МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра САПР

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Алгоритмы и структуры данных»

Тема: Алгоритмы на графах (Вариант 1)

Студент гр. 8309 _____ Иванов Д.К.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург

2020

Исходная формулировка задания:

Дан список возможных авиарейсов в текстовом файле в формате: Город отправления 1;Город прибытия 1;цена прямого перелета 1;цена обратного перелета 1.

Найти наиболее эффективный по стоимости перелет из города i в город j.

Цель работы:

Научиться работать с алгоритмом Дейкстры.

Организация данных:

Название	Описание работы метода	Оценка временной сложности
NewListNode	Добавление элемента в список смежности	O(1)
CreateGraph	Создание графа с выделением памяти под массив списков смежности	O(1)
AddEdge	Добавление ребра в графе. Создается список смежности и помещается в массив.	O(1)
NewMinHeapNode	Добавление элемента в двоичную кучу	O(1)
CreateMinHeap	Создание минимальной двоичной кучи	O(1)
SwapMinHeapNode	Функция, которая меняет местами два элемента кучи.	O(1)
MinHeapify	Восстановление свойства кучи по данному индексу и обновление позиций измененных элементов	O(logN)
IsEmpty	Проверка кучи на пустоту	O(1).
ExtractMin	Извлечение минимального элемента из кучи	O(logN)
DecreaseKey	Понижение значения цены данного ребра, выбирая элемент из кучи	O(Log N)

IsMinHeap	Проверка кучи на минимальность	O(1).
Finding	Возвращение нужной строки из массива строк	O(1)
Print	Выводит цены на проезд во все города из выбранного города	O(N).
Dijkstra	Реализация алгоритма Дейкстры. Сначала все элементы принимают значение INT_MAX (бесконечность), после меняется значение первого элемента списка, затем последовательно считаются цены для каждого из списков смежности (поля цены). После чего алгоритм выводится результат с помощью функции print	O(ElogV).
Count	Подсчет кол-ва вершин в файле для создания графа с нужным кол-вом вершин	O(N).
Checking(1-3)	Проверка строки на наличие в массиве строк (для инициализации каждого города)	O(1)

Описание реализованных unit-тестов

Название Unit-теста	Описание работы
TEST_METHOD(Test Newheapnode)	Проверка создания элемента кучи
TEST_METHOD(Test NewListNode)	Проверка создания узла списка смежности
TEST_METHOD(Test CreateGraph)	Проверка создания графа
TEST_METHOD(Test AddEdge)	Проверка создания ребра в графе
TEST_METHOD(Test Newheapnode)	Проверка создания элемента кучи
TEST_METHOD(Test Swap)	Проверка замены элементов кучи и результата
TEST_METHOD(Test Heapify)	Проверка восстановления свойств кучи

TEST_METHOD(Test Empty)	Проверка вывода ошибки при создании итератора для пустого Ассоциативного массива

Название Unit-теста	Описание работы
TEST_METHOD(Test Empty2)	Проверка кучи на пустоту, случай не пустого дерева
TEST_METHOD(Test Extract)	Проверка извлечения минимального элемента кучи
TEST_METHOD(Test Decrease)	Проверка понижения ключа
TEST_METHOD(Test Ismin)	Проверка кучи на минимальность
TEST_METHOD(Test Finding)	Проверка поиска строки по индексу
TEST_METHOD(Test Checking_test)	Проверка наличия строки в массиве
TEST_METHOD(Test Count_test)	Проверка подсчета кол-ва разных городов в файле
TEST_METHOD(Test Checking2_test)	Проверка поиска индекса строки в массие.

Программа

Source.cpp

```
#include"pch.h"
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include<string>
#include<fstream>
#include<iostream>

using namespace std;
//point1 - name of the city, numb - number of the city
struct node {
	int numb;
	string point1;
};
//  node in adjacency list
struct ListNode
```

```
{
	node dest;
	int cost;
	ListNode* next;
};

//  adjacency list
struct AdjList
{
	ListNode *head;
};

// A structure of a graph with array of adjacency lists
struct Graph
{
	int V;
	AdjList* array;
};

//  create a new adjacency list node
ListNode* newListNode(int dest1, int cost,string destination)
{
    ListNode* newNode = new ListNode;

	newNode->dest.point1 = destination;
	newNode->dest.numb = dest1;
	newNode->cost = cost;
	newNode->next = NULL;
	return newNode;
}

//  creates a graph with V vertices
Graph* createGraph(int V)
{
	Graph* graph = new Graph;
	graph->V = V;

	graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));

	for (int i = 0; i < V; ++i)
		graph->array[i].head = NULL;

	return graph;
}

// Adds an edge
void addEdge(Graph* graph, int src, int dest, int cost,int cost2,string source,string destination)
{

	ListNode* newNode = newListNode(dest, cost,destination);
	newNode->next = graph->array[src].head;
	graph->array[src].head = newNode;

	newNode = newListNode(src, cost2,source);
	newNode->next = graph->array[dest].head;
	graph->array[dest].head = newNode;
}

// min heap node
struct MinHeapNode
{
	int  v;
	int cost;
};
```

```
// min heap
struct MinHeap
{
	int size;
	int capacity;
	int *pos;	// This is needed for decreaseKey()
	MinHeapNode **array;
};

// create a new Min Heap Node
MinHeapNode* newMinHeapNode(int v, int cost)
{
	MinHeapNode* minHeapNode =new MinHeapNode;
	minHeapNode->v = v;
	minHeapNode->cost = cost;
	return minHeapNode;
}

//create a Min Heap
MinHeap* createMinHeap(int capacity)
{
	MinHeap* minHeap =new MinHeap;
	minHeap->pos = new int[capacity];
	minHeap->size = 0;
	minHeap->capacity = capacity;
	minHeap->array = ( MinHeapNode**) malloc(capacity * sizeof( MinHeapNode*));
	return minHeap;
}

// swap two nodes of min heap
void swapMinHeapNode(MinHeapNode** a, MinHeapNode** b)
{
	MinHeapNode* t = *a;
	*a = *b;
	*b = t;
}

// heapify at given idx
void minHeapify(MinHeap* minHeap, int idx)
{
	int smallest, left, right;
	smallest = idx;
	left = 2 * idx + 1;
	right = 2 * idx + 2;

	if (left < minHeap->size &&
		minHeap->array[left]->cost < minHeap->array[smallest]->cost)
		smallest = left;

	if (right < minHeap->size &&
		minHeap->array[right]->cost < minHeap->array[smallest]->cost)
		smallest = right;

	if (smallest != idx)
	{

		MinHeapNode *smallestNode = minHeap->array[smallest];
		MinHeapNode *idxNode = minHeap->array[idx];


		minHeap->pos[smallestNode->v] = idx;
		minHeap->pos[idxNode->v] = smallest;
```

```
		swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

		minHeapify(minHeap, smallest);
	}
}

// Heap Empty or not
bool isEmpty(MinHeap* minHeap)
{
	return minHeap->size == 0;
}

// extract minimum node from heap
MinHeapNode* extractMin(MinHeap* minHeap)
{
	if (isEmpty(minHeap))
		return NULL;


	MinHeapNode* root = minHeap->array[0];


	MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
	minHeap->array[0] = lastNode;


	minHeap->pos[root->v] = minHeap->size - 1;
	minHeap->pos[lastNode->v] = 0;

	minHeap->size--;
	minHeapify(minHeap, 0);

	return root;
}

// decrease dist value of a given vertex v
void decreaseKey(MinHeap* minHeap, int v, int cost)
{

	int i = minHeap->pos[v];


	minHeap->array[i]->cost = cost;


	while (i && minHeap->array[i]->cost < minHeap->array[(i - 1) / 2]->cost)
	{

		minHeap->pos[minHeap->array[i]->v] = (i - 1) / 2;
		minHeap->pos[minHeap->array[(i - 1) / 2]->v] = i;
		swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

		i = (i - 1) / 2;
	}
}


// 'v' is in min heap or not
bool isInMinHeap(MinHeap *minHeap, int v)
{
	if (minHeap->pos[v] < minHeap->size)
		return true;
	return false;
}
```

```
//find a string with given index
string finding(int src, string help[100]) {
	return help[src];
}
//print the solution
void print(string help[100],int cost[], int n,int src)
{
	string helping = finding(src, help);
	cout << "source is"<<" "<<helping<<endl;
	cout << endl;
	for (int i = 0;i < n;i++) {
		if(cost[i]!=0)
		cout << finding(i,help) << " " << cost[i] << endl;
	}
}


// The main function that calulates distances of shortest paths from src to all
void dijkstra(string help[100], Graph* graph, int src)
{
	int V = graph->V-1;
	int *cost=new int[V];

	MinHeap* minHeap = createMinHeap(V);

	for (int v = 0; v < V; ++v)
	{
		cost[v] = INT_MAX;
		minHeap->array[v] = newMinHeapNode(v, cost[v]);
		minHeap->pos[v] = v;
	}

	minHeap->array[src] = newMinHeapNode(src, cost[src]);
	minHeap->pos[src] = src;
	cost[src] = 0;
	decreaseKey(minHeap, src, cost[src]);

	minHeap->size = V;

	while (!isEmpty(minHeap))
	{

		MinHeapNode* minHeapNode = extractMin(minHeap);
		int u = minHeapNode->v;

		struct ListNode* pCrawl = graph->array[u].head;
		while (pCrawl != NULL)
		{
			int v = pCrawl->dest.numb;


			if (isInMinHeap(minHeap, v) && cost[u] != INT_MAX && pCrawl->cost + cost[u] <
cost[v])
			{
				cost[v] = cost[u] + pCrawl->cost;

				decreaseKey(minHeap, v, cost[v]);
			}
			pCrawl = pCrawl->next;
		}
	}

	print(help,cost, V,src);
}
```

```
//check a string in array
bool checking(string check[], string check1) {
	int i = 0;
	for (int i = 0;i < sizeof(check);i++) {
		string help = check[i];
		if (check[i] == check1)
			return false;
	}
	return true;
}
//count a number of cities in given file
int count() {
	fstream source;
	source.open("C:\\Users\\Alex\\Desktop\\laba2.txt", ios::in);
	int j = 0;
	int i = 0;
	int k = 0;
	int checks = 0;
	int counting = 0;
	char check = ' ';
	string check1[100];
	char *save = new char[100];
	while (!source.eof()) {
		i = 0;
		while ((check != '\0') && (!source.eof())) {

			checks = 0;
			source >> check;
			if (check == ';') {
				j++;
				save[k] = '\0';
				string help = (string)save;
				if (checking(check1, help) == true)
					counting++;
				check1[i] = help;
				i++;
				memset(save, 0, sizeof(save) / sizeof(save[0]));
				checks = 1;
				k = -1;
			}
			if (j == 2) {
				while (((check == '0') || (check == '1') || (check == '2') || (check == 
'3') || (check == '4') || (check == '5') || (check == '6') || (check == '7') || (check == '8') || 
(check == '9') || (check == ';') || (check == '/') || (check == 'A') || (check == 'N')) && 
(!source.eof()))
					source >> check;
				j = 0;
				k = 0;
				checks = 0;
			}
			if (checks == 0)
				save[k] = check;
			k++;
		}
		string help = (string)save;
		if (checking(check1, check1[k]) == true)
			counting++;
		check1[i] = help;
		i++;
	}
	return counting;
}

//check string in array
```

```
bool checking3(string check[100], string str) {
	int i = 0;
	for ( i = 0;i < sizeof(check);i++) {
		if (check[i] == str)
			return true;
	}
	check[i] = str;
	return false;
}
//find an index of string
int checking2(string check[100], string str) {
	int i = 0;
	for (i = 0;i < 100;i++) {
		if (check[i] == str)
			return i;
	}
}

int main() {
	fstream source;
	int f = 0;
	source.open("C:\\Users\\Alex\\Desktop\\laba2.txt", ios::in);
	int V = count();
	Graph *Graph = createGraph(V);
	char help[100];
	string helping[100];
	int j = 0;
	while (!source.eof()) {
		string point1, point2 = "";
		int i = 1;
		if (j == 0)
			i = 0;
		int cost, cost2 = 0;
		char check = ' ';
		while (check != ';') {
			source >> check;
			if (check == ';')
				break;
			help[i] = check;
			i++;
		}
		help[i] = '\0';
		point1 = (string)help;
		check = ' ';
		i = 0;
		while (check != ';') {
			//i = 0;
			source >> check;
			if (check == ';')
				break;
			help[i] = check;
			i++;
		}
		help[i] = '\0';
		point2 = (string)help;
		check = ' ';
		i = 0;
		while (check != ';') {
			//i = 0;
			source >> check;
			if (check == ';')
				break;
			help[i] = check;
			i++;
```

```
			}
			help[i] = '\0';
			if ((help[0] == 'N') && (help[1] == '/') && (help[2] == 'A')) {
				cost = INT_MAX;
			}
			else
				cost = stoi(help);
			source >> check;
			i = 0;
			while (((check == '0') || (check == '1') || (check == '2') || (check == '3') ||
(check == '4') || (check == '5') || (check == '6') || (check == '7') || (check == '8') || (check ==
'9') || (check == ';') || (check == '/') || (check == 'A') || (check == 'N')) && (!source.eof())) {
				//i = 0;
				if (check == ';')
					break;
				help[i] = check;
				i++;
				if (((check == '0') || (check == '1') || (check == '2') || (check == '3') ||
(check == '4') || (check == '5') || (check == '6') || (check == '7') || (check == '8') || (check ==
'9') || (check == ';') || (check == '/') || (check == 'A') || (check == 'N')) && (!source.eof()))
					source >> check;
			}
			help[i] = '\0';
			if ((help[0] == 'N') && (help[1] == '/') && (help[2] == 'A')) {
				cost2 = INT_MAX;
			}
			else
				cost2 = stoi(help);
			int point11, point12 = 0;
			if (checking3(helping, point1) == true)
				point11 = checking2(helping, point1);
			else {
				helping[f] = point1;
				point11 = f;
				f++;
			}
			if (checking3(helping, point2) == true)
				point12 = checking2(helping, point2);
			else {
				helping[f] = point2;
				point12 = f;
				f++;
			}
			addEdge(Graph, point11, point12, cost, cost2,point1,point2);
			help[0] = check;
			j++;
		}
		source.close();
		string city;
		cout << "enter a city" << endl;
		getline(cin, city);
		if (checking3(helping, city) == false)
			return 0;
		int index = checking2(helping,city);
		dijkstra(helping,Graph, index);
}
```

Unittest.cpp

```
#include "pch.h"
#include "CppUnitTest.h"
#include "C:\Users\user\source\repos\4.3\4.3\Source.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;
```

```
namespace UnitTest1{
	TEST_CLASS(UnitTest1)
	{
	public:

		TEST_METHOD(NewListNode)
		{
			int one = 1;
			int cost = 10;
			string destination = "Moscow";
			ListNode *test = newListNode(one, cost, destination);
			Assert::AreEqual(1, test->dest.numb);
			Assert::AreEqual((string)"Moscow", test->dest.point1);
			Assert::AreEqual(10, test->cost);
		}
		TEST_METHOD(CreateGraph)
		{
			int V = 9;
			Graph *graph = createGraph(V);
			Assert::AreEqual(9,graph->V);
		}
		TEST_METHOD(Addedge)
		{
			int V = 9;
			Graph *graph = createGraph(V);
			int point11 = 1;
			int point12 = 2;
			int cost = 10;
			int cost2 = 15;
			string point1 = "Saint-P";
			string point2 = "Moscow";

			addEdge(graph, point11, point12, 10, 15, point1, point2);
			Assert::AreEqual( graph->array[1].head->cost,10);
			Assert::AreEqual(graph->array[1].head->dest.point1, (string)"Moscow");
			Assert::AreEqual(graph->array[1].head->dest.numb, 2);
			Assert::AreEqual(graph->array[2].head->cost, 15);
			Assert::AreEqual(graph->array[2].head->dest.point1, (string)"Saint-P");
			Assert::AreEqual(graph->array[2].head->dest.numb, 1);
		}
		TEST_METHOD(newheapnode)
		{
			int V = 9;
			int cost = 15;
			MinHeapNode *save = newMinHeapNode(V, cost);
			Assert::AreEqual(9, save->v);
			Assert::AreEqual(15, save->cost);
		}
		TEST_METHOD(newheap)
		{
			int V = 9;
			//int cost = 15;
			MinHeap *save = createMinHeap(V);
			Assert::AreEqual(9, save->capacity);
			Assert::AreEqual(0, save->size);
		}

		TEST_METHOD(swap)
		{
			int V = 9;
			int cost = 15;
			MinHeapNode *save = newMinHeapNode(V, cost);
			int V1 = 12;
			int cost1 = 17;
```

```
        MinHeapNode *save1 = newMinHeapNode(V1, cost1);
        swapMinHeapNode(&save, &save1);
        Assert::AreEqual(12, save->v);
        Assert::AreEqual(17, save->cost);
        Assert::AreEqual(9, save1->v);
        Assert::AreEqual(15, save1->cost);
}
TEST_METHOD(Heapify)
{
        int V = 9;
        int cost = 15;
        MinHeapNode *save = newMinHeapNode(V, cost);
        int V1 = 12;
        int cost1 = 17;
        MinHeapNode *save1 = newMinHeapNode(V1, cost1);
        MinHeap *test = createMinHeap(V);
        for (int v = 0; v < 5; v++)
        {
                test->array[v] = newMinHeapNode(v, v+1);
                test->pos[v] = v;
        }

        minHeapify(test,0);
        Assert::AreEqual(0, test->array[0]->v);
        Assert::AreEqual(1, test->array[0]->cost);
}

TEST_METHOD(Empty)
{
        int V = 9;
        int cost = 15;
        MinHeap *save = createMinHeap(V);
        bool test = isEmpty(save);
        Assert::AreEqual(test, true);
}
TEST_METHOD(Empty2)
{
        int V = 9;
        int cost = 15;
        MinHeap *save = createMinHeap(V);
        for (int i = 0;i < 5;i++) {
                save->array[i] = newMinHeapNode(i, i + 1);
                save->size++;
        }

        bool test = isEmpty(save);
        Assert::AreEqual(test, false);
}
TEST_METHOD(Extract)
{
        int V = 9;
        int cost = 15;
        MinHeap *save = createMinHeap(V);
        for (int i = 0;i < 5;i++) {
                save->array[i] = newMinHeapNode(i, i + 1);
                save->size++;
        }

        MinHeapNode *test = extractMin(save);
        Assert::AreEqual(test->v, 0);
        Assert::AreEqual(test->cost, 1);
}
TEST_METHOD(decrease)
{
```

```
		int V = 9;
		int cost = 15;
		MinHeap *save = createMinHeap(V);
		for (int i = 0;i < 5;i++) {
			save->array[i] = newMinHeapNode(i, i + 1);
			save->pos[i] = i;
			save->size++;
		}
		decreaseKey(save, 1, 2);
		Assert::AreEqual(save->array[0]->v, 0);
		Assert::AreEqual(save->array[0]->cost, 1);
}
TEST_METHOD(ismin)
{
		int V = 9;
		int cost = 15;
		MinHeap *save = createMinHeap(V);
		for (int i = 0;i < 5;i++) {
			save->array[i] = newMinHeapNode(i, i + 1);
			save->pos[i] = i;
			save->size++;
		}
		bool test = isInMinHeap(save, 1);
		Assert::AreEqual(test, true);
}

TEST_METHOD(finding_test)
{
		string test[5];
		test[0] = "zero";
		test[1] = "one";
		test[2] = "two";
		test[3] = "three";
		test[4] = "four";
//		test[5] = "five";
		string testing = finding(1, test);
		Assert::AreEqual(testing, (string)"one");
}

TEST_METHOD(checking_test)
{
		string test[5];
		test[0] = "zero";
		test[1] = "one";
		test[2] = "two";
		test[3] = "three";
		test[4] = "four";
		//		test[5] = "five";
		bool testing = checking(test, test[1]);
		Assert::AreEqual(testing, false);
}
TEST_METHOD(count_test)
{
		int test = count();
		Assert::AreEqual(5, test);
}
TEST_METHOD(checking3_test)
{
		string test[5];
		test[0] = "zero";
		test[1] = "one";
		test[2] = "two";
		test[3] = "three";
		test[4] = "four";
```

```
            //     test[5] = "five";
            bool testing = checking3(test, test[1]);
            Assert::AreEqual(testing, true);
        }
        TEST_METHOD(checking2_test)
        {
            string test[5];
            test[0] = "zero";
            test[1] = "one";
            test[2] = "two";
            test[3] = "three";
            test[4] = "four";
            //     test[5] = "five";
            int testing = checking2(test, "one");
            Assert::AreEqual(1, testing);
        }
};
                                        }
```

Вывод

При написании программы были улучшены знания ООП, а также изучена работа алгоритма Дейкстры.