Farhan Zaman 12/4/18
CS21200 10:00AM

HW6 Report

This homework was designed to get us familiar with graphs, understand directed graphs make undirected graph,s but not vice versa, how to make an algorithm for the shortest paths, different ways of traversing. We learned to work with a lot of recursion and different data structures.

For Question 1, here is my header file:

```
#ifndef __ZAMAN_FARHAN_HW6_Q1_H__
#define __ZAMAN_FARHAN_HW6_Q1_H__
#include <iostream>
#include <cassert>
#include <cstdlib>
using namespace std;

template <class Item>
class GraphMatrix
{
	private:
		Item* label;
		size_t used;
		size_t capacity;
		unsigned int** weightededges;
		size_t vertices;
	public:
		GraphMatrix(size_t initsize=30);
		~GraphMatrix();
		GraphMatrix(const GraphMatrix& source);
		void addvertex(const Item& label);
		void addedge(const Item source,const Item target,size_t weight);
		void removeedge(const Item source,const Item target);
		void print();
		size_t size() const;
		size_t num_edges();
		bool is_edge(const Item source,const Item sources) const;
		Item* neighbor(const Item source);
};

#include "Zaman_Farhan_HW6_Q1.cpp"
#endif
```

For Question 1, here is my cpp file:

```
#ifndef __ZAMAN_FARHAN_HW6_Q1_CPP__
#define __ZAMAN_FARHAN_HW6_Q1_CPP__
#include "Zaman_Farhan_HW6_Q1.h"

template <class Item>
GraphMatrix<Item>::GraphMatrix(size_t initsize)
{
	label=new Item[initsize];
	used=0;
	capacity=initsize;
	weightededges=new unsigned int[initsize][initsize];
	vertices=0;
}

template <class Item>
GraphMatrix<Item>::~GraphMatrix()
{

}

template <class Item>
GraphMatrix<Item>::GraphMatrix(const GraphMatrix& source)
{
	used=source.used;
	capacity=source.capacity;
	vertices=source.vertices;
	for(int i=0; i<used; i++)
	{
		label[i]=source.label[i];
	}
	for(int i=0; i<used; i++)
	{
		for(int j=0; j<used; j++)
		{
			weightededges[i][j]=source.weightededges[i][j];
		}
	}
}

template <class Item>
void GraphMatrix<Item>::addvertex(const Item& label)
{
	assert(used<capacity);
	label[vertices]=label;
	used++;
}

template <class Item>
```

```
void GraphMatrix<Item>::addedge(const Item source,const Item target,size_t weight)
{
	assert(source<used && target<used);
	int findsource=0;
	int findtarget=0;
	for(int i=0; i<used; i++)
	{
		if(label[i]==source)
		{
			findsource=i;
		}
		if(label[i]==target)
		{
			findtarget=i;
		}
	}
	weightededges[findsource][findtarget]=weight;
}

template <class Item>
void GraphMatrix<Item>::removeedge(const Item source,const Item target)
{
	assert(source<used && target<used);
	int findsource=0;
	int findtarget=0;
	for(int i=0; i<used; i++)
	{
		if(label[i]==source)
		{
			findsource=i;
		}
		if(label[i]==target)
		{
			findtarget=i;
		}
	}
	weightededges[findsource][findtarget]=0;
}

template <class Item>
void GraphMatrix<Item>::print()
{
	for(int i=0; i<used; i++)
	{
		cout<<label[i]<<endl;
	}
	for(int i=0; i<used; i++)
	{
			for(int j=0; j<used; j++)
			{
				cout<<weightededges[i][j];
			}
			cout<<endl;
		}
}

template <class Item>
size_t GraphMatrix<Item>::size() const
{
		return vertices;
}

template <class Item>
size_t GraphMatrix<Item>::num_edges()
{
		size_t edges;
		for(int i=0; i<used; i++)
		{
			for(int j=0; j<used; j++)
			{
				if(weightededges[i][j]>0)
				{
					edges++;
				}
			}
		}
		return edges;
}

template <class Item>
bool GraphMatrix<Item>::is_edge(const Item source,const Item target) const
{
		assert(source<used && target<used);
		int findsource=0;
		int findtarget=0;
		for(int i=0; i<used; i++)
		{
			if(label[i]==source)
			{
				findsource=i;
			}
			if(label[i]==target)
			{
				findtarget=i;
			}
		}
```

```
	return weightededges[findsource][findtarget]>0;
}

template <class Item>
Item* GraphMatrix<Item>::neighbor(const Item source)
{
	Item* neighbors;
	neighbors=new Item[used];
	int findsource=0;
	int adj=0;
	for(int i=0; i<used; i++)
	{
		if(label[i]==source)
		{
			findsource=i;
		}
	}
	for(int j=0; j<used; j++)
	{
		if(weightededges[findsource][j]>0)
		{
			neighbors[adj]=label[j];
		}
	}
}

#endif
```

For Question 2, here is my header file:

```
#ifndef __ZAMAN_FARHAN_HW6_Q2_H__
#define __ZAMAN_FARHAN_HW6_Q2_H__
#include "Zaman_Farhan_HW3.h" //LINKED LIST CLASS
#include <iostream>
#include <cassert>
#include <cstdlib>
#include <set>
using namespace std;

template <class Item>
class GraphList
{
	private:
		Item* label;
		size_t used;
		size_t capacity;
		node<Item>** weightededges;
		size_t vertices;
	public:
		GraphList(size_t initSize=30);
		~GraphList();
		GraphList(const GraphList& source);
		void addvertex(const Item& label);
		void addedge(const Item source,const Item target,size_t weight);
		void removeedge(const Item source,const Item target);
		void print();
		size_t size() const;
		size_t num_edges();
		bool is_edge(const Item source,const Item sources) const;
		Item* neighbor(const Item source);
};

#include "Zaman_Farhan_HW6_Q2.cpp"
#endif
```

For Question 2, here is my cpp file:

```
#ifndef __ZAMAN_FARHAN_HW6_Q2_CPP__
#define __ZAMAN_FARHAN_HW6_Q2_CPP__
#include "Zaman_Farhan_HW6_Q2.h"

template <class Item>
GraphList<Item>::GraphList(size_t initsize)
{
	label=new Item[initsize];
	used=0;
	capacity=initsize;
	weightededges=new node<Item> [initsize];
	vertices=0;
}

template <class Item>
GraphList<Item>::~GraphList()
{

}

template <class Item>
GraphList<Item>::GraphList(const GraphList
& source)
{
	used=source.used;
	capacity=source.capacity;
	vertices=source.vertices;
	for(int i=0; i<used; i++)
	{
```

```
                label[i]=source.label[i];
                weightededges[i]=source.weightededges[i];
        }

}

template <class Item>
void GraphList<Item>::addvertex(const Item& label)
{
        assert(used<capacity);
        label[vertices]=label;
        weightededges[vertices]=NULL;
        used++;
}

template <class Item>
void GraphList<Item>::addedge(const Item source,const Item target,size_t weight)
{
        assert(source<used && target<used);
        int findsource=0;
        int findtarget=0;
        for(int i=0; i<used; i++)
        {
                if(label[i]==source)
                {
                        findsource=i;
                }
                if(label[i]==target)
                {
                        findtarget=i;
                }
        }
        weightededges[findsource][findtarget]=weight;
}

template <class Item>
void GraphList<Item>::removeedge(const Item source,const Item target)
{
        assert(source<used && target<used);
        int findsource=0;
        int findtarget=0;
        for(int i=0; i<used; i++)
        {
                if(label[i]==source)
                {
                        findsource=i;
                }
                if(label[i]==target)
                {
                        findtarget=i;
                }
        }
        weightededges[findsource]=0;
}

template <class Item>
void GraphList<Item>::print()
{
        for(int i=0; i<used; i++)
        {
                cout<<label[i]<<endl;
                cout<<weightededges[i][j]<<endl;
        }

}

template <class Item>
size_t GraphList<Item>::size() const
{
        return vertices;
}

template <class Item>
size_t GraphList<Item>::num_edges()
{
        size_t edges;
        for(int i=0; i<used; i++)
        {
                for(int j=0; j<used; j++)
                {
                        if(weightededges[i]>0)
                        {
                                edges++;
                        }
                }
        }
        return edges;
}

template <class Item>
bool GraphList<Item>::is_edge(const Item source,const Item target) const
{
        assert(source<used && target<used);
        int findsource=0;
        int findtarget=0;
        for(int i=0; i<used; i++)
```

```
	{
		if(label[i]==source)
		{
			findsource=i;
		}
		if(label[i]==target)
		{
			findtarget=i;
		}
	}
	return weightededges[findsource]>0;
}

template <class Item>
Item* GraphList<Item>::neighbor(const Item source)
{
	Item* neighbors;
	neighbors=new Item[used];
	int findsource=0;
	int adj=0;
	for(int i=0; i<used; i++)
	{
		if(label[i]==source)
		{
			findsource=i;
		}
	}
	for(int j=0; j<used; j++)
	{
		if(weightededges[j]>0)
		{
			neighbors[adj]=label[j];
		}
	}
}


#endif
```

For Question 3, here is my cpp file:

```
#ifndef __ZAMAN_FARHAN_HW6_Q3_CPP__
#define __ZAMAN_FARHAN_HW6_Q3_CPP__
#include "Zaman_Farhan_HW6_Q1.h"
#include <iostream>
#include <set>
#include <cstdlib>
#include <queue>
```

```
#include <cassert>
using namespace std;

template <class Process,class Item, class Item2>
void dfs(Process f, GraphMatrix<Item>& g,Item2 a,bool marked[])
{
	set<size_t>::iterator it;
	set<size_t> adj= g.neighbor(a);
	marked[a]=true;
	f(g[a]);
	for (it=adj.begin(); it!=adj.end(); it++)
	{
		if (!marked[*it])
		{
			dfs(f,g,*it,marked);
		}
	}
}

template <class Process,class Item,class Item2>
void depthfirst(Process f,GraphMatrix<Item>& g,Item2 start)
{
	bool marked[g.size()-1];
	assert(start<g.size());
	fill_n(marked,g.size(), false);
	dfs(f,g,start,marked);
}

template <class Process,class Item,class Item2>
void breadthfirst(Process f,GraphMatrix<Item>& g,Item2 start)
{
	bool marked[g.size()-1];
	set<size_t> adj;
	set<size_t>::iterator it;
	queue<size_t> vertex;
	assert(start<g.size());
	fill_n(marked,g.size(),false);
	marked[start] = true;
	f(g[start]);
	vertex.push(start);
	do{
		adj=g.neighbor(vertex.front());
		vertex.pop();
		for (it=adj.begin(); it!=adj.end(); it++)
		{
			if (!marked[*it])
			{
				marked[*it]=true;
```

```
                f(g[*it]);
                vertex.push(*it);
            }
        }
    }while (!vertex.empty());
}

#endif
```

For Question 4, here is my cpp file:

```
#ifndef __ZAMAN_FARHAN_HW6_Q4_CPP__
#define __ZAMAN_FARHAN_HW6_Q4_CPP__
#include "Zaman_Farhan_HW6_Q1.h"
#include <iostream>
#include <cassert>
#include <cstdlib>
#include <set>
#include <queue>
#include <functional>
using namespace std;

template <class Item>
void dijkstra(GraphMatrix<Item>& g, size_t x)
{
    int distance[g.size()];
    int pre[g.size()];
    int u=0;
    size_t v=0;
    for (int i=0; i<g.size(); i++)
    {
        pre[i]=0;
        distance[i]=2000000;
    }
    distance[x]=0;
    set<size_t> adj;
    set<int> vertex;
    priority_queue<int,vector<int>,greater<int>> q;
    for(int i=0; i<g.size();i++)
    {
        q.push(distance[i]);
    }
    while(!q.empty())
    {
        u=q.top();
        q.pop();
        vertex.insert(u);
        adj=g.neighbor(v);
```

```
            set<size_t>::iterator it;
            for (it=adj.begin(); it!=adj.end(); it++)
            {
                  v=*it;
                  if(distance[v]>distance[u]+g.getEdge(u,v))
                  {
                        pre[v]=u;
                        distance[v]=distance[u]+g.getEdge(u,v);
                  }
            }
      }
}

#endif
```

Some problems of this class is that a lot of the algorithms were just hard and assumed so much about the graphs and the problem. Following the etxtbook wasn’t helpful since they just used static arrays and they have a lot of convoluted ways of just doing it. I particularly had issues with understanding BF, and DF, as opposed to BFS and DFS, as in what's the difference and why not do the search.