**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»**

Тема: Потоки в сетях

|  |  |  |
| --- | --- | --- |
| Студент гр. 0302 |  | Касаткин А.А. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2022

Задача: Найти максимальный поток, используя алгоритм Эдмондса-Карпа

Описание алгоритма:

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 }}
   2. Для каждого ребра на найденном пути увеличиваем поток на {\displaystyle c\_{\min }}, а в противоположном ему — уменьшаем на {\displaystyle c\_{\min }}.
   3. 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 2.

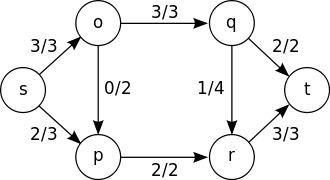
Чтобы найти кратчайший путь в графе, используем поиск в ширину:

1. Создаём очередь вершин *О*. Вначале *О* состоит из единственной вершины *s*.
2. Отмечаем вершину *s* как посещённую, без родителя, а все остальные как не посещённые.
3. Пока очередь не пуста, выполняем следующие шаги:
   1. Удаляем первую в очереди вершину *u*.
   2. Для всех дуг (*u*, *v*), исходящих из вершины *u*, для которых вершина *v* ещё не посещена, выполняем следующие шаги:
      1. Отмечаем вершину *v* как посещённую, с родителем *u*.
      2. Добавляем вершину *v* в конец очереди.
      3. Если *v* = *t*, выходим из обоих циклов: мы нашли кратчайший путь.
4. Если очередь пуста, возвращаем ответ, что пути нет вообще и останавливаемся.
5. Если нет, идём от *t* к *s*, каждый раз переходя к родителю. Возвращаем путь в обратном порядке.

Оценка временной сложности:

Maxflow O(VE)

Результат:

S O 3

S P 3

O Q 3

O P 2

P R 2

Q R 4

Q T 2

R T 3



Листинг:

**Main.cpp**

#include "Karp.h"

#include <fstream>

int main() {

Matrix<int> M;

Matrix<int> adj;

List<char> V;

{

ifstream file("input.txt");

if (!file.is\_open())

throw runtime\_error("File was not found");

int symb\_num = 0;

char symb;

int cap;

char v1, v2;

while (!file.eof()) {

file >> noskipws >> symb;

if ((symb\_num == 0 || symb\_num == 2) && !V.contains(symb))

V.push\_back(symb);

symb\_num++;

if (symb == '\n' || file.eof())

symb\_num = 0;

}

if (V.at(0) != 'S')

throw invalid\_argument("Graph should start with 'S'-vertex");

if (V.at(V.get\_size() - 1) != 'T')

throw invalid\_argument("Graph should finish with 'T'-vertex");

file.clear();

file.seekg(0);

List<int> temp\_list;

for (int i = 0; i < V.get\_size(); i++) {

for (int j = 0; j < V.get\_size(); j++)

temp\_list.push\_back(0);

M.push\_back(temp\_list);

temp\_list.reset();

}

while (!file.eof()) {

if (symb\_num != 4) {

file >> noskipws >> symb;

if (symb\_num == 0)

v1 = symb;

else if (symb\_num == 2)

v2 = symb;

}

else {

file >> cap;

M.at(V.get\_index((char)v1)).set(V.get\_index((char)v2), cap);

}

symb\_num++;

if (symb == '\n' || file.eof())

symb\_num = 0;

}

for (int i = 0; i < M.get\_size(); i++) {

temp\_list.reset();

for (int j = 0; j < M.at(i).get\_size(); j++) {

temp\_list.push\_back(M.at(i).at(j));

}

adj.push\_back(temp\_list);

}

for (int i = 0; i < adj.get\_size(); i++) {

for (int j = i; j < adj.at(i).get\_size(); j++) {

adj.at(j).set(i, adj.at(i).at(j));

}

}

temp\_list.clear();

}

cout << "Max flow: " << maxflow(V, M, adj);

return 0;

}

**Class.h**

#include <stdexcept>

#include <cstdio>

using namespace std;

template <class T>

class Node {

private:

Node\* next;

T data;

public:

Node() {

next = nullptr;

}

explicit Node(const T value) {

next = nullptr;

data = value;

}

~Node() {

next = nullptr;

}

void set\_next(Node\* const next\_el) {

next = next\_el;

};

void set\_data(T new\_data) {

data = new\_data;

};

Node\* get\_next() {

return next;

};

T get\_data() {

return data;

};

void clr\_next() {

delete this;

next = nullptr;

};

};

template <class T>

class Queue {

private:

Node<T>\* front;

Node<T>\* back;

Node<T>\* temp;

size\_t queue\_size;

public:

Queue() {

front = nullptr;

back = nullptr;

temp = nullptr;

queue\_size = 0;

}

~Queue() {

while (queue\_size != 0) {

this->takeTop();

}

front = nullptr;

back = nullptr;

temp = nullptr;

queue\_size = 0;

}

Node<T>\* get\_front() {

return front;

}

Node<T>\* get\_back() {

return back;

}

void push(T data) {

temp = new Node<T>(data);

if (queue\_size == 0) {

front = temp;

back = temp;

temp = nullptr;

queue\_size++;

return;

}

back->set\_next(temp);

back = back->get\_next();

temp = nullptr;

queue\_size++;

}

T takeTop() {

if (queue\_size == 0) {

throw std::out\_of\_range("The queue is empty");

}

T front\_data;

front\_data = front->get\_data();

if (queue\_size == 1) {

front = nullptr;

back = nullptr;

temp = nullptr;

queue\_size = 0;

return front\_data;

}

temp = front;

front = front->get\_next();

temp->clr\_next();

temp = nullptr;

queue\_size--;

return front\_data;

};

T getTop() {

if (queue\_size == 0) {

throw std::out\_of\_range("The queue is empty");

}

return front->get\_data();

}

size\_t get\_size() {

return queue\_size;

}

};

**EK.h**

#pragma once

#include <fstream>

#include "Class.h"

#include "List.h"

#include "Matrix.h"

using namespace std;

class Edmonds\_Karp

{

public:

struct pair {

int first;

long second;

};

List<char> listofvertexes;

Matrix<int> capacity;

Matrix<int> adj;

long INF = 100000;

~Edmonds\_Karp() {

capacity.clear();

adj.clear();

listofvertexes.clear();

}

int maxflow() {

if (listofvertexes.isEmpty())

throw invalid\_argument("Please enter data for the graph");

int maximum\_flow = 0;

List<int> parent(capacity.get\_size(), 0);

int new\_flow;

if (bfs(0, listofvertexes.get\_size() - 1, parent) == 0)

throw invalid\_argument("There are no ways from S to T at all");

while (new\_flow = bfs(0, listofvertexes.get\_size() - 1, parent)) {

maximum\_flow += new\_flow;

int cur = listofvertexes.get\_size() - 1;

while (cur != 0) {

int prev = parent.at(cur);

capacity.at(prev).set(cur, capacity.at(prev).at(cur) - new\_flow);

capacity.at(cur).set(prev, capacity.at(cur).at(prev) + new\_flow);

cur = prev;

}

}

return maximum\_flow;

}

void input(string input) {

ifstream file(input);

if (!file.is\_open())

throw runtime\_error("File was not found, check its name/location");

int symb\_num = 0;

char symb;

int cap;

char first\_vrtx, scnd\_vrtx;

//reading the vertexes list

while (!file.eof()) {

file >> noskipws >> symb;

if ((symb\_num == 0 || symb\_num == 2) && !listofvertexes.contains(symb))

listofvertexes.push\_back(symb);

symb\_num++;

if (symb == '\n' || file.eof())

symb\_num = 0;

}

if (listofvertexes.at(0) != 'S' || listofvertexes.at(listofvertexes.get\_size() - 1) != 'T')

throw invalid\_argument("Please start your graph with S and finish with T");

file.clear();

file.seekg(0);

List<int> temp;

for (int i = 0; i < listofvertexes.get\_size(); i++) {

for (int j = 0; j < listofvertexes.get\_size(); j++)

temp.push\_back(0);

capacity.push\_back(temp);

temp.reset();

}

//reading the avaliable capacity

while (!file.eof())

{

if (symb\_num != 4) {

file >> noskipws >> symb;

if (symb\_num == 0)

first\_vrtx = symb;

else if (symb\_num == 2)

scnd\_vrtx = symb;

}

else {

file >> cap;

capacity.at(listofvertexes.get\_index((char)first\_vrtx)).set(listofvertexes.get\_index((char)scnd\_vrtx), cap);

}

symb\_num++;

if (symb == '\n' || file.eof())

symb\_num = 0;

}

for (int i = 0; i < capacity.get\_size(); i++) {

temp.reset();

for (int j = 0; j < capacity.at(i).get\_size(); j++) {

temp.push\_back(capacity.at(i).at(j));

}

adj.push\_back(temp);

}

for (int i = 0; i < adj.get\_size(); i++) {

for (int j = i; j < adj.at(i).get\_size(); j++) {

adj.at(j).set(i, adj.at(i).at(j));

}

}

temp.reset();

List<int> parent(capacity.get\_size(), 0);

}

int bfs(int s, int t, List<int>& parent) {

//s - source, t - sink

for (int i = 0; i < parent.get\_size(); i++)

parent.set(i, -1);

parent.set(s, -2);

Queue <pair> queue;

queue.push({ s, INF });

while (queue.get\_size() != 0) {

int cur = queue.getTop().first;

int flow = queue.getTop().second;

queue.takeTop();

for (int next = 0; next < adj.at(cur).get\_size(); next++) {

if (parent.at(next) == -1 && capacity.at(cur).at(next)) {

parent.set(next, cur);

int new\_flow;

if (capacity.at(cur).at(next) < flow)

new\_flow = capacity.at(cur).at(next);

else

new\_flow = flow;

if (next == t)

return new\_flow;

queue.push({ next, new\_flow });

}

}

}

//if there are no ways from s to t

return 0;

}

};

**Karp.h**

#pragma once

#include "Matrix.h"

#include "Class.h"

#include "List.h"

#include <fstream>

long INF = 100000;

int bfs(int s, int t, Matrix<int>& M, Matrix<int>& adj, List<int>& temp\_list) {

for (int i = 0; i < temp\_list.get\_size(); i++)

temp\_list.set(i, -1);

temp\_list.set(s, -2);

struct pair {

long first;

long second;

};

Queue <pair> queue;

queue.push({ s, INF });

while (queue.get\_size() != 0) {

int curr = queue.getTop().first;

int flow = queue.getTop().second;

queue.takeTop();

for (int next = 0; next < adj.at(curr).get\_size(); next++) {

if (temp\_list.at(next) == -1 && M.at(curr).at(next)) {

temp\_list.set(next, curr);

int new\_flow;

if (M.at(curr).at(next) < flow)

new\_flow = M.at(curr).at(next);

else

new\_flow = flow;

if (next == t)

return new\_flow;

queue.push({ next, new\_flow });

}

}

}

return 0;

}

int maxflow(List<char>& V, Matrix<int>& M, Matrix<int>& adj) {

if (V.isEmpty())

throw invalid\_argument("Graph is empty");

int max\_flow = 0;

List<int> parent(M.get\_size(), 0);

int new\_flow;

if (bfs(0, V.get\_size() - 1, M, adj, parent) == 0)

throw invalid\_argument("No path S->T");

while (new\_flow = bfs(0, V.get\_size() - 1, M, adj, parent)) {

max\_flow += new\_flow;

int cur = V.get\_size() - 1;

while (cur != 0) {

int prev = parent.at(cur);

M.at(prev).set(cur, M.at(prev).at(cur) - new\_flow);

M.at(cur).set(prev, M.at(cur).at(prev) + new\_flow);

cur = prev;

}

}

return max\_flow;

}

List.h

#pragma once

#include <stdexcept>

#include <iostream>

using namespace std;

template<class T>

class List

{

private:

class Node

{

private:

T data;

Node\* next, \* prev;

public:

Node() : next(nullptr), prev(nullptr) {};

Node(T data) {

this->data = data;

next = nullptr;

prev = nullptr;

}

~Node() {

next = nullptr;

prev = nullptr;

}

void set\_data(T data) {

this->data = data;

}

T get\_data() {

return data;

}

Node\* get\_next() {

return next;

}

Node\* get\_prev() {

return prev;

}

void set\_next(Node\* pointer) {

next = pointer;

}

void set\_prev(Node\* pointer) {

prev = pointer;

}

};

Node\* head, \* tail;

Node\* get\_pointer(size\_t index)

{

if (isEmpty() || (index > get\_size() - 1))

{

throw out\_of\_range("Invalid argument");

}

else if (index == get\_size() - 1)

return tail;

else if (index == 0)

return head;

else

{

Node\* temp = head;

while ((temp) && (index--))

{

temp = temp->get\_next();

}

return temp;

}

}

public:

List() : head(nullptr), tail(nullptr) {}

List(int size, int value) {

while (size--) {

push\_back(value);

}

}

List(const List<T>& list) {

head = nullptr;

tail = nullptr;

Node\* temp = list.head;

while (temp) {

push\_back(temp->get\_data());

temp = temp->get\_next();

}

}

~List()

{

while (head)

{

tail = head->get\_next();

delete head;

head = tail;

}

head = nullptr;

}

void push\_back(T data)

{

Node\* temp = new Node;

temp->set\_data(data);

if (head)

{

temp->set\_prev(tail);

tail->set\_next(temp);

tail = temp;

}

else

{

head = temp;

tail = head;

}

}

void push\_front(T data)

{

Node\* temp = new Node;

temp->set\_data(data);

if (head)

{

temp->set\_next(head);

head->set\_prev(temp);

head = temp;

}

else

{

head = temp;

tail = head;

}

}

void push\_back(List<bool> ls2)

{

Node\* temp = ls2.head;

size\_t n = ls2.get\_size();

while ((temp) && (n--))

{

push\_back(temp->get\_data());

temp = temp->get\_next();

}

}

void push\_front(List& ls2)

{

Node\* temp = ls2.tail;

size\_t n = ls2.get\_size();

while ((temp) && (n--))

{

push\_front(temp->get\_data());

temp = temp->get\_prev();

}

}

void pop\_back()

{

if (head != tail)

{

Node\* temp = tail;

tail = tail->get\_prev();

tail->set\_next(nullptr);

delete temp;

}

else if (!isEmpty())

{

Node\* temp = tail;

tail = head = nullptr;

delete temp;

}

else

throw out\_of\_range("The list is empty");

}

void pop\_front()

{

if (head != tail)

{

Node\* temp = head;

head = head->get\_next();

head->set\_prev(nullptr);

delete temp;

}

else if (!isEmpty())

{

Node\* temp = head;

head = tail = nullptr;

delete temp;

}

else

throw out\_of\_range("The list is empty");

}

void insert(size\_t index, T data)

{

Node\* temp;

temp = get\_pointer(index);

if (temp == head)

push\_front(data);

else

{

Node\* newElem = new Node;

newElem->set\_data(data);

temp->get\_prev()->set\_next(newElem);

newElem->set\_prev(temp->get\_prev());

newElem->set\_next(temp);

temp->set\_prev(newElem);

}

}

T at(size\_t index)

{

Node\* temp;

temp = get\_pointer(index);

return temp->get\_data();

}

void remove(size\_t index)

{

Node\* temp;

temp = get\_pointer(index);

if (temp == head)

pop\_front();

else if (temp == tail)

pop\_back();

else

{

temp->get\_prev()->set\_next(temp->get\_next());

temp->get\_next()->set\_prev(temp->get\_prev());

delete temp;

}

}

void remove(T data) {

Node\* temp = head;

while (temp && temp->get\_data() != data)

temp = temp->get\_next();

if (!temp)

throw out\_of\_range("Invalid argument");

if (temp == head)

pop\_front();

else if (temp == tail)

pop\_back();

else

{

temp->get\_prev()->set\_next(temp->get\_next());

temp->get\_next()->set\_prev(temp->get\_prev());

delete temp;

}

}

bool contains(T symb) {

Node\* temp = head;

while (temp) {

if (temp->get\_data() == symb)

return true;

temp = temp->get\_next();

}

return false;

}

size\_t get\_size()

{

Node\* temp = head;

size\_t length = 0;

while (temp)

{

length++;

temp = temp->get\_next();

}

return length;

}

void print()

{

Node\* temp = head;

while (temp)

{

cout << temp->get\_data();

temp = temp->get\_next();

}

}

void clear()

{

while (head)

{

tail = head->get\_next();

delete head;

head = tail;

}

}

size\_t get\_index(T symb) {

Node\* temp = head;

size\_t index = 0;

while (temp) {

if (temp->get\_data() == symb)

break;

temp = temp->get\_next();

index++;

}

return index;

}

void set(size\_t index, T data)

{

Node\* temp;

temp = get\_pointer(index);

temp->set\_data(data);

}

bool isEmpty()

{

if (!head)

return true;

else

return false;

}

bool compare(List<T> list) {

if (get\_size() != list.get\_size())

return false;

for (int i = 0; i < get\_size(); i++) {

if (at(i) != list.at(i))

return false;

}

return true;

}

bool contains(List<char>& list) {

Node\* temp = head;

while (temp) {

if (temp->get\_data().compare(list))

return true;

temp = temp->get\_next();

}

return false;

}

void reset() {

head = nullptr;

tail = nullptr;

}

void reverse() {

Node\* temp1 = head, \* temp2 = tail;

T value;

for (size\_t i = 0; i < get\_size() / 2; i++) {

value = temp1->get\_data();

temp1->set\_data(temp2->get\_data());

temp2->set\_data(value);

temp1 = temp1->get\_next();

temp2 = temp2->get\_prev();

}

}

int get\_price(size\_t vertex1, size\_t vertex2) {

Node\* temp = head;

while (temp) {

if (temp->get\_data().first\_vertex == vertex1 && temp->get\_data().scnd\_vertex == vertex2)

return temp->get\_data().price;

temp = temp->get\_next();

}

}

};

**Matrix.h**

#pragma once

#include "List.h"

//List<List<T>>

template<class T>

class Matrix {

class Node {

private:

List<T> data;

Node\* next, \* prev;

public:

Node(List<T> data) {

this->data = data;

next = nullptr;

prev = nullptr;

}

~Node() {

next = nullptr;

prev = nullptr;

}

Node() : next(nullptr), prev(nullptr) {};

void set\_data(List<T>& data) {

this->data = data;

}

List<T>& get\_data() {

return data;

}

Node\* get\_next() {

return next;

}

Node\* get\_prev() {

return prev;

}

void set\_next(Node\* temp) {

next = temp;

}

void set\_prev(Node\* temp) {

prev = temp;

}

};

Node\* head, \* tail;

Node\* get\_node(size\_t index)

{

if (isEmpty() || (index > get\_size() - 1))

{

throw out\_of\_range("Invalid argument");

}

else if (index == get\_size() - 1)

return tail;

else if (index == 0)

return head;

else

{

Node\* temp = head;

while ((temp) && (index--))

{

temp = temp->get\_next();

}

return temp;

}

}

public:

Matrix() : head(nullptr), tail(nullptr) {}

Matrix(const Matrix<T>& list) {

head = nullptr;

tail = nullptr;

Node\* temp = list.head;

while (temp) {

push\_back(temp->get\_data());

temp = temp->get\_next();

}

}

~Matrix()

{

while (head)

{

tail = head->get\_next();

delete head;

head = tail;

}

head = nullptr;

}

void set(size\_t index, List<T> data)

{

Node\* temp;

temp = get\_node(index);

temp->set\_data(data);

}

void push\_back(List<T>& data)

{

Node\* temp = new Node;

temp->set\_data(data);

if (head)

{

temp->set\_prev(tail);

tail->set\_next(temp);

tail = temp;

}

else

{

head = temp;

tail = head;

}

}

void push\_front(List<T> data)

{

Node\* temp = new Node;

temp->set\_data(data);

if (head)

{

temp->set\_next(head);

head->set\_prev(temp);

head = temp;

}

else

{

head = temp;

tail = head;

}

}

void push\_back(Matrix<bool> scnd\_list)

{

Node\* temp = scnd\_list.head;

size\_t size = scnd\_list.get\_size();

while ((temp) && (size--))

{

push\_back(temp->get\_data());

temp = temp->get\_next();

}

}

void push\_front(Matrix& scnd\_list)

{

Node\* temp = scnd\_list.tail;

size\_t n = scnd\_list.get\_size();

while ((temp) && (n--))

{

push\_front(temp->get\_data());

temp = temp->get\_prev();

}

}

void insert(size\_t index, List<T> data)

{

Node\* temp;

temp = get\_node(index);

if (temp == head)

push\_front(data);

else

{

Node\* newel = new Node;

newel->set\_data(data);

temp->get\_prev()->set\_next(newel);

newel->set\_prev(temp->get\_prev());

newel->set\_next(temp);

temp->set\_prev(newel);

}

}

void pop\_back()

{

if (head != tail)

{

Node\* temp = tail;

tail = tail->get\_prev();

tail->set\_next(nullptr);

delete temp;

}

else if (!isEmpty())

{

Node\* temp = tail;

tail = head = nullptr;

delete temp;

}

else

throw out\_of\_range("The list is empty");

}

void pop\_front()

{

if (head != tail)

{

Node\* temp = head;

head = head->get\_next();

head->set\_prev(nullptr);

delete temp;

}

else if (!isEmpty())

{

Node\* temp = head;

head = tail = nullptr;

delete temp;

}

else

throw out\_of\_range("The list is empty");

}

void remove(List<T> data) {

Node\* temp = head;

while (temp && temp->get\_data() != data)

temp = temp->get\_next();

if (!temp)

throw out\_of\_range("Invalid argument");

if (temp == head)

pop\_front();

else if (temp == tail)

pop\_back();

else

{

temp->get\_prev()->set\_next(temp->get\_next());

temp->get\_next()->set\_prev(temp->get\_prev());

delete temp;

}

}

List<T>& at(size\_t index) {

Node\* temp;

temp = get\_node(index);

return temp->get\_data();

}

void remove(size\_t index) {

Node\* temp;

temp = get\_node(index);

if (temp == head)

pop\_front();

else if (temp == tail)

pop\_back();

else

{

temp->get\_prev()->set\_next(temp->get\_next());

temp->get\_next()->set\_prev(temp->get\_prev());

delete temp;

}

}

size\_t get\_size() {

Node\* temp = head;

size\_t length = 0;

while (temp)

{

length++;

temp = temp->get\_next();

}

return length;

}

void print() {

Node\* temp = head;

while (temp) {

temp->get\_data().print();

temp = temp->get\_next();

}

std::cout << std::endl;

}

void clear() {

while (head)

{

tail = head->get\_next();

delete head;

head = tail;

}

}

bool isEmpty() {

if (!head)

return true;

else

return false;

}

};

**Input.txt**

S O 3

S P 3

O Q 3

O P 2

P R 2

Q R 4

Q T 2

R T 3