**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по курсовой работе**

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

Вариант 2

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Ковтун А.С. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2021

## Постановка задачи. Описание класса и методов

Найти максимальный поток в сети, используя алгоритм Эдмондса-Карпа.

Сетью называется орграф без циклов с помеченными вершинами и дугами. Числа, которыми помечаются дуги сети, называются пропускными способностями дуг.

Примеры вершин сети: перекрёстки дорог, телефонные узлы, железнодорожные узлы, аэропорты, склады и т.д. Примеры дуг сети: дороги, трубы, телефонные и железнодорожные линии и т.д.

Сеть, у которой существует ровно один исток (s) и один сток(t), называется транспортной сетью.

В теории оптимизации и теории графов, задача о максимальном потоке заключается в нахождении такого потока по транспортной сети, что сумма потоков из истока, или, что то же самое, сумма потоков в сток максимальна.

## Оценка временной сложности алгоритмов

Функция readList(string fileName) считывает данные из файла, имя которого передаётся в функцию как параметр. Сложность данного алгоритма равна O(E), где E — число рёбер в исходном графе, поскольку функция считывает данные о каждом ребре графа.

Функция MaxFlow() находит размер максимального потока через данную сеть. Сложность данного алгоритма равна O(V\*E^2), где E — число рёбер в исходном графе, а V — число вершин.

Функция findFlow() находит путь и его максимальный поток. Сложность данного алгоритма равна O(E), где E — число рёбер в исходном графе. Функция MaxFlow() в цикле вызывает функцию findFlow(), пока остались пути с ненулевым потоком.

## Реализованные Unit-тесты

Реализованные Unit-тесты проверяют работу алгоритма на примере из условия задачи:

**Входные данные**

Содержимое входного файла.

4 5

S B 500

S C 1000

B C 1

B T 1000

C T 500

Результат работы программы: размер максимального потока — 1000.

**Код программы**

|  |
| --- |
| Main.cpp  #include <iostream>  #include"EdKarp.h"  int main()  {  EdKarp\* t = new EdKarp();  t->readList("in.txt");  cout << "MaxFlow is " << t->MaxFlow();  } |
| EdKarp.cpp |
| #include "EdKarp.h"  #include <fstream>  void EdKarp::readList(string fileName) {  fstream file;  char firstName, secondName;  int weight;  file.open(fileName, ios::in);  int countStr;  file >> Vertices >> countStr;  if (file.eof() || Vertices == 0) throw out\_of\_range("Graph is empty");  VerticesName = new char[Vertices];  int CountVerticesName = 0;  int\*\* arr\_check;  arr\_check = (int\*\*)malloc(sizeof(int\*) \* Vertices);  if (!arr\_check) {  throw std::out\_of\_range("Allocation error");  return;  }  else {  Graph\_Matrix = arr\_check;  }  int\*\* arr\_check\_1;  arr\_check\_1 = (int\*\*)malloc(sizeof(int\*) \* Vertices);  if (!arr\_check\_1) {  throw std::out\_of\_range("Allocation error");  return;  }  else {  maxStream = arr\_check\_1;  }  for (size\_t i = 0; i < Vertices; i++)  {  arr\_check[i] = (int\*)malloc(Vertices \* sizeof(int));  arr\_check\_1[i] = (int\*)malloc(Vertices \* sizeof(int));  if (!arr\_check[i]) { throw std::out\_of\_range("Allocation error"); return; }  else Graph\_Matrix[i] = arr\_check[i];  if (!arr\_check\_1[i]) { throw std::out\_of\_range("Allocation error"); return; }  else maxStream[i] = arr\_check\_1[i];  for (size\_t j = 0; j < Vertices; j++) {  Graph\_Matrix[i][j] = 0;  maxStream[i][j] = 0;  }  }  int found1, found2;  for (size\_t i = 0; i < countStr; i++)  {  file >> firstName >> secondName >> weight;  found1 = -1;  for (size\_t j = 0; j < CountVerticesName; j++)  if (VerticesName[j] == firstName)  found1 = j;  if (found1 == -1)  {  VerticesName[CountVerticesName] = firstName;  found1 = CountVerticesName;  CountVerticesName++;  }  found2 = -1;  for (size\_t j = 0; j < CountVerticesName; j++)  if (VerticesName[j] == secondName)  found2 = j;  if (found2 == -1)  {  VerticesName[CountVerticesName] = secondName;  found2 = CountVerticesName;  CountVerticesName++;  }  Graph\_Matrix[found1][found2] = weight;  }  for (size\_t i = 0; i < Vertices; i++)  {  if (VerticesName[i] == 'S')  from = i;  else if (VerticesName[i] = 'T')  to = i;  }  }  int EdKarp::MaxFlow() {  resultStream = 0;  int toAdd;  do  {  toAdd = findFlow();  cout << toAdd << endl;  resultStream += toAdd;  } while (toAdd > 0);  return resultStream;  }  int EdKarp::findFlow() {  dList path;  path.push\_back(from);  int\* flow = new int[Vertices];  int\* beforeVer = new int[Vertices];  for (size\_t i = 0; i < Vertices; i++)  {  flow[i] = 0;  beforeVer[i] = 0;  }  beforeVer[to] = -1;  flow[from] = INT\_MAX;  int curVer;  while ((beforeVer[to] == -1) && (!path.isEmpty()))  {  curVer = path.at(0);  for (size\_t i = 0; i < Vertices; i++)  {  if (((Graph\_Matrix[curVer][i] - maxStream[curVer][i]) > 0) && (flow[i] == 0))  {  path.push\_back(i);  beforeVer[i] = curVer;  if ((Graph\_Matrix[curVer][i] - maxStream[curVer][i]) < flow[curVer])  flow[i] = Graph\_Matrix[curVer][i];  else  flow[i] = flow[curVer];  }  }  path.pop\_front();  }  if (beforeVer[to] == -1)  return 0;  curVer = to;  while (curVer != from)  {  maxStream[beforeVer[curVer]][curVer] += flow[to];  curVer = beforeVer[curVer];  }  return flow[to];  } |
| DualList.cpp |
| #include "DualList.h"  using namespace std;  void dList::push\_back(int add)  {  if (size == 0) //if list is empty  {  head = new Node(add); //create head  tail = head;  }  else  {  Node\* curr = new Node(add);  curr->before = tail;  tail->next = curr;  tail = curr; //set new tail  }  size++;  }  void dList::push\_front(int add)  {  if (size == 0)  {  head = new Node(add);  tail = head;  }  else  {  Node\* curr = new Node(add);  curr->next = head;  head->before = curr;  head = curr; //set new head  }  size++;  }  size\_t dList::get\_size()  {  return size;  }  void dList::pop\_back()  {  if (size == 1)  {  size = 0; //setup zero position  head = NULL;  tail = NULL;  }  else if (size > 1) //if list is not empty  {  tail = tail->before; //set new tail  delete tail->next;  tail->next = NULL;  size--;  }  else //else return error  throw out\_of\_range("Segmentation fault");  }  void dList::pop\_front()  {  if (size == 1)  {  size = 0; //setup zero position  head = NULL;  tail = NULL;  }  else if (size > 1) //if list is not empty  {  head = head->next; //set new head  delete head->before;  head->before = NULL;  size--;  }  else //else return error  throw out\_of\_range("Segmentation fault");  }  void dList::remove(size\_t i)  {  if (i >= size) //if index is larger than size return error  throw out\_of\_range("Index is greater than list size");  else if (i == 0) // if index points to first  pop\_front();  else if (i == size - 1) // if index points to first  pop\_back();  else  {  Node\* cur = head;  while (i) //go to "i" item  {  cur = cur->next;  i--;  }  cur->before->next = cur->next; //cut item from list  size--;  delete cur;  }  }  int dList::at(size\_t i)  {  if (i >= size) //if index is larger than size return error  throw out\_of\_range("Index is greater than list size");  else if (i == size - 1)  {  return tail->content;  }  else {  Node\* cur = head;  while (i) //go to "i" item  {  cur = cur->next;  i--;  }  return cur->content;  }  }  void dList::insert(int add, size\_t i)  {  if (i >= size) //if index is larger than size return error  throw out\_of\_range("Index is greater than list size");  else if (i == 0) //if insert before first item  push\_front(add);  else  {  Node\* cur = head;  while (i)  {  cur = cur->next;  i--;  }  Node\* nadd = new Node(add, cur->before, cur); //create new Node with new connection  cur->before->next = nadd; //add poin to new Node  cur->before = nadd;  size++;  }  }  void dList::print\_to\_console()  {  Node\* cur = head;  while (cur != NULL)  {  cout << cur->content << "| ";  cur = cur->next;  }  }  void dList::clear()  {  if (head != NULL)  {  while (head->next != NULL) //delete all item  {  head = head->next;  delete head->before;  }  size = 0; //setup zero position  head = NULL;  tail = NULL;  }  }  void dList::set(size\_t i, int ins)  {  if (i >= size) //if index is larger than size return error  throw out\_of\_range("Index is greater than list size");  else {  Node\* cur = head;  while (i) //go to "i" item  {  cur = cur->next;  i--;  }  cur->content = ins; //change content  }  }  bool dList::isEmpty()  {  if (head == NULL)  return true;  return false;  }  void dList::push\_back(dList AddToEnd)  {  Node\* head\_AddToEnd = AddToEnd.head;  while (head\_AddToEnd != NULL)  {  this->push\_back(head\_AddToEnd->content); //push new item to the end with content from transmitted list  head\_AddToEnd = head\_AddToEnd->next;  }  }  Iterator\* dList::create\_iterator(int start) {  return new dualList\_Iterator(head, start, size);  }  bool dList::dualList\_Iterator::has\_next() {  return (current != NULL);  }  int dList::dualList\_Iterator::next() {  if (!has\_next()) {  throw std::out\_of\_range("No more elements");  }  int temp = current->content;  current = current->next;  return temp;  } |
| Iterator.h |
| #pragma once  class Iterator  {  public:  virtual int next() = 0;  virtual bool has\_next() = 0;  }; |
| DualList.h |
| #pragma once  #include "Iterator.h"  #include <iostream>  using namespace std;  class dList  {  private:  class Node  {  public:  Node(int content = 0, Node\* before = NULL, Node\* next = NULL)  {  this->content = content;  this->before = before;  this->next = next;  };  ~Node()  { }  int content;  Node\* next;  Node\* before;  private:  };  Node\* head; //head list  Node\* tail; //tail list  size\_t size; //list size  public:  dList(Node\* head = NULL, Node\* tail = NULL) {  this->head = head;  this->tail = tail;  size = 0;  };  void push\_back(int); // add item to end  void push\_front(int); //add to begin  size\_t get\_size(); // get list size  void pop\_back(); // delete last item  void pop\_front(); // delete first item  int at(size\_t); // get item by index  void remove(size\_t); // delete item by index  void insert(int, size\_t); // add item before "size\_t" item  void print\_to\_console(); // print item to console  void clear(); // clear list  void set(size\_t, int); // change "size\_t" item to "int"  bool isEmpty(); // test for emptiness  void push\_back(dList); // add another list to end  Iterator\* create\_iterator(int); // iterator  class dualList\_Iterator : public Iterator  {  private:  Node\* current;  int size;  public:  dualList\_Iterator(Node\* head, int start, int max) {  size = max;  current = head;  int i = 0;  if (head != NULL)  {  if (start >= size)  {  throw out\_of\_range("Index is greater than list size");  }  while ((i < start) && (current->next != NULL))  current = current->next;  }  };  int next();  bool has\_next();  ~dualList\_Iterator() {  delete current;  }  };  ~dList()  {  while (head->next != NULL) //while we can go next  {  head = head->next;  delete head->before;  }  size = 0;  delete head;  };  }; |
| EdKarp.h |
| #pragma once  #include "DualList.h"  class EdKarp  {  private:  int Vertices;  int\*\* Graph\_Matrix; //and also the capacity of the edges of the graph  int\*\* maxStream;  int from, to;  int resultStream;  char\* VerticesName;  int findFlow();  public:  void readList(string);  int MaxFlow();  }; |

**Результат программы**



Рисунок 1. Вывод программы.