МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО АВТОНОМНОГО ОБРАЗОВАТЕЛЬНОГО УЧРЕЖДЕНИЯ ВЫСШЕГО ОБРАЗОВАНИЯ

**«Национальный исследовательский технологический университет «МИСИС»**

**Институт Компьютерных Наук**

**Отчет**

**Задача построения максимального потока в сети. Алгоритм Диницы.**

**По курсу:** Комбинаторика и теория графов

**Ссылка на репозиторий:**

[**https://github.com/SadLiter/Combinatorics-and-graph-theory**](https://github.com/SadLiter/Combinatorics-and-graph-theory)

Волков Валентин Александрович

Группа БИВТ-23-6

**Отчет: Построение максимального потока в сети с использованием алгоритма Диница**

**Содержание**

1. Формальная постановка задачи
2. Теоретическое описание алгоритма и его характеристики
3. Сравнительный анализ с аналогичными алгоритмами
4. Перечень инструментов, используемых для реализации
5. Описание реализации и процесса тестирования

**1. Формальная постановка задачи**

**Задача:**  
Построение максимального потока в сети, представленной ориентированным графом. Поток должен быть максимальным, удовлетворяя следующим условиям:

1. **Ограничение пропускной способности:** Поток по любому ребру не может превышать его пропускную способность.
2. **Сохранение потока:** Для каждой вершины, кроме истока и стока, сумма входящих потоков должна быть равна сумме исходящих потоков.

**Входные данные:**

* Ориентированный граф G=(V,E), где:
  + V — множество вершин;
  + E — множество рёбер с пропускными способностями c(u,v) ≥ 0 для каждого ребра (u,v) ∈ E.
* Две выделенные вершины: исток s ∈ V и сток t ∈ V.

**Выходные данные:**  
Максимальный поток f, который можно передать из истока s в сток t.

**2. Теоретическое описание алгоритма и его характеристики**

**Описание алгоритма Диница**

Алгоритм Диница использует метод построения уровневого графа и поиска блокирующих потоков. Основные шаги:

1. **Построение уровневого графа (BFS):**
   * Выполняется обход в ширину от истока s, чтобы назначить каждому узлу уровень. Уровень вершины равен минимальному числу рёбер от s до этой вершины.
   * Если сток t недостижим, алгоритм завершает работу.
2. **Поиск блокирующего потока (DFS):**
   * Выполняется обход в глубину, начиная с истока s, для нахождения всех путей до стока t в уровневом графе.
   * Потоки по найденным путям увеличиваются до тех пор, пока хотя бы одно ребро остаётся не полностью заполненным.
3. **Повторение:**
   * Если блокирующий поток был найден, уровневый граф перестраивается, и процесс повторяется.

**Характеристики алгоритма**

* **Временная сложность:**
  + O(V^2 E) для общего случая.
  + O(VElogC) при использовании дискретизации пропускных способностей.
* **Пространственная сложность:**
  + O(V+E) для хранения графа и уровневого графа.
* **Применимость:**
  + Эффективен для плотных графов и графов с большими потоками.

**3. Сравнительный анализ с аналогичными алгоритмами**

**Сравнение с алгоритмами Форда-Фалкерсона и Эдмондса-Карпа**

| **Критерий** | **Алгоритм Диница** | **Форд-Фалкерсон** | **Эдмондс-Карп** |
| --- | --- | --- | --- |
| **Временная сложность** | O(V2E)O(V^2 E) | O(E⋅f) | O(V⋅E^2) |
| **Подход** | Уровневый граф и блокирующий поток | Любой путь увеличения | BFS для кратчайших путей |
| **Скорость на практике** | Быстрая | Медленная | Средняя |
| **Сложность реализации** | Средняя | Простая | Средняя |
| **Применимость** | Плотные графы | Графы с малыми потоками | Универсальный |

**Вывод:**

Алгоритм Диница превосходит другие методы для графов с большими потоками и плотной структурой. Он строит уровневые графы и эффективно ищет блокирующие потоки, что позволяет сократить общее число операций.

**4. Перечень инструментов, используемых для реализации**

Для реализации алгоритма Диница использовались следующие инструменты:

* **Языки программирования:**
  + **Python 3.9+**: Простота реализации и тестирования.
  + **C++:** Для высокопроизводительной реализации.
* **Среда разработки:** Visual Studio Code / GCC
  + Удобство написания кода и встроенная поддержка Python.
* **Модуль unittest:** Для тестирования алгоритма.
* **C++:** Стандартные библиотеки (STL).
* **Система контроля версий:** Git
  + Для управления изменениями в коде.
* **Текстовый редактор:** Notepad++ (для подготовки входных данных).

**5. Описание реализации и процесса тестирования**

**Реализация алгоритма**

Код алгоритма Диница реализован в файле dinic.py. Основные компоненты:

1. **Класс Edge:**
   * Представляет ребро графа с пропускной способностью и текущим потоком.
2. **Класс Dinic:**
   * Методы:
     + add\_edge: Добавляет ребро и его обратное ребро.
     + bfs: Строит уровневый граф.
     + dfs: Находит блокирующий поток.
     + max\_flow: Возвращает максимальный поток между истоком и стоком.
3. **Функция main:**
   * Читает входные данные, строит граф и вычисляет максимальный поток.

**Реализация на C++**

Код на C++ (dinic.cpp) использует стандартные контейнеры (векторы, очереди) и обеспечивает высокую производительность. Основные компоненты:

1. **add\_edge**: Добавляет прямое и обратное рёбра.
2. **bfs**: Построение уровневого графа.
3. **dfs**: Поиск блокирующего потока.
4. **max\_flow**: Вычисление максимального потока.

**Пример входных данных**

6 10

0 1 16

0 2 13

1 2 10

1 3 12

2 1 4

2 4 14

3 2 9

3 5 20

4 3 7

4 5 4

0 5

**Пример вывода программы**

Максимальный поток: 23

**Процесс тестирования**

Тестирование проводилось с использованием модуля unittest. Для проверки корректности были подготовлены тестовые случаи:

1. **Пустой граф:**  
   Проверяется отсутствие потока, если рёбра отсутствуют.
2. **Граф с одним ребром:**  
   Проверяется, что поток равен пропускной способности единственного ребра.
3. **Сложные графы:**  
   Проверяются графы с несколькими путями, циклами и параллельными рёбрами.
4. **Большие графы:**  
   Проводится тестирование на графах с большим количеством рёбер и вершин для проверки производительности.

Для C++ входные данные можно генерировать в input.txt и проверять корректность вывода.

**Код тестирования**

import unittest

from dinic import Dinic

class TestDinicAlgorithm(unittest.TestCase):

def test\_empty\_graph(self):

dinic = Dinic(2)

self.assertEqual(dinic.max\_flow(0, 1), 0)

def test\_single\_edge(self):

dinic = Dinic(2)

dinic.add\_edge(0, 1, 10)

self.assertEqual(dinic.max\_flow(0, 1), 10)

def test\_complex\_graph(self):

dinic = Dinic(6)

edges = [

(0, 1, 16),

(0, 2, 13),

(1, 2, 10),

(1, 3, 12),

(2, 1, 4),

(2, 4, 14),

(3, 2, 9),

(3, 5, 20),

(4, 3, 7),

(4, 5, 4),

]

for u, v, c in edges:

dinic.add\_edge(u, v, c)

self.assertEqual(dinic.max\_flow(0, 5), 23)

if \_\_name\_\_ == "\_\_main\_\_":

unittest.main()

## 6. Преимущества реализации на Python и C++

### Python

* Быстрая разработка и легкость тестирования.
* Подходит для анализа небольших графов и демонстрации работы алгоритма.

### C++

* Высокая производительность и низкое время выполнения.
* Подходит для работы с большими графами (до сотен тысяч рёбер).

## 7. Заключение

Алгоритм Диница эффективно решает задачу построения максимального потока в сети. Реализация на Python позволяет быстро разрабатывать и отлаживать алгоритм, а C++ обеспечивает высокую производительность на больших графах.

**Основные выводы:**

1. Алгоритм Диница превосходит альтернативы (например, Форда-Фалкерсона) на плотных графах.
2. Реализация на Python подходит для обучения и отладки, а C++ — для производственного использования.