# Autumn training camp in MIPT, november, 2015
# lection notes

Compiled Friday 20th November, 2015 at 12:58

## Contents

# 1. Flows

## 1.1. Definitions

Let we have oriented graph $G$ of $n$ vertices and $m$ edges. Between each two vertices $v$ and $u$ of $G$ there is edge with capacity $c_{v,u} \geq 0$. If we want to say "there is no edge", let $c_{v,u} = 0$.

**Def 1.1.1.** *Flow from $s$ to $t$ – function $f_{v,u} \in \mathbb{R}$, which satisfies three propetries*

*1. $\forall v, u \quad f_{v,u} \leq c_{v,u}$ (flow is not greater than capacity)*
*2. $\forall v, u \quad f_{v,u} = -f_{u,v}$ (antisymmetry)*
*3. $\forall v \neq s, t \quad \sum_u f_{v,u} = 0$ (conservation of flow)*

The $s$ is called *source*, and the $t$ is called *sink*.
Let denote $f_e$ and $c_e$ – flow and capacity for edge $e$.

**Def 1.1.2.** *Cut between $s$ and $t$ – partition of vertices of the graph $V\colon V = S \sqcup T, s \in S, t \in T$. ($S \cap T = \varnothing$, source is in $S$, sink is in $T$)*

Let denote cut as $(S, T)$.

**Def 1.1.3.** *Value of flow $|f| = \sum_v f_{s,v}$ (sum of flow from source $s$).*

Value of flow may be negative. It does not contradict the definition.

**Def 1.1.4.** *Value of cut $C(S,T) = \sum\limits_{v \in S, u \in T} c_{v,u}$ (sum of capacities of edges throw the cut).*

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

This is just a notation. $A$ and $B$ may intersect or even coincide.

**Def 1.1.6.** *Max flow – $f\colon |f| = \max$*

**Def 1.1.7.** *Min cut – $(S,T)\colon C(S,T) = \min$*

**Def 1.1.8.** *Edge $e$ is saturated $\Leftrightarrow f_e = c_e$*

**Def 1.1.9.** *Residual network $G_f$ – graph of edges $c_e - f_e > 0$.*
*Capacities of edges in residual network are $c_e - f_e$.*

**Def 1.1.10.** *Augumenting path – path from $s$ to $t$ throw edges $e\colon f_e < c_e$.*
*In other words "path in residual network"*

**Def 1.1.11.** *Circulation – flow of zero value.*

For example, any cycle is a circulation.

## 1.2. Theorem and algorithm of Ford-Fulkerson

**Lm 1.2.1.** Value of flow equals to value of the flow, flowing throw the cut:
$\forall$ cut $(S, T), \forall$ flow $f \quad |f| = F(S, T)$

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

**Lm 1.2.2.** Any flow is no more than any cut: $\forall$ cut $(S, T)$, flow $f \quad |f| \leq C(S, T)$

*Proof.* $|f| = F(S, T) \leq C(S, T)$ ■

**Lm 1.2.3.** *About augumenting path.* If there is an augumenting path, flow is not max flow.

*Proof.* Let there is an augumenting path. Let increase flow along the path, and decrease flow in opposite direction. All 3 propetries are satisfied, value of the flow is increased. ■

**Algorithm of Ford-Fulkerson.**
The algorithm searches for max flow in graph with integer capacities.

```
$f \leftarrow 0$
while augumenting path exists:
    find it using dfs, which uses only not saturated edges
    let path is:  $s = v_1, v_2, \ldots, v_k = t$
    $x = \min_{i=1..k-1}\left[c_{v_{i+1},v_i} - f_{v_{i+1},v_i}\right]$ ($x > 0$; $x =$ how much flow we may push along the path)
    increase the flow along the path by $x$
return $f$
```

If $c_{v,u}$ are integer then $x \geq 1$, so the algorithm somewhen finishes.
Maximality of resulting flow follows from the theorem:

**Теорема 1.2.4. Ford-Fulkerson's.** $\max |f| = \min C(S, T)$.

*Proof.* Let show cut, whose value is same as value of the flow. It follows (use Lm 1.2.2) that flow is max flow and cut is min cut. Let take flow obtained by Ford-Fulkerson's algorithm. Let launch `dfs` from $s$ using only not saturated edges ($f_{u,v} < c_{u,v}$). Such `dfs` finishes in $\mathcal{O}(n+m)$. It does not reach $t$, because there is no augumenting path. Vertices of the graph are splitted into two sets – marked by `dfs` ($S$), and not marked ($T$). All edges from $a \in S$ to $b \in T$ satisfy $f_{a,b} = c_{a,b}$, in other case `dfs` would go throw the edge and visit $b$. So $F(S, T) = C(S, T) \Rightarrow |f| = C(S, T)$. ■

*Implication* 1.2.5. Algorithm to find min cut.
If we already have max flow then we may build min cut in $\mathcal{O}(n + m)$ time.

Working time of Ford-Fulkerson's algorithm is $\mathcal{O}(|f| \cdot m)$, but in practice it is smaller. Any way there are tests such that even Ford-Fulkerson with randomizatin in `dfs` (iterate edges in random order) works in exponential of $n$ time.

Algorithm of Ford-Fulkerson is applicable for integer capacities.
For ratio capacities it can work infinitely long.

## 1.3. Problems

**Problem about maximum matching in bipartite graph**

It may be solved in $\mathcal{O}(nm)$, using max flow. Add source to the first part of vertices, add sink to the second part, find max flow. $|f| \le n$. Edges of bipartite graph (edges between parts), wich are saturated by the max flow are edges of the maximum matching.

**Problem about matrix recovery**

You are given sums in rows and columns of square matrix. All numbers in the matrix are from 0 to 100. The problem: recovery the matrix. The solution: let consider bipartite graph with source and sink. The first part – rows, the second part – columns. Let add edges from source to rows witch capacity "sum in the row". Also let add same edges from columns to sink. Cell of the matrix – edges between row and column with capacity 100. Let find max flow in this graph. If all edges from source and all edges to sink are saturated then matrix exists and flow between rows and columns describes the matrix.

**Problem about people and flights**

There is an airport. Some flights are going the same direction, each can transfer $k_i$ and will flight out at $t_i$. There are some people, each human has range of acceptable times $[l_i..r_i]$, when he wants to flight out. The problem is to distribute humans to flights. Let build graph: from source edges to humans with capacity 1, from humans edges to flights, from flights to sink edges with capacity $k_i$.

## 1.4. Implementation

To make `dfs` work in $\mathcal{O}(n+m)$ time, let store lists of adjacent edges for each vertex. To maintain antisymmetry of flow, each edge has paired backward edge.

```
struct Edge {
  int from, to;
  int next; // number of the next edge from vertex "from"
  int f; // flow
  int c; // capacity
};
int n, m;
vector<Edge> edges;
vector<int> head; // number of the first edge for each vertex

void add_edge( int a, int b, int capacity ) {
  edges.push_back(Edge {a, b, head[a], 0, capacity});
  head[a] = edges.size() - 1;
}
void read() {
  cin >> n >> m; // number of vertices and edges
  edges.reserve(2 * m); // each edge has paired backward edge
  head = vector<int>(n, -1); // -1 means empty list
  while (m--) {
    int a, b, c;
    cin >> a >> b >> c, a--, b--; // in input vertices usually are numbered 1..n
    add_edge(a, b, c); // forward edge
    add_edge(b, a, 0); // backward edge
  }
}
```

Now that edge number `i` has paired backward edge number `i ^ 1`. To iterate all edges from vertex `i` you should start from edge `head[i]`: `for (int e = head[i]; e != -1; e = edges[e].next)`. In Ford-Fulkerson algorithm we should find path and increase flow throw the path. We will increase the flow while backtracking.

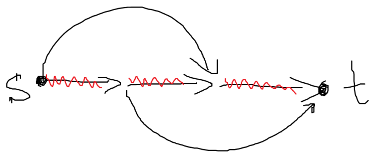
```
int cc;
vector<int> u; // Iff u[i] = cc then vertice is marked

int dfs( int v, int flow ) {
  if (v == t)
    return flow;
  u[v] = cc; // do not visit any vertice twice
  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; // incerease forward edge, decrease backward
      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++; // command "cc++" unmarks all vertices
  return f;
}
```

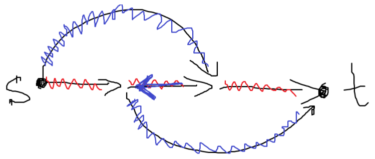
Optimizations:

1. You may store `c-f` instead of storing both `f` and `c`.
2. Almost always you should not store `from`.

**<u>Remark:</u>** without backward edges algorithm fails:



If the first `dfs` finds the red path, the second `dfs` can not find a path without backward edges. Using backward edges it can find the blue path.



## 1.5. Flow decomposition

**Decompose flow into paths** .

Path from $s$ to $t$, through which flows $x$ items is a flow of value $x$. Ford-Fulkerson algorithm works as Алгоритм Форда-Фалкерсона works as "let add small paths-flows and obtain big flow". Now that we should solve opposite problem:

**Def 1.5.1.** *Decomposition: you have to describe flow as sum of some pathes and a circulation. Additional restricton: in the decomposition there are no opposite edges.*

Red and blue paths on the picture above are not correct decomposition, because one edge is used in both directions.

Algorithm to build decomposition: cleave another path by `dfs` using edges with non zero flow.

```
int getPath( int v, int flow ) {
  if (v == t)
    return flow;
  u[v] = cc;
  for (int x, i = head[v]; i != -1; i = edges[e].next) {
    Edge &e = edges[i];
    if (e.f > 0 && u[e.to] != cc && (x = getPath(e.to, min(flow, e.f))) > 0) {
      e.f -= x, edges[i ^ 1].f += x; // cleave the path
      return x;
    }
  }
  return 0;
}
```

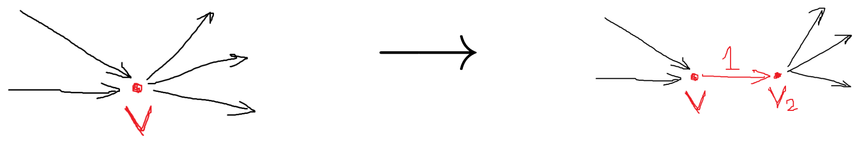
Working time is $\mathcal{O}(m^2)$, because `getPath` works in $\mathcal{O}(m)$ time
and after each launch of `getPath` number of edges with zero flow incereases.
*Exercise:* improve working time up to $\mathcal{O}(nm)$.

## 1.6. Problem about $k$ paths

Now that using decomposition of flow into paths we can solve following problems:

1. Find $k$ disjoint by edges paths from $s$ to $t$ in oriented graph.
2. Find $k$ disjoint by edges paths from $s$ to $t$ in not oriented graph.
3. Find $k$ disjoint by vertices paths from $s$ to $t$ in oriented graph.
4. Find $k$ disjoint by vertices paths from $s$ to $t$ in not oriented graph.

We can solve all these problems in $\mathcal{O}(km)$: launch Ford-Fulkerson's algorithm, then decompose the flow. In not oriented graph a not oriented edge can be described as two oriented edges. To make pathes disjoint by vertices, let make a copy for each vertex:



Disjoint by edges path in new graph are disjoint by vertices path in initial graph.

## 1.7. Several sources and sinks

We have to find several disjoint paths. Each path should begin in a vertex from set $A$. Each path should end in a vertex from set $B$. In other words we have to find flow in graph with several sources and sinks. Let add global source – vertex $s$ with edges from $s$ to all $A$. Add global sink – vertex $t$ with edges from $B$ to $t$. Find flow from $s$ to $t$.

## 1.8. Algorithms for max flow problem

**Edmonds–Karp algorithm** – let use `bfs` instead of `dfs`. It turns out, number of augumenting paths do not exceed $\frac{nm}{2}$, so algorithm works in time $\mathcal{O}(nm^2)$ and also works for graphs with ratio capacities.

We won't prove asymptotic of working time, you may read it here.

**Scaling flow algorithm** – let look for augumenting path, along which we can push $k = 2^t$ extra flow. Main condition in `dfs` is transformed:

```
if (e.f + k <= e.c && u[e.to] != cc && (x = dfs(e.to, min(flow, e.c - e.f)))> 0){
```

Main loop of algorithm is transformed:

```
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.** On graphs with integer capacities Scaling algorithm works in time $\mathcal{O}(m^2 \log U)$, where $U$ – maximal capacity.

*Proof.* We iterate $\log U$ different $k$, `dfs` works in $\mathcal{O}(m)$, now we have to show that for each $k$ we'll get only $\mathcal{O}(m)$ paths. Denote flow before looking for paths of value $k$ as $f_1$. Denote flow after looking for all paths of size $k$ as $f_2$. For $f_1$ there is no "augumenting paths of size $2k$", so there is such cut that all edges $e$, going throw the cut, satisfy $c_e - f_e < 2k$. Consider decomposition of flow $f_2 - f_1$. Each path in decomposition has size at least $k$ and goes throw the cut. So there are no more than $\frac{m \cdot 2k}{k} = 2m = \mathcal{O}(m)$ paths. ∎

**Remark.** In practice Scaling algorithm works in $\mathcal{O}(m^2)$. ;)

## 1.9. Advertisement

There are some cool and quiet easy algorithms:

1. Dinic's algorithm
   a) Finds max flow in $\mathcal{O}(n^2m)$, with Scaling idea works in $\mathcal{O}(nm\log U)$.
   b) Gives Hopcroft–Karp algorithm to find matching in bipartite graph in time $\mathcal{O}(m\sqrt{n})$.
   c) Finds max flow in networks with unit capacities in $\mathcal{O}(\min(m^{1/2}, n^{2/3})m)$ time.
2. Family of algorithms based on "preflow" idea and using "global relabeling" optimization.
   In practice these algorithms are good. Not worse than $nm$.
3. Ahuja-Orlin's algorithm based on ideas of "preflow" and "scaling of excess".
   It works in $\mathcal{O}(nm + n^2\log U)$ time.

# 2. Mincost Flows

## 2.1. Definitions

You are given weighted oriented graph. $w_e$ – weight of edge $e$. Each edge $e$ has backward pair $e'$. We already now, $f_e = -f_{e'}$. Let define weight of backward edge equal to $-w_e$. Let define weight of flow $f$ equal to $W(f) = \frac{1}{2}\sum_e f_e w_e$. Note, $W(f) = \sum_{e \in E} f_e w_e$, where $E$ – set of forward edges. Notation $W(x)$ we also will use for path $p$ ($W(p)$ – weight of the path), circulation $z$ ($W(z)$ – weight of the circulation). Let define common problems:

- **Min cost flow:** to find flow $f\colon W(f) = \min$
- **Min cost max flow:** to find flow $f\colon |f| = \max, W(f) = \min$
- **Min cost k-flow:** to find flow $f\colon |f| = k, W(f) = \min$

## 2.2. Algorithm for "min cost k-flow" problem

**Lm 2.2.1.** If there is negative cycle in residual network then flow is not mincost.

*Proof.* Let $C$ – negative cycle in $G_f$, consider flow $f + C$ (increase flow along the cycle). Value was not changed. Weight was decreased. ■

**Lm 2.2.2.** If there are no negative cycles in residual network then flow is mincost.

*Proof.* Let there is flow $f_1$ and flow $f_2\colon |f_1| = |f_2|, W(f_1) > W(f_2)$, Consider flow $h = f_2 - f_1$ in network $G_{f_1}$. It's correct flow because of $(f_2 - f_1)_e \le (c - f_1)_e$. Note $|h| = 0$, $W(h) < 0$, it's negative circulation. Circulation can be decomposed into cycles. One of these cycles has negative weight. ■

**Lm 2.2.3.** Denote mincost k-flow as $f_k$. Then $f_{k+1} = f_k + \texttt{shortestPath} + \texttt{zeroCirculation}$.

*Proof.* Consider flow $h = f_{k+1} - f_k$ in $G_{f_k}$. $|h| = 1$, decomposition of $h$ is path $p$ and circulation $z$. $f_{k+1}$ is mincost, so $W(z) \le 0$ and $W(p) = W(\texttt{shortestPath})$.
Using lemma 2.2.1 we have $W(z) \ge 0 \Rightarrow W(z) = 0$. ■

**Algorithm for "mincost k-flow" problem** (works only for graphs without negative cycles).

```
$f \leftarrow 0$ (there are no negative cycles)
for i=1..k:
    find shortest path using Bellman-Ford algorithm
    increase flow along the path by 1
return $f$
```

Algorithm works in $\mathcal{O}(knm)$ time.

Denote $d_k = W(f_{k+1}) - W(f_k)$ (weight of $k$-th augumenting path). From proof of Edmonds-Karp algorithm [2] we have $d_{k+1} \ge d_k$. So we may optimize the algorithm described above, let push not one unit of flow, but $\min_{e \in path}(c_e - f_e)$ units.

## 2.3. Algorithm for "min cost flow" problem

We already know $d_{k+1} \geq d_k$. Weight of the flow decreases while $d_k < 0$.
Algorithm: let start with $f_0$ and while $d_k < 0$ add the path to the flow.
Working time is $\mathcal{O}(nm|f|)$, where $|f|$ – value of the seeking flow.

## 2.4. Implementation of mincost flow algorithm

Let modificate the code [1], we'll get:

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

queue<int> q;
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); // forward edge
add_edge(a, b, 0, -w); // backward edge
```

This implementation of Bellman-Ford algorithm uses $\mathcal{O}(n)$ of memory and $\mathcal{O}(nm)$ of time in the worst case. In average case it works in $\mathcal{O}(m)$. This implementation is known as "Shortest Path Faster Algorithm" (SPFA). Please, do not confuse this implementation with "Levit's algorithm". Levit pushes some vertices to front of queue.

## 2.5. Negative cycles

If there are negative cycles in the graph then $f_0 \neq 0$. To find $f_0$, let use lemmas 2.2.1 and 2.2.2. Start with empty flow. Then while there is negative cycle in residual network, let increase the flow along the cycle. We can use Bellman-Ford algorithm to find negative cycle in $\mathcal{O}(nm)$ time.

This approach gives us another algorithm to find $f_k$.

**Algorithm for "mincost k-flow" problem**

a) Find any flow of value $k$.
b) While there is negative cycle in residual network, let increase the flow along the cycle.
c) If there are no negative cycles the flow is mincost 2.2.2.

This algorithm works especially good, if we look for "cycle of minimal mean weight"

## 2.6. Advertisement

Idea of Johnson's potentials allows us to use Dijkstra instead of Bellman-Frd. Now that algorithm for mincost k-flow problem works in $\mathcal{O}(km \log n)$ time and even in $\mathcal{O}(k(m + n \log n))$ time.

Note, we do not know yet how to find mincode in polynomial of $n$, $m$ and $\log U$ time. Such algorithms exist. The most simple is based on idea of "capacity scaling" and works in time $\mathcal{O}(\texttt{Dijkstra}\ m \log U)$ ($m \log U$ times launches Dijkstra's algorithm).