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

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Построение и анализ алгоритмов»

Тема: Потоки в сети

Студент гр. 8304 Нам Ё Себ

Преподаватель Размочаева Н.В.

Санкт-Петербург

2020

Вариант 6.

Цель работы.

Построение и анализ алгоритма Форда-Фалкерсона на основе на решения задачи о нахождении максимального потока в сети.

Основные теоретические положения.

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).

Индивидуальное задание

Поиск не в глубине и не в ширину, а по правилу: каждый раз выполняется переход по дуге, соединяющей вершины, имена которых в алфавите ближе всего друг к другу. Если таких дуг несколько, то выбрать ту, имя конца которой в алфавите ближайшее к началу алфавита.

Описание алгоритма.

На вход алгоритму подается граф для поиска максимального потока, исток и сток графа. После чего производится поиск соседних вершин по правилу индивидуализации.

Поиск соседних вершин начинается из истока. Переход в соседнюю вершину осуществляется если она ранее не была посещена и остаточная пропускная способность по ребру до неё больше нуля. При этом (согласно индивидуализации) выбирается соседняя вершина, разность имён её и текущей вершины наименьшая. При равенстве этих разностей выбирается вершина, имя которой находится ближе к началу алфавита.

При нахождении пути до стока, из ребер пути находится ребро с минимальной пропускной способностью и её величина вычитается из пропускных способностей всех рёбер пути от истока к стоку, а в обратном направлении найденная минимальная пропускная способность прибавляется (если ребра на обратном пути нет, то оно достраивается). Величина максимального потока увеличивается также на минимальную пропускную способность пути.

В случае если на какой-то итерации алгоритма ни одной соседней вершины не будет обнаружено, либо имеющиеся не будут соответствовать требованиям (описанным во втором абзаце описания алгоритма), и при этом алгоритм вернётся к истоку, то это будет означать, что больше сквозных путей в данном графе построить нельзя, поэтому в таком случае осуществляется переход к формированию ответа. Если же соседних вершин нет, но текущая вершина не является истоком, то производится откат к предыдущей вершине, оставляя текущую вершину посещённой чтобы больше к ней не возвращаться на данной итерации. На последнем этапе алгоритма формируется ответ. Фактический поток через каждое ребро определяется как разность первоначальной и конечной пропускных способностей. Если она отрицательная, и ребро было задано изначально, то фактическое значение потока через него равняется нулю.

Сложность алгоритма по операциям: O (E * F), E – число ребер в графе, F – максимальный поток. Сложность алгоритма по памяти: O (N + E), N – количество вершин, E – количество ребер.

Описание функций и структур данных

```
1. class Neighbor {
public:
        char vertex;
        int start_flow;
        int flow;
        int final_flow;

        Neighbor(char vertex, int start_flow) : vertex(vertex),
```

```
start_flow(start_flow), flow(start_flow) {}
        Neighbor(char vertex) : vertex(vertex) {}
```

Класс для хранения соседа (данных о ребре до соседней вершины).

Имеются конструкторы, первый инициализирует все поля структуры, а второй только поле vertex

2.
```
class Vertex {
public:
        char vertex;
        char prev_vertex;
        std::vector<Neighbor> neighbors;

        Vertex() {}
        Vertex(char vertex) : vertex(vertex) {}
};
```

Класс для хранения вершины графа. Имеются конструкторы для создания вершин без инициализации полей и с инициализацией поля vertex

3. `int maxFlowCount(std::vector<Vertex> vector_of_vertex, Vertex stock);`

Функция принимает вектор вершин графа и сток. Находит максимальный поток сквозного пути и возвращает его.

4. `void recountFlow(std::vector<Vertex>& vector_of_vertex, Vertex stock, int max_flow);`

Функция принимает ссылку на вектор вершин графа, сток и максимальный поток сквозного пути. Пересчитывает текущую пропускную способность всех рёбер сквозного пути (от истока к стоку и наоборот), создавая недостающие рёбра.

5. `bool isExist(std::vector<Vertex> vector, char vertex);`

Функция принимает вектор вершин и вершину и возвращает true или false в зависимости от того, есть ли вершина vertex в векторе vector или нет соответственно.

6. `int findVertex(std::vector<Vertex> vector, char vertex);`

Функция принимает вектор вершин и вершину и возвращает индекс

найденной вершины в векторе вершин. Если вершину не удалось найти функция возвращает -1

7. `int findNeighborIndex(Vertex vertex, char neighbor);`

   Функция принимает вершину и соседа(ребро) и возвращает индекс найденного соседа в вершине vertex. Если соседа не удалось найти функция возвращает -1

8. `bool cmpVertex(const Vertex& a, const Vertex& b);`

   Функция принимает две константные ссылки на вершины и сравнивает их символы. Если значение символа(ASCII код) первой вершины меньше второй возвращается true, иначе false.

9. `bool cmpNeighbors(const Neighbor& a, const Neighbor& b);`

   Функция принимает две константные ссылки на соседи(рёбра) и сравнивает их символы. Если значение символа(ASCII код) первого соседа меньше второго возвращается true, иначе false.

Вывод промежуточной информации.

Во время основной части работы алгоритма происходит вывод промежуточной информации, а именно, выбранная на данном шаге вершина (поиск в ширину), величину минимальной пропускной способности для данного пути, поток в ребрах, входящих в состав ранее найденного пути.

Тестирование.

Таблица 1 – Результаты тестирование

Ввод	**Вывод**
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
5 a d a b 20 b c 20 c d 20 a c 1 b d 1	21 a b 20 a c 1 b c 19 b d 1 c d 20
9 a d a b 8 b c 10 c d 10 h c 10 e f 8 g h 11	18 a b 8 a g 10 b c 0 b e 8 c d 10 e f 8 f d 8

b e 8 a g 10 f d 8	g h 10 h c 10
16 a e a b 20 b a 20 a d 10 d a 10 a c 30 c a 30 b c 40 c b 40 c d 10 d c 10 c e 20 e c 20 b e 30 e b 30 d e 10 e d 10	60 a b 20 a c 30 a d 10 b a 0 b c 0 b e 30 c a 0 c b 10 c d 0 c e 20 d a 0 d c 0 d e 10 e b 0 e c 0 e d 0

Вывод.

В ходе работы был построен и анализирован алгоритм Форда-Фалкерсона на основе решения задачи о нахождении максимального потока в сети. Исходный код программы представлен в приложении А.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД

```
#include <iostream>
#include <algorithm>
#include <vector>


class neighbor {
public:
	char vertex;
	int start_flow;
	int flow;
	int final_flow;

	neighbor(char vertex, int start_flow) : vertex(vertex),
start_flow(start_flow), flow(start_flow) {}
	neighbor(char vertex) : vertex(vertex) {}
};

class Vertex {
public:
	char vertex;
	char prev_vertex;
	std::vector<neighbor> neighbors;

	Vertex() {}
	Vertex(char vertex) : vertex(vertex) {}
};

bool isExist(std::vector<Vertex> vector, char vertex) {
	for (int i = 0; i < vector.size(); ++i)
		if (vector[i].vertex == vertex)
			return true;

	return false;
}

int findVertex(std::vector<Vertex> vector, char vertex) {
	for (int i = 0; i < vector.size(); i++)
		if (vertex == vector[i].vertex)
			return i;

	return -1;
}

int findneighborIndex(Vertex vertex, char neighbor) {
	if (vertex.neighbors.empty())
		return -1;

	for (int i = 0; i < vertex.neighbors.size(); ++i)
		if (vertex.neighbors[i].vertex == neighbor)
			return i;

	return -1;
}

int maxFlowCount(std::vector<Vertex> vector_of_vertex, Vertex stock) {		//
Данная функция считает максимальный поток в сквозном пути
	int flow, min_flow, index_prev, index_neighbor;
	std::vector<int> flows;
	Vertex current = stock;		// Текущая вершина - сток
```

```
	while (current.prev_vertex != '0')
                                        // Пока не дошли до истока,
	{
                                                                // находим вершину
предыдущую текущей
		index_prev = findVertex(vector_of_vertex, current.prev_vertex);
                                // в сквозном пути, а в ней величину потока
		index_neighbor = findneighborIndex(vector_of_vertex[index_prev],
current.vertex);        // до текущей вершины. Добавляем найденное
		flow = vector_of_vertex[index_prev].neighbors[index_neighbor].flow;
                        // значение в вектор с потоками. Текущей
		flows.push_back(flow);
                                                // вершиной становится предыдущая в
		current = vector_of_vertex[index_prev];
                                        // сквозном пути.
	}


	min_flow = flows[0];

	for (int i = 1; i < flows.size(); ++i)          // Находим минимальный
		if (flows[i] < min_flow)                        // из потоков сквозного
пути
			min_flow = flows[i];

	return min_flow;        // Возвращаем максимальный поток сквозного пути
}

void recountFlow(std::vector<Vertex>& vector_of_vertex, Vertex stock, int max_flow)
{           // Данная функция производит пересчет
	int index_current, index_prev, index_neighbor_current, index_neighbor_prev;
                // потока по сквозному пути
	Vertex current = stock;       // Текущая вершина - сток

	while (current.prev_vertex != '0')
                                                                        // Пока не дошли
до истока,
	{

	// находим текущую и предыдущую
		index_current = findVertex(vector_of_vertex, current.vertex);
                                                        // текущей вершины, а в них
ребра
		index_prev = findVertex(vector_of_vertex, current.prev_vertex);
                                                        // друг до друга.
		index_neighbor_current =
findneighborIndex(vector_of_vertex[index_current], current.prev_vertex);

		index_neighbor_prev = findneighborIndex(vector_of_vertex[index_prev],
current.vertex);

		if (index_neighbor_current == -1)
                        // Если из текущей вершины нет ребра
		{
                                                // до предыдущей в сквозном пути
			neighbor neighbor = neighbor(current.prev_vertex);
			// вершины, то оно создается и
			neighbor.flow = max_flow;
                                // его пропускная способность
			neighbor.start_flow = 0;
                                // инициализируется максимальным
```

```
                vector_of_vertex[index_current].neighbors.push_back(neighbor);

            }
            else
                                                                        // Иначе
пропускная способность

      vector_of_vertex[index_current].neighbors[index_neighbor_current].flow +=
max_flow;         // увеличивается на величину максимального

                                                                              //
потока данного свозного пути
            vector_of_vertex[index_prev].neighbors[index_neighbor_prev].flow -=
max_flow;         // Пропускная способность из предыдущей в сквозном
            current = vector_of_vertex[index_prev];
                                          // пути вершины до текущей уменьшается на
величину
      }
}

bool cmpVertex(const Vertex& a, const Vertex& b) {
      if (a.vertex < b.vertex)
            return true;
      else
            return false;
}

bool cmpneighbors(const neighbor& a, const neighbor& b) {
      if (a.vertex < b.vertex)
            return true;
      else
            return false;
}


int main() {
      int count, start_flow, max_flow, index;
      int Pmax = 0;
      char source, stock, start, end;
      std::vector<Vertex> vector_of_vertex, visited_vertex;

      std::cin >> count >> source >> stock;

      for (int i = 0; i < count; ++i)           // Создается структура графа
      {
            std::cin >> start >> end >> start_flow;
            Vertex first;
            Vertex second;

            if (!isExist(vector_of_vertex, start))
            {
                        // Если начальной вершины ещё нет
                  first = Vertex(start);
            // в векторе вершин, то создаём её,
                  neighbor neighbor = neighbor(end, start_flow);        // её
соседа(конечную вершину) и
                  first.neighbors.push_back(neighbor);
      // добавляем соседа в вектор соседей.
                  vector_of_vertex.push_back(first);
      // Затем добавляем вершину в вектор вершин
            }
            else
```

```
            {
                                    // Иначе если начальная вершина уже есть
                  neighbor neighbor = neighbor(end, start_flow);
      // в векторе вершин, то создаём её соседа,
                  index = findVertex(vector_of_vertex, start);
            // находим начальную вершину в векторе вершин
                  vector_of_vertex[index].neighbors.push_back(neighbor);
      // и добавляем соседа в вектор соседей найденной
            }
                                    // вершины.

            if (!isExist(vector_of_vertex, end))            // Если конечной
вершины нет в векторе вершин,
            {                                                                   // то
она создается и добавляется в вектор вершин
                  second = Vertex(end);
                  vector_of_vertex.push_back(second);
            }
      }

      index = findVertex(vector_of_vertex, source);
      vector_of_vertex[index].prev_vertex = '0';
      Vertex current = vector_of_vertex[index];       // Текущая вершина - исток
      visited_vertex.push_back(current);

      while (1) {
            int index_min_priority = -1;
            int priority, min_priority;
            bool near_start;

            for (int i = 0; i < current.neighbors.size(); i++)                //
Поиск соседа текущей вершины
            {


                  if (current.neighbors[i].flow > 0 && !isExist(visited_vertex,
current.neighbors[i].vertex))       // Если пропускная способность пути до соседа
                  {

            // больше нуля и сосед не находится в векторе
                        priority = abs(current.neighbors[i].vertex -
current.vertex);                                                    // посещенных
вершин, то рассчитывается приоритет

                        if (index_min_priority == -1 || priority < min_priority)
            // Ищется сосед с минимальным приоритетом и в случае
                        {
                                          // равенства приоритетов, выбирается
сосед
                              min_priority = priority;
                              // имя которого в алфавите ближайшее к началу
алфавита.
                              index_min_priority = i;



                              if (current.neighbors[i].vertex < current.vertex)
            // Так как при одинаковом приоритете соседи могут находиться
                                    near_start = true;
                              // либо до, либо после на одинаковом "расстоянии" от
текущей
```

```
                              else
                              // вершины, то переменная near_start отвечает за это
положение.
                                    near_start = false;
                        }
                        else if (priority == min_priority)
                              if (current.neighbors[i].vertex < current.vertex &&
near_start == false)
                              {
                                    index_min_priority = i;
                                    near_start = true;
                              }
                  }
            }

            if (index_min_priority != -1)
                                                            // Если сосед был
найден
            {
                                                                        // он
находится в векторе
                  index = findVertex(vector_of_vertex,
current.neighbors[index_min_priority].vertex);        // вершин и предыдущей
вершиной
                  vector_of_vertex[index].prev_vertex = current.vertex;
                                          // к нему указывается текущая
                  current = vector_of_vertex[index];

                  visited_vertex.push_back(current);

                  if (current.vertex == stock)
                              // Если на данной итерации дошли до стока
                  {
                                                // рассчитывается
максимальный поток,
                        max_flow = maxFlowCount(vector_of_vertex, current);
                              // производится пересчет пропускных способностей
пути,
                        recountFlow(vector_of_vertex, vector_of_vertex[index],
max_flow);        // рассчитывается итоговый максимальный поток
                              Pmax += max_flow;

                              visited_vertex.clear();
                              index = findVertex(vector_of_vertex, source);
                                    // Текущей вершиной становится исток
                              current = vector_of_vertex[index];
                              visited_vertex.push_back(current);
                  }
            }
            else        // Если сосед не был найден
            {
                  if (current.prev_vertex == '0')           // Если текущая вершина
- исток, то больше сквозных путей не построить.
                        break;            // Переход к формированию решения
                  else        // Иначе, откат к предыдущей вершине
                  {
                        index = findVertex(vector_of_vertex, current.prev_vertex);
                        current = vector_of_vertex[index];
                  }
            }
      }
```

```
      for (int i = 0; i < vector_of_vertex.size(); ++i)           // Рассчитываются
конечные потоки через все ребра в графе
            for (int j = 0; j < vector_of_vertex[i].neighbors.size(); ++j)
                  vector_of_vertex[i].neighbors[j].final_flow =
vector_of_vertex[i].neighbors[j].start_flow -
vector_of_vertex[i].neighbors[j].flow;

      sort(vector_of_vertex.begin(), vector_of_vertex.end(), cmpVertex);
      // Сортировка графа
      for (int i = 0; i < vector_of_vertex.size(); ++i)
            sort(vector_of_vertex[i].neighbors.begin(),
vector_of_vertex[i].neighbors.end(), cmpneighbors);

      // Вывод результата
      std::cout << Pmax << std::endl;

      for (int i = 0; i < vector_of_vertex.size(); ++i)
            for (int j = 0; j < vector_of_vertex[i].neighbors.size(); ++j)
                  if (vector_of_vertex[i].neighbors[j].start_flow != 0)       //
Если ребро было задано изначально(а не создано во время алгоритма)
                  {
                        std::cout << vector_of_vertex[i].vertex << " " <<
vector_of_vertex[i].neighbors[j].vertex << " ";

                        if (vector_of_vertex[i].neighbors[j].final_flow >= 0)
                              std::cout <<
vector_of_vertex[i].neighbors[j].final_flow;
                        else
                              std::cout << 0;

                        std::cout << std::endl;
                  }
      return 0;
}
```