Hello Barcelona Programming Bootcamp, September, 2018 lecture notes

Contents

1. Flows **1**
1.1. Definitions . . . 1
1.2. Ford-Fulkerson Theorem and Algorithm . . . 2
1.3. Problems . . . 3
1.4. Implementation . . . 3
1.5. Flow decomposition . . . 6
1.6. Problems about k paths . . . 6
1.7. Algorithms for max flow problem . . . 7
1.8. Further read . . . 8

2. Mincost Flows **9**
2.1. Definitions . . . 9
2.2. Algorithm for "min cost k-flow" problem . . . 9
2.3. Algorithm for "min cost flow" problem . . . 10
2.4. Implementation of mincost flow algorithm . . . 10
2.5. Negative cycles . . . 11
2.6. Further read . . . 11

1. Flows

I want to acknowledge the work of Sergey Kopeliovich who was the original author of a given text and whose hand-drawn pictures are still present in it :)

1.1. Definitions

Suppose we have a directed graph G consisting of n vertices and m edges. Each edge $(u, v) \in E$ of the graph is associated with a non-negative real number $c_{u,v} \geq 0$ called *capacity*. For $(u, v) \notin E$ we assume $c_{u,v} = 0$.

Def 1.1.1. *s–t flow is a function $f : V \times V \to \mathbb{R}$ satisfying the following three properties:*
1. $\forall v, u \quad f_{v,u} \leq c_{v,u}$ (flow value is bounded by the capacity of the edge)
2. $\forall v, u \quad f_{v,u} = -f_{u,v}$ (skew symmetry)
3. $\forall v \neq s, t \quad \sum_u f_{v,u} = 0$ (flow conservation)

Vertices s and t are special, s is called a *source* and t is called a *sink*.
We will denote as f_e and c_e flow and capacity of the edge e.

Def 1.1.2. *A cut is a partition of V consisting of two disjoint sets $A \sqcup B = V$.*

Def 1.1.3. *s–t cut is a cut (S, T) such that $s \in S$ and $t \in T$.*

Def 1.1.4. *Value of the flow $|f| = \sum_v f_{s,v}$ (total flow through the source) s.*

A simple exercise: show that $|f| = \sum_v f_{s,v} = -\sum_u f_{u,t}$.

Def 1.1.5. *Capacity of the cut $c(A, B) = \sum\limits_{v \in A, u \in B} c_{v,u}$ (sum of capacities of edges crossing the cut).*

Note that in general case $c(A, B) \neq c(B, A)$ because the definition above is not symmetric and c may not be a symmetric function.

Def 1.1.6. *Denote $f(A, B) = \sum\limits_{v \in A, u \in B} f_{v,u}$*

Note that A and B are arbitrary sets in definition above, they may be intersecting or even coincide. In particular, $f(A, A) = 0$ for any set A.

Def 1.1.7. *Max flow is such flow f that $|f| = \max$.*

Def 1.1.8. *Min cut is such cut s–t cut that $c(S, T) = \min$.*

Def 1.1.9. *Residual capacity of edge e in the flow f is $r_e = c_e - f_e$.*

The residual capacity of the edge is the maximum extra flow that can be added to the edge. Note that if there exists a non-zero flow through the edge (v, u), then the residual capacity of the opposite edge (u, v) is non-zero: $r(u, v) = c(u, v) - f(u, v) = c(u, v) + f(v, u) > 0$. This means that we can “cancel” the flow by sending the flow through the same edge in opposite direction.

Def 1.1.10. *Edge e is saturated $\Leftrightarrow f_e = c_e \Leftrightarrow r_e = 0$*

Def 1.1.11. *Residual network G_f is the graph of edges e such that $r_e > 0$.*

Def 1.1.12. *Augmenting path is the path from s to t through the edges with non-zero residual capacity.*

Def 1.1.13. *Circulation is the flow of zero value.*

For example, flow taking constant value on all edges of some fixed cycle is a circulation.

1.2. Ford-Fulkerson Theorem and Algorithm

Lm 1.2.1. Value of the flow is equal to the total flow on the edges of any s–t cut: $\forall$ cut $(S,T), \forall$ flow $f \quad |f| = f(S,T)$

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

As a consequence, we derive the following lemma:

Lm 1.2.2. The value of any flow is no more than the capacity of any cut: $\forall$ cut (S,T), flow $f \quad |f| \le c(S,T)$

Proof. $|f| = f(S,T) \le c(S,T)$ ∎

As a consequnce, if for a flow f there exists a cut (S,T) such that $|f| = c(S,T)$, then f is a maximum flow.

Theorem 1.2.3. [**Ford, Fulkerson**, max-flow min-cut theorem] $\quad \max|f| = \min c(S,T)$.

Proof. If there exists an augmenting path P then the flow is actually not maximum (consider the minimum residual capacity r over all edges of P and consider the flow $f' = f + rP$ where rP is the flow of value r through the path P. It's easy to see that f' is a correct flow and $|f'| = |f| + r > |f|$).

So, there exists no augmenting path from s to t. Consider S = set of all vertices reachable from s via the non-zero edges of a residual network and $T = V \setminus S$. For any edge (v,u) such that $v \in S$ and $u \in T$, $r_{v,u} = 0 \Leftrightarrow f_{v,u} = c_{v,u}$, so $f(S,T) = c(S,T)$. Hence, the flow f is maximum. ∎

Algorithm of Ford-Fulkerson.

The algorithm builds the maximum flow in graph with integral capacities.

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

If $c_{v,u}$ are integral then $x \ge 1$, so the running time is finite.

A corollary is that we have an algorithm of finding min cut having the max flow in $\mathcal{O}(n+m)$ time. The S component should consist of all vertices reachable from s via the non-zero edges of the residual network, the T component should consist of all remaining vertices.

Working time of Ford-Fulkerson's algorithm is $\mathcal{O}(|f| \cdot m)$, but in real life it is usually pretty efficient. Though there exist tests such that even the implementation iterating over edges in random order works in exponential by n time.

Algorithm of Ford-Fulkerson works fine for integral capacities.

For rational capacities it may work infinitely long and even do not converge to the correct answer.

1.3. Problems

Maximum matching in bipartite graph

It may be solved in $\mathcal{O}(nm)$, using max flow. Add source to the left side of the graph, add sink to the right side, connect the source to all vertices in the left part with a unit capacity edges, connect the right part with a sink with unit capacity edges also, find the maximum flow. Since we have $|f| \leq n$, the running time will be $O(nm)$. Edges of between two parts that are saturated by the max flow are the edges of the maximum matching.

Matrix recovery problem You are given sums in rows and columns of square matrix. All numbers in the matrix are between 0 and 100. The problem is to recover the cells of the matrix.

Consider the bipartite graph with the first part corresponding to the rows, and the second part corresponding to the columns. Let's 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 corresponds to the edges between row and column with capacity 100. Let's find max flow in this graph. If all edges from source and all edges to sink are saturated then the matrix exists and flow between rows and columns describes the matrix.

Projects and instruments We have n projects and m instruments. For each project we know the profit of its completion p_i and for each instrument we know its cost c_j. Also for each project it is known which instruments should be bought in order to complete the corresponding project, instruments may be reused between projects. What is the maximum possible overall profit?

Draw a bipartite graph with nodes in the left part corresponding to the projects and the nodes in the right part corresponding to the instruments. Connect the source to the nodes in the left part by edges with capacities p_i, connect the nodes in the right part to the sink by edges with capacities c_j. Connect each project to each instrument with the infinite capacity.

Assume that we complete all the projects from the S component of the minimum cut in this graph and we buy all the instruments from the S component. First of all, all dependencies are met because otherwise it would mean that there exists a project from S and an instrument from T such that they are connected with the edge of infinite capacity that is pretty strange for a minimum cut.

The overall capacity of the cut is equal to $\sum\limits_{i \notin S} p_i + \sum\limits_{j \in S} c_j = \sum\limits_{i} p_i - \sum\limits_{i \in S} p_i + \sum\limits_{j \in S} c_j$. Note that the first term is constant, so it can be discarded.

$-\sum\limits_{i \in S} p_i + \sum\limits_{j \in S} c_j \to \min \Leftrightarrow \sum\limits_{i \in S} p_i - \sum\limits_{j \in S} c_j \to \max$, so we achieve the desired maximum profit.

1.4. Implementation

To make `dfs` work in $\mathcal{O}(n+m)$ time we can store lists of adjacent edges for each vertex. To maintain the skew-symmetry of flow, each edge has a corresponding opposite edge.

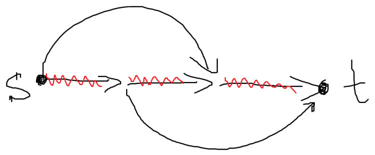
```
struct edge {
    int to;
    int next; // number of the next edge from the vertex "from"
    int f; // flow
    int c; // capacity
};

int n, m;
vector<edge> edges;
vector<int> first; // number of the first edge for each vertex
```

```

void add_edge(int a, int b, int capacity) {
    edges.push_back({b, first[a], 0, capacity});
    edges.push_back({a, first[b], 0, 0});
    first[a] = edges.size() - 2;
    first[b] = edges.size() - 1;
}

void init() {
    cin >> n >> m; // number of vertices and edges
    edges.reserve(2 * m); // each edge has an opposite edge
    first = vector<int>(n, -1); // -1 means the end of the list
    for (int i = 0; i < m; i++) {
        int a, b, c;
        cin >> a >> b >> c;
    }
}
```

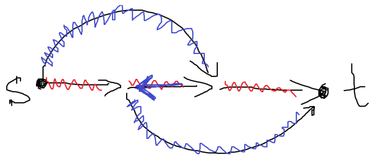
The nice observation is that the edge number `e` opposite is the edge number `e ^ 1`. To iterate over all edges from vertex `v` you should start from edge `first[v]`: `for (int e = first[v]; e != -1; e = edges[e].next)`.

```
// Instead of clearing the "used" array we increase the variable "cur_time" for
    each
// iteration of DFS and store the timestamp of the last visit.
int cur_time = 0;
vector<int> timestamp;

int DFS(int v, int flow = inf)  {
    if (v == t)
        return flow;
    timestamp[v] = cur_time; // do not visit any vertex twice
    for (int e = first[v]; e != -1; e = edges[e].next) {
        if (edges[e].f < edges[e].c && timestamp[edges[e].to] != cur_time) {
            int pushed = dfs(e.to, min(flow, edges[e].c - edges[e].f)); // the
                actual value of the flow we were able to push
            if (pushed > 0) {
                edges[e].f += x, edges[e ^ 1].f -= x; // increase the direct edge
                    , decrease the opposite one
           }
            return x;
        }
    }
    return 0;
}

int maxFlow() {
    init();
    cur_time = 0;
    timestamp = vector<int>(n, 0);
    int f = 0, add;
    while (true) {
        cur_time++;
        add = dfs(s);
        if (add > 0) {
            f += add;
        } else {
            break;
        }
    }
    return f;
}
```

Remark: why the skew-symmetry and the opposite edges are important:



If the first `dfs` finds the red path, the second `dfs` can not find a path without opposite edges. By using opposite edges it can find the blue path.



1.5. Flow decomposition

Decompose flow into paths .

Def 1.5.1. *Decomposition: you have to express flow as a sum of some paths and a circulation. Additional restricton: in the decomposition there should be no opposite edges (i. e. the sign of the flow over each of the edges should be the same as in the flow itself)*

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

Algorithm to build decomposition: subtract paths one by one by doing the similar `dfs` using the edges with non zero flow.

```
// timestamp should be increased after each call of this function
int getPath(int v, int flow = 1e9) {
    if (v == t)
        return flow;
    timestamp[v] = cur_time;
    for (int e = first[v]; e != -1; e = edges[e].next) {
        if (edges[e].f > 0 && timestamp[edges[e].to] != cur_time) {
            int pushed = getPath(e.to, min(flow, e.f));
            if (pushed > 0) {
                edges[e].f -= x, edges[e ^ 1].f += x; // subtract 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 non-zero flow decreases.

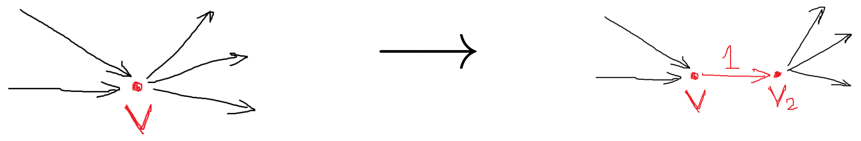
Exercise: improve the working to $\mathcal{O}(nm)$

1.6. Problems about k paths

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

1. Find k disjoint by edges paths from s to t in directed graph.
2. Find k disjoint by edges paths from s to t in undirected graph.
3. Find k disjoint by vertices paths from s to t in directed graph.
4. Find k disjoint by vertices paths from s to t in undirected graph.

We can solve all these problems in $\mathcal{O}(km)$: launch Ford-Fulkerson's algorithm, then decompose the flow. In undirected graph a bidirectional edge can be represented as two directed edges. To make paths disjoint by vertices, let's make a copy for each vertex:



Edge-disjoint paths in the new graph are vertex-disjoint in the original graph.

1.7. Algorithms for max flow problem

Edmonds–Karp algorithm – let's use `BFS` instead of `DFS`. It turns out that number of augumenting paths do not exceed $\frac{nm}{2}$ when using BFS, so the algorithm works in time $\mathcal{O}(nm^2)$ and also works for graphs with real capacities.

We won't prove the running time, you may read it here.

Flow scaling algorithm – let's look for na augumenting path along which we can push at least $k = 2^t$ extra flow. Now main condition in `dfs` looks like the following: `if (edges[e].f + k <= edges[e].c && timestamp[edges[e].to] != cur_time)...` Main loop of algorithm is transformed:

```
int maxFlow() {
    init();
    cur_time = 1, timestamp = vector<int>(n, 0);
    int add;
    for (int k = (1 << 30); k > 0; k = k / 2, cur_time++) {
        while ((add = dfs(s, k)) > 0)
            f += add, cur_time++;
    }
    return f;
}
```

Theorem 1.7.1. On graphs with integer capacities scaling algorithm works in time $\mathcal{O}(m^2 \log U)$ where U is the maximum capacity.

Exercise: prove this fact (estimate the remaining flow value after the end of each iteration by noticing the existence of a cut with a small residual capacities).

1.8. Further read

There are some cool and more efficient algorithms:

1. Dinic's algorithm
   a) Finds max flow in $\mathcal{O}(n^2 m)$ or in $\mathcal{O}(nm \log U)$ with capacity scaling.
   b) Provides us with Hopcroft–Karp algorithm for finding matching in a 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" concept.
   The best practical algorithms with running time $O(n^3)$ or $O(n^2\sqrt{m})$.
3. Orlin's algorithm is the theoretically best algorithm working in time $O(nm)$. It is completely impractical but very cool :)

2. Mincost Flows

2.1. Definitions

We have a directed graph with cost function a_e, the cost of the edge e.

Denote the opposite for the edge e as e'. As we already know, $f_e = -f_{e'}$.

Assume that the cost of the opposite edge is equal to $-a_e$. Define the cost of the whole flow f be equal to $a(f) = \sum_{e \in E} f_e a_e$ where E is the set of edges (direct, without opposite ones).

We will also use the notation $a(p)$ for the cost of the path p or $a(z)$ for the cost of circulation z.

Here is the list of three problems that arise with cost flows:

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

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

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

Proof. Let C be the negative cycle in G_f, consider the flow $f + \varepsilon C$ (increase the flow by ε along the cycle).
Value of the flow is the same, although the cost has been decreased. ■

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

Proof. Let there be flow f_1 and flow $f_2\colon |f_1| = |f_2|, a(f_1) > a(f_2)$. Consider the flow $h = f_2 - f_1$ in the network G_{f_1}. It's a correct flow because $(f_2 - f_1)_e \le (c - f_1)_e$. Also notice that $|h| = 0$ and $a(h) < 0$, so it's a negative circulation. Circulation can be decomposed into cycles and one of these cycles has to be of negative cost that contradicts the assumption of the lemma. ■

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

Proof. Consider the flow $h = f_{k+1} - f_k$ in G_{f_k}.

$|h| = 1$, decomposition of h consists of the path p and the circulation z. f_{k+1} is mincost, so $a(z) \le 0$ and $a(p) = a(\texttt{shortestPath})$.
Using lemma 2.2.1 we have $a(z) \ge 0 \Rightarrow a(z) = 0$. ■

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

```
$f \leftarrow 0$ (the zero flow is the minimum 0-flow)
for i=1..k:
    find the shortest path using Bellman-Ford algorithm
    increase the flow along the path by 1
return $f$
```

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

Denote $d_k = a(f_{k+1}) - a(f_k)$ (cost of k-th augumenting path). From the proof of Edmonds-Karp algorithm [2] we have $d_{k+1} \ge d_k$. So we may optimize the algorithm described above by pushing not only 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 that $d_{k+1} \geq d_k$. Weight of the flow decreases while $d_k < 0$.

Algorithm: let's start with f_0 and while $d_k < 0$ add the shortest path to the flow. Stop when $d_k \geq 0$ or doesn not exist.

Working time is $\mathcal{O}(nm|f|)$, where $|f|$ is the value of the desired flow.

2.4. Implementation of mincost flow algorithm

Let's fix the code [1]:

```

void add_edge(int a, int b, int capacity, int cost) {
    edges.push_back({b, first[a], 0, capacity, cost});
    edges.push_back({a, first[b], 0, 0, -cost});
    first[a] = edges.size() - 2;
    first[b] = 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 inf = 1e9;
    for (int i = 0; i < n; i++)
        dist[i] = infty, in_queue[i] = 0;
    relax(s, 0);
    while (!q.empty()) {
        int v = q.front(); q.pop();
        in_queue[v] = 0;
        for (int i = first[v]; i != -1; i = edges[e].next) {
            Edge &e = edges[i];
            if (e.f < e.c)
                relax(e.to, d[v] + e.cost);
        }
    }
    return dist[t] < inf;
}
```

This implementation of Bellman-Ford algorithm uses $\mathcal{O}(n)$ of memory and $\mathcal{O}(nm)$ running time in the worst case. In average case it works pretty fast.

This implementation is known as "Shortest Path Faster Algorithm" (SPFA). It is wrong to call it “Levit's algorithm” since in the Levit's algorithm the vertices are sometimes pushed to the front of the queue.

2.5. Negative cycles

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

This approach gives provides us with another algorithm of finding f_k.

Algorithm for "mincost k-flow" problem

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

This algorithm works in polynomial time if we look for a "cycle of minimal mean cost"

2.6. Further read

Idea of Johnson's potentials allows us to use Dijkstra instead of Bellman-Ford. Using it, the algorithm for mincost k-flow problem works in $\mathcal{O}(km \log n)$ time or even in $\mathcal{O}(k(m + n \log n))$ time.

Note that we do not know yet how to find min-cost flow in polynomial by n, m, $\log U$ and $\log C$ time. Though such algorithms exist, furthermore, there exist algorithms that do not depend on $\log U$ nor $\log C$. Such algorithms exploit the ideas of cost scaling and capacity scaling a bit similar to the capacity scaling for the usual max-flow.