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: Максимальный поток сети

Студентка гр. 9383 _____ Карпекина А.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Изучить и реализовать алгоритм Форда-Фалкерсона для поиска максимального потока в сети.

Постановка задачи.

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 имен вершин и целого неотрицательного числа - пропускной способности (веса).

Входные данные:

N- количество ориентированных рёбер графа

*v*0 - исток

vn - сток

vi vj ωij - ребро графа

...

Выходные данные:

Pmax - величина максимального потока

vi vj ωij - 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).

Вариант 3.

Поиск в глубину. Рекурсивная реализация.

Описание алгоритма.

Алгоритм Форда-Фалкерсона запускает поиск в глубину до тех пор, пока путь возможно найти. После находит ребро с минимальной пропускной способностью и уменьшает пропускную способность всех рёбер, содержащихся в этом пути.

Сложность алгоритма в худшем случае равна:

$O(F|E|)$, где F - максимальный поток, E - множество дуг

Описание структур и функций.

Для представления графа был реализован класс Graph. Этот класс содержит вектор указателей на элементы класса Node.

В классе Graph реализованы следующие методы:

- bool IsThere (char Mark) - определяет наличие в графе вершины с заданным именем;
- bool AppendNode (char Mark) - добавляет новые узлы в граф;
- void AppendEdge (char markfir, char marksec, int EdgeWeight, bool fl = false) - для добавления новых дуг в граф;
- unsigned int FordFulkerson (char SourceMark, char DrainMark) - реализация алгоритма поиска максимального потока в графе;
- bool DFS (Node * actual, char DrainMark, vector < pair < Node *, edge >> &way) - обход графа в глубину;
- void SortMark () для упорядочивания в лексикографическом порядке;
- friend ostream & operator << (ostream & out, const Graph & graph) для вывода графа.

Класс Node содержит поля:

- mark - хранение имени узла;
- SetOfEdges - вектор инцидентных рёбер данной вершины;
- attend - хранит информацию о посещении вершины;

В классе Node реализованы следующие методы:

- char Mark () const - возвращает имя данного узла;
- void AppendEdge (Node * node, unsigned int EdgeWeight, bool fl = false) - для добавления инцидентных рёбер в вектор SetOfEdges;
- vector < edge > edges ()const - возвращает вектор SetOfEdges;
- bool IsAttend () const - определяет, была ли вершина посещена ранее;
- void Attend (bool visit) - изменяет статус просмотренности вершины;
- void SortRealandMark () - сортирует вектор SetOfEdges по фиктивности ребра и имени конечной вершины;
- void SortMark () - сортирует вектор SetOfEdges по имени конечной вершины;
- bool CutBandwidth (edge ActualEdge, unsigned int stream) - уменьшает вес ребра;
- bool EnhanceBandwidth (Node * node, unsigned int stream) - увеличивает вес ребра;
- friend ostream & operator << (ostream & out, const Node * node) - вывод вершины.

Тестирование.

Таблица 1 - результаты тестирования

Тест	Входные данные	результат работы алгоритма
№1	7	12

	a f a b 7 a c 6 b d 6 c f 9 d e 3 d f 4 e c 2	a b 6 a c 6 b d 6 c f 8 d e 2 d f 4 e c 2
№2	9 a d a b 8 b c 10 c d 10 h c 10 e f 8 g h 11 b e 8 a g 10 f d 8	18 a b 8 a g 10 b c 0 b e 8 c d 10 e f 8 f d 8 g h 10 h c 10
№3	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	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

	b e 30 e b 30 d e 10 e d 10	e c 0 e d 0

Вывод.

В ходе работы было изучено понятие потока, а также алгоритм Форда-Фалкерсона для поиска максимального потока в сети. Алгоритм был рекурсивно реализован с использованием обхода в глубину.

ПРИЛОЖЕНИЕ А
Исходный код программы

Название файла: lb3.cpp

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

using namespace std;

class Node;
using Bandwidth = pair < unsigned int, unsigned int >;
using edge = tuple < Node *, Bandwidth, bool >;

class Node
{
public:
  Node (char Mark):mark (Mark)
  {
    this->attend = false;
  }

  char Mark () const
  {
    return this->mark;
  }

  void AppendEdge (Node * node, unsigned int EdgeWeight, bool fl = false)
  {
  for (auto & edge:this->SetOfEdges)
```

```
    if (get < Node * >(edge)->Mark () == node->Mark ()
          && get < bool > (edge) == fl)
        {
          get < Bandwidth > (edge).second += EdgeWeight;
          return;
        }
  this->SetOfEdges.emplace_back (node, make_pair (0, EdgeWeight), fl);
}

vector < edge > edges ()const
{
  return this->SetOfEdges;
}

bool IsAttend () const
{
  return this->attend;
}

void Attend (bool visit)
{
  this->attend = visit;
}

void SortRealandMark ()
{
  sort (this->SetOfEdges.begin (), this->SetOfEdges.end (),
          [](edge edfir, edge edsec)
          {
          if (get < Node * >(edfir)->Mark () == get <
```

```
            Node * >(edsec)->Mark ())return get < bool > (edfir);
        else
        return get < Node * >(edfir)->Mark () < get <
        Node * >(edsec)->Mark ();}
  );
}

void SortMark ()
{
  sort (this->SetOfEdges.begin (), this->SetOfEdges.end (),
        [](edge edfir, edge edsec)
        {
        return get < Node * >(edfir)->Mark () < get <
        Node * >(edsec)->Mark ();}
  );
}

bool CutBandwidth (edge ActualEdge, unsigned int stream)
{
  if (get < Bandwidth > (ActualEdge).second - get < Bandwidth >
       (ActualEdge).first < stream)
    return false;

  if (get < bool > (ActualEdge))
    {
    for (auto & edge:this->SetOfEdges)
        if (get < Node * >(edge)->Mark () == get <
          Node * >(ActualEdge)->Mark ()
          && get < bool > (edge) == get < bool > (ActualEdge))
         {
```

```
            get < Bandwidth > (edge).second -= stream;
            break;
           }
       if (get < Node * >(ActualEdge)->EnhanceBandwidth (this, stream))
        return true;
  for (auto & edge:this->SetOfEdges)
       if (get < Node * >(edge)->Mark () == get <
          Node * >(ActualEdge)->Mark ()
          && get < bool > (edge) == get < bool > (ActualEdge))
         {
          get < Bandwidth > (edge).second += stream;
          break;
         }
      return false;
  }
 else
  {
  for (auto & edge:this->SetOfEdges)
       if (get < Node * >(edge)->Mark () == get <
          Node * >(ActualEdge)->Mark ()
          && get < bool > (edge) == get < bool > (ActualEdge))
         {
          get < Bandwidth > (edge).first += stream;
          break;
         }
      get < Node * >(ActualEdge)->AppendEdge (this, stream, true);
      return true;
  }
 return false;
}
```

```
  friend ostream & operator << (ostream & out, const Node * node)
  {
  for (auto & edge:node->SetOfEdges)
     if (!get < bool > (edge))
         out << node->Mark () << ' ' << get <
           Node * >(edge)->Mark () << ' ' << get < Bandwidth >
           (edge).first << '\n';
    return out;
  }

protected:
  bool EnhanceBandwidth (Node * node, unsigned int stream)
  {
  for (auto & edge:this->SetOfEdges)
     if (get < Node * >(edge)->Mark () == node->Mark ()
          && !get < bool > (edge))
         {
          get < Bandwidth > (edge).first -= stream;
          return true;
         }
    return false;
  }

private:
  char mark;
  vector < edge > SetOfEdges;
  bool attend;

};
```

```
class Graph
{
public:
  Graph () = default;

  ~Graph ()
  {
  for (auto & node:this->SetOfNodes)
     delete node;
  }

  bool IsThere (char Mark)
  {
  for (auto & node:this->SetOfNodes)
     if (node->Mark () == Mark)
         return true;
   return false;
  }

  bool AppendNode (char Mark)
  {
   if (!this->IsThere (Mark))
     this->SetOfNodes.push_back (new Node (Mark));
   else
     return false;
   return true;
  }

  void AppendEdge (char markfir, char marksec, int EdgeWeight, bool fl =
```

```
                    false)
{
  if (markfir == marksec)
    return;
  Node *nodefir, *nodesec;
for (auto & node:this->SetOfNodes)
    {
        if (node->Mark () == markfir)
          nodefir = node;
        if (node->Mark () == marksec)
          nodesec = node;
    }
  nodefir->AppendEdge (nodesec, EdgeWeight, fl);
}

unsigned int FordFulkerson (char SourceMark, char DrainMark)
{
  Node *Source = nullptr, *Drain = nullptr;
for (auto & node:this->SetOfNodes)
    {
        if (node->Mark () == SourceMark)
          Source = node;
        if (node->Mark () == DrainMark)
          Drain = node;
    }
  if (Source == nullptr || Drain == nullptr)
    throw "Such element doesnt exist";
  vector < pair < Node *, edge >> way;
  while (DFS (Source, DrainMark, way))
    {
```

```
        unsigned int ActualStream = numeric_limits < unsigned int >::max ();
    for (auto & obj:way)
        if (get < Bandwidth > (obj.second).second - get < Bandwidth >
          (obj.second).first < ActualStream)
         ActualStream =
          get < Bandwidth > (obj.second).second - get < Bandwidth >
          (obj.second).first;
    for (auto & obj:way)
        obj.first->CutBandwidth (obj.second, ActualStream);
       way.clear ();
    }
  int MaxStream = 0;
 for (auto & edge:Source->edges ())
    MaxStream += get < Bandwidth > (edge).first;
  this->SortMark ();
  return MaxStream;
 }

 friend ostream & operator << (ostream & out, const Graph & graph)
 {
 for (auto & node:graph.SetOfNodes)
    out << node;
  return out;
 }

protected:
 bool DFS (Node * actual, char DrainMark, vector < pair < Node *,
          edge >> &way)
 {
  if (actual->Mark () == DrainMark)
```

```
    return true;
  actual->SortRealandMark ();
  actual->Attend (true);
for (auto & edge:actual->edges ())
    if (!get < Node * >(edge)->IsAttend ()
          && get < Bandwidth > (edge).second != get < Bandwidth >
          (edge).first)
         {
          if (DFS (get < Node * >(edge), DrainMark, way))
            {
              way.emplace_back (actual, edge);
              actual->Attend (false);
              return true;
            }
         }
  actual->Attend (false);
  return false;
}

void SortMark ()
{
for (auto & node:this->SetOfNodes)
    node->SortMark ();
  sort (this->SetOfNodes.begin (), this->SetOfNodes.end (),
          [](Node * nodefir, Node * nodesec)
          {
          return nodefir->Mark () < nodesec->Mark ();}
  );
}
```

```
private:
  vector < Node * >SetOfNodes;

};

int
main ()
{
  Graph graph;
  int NumOrientEdges, EdgeWeight;
  char Source, Drain, SENode, DENode;
  cin >> NumOrientEdges;
  cin >> Source;
  cin >> Drain;
  for (int i = 0; i < NumOrientEdges; i++)
    {
      cin >> SENode >> DENode >> EdgeWeight;
      graph.AppendNode (SENode);
      graph.AppendNode (DENode);
      graph.AppendEdge (SENode, DENode, EdgeWeight);
    }
  auto MaxStream = graph.FordFulkerson (Source, Drain);
  cout << MaxStream << '\n' << graph;
  return 0;
}
```