#include <iostream>

class LabDFS : public DFS
{
public:
    LabDFS(Graph *g) : DFS(g) {}

protected:
    void traverseDiscovery(Edge *e, Vertex *from);
};

void LabDFS::traverseDiscovery(Edge *e, Vertex *from)
```

```
{
    std::cout << from->getElement() << " " << e->getElement() << " "
              << graph->opposite(from, e)->getElement() << "\n";
}
```

BFS.hpp

```
#pragma once
#include <iostream>
#include "Decorator.h"
#include "Graph.h"

class BFS
{
public:
    BFS(Graph *g);
    void initialize();
    void bfsTraversal(Vertex *v);
    void visit(Vertex *v) { v->set("visited", yes); }
    void visit(Edge *e) { e->set("visited", yes); }
    void unvisit(Vertex *v) { v->set("visited", no); }
    void unvisit(Edge *e) { e->set("visited", no); }
    bool isVisited(Vertex *v) { return v->get("visited") == yes; }
    bool isVisited(Edge *e) { return e->get("visited") == yes; }

private:
    Graph *graph;
    Vertex *start;
    Object *yes, *no;
};

BFS::BFS(Graph *g) : graph(g), yes(new Object), no(new Object) {}

void BFS::initialize()
{
    std::list<Vertex *> verts = graph->getVertices();
    for (auto v : verts)
    {
        unvisit(v);
    }

    std::list<Edge *> edges = graph->getEdges();
    for (auto e : edges)
    {
        unvisit(e);
    }
}

void BFS::bfsTraversal(Vertex *v)
{
    //initialize level 0 to contain the start vertex
    std::vector<std::list<Vertex *>> levels(1, std::list<Vertex *>(1, v));
    int i = 0;
    Vertex *w;
    while (!levels[i].empty())
    {
        levels.push_back(std::list<Vertex *>());
        for (auto u : levels[i])
        {
            for (auto e : graph->outgoingEdges(u))
            {
                if (!isVisited(e))
                {
                    w = graph->opposite(u, e);
                    visit(e);
                    if (!isVisited(w))
                    {
                        visit(w);
                        levels[i + 1].push_back(w);
                        std::cout << u->getElement() << " " << e->getElement()
                                  << " " << w->getElement() << "\n";
                    }
                }
            }
        }
        i++;
    }
}
```

Output



```
Depth First Search:
A 1 B
B 3 C
C 4 D
D 5 E

Breadth First Search:
A 1 B
A 2 D
B 3 C
D 5 E

DAG Topological Ordering
B
C
A
D
E
```