**ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ**

**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**

**«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»**

**Максимальные потоки в транспортной сети**

**Исполнитель**

Студент группы БПИ173

И.А. Кудря

**Москва 2019**

Оглавление

**ПОСТАНОВКА ЗАДАЧИ3**

**ОПИСАНИЕ АЛГОРИТМОВ И ИСПОЛЬЗОВАННЫХ СТРУКТУР ДАННЫХ4**

**ОПИСАНИЕ ПЛАНА ЭКСПЕРИМЕНТА9**

**РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТА10**

**СРАВНИТЕЛЬНЫЙ АНАЛИЗ АЛГОРИТМОВ11**

**ЗАКЛЮЧЕНИЕ24**

**СПИСОК ЛИТЕРАТУРЫ24**

1. **ПОСТАНОВКА ЗАДАЧИ**

​Реализовать три алгоритма расчета максимального потока в транспортной сети, провести экспериментальное исследование их эффективности и анализ полученных результатов.

1. ​Изучить самостоятельно постановку задачи о максимальном потоке в транспортной сети и методы её решения;
2. ​Разработать с использованием языка C++ программу, реализующую следующие алгоритмы расчета значения максимального потока в заданной транспортной сети:
   1. Базовая реализация метода Форда-Фалкерсона (Ford–Fulkerson algorithm);
   2. Метода Форда-Фалкерсона в версии Эдмондса-Карпа (Edmonds–Karp algorithm);
   3. Алгоритм Ефима Диница (Dinic's/Dinitz's algorithm);
3. ​Провести вычислительный эксперимент с целью оценки реализованных алгоритмов расчёта максимального потока;
4. ​Подготовить отчет по итогам работы, содержащий постановку задачи, описание алгоритмов с оценкой их сложности, задействованных структур данных, описание реализации, описание использованных инструментов (например, если использовались скрипты автоматизации), план эксперимента, полученные результаты измерения эффективности алгоритмов и их анализ, оценку соответствия результатов экспериментальной проверки теоретическим оценкам эффективности исследуемых алгоритмов.
5. **ОПИСАНИЕ АЛГОРИТМОВ И ИСПОЛЬЗОВАННЫХ СТРУКТУР ДАННЫХ**

**Ford–Fulkerson algorithm**

#include **"FordFulkerson.h"**#include **<limits.h>**#include **<cstring>**#include **<queue>**#include **<iostream>**#include **<vector>**#include **"FordFulkerson.h"  
  
using namespace** std;  
  
FordFulkerson::FordFulkerson(vector<vector<**int**>> graph, vector<pair<**int**,**int**>> pairs)  
{  
 G = graph;  
 **this**->pairs = pairs;  
}  
  
**int** FordFulkerson::fordFulkerson(vector<vector<**int**>> &G, **int** s, **int** t)  
{  
 **for** (**int** flow = 0;;)  
 {  
 vector<**char**> visitedArr(G.size(), **false**);  
 **int** df = findPath(G,visitedArr,s,t,**INT\_MAX**);  
 **if** (df == 0)  
 **return** flow;  
 flow += df;  
 }  
}  
*/// DFS  
/// \param G graph  
/// \param visitedArr visited points  
/// \param u  
/// \param t  
/// \param f поток  
/// \return***int** FordFulkerson::findPath(vector<vector<**int**>> &G, vector<**char**> &visitedArr, **int** u, **int** t, **int** f)  
{  
 **if** (u == t)  
 **return** f;  
 visitedArr[u] = **true**;  
 **for** (**int** v = 0; v < visitedArr.size(); v++)  
 {  
 **if** (!visitedArr[v] && G[u][v] > 0)  
 {  
 **int** df = findPath(G, visitedArr, v, t, min(f, G[u][v]));  
 **if** (df > 0) {  
 G[u][v] -= df;  
 G[v][u] += df;  
 **return** df;  
 }  
 }  
 }  
 **return** 0;  
}  
  
**void** FordFulkerson::run()  
{  
 **for**(**int** i=0; i<pairs.size(); ++i)  
  
 {  
 **int** res = fordFulkerson(G, pairs[i].first, pairs[i].second);  
  
 }  
}

**Edmonds–Karp algorithm**

#include **"EdmondCarp.h"**#include **<iostream>**#include **<queue>  
using namespace** std;  
  
  
EdmondCarp::EdmondCarp(vector<vector<**int**>> G, vector<pair<**int**, **int**>> pairs)  
{  
 **this**->G = G;  
 **this**->pairs = pairs;  
}  
*/\*Возвращает true, если есть путь от источника 's' к стоку 't' в  
 остаточный граф. Также заполняет parent [] для хранения пути \*/***bool** bfs(vector<vector<**int**>>& G, **int** s, **int** t, vector<**int**>& parent)  
{  
 **int** size = parent.size();  
 *// Создаем посещенный массив и отмечаем все вершины как не посещенные* vector<**char**> visited(size,**false**);  
  
 queue <**int**> q;  
 q.push(s);  
 visited[s] = **true**;  
 parent[s] = -1;  
  
 *// Standard BFS цикл* **while** (!q.empty())  
 {  
 **if** (visited[t])  
 **break**;  
 **int** u = q.front();  
 q.pop();  
  
 **for** (**int** v=0; v<size; v++)  
 {  
 **if** (visited[v] || G[u][v]<=0)  
 {  
 **continue**;  
 }  
 q.push(v);  
 parent[v] = u;  
 visited[v] = **true**;  
 }  
 }  
  
  
 *//Если мы достигли стока в BFS, начиная с исходного кода, вернемся  
 //* **return** (visited[t]);  
}  
  
*//Возвращает максимальный поток от s до t в данном графе***int** EdmondCarp::edmondCarp(vector<vector<**int**>> &graph, **int** s, **int** t)  
{  
 **int** u, v;  
  
  
 vector<vector<**int**>> rGraph; /  
 copy(graph.begin(),graph.end(),back\_inserter(rGraph));  
  
 vector<**int**> parent(graph.size());  
  
 **int** maxflow = 0;  
  
  
 **while** (bfs(rGraph, s, t, parent))  
 {  
 *//Находим минимальную остаточную емкость ребер  
 // путь заполнен BFS. Или найти максимальный поток  
 // через найденный путь.* **int** pathflow = 1000000000;  
 **for** (v=t; v!=s; v=parent[v])  
 {  
 u = parent[v];  
 pathflow = min(pathflow, rGraph[u][v]);  
 }  
  
 *//обновить остаточные емкости ребер и обратных ребер  
 // по пути* **for** (v=t; v != s; v=parent[v])  
 {  
 u = parent[v];  
 rGraph[u][v] -= pathflow;  
 rGraph[v][u] += pathflow;  
 }  
  
 maxflow += pathflow;  
 }  
  
 **return** maxflow;  
}  
**void** EdmondCarp::run()  
{  
 **for** (**int** i=0; i<pairs.size(); ++i)  
 {  
 **int** res = edmondCarp(G, pairs[i].first, pairs[i].second);  
 *//cout <<"Result of EdmondCarp Algorithm " << res << endl;* }  
}

**Dinic's algorithm**

#include **"Dinic.h"**#include **<list>**#include **<iostream>  
using namespace** std;  
Dinic::Dinic(vector<vector<**int**>> matrix, **int** V, vector<std::pair<**int**, **int**>> pairs)  
{  
 adject = **new** vector<Edge>[V];  
 **this**->V = V;  
 level = **new int**[V];  
 readFromMatrix(matrix);  
 **this**->pairs = pairs;  
}  
  
  
**bool** Dinic::bfs(**int** s, **int** t)  
{  
 **for** (**int** i = 0 ; i < V ; i++)  
 level[i] = -1;  
  
 level[s] = 0; *// Глубина графа  
  
 //Создается очередь, поставлена в очередь исходная вершина  
 // и отмечаем исходную вершину как посещенную  
 // массив level [] также работает как посещаемый массив.* list<**int**> q;  
 q.push\_back(s);  
  
 vector<Edge>::iterator i ;  
 **while** (!q.empty())  
 {  
 **int** u = q.front();  
 q.pop\_front();  
 **for** (i = adject[u].begin(); i != adject[u].end(); i++)  
 {  
 Edge &e = \*i;  
 **if** (level[e.v] < 0 && e.flow < e.C)  
 {  
 *// Level of current vertex is,  
 // level of parent + 1* level[e.v] = level[u] + 1;  
  
 q.push\_back(e.v);  
 }  
 }  
 }  
  
 *//  
 //Если мы не можем добраться до раковины,  
 // возвращаем false, иначе true* **return** level[t] >= 0;  
}  
  
**int** Dinic::dfs(**int** u, **int** flow, **int** t, **int** start[])  
{  
 **if** (u == t)  
 **return** flow;  
  
 *//Пересекаем все соседние ребра.* **for** ( ; start[u] < adject[u].size(); start[u]++)  
 {  
  
 Edge &e = adject[u][start[u]];  
  
 **if** (level[e.v] == level[u]+1 && e.flow < e.C)  
 {  
 *// находим минимальный поток из u в v* **int** curr\_flow = min(flow, e.C - e.flow);  
  
 **int** temp\_flow = dfs(e.v, curr\_flow, t, start);  
  
 *// поток больше чем 0* **if** (temp\_flow > 0)  
 {  
 *// добавить поток к ребру* e.flow += temp\_flow;  
  
  
 adject[e.v][e.rev].flow -= temp\_flow;  
 **return** temp\_flow;  
 }  
 }  
 }  
  
 **return** 0;  
}  
  
**int** Dinic::dinicMaxflow(**int** s, **int** t)  
{  
 **if** (s == t)  
 **return** -1;  
  
 **int** total = 0; *// Initialize result* **while** (bfs(s, t) == **true**)  
 {  
 *// подсчитать количество посещенных ребер  
 // from V { 0 to V }* **int** \*start = **new int**[V+1];  
  
  
 **while** (**int** flow = dfs(s, **INT\_MAX**, t, start))  
  
 *//Добавить поток пути к общему потоку* total += flow;  
 }  
  
 *// return maximum flow* **return** total;  
}  
**void** Dinic::addEdge(**int** u, **int** v, **int** C)  
{  
   
 Edge a{v, 0, C, (**int**)adject[v].size()};  
  
   
 Edge b{u, 0, 0, (**int**)adject[u].size()};  
  
 adject[u].push\_back(a);  
 adject[v].push\_back(b); *// reverse edge*}  
**void** Dinic::readFromMatrix(vector<vector<**int**>> matrix)  
{  
 **for** (**int** i=0; i<matrix[0].size(); ++i) {  
 **for** (**int** j=0; j<matrix.size(); ++j) {  
 **if** (matrix[i][j] !=0) {  
 addEdge(i, j, matrix[i][j]);  
 }  
 }  
 }  
}  
  
**void** Dinic::run()  
{  
 **for**(**int** i=0; i<pairs.size(); ++i)  
 {  
 **int** res = dinicMaxflow(pairs[i].first, pairs[i].second);}  
}

1. **ОПИСАНИЕ ПЛАНА ЭКСПЕРИМЕНТА**

Для проверки работы алгоритма было выдано 40 файлов с входными данными разных размеров. Это нужно было, чтобы наглядно показать разницу в эффективности работы алгоритмов на разных данных. Чтобы отследить это, можно было пойти двумя способами:

1. считать количество операций (такую методику мы использовали летом на практике)
2. считать количество пройденных наносекунд/миллисекунд.

Конечно же, у каждого способа есть свои плюсы. Например, 1 позволяет показать более точно вне зависимости от мощности компьютера, на котором и запускалась программа. 2 же показывает более понятные для человека цифры и даёт понять сколько потребовалось времени на разные алгоритмы.

Для того, чтобы замерить результаты работы алгоритма, потребовалось воспользоваться библиотекой time.h. Использовалось стандартная функция clock(), которая должна была применяться один раз до запуска алгоритма и второй раз после, а потом уже шёл подсчет времени.

Чтобы прогнать все тесты, мне пришлось воспользоваться более мощным компьютером (ноутбук значительно медленнее и ненадежнее). В итоге все тесты были прогнаны за 3 часа, а результаты времени работы алгоритма были записаны в файл.

Также хочется сказать, что файлы вплоть до 1510 были прогнаны 10 раз и было подсчитано среднее время для более честной оценки. Файлы после 1510 были прогнаны по одному разу в силу долгого времени выполнения алгоритмов (особенно Форда-Фалкерсона).

1. **РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ**

|  |  |  |  |
| --- | --- | --- | --- |
| Файл | Алг. Ford–Fulkerson | Алг. Edmonds–Karp | Алг. ​Dinic |
| input\_10\_0.0.txt | 0.01150 | 0.01260 | 0.01810 |
| input\_10\_0.5.txt | 0.01220 | 0.01830 | 0.01060 |
| input\_10\_1.0.txt | 0.06660 | 0.011060 | 0.06770 |
| input\_10\_disco.txt | 0.02710 | 0.06190 | 0.03870 |
| input\_310\_0.0.txt | 28 | 9 | 1 |
| input\_310\_0.5.txt | 2753 | 282 | 35 |
| input\_310\_1.0.txt | 1874 | 700 | 88 |
| input\_310\_disco.txt | 837 | 100 | 5 |
| input\_610\_0.0.txt | 88 | 58 | 2 |
| input\_610\_0.5.txt | 9289 | 2162 | 150 |
| input\_610\_1.0.txt | 43789 | 4686 | 67 |
| input\_610\_disco.txt | 6511 | 798 | 12 |
| input\_910\_0.0.txt | 559 | 245 | 2 |
| input\_910\_0.5.txt | 66457 | 7235 | 451 |
| input\_910\_1.0.txt | 91762 | 15756 | 145 |
| input\_910\_disco.txt | 28682 | 2197 | 29 |
| input\_1210\_0.0.txt | 949 | 404 | 6 |
| input\_1210\_0.5.txt | 169617 | 16448 | 488 |
| input\_1210\_1.0.txt | 129899 | 34194 | 254 |
| input\_1210\_disco.txt | 86407 | 6105 | 54 |
| input\_1510\_0.0.txt | 10001 | 333 | 13 |
| input\_1510\_0.5.txt | 189414 | 30915 | 230 |
| input\_1510\_1.0.txt | 641301 | 73165 | 464 |
| input\_1510\_disco.txt | 105015 | 11659 | 84 |
| input\_1810\_0.0.txt | 2768 | 445 | 4 |
| input\_1810\_0.5.txt | 663003 | 60728 | 1889 |
| input\_1810\_1.0.txt | 1364872 | 135018 | 778 |
| input\_1810\_disco.txt | 346089 | 21543 | 124 |
| input\_2110\_0.0.txt | 1077 | 504 | 8 |
| input\_2110\_0.5.txt | 1055331 | 112831 | 3311 |
| input\_2110\_1.0.txt | 1849971 | 204731 | 1004 |
| input\_2110\_disco.txt | 498762 | 32949 | 178 |
| input\_2410\_0.0.txt | 1242 | 364 | 4 |
| input\_2410\_0.5.txt | 571720 | 129522 | 2181 |
| input\_2410\_1.0.txt | 2746688 | 305566 | 1330 |
| input\_2410\_disco.txt | 756108 | 55420 | 242 |
| input\_2710\_0.0.txt | 5583 | 1833 | 10 |
| input\_2710\_0.5.txt | 2354679 | 190638 | 495 |
| input\_2710\_1.0.txt |  | .30 | .30 |
| input\_2710\_disco.txt | .00 | . | .30 |

1. **СРАВНИТЕЛЬНЫЙ АНАЛИЗ АЛГОРИТМОВ**

Ниже представлена диаграммы результатов работы алгоритмов (40 диаграмм для каждого файла + 3 диаграммы для каждого алгоритма всех файлов)

1. **ЗАКЛЮЧЕНИЕ**

Как мы видим из графиков, с увеличением количество вершин увеличивается и время выполнения поиска максимального потока независимо от алгоритма. Также нельзя не заметить, что Алгоритм Ефима Диница очень эффективный и работает в сотни, а то и в тысячи раз быстрее, чем его аналоги.

1. **СПИСОК ЛИТЕРАТУРЫ**
2. [Электронный ресурс]. URL: <https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem>/ (дата обращения: 06.04.2019)
3. [Электронный ресурс]. URL: <https://ru.wikipedia.org/wiki/Алгоритм_Форда_—_Фалкерсона/> (дата обращения: 06.04.2019)
4. [Электронный ресурс]. URL: <https://brilliant.org/wiki/ford-fulkerson-algorithm/> (дата обращения: 06.04.2019)
5. [Электронный ресурс]. URL: <https://www.tutorialspoint.com/Ford-Fulkerson-Algorithm> (дата обращения: 07.04.2019)
6. [Электронный ресурс]. URL: [https://ru.wikipedia.org/wiki/Алгоритм\_Эдмондса\_—\_Карпа](https://ru.wikipedia.org/wiki/Алгоритм_Эдмондса_—_Карпа%20%20)  (дата обращения: 08.04.2019)
7. [Электронный ресурс]. URL: [http://e-maxx.ru/algo/edmonds\_karp/](http://e-maxx.ru/algo/edmonds_karp/%20) (дата обращения: 08.04.2019)
8. [Электронный ресурс]. URL: [https://icmmg.nsc.ru/sites/default/files/pubs/om2014-3.pdf](https://icmmg.nsc.ru/sites/default/files/pubs/om2014-3.pdf%20) (дата обращения: 09.04.2019)
9. [Электронный ресурс]. URL[: https://ru.wikipedia.org/wiki/Алгоритм\_Диница](file:///C:\Users\theli\AppData\Roaming\Microsoft\Word\:%20https:\ru.wikipedia.org\wiki\Алгоритм_Диница) (дата обращения: 010.04.2019)
10. [Электронный ресурс]. URL: <http://e-maxx.ru/algo/diniс> (дата обращения: 10.04.2019)
11. [Электронный ресурс]. URL: <https://www.geeksforgeeks.org/dinics-algorithm-maximum-flow/> (дата обращения: 11.04.2019)