**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчет

**по лабораторной работе №3**

**по дисциплине «Алгоритмы и структуры данных»**

Тема: Алгоритмы на графах

**Вариант 1**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9302 |  | Кнауб К.В. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2021

# Постановка задачи

Дан список возможных авиарейсов в текстовом файле в формате:

Город отправления 1;Город прибытия 1;цена прямого перелета 1;цена обратного перелета 1. Город отправления 2;Город прибытия 2;цена перелета 2;цена обратного перелета 1. Город отправления N;Город прибытия N;цена перелета N;цена обратного перелета N. В случае, если нет прямого или обратного рейса, его цена будет указана как N/A (not available). Найти наиболее эффективный по стоимости перелет из города i в город j.

# Описание реализуемого алгоритма и структур данных

В данной лабораторной работе написаны классы city и CityNode. сity требуется для хранения названия и цены перелета, а CityNode нужен для хранения городов и минимальной дистанции до каждого из городов из начального города.

|  |  |
| --- | --- |
| **Название** | **Описание** |
| T pop(); | добавление элемента в список |
| void push() | удаление элемента |
| int getSize() | получение размера списка |
| void sort() | сортировка списка |

|  |  |
| --- | --- |
| **Название** | **Описание** |
| bool setFunction () | Ищет города в списке |
| bool isAllVisited () | Проверяет все города |
| int minimalPath () | Ищет город с минимальным расстоянием из начального города |
| int getIndex() | Определяет индекс города по его названию |
| int isJoin () | Определяет, есть ли путь из начального города в конечный город |

# 1.Считываем входные данные из файла

# 2.Составляем список городов

# 3.Составляем список смежности

# 4.Запрашиваем начальный и конечный город

# 5.По алгоритму Дейкстры посещаем все узлы и высчитываем наименьшую стоимость перелета, проходя по каждой вершине один раз и заменяя большие стоимости на меньшие.

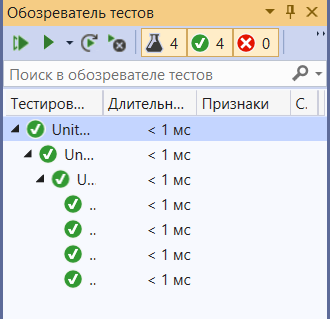
# 6.И в конце восстанавливаем путь

# Оценка временной сложности каждого алгоритма

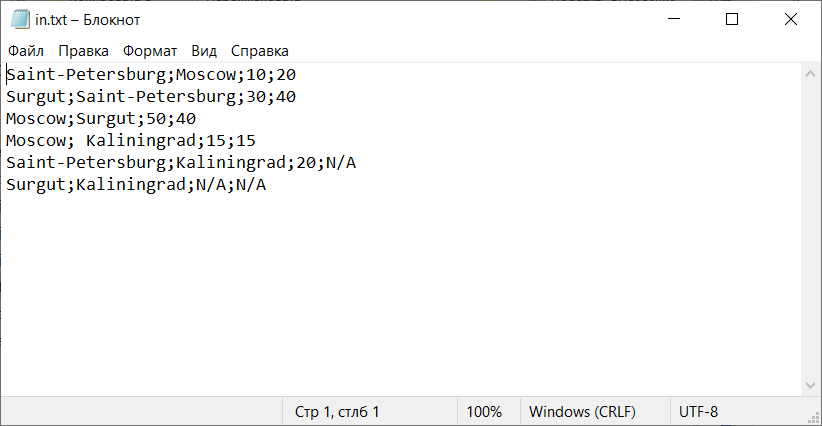
|  |  |
| --- | --- |
| bool setFunction () |  |
| bool isAllVisited () |  |
| int minimalPath () |  |
| int getIndex() |  |
| int isJoin () |  |

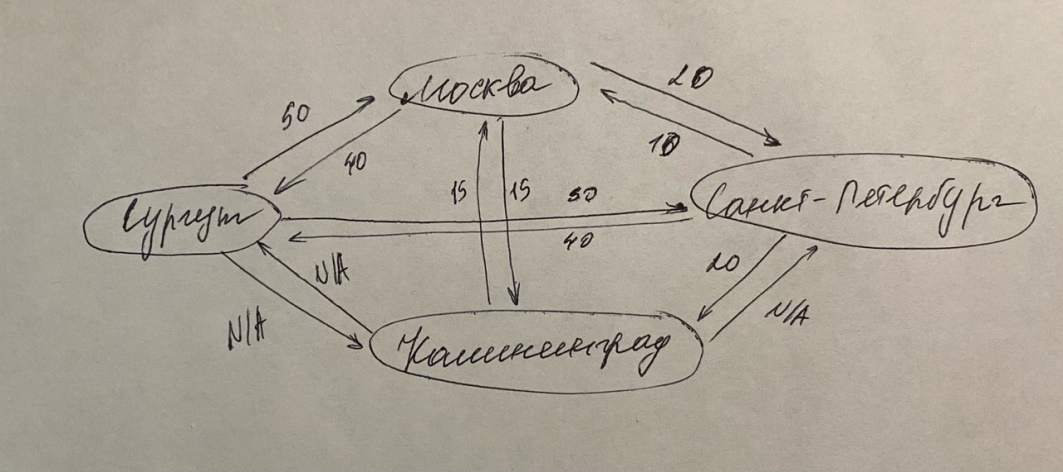
# Описание реализованных unit-тестов

|  |
| --- |
| **Название теста** |
| setFunction\_test |
| isAllVisited\_test |
| PushBack\_test |
| PopFront\_test |

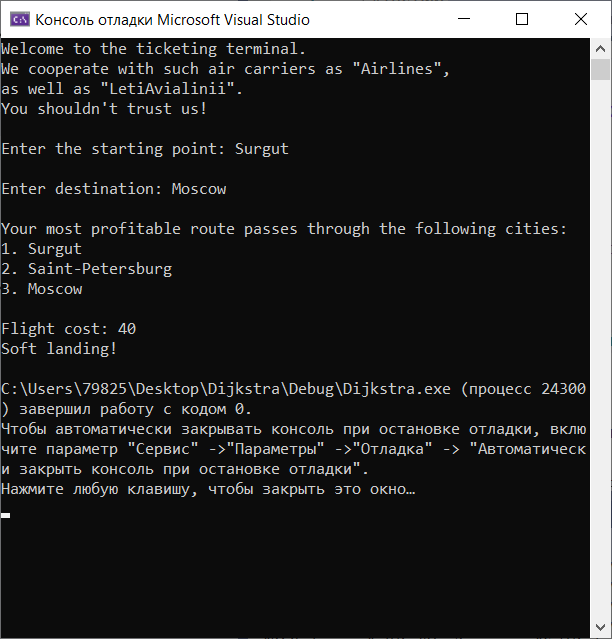


# Пример работы

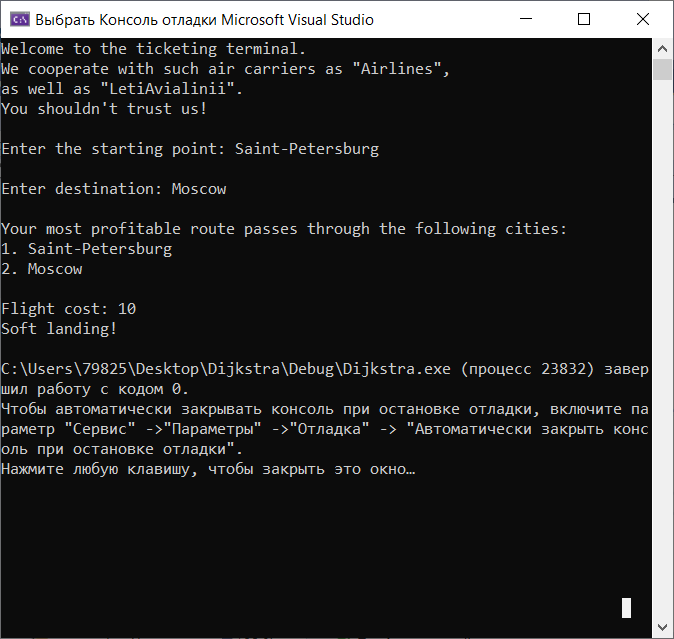




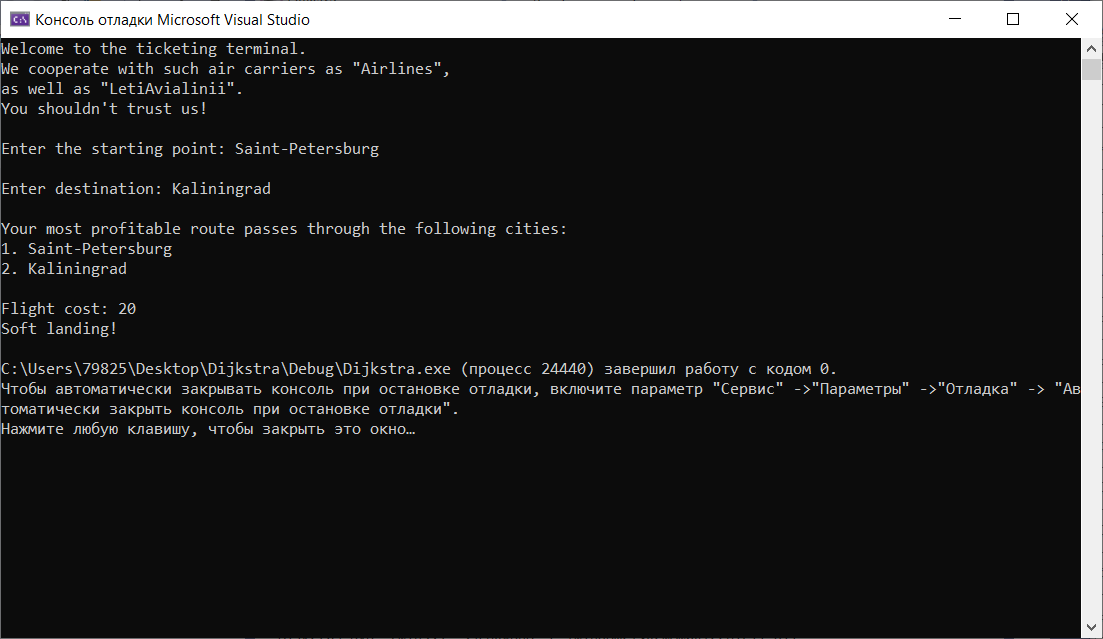
Кратчайший путь от Сургута до Москвы



Кратчайший путь от Санкт-Петербурга до Москвы



Кратчайший путь от Санкт-Петербурга до Калининграда



# Листинг

|  |
| --- |
| **Cities.h** |
| #ifndef CITIES\_H  #define CITIES\_H  #include <string>  using namespace std;  class city {  public:  city(string nameP, int costP) {  name = nameP;  cost = costP;  };  string name;  int cost;  };  class CityNode {  public:  CityNode() {};  CityNode(string nameP) {  name = nameP;  }  string name;  int disatnce;  bool isVisited;  };  #endif |
| **someFunction.h** |
| #ifndef SOMEFUNCTION\_H  #define SOMEFUNCTION\_H  #include <string>  #include "List.h"  #include "Cities.h"  using namespace std;  bool setFunction(List<string>& arr, string str) {  for (int i = 0; i < arr.getSize(); i++) {  string a = arr[i];  if (arr[i] == str)  return true;  }  return false;  }  bool isAllVisited(CityNode\* pcity, int count) {  for (int i = 0; i < count; i++) {  if (pcity[i].isVisited == false)  return false;  }  return true;  }  int minimalPath(CityNode\* pcity, int count) {  int index = -1;  int min = INT\_MAX;  for (int i = 0; i < count; i++) {  if (pcity[i].disatnce < min && pcity[i].isVisited == false) {  min = pcity[i].disatnce;  index = i;  }  }  return index;  }  int getIndex(string\* arr, string str) {  for (int i = 0; i < arr->size(); i++) {  if (arr[i] == str)  return i;  }  return -1;  }  int isJoin(string\* indices, List<city>\*\* contiguityList, int begin, int end) {  for (int i = 0; i < contiguityList[begin]->getSize(); i++)  if (getIndex(indices, (\*contiguityList[begin])[i].name) == end)  return (\*contiguityList[begin])[i].cost;  return -1;  }  #endif |
| **List.h** |
| #ifndef List\_H  #define List\_H  #include <string>  template<class T>  class List {  private:  class nodeList;  nodeList\* \_head;  int \_size;  public:  List();  ~List();  T pop();  T pop(int);  void push(T value);  int getSize();  void sort();  const T operator[](const int index) const;  };  template <class T>  class List<T>::nodeList {  private:  T \_value;  nodeList\* \_next;  public:  nodeList() : \_next(nullptr) {};  nodeList(T value, nodeList\* next) :  \_value(value), \_next(next) {}  T getValue() {  return \_value;  }  void setValue(T value) {  \_value = value;  }  nodeList\* getNext() {  return \_next;  }  void setNext(nodeList\* next) {  \_next = next;  }  };  template <class T>  List<T>::List() : \_head(nullptr), \_size(0) {}  template<class T>  List<T>::~List()  {  while (\_size > 0)  this->pop();  }  template <class T>  T List<T>::pop() {  if (\_size == 0)  throw;  if (\_size == 1) {  T tempNextVal = \_head->getValue();  delete \_head;  \_head = nullptr;  this->\_size = 0;  return tempNextVal;  }  nodeList\* temp = \_head;  while (temp->getNext()->getNext() != nullptr) temp = temp->getNext();  T tempNextVal = temp->getNext()->getValue();  delete temp->getNext();  \_size--;  temp->setNext(nullptr);  return tempNextVal;  }  template <class T>  T List<T>::pop(int index) {  if (\_size == 1 || \_size == 0 || index == \_size - 1) {  return this->pop();  }  if (index == 0) {  nodeList\* temp = \_head->getNext();  T tempvalue = \_head->getValue();  \_head = temp;  \_size--;  return tempvalue;  }  nodeList\* temp = \_head;  for (int i = 0; i < index - 1; i++)  temp = temp->getNext();  T tempNextVal = temp->getNext()->getValue();  nodeList\* tempNext = temp->getNext()->getNext();  \_size--;  temp->setNext(tempNext);  return tempNextVal;  }  template <class T>  void List<T>::push(T value) {  if (\_size == 0) {  \_head = new nodeList(value, nullptr);  \_size++;  return;  }  nodeList\* temp = \_head;  while (temp->getNext() != nullptr) temp = temp->getNext();  temp->setNext(new nodeList(value, nullptr));  \_size++;  return;  }  template <class T>  int List<T>::getSize() {  return \_size;  }  template <class T>  void List<T>::sort() {  if (\_size == 1 || \_size == 0)  return;  nodeList\* temp = \_head;  bool f = false;  for (int i = 0; i < \_size - 1; i++) {  temp = \_head;  f = false;  for (int j = 0; j < \_size - 1 - i; j++) {  if ((temp->getValue()).cost > (temp->getNext()->getValue()).cost) {  T tempVal = temp->getValue();  temp->setValue(temp->getNext()->getValue());  temp->getNext()->setValue(tempVal);  f = true;  }  temp = temp->getNext();  }  if (!f)  return;  }  }  template <class T>  const T List<T>::operator[](const int index) const{  nodeList\* it = \_head;  if (index < 0 || index > \_size)  throw;  for (int i = 0; i < index; i++) {  it = it->getNext();  }  return it->getValue();  }  #endif |
| **main.cpp** |
| #include <iostream>  #include <fstream>  #include <string>  #include "List.h"  #include "someFunction.h"  #include "Cities.h"  #define bigNumber INT\_MAX  using namespace std;  int main() {  cout << "Welcome to the ticketing terminal.\nWe cooperate with such air carriers as \"Airlines\", \nas well as \"LetiAvialinii\". \nYou shouldn't trust us!\n\n";  fstream file;  file.open("in.txt");  int countString = 0;  string tempString;  while (!file.eof()){  getline(file, tempString);  countString++;  }  file.close();  file.open("in.txt");  string\*\* inputInfo = new string\*[countString];  for (int i = 0; i < countString; i++) {  inputInfo[i] = new string[4];  for (int j = 0; j < 3; j++) {  getline(file, tempString, ';');  if (tempString != "N/A")  inputInfo[i][j] = tempString;  else  inputInfo[i][j] = "";  }  getline(file, tempString);  if (tempString != "N/A")  inputInfo[i][3] = tempString;  else  inputInfo[i][3] = "";  }  file.close();  List<string> cities;  for (int i = 0; i < countString; i++) {  if (!setFunction(cities, inputInfo[i][0])) cities.push(inputInfo[i][0]);  if (!setFunction(cities, inputInfo[i][1])) cities.push(inputInfo[i][1]);  }  List<city>\*\* contiguityList = new List<city>\*[cities.getSize()];  for (int i = 0; i < cities.getSize(); i++) {  city\* bufUno = new city(cities[i], 0);  List<city>\* bufTvino = new List<city>;  bufTvino->push(\*bufUno);  contiguityList[i] = bufTvino;  }  for (int i = 0; i < countString; i++) {  int j = 0;  if (inputInfo[i][2] != "") {  while (inputInfo[i][0] != (\*contiguityList[j])[0].name)  j++;  city\* temp = new city(inputInfo[i][1], stoi(inputInfo[i][2]));  (\*contiguityList[j]).push(\*temp);  }  j = 0;  if (inputInfo[i][3] != "") {  while (inputInfo[i][1] != (\*contiguityList[j])[0].name)  j++;  city\* temp = new city(inputInfo[i][0], stoi(inputInfo[i][3]));  (\*contiguityList[j]).push(\*temp);  }  }  for (int i = 0; i < cities.getSize(); i++) {  (\*contiguityList[i]).sort();  }  string main, destination;  cout << "Enter the starting point: ";  cin >> main;  cout << "\nEnter destination: ";  cin >> destination;  CityNode\* cityNode = new CityNode[cities.getSize()];  string\* indices = new string[cities.getSize()];  for (int i = 0; i < cities.getSize(); i++) {  indices[i] = (\*contiguityList[i])[0].name;  cityNode[i].name = (\*contiguityList[i])[0].name;  cityNode[i].isVisited = false;  cityNode[i].disatnce = bigNumber;  if ((\*contiguityList[i])[0].name == main)  cityNode[i].disatnce = 0;  }  while (!isAllVisited(cityNode, cities.getSize())) {  int curNodeIndex = minimalPath(cityNode, cities.getSize());  for (int i = 1; i < (\*contiguityList[curNodeIndex]).getSize(); i++ )  if (cityNode[getIndex(indices, (\*contiguityList[curNodeIndex])[i].name)].isVisited == false && cityNode[curNodeIndex].disatnce + (\*contiguityList[curNodeIndex])[i].cost < cityNode[getIndex(indices, (\*contiguityList[curNodeIndex])[i].name)].disatnce)  cityNode[getIndex(indices, (\*contiguityList[curNodeIndex])[i].name)].disatnce = cityNode[curNodeIndex].disatnce + (\*contiguityList[curNodeIndex])[i].cost;  else  continue;  cityNode[curNodeIndex].isVisited = true;  }  int\* ver = new int[cities.getSize()];  int end = getIndex(indices, destination);  int start = getIndex(indices, main);  ver[0] = end;  int k = 1;  int weight = cityNode[end].disatnce;  while (end != start)  for (int i = 0; i < cities.getSize(); i++)  if (isJoin(indices, contiguityList, i, end) != -1) {  int temp = weight - isJoin(indices, contiguityList, i, end);  if (temp == cityNode[i].disatnce) {  weight = temp;  end = i;  ver[k] = i;  k++;  }  }  cout << "\nYour most profitable route passes through the following cities: \n";  int countDown = 1;  for (int i = k - 1; i >= 0; i--) {  cout << countDown << ". " << indices[ver[i]] << "\n";  countDown++;  }  end = getIndex(indices, destination);  cout << "\nFlight cost: " << cityNode[end].disatnce;  cout << "\nSoft landing!\n";  } |
| **UnitTest3.cpp** |
| #include "pch.h"  #include "CppUnitTest.h"  #include "../Dijkstra/list.h"  #include <string>  #include "../Dijkstra/someFunction.h"  #include "../Dijkstra/Cities.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  using namespace std;  namespace UnitTest3  {  TEST\_CLASS(UnitTest3)  {  public:    TEST\_METHOD(setFunction\_test)  {  List<string> list;  string str;  list.push("one");  list.push("two");  list.push("thee");  str = ("two");  bool flag = setFunction(list, str);  Assert::IsTrue(flag);  }  TEST\_METHOD(isAllVisited\_test)  {  CityNode\* node = new CityNode[5];  for (int i = 0; i < 5; i++)  {  node[i].isVisited = true;  }  Assert::IsTrue(isAllVisited(node, 5));  }    TEST\_METHOD(PushBack\_test)  {  List<int> list;  list.push(1);  list.push(2);  list.push(3);  list.push(4);  Assert::IsTrue(list[2] == 3);  }  TEST\_METHOD(PopFront\_test)  {  List<int> list;  list.push(1);  list.push(2);  list.push(3);  list.push(4);  list.pop();  Assert::IsTrue(list[2] == 3);  }  };  } |