

Оглавление

Глава 1. Алгоритмы решения задачи о максимальном потоке	3
1.1. Определение задачи о максимальном потоке	3
1.2. Алгоритм Форда-Фалкерсона	4
1.2.1. Описание алгоритма	4
1.2.2. Анализ метода Форда-Фалкерсона	6
1.3. Алгоритм Эдмондса-Карпа	7
1.3.1. Описание алгоритма	7
1.3.2. Анализ метода Эдмондса-Карпа	8
1.4. Алгоритм Диница	9
1.4.1. Описание алгоритма	9
1.4.2. Анализ алгоритма Диница	11
1.5. Алгоритм Слейтера-Тарьяна	13
1.5.1. Описание алгоритма	13
1.5.2. Анализ алгоритма Слейтера-Тарьяна	15
1.6. Алгоритм проталкивания предпотока	16
1.6.1. Описание алгоритма	16
1.6.2. Анализ алгоритма проталкивания предпотока	17
1.7. Сравнение алгоритмов	18
Глава 2. Практическая реализация алгоритмов решения задачи о максимальном потоке	19
2.1. Выбор структуры данных для хранения информации о вершинах и ребрах графа	19
2.2. Особенности реализации алгоритмов решения задачи о максимальном потоке	20
2.3. Результаты запуска кода	21
Заключение	23
Список литературы	24
Приложение	25

Введение

Задача о максимальном потоке изучается уже более полувека. Это обусловлено тем, что практическая значимость решения данной задачи велика. Методы решения задачи применяются, например, в транспортных и коммуникационных сетях, при моделировании различных процессов физики и химии.

В 1951 году Джордж Данциг впервые сформулировал задачу в общем виде. В 1955 году, Лестер Форд и Делберт Фалкерсон впервые построили алгоритм, специально предназначенный для решения этой задачи. Их алгоритм получил название алгоритм Форда-Фалкерсона. В дальнейшем решение задачи во много раз улучшилось.

В данной работе представлены примеры алгоритмов иллюстрирующие решение задачи. В практической части работы реализованы алгоритмы Форда-Фалкерсона, Эдмондса-Карпа и Диница на языке программирования C++ в среде разработки CLion.

Глава 1. Алгоритмы решения задачи о максимальном потоке

1.1. Определение задачи о максимальном потоке

Транспортная сеть $G = (V, E)$ представляет собой ориентированный граф, в котором каждое ребро $(u, v) \in E$ имеет неотрицательную пропускную способность $c(u, v) > 0$. В транспортной сети выделяются две вершины: источник - s и сток - t . Поток в G является действительная функция $f: V \times V \rightarrow R$, удовлетворяющая следующим трем условиям:

1. Ограничение пропускной способности: $f(u, v) \leq c(u, v)$ для всех $u, v \in V$
2. Антисимметричность: $f(u, v) = -f(v, u)$ для всех $u, v \in V$
3. Сохранение потока: для всех $u \in V - \{s, t\}$ $\sum_{v \in V} f(u, v) = 0$

Величина потока определяется так: $|f| = \sum_{v \in V} f(s, v)$, т.е. как суммарный поток, выходящий из источника.

В задаче о максимальном потоке дана некоторая транспортная сеть G с источником s и стоком t , и необходимо найти поток максимальной величины [1].

1.2. Алгоритм Форда-Фалкерсона

Впервые был предложен в 1956г. До этого времени задача решалась с помощью методов линейного программирования, что было крайне неэффективно (зависит от конкретного алгоритма. Для симплекс-метода экспоненциальна). Алгоритм является псевдополиномиальным и имеет оценку $O(nm \log U)$, где $m = |E|$, $n = |V|$, $U = \max(C_{ij})$. Идея алгоритма заключается в том, что изначально величине потока присваивается значение 0, т.е. $f(u, v) = 0$ для всех $u, v \in V$. Затем величина потока итеративно увеличивается посредством поиска увеличивающего пути (путь от источника s к стоку t , вдоль которого можно послать больший поток). Процесс повторяется, пока можно найти увеличивающий путь.

```
1 Задаем начальное значение потока f равным 0
2 while существует увеличивающий путь p
3   do увеличиваем поток f вдоль пути p
4 return f
```

Листинг 1. Псевдокод алгоритма Форда-Фалкерсона

1.2.1. Описание алгоритма

Дан граф $G(V, E)$ с пропускной способностью $c(u, v)$ и потоком $f(u, v) = 0$ для ребер из u в v . Необходимо найти максимальный поток из источника s в сток t . На каждом шаге алгоритма действуют три условия:

1. $f(u, v) \leq c(u, v)$ - поток из u в v не превосходит пропускной способности
2. $f(u, v) = -f(v, u)$
3. $\sum_v f(u, v) = 0 \Leftrightarrow f_{in}(u) = f_{out}(u)$ для всех узлов u , кроме s и t . Поток не изменяется при прохождении через узел

Остаточная сеть:
 $G_f(V, E_f)$ — сеть с пропускной способностью $c_f(u, v) = c(u, v) - f(u, v)$ и без потока.

На вход алгоритма подаётся граф G с пропускной способностью c , источник s и сток t .
В итоге на выходе максимальный поток f из s в t .

```

1  $f(u, v) := 0$  для всех ребер  $(u, v)$ 
2 while (существует путь  $p$  из  $s$  в  $t$  в остаточной сети  $G_f$ )
3   do { 1. Найти  $cf(p) = \min\{cf(u, v) \mid (u, v) \text{ принадлежит } p\}$ 
4         2. for каждого ребра  $(u, v)$  in  $p$ 
5           сделайте { 1.  $f(u, v) := f(u, v) + cf(p)$ 
6                       2.  $f(v, u) := f(v, u) - cf(p)$  }
7   }

```

Листинг 2. Расширенный псевдокод алгоритма Форда-Фалкерсона

Строка 1 инициализируют поток f значением 0. В цикле *while* в строках 2-7 выполняется неоднократный поиск увеличивающегося пути p в G_f , и поток f вдоль пути p увеличивается на остаточную пропускную способность $c_f(p)$. Когда увеличивающихся путей больше нет, алгоритм выдает нам максимальный поток f . На рисунке 1 показаны результаты каждой итерации при тестовом выполнении. Остаточная сеть на рисунке 1а - это исходная сеть G ; поток f , показанный на рисунке 1.д, является максимальным потоком.

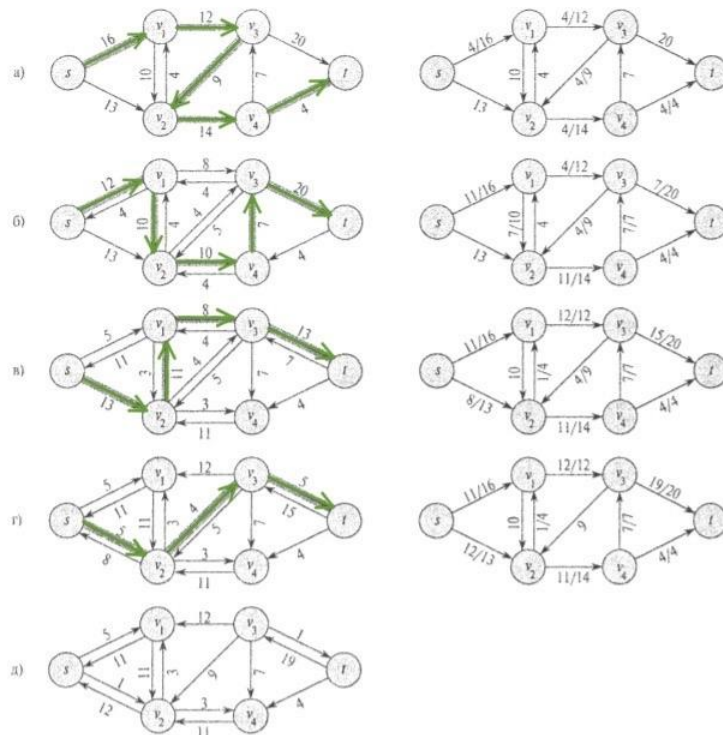


Рисунок 1. Работа базового алгоритма Форда-Фалкерсона. В левой части каждого рисунка показана остаточная сеть G_f с выделенным увеличивающимся путем p ; в правой части показан поток f , который получается в результате прибавления f_p к f .

1.2.2. Анализ метода Форда-Фалкерсона

Время выполнения алгоритма Форда-Фалкерсона зависит от того как именно ищется путь p . Он может быть найден с помощью поиска в ширину, тогда алгоритм выполнит свою работу за полиномиальное время, или поиска в глубину. Если выбрать неудачный метод поиска пути, то метод Форда-Фалкерсона может работать бесконечно: величина потока будет последовательно увеличиваться, но она не придет к выводу максимального значения потока. Из этого следует, что этот алгоритм гарантированно будет работать с целочисленными пропускными способностями. Если же пропускные способности - это рациональные числа, можно использовать округление, которое приведет их к целочисленным. Тогда первая строка алгоритма будет занимать $\Theta(E)$. Работа строк 1-7 будет выполнена не более чем $|f^*|$ раз, так как величина потока за каждую итерацию увеличивается как минимум на один. Таким образом, каждая итерация цикла *while* занимает $O(E)$, где E - число ребер в графе, следовательно, общее время на выполнение алгоритма составляет - $O(E|f^*|)$.

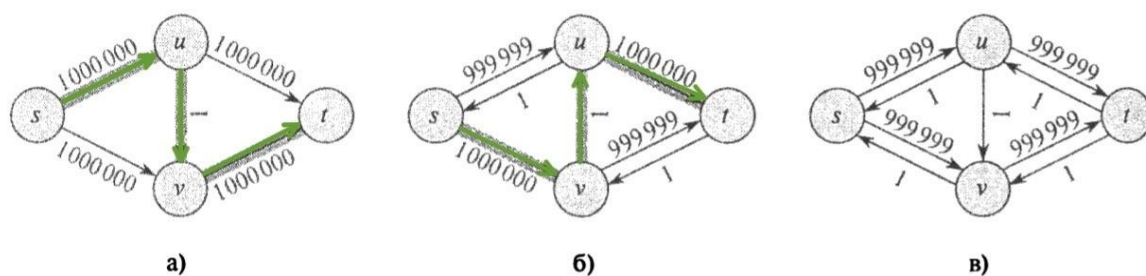


Рисунок 2. Транспортная сеть, для которой выполнение алгоритма Форда-Фалкерсона может занимать время $O(E|f^*|)$, $|f^*| = 2\,000\,000$

На рисунке 2 показано, что может произойти в простой транспортной сети, с большим значением $|f^*|$. Если рассмотреть эту задачу, то можно увидеть путь $s \rightarrow u \rightarrow v \rightarrow t$, показанный на рисунке 2, следовательно, поток после первой итерации будет иметь значение 1. Полученная остаточная сеть показана на рисунке 2б. Если в ходе второй итерации будет найден путь, показанный на рисунке 2б, поток станет равным 2. На рисунке 2в показана соответствующая остаточная сеть. Можно продолжать процедуру, выбирая увеличивающийся путь $s \rightarrow u \rightarrow v \rightarrow t$ для итераций с нечетным номером и $s \rightarrow v \rightarrow u \rightarrow t$ для итераций с четным номером. В ходе реализации этого алгоритма нам придется выполнить 2 000 000 итераций, при этом величина потока будет увеличиваться только на 1.

В ходе анализа алгоритма Форда-Фалкерсона было выяснено, что время выполнения напрямую зависит от того, как именно выполняется поиск пути p . Также был найден существенный минус данного алгоритма: в простой транспортной сети, с большим значением $|f^*|$ может происходить огромное количество итераций и при этом величина потока будет увеличиваться не существенно.

1.3. Алгоритм Эдмондса-Карпа

Данный алгоритм является частным случаем алгоритма Форда-Фалкерсона и решает его главный недостаток, то есть можно реализовать вычисление увеличивающегося пути p в строке 2 (листинг 2) как поиск в ширину.

1.3.1. Описание алгоритма

1. Обнуляем все потоки. Остаточная сеть изначально совпадает с исходной сетью.
2. В остаточной сети находим кратчайший путь из источника в сток. Если такого пути нет, останавливаемся.
3. Пускаем через найденный увеличивающий путь максимально возможный поток:
 - 3.1. На найденном пути в остаточной сети ищем ребро с минимальной пропускной способностью c_{min} .
 - 3.2. Для каждого ребра на найденном пути увеличиваем поток на c_{min} , а в противоположном ему - уменьшаем на c_{min} .
 - 3.3. Модифицируем остаточную сеть. Для всех ребер на найденном пути, а также для противоположных им ребер, вычисляем новую пропускную способность. Если она стала ненулевой, добавляем ребро к остаточной сети, а если обнулилась, стираем его.
4. Возвращаемся на шаг 2.

Реализация поиска в ширину:

1. Создаем очередь вершин O . Вначале O состоит из единственной вершины s .
2. Отмечаем вершину s как посещенную, без родителя, а все остальные как непосещенные.
3. Пока очередь не пуста, выполняем следующие шаги:
 - 3.1. Удаляем первую в очереди вершину u .
 - 3.2. Для всех дуг (u, v) , исходящих из вершины u , для которых вершина v еще не посещена, выполняем следующие шаги:
 - 3.2.1. Отмечаем вершину v как посещенную, с родителями u .
 - 3.2.2. Добавляем вершину v в конец очереди.
 - 3.2.3. Если $v = t$, выходим из обоих циклов: мы нашли кратчайший путь.
4. Если очередь пуста, возвращаем ответ, что пути нет вообще и останавливаемся.
5. Если нет, идем от t к s , каждый раз переходя к родителю. Возвращаем путь в обратном порядке.

1.3.2. Анализ метода Эдмондса-Карпа

Анализ зависит от расстояний между вершинами остаточной сети G_f .

Лемма 1. Если для некоторой транспортной сети $G = (V, E)$ с источником s и стоком t выполняется алгоритм Эдмондса-Карпа, то для всех вершин $v \in V - \{s, t\}$ длина кратчайшего пути $\delta_f(s, v)$ в остаточной сети G_f монотонно возрастает с каждым увеличением потока.

Теорема 1. Если для некоторой транспортной сети $G = (V, E)$ с источником s и стоком t выполняется алгоритм Эдмондса-Карпа, то общее число увеличений потока, выполняемое данным алгоритмом, составляет $O(VE)$.

Данная теорема устанавливает верхний предел количества итераций алгоритма Эдмондса-Карпа.

В ходе анализа данного алгоритма было выяснено, что если увеличивающийся путь находится посредством поиска в ширину, тогда каждую итерацию метода Форда-Фалкерсона можно выполнить за время $O(E)$, следовательно, общее время выполнения алгоритма Эдмондса-Карпа составляет $O(VE^2)$.

1.4. Алгоритм Диница

Данный алгоритм был опубликован в 1970 году Ефимом Диницем. Основная идея метода – алгоритм состоит из фаз, на которых поток увеличивается сразу вдоль всех кратчайших цепей определенной длины. Для этого на i -той фазе строится вспомогательная бесконтурная сеть (layered network). Эта сеть содержит все увеличивающие цепи, длина которых не превышает k_i , где k_i – длина кратчайшего пути из s в t . Величину k_i называют длиной вспомогательной сети.

1.4.1. Описание алгоритма

Пусть $G = ((V, E), c, s, t)$ – транспортная сеть, в которой $c(u, v)$ и $f(u, v)$ – соответственно пропускная способность и поток через ребро (u, v) . Остаточная пропускная способность – отображение $c_f: V \times V \rightarrow R^+$ определенное как:

1. Если $(u, v) \in E$,

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

$$c_f(u, v) = f(u, v)$$
2. $c_f(u, v) = 0$ иначе.

Остаточная сеть – граф $G_f = ((V, E_f), c_f|_{E_f}, s, t)$, где
 $E_f = \{(u, v) \in V \times V | c_f(u, v) > 0\}$.

Дополняющий путь – s – t путь в остаточном графе G_f .

Пусть $dist(v)$ – длина кратчайшего пути из s в t в графе G_f . Тогда вспомогательная сеть графа G_f – граф $G_L = (V, E_L, c_f|_{E_L}, s, t)$, где $E_L = \{(u, v) \in E_f | dist(v) = dist(u) + 1\}$.

Блокирующий поток – s – t поток f такой, что граф $G^* = (V, E_L^*, s, t)$ с $E_L^* = \{(u, v) | f(u, v) < c_f|_{E_L}(u, v)\}$ не содержит s – t пути [2].

```

1 // работа алгоритма в i-ой фазе
2 while ...{
3     найдем L(s, t) в Gf, с помощью поиска в ширину
4     if L(s, t) - пустой return f
5     else L ← L(s, t);
6     // цикл итерации
7     while L не пустой{
8         //инвариант итерации: L is in union of all shortest augmenting path
9         P ← PathFinding(L);
10        Sat ← FlowChange(P);
11        // Cleaning(L){
12            Removal of edges in Sat;
13            Qr, Ql ← Sat;
14            RightPass(Qr);
15            LeftPass(Ql);
16        }
17    }
18 }

```

Листинг 3. Псевдокод алгоритма Диница

Рассмотрим работу алгоритма Диница в i-ой фазе (Листинг 3).

1. С помощью поиска в ширину движемся из источника сети в сток по доступным дугам, добавляем их в S_k и увеличивая k . Дуга u добавляется с $c_k(u) = c(u) - f(u)$. Как только достигнут сток сети, он отмечается величиной k и она становится фиксированной величиной. Далее поиск в ширину производится уже не из вершин v , для которых $q(s, v) \geq k$. Таким образом, вспомогательная сеть будет содержать подсеть исходной. Если поиск не достиг стока, то работа алгоритма прекращается.
 2. В построенной бесконтурной сети длины k ищется псевдо максимальный поток(поток f , для которого не существует увеличивающих цепей длины k).
 - 2.1. Для построения данного потока используется поиск в ширину(с ограничением на длину пути). Пусть на j-ой итерации был найден путь из s - t . Пустим по этому пути поток f_j . Это означает, что как минимум одна дуга вспомогательной сети является насыщенной. Удаляем все насыщенные дуги. В результате могут образоваться вершины, из которых не выходит\не входит ни одна дуга или они изолированы. Их также следует удалить со всеми инцидентными им дугами. Корректировка производится до тех пор, пока в вспомогательной сети не останется ни одного тупика. Изменяем пропускные способности оставшихся дуг по формуле: $c_k(u) = c_k(u) - f_j(u)$. Поиск f_j следует продолжать до тех пор, пока вспомогательная сеть $\neq \emptyset$.
 3. Найденный поток переносится в исходную сеть. Переходим к пункту 1.
- После завершения работы Алгоритма исходная сеть будет содержать максимальный поток[2].

1.4.2. Анализ алгоритма Диница

Пусть вся многоуровневая сеть $L = L(s, t)$ размера $O(|E|)$ исчезает после одной итерации. Например, если первый пограничный слой состоит только из одного ребра из s минимальной емкости, то первая выполненная итерация насыщает это ребро и, таким образом, отключает t от s . Используем анализ для определения общей стоимости всех очисток в течение одного этапа. Отнесем стоимость всех соответствующих операций на удаленные элементы L следующим образом. Удаленное ребро оплачивает операции, примененные к нему, в общей сложности $O(l)$ и за проверку двух его конечных вершин; проверка того, является ли список пустым или нет, также стоит $O(l)$. Следовательно, общие затраты во время фазы составляют $O(|E| + |V|)$, что равно $O(|E|)$, так как L подключен. Построение многоуровневой сети при инициализации фазы стоит $O(|E|)$. Следовательно, общая стоимость всех операций со структурой данных многоуровневой сети, за исключением поиска пути, составляет $O(|E|)$ за фазу. Общая стоимость ускоренной итерации равна $O(l) = O(|V|)$, где l - длина многоуровневой сети. Обозначим общее количество итераций через $\#it$ и количество фаз через $\#ph$ (в общем случае оно может быть бесконечным). Затем общее время работы ограничено как $O(\#it|V| + \#ph|E|)$. Во-первых, это лучше, чем граница $O(\#it|E|)$. Действительно, намного быстрее, чем алгоритм Форда-Фалкерсона, даже если алгоритм Форда-Фалкерсона использует только самые короткие пути увеличения. Во-вторых, существенная структура алгоритма Форда-Фалкерсона при ускорении остается точно такой же, поэтому ни доказательства конечности, ни оценки числа итераций, когда-либо доказанных для алгоритма Форда-Фалкерсона, не страдают от ускорения. Пусть метод поддержания структуры данных будет идеальным, если перед каждой итерацией структура данных как бы строится с нуля на основе текущих данных. После любой итерации на этапе с многоуровневой сетью длины l обновленная многоуровневая сеть представляет собой объединение всех дополнительных путей длины l , в то время как более короткого пути увеличения не существует (т. е. текущий поток). Следствием является то, что ни один край, насыщенный в течение определенной фазы, не может быть ненасыщенным позже в течение той же фазы. Таким образом, ни одно ребро, удаленное из G_f , не может быть восстановлено на нем в течение той же фазы. Также не существует старых путей длиной меньше E . Имеется не более $|V| - 1$ фаз, поскольку расстояние от s до t не может превышать $|V| - 1$. Легко видеть, что количество итераций в течение одной фазы составляет не более $|E|$, так как по крайней мере одно из не более $|E|$ ребер L удаляется из него на каждой итерации. Следовательно, общее время выполнения на каждом этапе, которое состоит из ускоренного времени итерации и времени построения и обслуживания многоуровневой сети, равно $O(|E| \cdot |V| + |E|) = O(|V||E|)$. Таким образом, конечен, и его общее время работы равно $O(|V|^2|E|)$. Мы приходим к нашему основному результату: алгоритм Диница строит поток максимума за время $O(|V|^2|E|)$.

Давайте подведем итог тому, что установлено о поведении алгоритма Диница - он состоит из фаз. Каждый из них содержит итерации, изменяющие поток с

использованием кратчайших путей расширения фиксированной длины, т.е. в начале каждой фазы расширенная BFS создает за $O(|E|)$ время многоуровневую сетевую структуру данных L , длиной E . Многоуровневая сеть постоянно поддерживается в течение фазы как объединение всех кратчайших увеличивающихся путей длины l , пока она не исчезнет за общее время $O(|E|)$. Многоуровневая структура L и отсутствие в ней тупиков позволяют выполнять каждую итерацию алгоритм Форда-Фалкерсона за $O(t) = O(|V|)$ время. Многоуровневая сеть строго обрезается после каждой итерации алгоритма Форда-Фалкерсона. Поэтому количество итераций на каждом этапе ограничено $|E|$. Когда многоуровневая сеть исчезает, не существует пути увеличения длины, меньшей или равной l , т.е. текущему потоку. Следовательно, длина следующей многоуровневой сети, равная длине кратчайшего в настоящее время пути расширения, строго больше, чем $t-s$. Поскольку длина L растет от фазы к фазе, существует не более $|V|$ в 1 фазе. Когда выполнение работы алгоритма останавливается, текущий поток максимален. Время работы равно $O(|V|^2|E|)$ [2].

1.5. Алгоритм Слейтера-Тарьяна

Слейтер и Тарьян в 1981 г. разработали на основе деревьев, использованных в алгоритме Галила-Наамада, улучшенную структуру данных. Их динамические деревья позволяют производить операции слияния (*link*) и разделения (*cut*) за время $O(\log n)$. Модификации деревьев Слейтера-Тарьяна используются для повышения оценки быстродействия практически во всех современных алгоритмах. Основная идея метода – алгоритм состоит из фаз, на которых поток увеличивается сразу вдоль всех кратчайших цепей определенной длины. Для этого на i -той фазе строится вспомогательная бесконтурная сеть (*layered network*). Эта сеть содержит все увеличивающиеся цепи, длина которых не превышает k_i , где k_i – длина кратчайшего пути из s в t . Величину k_i называют длиной вспомогательной сети[4].

1.5.1. Описание алгоритма

Для хранения информации о различных путях в сети и быстрого изменения этой информации, предлагается воспользоваться лесом деревьев. Одна и та же вершина сети не может одновременно содержаться в двух деревьях. Деревья допускают над собой два вида операций – слияния (*link*) и разделения (*cut*).

Link(v, w) – соединяет две вершины из различных деревьев, добавляя дугу (v, w). В результате 2 дерева сливаются в одно.

Cut(v, w) – Разделяет дерево, содержащее дугу (v, w) на два, с помощью удаления дуги (v, w). Вершины v и w должны быть смежными вершинами одного дерева.

Следует различать корневые и свободные деревья. В случае корневых деревьев операция *Link*(v, w) разрешена, только если v – корень дерева. Результатом слияния будет дерево с корнем, равным корню дерева, которому принадлежала вершина w . Предполагается, что дуги корневого дерева ориентированы по направлению к корню, т.е. от потомка к родительскому узлу. Заявленная оценка быстродействия справедлива как для корневых, так и для свободных деревьев.

Для применения подобной структуры к сетевым задачам, вводятся также 5 дополнительных операций:

Root(v) – возвращает корень дерева, содержащего v .

Cost(v, w) – возвращает “стоимость” (вес) дуги (v, w), т.е. вещественной число.

Find_min(v) – возвращает дугу минимальной стоимости на пути из v к корню дерева, в котором находится v . Если этот путь не содержит дуг, возвращается специальное *null* значение. Если дуг минимальной стоимости несколько, результатом будет ближайшая к корню дуга.

Update(v, x) – увеличивает стоимость всех дуг пути из v к корню дерева, в котором находится v , на x . x – вещественное число.

Evert(v) - изменяет дерево, содержащее вершину v , делая v корнем этого дерева.

Операция *Link* модифицируется для трех параметров: *Link*(v, w, x) – отличие лишь в том, что добавляемой дуге (v, w) присваивается стоимость x .

Первых 6 операций достаточно для применения деревьев Слейтера-Тарьяна к задаче о поиске тупикового потока. Операция Evert необходима лишь в случае использования свободных деревьев.

Нахождение тупикового потока с помощью динамических деревьев.

Инициализация. Помещаем в нашу структуру все вершины сети (без дуг), получим множество “деревьев”, состоящих из одной изолированной вершины.

1. Пусть $v = \text{root}(s)$. Если $v = t$, перейдем к шагу 4 иначе к шагу 2.
2. ($v \neq t$). Если из вершины v не исходит ни одной дуги, перейдем к шагу 3. Иначе, выбираем дугу (v, w) , выполняем $\text{Link}(v, w, c(v, w))$ и переходим к шагу 1.
3. (все пути из v в t блокированы). Если $v = s$, прерываем работу алгоритма. Иначе, для всех входящих в v дуг (содержащихся в этом дереве) выполняем $\text{cut}(u, v)$, и переходим к Шагу 1.
4. (s и t в одном дереве, $t = \text{root}(s)$, путь из s в t – увеличивающий поток путь). Пусть $(v, w) = \text{Find_min}(s)$. (нашли дугу минимальной пропускной способности). Пусть $c_{\min} = \text{cost}(v, w)$. Выполняем $\text{Update}(s, -c_{\min})$ и переходим к шагу 5.
5. (удаление дуг с нулевой пропускной способностью). Пусть $(v, w) = \text{Find_min}(s)$. Если $\text{cost}(v, w) = 0$, то $\text{cut}(c, w)$ и выполняем шаг 5 заново. Иначе переходим к шагу 1.

После окончания работы цикла, максимальный поток определяется разницей пропускных способностей дуг до и после выполнения алгоритма.

1.5.2. Анализ алгоритма Слейтера-Тарьяна

Результат будет получен за $O(nm \log n)$.

Деревья Слейтера-Тарьяна (и их модификации) применяются не только для нахождения тупиковых потоков, они используются в задаче нахождения наиболее удаленного от корня общего предка для двух заданных вершин, задаче нахождения поддеревьев минимальной стоимости при различных условиях (для решения используются свободные деревья) и в симплекс методе решения сетевых задач.

1.6. Алгоритм проталкивания предпотока

Данный алгоритм решает задачу нахождения максимального потока в транспортной сети. Алгоритм не является частным случаем алгоритма Форда-Фалкерсона. Был опубликован в 1986 г. Гольдбергом и Тарьяном. В основе алгоритма лежат 2 операции: наращивание потока (*Push*) и изменение метки вершины (*Relabel*), в связи с этим, метод примененный в алгоритме получил название *Push-Relabel*. Данный метод внес большой вклад в исследование проблемы максимального потока. Многие современные алгоритмы основаны на *Push-Relabel* методе. Алгоритмы проталкивания предпотока работают более локальным способом, чем метод Форда-Фалкерсона. Главное отличие от других алгоритмов это то, что обрабатываются вершины по одной, рассматривая только соседей данной вершины в остаточной сети, а также не обеспечивают в ходе своего выполнения свойство сохранения потоков.

1.6.1. Описание алгоритма

В алгоритме проталкивания предпотока выполняются две основные операции: проталкивание избытка потока от вершины к одной из соседних с ней вершин и подъем вершины. Применение этих операций зависит от высот вершин.

Пусть $G = (V, E)$ - транспортная сеть с источником s и стоком t , а f - некоторые предпоток G . Функция $h: V \rightarrow \mathbb{N}$ является функцией высоты, если $h(s) = |V|$, $h(t) = 0$ и $h(u) \leq h(v) + 1$ для любого остаточного ребра $(u, v) \in E_f$.

Основная операция $Push(u, v)$ может применяться тогда, когда u является переполненной вершиной, $c_f(u, v) > 0$ и $h(u) = h(v) + 1$.

1.6.2. Анализ алгоритма проталкивания предпотока

При каждой разрядке без подъема смещается на одну позицию вправо. Список L содержит $V-2$ вершин, следовательно, больше $V-2$ подряд разрядок выполнить нельзя. Тогда количество подъемов $O(V^2)$, а количество разрядок $O(V^3)$. Время выполнения подъемов составляет $O(VE)$. Общее время проталкиваний составляет $O(V^3)$. Таким образом суммарное время выполнения алгоритма составляет $O(V^3 + VE)$.

1.7. Сравнение алгоритмов

Таким образом если резюмировать данные, полученные в ходе анализа, можно сделать следующие выводы: почти каждый алгоритм это улучшенная версия алгоритма Форда-Фалкерсона. Также все они находят решение задачи о максимальном потоке, где дана некоторая транспортная сеть G с источником s и стоком t , и необходимо найти поток максимальной величины. “Алгоритм Слейтера-Тарьяна” имеет одну главную особенность - это использование динамических деревьев, благодаря которым алгоритм работает преимущественно быстрее других. Главным отличием алгоритма “Проталкивания предпотока” от других алгоритмов это то, что обрабатываются вершины по одной, рассматривая только соседей данной вершины в остаточной сети, а также не обеспечивают в ходе своего выполнения свойство сохранения потоков. Сделан вывод о том, что использовать алгоритм Форда-Фалкерсона в программах не рекомендуется, так как он может очень долго выполнять свои итерации.

Глава 2. Практическая реализация алгоритмов решения задачи о максимальном потоке

2.1. Выбор структуры данных для хранения информации о вершинах и ребрах графа

Задача о максимальном потоке связана с решением задач с транспортными сетями, их, в свою очередь, проще всего представить в виде графов. Граф – совокупность точек, соединенных линиями. Точки называются вершинами, или узлами, а линии – ребрами, или дугами. Следовательно, при выборе структуры данных необходимо учитывать то, что нам необходимо будет хранить две связанные между собой переменные (узел и вес ребра). Под данный критерий подходит два варианта структуры – это массив (хранить граф в виде матрицы смежности) или реализовать собственный класс “Граф”, где будет храниться вся необходимая информация. За основу данного класса можно взять реализацию структуры деревьев. Дерево — одна из наиболее широко распространенных структур данных в информатике, эмулирующая древовидную структуру в виде набора связанных узлов. Является связным графом, не содержащим циклы. Реализацию данной структуры можно посмотреть в приложении 1.

2.2. Особенности реализации алгоритмов решения задачи о максимальном потоке

Для реализации данных алгоритмов потребовалось написать дополнительные алгоритмы, такие как поиск в ширину и в глубину. Поиск в ширину — один из простейших алгоритмов обхода графа, являющийся основой для многих важных алгоритмов для работы с графами, например, для решения задачи о максимальном потоке. Данный алгоритм реализован в алгоритме Эдмондса-Карпа для решения проблемы: выполнения огромного количества итерации в простой транспортной сети, с большим значением $|f^*|$. Далее был реализован алгоритм поиска в глубину. Поиск в глубину — один из методов обхода графа. Стратегия поиска в глубину состоит в том, чтобы идти «вглубь» графа, насколько это возможно. Данный алгоритм реализуется рекурсивно для перебора всех исходящих из рассматриваемой вершины ребра. Главная особенность реализации алгоритма Диница это то, что потребовалось переписать вышеописанные алгоритмы, так как он требует особенного выполнения этих алгоритмов. Псевдокод для данных алгоритмов был приведен в **1 главе**. Реализация кода в приложении 1.

2.3. Результаты запуска кода

Для проверки верности работы алгоритмов были написаны вводные данные. И рассчитан их максимальный поток. Как говорилось выше работа алгоритма Форда-Фалкерсона подходила не для всех тестовых значений, далее это будет проиллюстрировано на различных тестах. Перед тем как писать тесты необходимо знать, как правильно вводить значения. В первой строке данных вводятся четыре параметра. Первый это количество вершин в графе, второй это количество ребер, третий параметр - это исток, а четвертый сток. Далее на каждой новой строке(количество равно количеству ребер) описываются ребра. Третий параметр в данном случае - это вес ребра. Первый и второй это координаты вершин, между которыми лежит данное ребро.

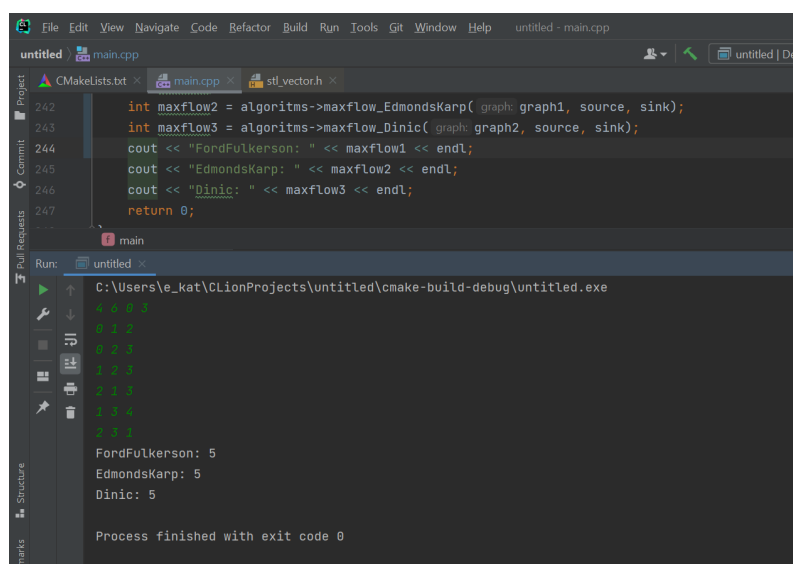
Итак, первый тест.

```
4 6 0 3
0 1 2
0 2 3
1 2 3
2 1 3
1 3 4
2 3 1
```

Тест 1.

Вводные значения: имеем граф на четырех вершинах с шестью ребрами. Необходимо найти максимальный поток из нулевой вершины в третью.

Вывод: максимальный поток при тесте 1 должен равняться пяти.



The screenshot shows a C++ IDE with a file named 'main.cpp' open. The code defines two graphs, 'graph1' and 'graph2', and calculates their maximum flow using three algorithms: FordFulkerson, EdmondsKarp, and Dinic. The output of the program is displayed in the 'Run' window, showing the maximum flow for each graph and algorithm.

```
int maxflow2 = algorithms->maxflow_EdmondsKarp( graph1, source, sink);
int maxflow3 = algorithms->maxflow_Dinic( graph2, source, sink);
cout << "FordFulkerson: " << maxflow1 << endl;
cout << "EdmondsKarp: " << maxflow2 << endl;
cout << "Dinic: " << maxflow3 << endl;
return 0;
```

Run: untitled x
C:\Users\le_kat\CLionProjects\untitled\cmake-build-debug\untitled.exe

```
4 6 0 3
0 1 2
0 2 3
1 2 3
2 1 3
1 3 4
2 3 1

FordFulkerson: 5
EdmondsKarp: 5
Dinic: 5

Process finished with exit code 0
```

Рисунок 1. Результат работы алгоритмов при тесте 1.

На рисунке 1 показаны результаты выполнения алгоритмов. В данном случае алгоритм Форда-Фалкерсона справился с вводимыми данными.

Посмотрим на работу алгоритмов с большими данными.

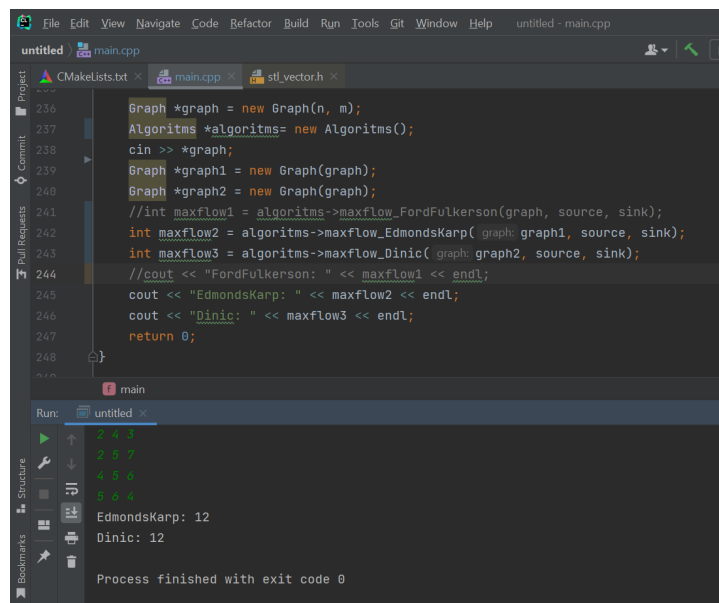
```
7 11 0 6
0 1 5
0 2 10
0 4 4
1 3 1
1 6 3
2 3 7
3 6 5
2 4 3
2 5 7
4 5 6
5 6 4
```

Тест 2.

Вводные значения: имеем граф на семи вершинах с одиннадцатью ребрами.

Необходимо найти максимальный поток из нулевой вершины в шестую.

Вывод: максимальный поток при тесте 2 должен равняться 12.



The screenshot shows a C++ IDE with a file named 'main.cpp'. The code defines a 'Graph' class and an 'Algorithms' class. It creates a graph with 7 vertices and 11 edges, then uses three different algorithms to find the maximum flow: Ford-Fulkerson, Edmonds-Karp, and Dinic. The output shows that Edmonds-Karp and Dinic both return 12, while Ford-Fulkerson is commented out. The console output at the bottom confirms the results: 'EdmondsKarp: 12' and 'Dinic: 12'.

```
236 Graph *graph = new Graph(n, m);
237 Algorithms *algorithms = new Algorithms();
238 cin >> *graph;
239 Graph *graph1 = new Graph(graph);
240 Graph *graph2 = new Graph(graph);
241 //int maxflow1 = algorithms->maxflow_FordFulkerson(graph, source, sink);
242 int maxflow2 = algorithms->maxflow_EdmondsKarp( graph, graph1, source, sink);
243 int maxflow3 = algorithms->maxflow_Dinic( graph, graph2, source, sink);
244 //cout << "FordFulkerson: " << maxflow1 << endl;
245 cout << "EdmondsKarp: " << maxflow2 << endl;
246 cout << "Dinic: " << maxflow3 << endl;
247 return 0;
248 }
```

Run: untitled x

EdmondsKarp: 12
Dinic: 12

Process finished with exit code 0

Рисунок 2. Результат работы алгоритмов при тесте 2.

Как можно заметить на рисунке 2 закомментированы вызовы функции алгоритма Форда-Фалкерсона. Это было сделано из-за того, что при введении теста 2 “Форд-Фалкерсон” начинает производить много итерации из-за которых выполнение программы сводится к минимуму. Два других алгоритма вывели верный результат. Это говорит нам о том, что алгоритмы были реализованы верно.

Заключение

Данные алгоритмы были реализованы в среде разработки CLion 2021.3. Эти алгоритмы широко применяются в различных программах. Задача о максимальном потоке изучается уже более 60 лет. Интерес к ней подогревается огромной практической значимостью этой проблемы. Методы решения задачи применяются на транспортных, коммуникационных, электрических сетях, при моделировании различных процессов физики и химии, в некоторых операциях над матрицами, для решения родственных задач теории графов, и даже для поиска Web-групп в WWW. Исследования данной задачи проводятся во множестве крупнейших университетов мира. Данные алгоритмы затрачивают по оценке : $O(nmU)$, $O(nm^2)$, $O(n^2m)$, что является не самым лучшим показателем решения данной задачи так , например, алгоритм Слейтора-Тарьяна составляет оценку $O(\log n)$. В заключении хотелось бы добавить, что данная задача о максимальном потоке является очень востребованной и необходимо в жизни людей.

Список литературы

1. Алгоритмы. Построение и анализ. / Т. Кормен [и др.]. – Москва : Издательский дом "Вильямс", 2005. – 734 с.
2. Dinitz, Yefim Dinitz' Algorithm: The Original Version and Even's Version / Yefim Dinitz. – Conference Paper, 2006. – 218-240 с.
3. В. Н. Korte, Jens Vygen : Blocking Flows and Fujishige Algorithm — Springer Berlin Heidelberg , 2008. - 174 -176 с.
4. Sleator D. D., Tarjan R. E. A Data Structure for Dynamic Trees. J. Comput. System Sci.,26:362-391, 1983.
5. Goldberg A. V., Tarjan R. E. A New Approach to the Maximum Flow Problem. J. Assoc. Comput. Mach., 35:921-940, 1988

Приложение

Приложение 1

```
#include <iostream>
#include <vector>
#include <set>
#include <algorithm>
#include <queue>
#include <fstream>

using namespace std;

using std::vector;

class Graph {
private:
    int numPoints, vertex;
    vector<int> head, nxt, to, capacity;
    int tot;
public:
    Graph(int n, int m) {
        numPoints = n;
        vertex = m;
        tot = 1;
        head.resize(n);
        nxt.resize(2 * m + 2, 0);
        to.resize(2 * m + 2);
        capacity.resize(2 * m + 2);
    }

    Graph(Graph *graph) {
        numPoints = graph->numPoints;
        head = graph->head;
        nxt = graph->nxt;
        to = graph->to;
        capacity = graph->capacity;
        tot = graph->tot;
    }

    ~Graph() {}

    void add(int x, int y, double weight) {
        nxt[++tot] = head[x], head[x] = tot, to[tot] = y, capacity[tot] = weight;
        swap(x, y);
        nxt[++tot] = head[x], head[x] = tot, to[tot] = y, capacity[tot] = 0;
    }
}
```

```

friend class Algorithms;

friend ostream &operator<<(ostream &s, Graph graph); //вывод
friend istream &operator>>(istream &s, Graph &graph); //ввод

};

ostream& operator<<(ostream& s, Graph graph){

    if (typeid(s) == typeid(ofstream)){ // работа с файлами
        // сохраняем размерности
        s << graph.numPoints << " " << graph.vertex << " ";
        // сохраняем элементы матрицы
        for (int j = 0; j < graph.vertex; j++)
            s << graph.capacity[j] << " \n";

    }else{
        // вывод в консоль
        int i, j;
        for (j = 0; j < graph.vertex; j++)
            s << graph.capacity[j] << " \n";
    }
    return s;
}

istream &operator>>(istream &s, Graph &graph) {

    if (typeid(s) == typeid(ifstream)) { //ввод из файла

        // читаем размерности
        int n, m, source, sink;
        cin >> n >> m >> source >> sink;

        // чтение элементов
        for (int i = 0; i < graph.vertex; i++) {
            int a, b, capacity;
            cin >> a >> b >> capacity;
            graph.add(a, b, capacity);
        }
    } else {
        // ввод из консоли
        for (int i = 0; i < graph.vertex; i++) {
            int a, b, capacity;
            cin >> a >> b >> capacity;
            graph.add(a, b, capacity);
        }
    }
    return s;
}

```

```

#define INF 0x7fffffff
using std::queue;
using std::pair;
using std::make_pair;

class Algorithms{
private:
    int DFS(Graph *graph, int s, int t, int flow);

    int BFS(Graph *graph, int s, int t, pair<int, int> *ptr, int *flow);

    int DFS_Dinic(Graph *graph, int s, int t, int flow, int *use);

    bool BFS_Dinic(Graph *graph, int s, int t, int *use);

    vector<bool> used;
public:

    int maxflow_FordFulkerson(Graph *graph, int source, int sink) {
        int maxflow = 0, augmentation;
        used.resize(graph->numPoints, false);
        do {
            used.resize(used.size(), false);
            augmentation = DFS(graph, source, sink, INF);
            maxflow += augmentation;
        } while (augmentation > 0);

        return maxflow;
    }

    int maxflow_EdmondsKarp(Graph *graph, int source, int sink) {
        int maxflow = 0, augmentation;
        int flow[graph->numPoints];
        pair<int, int> ptr[graph->numPoints];

        while ((augmentation = BFS(graph, source, sink, ptr, flow)) != -1) {
            maxflow += augmentation;
            int t = sink;
            while (t != source) {
                graph->capacity[ptr[t].first] -= augmentation;
                graph->capacity[ptr[t].first ^ 1] += augmentation;
                t = ptr[t].second;
            }
        }
        return maxflow;
    }
}

```

```

int maxflow_Dinic(Graph *graph, int source, int sink) {
    int maxflow = 0, augmentation;
    int use[graph->numPoints];
    while (BFS_Dinic(graph, source, sink, use)) {
        while ((augmentation = DFS_Dinic(graph, source, sink, INF, use)) > 0) {
            maxflow += augmentation;
        }
    }
    return maxflow;
}

};

int Algorithms::DFS(Graph *graph, int s, int t, int flow) {
    if (s == t) return flow;
    for (int i = graph->head[s]; i; i = graph->nxt[i]) {
        if (graph->capacity[i] <= 0 && used[graph->to[i]]) continue;
        used[graph->to[i]] = true;
        int df = DFS(graph, graph->to[i], t, std::min(flow, graph->capacity[i]));
        if (df > 0) {
            graph->capacity[i] -= df;
            graph->capacity[i ^ 1] += df;
            return df;
        }
    }
    return 0;
}

int Algorithms::BFS(Graph *graph, int s, int t, pair<int, int> *ptr, int *flow) {
    for (int i = 0; i < graph->numPoints; i++) ptr[i].first = -1, ptr[i].second = -1;
    flow[s] = INF;
    queue<int> que;
    que.push(s);
    while (!que.empty()) {
        int x = que.front();
        que.pop();
        if (x == t) break;
        for (int i = graph->head[x]; i; i = graph->nxt[i]) {
            if (graph->to[i] != s && ptr[graph->to[i]].first == -1 && graph->capacity[i] > 0) {
                ptr[graph->to[i]].first = i;
                ptr[graph->to[i]].second = x;
                flow[graph->to[i]] = min(flow[x], graph->capacity[i]);
                que.push(graph->to[i]);
            }
        }
    }
    return ptr[t].second == -1 ? -1 : flow[t];
}

```

```

int Algorithms::DFS_Dinic(Graph *graph, int s, int t, int flow, int *use) {
    if (s == t) return flow;
    int curflow = 0;
    for (int i = graph->head[s]; i; i = graph->nxt[i]) {
        int y = graph->to[i];
        if (use[y] == use[s] + 1 && graph->capacity[i] > 0) {
            curflow = DFS_Dinic(graph, y, t, std::min(flow, graph->capacity[i]), use);
            if (curflow > 0) {
                graph->capacity[i] -= curflow;
                graph->capacity[i ^ 1] += curflow;
                return curflow;
            }
        }
    }
    return 0;
}

```

```

bool Algorithms::BFS_Dinic(Graph *graph, int s, int t, int *use) {
    for (int i = 0; i < graph->numPoints; i++) use[i] = -1;
    use[s] = 0;
    queue<int> que;
    que.push(s);
    while (!que.empty()) {
        int x = que.front();
        que.pop();
        for (int i = graph->head[x]; i; i = graph->nxt[i]) {
            int y = graph->to[i];
            if (use[y] == -1 && graph->capacity[i] > 0) {
                use[y] = use[x] + 1;
                que.push(y);
            }
        }
    }
    return (use[t] != -1);
}

```

```

int main() {
    int n, m, source, sink;
    cin >> n >> m >> source >> sink;

    Graph *graph = new Graph(n, m);
    Algorithms *algorithms = new Algorithms();
    cin >> *graph;
    Graph *graph1 = new Graph(graph);
    Graph *graph2 = new Graph(graph);
    int maxflow1 = algorithms->maxflow_FordFulkerson(graph, source, sink);
}

```

```
int maxflow2 = algorithms->maxflow_EdmondsKarp(graph1, source, sink);
int maxflow3 = algorithms->maxflow_Dinic(graph2, source, sink);
cout << "FordFulkerson: " << maxflow1 << endl;
cout << "EdmondsKarp: " << maxflow2 << endl;
cout << "Dinic: " << maxflow3 << endl;
return 0;
}
```