Осенние сборы в МФТИ, ноябрь 2015
конспект лекций

Собрано 17 ноября 2015 г. в 15:01

Содержание

1. Flows

1.1. Определения

Пусть дан ориентированный граф G из n вершин и m рёбер. Между каждыми двумя вершинами v и u графа G есть ребро с пропускной способностью $c_{v,u} \geq 0$. Если мы хотим, чтобы ребра не было, скажем $c_{v,u} = 0$.

Def 1.1.1. *Поток из s в t – функция $f_{v,u} \in \mathbb{R}$, обладающая тремя свойствами*

1. $\forall v, u \quad f_{v,u} \leq c_{v,u}$ *(течёт не больше пропускной способности)*
2. $\forall v, u \quad f_{v,u} = -f_{u,v}$ *(антисимметричность)*
3. $\forall v \neq s, t \quad \sum_u f_{v,u} = 0$ *(сохранение потока)*

При этом вершина s называется *истоком*, а вершина t *стоком*.
Иногда мы будем писать f_e и c_e – поток и пропускная способность для ребра e.

Def 1.1.2. *Разрез между s и t – разбиение множества вершин V: $V = S \sqcup T, s \in S, t \in T$. ($S \cap T = \varnothing$, исток лежит в S, сток лежит в T)*

Разрез будем обозначать (S, T).

Def 1.1.3. *Величина потока $|f| = \sum_v f_{s,v}$ (сумма вытекающего из истока s).*

Величина потока может получиться отрицательной. Определениям это не противоречит.

Def 1.1.4. *Величина разреза $C(S,T) = \sum\limits_{v \in S, u \in T} c_{v,u}$ (сумма пропускных способностей рёбер, проходящих через разрез).*

Def 1.1.5. *Обозначение $F(A,B) = \sum\limits_{v \in A, u \in B} f_{v,u}$*

Это лишь обозначение. A и B могут пересекаться или даже совпадать.

Def 1.1.6. *Максимальный поток – f: $|f| = \max$*

Def 1.1.7. *Минимальный разрез – (S,T): $C(S,T) = \min$*

Def 1.1.8. *Ребро насыщенно $\Leftrightarrow$ поток по ребру равен его пропускной способности*

Def 1.1.9. *Остаточная сеть G_f – граф из рёбер, для которых $c_{u,v} - f_{u,v} > 0$. Иногда в остаточной сети так же определяют пропускные способности рёбер: $c_{u,v} - f_{u,v}$*

Def 1.1.10. *Дополняющий путь – путь из s в t, для всех рёбер которого $f_{v,u} < c_{v,u}$. Иначе говоря "путь в остаточной сети"*

Def 1.1.11. *Циркуляция – поток размера ноль.*

Например, цикл является циркуляцией.

1.2. Теорема и алгоритм Форда-Фалкерсона

Lm 1.2.1. Величина потока равна потоку, протекающему через разрез:
$\forall$ разрез $(S,T), \forall$ поток $f \quad |f| = F(S,T)$

Доказательство. $|f| = F(\{s\},V) = F(S,V) = F(S,S) + F(S,T) = 0 + F(S,T) = F(S,T)$ ■

Lm 1.2.2. Поток не больше разреза: $\forall$ разрез (S,T), поток $f \quad |f| \le C(S,T)$

Доказательство. $|f| = F(S,T) \le C(S,T)$ ■

Lm 1.2.3. *О дополняющем пути.* Если есть дополняющий путь, поток не максимален.

Доказательство. Пусть есть дополняющий путь: увеличим поток по прямым рёбрам, уменьшим по обратным. Все 3 свойства потока выполняются, размер увеличился. ■

Алгоритм Форда-Фалкерсона.
Алгоритм ищет максимальный поток в графе с целочисленными пропускными способностями.

$f \leftarrow 0$
`while существует дополняющий путь:`
    `ищем его поиском в глубину по ненасыщенным рёбрам`
    `пусть путь:` $s = v_1, v_2, \dots, v_k = t$
    $x = \min_{i=1..k-1}\left[c_{v_{i+1},v_i} - f_{v_{i+1},v_i}\right]$ $(x > 0;$ $x =$ `сколько потока можно толкнуть по пути`$)$
    `увеличим поток по пути на` x
`return` f

Если $c_{v,u}$ целые, то $x \ge 1$, поэтому алгоритм работает работает за конечное время. Максимальность потока будет следовать из следующей теоремы:

Теорема 1.2.4. Форда-Фалкерсона. $\max|f| = \min C(S,T)$.

Доказательство. Для доказательства предъявим разрез и поток одинакового размера, из этого по 1.2.2 будет следовать, что поток максимален, а разрез минимален. Возьмём поток, полученный алгоритмом Форда-Фалкерсона, запустим `dfs` из истока s по рёбрам, по которым можно увеличить потока ($f_{u,v} < c_{u,v}$). Такой `dfs` отработает за время $\mathcal{O}(n+m)$, до t он не дойдёт, так как дополняющих путей нет. Вершины графа разбились на посещённые `dfs`-ом – S, и непосещённые – T. Заметим, что все рёбра из $a \in S$ в $b \in T$ обладают свойством, что $f_{a,b} = c_{a,b}$, иначе `dfs` прошёл бы по ребру и посетил бы b. Поэтому $F(S,T) = C(S,T) \Rightarrow |f| = C(S,T)$. ■

Следствие 1.2.5. Поиск минимального разреза.
Мы научились по готовому максимальному потоку за $\mathcal{O}(n+m)$ искать минимальный разрез.

Время работы алгоритма Форда-Фалкерсона равно $\mathcal{O}(|f| \cdot m)$, но зачастую гораздо меньше. Тем не менее, помните, существуют тесты, на которых даже Форд-Фалкерсон с разндомизированным `dfs` (перебирает рёбра в случайном порядке) работает экспоненциальное от n время.

Алгоритм Форда-Фалкерсона применим для целочисленных пропускных способностей. Для вещественных весов он может работать бесконечно долго.

1.3. Задачи

Задача о максимальном паросочетании в двудольном графе

Решается поиском потока за $\mathcal{O}(nm)$. Добавим исток к первой доли, сток ко второй, найдём максимальный поток. $|f| \leq n$. Рёбра двудольного графа, по которым течёт поток – максимальное паросочетание.

Задача о восстановлении матрицы

Даны суммы в строках и столбцах квадратной матрицы и информация, что все числа в матрице от 0 до 100. Задача: восстановить любую возможную матрицу. Решение: исток, первая доля – строки, вторая доля – столбцы. Рёбра из истока в строки имеют пропускную способность "сумма в стороке", аналогично рёбра из столбцов в сток. Клетке матрицы соответсвует ребро между строкой и столбцой, имеющее пропускную способность 100. Найдём в полученномм графе максимальный поток. Если все рёбра из истока и в сток насыщенны, матрица существует, матрицу задают величины потоков по рёбрам, соответсвующим клеткам.

Задача о посадке людей в самолёты

Даны самолёты, у каждого есть вместимость k_i и время вылета t_i. Даны люди, у каждого есть диапазон времён $[l_i..r_i]$, когда человек хочет улететь. Задача: распределить всех людей по самолётам. Строим граф: из истока в людей пропускная способность 1, из людей, в подходящие им самолёта пропускная способность 1, из самолётов в сток пропускная способность k_i.

1.4. Реализация

На практике, чтобы `dfs` работал за $\mathcal{O}(n+m)$ хранят не матрицу смежности, а список рёбер. Чтобы выполнялась антисимметричность потока, для каждого ребра есть обратное ему.

```
1  struct Edge {
2    int from, to; // ребро из from в to
3    int next; // номер следующего ребра из вершины from
4    int f; // flow (поток)
5    int c; // capacity (пропускная способность)
6  };
7  int n, m;
8  vector<Edge> edges;
9  vector<int> head; // для каждой вершины номер первого ребра, начало списка
10
11 void add_edge( int a, int b, int capacity ) {
12   edges.push_back(Edge {a, b, head[a], 0, capacity});
13   head[a] = edges.size() - 1;
14 }
15 void read() {
16   cin >> n >> m; // количество вершин, количество рёбер
17   edges.reserve(2 * m); // каждому ребру нужно обратное
18   head = vector<int>(n, -1); // -1 обозначает пустой список
19   while (m--) {
20     int a, b, c;
21     cin >> a >> b >> c, a--, b--; // во входных данных вершины обычно нумеруются с 1
22     add_edge(a, b, c); // прямое ребро
23     add_edge(b, a, 0); // обратное ребро
24   }
25 }
```

Теперь у ребра номер `i` есть обратное `i ^ 1`, а чтобы перебрать рёбра из вершины `i` следует начать с ребра `head[i]`, получится `for (int e = head[i]; e != -1; e = edges[e].next)`. В алгоритме Форда-Фалекрсона после того, как мы нашли путь, нужно увеличить по нему поток, это можно делать на обратном ходе `dfs`-а (возврат из рекурсии).

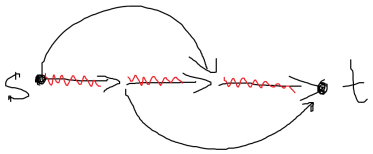
```
int cc;
vector<int> u; // Если u[i] = cc, вершина помечена, иначе не помечена

int dfs( int v, int flow ) {
  if (v == t)
    return flow;
  u[v] = cc; // dfs должен работать за линейное время, каждую вершину посещаем один раз
  for (int x, i = head[v]; i != -1; i = edges[e].next) {
    Edge &e = edges[i];
    if (e.f < e.c && u[e.to] != cc && (x = dfs(e.to, min(flow, e.c-e.f))) > 0) {
      e.f += x, edges[i ^ 1].f -= x; // увеличили по прямому, уменьшили по обратному
      return x;
    }
  }
  return 0;
}
int maxFlow() {
  read();
  cc = 1, u = vector<int>(n, 0);
  int f = 0, add;
  while ((add = dfs(s, INT_MAX)) > 0)
    f += add, cc++; // cc++ делает все вершины не помеченными
  return f;
}
```

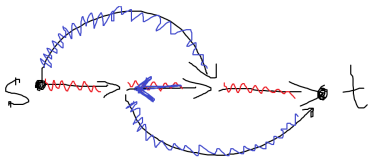
Оптимизации:

1. Можно вместо `f` и `c` хранить сразу `c - f`.
2. Почти всегда можно не хранить `from`.

Замечение: Без обратных рёбер бы алгоритм не работал:



Если бы первый `dfs` нашёл красный путь, второй `dfs` без обратных рёбер не смог бы найти путь. А с обратными смог бы найти вот такой синий путь:



1.5. Декомпозиция потока на пути

Декомпозиция потока на пути .

Заметим, что путь из s в t, по которому течёт x единиц потока, можно рассматривать, как поток размера x. Алгоритм Форда-Фалкерсона работает по принципу "сложим маленькие потоки-пути, получим большой поток". Теперь рассмотрим обратную задачу:

Def 1.5.1. *Задача декомпозиции: дан поток, представить его в виде объединения путей и циркуляции с дополнительным ограничением "в декомпозиции не должно быть взаимно обратных рёбер".*

То есть красный и синий путь на картинке выше не являются корректной декомпозицией.
Алгоритм поиска: очередной путь отщепляем `dfs`-ом по рёбрам с ненулевым потоком.

```
1  int getPath( int v, int flow ) {
2    if (v == t)
3      return flow;
4    u[v] = cc;
5    for (int x, i = head[v]; i != -1; i = edges[e].next) {
6      Edge &e = edges[i];
7      if (e.f > 0 && u[e.to] != cc && (x = getPath(e.to, min(flow, e.f))) > 0) {
8        e.f -= x, edges[i ^ 1].f += x; // отщепили путь
9        return x;
10     }
11   }
12   return 0;
13 }
```

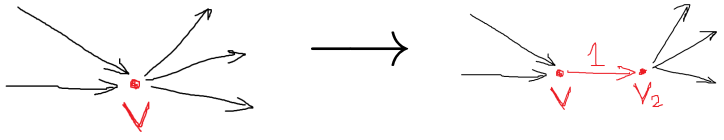
Время работы такого алгоритма $\mathcal{O}(m^2)$, так как `getPath` работает за $\mathcal{O}(m)$,
и после каждого вызова `getPath` увеличивается количество рёбер с нулевым f_e.
Упражнение: улучшить время декомпозиции до $\mathcal{O}(nm)$.

1.6. Задача про k путей

Теперь, когда мы умеем раскладывать поток на пути, можно решить следующие задачи:

1. Найти в орграфе k непересекающихся по рёбрам путей из s в t.
2. Найти в неорграфе k непересекающихся по рёбрам путей из s в t.
3. Найти в орграфе k непересекающихся по вершинам путей из s в t.
4. Найти в неорграфе k непересекающихся по вершинам путей из s в t.

Все задачи решаются за $\mathcal{O}(km)$: сперва запускаем Форда-Фалкерсона, затем декомпозицию потока. Чтобы работать с неориентированным графом нужно неориентированное ребро представить, как два ориентированных. Чтобы пути непересекались по вершинам, научимся раздваивать вершину:



В графе с раздвоенными вершинами пути, непересекающиеся по рёбрам, непересекаются по вершинам в исходном графе.

1.7. Несколько стоков и истоков

Пусть мы хотим найти несколько непересекающихся путей. Каждый должен начинаться в одной из вершин множества A и заканчиваться в вершинах множества B. То есть, нас просят найти поток на графе с несколькими истоками и стоками. Добавим исток: вершину s и рёбра из s в A. Добавим сток: вершину t и рёбра из B в t. Будем искать поток из s в t.

1.8. Алгоритмы поиска потока

Алгоритм Эдмондса-Карпа – вместо `dfs` (поиск в глубину) используем `bfs` (поиск в ширину). Оказывается, количество дополняющих путей в таком случае не более $\frac{nm}{2}$, поэтому алгоритм работает за $\mathcal{O}(nm^2)$ и подходит для графов с вещественными весами.

Время работы оставим без доказательства, его можно посмотреть в ИТМО-конспектах

Алгоритм масштабирования потока (Scaling) – будем искать дополняющие пути, по которым можно увеличить поток хотя бы на $k = 2^t$. Для этого в `dfs` главный `if` поменяем вот так:
`if (e.f + k <= e.c && u[e.to] != cc && (x = dfs(e.to, min(flow, e.c - e.f)))> 0){`
А поиск потока теперь выглядит так:

```
int maxFlow() {
  read();
  cc = 1, u = vector<int>(n, 0);
  // MAX_CAPACITY <= 2^t <= 2*MAX_CAPACITY
  for (int k = (1 << t); k > 0; k = k / 2, cc++)
    while ((add = dfs(s, k, INT_MAX)) > 0)
      f += add, cc++;
  return f;
}
```

Теорема 1.8.1. Время работы Scaling алгоритма поиска потока на графе с целочисленными пропускными способностями равно $\mathcal{O}(m^2 \log U)$, где U – максимальная пропускная способность.

Доказательство. Различных k мы переберём $\log U$ штук, `dfs` работает за $\mathcal{O}(m)$, осталось показать, что для каждого k мы найдём $\mathcal{O}(m)$ путей. Для этого обозначим за f_1 поток до поиска путей размера k, а за f_2 поток после того, как все пути размера k найдены. f_1 обладает свойством "нет дополняющих путей размера $2k$", значит, есть такой разрез, что для всех рёбер, проходящих через разрез, $c_e - f_e < 2k$. Посмотрим на декомпозицию потока $f_2 - f_1$ на пути. Каждый такой путь имеет размер не меньше k и проходит через разрез. Получаем, что путей не больше чем $\frac{m \cdot 2k}{k} = 2m = \mathcal{O}(m)$. ■

Замечание. На самом деле на практике алгоритм работает за $\mathcal{O}(m^2)$. ;)

1.9. Реклама

Из крутых и при этом не сложных алгоритмов так же стоит заметить:

1. Алгоритм Диница
   a) Ищет поток за $\mathcal{O}(n^2m)$, при соединении с идеей scaling даже за $\mathcal{O}(nm \log U)$.
   b) Позволяет искать паросочетание за $\mathcal{O}(m\sqrt{n})$, использование алгоритма Диница для поиска паросочетание в двудольном графе называется алгоритмом Хопкрофта-Карпа.
   c) Позволяет искать поток в сетях с единичными пропускными способностями за $\mathcal{O}(\min(m^{1/2}, n^{2/3})m)$.
2. Семейство алгоритмов поиска потока, основанных на идее поиска предпотока (preflow), и использующих идею global relabeling. Такие алгоритмы хорошо себя ведут на практике. Не хуже nm.
3. Алгоритм от Ahuja и Orlin 1992-го года, позволяеющей с помощью идей предпотока и масштабирования избытка искать поток за $\mathcal{O}(nm + n^2 \log U)$.

2. Mincost Flows

2.1. Формулировка задачи

Дан ориентированный взвешенный граф. w_e – вес ребра. При поиске потока у каждого ориентированного ребра e есть обратное ребро e'. Мы уже знаем, что $f_e = -f_{e'}$. Определим вес обратного ребра равным $-w_e$. Теперь для потока f определим вес $W(f) = \frac{1}{2}\sum_e f_e w_e$. Заметим, что $W(f) = \sum_{e \in E} f_e w_e$, где E – множество прямых рёбер.
Обозначение $W(x)$ будем применять не только к потокам.
Если p – путь, $W(p)$ – вес пути. Если z – циркуляция, $W(z)$ – вес циркуляции.

- **Min cost flow:** найти поток f: $W(f) = \min$
- **Min cost max flow:** найти поток f: $|f| = \max, W(f) = \min$
- **Min cost k-flow:** найти поток f: $|f| = k, W(f) = \min$

2.2. Алгоритм поиска min cost k-flow

Lm 2.2.1. Если в остаточной сети есть отрицательный цикл, поток не mincost

Доказательство. Пусть C – отрицательный цикл, заменим f на $f + C$ (увеличим поток по циклу). Размер не изменился, вес уменьшился. ■

Lm 2.2.2. Если в остаточной сети нет отрицательных циклов, поток mincost

Доказательство. Пусть есть наш поток f_1 и поток f_2: $|f_1| = |f_2|, W(f_1) > W(f_2)$, рассмотрим поток $h = f_2 - f_1$ в сети G_{f_1}. Это корректный поток, так как $(f_2 - f_1)_e \le (c - f_1)_e$. $|h| = 0$, $W(h) < 0$, то есть перед нами отрицательная циркуляция. Осталось заметить, что так же, как поток можно разбить на пути и циркуляцию, циркуляцию можно разбить на циклы, один из них будет отрицательным. ■

Lm 2.2.3. Обозначим за f_k mincost k-flow. Тогда $f_{k+1} = f_k + \texttt{shortestPath} + \texttt{zeroCirculation}$.

Доказательство. Рассмотрим поток $h = f_{k+1} - f_k$ в G_{f_k}. $|h| = 1$, декомпозиция h – это путь p плюс циркуляция z. Из того, что f_{k+1} mincost, имеем $W(z) \le 0$, а $W(p) = W(\texttt{shortestPath})$. По лемме 2.2.1 имеем $W(z) \ge 0 \Rightarrow W(z) = 0$. ■

Алгоритм поиска mincost k-flow в графе без отрицательных циклов

```
f ← 0 (нет отрицательных циклов)
for i=1..k:
    ищем Форд-Беллманом кратчайший путь
    увеличим поток по пути на 1
return f
```

Алгоритм работает за $\mathcal{O}(knm)$.

Обозначим $d_k = W(f_{k+1}) - W(f_k)$ (вес k-го дополняющего пути). Из доказательства алгоритма Эдмондса Карпа [2] следует, что $d_{k+1} \ge d_k$. Поэтому можно соптимизировать: толкать не единицу потока, а столько, сколько сможем, $\min_{e \in path}(c_e - f_e)$.

2.3. Алгоритм поиска mincost flow

Мы уже знаем, что $d_{k+1} \geq d_k$. Вес потока уменьшается, пока $d_k < 0$.
Алгоритм: начинаем с f_0, наращиваем поток, пока $d_k < 0$.
Время работы $\mathcal{O}(nm|f|)$, где $|f|$ – размер искомого потока.

2.4. Реализация mincost flow

Модифицируем код [1], получим

```
void add_edge( int a, int b, int capacity, int weight ) {
  edges.push_back(Edge {a, b, head[a], 0, capacity, weight});
  head[a] = edges.size() - 1;
}

queue<int, vector<int>> q; // второй параметр обеспечивает скорость работы
void relax( int v, int d ) {
  if (dist[v] > d)
    return;
  dist[v] = d;
  if (!in_queue[v])
    q.push(v), in_queue[v] = 1;
}
bool ford_bellman() {
  const int infty = 1e9;
  for (int i = 0; i < n; i++)
    dist[i] = infty, in_queue[i] = 0;
  relax(s, 0);
  while (q.size()) {
    int v = q.front(); q.pop();
    in_queue[v] = 0;
    for (int i = head[v]; i != -1; i = edges[e].next) {
      Edge &e = edges[i];
      if (e.f < e.c)
        relax(e.to, d[v] + e.weight);
    }
  }
  return dist[t] < infty;
}
...
add_edge(a, b, c, w); // прямое ребро
add_edge(a, b, 0, -w); // обратное ребро
```

Данная реализация Форд-Беллмана в худшем случае использует $\mathcal{O}(n)$ дополнительной памяти и $\mathcal{O}(nm)$ времени. В среднем работает за $\mathcal{O}(m)$. Называется эта реализация "Форд-Беллман с очередью" или "Shortest Path Faster Algorithm" (SPFA). Не путайте, пожалуйста, эту реализацию с алгоритмом Левита, который добавляет в начало очереди.

2.5. Отрицательные циклы

Если в графе есть отрицательные циклы, $f_0 \neq 0$. Чтобы найти f_0, воспользуемся леммами 2.2.1 и 2.2.2. Начнём с пустого потока, пока в остаточной сети G_f есть отрицательный цикл, увеличим поток по циклу. Отрицательный цикл можно искать за $\mathcal{O}(nm)$ с помощью алгоритма Форда-Беллмана.

Так можно искать не только f_0, но и f_k.

Другой алгоритм поиск mincost k-flow

a) Найдём какой-нибудь поток размера k любым алгоритмом поиска потока.
b) Пока в остаточной сети есть отрицательный цикл, увеличим поток по циклу.
c) Если нет отрицательных циклов, по лемме 2.2.2 поток mincost.

Этот алгоритм работает особенно хорошо, если искать не произвольный отрицательный цикл, а "цикл минимального среднего веса"

2.6. Реклама

Идея потенциалов Джонсона позволяет Форд-Беллмана заменить на Дейкстру. Алгоритм поиска mincost k-flow после этого работает за $\mathcal{O}(km \log n)$ и даже $\mathcal{O}(k(m + n \log n))$.

Заметим, что мы не научились искать mincost потока за полиномиальное от n, m и $\log U$ время. Такие алгоритмы существуют. Самый простой из них основан на идее "capacity scaling" и работает за время $\mathcal{O}(\texttt{Dijkstra}\ m \log U)$ (столько раз запускает Дейкстру).