Network flows

Guided tour

Scene: *Maximal flow*

This scene shows a *network*, which is an oriented graph together with its two node, called a *source* (green color) and a *sink* (blue color). Moreover each edge of the graph is associated with a non-negative real number, called a *capacity* of the edge.

Flow in a network is a function t, which associates each edge h with a real number $t(h)$, such that $0 \leq t(h) \leq c(h)$, where $c(h)$ is the capacity of the edge h and, moreover, if v is a node different from the source and the sink, then the sum of flows of all edges starting in v is equal to the sum of flows of all edges finishing in v.

It is possible to imagine that the nodes are cities, edges are sections of railroad or roads, used to transport certain medium from one city (source) into another city (sink), assuming that every road has a limited throughput ans the medium is really transported, not stored, consumed or discarded in any other city than the sink.

It is also possible to imagine that edges are segments of water or oil pipelines, in which case the requirement of zero loss is naturally fulfilled (provided that the tubes do not leak).

Some algorithms allow temporarily an excess or a deficit of the flow in internal nodes of the network. Using analogy with a water pipelines, an excess in a node will be shown as a color column above the node, representing a buffer tank with variable surface level proportional to the difference of the in-flow and the out-flow. If the excess is negative (i.e. there is a deficit), the column will be below the node, representing a well used to cover the deficit.

Within the above restrictions we try to transport the largest amount of the medium from the source to the sink. It is clear that this amount, called *size* of the flow, is equal to the excess of the sink (where the out-flow is usually zero, even though the definition allows a positive out-flow), and is simultaneously equal to the deficit of the source (where there is usually no in-flow). The largest flow in the network shown in the screen can be displayed by clicking the button $\boxed{\text{Result}}$.

The scene allows usual network modification as addition, deletion and mobing a node and additon and deletion of edges and changing their capacity. It is also possible to change a source or a sink and to load one of pre-defined examples.

Two possible ways of network display are possible depending on a choice on a control bar:

- Numerical method: each edge is associated with a number representing the flow through the edge and by checking the checkbox Capacity also its capacity is shown (as a number pair "capacity/flow"); all edges have the same width regardless of their flow and capacity.

- Graphical method: the width os an edge is proportiona to its capacitz a the edge is divideded longitudeally into two stripes, of which the red one hes the width proportional to the flow through the edge and yellow one has the width proportional to the difference of the capacity and the flow and shows how much the flow could be increased without violating the capacity restriction. At the beginning of the computation we assume a zero flow and therefore the edges are completely yellow, while an edge with maximln possible flow is red.

Scene: *Dinitz algorithm*

Dinitz algorithm is in fact an implementation of Ford-Fulkerson algorithm. Ford-Fulkerson algorithm does not determine, which generalized augmenting path is used if there are more than one such path. Dinitz algorithm chooses the path that contains the smallest number of edges (breaking ties arbitrarily). As it will be shown, such a simple trick brings a surprising change of the worst behavior of the algorithm. The algorithm was discovered independently by Edmonds a Karp, but Dinitz algorithm includes a preprocessing that makes the computation even faster.

The basic scheme algorithmu is the following:

Dinitz algorithm
Input: A network
Output: A maximum flow in the input network
Choose an initial flow;
Construct the residual network;
Outer-loop:
    determine layers;
    if there is no path source→sink in the residual network **then goto** finish;
    create a clean network as a copy of the residual network;
    delete all non-accessible edges from the clean network;
    delete all non-forwarding edges from the clean network;
    Inner-loop:
        clean the clean network;
        if the clean network has no edges **then goto** Outer-loop;
        find and process a source→sink path in the clean network;
        goto Inner-loop;
finish: determine flows in edges using their residual capacities.

The zero flow is usually used as an initial flow. The steps of initializing the flow, construction of the residual network and final recomputation of residual capacities to flows have already been discussed in Ford-Fulkerson algorithm. The remaining operation will be shown very briefly in animation. A detailed explanation will be given later (in many cases it consists in application of known principles, as e.g. breadth-first search or processing of an augmenting path in Ford-Fulkerson algorithm).

Observe the computation step-by-step and consult simultaneously the following explanation of the steps of the algorithm.

Step: *Construction of layers*

Using breadth-first search of the residual network starting in the source determines for each node v the minimal number of edges of the residual network that must be traversed to get from the source into the node v. For a node v this number is denoted by $L(v)$. On the display nodes are divided into layers, represented by vertical columns. Each colum consists of all nodes with the same value of the label $L(v)$.

In general it could happen that certain nodes are not reachable by a path of the residual network from the source. Such nodes would be displayed at the right side of the display.

If there is no path in the residual network from the source to the sink, then this fact is found during the search, and the computation ends. As it has already been shown, this means that the maximal flow has been found.

Step: *Deletion of non-accessible edges*

If (u, v) is an edge of the residual network such that $L(u) = \infty$ after the construction of layers, then u can not be accessed from the source by an oriented path in the residual network. Such edges could not be used later to augment the flow and therefore they are deleted from the clean network.

Step: *Deletion of non-forwarding edges*

This step is reached when the search determines (among others) that there is a existuje a path from the source z into the sink s in the residual network. $L(s)$ determines the length of this path. Thus, Dinitz algorithm requires to find an augmenting source→sink path in the residual network that has the minimal length $L(s)$.

It can be found by observing the computation that the number of such paths is usually larger, sometimes very large and many of them are used to augment the flow. It is therefore useful to pre-process the residual network in such a way that it will be easier and faster to look for such paths. The preprocessing, missing in Edmonds-Karp algorithm, makes Dinitz algorithm asymptotically faster in the worst case.

It is clear from the drawing of the network on the display that a minimum length $L(s)$ source→sink augmenting path can not be "delayed" at any layer of the network and each its edge must proceed one layer to the right. An edge with both ends in the same layer or even an edge that is oriented to the left, can not belong to any minimal path.

The non-forwarding edges, mentioned in the previous paragraph, can easily be detected by values L of their end nodes. An edge with an original node v and a terminal node w such that $L(w) \leq L(v)$ is useless. such edges are marked as yellow, the remaining useful edges are blue.

No yellow edge can become a part of a shortest source→sink path in the residual network. The converse is obviously false: there might exist a blue edge that belongs to no shortest source→sink path. We get rid of such edges later.

An implementation of the step is so simple that it will not be analyzed in more detail.

Step: *Cleaning of residual network*

In the previous step we identified yellow edges that are part of no minimum source→sink length path in the residual network. However, certain blue edges should also be deleted, because they lead to a dead-end (a node with no blue out-edges). Such edges are made yellow in the present step. A detailed explanation of the step is given in the forthcomming scenes, in this one only the result is shown (?only uncommented animation is given). Let us remind that a dead-end part of the network could be long and it takes long time to find out that it is traversed by no minimum length path.

Clean the network and then go back to the state at the beginning of the cleaning step; it is clear that in the clean network represented by blue edges *any* path arrives eventually to the sink and can be found with no backtracking. Moreover any source→sink path in the clean network consists of the minimum number of edges.

Step: *Finding and processing of an augmenting path*

Finding and processing of an augmenting path in the clean network is now very simple, we go forward using blue edges and we eventually arrive to the sink. Processing of the path is the same as in Ford-Fulkerson algorithm: a minimal residual capacity of the path is determined and the residual capacities of edges of the path are decreased by this value and the residual capacities of the opposite edges are increased by the same value. Thus, the set of edges of the residual network changes in the following two ways:

- the residual capacity of at least one edge of the path *drops to zero* and hence such an edge is removed from the residual network;
- the residual capacity of any edge, which is the opposite edge of an edge of the path, increases; if such an edge did not belong to the residual network, it becomes a part of it after processing of the path.

In order to execute the path processing, the button Step must be pressed three times. The path is marked green after the first click. Tha second click shows changes of the residual network:

- edges of the path with smaller but non-zero residual capacity after processing (that remain in the residual network) remain green,
- edges of the path with zero residual capacity after processing (that are deleted from the residual network) will be white and unframed, and
- edges with larger residual capacity after processing are displayed in two shades of red; edges that have already been in the residual network will be light red, while edges that previously had a zero residual capacity will be dark red.

Let us now look how the changes of the residual network influence the clean network. Red edges, regardless of their pervious presence or absence in the residual network, do not belong to the clean network, because they are oriented backwards, in the direction from the sink to the source. No edge is therefore added into the clean network. On the other hand, a white edge (or edges) with the residual capacity that was decreased to zero, is not any more a member of the residual network, and therefore it should be deleted from the clean network as well.

Click the button $\boxed{\text{Step}}$ again to see a new clean network (blue edges) and the residual network (blue and yellow edges). However, the true clean network is often much smaller, because removal of saturated (white) edges could create dead-ends. It is therefore necessary to apply the cleaning operation again to remove them and only then a new path can be searched for.

Step: *Finishing*

....

Scene: *Why to choose short paths*

Repeat the computation once again; steps are longer than in the previous scene, because path processing is done in one step. The computation is divided to *phases*, which are determined by (re)entry to the label "Outerloop" in the pseudocode of the previous scene.

It is clear that any minimum length source→sink path in the residual network is also a path in the clean network represented by blue edges.

During any phase the shortest source→sink path in the residual network has the same length, but the number of paths of this length *strictly decreases* and *no new path* of this length or shorter appears. The phase ends when no path of the length considered in the phase remain in the residual network. Hence when a phase ends, the length of the shortest source→sink path in the residual network is *increased*.

Perform step-by-step computation again to verify this statement. The phase number and the length of the shortest source→sink path in the residual network is given in the corner of the display.

It is clear from the above considerations that the number of phases, i.e. the number of repetitions of the outer loop can not be greater than $n-1$, where n is the number of the network nodes. This is because a path from the source to the sink can not involve more than $n-1$ edges, but it has at leas one edge. Then it is sufficient to realize that the length of the shortest path can only have values $1, \ldots, n-1$, and it increases by 1 at any phase.

Scene: *Why not to choose long paths*

This scene is a short sequence of three steps to show what could happen if the flow is not augmented using the shortest path in the residual network.

The first snapshot is a network in certain moment of the computation of Dinitzova algorithm, blue edges represent the clean network and, together with yellow edges, the residual network. The shortest path from the source to the sink in the residual network (and therefore in the clean network as well) has length 5. We are in the moment when a new path is to be found and processed.

Click of the button $\boxed{\text{Step}}$ selects a path from the source to the sink, but not according to Dinitz algorithm, but a path in the residual network, which has more than minimum number of edges. The path necessarily involves yellow edges that belong to the residual network, but are not members of the clean network. Note that Ford-Fulkerson algorithm admit such a choice of a path. The next steps are color representation of path processing in the same way as in the previous scene.

In one of the processing steps we will see that one of the path edges, green or white, goes back to the layer that is closer to the source. As a result, the opposite edge (red) has forward direction to the sink. In contrary to Dinitz algorithm a new forward edge could appear in the residual network. The situation in this scene is even such that such an edge skips over several layers and creates a new path in the residual network that is even *shorter* than the previous minimal length. Thus, if the choice of the path is not careful enough, the evolution of the length of the shortest path in not monotone as it is in Dinotz algorithm and this is why Ford-Fulkerson algorithm can re-create paths that have already been processed and to work as long as it was shown above.

Scene: *Why not to choose long paths - cont'd*

The samr scene as the previous one, but a path is chosen by the used. our task is to find a path such that its processing creates a new path source-to-sink path in the residual network, which is as long as the present shortest path, or even shorter.

Scene: *Layer determination*

This scene makes it possible to create layers of the network step-by-step. It will not be commented in detail, because it consists of breadth-first search that has already been explained in previous chapters.

Scene: *Cleaning the network*

Cleaning the residual network to obtain a clean network is routine operation which, however, is not completely trivial. It is convenient to perform it as described below/ You can go through it step-by-step, while the other parts of the computation are skipped over.

The first part of the cleaning determines the in- and out-degrees of vertices and then determines dead-ends that will be stored in queues.

Step: *Initialization of degree computation*

For each node v two variables $d(v)_{in}$ and $d(v)_{out}$ are created and initialized to 0. Later the variable will become equal to the in- and out-degree of v. Two empty queues Q_{in} and Q_{out} are created.

Step: *Degree computation*

For each blue edge with the original node v and the terminal node w the values $d(v)_{out}$ and $d(w)_{in}$ are both incremented by 1.

Step: *Queue initialization*

For all nodes v if v is not the sink and $d(v)_{out} = 0$, then v is put into the queue Q_{out}, and if v is not the source and $d(v)_{in} = 0$, then v is put into the queue Q_{in}.

After these steps all dead-end nodes are in the queues. Now all edges originating or terminating in dead-ends are removed. Since thic could create other dead-ends, the operation must generally be repeated many times.

Step: *Removal all no-way-out nodes*

While Q_{out} is non-empty, perform the following: pop a node v of the queue Q_{out} and for all edges (w, v) comming into v execute the following: the edge (w, v) is deleted from the clean network (and re-colored to yellow), $d_{out}(w)$ is decremented by 1 and if the degree $d_{out}(w)$ drops to 0, put w into the queue Q_{out}.

In the same way Q_{in} is processed:

Step: *Removal all no-way-in nodes*

While Q_{in} is non-empty, perform the following: pop a node v of the queue Q_{in} and for all edges (v, w) starting in v execute the following: the edge (v, w) is deleted from the clean network (and re-colored to yellow), $d_{in}(w)$ is decremented by 1 and if the degree $d_{in}(w)$ drops to 0, put w into the queue Q_{in}.

During the animation nodes of the queues Q_{in} and Q_{out} are black. Edges considered during dead-end removal are first marked as red (simultaneously for each considered node) and then removed from the clean network (i.e. re-colored to yellow). When both queues are empty, all dead ends removed.

At the end of each phase it happens that the clean network is cleaned so well that noblue edge remains. This indicates that all paths of the given length were processed, while longer paths were ignored during the phase. In order to build a new clean network, the residual network must be processed again from scratch and create new clean network by repeated cleaning.

A clever reader surely noticed that processing of the queue Q_{in} is not necessary: dead-end must be deleted to avoid possibility to reach them while looking for a source-to-sink path starting from the source. However, members of the queue Q_{in} are nodes tha could not be reached during such search, because they possess no in-edges. Thus, such nodes could not create problems and there is no deed to loose time by their removal.

Operations using Q_{in} are used in the applet to make the network more "clean". Since processing of the queue Q_{in} is quite analogous to the queue Q_{out}, it does not represent extra difficulties to the reader. However, the code presented in the laboratory does not involve cleaning of Q_{in}.

Scene: *Finding and processing of the path*

This scene just repeats a similar scene of Ford-Fulkerson algorithm. It is presented just to make it clear that it could not happen that depth-first search of a parh would backtrack. Each new node brings the path closer to the sink by one layer and hence (unlike in Ford-Fulkerson algorithm, but also in Edmonds and Karp algorithm) the time necessary to find a path is (even in the worst case) proportional to the path length (and not to the number of edges of the network).

Laboratory

An input for all algorithms in this chapter is a *network*, i.e. an oriented graph $G = (V, E)$ together with two distinct vertices from the set V - a *source* s and a *sink* t and a function $c : E \rightarrow < 0, \infty)$, called a *capacity*.

It is assumed that $(u, v) \in E$ implies $(v, u) \in E$. If this condition is not fulfilled, missing opposite edges will be inserted into E and $c(e) = 0$ for any such added edge. If $e = (u, v) \in E$, then the edge (v, u) will be denoted by e^{op}.

Given any real-valued function ϕ defined on E and $e = (u, v) \in E$, we will often write $\phi(u, v)$ instead of $\phi(e)$ or $\phi((u, v))$. For any such function and any vertex $u \in E$ we will define

$$EXC_G(\phi, u) = \sum_{v,(v,u)\in E} \phi(v, u) - \sum_{w,(u,w)\in E} \phi(u, w).$$

If there is no danger of misunderstanding, a subscript G is often ommited. $EXC_G(\phi, u)$ is called an *excess* of ϕ in the vertex u.

A *pre-flow* in a network $(G = (V, E), s, t, c)$ is a function $f : c \rightarrow < 0, \infty)$ such that

1. $f(e) \leq c(e)$ for each edge $e \in E$;
2. $EXC_G(f, u) = 0$ for any vertex u different from the source s and the sink t;
3. $f(e) > 0$ implies $f(e^{op}) = 0$ for any edge $e \in E$.

The *size* of the pre-flow f is $EXC_G(f,t)$.

A *flow* is a pre-flow f such that $EXC_G(f,u) = 0$ for any u different from the source s and the sink t. Note that a flow is a special case of a pre-flow.

Given a pre-flow f in a network $(G = (V,E), s, t, c)$, a residual capacity of an edge $e \in E$ with respect to f is

$$R(e) = c(e) - f(e) + f(e^{op}).$$

Note that a residual capacity of any edge with respect to any pre-flow is non-negative.

All our algorithms will start with a zero flow, defined by $f(e) = 0$ for each $e \in E$. However, if any other flow is known, it can be used as an initial flow. We will use the following procedure

Initialize a flow f
for each $e \in E$ **do** $f(e) = 0$;

Instead of working with flow, our algorithms will be formulated to work in a *residual network*. Given a network $(G = (V,E), s, t, c)$ and a flow f, the corresponding residual network is a graph $G' = (V, E')$, where E' is the set of all edges $e \in E$ such that their residual capacity $R(e) = c(e) - f(e) + f(e^{op})$ is positive. (Note that even en edge with zero capacity can have positive residual capacity.) Edges of the residual network are labeled by their residual capacities. We will use the following procedure

Create a residual network
    over a network $(G = (V,E), s, t, c)$
    with respect to a flow f
$E' := \emptyset$;
for each $e \in E$ **do begin**
    $r(e) := c(e) - f(e) + f(e^{op})$;
    if $r(e) > 0$ **then** insert e into E';
    end;

When working with residual capacities, we have to recover the corresponding values of the flow. This is done by the following procedure

Restore the flow
for each $e \in E$ **do**
    if $c(e) \geq r(e)$ **then**$f(e) := c(e) - r(e)$ **else** $f(e) := 0$;

In Goldberg algorithm we will work with pre-flows and we will need to know excess of any node. Since the computing of excess is time consuming, we will use variables $exc(u)$, $u \in V$ that will updated in such a way that $exc(u)$ will alwas be equal to the excess $EXC(u)$. Note, however, that EXC and exc are two distinct entities and it will be necessary to prove that they have always the same values. The operation of setting the initial zero flow can be extended to compute initial values of exc as follows:

Initiate exc
for each $u \in V$ **do** $exc(u) = 0$;

Any amendment (augmentation) of the flow will be done by the following operation

Propagate excess $\Delta > 0$ along an edge $e = (u, v) \in E'$
Condition of applicability: $\Delta \leq r(e)$
decrease $r(e)$ by Δ; **if** $r(e) = 0$ **then** remove e from E';
increase $r(e^{op})$ by Δ; **if** $e \notin E'$ **then** insert e into E';
decrease $exc(u)$ by Δ;
increase $exc(v)$ by Δ;

If an information about values of excess of nodes is not necessary, two statement operating with the exc function could be omitted.

In this way Ford-Fulkerson algorithm can be formulated in the following way

Initiate a flow;
create a residual network;
loop:
    find a path $s = u_0, \ldots, u_k = t$ in the residual network;
    if no such path exists **then goto** *finish*;
    $\Delta := min_{i=1,\ldots,k} r(u_{i-1}, u_i)$;
    for $i = 1, \ldots, k$ **do** propagate excess Δ along (u_{i-1}, u_i);
    goto *loop*;
finish: restore the flow;

Dinitz algorithm is a special implementation of Ford-Fulkerson algorithm. It uses a *clean* network (V, E''), which is a subnetwork of the residual network, i.e. $E'' \subset E'$.

Input: A network $((V, E), s, t, c)$
Output: A maximal flow f
Variables: integer-valued functions *layer*, *outdeg* defined on the set V of vertices;
E', the set of edges of the residual network;
E'', the set of edges of the clean network;

begin
initiate a flow;
create a residual network;
outer-loop:
  create layers;
  if there is no source-to-sink path in (V, E') **then goto** *finish*;
  $E'' := E'$;
  initialize the clean network;
compute degrees in the clean network;
  inner-loop:
    clean E'';
    if $outdeg(s) = 0$ **then goto** *outer-loop*;
    find path;
    process path;
    goto *inner-loop*;
finish: restore the flow;
end.

The procedures of initiating a flow, creating a residual network and restoring the flow have already been explained, the remaining ones follow:

Creating layers
Variables: a FIFO queue Q of nodes;
begin
$Q := \{s\}$;
$layer(s) := 0$;
while $Q \neq \emptyset$ **do begin**
  $v :=$ the first element of the queue Q;
  for each edge $(v, w) \in E'$ starting in v **do**
  if $layer(w)$ is undefined **then** $layer(w) := layer(v) + 1$;
  end;
if $layer(t)$ is undefined **then** there is no source-to-sink path in (V, E');
end.

Initializing the clean network
begin
$E'' := \emptyset$;
for each edge $(u, v) \in E'$ **do if** $layer(u) < layer(v)$ **then** put (u, v) into E'';
end.

Computing degrees in the clean network
begin
for each vertex $v \in V$ **do** $outdeg(v) := 0$;
for each edge $(u, v) \in E'$ **do** increment $outdeg(u)$ by 1;
end.

Cleaning E''
Variables: set S of nodes;
begin
$S := 0$;
for each vertex $v \in V$ **do if** $outdeg(v) = 0$ **then** put v into S;
while $S \neq \emptyset$ **do begin**
    choose a vertex $v \in S$ and remove it from S;
    for each edge $(u, v) \in E''$ **do begin**
        remove (u, v) from E'';
        decrease $outdeg(u)$ by 1;
        if $outdeg(u) = 0$ **then** insert u into S;
        end;
    end;
end.

Finding a path in the clean network
Variables: a path P in the clean network
    u, the last vertex of the path P;
begin
$P = (s)$; $u = s$;
while $u \neq t$ **do begin**
    choose an edge $(u, w) \in E''$;
    append w to the path P; $u = v$;
    end;
end.

Process a path $s = u_0, \ldots, u_k = t$
Variables: a number Δ
begin
$\Delta = min_{i=1,\ldots,k} r(u_{i-1}, u_i)$;
for $i = 1, \ldots, k$ **do** propagate excess Δ along the edge (u_{i-1}, u_i);
end.

School

`Tato cast je znacne nedodelana !!!`

The goal of this section is to prove formally properties Dinitz algorithm. We use definitions and algorithms as introduced in the laboratory.

Let us repeat that the input is a network (G, s, t, c), where $G = (V, E)$, in all cases and we want to find a flow f in the network that maximizes $EXC_G(f, t)$ among all flows in the network.

Let us repeat that we assume $e = (u, v)$ is an edge of G if and only if $e^{op}(v, u)$ is an edge of G and the residual network for a given flow f is the graph (V, E_f), where E_f is the set of all edges $e \in E$ such that their residual capacity $r(e) = c(e) - f(e) + f(e^{op})$ is positive. $L(v)$ is the distance (the number of edges of the shortest path) from the source into v, as determined during the operation of layer construction.

We will first present several properties of the clean network.

Lemma 1 *After the operation of removal of non-forwarding edges, if (u, v) is an edge of the clean network, then both $L(u)$ and $L(v)$ are finite and $L(u) + 1 = L(v)$.*

Proof: Inaccessible edges, i.e. edges such that $L(u) = \infty$, are deleted from the clean network before the removal of non-forwarding edges.

$L(u) + 1 \geq L(v)$ for each edge of the residual network, because if $L(u)$ is finite, it is accessed during the layer construction, and when u is processed, $L(v)$ is either smaller than $L(u) + 1$ or it is set to this value. Moreover, $L(v)$ is never increased during the remaining part of the layer construction.

Edges that do not fulfill the condition are removed during removal of non-forwarding edges. ♣

Lemma 2 *Any path from the source to a node u in the clean network involves $L(u)$ edges.*

Proof: By induction on the number of edges of a path: the lemma is true for a path with no edges and starting and ending in the source. If the lemma is true for path with less than k edges and $s = u_0, u_1, \ldots, u_{k-1}, u_k$ is a path in the clean network starting in the source u_0, then $L(u_{k-1}) = k-1$ from the induction assumption, and $L(u_k) = L(u_{k-1}) + 1 = (k-1)+1 = k$ in view of the previous lemma. ♣

Lemma 3 *Any path from the source s to the sink t in the clean network has $L(t)$ edges.*

Proof: Follows immediately from the previous lemma. ♣

Lemma 4 *At any moment during execution of the inner loop and for each node u, $d(u)$ is the number of the edges of the clean network that start in u.*

Proof: When values $d(u)$ are initialized just before entering the inner loop, this property obviously holds. Later any removal of an edge (u, v) of the clean network is associated with decrementing $d(u)$ by one. ♣

Lemma 5 *After cleaning, if (u, v) is an edge of the clean network after cleaning, then either v is the sink or $d(v) > 0$ and there is an edge of the clean network starting in v.*

Proof: If, during cleaning, (u, v) is an edge of the clean network such that $d(v) = 0$, then v is put into the queue and the edge is later removed from the clean network. ♣

Lemma 6 *After cleaning, if v is a node such that $L(v) \geq L(t)$, then either v is the sink or there is no node u such that (u, v) is an edge of the clean network.*

Proof: Assume that $v \neq t$, $L(v) \geq L(t)$, but that, after cleaning, there is an edge (v, v_1) in the clean network. Let $v = v_0, v_1, \ldots, v_\ell$ be the longest path in the clean network starting in v. Such a path exists, because v_0, v_1 is one such path and Lemma ? implies that $L(v_0) < L(v_1) < \cdots < L(v_\ell)$ and therefore any node occurs at most once in the path and since the graph is finite, there is only finitely many such paths.

The maximality of the path means that there is no edge of the clean network starting in v_ℓ. ??? ♣

Lemma 7 *After cleaning, any path $s = u_0, u_1, \ldots, u_{k-1}, u_k$ in the clean network can be completed to a path source→sink path $s = u_0, u_1, \ldots, u_{k-1}, u_k, \ldots, u_\ell = t$ in the clean network.*

Proof: By induction on k. If $k = L(s)$, then $L(u_k) = L(s)$ and hence u_k is the sink.

Let $k < L(s)$ and the lemma is true for longer paths. It follows from Lemma ? that $L(u_k) = k < L(s)$ and therefore u_k is not the sink. In view of Lemma ? there is a node v such that (u_k, v) is an edge of the clean network. Now, it follows from the induction hypothesis that the path $s = u_0, u_1, \ldots, u_{k-1}, u_k, v$ is extendible to a path from the source to the sink. ♣

The lemmae immediately imply that the above described procedure for finding a path can always find an edge (u, v) of the clean network to extend the path and it will eventually arrive to the sink, creating a path of the length $L(t)$, while no shorter path from the source to the sink exists not only in the clean network.

`Nasledujici veta je nejdulezitejsi z cele analyzy Dinitzova algoritmu; jeji dukaz je udelan prilis strucne, ale jiste si budete umet detaily doplnit.`

Lemma 8 *Each iteration of the inner loop destroys at least one source→sink edge of the length $L(t)$ in the clean network and does not create any path in the residual network of the length $L(t)$ or shorter.*

Proof: When processing a path, the edge with the smallest residual capacity among edges of the path will be (in general temporarily) deleted from the residual network, because its residual capacity drops to zero after processing. Since each edge of the clean network belong to a path from the source to the sink in the residual network, and each such path has $L(t)$ edges, at least one path of the length $L(t)$ in the clean network is destroyed.

If a path $s = u_0, u_1, \ldots, u_k = t$ is processed, certain edges could appear in the residual network. Such edges have had their residual capacity equal to 0, and during their processing the residual capacity increased.

Only edges, for which the residual capacity increases are edges that are opposite to the edges of the processed path, i.e. the edges $(u_k, u_{k-1}), \ldots, (u_1, u_0)$. However, if $0 < i \geq k$, then during processing the length of the shortest path from the source to u_i is $L(u_i)$, the length of the shortest path from the source to u_{i-1} is $L(u_{i-1}) = L(u_i) - 1$ and therefore the length of the shortest path from u_{i-1} to the sink is $L(t) - L(u_{i-1}) = L(t) - L(u_i) + 1$. As a consequence, any path including (u_i, u_{i-1}) in the residual network has length at least $L(u_i) + 1 + (L(t) - L(u_{i-1})) = L(t) + 2$ and therefore no new edges of the residual network can belong to a path of the length $L(t)$ or smaller in the residual network. ♣

Theorem 9 *If the network has N nodes and M edges, then the outer loop of the Dinitz algorithm is executed at most N times, and, during one execution of the body of the outer loop, the inner loop is executed at most M times.*

Proof: As it follows from the previous lemma, during execution of the outer loop no new path of the length $L(t)$ appears in the residual network. It follows that the distance from the source to the sink *increases* from one execution of the body of the outer loop to the next one. Since $L(t)$ is a positive integer smaller to $N-1$, the outer loop is not executed more than N times.

Moreover, each iteration of the inner loop removes at least one edge from the clean network and adds no edge into the clean network. Therefore after at most M steps the clean network becomes empty and a jump to the beginning of the outer loop is forced. ♣