**МИНОБРНАУКИ РФ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА) “ЛЭТИ”**

**ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ И ИНФОРМАТИКИ**

**КАФЕДРА САПР**

**Отчёт по курсовой работе   
по дисциплине «Алгоритмы и структуры данных»**

Тема: «Алгоритмы поиска максимального потока в графе»

**Вариант 2.**

**Преподаватель: \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Тутуева А.В.**

**Студент гр.9302: \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Новокрещенов Д.К.**

**Санкт-Петербург**

**2021**

**Содержание**

[1. Постановка задачи 3](#_Toc74123476)

[2. Обоснование выбора используемых структур данных 3](#_Toc74123477)

[3. Описание алгоритма решения 3](#_Toc74123478)

[4. Пример работы программы 4](#_Toc74123479)

[5. Листинг программы 4](#_Toc74123480)

[ЗАКЛЮЧЕНИЕ 16](#_Toc74123481)

# Постановка задачи

Входные данные: текстовый файлы со строками в формате V1, V1, P, где V1, V2 направленная дуга транспортной сети, а P – ее пропускная способность. Исток всегда обозначен как S, сток – как T. Найти максимальный поток в сети используя алгоритм Эдмондса-Карпа.

# Обоснование выбора используемых структур данных

**class Graph**

|  |  |  |
| --- | --- | --- |
| **Компонент** | **Описание** | |
| int\*\* matrix\_c, \*\*matrix\_f | Матрицы, необходимые для работы алгоритма  C – матрица путей, F – матрица потоков | |
| int num\_node; | Количество узлов графа | |
| BiList<string> name\_node, info; | Списки, обеспечивающие работу алгоритма | |
| **Метод** | **Описание** | **Оценка временной сложности** |
| Graph(); | Конструктор |  |
| ~Graph(); | Деструктор |  |
| void create\_matrix(); | Создание матрицы |  |
| void input(string name); | Ввод информации из файла |  |
| int EdmondsKarp(); | Алгоритм поиска макс. потока |  |
| void clear(); | Очистка данных |  |
| n – количество узлов, m – количество рёбер (по реализации поиск рёбер O(n2) ) | | |

# Описание алгоритма решения

1. Обнуляем все потоки. Остаточная сеть изначально совпадает с исходной сетью.
2. В остаточной сети находим *кратчайший* путь из источника в сток. Если такого пути нет, останавливаемся.
3. Пускаем через найденный путь (он называется **увеличивающим путём** или **увеличивающей цепью**) максимально возможный поток:
   1. На найденном пути в остаточной сети ищем ребро с минимальной пропускной способностью  {\displaystyle c\_{\min }}Для каждого ребра на найденном пути увеличиваем поток на , а в противоположном ему — уменьшаем на {\displaystyle c\_{\min }} .
   2. Модифицируем остаточную сеть. Для всех рёбер на найденном пути, а также для противоположных им рёбер, вычисляем новую пропускную способность. Если она стала ненулевой, добавляем ребро к остаточной сети, а если обнулилась, стираем его.
4. Возвращаемся на шаг II.

# Пример работы программы

|  |  |
| --- | --- |
| **Граф** | **Результат** |
|  |  |
|  |  |

# Листинг программы

|  |
| --- |
| main.cpp |
| #include <iostream>  #include <vector>  #include "List.h"  #include "Graph.h"  using namespace std;  int main()  {  setlocale(LC\_ALL, "Rus");  Graph Test;  Test.input("test.txt");  cout << "Максимальный поток из S в T: " << Test.EdmondsKarp();  } |
| List.h |
| #pragma once  #include <cstddef>  #include <string>  using namespace std;  template<class Type>  class BiList  {  public:  BiList();  ~BiList();  void push\_back(Type);  void push\_front(Type);  void pop\_back();  void pop\_front();  void insert(Type, size\_t);  Type at(size\_t);  void remove(size\_t);  size\_t get\_size();  void print\_to\_console();  void clear();  void set(size\_t, Type);  bool isEmpty();  void reverse();  void swap(size\_t, size\_t);  int find(Type);  private:  struct Node  {  Type data;  Node\* prev = NULL;  Node\* next = NULL;  };  Node\* head;  Node\* last;  size\_t size;  }; |
| List.cpp |
| #pragma once  #include "List.h"  #include <iostream>  template<class Type>  BiList<Type>::BiList()  {  head = NULL;  last = NULL;  size = 0;  }  template<class Type>  BiList<Type>::~BiList()  {  while (head)  {  last = head->next;  delete head;  head = last;  }  }  //The function inserts a new node at back of the list.  template<class Type>  void BiList<Type>::push\_back(Type value)  {  Node\* temp = new Node;  temp->next = NULL;  temp->data = value;  if (!this->isEmpty())  {  temp->prev = last;  last->next = temp;  last = temp;  }  else {  temp->prev = NULL;  head = last = temp;  }  size++;  }  //The function inserts a new node at front of the list.  template<class Type>  void BiList<Type>::push\_front(Type value)  {  Node\* temp = new Node;  temp->prev = NULL;  temp->data = value;  if (!this->isEmpty())  {  temp->next = head;  head->prev = temp;  head = temp;  }  else  {  temp->prev = NULL;  temp->next = NULL;  head = last = temp;  }  size++;  }  //The function deletes the last node in the list.  template<class Type>  void BiList<Type>::pop\_back()  {  if(size==0)  return;  if (size == 1)  {  delete last;  last = head = NULL;  size--;  return;  }  delete last->next;  last = last->prev;  last->next = NULL;  size--;  }  //The function deletes the first node in the list.  template<class Type>  void BiList<Type>::pop\_front()  {  if (size == 0)  return;  if (size == 1)  {  delete head;  head = last = NULL;  size--;  return;  }  delete head->prev;  head->prev = NULL;  head = head->next;  size--;  }  //The function inserts a new node after the node with the number "index".  template<class Type>  void BiList<Type>::insert(Type value, size\_t index)  {  if ((size == 0 && index==0) || index >= size)  throw "Incorrect index.";  if (index == 0)  {  this->push\_front(value);  return;  }  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  Node\* temp = new Node;  temp->data = value;  temp->next = cursor;  temp->prev = cursor->prev;  cursor->prev = temp;  temp->prev->next = temp;  size++;  }  //The function gets the value from the node with the number "index".  template<class Type>  Type BiList<Type>::at(size\_t index)  {  if (index >= size || index<0)  throw "Incorrect index.";  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  return cursor->data;  }  //The function deletes the node with the "index" number.  template<class Type>  void BiList<Type>::remove(size\_t index)  {  if (index >= size || index <= 0)  throw "Incorrect index.";  if (index == 0)  {  this->pop\_front();  return;  }  if (index == size-1)  {  this->pop\_back();  return;  }  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  Node\* temp = cursor;  cursor->prev->next = cursor->next;  cursor->next->prev = cursor->prev;  delete temp;  size--;  }  //The function gets the number of items in the list  template<class Type>  size\_t BiList<Type>::get\_size()  {  return size;  }  //The function prTypes all list in console.  template<class Type>  void BiList<Type>::print\_to\_console()  {  Node\* cursor = head;  if (head)  {  while (cursor->next)  {  std::cout << cursor->data << "<-->";  cursor = cursor->next;  }  std::cout << cursor->data;  }  else throw "List is empty.";  }  //The function deletes all node in list.  template<class Type>  void BiList<Type>::clear()  {  while (head!=NULL)  this->pop\_front();  head = last = NULL;  }  //The function sets a new value in the node with the number "index"  template<class Type>  void BiList<Type>::set(size\_t index, Type value)  {  if (index >= size || index<0)  throw"Incorrect index.";  Node\* cursor = head;  for (size\_t i = 0; i < index; i++)  cursor = cursor->next;  cursor->data = value;  }  //The function checks the list for emptiness.  template<class Type>  bool BiList<Type>::isEmpty()  {  if (head==NULL) return true;  return false;  }  //The function reverses a list.  template<class Type>  void BiList<Type>::reverse()  {  if (!head || !head->next)  return;  last = head;  Node\* temp = NULL;  Node\* current = head;  while (current != NULL)  {  temp = current->prev;  current->prev = current->next;  current->next = temp;  current = current->prev;  }  head = temp->prev;  }  template<class Type>  void BiList<Type>::swap(size\_t first, size\_t second)  {  Node\* temp\_f = head, \* temp\_s = head;  for (size\_t i = 0; i < first; i++)  temp\_f = temp\_f->next;  for (size\_t i = 0; i < second; i++)  temp\_s = temp\_s->next;  Type temp = temp\_f->data;  temp\_f->data = temp\_s->data;  temp\_s->data = temp;  }  template<class Type>  int BiList<Type>::find(Type info)  {  Node\* cur = head;  int i = 0;  while (i < size)  {  if (cur->data == info)  return i;  cur = cur->next;  i++;  }  return -1;  } |
| Graph.h |
| #pragma once  #include "List.h"  #include <string>  using namespace std;  class Graph  {  private:  int\*\* matrix\_c, \*\*matrix\_f, num\_node;  BiList<string> name\_node, info;  void create\_matrix();  public:  Graph();  ~Graph();  void input(string name);  int EdmondsKarp();  void clear();  }; |
| Graph.cpp |
| #include "Graph.h"  #include "List.h"  #include "List.cpp"  #include <cstddef>  #include <iomanip>  #include <fstream>  #define inf INT\_MAX/2-1  Graph::Graph()  {  matrix\_c = NULL;  matrix\_f = NULL;  num\_node = 0;  }  Graph::~Graph()  {  clear();  }  void Graph::input(string filename)  {  if (filename == "")  filename = "test.txt";  string temp;  ifstream filereader;  filereader.open(filename.c\_str(), ios::in);  if (filereader.bad())  throw "Bad File Name";  while (!filereader.eof())  {  getline(filereader, temp);  info.push\_back(temp);  }  create\_matrix();  int n = num\_node;  }  int Graph::EdmondsKarp()  {  while(1)  {  int\* from = new int[num\_node];  int\* temp = new int[num\_node];  for (size\_t i = 0; i < num\_node; i++)  from[i] = -1;  int h = 0, t = 0;  temp[t++] = 0;  from[0] = 0;  for (int cur; h < t;)  {  cur = temp[h++];  for (size\_t v = 0; v < num\_node; v++)  if (from[v] == -1 && (matrix\_c[cur][v] - matrix\_f[cur][v]) > 0)  {  temp[t++] = v;  from[v] = cur;  }  }  if (from[num\_node-1] == -1)  break;  int cf = inf;  for (int cur = num\_node - 1; cur != 0; )  {  int prev = from[cur];  cf = min(cf, matrix\_c[prev][cur] - matrix\_f[prev][cur]);  cur = prev;  }  for (int cur = num\_node - 1; cur != 0; )  {  int prev = from[cur];  matrix\_f[prev][cur] += cf;  matrix\_f[cur][prev] -= cf;  cur = prev;  }  }  int flow = 0;  for (int i = 0; i < num\_node; i++)  if (matrix\_c[0][i])  flow += matrix\_f[0][i];  return flow;  }  void Graph::clear()  {  for (size\_t i = 0; i < num\_node; i++)  {  delete[] matrix\_c[i];  delete[] matrix\_f[i];  }  delete matrix\_c;  delete matrix\_f;  }  void Graph::create\_matrix()  {  BiList<string\*> all\_data;  for (size\_t i = 0; i < info.get\_size(); i++)  {  string temp = info.at(i);  string\* tmp = new string[3];  size\_t cur = 0;  for (size\_t j = 0; j < 3; j++)  {  while (cur< temp.size() && temp[cur]!= ' ')  {  tmp[j] += temp[cur];  cur++;  }  cur++;  }  all\_data.push\_back(tmp);  for (size\_t j = 0; j < 2; j++)  if (name\_node.find(tmp[j]) == -1)  name\_node.push\_back(tmp[j]);  }  try {  name\_node.swap(name\_node.find("S"), 0);  name\_node.swap(name\_node.find("T"), name\_node.get\_size()-1);  }  catch(char\*)  {  throw "Bad Data Error";  }  num\_node = name\_node.get\_size();  matrix\_c = new int\* [num\_node];  matrix\_f = new int\* [num\_node];  for (size\_t i = 0; i < num\_node; i++)  {  matrix\_c[i] = new int[num\_node];  matrix\_f[i] = new int[num\_node];  for (size\_t j = 0; j < num\_node; j++)  {  matrix\_c[i][j] = 0;  matrix\_f[i][j] = 0;  }  }  for (size\_t i = 0; i < all\_data.get\_size(); i++)  {  string\* temp = all\_data.at(i);  matrix\_c[name\_node.find(temp[0])][name\_node.find(temp[1])] = stoi(temp[2]);  }  } |
| AlgoTest.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include "../CourseWork/Graph.h"  #include "../CourseWork/List.h"  #include "../CourseWork/Graph.cpp"  #include "../CourseWork/List.cpp"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace ALgoTest  {  TEST\_CLASS(BiListTest)  {  public:  BiList<int> Empty;  BiList<int> NotEmpty;  TEST\_METHOD\_INITIALIZE(SetUp)  {  for (size\_t i = 0; i < 3; i++)  NotEmpty.push\_back(i + 1);  }  TEST\_METHOD(IsEmpty\_for\_NotEmpty)  {  Assert::AreEqual(NotEmpty.isEmpty(), false);  }  TEST\_METHOD(IsEmpty\_for\_Empty)  {  Assert::AreEqual(Empty.isEmpty(), true);  }  TEST\_METHOD(Get\_Size\_for\_Empty)  {  size\_t real\_size = 0;  Assert::AreEqual(Empty.get\_size(), real\_size);  }  TEST\_METHOD(Get\_Size\_for\_NotEmpty)  {  size\_t real\_size = 3;  Assert::AreEqual(NotEmpty.get\_size(), real\_size);  }  TEST\_METHOD(at\_correct\_index)  {  Assert::AreEqual(NotEmpty.at(2), 3);  }  TEST\_METHOD(at\_incorrect\_index)  {  try  {  Empty.at(4);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Incorrect index.");  }  }  TEST\_METHOD(Push\_Back\_NotEmpty)  {  NotEmpty.push\_back(5);  Assert::AreEqual(NotEmpty.at(3), 5);  }  TEST\_METHOD(Push\_Front\_NotEmpty)  {  NotEmpty.push\_front(0);  Assert::AreEqual(NotEmpty.at(0), 0);  }  TEST\_METHOD(remove\_correct\_index)  {  int temp = NotEmpty.at(2);  NotEmpty.remove(1);  Assert::AreEqual(NotEmpty.at(1), temp);  }  TEST\_METHOD(Pop\_Back\_for\_NotEmpty)  {  size\_t real\_size = NotEmpty.get\_size() - 1;  NotEmpty.pop\_back();  Assert::AreEqual(NotEmpty.get\_size(), real\_size);  }  TEST\_METHOD(Pop\_Back\_for\_Empty)  {  Empty.pop\_back();  Assert::AreEqual(Empty.isEmpty(), true);  }  TEST\_METHOD(Pop\_Front\_for\_NotEmpty)  {  size\_t real\_size = NotEmpty.get\_size() - 1;  NotEmpty.pop\_front();  Assert::AreEqual(NotEmpty.get\_size(), real\_size);  }  TEST\_METHOD(insert\_correct\_index)  {  NotEmpty.insert(4, 1);  Assert::AreEqual(NotEmpty.at(1), 4);  }  TEST\_METHOD(insert\_incorrect\_index)  {  try  {  Empty.insert(2, 0);  }  catch (const char\* error)  {  Assert::AreEqual("Incorrect index.", error);  }  }  TEST\_METHOD(remove\_incorrect\_index)  {  try  {  Empty.remove(2);  }  catch (const char\* error)  {  Assert::AreEqual("Incorrect index.", error);  }  }  TEST\_METHOD(reverse\_for\_NotEmpty)  {  int last\_inf = NotEmpty.at(NotEmpty.get\_size() - 1);  NotEmpty.reverse();  Assert::AreEqual(NotEmpty.at(0), last\_inf);  }  TEST\_METHOD(Pop\_Front\_for\_Empty)  {  Empty.pop\_front();  Assert::AreEqual(Empty.isEmpty(), true);  }  TEST\_METHOD(set\_correct\_index)  {  NotEmpty.set(2, 3);  Assert::AreEqual(NotEmpty.at(2), 3);  }  TEST\_METHOD(set\_incorrect\_index)  {  try  {  NotEmpty.set(10000000, 2);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Incorrect index.");  }  }  TEST\_METHOD(Test\_Clear\_NotEmpty)  {  NotEmpty.clear();  size\_t real\_size = 0;  Assert::AreEqual(NotEmpty.get\_size(), real\_size);  }  TEST\_METHOD(Test\_Clear\_Empty)  {  Empty.clear();  size\_t real\_size = 0;  Assert::AreEqual(Empty.get\_size(), real\_size);  }  TEST\_METHOD(Push\_Back\_Empty)  {  Empty.push\_back(5);  Assert::AreEqual(Empty.at(0), 5);  Empty.pop\_back();  }  TEST\_METHOD(Push\_Front\_Empty)  {  Empty.push\_front(0);  Assert::AreEqual(Empty.at(0), 0);  }  };  TEST\_CLASS(ALgoTest)  {  public:    TEST\_METHOD(HardTest)  {  setlocale(LC\_ALL, "Rus");  Graph Test;  Test.input("C:\\Users\\fastf\\Desktop\\ПРОГА\\4 семестр\\АиСД\\CourseWork\\CourseWork\\test.txt");  Assert::AreEqual(Test.EdmondsKarp(), 30);  }  TEST\_METHOD(EasyTest)  {  setlocale(LC\_ALL, "Rus");  Graph Test;  Test.input("C:\\Users\\fastf\\Desktop\\ПРОГА\\4 семестр\\АиСД\\CourseWork\\CourseWork\\easy.txt");  Assert::AreEqual(Test.EdmondsKarp(), 5);  }  };  } |

# ЗАКЛЮЧЕНИЕ

Для написания данной курсовой работы потребовались знания, полученные за четвертый семестр обучения. Программа успешно выполняет поиск максимального потока в графе. При ошибочном вводе программа сообщает об этом пользователю.