

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

ОТЧЁТ
по лабораторной работе №3
по дисциплине «Построение и анализ алгоритмов»
Тема: Потоки в сети

Студент гр.8382

Фильцин И.В.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Задание

Вар. 5. Поиск не в глубину и не в ширину, а по правилу: каждый раз выполняется переход по дуге, имеющей максимальную остаточную пропускную способность. Если таких дуг несколько, то выбрать ту, которая была обнаружена раньше в текущем поиске пути.

Найти максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

Сеть (ориентированный взвешенный граф) представляется в виде триплета из имён вершин и целого неотрицательного числа - пропускной способности (веса).

Входные данные:

N - количество ориентированных рёбер графа

v_0 - исток

v_n - сток

$v_i v_j w_{ij}$ - ребро графа

...

Выходные данные:

P_{max} - величина максимального потока

$v_i v_j w_{ij}$ - ребро графа с фактической величиной протекающего потока

...

В ответе выходные рёбра отсортируйте в лексикографическом порядке по первой вершине, потом по второй (в ответе должны присутствовать все указанные входные рёбра, даже если поток в них равен 0).

Пример входных данных:

7

a

f

a b 7

a c 6

b d 6

c f 9

d e 3

d f 4

e c 2

Соответствующие выходные данные:

12

a b 6

a c 6

b d 6

c f 8

d e 2

d f 4

e c 2

Ход работы

Описание алгоритма

Алгоритм Форда-Фалкерсона итеративно увеличивает значение потока. Вначале поток равен 0 для каждой дуги. При выполнении каждой итерации алгоритма находится некоторый увеличивающий путь из истока в сток. Данный путь используется для увеличения потока. В найденном пути вычисляется максимальный поток, который можно пропустить через все дуги данного пути не вызвав переполнения. Для каждой дуги в прямом направлении поток увеличивается на вычисленное значение, а для каждой дуги в противоположном

направлении - уменьшается. Алгоритм продолжает свою работу пока может быть найден хотя бы 1 путь из истока в сток. По окончании работы алгоритма, максимальный поток в графе равен сумме потоков, исходящих из стока. При этом сумма потоков, исходящих из стока равна сумме потоков, входящих в сток.

Вычислительная сложность

Время выполнения алгоритма зависит от того, как именно выполняется поиск пути. В реализованном алгоритме (См. исх. код в приложении А) используется жадный поиск: в приоритете дуга, имеющая большую остаточную пропускную способность.

Т.к. общее количество рёбер равно $|E|$, всего выполняется $|E|$ итераций цикла. Время работы жадного поиска зависит от поиска дуги с максимальной остаточной пропускной способности. Т.к. в памяти хранятся дуги с максимальной пропускной способностью > 0 , каждая такая операция поиска выполняется за время равное $O(|E|)$. Т.к. каждая вершина рассматривается ровно по одному разу, итоговое время работы поиска пути составляет $O(|V| + |E||E|) = O(|E|^2)$. На каждом шаге алгоритма поток увеличивается минимум на единицу, следовательно, алгоритм сходится не более чем за $O(f^*)$ шагов (где f^* - максимальное значение потока). Таким образом, общее время работы алгоритма составляет $O(|E|^2 f^*)$.

Сложность по памяти

Для хранения графа в памяти в виде матрицы смежности необходимо $O(|V|^2)$ дополнительной памяти.

Описание функций и структур данных

В исходном коде используются следующие структуры данных:

Graph {*node* : *Map* < *char*, *Node* >} - Граф, который в поле *node* содержит все вершины;

Node {*to* : *Map* < *char*, *Edge* >} - Вершина, которая в поле *to* хранит все дуги из неё;

Edge {*capacity* : *usize*, *flow* : *usize*, *real* : *bool*} - Дуга, которая характеризуется максимальной пропускной способностью *capacity*, текущим потоком *flow* и "реальностью" *real* (*real* выставляется в *true*, если дуга "настоящая", т.е. содержится в исходном графе). Для структур *Graph* и *Node* был перегружен оператор [*char*] для упрощенной индексации.

Для нахождения максимального потока используется функция

fn run(*graph* : &*mut Graph*, *from* : *char*, *to* : *char*) — > *usize*, которая принимает граф *graph*, исток *from* и сток *to*. Функция модифицирует поток в дугах переданного графа и возвращает в качестве результата максимальный поток.

Для поиска пути используется функция

fn find_path(*graph* : &*Graph*, *from* : *char*, *to* : *char*) — > *Option* < (*usize*, *Vec* < *char* >) >, которая принимает граф *graph*, начальную вершину *from* и конечную вершину *to*. Если путь из *from* в *to* был найден, функция возвращает *Some*(*max_flow*, *path*), где *max_flow* - максимальный поток, который можно пустить по найденному пути не вызвав переполнения, а *path* - найденный путь. Если путь найден не был, функция возвращает *None*.

Тестирование

Написанный алгоритм был протестирован при помощи юнит тестов (См. исх. код в приложении Б). Данные тестов представлены в табл. 1

Таблица 1 Тестирование программы

Граф	Исток => Сток	Результат работы
a b 7 a c 6 b d 6 c f 9 d e 3 d f 4 e c 2	a => f	12 a b 6 a c 6 b d 6 c f 8 d e 2 d f 4 e c 2
a b 100 a c 100 b c 1 b d 100 c d 100	a => d	200 a b 100 a c 100 b c 0 b d 100 c d 100
a b 1 a c 1	a => d	0 a b 0 a c 0
a b 3 a c 2 c e 1 b e 4 b d 2 e d 5	a => d	4 a b 3 a c 1 b d 1 b e 2 c e 1 e d 3

Итог работы

В ходе лабораторной работы был реализован алгоритм Форда-Фалкерсона для поиска максимального потока в графе. Была оценена асимптотика данного алгоритма.

ПРИЛОЖЕНИЕ А

Исходный код программы

```
mod tests;

use std::collections::btree_map::{BTreeMap, Entry};
use std::ops::{Index, IndexMut};
use std::io::stdin;
use std::fmt::{Display, Formatter};

struct Graph {
    node: BTreeMap<char, Node>
}

struct Node {
    to: BTreeMap<char, Edge>
}

struct Edge {
    capacity: usize,
    flow: usize,
    real: bool
}

impl Graph {
    fn new() -> Self {
        Graph {
```



```

        node: BTreeMap::new()
    }
}

fn add_edge(&mut self, from: char, to: char, capacity
: usize) {
    self[from][to] = Edge::new(capacity);
}

}

impl Index<char> for Graph {
    type Output = Node;

    fn index(&self, idx: char) -> &Self::Output {
        self.node.get(&idx).unwrap()
    }
}

impl IndexMut<char> for Graph {
    fn index_mut(&mut self, idx: char) -> &mut Self::
Output {
        match self.node.entry(idx) {
            Entry::Occupied(v) => v.into_mut(),
            Entry::Vacant(v) => v.insert(Node::new())
        }
    }
}

```

```

impl Display for Graph {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::
Result {
        for (from, node) in &self.node {
            for (to, edge) in &node.to {
                if edge.real == true {

                    let flow = if self.node.contains_key(&to)
                                && self[*to]
                                .to.contains_key(&from)
                                && self[*to][*from].real
                                == true {
                        let from_to = edge.flow;
                        let to_from = self[*to][*from].flow;

                        if from_to > to_from {
                            from_to - to_from
                        } else {
                            0
                        }
                    } else {
                        edge.flow
                    };

                    write!(f, "{} {} {}\\n", *from, *to, flow)

```

```

        .unwrap();

        }

    }

}

Ok(())
}
}

impl Node {
    fn new() -> Self {
        Node {
            to: BTreeMap::new()
        }
    }
}

impl Index<char> for Node {
    type Output = Edge;

    fn index(&self, idx: char) -> &Self::Output {
        self.to.get(&idx).unwrap()
    }
}

impl IndexMut<char> for Node {

```

```

fn index_mut(&mut self, idx: char) -> &mut Self::
Output {
    match self.to.entry(idx) {
        Entry::Occupied(v) => v.into_mut(),
        Entry::Vacant(v) => v.insert(Edge::default())
    }
}
}

```

```

impl Edge {
    fn new(capacity: usize) -> Self {
        Edge {
            capacity,
            flow: 0,
            real: true
        }
    }

    fn get_allow_flow(&self) -> usize {
        self.capacity - self.flow
    }
}

```

```

impl Default for Edge {
    fn default() -> Self {
        let mut edge = Edge::new(0);
        edge.real = false;
    }
}

```

```

        edge
    }
}

struct Input {
    from: char,
    to: char,
    value: usize,
}

impl Input {
    fn new(string: &String) -> Self {
        let from = string.bytes().nth(0).unwrap() as char;
        let to = string.bytes().nth(2).unwrap() as char;
        let slice = if string.chars().last().unwrap() == '\n' {
            &string[4..string.len() - 1]
        } else {
            &string[4..]
        };
        let value: usize = slice.parse().unwrap();
        Input { from, to, value }
    }
}

fn find_path(graph: &Graph, from: char, to: char) ->
    Option<(usize, Vec<char>)> {

```

```

printlnНаходим!(" путь");
let mut stack = Vec::new();
stack.push(from);

let mut parent = BTreeMap::new();

while !stack.is_empty() {
    // I - Достаем последнюю добавленную вершину
    let current = *stack.last().unwrap();

    printlnТекущая!(" рассматриваемая вершина {}",
current);

    if current == to { break; }

    let max_dest =
        // II - Если есть хотя бы 1 путь из данной
вершины в другую
        if graph.node.contains_key(&current) {
            // Находим дугу с максимальной остаточной
пропускной способностью
            graph[current].to.iter().fold(None, |min, (dest
, edge)| match min {
                None if parent.get(dest).is_none() &&
edge.get_allow_flow() > 0 => Some((dest, edge)),
                Some((_, max_edge)) if parent.get(dest)
.is_none()

```

```

        && edge.get_allow_flow() >
max_edge.get_allow_flow() => Some((dest, edge)),
        expr => expr
    ))
} else { None };

if let Some((dest, edge)) = max_dest {
    // III - Если нашли путь, переходим к следующей
вершине
    printlnНашли!(" путь в {} со значением {}", dest,
edge.get_allow_flow());
    parent.insert(*dest, current);
    stack.push(*dest);
} else {
    // Если не нашли путь, возвращаемся к прошлой
вершине
    printlnПути!(" из данной вершины нет!
Возвращаемся назад");
    stack.pop();
}

}

if let Some(prev) = parent.get(&to) {
    let mut result = vec![to, *prev];
    let mut min_flow = graph[*prev][to].get_allow_flow
();

```

```

    let mut prev = *prev;
    while prev != from {
        let prev_from = *parent.get(&prev).unwrap();
        result.push(prev_from);
        let flow = graph[prev_from][prev].get_allow_flow
();
        if flow < min_flow {
            min_flow = flow;
        }
        prev = prev_from;
    }

    printlnНайденный!(" путь перевёрнутый(): {:?}",
result);

    Some((min_flow, result))
} else {
    None
}
}
}

```

```

fn run(graph: &mut Graph, from: char, to: char) ->
    usize {
    let mut result: usize = 0;

    // I - Находим путь и максимальный поток для него
    while let Some((max_flow, path)) = find_path(&graph,
from, to) {
        printlnОбновляем!(" поток для найденного пути

```



```

максимально( возможный поток = {})", max_flow);

    // II - Обновляем поток для вершин из найденного
пути
    for i in 0..path.len() - 1 {
        let to = path[i];
        let from = path[i + 1];

        if graph[from][to].real == false {
            graph[to][from].flow -= max_flow;
            graph[from][to].capacity -= max_flow;
        } else {
            graph[from][to].flow += max_flow;
            graph[to][from].capacity += max_flow;
        }
    }

    printlnТекущее!(" значение потоков:\n {}", graph);

    result += max_flow;
}

result
}

fn main() {
    let mut buffer = String::new();

```

```

stdin().read_line(&mut buffer).unwrap();
let n: usize = buffer[0..buffer.len() - 1].parse()
    .unwrap();
buffer.clear();

stdin().read_line(&mut buffer).unwrap();
let from = buffer.bytes().nth(0).unwrap() as char;
buffer.clear();

stdin().read_line(&mut buffer).unwrap();
let to = buffer.bytes().nth(0).unwrap() as char;
buffer.clear();

let mut graph = Graph::new();

for _ in 0..n {
    buffer.clear();
    stdin().read_line(&mut buffer).unwrap();

    let input = Input::new(&buffer);
    graph.add_edge(input.from, input.to, input.value);
}

let max = run(&mut graph, from, to);
println!("{}", max);
println!("{}", graph);
}

```

ПРИЛОЖЕНИЕ Б

Исходный код тестов

```
#[cfg(test)]  
  
mod tests {  
    use crate::*;  
  
    macro_rules! graph {  
        ($(($from: expr => $to: expr) ($val: expr)),*) => {  
            {  
                let mut temp_graph = Graph::new();  
                $(  
                    temp_graph.add_edge($from, $to, $val);  
                )*  
                temp_graph  
            }  
        }  
    }  
  
    macro_rules! expect_edge {  
        ($graph: expr, $(($from: expr => $to: expr) ($val:  
        expr)), *) => {  
            {  
                $(  
                    assert_eq!($graph[$from][$to].flow, $val);  
                )*  
            }  
        }  
    }  
}
```

```
}  
}
```

```
#[test]
```

```
fn case_1() {
```

```
    let mut graph = graph!(
```

```
        ('a' => 'b') (7),
```

```
        ('a' => 'c') (6),
```

```
        ('b' => 'd') (6),
```

```
        ('c' => 'f') (9),
```

```
        ('d' => 'e') (3),
```

```
        ('d' => 'f') (4),
```

```
        ('e' => 'c') (2)
```

```
    );
```

```
    assert_eq!(run(&mut graph, 'a', 'f'), 12);
```

```
    expect_edge!(
```

```
        graph,
```

```
        ('a' => 'b') (6),
```

```
        ('a' => 'c') (6),
```

```
        ('b' => 'd') (6),
```

```
        ('c' => 'f') (8),
```

```
        ('d' => 'e') (2),
```

```
        ('d' => 'f') (4),
```

```
        ('e' => 'c') (2)
```

```

    )
}

#[test]
fn case_2() {
    let mut graph = graph!(
        ('a' => 'b') (100),
        ('a' => 'c') (100),
        ('b' => 'c') (1),
        ('b' => 'd') (100),
        ('c' => 'd') (100)
    );

    assert_eq!(run(&mut graph, 'a', 'd'), 200);

    expect_edge!(
        graph,
        ('a' => 'b') (100),
        ('a' => 'c') (100),
        ('b' => 'c') (0),
        ('b' => 'd') (100),
        ('c' => 'd') (100)
    )
}

#[test]
fn case_3() {

```

```

let mut graph = graph!(
    ('a' => 'b') (1),
    ('a' => 'c') (1)
);

assert_eq!(run(&mut graph, 'a', 'd'), 0);

expect_edge!(
    graph,
    ('a' => 'b') (0),
    ('a' => 'c') (0)
)
}
}

```