Национальный исследовательский технологический университет «МИСиС»
Институт Информационных технологий и компьютерных наук (ИТКН)

Дисциплина «Комбинаторика и теория графов»

Отчет по теме «Задача построения максимального потока в сети. Алгоритм Диницы»

Выполнил:

Студент группы БИВТ 23-8

Хомушку Марк

Ссылка на репозиторий:

https://github.com/greeveus/protoss

Москва 2024

Оглавление

Постановка задачи 2

Теоретическое описание 2

Харакетристика 6

Реализация 10

Процесс тестирования 15

Постановка задачи

Задача построения максимального потока в сети представляет собой одну из фундаментальных задач теории графов и оптимизации. Она состоит в определении максимального возможного потока из истока (source) в сток (sink) через ориентированную сеть с заданными пропускными способностями рёбер.

Описание задачи

Дана ориентированная сеть $G=(V,E)$, где:

- V — множество вершин,
- E — множество рёбер,
- Каждое ребро $(u,v) \in E$ имеет пропускную способность $c \geq 0$.

В сети выделены две особые вершины:

- s (исток),
- t (сток).

Требуется найти максимальный поток из s в t, то есть распределение потока f по рёбрам сети, удовлетворяющее следующим условиям:

1. Ограничение пропускной способности: Для любого $(u,v) \in E$ выполняется $0 \leq f(u,v) \leq c(u,v)$.
2. Сохранение потока: Для любой вершины $u \neq s,t$ сумма входящих потоков равна сумме исходящих.
3. Максимизация потока: Суммарный поток, выходящий из источника s, должен быть максимальным.

Теоретическое описание

Основная идея

Алгоритм Диница основывается на идее разделения задачи поиска максимального потока на итеративные этапы с использованием уровневой сети. На каждом этапе сначала строится уровневый граф с использованием алгоритма поиска в ширину (BFS), который определяет минимальные расстояния от истока до всех вершин. Затем в этой сети выполняется поиск

блокирующего потока с помощью алгоритма углубленного поиска (DFS), эффективно улучшая текущее решение. Этот процесс продолжается до тех пор, пока в дополняющей сети существует путь из истока в сток.

Определения:

Обход в ширину (Breadth-First Search, BFS) — это фундаментальный алгоритм теории графов, используемый для поиска и обхода всех вершин графа. Основная идея BFS заключается в том, чтобы исследовать все вершины на текущем уровне перед переходом к вершинам следующего уровня.

Шаги алгоритма BFS

1. Инициализация:
    - Поместить исходную вершину s в очередь.
    - Пометить s как посещённую.
    - Назначить уровень level(s)=0.
2. Основной цикл:
    - Пока очередь не пуста:
        1. Извлечь вершину u из начала очереди.
        2. Для каждой вершины v, смежной с u
            - Если v ещё не посещена:
                - Пометить v как посещённую.
                - Назначить уровень level(v)=level(u)+1.
                - Добавить v в конец очереди.
3. Завершение:
    - Алгоритм завершается, когда очередь становится пустой, то есть когда все достижимые от s вершины были посещены.

Применение BFS

В алгоритме Диница BFS используется для построения уровневого графа.

Обход в глубину (Depth-First Search, DFS) — это фундаментальный алгоритм теории графов, используемый для поиска и обхода всех вершин графа. Основная идея DFS заключается в том, чтобы исследовать как можно глубже по каждой ветви графа перед тем, как откатиться назад и перейти к другим ветвям.

Шаги алгоритма DFS

1. Инициализация:
    - Выбрать исходную вершину s и пометить её как посещённую.
    - Поместить s в стек.
2. Основной цикл (рекурсивный или с использованием стека):
    - Пока стек не пуст:
        1. Извлечь вершину u из вершины стека.
        2. Для каждой вершины v, смежной с u
            - Если v ещё не посещена:
                - Пометить v как посещённую.
                - Поместить v в стек для дальнейшего исследования.
3. Завершение:
    - Алгоритм завершается, когда все достижимые от s вершины были посещены.

Применение DFS

В алгоритме Диница DFS используется для поиска путей увеличения потока в построенной уровневой сети. После построения уровневой сети с помощью BFS, DFS последовательно в глубину ищет все возможные пути от истока s к стоку t в этой уровневой сети и добавляет поток по найденным путям до тех пор, пока не будет достигнут блокирующий поток.

Уровневый граф — это вспомогательная структура, используемая в алгоритме Диница для организации поиска блокирующих потоков. Он строится на основе текущего состояния потоков и обеспечивает слоистую структуру сети, где все пути от источника к стоку проходят через вершины последовательно увеличивающихся уровней.

Блокирующий поток — это поток, который нельзя дополнительно увеличить без изменения уровневого графа. В контексте уровневого графа блокирующий поток заполняет все возможные пути от s до t, не оставляя возможности для дальнейшего увеличения потока по текущим уровням.

Метод поиска блокирующего потока:

- Используется поиск в глубину (DFS) для нахождения всех возможных увеличивающих путей в уровневом графе.

- На каждом найденном пути определяется минимальная остаточная пропускная способность, и поток увеличивается на это значение.
- Процесс продолжается до тех пор, пока мы не сможем найти новые пути для увеличения потока в текущем уровневом графе.

Остаточная сеть $G_f=(V,E_f)$ строится на основе исходной сети $G=(V,E)$ и текущего потока f. Она включает все оригинальные рёбра с остаточной пропускной способностью, а также обратные рёбра, позволяющие корректировать поток.

Формирование Остаточной Сети:

1. Остаточная Пропускная Способность Оригинальных Ребер: Для каждого ребра $(u,v)\in E$ остаточная пропускная способность определяется как:

   $$c_f(u,v)=c(u,v)-f(u,v)$$

   где $c(u,v)$ — исходная пропускная способность, а $f(u,v)$ — текущий поток по этому ребру.

2. Добавление Обратных Ребер: Для каждого ребра $(u,v)\in E$ добавляется обратное ребро (v,u) с пропускной способностью:

   $$c_f(v,u)=f(u,v)$$

   Это позволяет алгоритму уменьшать поток по ребру (u,v) при необходимости.

Итоговое описание алгоритма

1: Инициализация

- Инициализируем поток $f(u,v)=0$ для всех рёбер $(u,v)\in E$.

2: Построение Остаточной Сети

- На основе текущего потока f строим остаточную сеть $G_f=(V,E_f)$ включающую как прямые, так и обратные рёбра с соответствующими остаточными пропускными способностями.

3: Построение Уровневого Графа

- Выполняем BFS от источника s, присваивая уровни каждой достижимой вершине.
- Уровневый граф включает только те рёбра, которые соединяют вершины из соседних уровней и имеют положительную остаточную пропускную способность.

4: Проверка возможности достижения стока

- Если сток t недостижим в текущем графе, алгоритм завершается, и текущий поток является максимальным.

5: Поиск Блокирующего Потока

- Используя DFS, находим возможные пути от s до t в уровневом графе.
- Потоки направляются только по рёбрам уровневого графа, обеспечивая отсутствие циклов и улучшая эффективность поиска.

6: Обновление потоков и повторение

- После нахождения блокирующего потока обновляются значения потоков по рёбрам сети.
- Повторяются этапы построения остаточной сети, уровневого графа и поиска блокирующих потоков до тех пор, пока существует путь от s до t в уровневом графе.

Харакетристика

Временная сложность

1. **Построение Уровневого Графа (BFS):**
    - Время выполнения: O(E)
    - Описание: На каждой итерации алгоритм выполняет поиск в ширину (BFS) для построения уровневого графа. Время выполнения BFS линейно зависит от количества рёбер, поскольку каждый ребро рассматривается не более одного раза за каждую итерацию BFS.
2. **Поиск Блокирующего Потока (DFS):**
    - Время выполнения: O(E) на каждую итерацию поиска блокирующего потока.

- Описание: После построения уровневого графа выполняется поиск в глубину (DFS) для нахождения и отправки блокирующих потоков. Время выполнения DFS также линейно зависит от количества рёбер.

3. **Количество Итераций (Уровневых Графов)**
   Время Выполнения: O(V) итераций.
   Пояснение:

   - Рост Уровней: В каждой итерации BFS увеличивает уровень вершины, минимально необходимого количества рёбер от источника.
   - Максимальное Количество Уровней: В худшем случае, максимальное количество уровней в графе не превышает V (количество вершин), поскольку каждое новое ребро может увеличивать уровень на 1.

4. **Итоговая Временная Сложность:**
   При этом, поскольку алгоритм может требовать до O(V) итераций для достижения максимального потока, общая временная сложность составляет: O(V^2 E)

Пространственная сложность

Алгоритм Диница имеет пространственную сложность:

O(V+E)

где:

- V — количество вершин в графе.
- E — количество рёбер в графе.

Описание:

- Хранение Графа: Используется список смежности для хранения рёбер, что требует O(V+E) памяти.
- Дополнительные Структуры: Массив уровней O(V) и массив указателей для DFS O(V).

Сравнительный анализ

Сравнительный Анализ Алгоритмов Диница, Форда-Фалкерсона и Эдмондса-Карпа

1. Временная Сложность

Алгоритм	Временная Сложность	Комментарий
Форд-Фалкерсона	O(E * max(f))	Использует DFS для поиска увеличиваюищих путей.
Эдмондса-Карпа	O(V×E^2)	Использует BFS для поиска кратчайших путей.
Диница	O(V^2×E)	Более эффективен на большинстве графов благодаря использованию уровневых графов и поиска блокирующих потоков — сочетает в себе DFS и BFS. В некоторых случаях может достигать O(V×E).

2. Пространственная Сложность

Алгоритм	Пространственная Сложность	Комментарий
Форд-Фалкерсона	O(V+E)	Простое хранение графа через списки смежности.
Эдмондса-Карпа	O(V+E)	Аналогично Форда-Фалкерсона, использует списки смежности и дополнительные структуры для BFS.
Диница	O(V+E)	Также использует списки смежности, а дополнительно требует хранение уровней и указателей для оптимизации DFS.

3. Практическая Эффективность

Алгоритм	Преимущества	Недостатки
Форд-Фалкерсона	- Простота реализации. - Подходит для небольших графов или	- Может быть неэффективен на больших графах.

Алгоритм	Преимущества	Недостатки
	графов с небольшими пропускными способностями.	- Время выполнения сильно зависит от выбора путей.
Эдмондса-Карпа	- Гарантированная конечная временная сложность. - Избегает зацикливания за счёт использования BFS.	- Медленнее современных алгоритмов. - Требует выполнения BFS на каждой итерации.
Диница	- Высокая эффективность на большинстве графов. - Оптимизирован для плотных и разреженных графов. - Использование уровневых графов снижает количество итераций.	- Более сложная реализация. - В худшем случае может быть менее эффективен, чем Push-Relabel.

4. Особенности и Применимость

Алгоритм	Особенности	Применимость
Форд-Фалкерсона	- Неограниченная временная сложность. - Может работать неоптимально без определённого выбора путей.	- Образовательные цели. - Небольшие или простые графы. - Случаи, где важна простота реализации, а не максимальная эффективность.
Эдмондса-Карпа	- Использует BFS для поиска кратчайших путей, что улучшает производительность по сравнению с произвольным выбором путей.	- Средние по размеру графы. - Приложения, требующие предсказуемой временной сложности. - Системы с ограниченными ресурсами, где требуется гарантия завершения.
Диница	- Использует уровневые графы и блокирующие потоки. - Эффективно управляет потоками через оптимизированные структуры.	- Большие и плотные графы. - Приложения, требующие высокой производительности. - Сложные сетевые задачи в

Алгоритм	Особенности	Применимость
		сетевых технологиях, логистике и других областях.

5. Преимущества Алгоритма Диница по Сравнению

1. Более Высокая Эффективность:
    - В большинстве случаев алгоритм Диница работает быстрее, чем Форда-Фалкерсона и Эдмондса-Карпа, особенно на больших и плотных графах.
2. Использование Уровневых Графов:
    - Позволяет структурировать поиск потоков, снижая количество итераций и обеспечивая более целенаправленный подход к нахождению увеличивающих путей.
3. Оптимизация Поиска Потоков:
    - Метод блокирующих потоков позволяет одновременно отправлять потоки по нескольким путям, что ускоряет процесс достижения максимального потока.
4. Гибкость и Универсальность:
    - Эффективен как на разреженных, так и на плотных графах, адаптируясь к различным структурам сетей.

Перечень инструментов

1. **Язык программирования:** Python версии 3.11
2. **Среда разработки:** IDLE (Python 3.11 64 bit)

Реализация

```
from collections import deque


class Edge:
    """
    Класс для представления ребра в сети потока.
    Каждый экземпляр содержит информацию о вершине назначения, индексе обратного ребра,
    пропускной способности и текущем потоке.
    """
```

```
    def __init__(self, to, rev, capacity):
        self.to = to          # Вершина назначения
        self.rev = rev        # Индекс обратного ребра в списке рёбер вершины 'to'
        self.capacity = capacity  # Остаточная пропускная способность
        self.flow = 0         # Текущий поток через ребро

class Dinic:
    """
    Класс для реализации Алгоритма Диница для нахождения максимального потока в сети.
    """
    def __init__(self, N):
        """
        Инициализация графа.

        N: Количество вершин в графе
        """
        self.size = N
        self.graph = [[] for _ in range(N)]  # Список смежности для каждой вершины
        self.level = [ -1 ] * N              # Уровни вершин
        self.ptr = [0] * N                   # Текущие индексы для оптимизации DFS

    def add_edge(self, fr, to, capacity):
        """
        Добавление ребра в граф.

        fr: Вершина начала ребра
        to: Вершина конца ребра
        capacity: Пропускная способность ребра
        """
        forward = Edge(to, len(self.graph[to]), capacity)
        backward = Edge(fr, len(self.graph[fr]), 0)  # Обратное ребро с нулевой пропускной способностью
        self.graph[fr].append(forward)
        self.graph[to].append(backward)
```

```
    def bfs(self, s, t):
        """
        Поиск в ширину для построения уровневого графа.

        s: Источник
        t: Сток
        True, если сток достижим из источника, иначе False
        """
        queue = deque()
        queue.append(s)
        self.level = [-1] * self.size
        self.level[s] = 0  # Уровень источника равен 0

        while queue:
            v = queue.popleft()
            for edge in self.graph[v]:
                if self.level[edge.to] == -1 and edge.flow < edge.capacity:
                    self.level[edge.to] = self.level[v] + 1
                    queue.append(edge.to)
                    print(f"Вершина {edge.to} достигнута из вершины {v} с уровнем
{self.level[edge.to]}")
                    if edge.to == t:
                        return True  # Сток найден, можно прекратить BFS

        return self.level[t] != -1  # Возвращает True, если сток был достигнут

    def dfs(self, v, t, pushed):
        """
        Поиск в глубину для отправки потока.

        :param v: Текущая вершина
        :param t: Сток
        :param pushed: Поток, который нужно отправить
        :return: Поток, который был отправлен
        """
```

```
        if v == t:
            return pushed  # Поток достиг стока
        while self.ptr[v] < len(self.graph[v]):
            edge = self.graph[v][self.ptr[v]]
            if self.level[edge.to] == self.level[v] + 1 and edge.flow < edge.capacity:
                # Минимальная остаточная пропускная способность на пути
                tr = self.dfs(edge.to, t, min(pushed, edge.capacity - edge.flow))
                if tr > 0:
                    # Обновление потока и обратного потока
                    edge.flow += tr
                    self.graph[edge.to][edge.rev].flow -= tr
                    print(f"Отправлен поток {tr} через ребро {v} -> {edge.to}")
                    return tr
            self.ptr[v] += 1  # Перейти к следующему ребру
        return 0  # Невозможно отправить больше потока из этой вершины

    def max_flow(self, s, t):
        """
        Вычисление максимального потока из вершины s в вершину t.

        :param s: Источник
        :param t: Сток
        :return: Максимальный поток
        """
        flow = 0
        while self.bfs(s, t):
            self.ptr = [0] * self.size  # Сброс указателей для DFS
            pushed = self.dfs(s, t, float('inf'))
            while pushed > 0:
                flow += pushed
                print(f"Текущий общий поток: {flow}")
                pushed = self.dfs(s, t, float('inf'))
        return flow

def main():
```

```
    """
    Основная функция для демонстрации работы Алгоритма Диница.
    Создаётся пример сети, аналогичный приведённому в отчёте.
    """
    N = 4
    s = 0  # Источник
    t = 3  # Сток

    # Инициализация Алгоритма Диница
    dinic = Dinic(N)

    # Добавление рёбер (источник, вершина, пропускная способность)
    dinic.add_edge(0, 1, 3)  # s -> a
    dinic.add_edge(0, 2, 2)  # s -> b
    dinic.add_edge(1, 2, 1)  # a -> b
    dinic.add_edge(1, 3, 2)  # a -> t
    dinic.add_edge(2, 3, 3)  # b -> t

    # Вычисление максимального потока
    max_flow = dinic.max_flow(s, t)
    print(f"\nМаксимальный поток: {max_flow}")

    # Вывод потоков по рёбрам
    print("\nПотоки по рёбрам:")
    for u in range(N):
        for edge in dinic.graph[u]:
            if edge.capacity > 0:
                print(f"{u} -> {edge.to} : {edge.flow}")

if __name__ == "__main__":
    main()
```

Описание реализации

Основные Компоненты Реализации

1. Класс Edge представляет собой ребро в сети потока и содержит информацию о вершине назначения, индексе обратного ребра, пропускной способности и текущем потоке.
2. Класс Dinic реализует сам алгоритм Диница, управляя графом и вычисляя максимальный поток.
3. Методы add_edge**,** bfs**,** dfs**,** max_flow

   __init__(self, N): Инициализирует граф с N вершинами. Создаёт список смежности, массив уровней и массив указателей для оптимизации поиска.

   add_edge(self, fr, to, capacity): Добавляет прямое ребро из вершины fr в вершину to с заданной пропускной способностью, а также обратное ребро с нулевой пропускной способностью. Это необходимо для корректного управления потоками и возможности уменьшения потока по обратным рёбрам

   bfs(self, s, t): Строит уровневый граф с помощью поиска в ширину (BFS). Присваивает уровня каждой достижимой вершине от источника s. Возвращает True, если сток t достижим, иначе False.

   dfs(self, v, t, pushed): Выполняет поиск в глубину (DFS) для отправки потока от вершины v до стока t. Отправляет максимально возможный поток по найденному пути и обновляет соответствующие потоки и обратные рёбра.

   max_flow(self, s, t): Основной метод для вычисления максимального потока из вершины s в вершину t. Повторяет процесс построения уровневого графа и отправки блокирующих потоков до тех пор, пока сток остаётся достижимым.

4. Функция main для демонстрации работы алгоритма

Процесс тестирования

Входной граф (Рис 1)

```
    # Добавление рёбер (источник, вершина, пропускная способность)
    dinic.add_edge(0, 1, 3)  # s -> a
    dinic.add_edge(0, 2, 2)  # s -> b
    dinic.add_edge(1, 2, 1)  # a -> b
    dinic.add_edge(1, 3, 2)  # a -> t
    dinic.add_edge(2, 3, 3)  # b -> t
```

Рис 1

Вывод (Рис 2)

```
Вершина 1 достигнута из вершины 0 с уровнем 1
Вершина 2 достигнута из вершины 0 с уровнем 1
Вершина 3 достигнута из вершины 1 с уровнем 2
Отправлен поток 2 через ребро 1 -> 3
Отправлен поток 2 через ребро 0 -> 1
Текущий общий поток: 2
Отправлен поток 2 через ребро 2 -> 3
Отправлен поток 2 через ребро 0 -> 2
Текущий общий поток: 4
Вершина 1 достигнута из вершины 0 с уровнем 1
Вершина 2 достигнута из вершины 1 с уровнем 2
Вершина 3 достигнута из вершины 2 с уровнем 3
Отправлен поток 1 через ребро 2 -> 3
Отправлен поток 1 через ребро 1 -> 2
Отправлен поток 1 через ребро 0 -> 1
Текущий общий поток: 5

Максимальный поток: 5

Потоки по рёбрам:
0 -> 1 : 3
0 -> 2 : 2
1 -> 2 : 1
1 -> 3 : 2
2 -> 3 : 3
```

Рис 2

Ссылка на репозиторий**:** https://github.com/greeveus/protoss