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

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Построение и анализ алгоритмов»

ТЕМА: ПОТОКИ В СЕТИ

Студент гр. 9382 _____ Кодуков А.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы:
Изучить и использовать на практике алгоритм Форда-Фалкерсона.

Задание:
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 ω_{ij} - ребро графа
v_i v_j ω_{ij} - ребро графа
...

Выходные данные:
P_{max} - величина максимального потока
v_i v_j ω_{ij} - ребро графа с фактической величиной протекающего потока
v_i v_j ω_{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).

Sample Input:

```
7
a
```

```
f
a b 7
a c 6
b d 6
c f 9
d e 3
d f 4
e c 2
```

Sample Output:

```
12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2
```

Вар. 2. Поиск в ширину. Обработка совокупности вершин текущего фронта как единого целого, дуги выбираются в порядке уменьшения остаточных пропускных способностей.

Описание алгоритма:

Алгоритм нахождения пути:

Для каждой следующей вершины фронта рассматриваются все не посещенные соседи с остаточной пропускной способностью большей 0. Соседи сортируются по уменьшению остаточной пропускной способности и добавляются во фронт. Затем из фронта достается следующая по очереди добавления вершина. Процесс повторяется, пока не будет достигнута конечная вершина.

Алгоритм поиска максимального потока:

Находится первый доступный путь с ненулевой остаточной пропускной способностью, затем по этому пути пропускается поток соответствующий

минимальной из пропускных способностей ребер этого пути, то есть от всех ребер отнимается значение этого потока, а к обратным ребрам – прибавляется. Полученное значение прибавляется к счетчику максимального потока. Алгоритм продолжает работу пока возможно найти доступный путь.

Сложность алгоритма

В худшем случае алгоритм увеличивает поток на каждой итерации на 1, тогда всего итераций будет F – величина максимального потока. На каждой итерации совершается поиск в ширину, его сложность – *O(V + E)*. Тогда итоговая сложность по времени составляет *O(F * (V + E))*

Сам алгоритм Форда-Фалкерсона использует исходный граф, поэтому не требует дополнительной памяти. Поиск в ширину хранит путь, предка каждой вершины в обходе, и посещенные вершины, поэтому итоговая сложность по памяти $O(3 * V)$

Функции и структуры данных:

Структуры данных:

`typedef std::pair<type, double> path_to` – вершина и пропускная способность ее дуги

`typedef std::map<type, double> edges_end` – ребра одной вершины

`typedef std::map<type, edges_end> edges_type` – все ребра (структура хранения графа – список смежности)

`typedef std::vector<path_to> bfs_neighbours` – массив для соседей одной вершины перед сортировкой

`std::vector<path_to>, comparator> pr_queue` - очередь из вершин, отсортированная по убыванию дуг

`typedef std::vector<type> path_stack` – стек для текущего пути

`std::set<type> visited` – посещенные вершины

Реализованные функции:

Инициализация графа

Сигнатура: `stream_finder(std::istream &input, std::ostream &output, bool file)`

Аргументы:

- input – поток ввода
- output – поток вывода
- file – является ли ввод файловым

Алгоритм:

- Считать начало и конец искомого пути
- Считать ребра

Вывод ребер, исходящих из одной вершины

Сигнатура: `void print_edges_vert(const edges_end &q, const type &vert, std::ostream &output)`

Аргументы:

- q – очередь смежных вершин
- vert – первая вершина ребра
- output – поток вывода

Вывод всех ребер графа

Сигнатура: `void print_all_edges(const edges_end &e, std::ostream &output)`

Аргументы:

- e – ребра
- output – поток вывода

Вывод фронта

Сигнатура: `void print_frontier(const edges_end &f, std::ostream &output)`

Аргументы:

- f – фронт
- output – поток вывода

Вывод всего графа

Сигнатура: `void print_graph(, std::ostream &output)`

Аргументы:

- output – поток вывода

Поиск пути

Сигнатура: `void update_path(std::ostream &output)`

Аргументы:

- output – поток вывода

Алгоритм:

- Инициализировать текущую вершину стартовой вершиной
- Пока путь не найден (алгоритм перешел на финишную вершину)
    - Рассмотреть все ребра, исходящие из текущей вершины
    - Добавить все не посещённые вершины с остаточной пропускной способностью ребра больше нуля во фронт
    - Отсортировать фронт по убыванию остаточной пропускной способностью соответствующих ребер
    - Достать следующую вершину из фронта

Алгоритм Форда-Фолкерсона

Сигнатура: `double find_max_flow(std::ostream &output)`

Аргументы:

- output – поток вывода

Алгоритм:

- Пока есть доступный путь
    - Найти минимальную вместимость ребра на этом пути
    - Отнять от остаточных пропускных способностей ребер минимальную, прибавить для обратных ребер
    - Обновить текущий поток ребер на пути
    - Обновить текущий максимальный поток

Тестирование:

№	Входные данные	Вывод
1	7 a f a b 7 a c 6 b d 6 c f 9 d e 3 d f 4 e c 2	12 a b 6 a c 6 b d 6 c f 8 d e 2 d f 4 e c 2
2	5 a d a b 1000 a c 1000 b c 1 b d 1000 c d 1000	2000 a b 1000 a c 1000 b c 0 b d 1000 c d 1000
3	12 a d a b 20 a c 30 a d 10 b a 20 b c 40 b d 30 c a 30 c b 40 c d 20 d b 30 d c 20 d d 10	60 a b 20 a c 30 a d 10 b a 0 b c 0 b d 30 c a 0 c b 10 c d 20 d b 0 d c 0 d d 0
4	11 a h a b 3 b e 1 a c 1 c e 2 a d 2 d e 4 e g 3 e f 2 f h 3 g h 1 d f 1	4 a b 1 a c 1 a d 2 b e 1 c e 1 d e 1 d f 1 e f 2 e g 1 f h 3 g h 1
5	4 a	0 a b 0

	d a c 1 a b 1 c b 1 b c 1	a c 0 b c 0 c b 0

Вывод:

В результате выполнения работы был изучен и реализован алгоритмы Форда-Фолкерсона.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <algorithm>
#include <cmath>
#include <fstream>
#include <iostream>
#include <map>
#include <queue>
#include <set>
#include <string>
#include <vector>

bool file;

// vertex type
typedef char type;

// vertex + some number
typedef std::pair<type, double> path_to;
// comparator for priority queue
int compare(const void *lhs, const void *rhs) {
  return ((path_to *)lhs)->second - ((path_to *)rhs)->second;  ///
}

// edges of one vertex
typedef std::map<type, double> edges_end;
// array of neighbours for bfs to sort
typedef std::vector<path_to> bfs_neighbours;
// vertex and it's edges
typedef std::map<type, edges_end> edges_type;
// stack for current path
typedef std::vector<type> path_stack;

class stream_finder {
 private:
  edges_type edges;        // all edges with residual capacities
  edges_type capacity;     // all edges with their current flows
  type start, end;         // source and drain
  std::set<type> visited;  // visited vertices
  path_stack path;

 public:
  stream_finder() {}
  stream_finder(std::istream &input, std::ostream &output, bool file) {
    std::string str;
    int num;

    std::cin.clear();  // на случай, если предыдущий ввод завершился с ошибкой
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

    if (!file) std::cout << "Input num of edges:\n";
    input >> num;
    // read source and drain
    if (!file) std::cout << "Input source and drain:\n";
    input >> start >> end;
    type first, second;
    double weight;
    // read edges
    if (!file) std::cout << "Input edges (<first> <second> <capacity>):\n";
    for (int i = 0; i < num; i++) {
      input >> first >> second >> weight;
      edges[first][second] = weight;
    }
```

```
  }

  void print_edges_vert(const edges_end &e, const type &vert, std::ostream &output)
{
    for (auto &el : e)
      output << vert << "-" << el.first << ": " << el.second << "\n";
  }

  void print_frontier(const std::queue<path_to> &f, std::ostream &output) {
    std::queue<path_to> tmp = f;
    std::vector<path_to> arr;
    while (!tmp.empty()) {
      arr.push_back(tmp.front());//top());
      tmp.pop();
    }
    for (auto &el : arr) output << el.first << ": " << el.second << " ";
    output << "\n";
  }

  void print_all_edges(const edges_type &e, std::ostream &output) {
    for (auto &el : e) {
      print_edges_vert(el.second, el.first, output);
    }
  }

  void print_graph(std::ostream &output) {
    for (int i = 0; i < 15; i++) output << "*";
    output << "\n";
    output << "Graph:\n";
    output << "\nStart: " << start << "\nEnd :" << end;
    output << "\nEdges:\n";
    print_all_edges(edges, output);
    for (int i = 0; i < 15; i++) output << "*";
    output << "\n";
  }

  // print flows of edges
  void print_capacity(std::ostream &output) {
    for (auto &a : capacity)
      for (auto &b : a.second)
        output << a.first << " " << b.first << " " << b.second << "\n";
  }

  // Find path
  void update_path(std::ostream &output) {
    type cur = start;          // current vertex
    bool path_found = false;   // searching end flag

    std::queue<path_to> frontier;  // unvisited vertices with edge capacity
    std::map<type, type> came_from;  // key - vertex, value - previous vertex on
path

    path.clear();
    visited.clear();

    output << "\n\nBegin path finding...\n\n";

    // search cycle
    while (!path_found) {
      bfs_neighbours cur_pathes; // edges of current vertex
      output << "\nCurrent vertex: " << cur << "\n";
      // get edges
      if (edges.find(cur) != edges.end()) {
        auto found = edges.find(cur)->second;
        for (auto &el : found)
          cur_pathes.push_back(std::make_pair(el.first, el.second));
        print_edges_vert(found, cur, output);
```

```
      } else {
        cur_pathes = bfs_neighbours();
        output << "No edges\n";
      }
      output << "Current edges: \n";
      auto iter_visited = visited.end();
      int num = (int)cur_pathes.size();
      // sort neighbours by capacity
      std::qsort(cur_pathes.data(), num, sizeof(path_to), compare);
      // add all unvisited neighbours to frontier
      for (auto &vert : cur_pathes){
        output << "Checking path " << cur << "-" << vert.first << "\n";
        // check if capacity of edge > 0 and it wasnt visited
        if (vert.second > 0 && (visited.find(vert.first) == visited.end())){
          output << "  It wasn't visited earlier and capacity > 0, add to
frontier\n";
          // add to frontier
          frontier.push(vert);
          came_from[vert.first] = cur;
          // check if path found
          if (vert.first == end) {
            output << "Current vertex is finish, path was found!\n";
            path_found = true;
            break;
          }
          visited.emplace(vert.first);
        } else {
          output << "  It was visited earlier or capacity == 0\n";
        }
      }
      if (!path_found) {
        if (frontier.empty()) {
          output << "No more pathes\n";
          break;
        }
        // get next vertex from frontier
        output << "Frontier:\n";
        print_frontier(frontier, output);
        cur = frontier.front().first;
        frontier.pop();
      }
    }
    // Get path
    if (path_found) {
      type tracker = end;
      while (tracker != start) {
        path.push_back(tracker);
        tracker = came_from[tracker];
      }
      path.push_back(start);
      std::reverse(path.begin(), path.end());
      output << "Path: ";
      for (auto &v: path)
        output << v;
      output << "\n";
    }
  }

  // count max flow of net
  double find_max_flow(std::ostream &output) {
    double flow = 0;

    // set all capacities to 0
    for (auto &v1: edges)
      for (auto &v2: v1.second)
        capacity[v1.first][v2.first] = 0;
    // continue if path exists
```

```
    while (update_path(output), !path.empty()) {
      type a = start, b;
      // count min capacity of path
      output << "Capacity of path:\n  ";
      double min_capacity = std::numeric_limits<double>::max();
      for (int i = 1; i < path.size(); i++) {
        b = path[i];
        double cur_capacity = edges[a][b];
        if (cur_capacity < min_capacity)
          min_capacity = cur_capacity;
        output << a << "-(" << cur_capacity << ")-" << b << " ";
        a = b;
      }
      output << "\nMin capacity: " << min_capacity << "\n";
      // update graph
      a = start;
      for (int i = 1; i < path.size(); i++) {
        b = path[i];
        edges[a][b] -= min_capacity;
        edges[b][a] += min_capacity;
        a = b;
      }
      output << "Residual capacities:\n";
      print_all_edges(edges, output);
      // update current max flow
      flow += min_capacity;
      // update capacities
      a = start;
      for (int i = 1; i < path.size(); i++) {
        b = path[i];
        if (capacity.find(a) != capacity.end() &&
            capacity[a].find(b) != capacity[a].end())
          capacity[a][b] += min_capacity;
        else
          capacity[b][a] -= min_capacity;
        a = b;
      }
    }
    return flow;
  }
};

int main() {
  stream_finder sf;

  std::cout << "File input (from input.txt)? 1 - yes, other - no:";
  char ch;
  std::cin >> ch;

  std::ofstream output("output.txt");
  if (ch == '1') {
    std::ifstream input("input.txt");
    sf = stream_finder(input, output, true);
  } else {
    sf = stream_finder(std::cin, output, false);
  }

  sf.print_graph(output);
  double res = sf.find_max_flow(output);

  output << "Result:\n";
  output << "\n" << res << "\n";
  sf.print_capacity(output);
  std::cout << "Full log in output.txt\n";

  std::cout << "Result:\n";
  std::cout << "\n" << res << "\n";
```

```
  sf.print_capacity(std::cout);
  return 0;
}
```