Dinitz' Algorithm: The Original Version and Even's Version

Yefim Dinitz

Dept. of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel
dinitz@cs.bgu.ac.il

Abstract. This paper is devoted to the max-flow algorithm of the author: to its original version, which turned out to be unknown to non-Russian readers, and to its modification created by Shimon Even and Alon Itai; the latter became known worldwide as "Dinic's algorithm". It also presents the origins of the Soviet school of Algorithms, which remain unknown to the Western Computer Science community, and the substantial influence of Shimon Even on the fortunes of this algorithm and its author.

1 Introduction

The reader may be aware of the so called "Dinic's algorithm" [4][1], which is one of the first (strongly) polynomial max-flow algorithms, while being both one of the easiest to implement and the fastest in practice. This introduction discusses two essential impacts on its fortune: one which supported its invention, and another which made it famous, though changing it partly.

The impact of the late Shimon Even dates back to 1975. You may ask how a person in Israel could influence something in the former USSR, when it was almost impossible both to travel abroad from the USSR and to publish abroad or even to communicate with the West? This question may be cleared up by considering the major impact made by the early Soviet school of computing and algorithms.

The following anecdote sheds some light on how things were done in the USSR. Shortly after the "iron curtain" fell in 1990, an American and a Russian, who had both worked on the development of weapons, met. The American asked: "When you developed the Bomb, how were you able to perform such an enormous amount of computing with your weak computers?". The Russian responded: "We used better algorithms."

This was really so. Russia had an old tradition of excellence in Mathematics. Besides, the usual Soviet method for attacking hard problems was to combine pressure from the authorities with people's enthusiasm. When Stalin decided to develop the Bomb, many bright mathematicians, e.g., Izrail Gelfand and my first Math teacher, Alexander Kronrod, put aside their mathematical studies and delved deeply into the novel area of computing. They have collected and created teams of talented people, and succeeded. The teams continued to grow and work on the theory and practice of computing.

The supervisor of my M.Sc. thesis was George Adel'son-Vel'sky, one of the fathers of Computer Science. Among the students in his group at that time were M. Kronrod (one of the future "Four Russians", i.e. the four authors of [3]), A. Karzanov (the future author of

[1] In this paper, the references are given in the chronological order.

the $O(n^3)$ network flow algorithm [9]) and other talented school pupils of A. Kronrod. This was in 1968, long after the Bomb project had been completed. The work on the foundations of the chess program "Kaissa", created by members of A. Kronrod's team under the guidance of Adel'son-Vel'sky, was almost finished; "Kaissa" won the first world championship in 1974. Adel'son-Vel'sky's new passion became discrete algorithms, which he felt had a great future.

The fundamental contribution of Adel'son-Vel'sky to Computer Science was AVL-trees. He (AV) and Eugene Landis (L) published the paper [1] about AVL-trees, which consists of just a few pages. Besides solving an important problem, it presented a bright approach to data structure maintenance. While this approach became standard in the USSR, it was still unknown in the West. No reaction immediately followed their publication, until several years later another paper, 15 pages long, was published by a researcher (unknown to me), who understood how AVL-trees work and explained this to the Western community, in its own language. Since then, AVL-trees and the data structure maintenance approach became corner-stones of Computer Science.

We, Adel'son-Vel'sky's students, absorbed the whole paradigm of the Soviet computing school from his lectures. This paradigm consisted of eagerness to develop economical algorithms based on the deep investigation of a problem and on the use of data structure maintenance and amortized running time analysis as necessary components. All this became quite natural for us in 1968, seventeen years before the first publication in the West on amortized analysis by R. Tarjan [20]. Hence, it was not surprising that my network flow algorithm, invented in January 1969, improved the Ford&Fulkerson algorithm by using and maintaining a layered network data structure and employing a delicate amortized analysis of the running time.

However then, such an approach was very unusual in the West. Shimon Even and (his then Ph.D. student) Alon Itai at the Technion (Haifa) were very curious and intrigued by the two new network flow algorithms: mine [4] and that of Alexander Karzanov [9] in 1974. It was very difficult for them to decipher these two papers (each compressed into four pages, to meet the page restriction of the prestigious journal Doklady). However, Shimon Even was not used to give up. After a three-day long effort, Even and Itai understood both papers, except for the layered network maintenance issue. The gaps were spanned by using Karzanov's concept of blocking flow (which was implicit in my paper), and by a beautiful application of DFS for finding each augmenting path.

It is well known that Shimon Even was an excellent lecturer. During the next couple of years, Even presented "Dinic's algorithm" in lectures, which he gave in many leading research universities of the West. The result was important, the idea was fresh, the algorithm was very nice, and the combination of BFS for constructing the layered network and DFS for operating it was fascinating. "Dinic's algorithm" was a great success, and gained a place in the annals of the Computer Science community. Hardly anyone was aware that the algorithm, taught in many universities since then, is not the original version, and that a considerable part of the beauty of its known version—combining BFS and DFS—was due to the contribution of Even and Itai. Also, its name was rendered incorrectly as [dinik] instead of [dinits].

The original paper, published in a Soviet journal, was not understood also by others in the West. This algorithm and many other achievements of the Moscow school of algorithms in the network flow area were published as a book [10]. This book was well known all over the former USSR, and students of many leading Soviet Universities studied its contents as an advanced course. After more than 15 years, the book appeared in the West (in Russian)

and was reviewed in English by A. Goldberg and D. Gusfield [24]. A few years later, I finally explained the original version of my algorithm, as published in [4], to Shimon Even.

The rest of the paper is devoted to the description of my algorithm and its various versions and to the research on max-flow algorithms following it. The presentation is a bit didactic; it shows that serious and devoted work towards maximal understanding of a phenomenon and the best implementation of algorithms may bring important, unexpected fruits.

2 The Original Dinitz' Algorithm

2.1 The Max-Flow Problem and the Ford&Fulkerson Algorithm

Max-Flow Problem Let us recall the max-flow problem definition. A capacitated directed graph $G = (V, E, c)$, where c is a non-negative capacity function on edges, with vertices s distinguished as the source and t as the sink is given. A flow is defined as a function on the (directed) edges, f, satisfying the following two laws:

- *Capacity constraint*: $\forall e \in E : 0 \leq f(e) \leq c(e)$, and
- *Flow conservation*: $\forall v \in V \setminus \{s, t\} : \sum_{(v,u)\in E} f(v, u) - \sum_{(u,v)\in E} f(u, v) = 0$.

The quantity $\sum_{(v,u)\in E} f(v, u) - \sum_{(u,v)\in E} f(u, v)$ is called the net flow from v. The value of a flow f is defined as the net flow from s (which is equal to the negated net flow from t). The task is to find a flow of the maximal value, called "maximum flow". Initially, some feasible flow function, f_0, is given, which is a zero function by default.

Another equivalent is formed by considering a flow as a *skew-symmetric* function on the pairs of vertices connected by at least one edge in the given graph. For such a pair v, u, if one of the edges between them is absent, in E, we add it to E with capacity zero (clearly, the problem remains equivalent). Given any feasible flow function, f, we define the new function $\bar{f}$ by setting $\bar{f}(v, u) = f(v, u) - f(u, v)$ for any edge (v, u) in the (extended) E. This represents the total flow from v to u on the pair of edges (v, u) and (u, v) together. It is easy to see that $\bar{f}$ satisfies the following three constraints:

- *Capacity constraint*: $\forall (v, u) \in E : \bar{f}(v, u) \leq c(v, u)$,
- *Skew symmetry*: $\forall (v, u) \in E : \bar{f}(u, v) = -\bar{f}(v, u)$, and
- *Flow conservation*: $\forall v \in V \setminus \{s, t\} : \sum_{(v,u)\in E} \bar{f}(v, u) = 0$.

The net flow from v is defined then as $\sum_{(v,u)\in E} \bar{f}(v, u)$. The reverse transformation, from any flow function $\bar{f}$ in the skew-symmetric form, is attained by the formula $f(e) = \max(0, \bar{f}(e))$. It results in a feasible flow function according to the original definition. Both transformations keep the flow value. Hence, using any one of them may be considered legal.

We say that an edge (v, u) is *saturated*, if $\bar{f}(v, u)$ is equal to $c(v, u)$. We call the difference $c(v, u) - \bar{f}(v, u)$ the *current* or *residual* capacity of the edge (v, u), and denote it by $c_f(v, u)$. In the original flow definition terms, the property of being saturated is equivalent to the combined property: $f(v, u) = c(v, u)$ & $f(u, v) = 0$, while the residual capacity is equal to $c(v, u) - f(v, u) + f(u, v)$. The meaning of residual capacity is "how much may be added to the flow from v to u on edges (v, u) and (u, v) together"; the formulae use the essential equivalence between increasing the flow in (v, u) and decreasing it in (u, v). Both flow representations are

widely known; while Ford and Fulkerson introduced the first one in [2], e.g. a popular textbook [22] uses the second one.

In what follows, we relate to flows in their skew-symmetric form, where the concepts of saturated edge and residual capacity are easier to handle. Nevertheless, for simplicity, we denote flows by regular Latin letters, e.g. f, not $\bar{f}$, and in explanations, we relate to $f(v,u)$ as the flow on edge (v,u).

Ford&Fulkerson's Algorithm The classic Ford&Fulkerson algorithm [2] finds a maximum flow using the following natural idea: augmenting the current flow iteratively, by pushing an additional amount of flow on a single source-sink path at each iteration, as long as possible. Let f denote the current flow. A path from s to t is sought, such that none of its edges is saturated by f; such a path is called "augmenting". When an augmenting path, P, is found, its current capacity $\epsilon(P) = \min_{e \in P} c_f(e) > 0$ is computed. Then, the flow on every edge of P is increased by $\epsilon(P)$ (of course, the corresponding update of the skew-symmetric flow values at the opposite edges—decreasing them by $\epsilon(P)$—is made as well; we will not mention this, in what follows). As a result of this iteration, the flow value grows by $\epsilon(P)$ and at least one edge of P becomes saturated.

For convenience sake, let us define the *residual network* $G_f = (V, E_f)$, where E_f consists of all unsaturated edges in E. Now, an augmenting path is simply any path in G_f from s to t. Clearly, it is easy in finding an augmenting path, given G and f: just run any search algorithm (e.g. BFS or DFS) on G_f, from s. It is easy to see that, after each iteration, using an augmented path P, the change of the residual network is as follows: it loses all the edges of P which became saturated and acquires all the edges *opposite* to the edges of P that were saturated before the iteration.

The famous theorem of Ford and Fulkerson states that when no augmenting path exists (i.e. t is disconnected from s in G_f), the current flow is maximum. Clearly, if the initial data—capacities and the initial flow—is integer, the current flow remains integer and eventually becomes maximum. That is, the Ford&Fulkerson algorithm (henceforth, *FF*) is finite and pseudo-polynomial in the integer case. For the general case, Ford and Fulkerson provided an example of a network with an execution of FF in it which runs infinitely; moreover, the flow value converges at a *quarter* of the maximal possible one. So, the question of the existence of a polynomial, or even of a finite or converging algorithm, for the general case, remained open. This was settled affirmatively by J. Edmonds and R.M. Karp [5] and by Y. Dinitz [4], independently. The algorithms suggested in these papers are modifications of FF.

2.2 Layered Network Data Structure for Accelerating Iterations

Layered Network The initial intention of [4] was just to accelerate iterations of FF by means of a smart data structure. Recall that all the parts of an iteration of FF, except for finding an augmenting path, P, cost time proportional to the length of P, that is $O(|P|) = O(|V|)$. Indeed, all computations and updates are made along P, while the remainder of the data is not touched. However, the search running on G costs $O(|E|)$ (one may even say $\Theta(|E|)$, according to the search algorithms). Notice that $O(|E|)$ is substantially more than $O(|P|)$ or $O(|V|)$, and is $O(|V|^2)$ in the general case. So, finding an augmenting path is the computational time bottleneck of an iteration of FF.

We will use BFS as the search algorithm. Let us try to save the information achieved at a BFS run for the following iterations. Notice that at least one edge of the BFS tree, lying on the path from s to t, is saturated at the iteration, so it must be erased from the tree. Thus, s and t become disconnected in this tree, and there are no easy means to connect them again using unsaturated edges.

Let us try to enrich the data structure, while building it. Recall that BFS arranges vertices into layers, according to their distance (number of edges in the shortest path) from s. The BFS tree includes only the *first edge found*, leading to each vertex from the previous layer; it ignores all other edges coming to it from that layer, though they may be no less useful than that first edge. The idea is to keep *all those edges* in our data structure.

We begin from some definitions for an arbitrary digraph H. Let a source vertex s be given. Let $dist(v,u)$ denote the distance to a vertex u from vertex v; we denote $dist(v) = dist(s,v)$. Let the *ith vertex layer* V_i be the set of vertices with $dist(v) = i$ (where $V_0 = \{s\}$), and the *ith edge layer* E_i be the set of all the edges of H going from V_{i-1} to V_i. We define $L(s)$ as the digraph $(\cup V_i, \cup E_i)$. Notice that there is a straightforward extension of BFS building $L(s)$, with the same running time $O(|E|)$ as that of the regular BFS (in what follows "the extended BFS"). It is easy to see, by the properties of BFS, that $L(s)$ *is the union of all the vertices and edges of all shortest paths from s in H.*

Since we are interested in the paths from s to another given vertex, t, let us prune $L(s)$ to $\hat{L}(s,t)$, following [4]. It contains ℓ layers, where $\ell = dist(s,t)$ is called the *length* of $\hat{L}(s,t)$. The vertices of its ith layer, $\hat{V}_i$, are characterized by a double property: they are at distance i from s, while t is at distance $\ell - i$ from them. The ith layer of its edges, $\hat{E}_i$, consists of all the edges of H going from $\hat{V}_{i-1}$ to $\hat{V}_i$. The pruning of $L(s)$ to $\hat{L}(s,t)$ is easy: we just run the extended BFS once more, but *on $L(s)$ from t, using the opposite edge direction.*

Claim. The layered sub-graph $\hat{L}(s,t)$ is the union of all the vertices and edges residing on all the shortest paths from s to t.

Proof. For any shortest path from s to t, its ith vertex is at distance i from s, while t is at distance $\ell - i$ from it. Conversely, for any vertex in $\hat{V}_i$, there is a path from s to v of length i and a path from v to t of length $\ell - i$; their concatenation is a shortest path from s to t. Similar reasons suffice for the edges of $\hat{E}_i$. □

An easy property of $L(s)$ and $\hat{L}(s,t)$ is that they have no "dead-ends": vertices without any incoming edge, except for s (for both $L(s)$ and $\hat{L}(s,t)$) and those with no outgoing edge, except for t (for $\hat{L}(s,t)$ only). Due to the layered structure and to this property, finding a shortest path from s to t, given any one of $L(s)$ and $\hat{L}(s,t)$, is almost as simple and exactly as fast (in time linear in its length) as using the BFS tree, by means of the following procedure:

***PathFinding*:** Beginning from t, we choose one of its incoming edges and go to its tail, then we choose an edge incoming to the tail, and so on; in ℓ steps from layer to layer, we arrive at layer 0, that is, at s. The path consisting of the chosen edges (in the reverse order), is a shortest path from s to t.

Remark: Notice that given $\hat{L}(s,t)$, a similar procedure executed from s, in the direction of the edges, constructs a path from it to t as well, but in its natural order.

Layered Network Maintenance Suppose that, given a flow network G and a flow f in it, we had built the sub-graph $\hat{L}(s,t)$ of the residual network G_f (by two extended BFSs). Having found an augmenting path (by $PathFinding$), we pushed the flow along it (as in FF). A natural idea is to find the next augmenting path, somehow using the existing $\hat{L}(s,t)$. We accomplish this by adjusting it to the new flow.

Throughout the algorithm, we will use the *layered network* data structure, which is a digraph of the form as above: its vertex set consists of consequent layers, so that the leftmost one is $\{s\}$, while each its edge goes from some layer to the next one. We begin by initializing our data structure $\hat{L}$ as $\hat{L}(s,t)$ of the residual network. The maintenance of $\hat{L}$, after the iteration, using an augmenting path P, is as follows. We must first remove from $\hat{L}$ all the edges of P that became saturated; clearly, we can extend the flow changing procedure of FF to provide a list of such edges, Sat, in the same time $O(\ell)$. Notice that updated $\hat{L}$ will be contained in the current residual network, since only the edges such as above disappeared from the latter network as a result of the flow change. Therefore, any path from s to t found in the updated $\hat{L}$ would be an augmenting path, w.r.t. the current flow.

Observe further that applying $PathFinding$ to updated $\hat{L}$ might be stuck at a dead-end vertex with no incoming edges. In order to restore the original property of the layered network, to have no dead-ends, let us apply to it the following procedure ***Cleaning***. We initialize the left queue of edges Q^l and right queue of edges Q^r by the list of saturated edges Sat. The main loop of $Cleaning$ consists of the right and left passes, processing edges in Q^r and Q^l, respectively, one by one (in an arbitrary order), as follows:

RightPass For each edge $e \in Q^r$, if its right end-vertex has no incoming edges, then it is deleted from $\hat{L}$, together with all its outgoing edges, while inserting those edges into Q^r.

LeftPass For each edge $e \in Q^l$, if its left end-vertex has no outgoing edges, then it is deleted from $\hat{L}$, together with all its incoming edges, while inserting those edges into Q^l.

Notice that any deleted element of $\hat{L}$ is absent in all paths from s to t in the current $\hat{L}$; in another words, the above procedure does not destroy any path from s to t in $\hat{L}$. If $Cleaning$ empties $\hat{L}$, it reports on its *vanishing*.

It is easy to see that the cleaned $\hat{L}$ contains no dead-ends. Indeed, a vertex becomes a dead-end when its last incoming or last outgoing edge is removed from $\hat{L}$; thus, any such event should be detected when processing that edge and that vertex should be removed from $\hat{L}$ during $Cleaning$. Now, assuming that $Cleaning$ has not caused $\hat{L}$ to vanish, $PathFinding$ in the cleaned $\hat{L}$ should be executed without any problem. This allows us to find an augmenting path and to execute the entire next iteration of FF once more in $O(\ell)$ time. Notice that this time includes neither the time of building $\hat{L}$, nor of its cleaning. This is intentional: it is quite usual to count the cost of initializing and maintaining a data structure separately.

Such accelerated iterations are executed until $\hat{L}$ vanishes. The part of algorithm beginning from the building a layered network and finishing at its vanishing point is called a *phase*. The algorithm consists of consequent phases, repeated until the next layered network construction cannot be built. This happens when the current residual network G_f contains no path from s to t. In this case, by the Ford&Fulkerson theorem, the current flow f is maximum, i.e. the problem is solved. In what follows, we refer to the suggested algorithm as the *Dinitz algorithm*, or *DA*. Its general description is as follows (the iteration invariant is discussed and proved below):

The Original Dinitz Algorithm

Input:
a flow network $G = (V, E, c, s, t)$
a feasible flow f, in G (equal to zero, by default)

/* *Phase Loop*: */
dowhile
begin
Build $\hat{L}(s,t)$ in G_f, using the extended BFS;
if $\hat{L}(s,t) = \emptyset$ **then return** f
else $\hat{L} \leftarrow \hat{L}(s,t)$;

/* *Iteration Loop*: */
while $\hat{L}$ is not empty **do**
/* *Iteration Invariant*: $\hat{L}$ is the union of all shortest augmenting paths */
begin
$P \leftarrow PathFinding(\hat{L})$;
$Sat \leftarrow FlowChange(P)$;
/* $Cleaning(\hat{L})$: */
begin
Removal of edges in Sat;
$Q^r, Q^l \leftarrow Sat$;
$RightPass(Q^r)$;
$LeftPass(Q^l)$;
end;
end;

end;

2.3 Algorithm Analysis

Two aspects of DA have to be analyzed:

- How much does it cost to maintain the layered network?
- How many iterations are contained in a phase? how many phases are contained in the algorithm?

Layered Network Maintenance Cost It is easy to construct an extremal example, where the entire layered network $\hat{L} = \hat{L}(s,t)$ of size $O(|E|)$ vanishes after a single iteration. For example, if the first edge layer consists of just a single edge from s, of the minimal capacity, then the first executed iteration saturates this edge and thus disconnects t from s.

We use the amortized analysis for bounding the total cost of *all* cleanings during a single phase. Let us charge the cost of all relevant maintenance operations to *removed elements* of $\hat{L}$, as follows. A removed *edge* pays for the operations applied to it, to the total of $O(1)$, and for checking its two end-vertices; note that checking whether a list it is empty or not costs $O(1)$ as well. A removed *vertex* pays for operations applied to itself and for the arrangement of the loop to remove its incident edges (but not for iterations of that loop), which also totals in $O(1)$. Therefore, the overall maintenance cost during a phase is $O(|E| + |V|)$, which is $O(|E|)$, since $\hat{L}$ is connected.

Running Time w.r.t. Iterations and Phases Recall that the construction of the layered network, when initializing a phase, costs $O(|E|)$ as well. Hence, the total cost of all layered network data structure operations, except for $PathFinding$, is $O(|E|)$ per phase. Recall that the total cost of an accelerated iteration is $O(\ell) = O(|V|)$, where ℓ is the length of the layered network.

Let us denote the total number of iterations of DA by $\#it$ and the number of phases by $\#ph$ (it might be infinite, in general). Then, the total running time of DA is bounded as $O(\#it \cdot |V| + \#ph \cdot |E|)$. First, this is better than the bound $O(\#it \cdot |E|)$ of FF. Indeed, DA is much faster than FF, even if FF uses shortest augmenting paths only (see results of an experiment comparing them, among other max-flow algorithms, in [13]). Second, the essential structure of FF, when accelerated, remains exactly the same, so neither finiteness proofs, nor bounds for the number of iterations ever proved for FF suffer from the accelerating.

However, we still have no provable reason to consider DA be *theoretically* faster than FF. Indeed, there might be just a single iteration during a phase, in the worst case, and there is no provable evidence that the average number of iterations per phase $\frac{\#it}{\#ph}$ is higher than $\Omega(1)$. To summarize for the time being, DA seems to be just helpful heuristics for FF.

The Maintenance of the Layered Network is Perfect Let us continue analyzing the data structure business, presenting its purity and beauty. A method of data structure maintenance is considered perfect if, before every iteration, the data structure is as if built from scratch, w.r.t. the current data. The method chosen in the original paper [4] and described above is such. The following proposition implies straightforwardly that $\hat{L}$ *coincides with* $\hat{L}(s,t)$ *of the current residual network before every iteration of DA.*

Proposition 1. *After any iteration at the phase of DA with the layered network of length ℓ, the updated layered network is the union of all augmenting paths of length ℓ, while there is no shorter augmenting path (w.r.t. the current flow).*

Proof. Let us consider an arbitrary iteration in the phase with the layered network of length ℓ. We relate to the function $d(v)$ on vertices as the distance from s to v in the residual network *at the beginning of the phase*. In other words, $d(v)$ is the number of the layer to which v was related by the first extended BFS, during the process of the building of $L(s)$.

Recall that, during this phase, the flow on an edge (v,u) is increased only if $d(u) = d(v)+1$; only those edges may be saturated and thus *removed* from G_f. Also, the flow on an edge (u,v) is decreased only if $d(v) = d(u) - 1$; only those edges may be unsaturated and thus *added* to G_f; we call them "new". An immediate consequence is that the value of function d is defined for any vertex reachable from s in the current residual network G_f (equivalently, any such vertex was reachable from s also at the beginning of the phase). Another easy consequence is that no edge saturated during a certain phase can be unsaturated later during the same phase. Thus, no edge removed from G_f can be restored to it during the same phase.

First, let us consider augmenting paths of length ℓ without new edges (henceforth "old paths"), and prove that they are the same in the updated $\hat{L}$ and in the current G_f (we use the generic notion G_f for the residual network, assuming f changes from iteration to iteration).

Let us consider an arbitrary old path, P, of length ℓ; by construction, it was contained in the layered network $\hat{L}$ at the beginning of the phase. Assume that P is contained in G_f after some iteration. This means that none of its edges were saturated during the phase. So, none of

its edges were removed from $\hat{L}$ as saturated. Hence, the existence of P in $\hat{L}$ is self-supporting, since it prevents its nodes from ever being removed from $\hat{L}$ as dead-ends. Therefore, P is contained in the current layered network $\hat{L}$. Now assume that P is contained in the current $\hat{L}$. This means that none of its edges were saturated during the phase. Thus, P is contained in G_f. Summarizing, the layered network and the residual network contain the same old paths of length ℓ. Recall that there exist no old paths of length less than ℓ.

We now prove that the length of any "new" (not old) path, P', is at least $\ell + 2$. Let P' contain k new edges; since P is new, $k \geq 1$. Let us concentrate on the change of function d along P' (recall that d is defined at all its vertices). Its total increase, from $d(s) = 0$ to $d(t) = \ell$, is exactly ℓ. Along any one of its old edges d increases by at most 1. Along any one of its new edges d decreases by exactly 1. Hence, the number of old edges should be at least $\ell + k$. That is, the length of P' is at least $\ell + 2k \geq \ell + 2$, as required.

Thus, we see that after any iteration of the phase, there is no augmenting path of length less than ℓ, while all the augmenting paths of length ℓ, if any, are contained in the current $\hat{L}$.

Moreover, since cleaning restores the property of $\hat{L}$ of having no dead-ends, reasoning, as in the proof of Claim in Section 2.2, shows that all the vertices and edges of $\hat{L}$, if any, belong to the paths from s to t of length ℓ. This suffices. □

At this point, consider the moment after the last iteration of some phase. At that moment, $\hat{L}$ had vanished, so it contains no path from s to t of length ℓ. By Proposition 1, there is no augmenting path of length ℓ or shorter, w.r.t. the current flow. Therefore, the distance from s to t in G_f is at least $\ell + 1$, and so is the length of the layered network built at the beginning of the next phase (if the flow is not yet maximum). Thus, as a by-product of the perfect method of maintaining the layered network data structure, we have arrived at the remarkable property of DA that *the length of the layered network grows strictly from phase to phase.*

The Polynomial Time Bound of DA An easy consequence of the foregoing property is that there are at most $|V| - 1$ phases in DA, since the distance from s to t cannot exceed $|V| - 1$.

It is easy to see that the number of iterations during a single phase is at most $|E|$, since at least one out of at most $|E|$ edges of $\hat{L}$ is removed from it at every iteration. Therefore, the total running time at each phase, which consists of the accelerated iteration times and the layered network building and maintenance time, is $O(|E| \cdot |V| + |E|) = O(|V||E|)$. Thus, DA is finite and its total running time is $O(|V|^2|E|)$. We arrive at our main result:

Theorem 1. *The Dinitz algorithm builds a maximum flow in time* $O(|V|^2|E|)$.

Summary Let us summarize what is established on the behavior of DA.

- DA consists of phases. Each one contains iterations changing the flow using shortest augmenting paths of a fixed length, ℓ.
- At the beginning of each phase, the extended BFS builds in $O(|E|)$ time a layered network data structure, $\hat{L}$, of length ℓ. The layered network is constantly maintained during the phase in the total time $O(|E|)$, as the union of all shortest augmenting paths of length ℓ, until it vanishes.

- The layered structure of $\hat{L}$ and absence of dead-ends in it allow for the execution of every iteration of FF in $O(\ell) = O(|V|)$ time.
- The layered network is strictly pruned after each iteration of FF. Therefore, the number of iterations at each phase is bounded by $|E|$.
- When the layered network vanishes, there is no augmenting path of length lesser than or equal to ℓ, w.r.t the current flow. Hence, the length of the next layered network, equal to the length of the currently shortest augmenting path, is strictly greater than ℓ.
- Since the length of $\hat{L}$ grows from phase to phase, there are at most $|V| - 1$ phases.
- When DA stops, the current flow is maximum.
- The running time of DA is $O(|V|^2|E|)$.

A Historical Remark In Adel'son-Vel'sky's Algorithms class, the lecturer had a habit of giving the problem to be discussed at the next meeting as an exercise to students. The DA was invented in response to such an exercise. At that time, the author was not aware of basic facts regarding FF. In particular, neither of FF, nor of the idea of decreasing a flow on opposite edges during a flow push along a path. As a consequence, Proposition 1 turned out to be quite natural, following from *just exhausting* the set of all paths of length ℓ, since the issue of "new" augmenting paths had not arisen at all. So, the entire effort of the inventor was devoted to suggesting the best accelerating data structure.

Because of the above gap in knowledge, the first time bound was $O(|V||E|)$, since the total number of iterations had been erroneously considered to be $|E|$, according to the maximal possible number of saturated edges. After learning about the inverse edges of Ford and Fulkerson, the author completed the proof of Proposition 1 and corrected the time bound.

Ignorance sometimes has its merits. Very probably, DA would not have been invented then, if the idea of possible saturated edge desaturation had been known to the author.

3 The Version of Shimon Even and Alon Itai

There are various approaches for using a data structure for a sequence of iterations of an algorithm. The perfect way, as chosen in [4] and described above, is not the goal from the point of view of the algorithm itself. The updated data structure may not be like one built from scratch, but it should work; the only requirement is that the desired bound for the running time can be proved. This was the approach chosen by Shimon Even and Alon Itai. In what follows, we first relax some requirements for the layered network data structure, using the notation defined in Section 2. Then we describe the version of DA of Even and Itai, as it appears in the Even's textbook [14].

Note that using the two-terminal layered network $\hat{L}(s,t)$ is a luxury; the one-terminal layered network $L(s)$ is quite sufficient. Indeed, in order for $PathFinding$ to work, a layered network should have no dead-ends in the direction of s (i.e. without incoming edges), while the existence of dead-ends in the direction of t does no harm. Hence, DA may switch to the layered network L, which is initialized by $L(s)$. In fact, the building of $L(s)$ may be stopped upon finishing the ℓth layer.

Moreover, during the update of L, DA may remove dead-ends in the direction of s only; that is, at $Cleaning$, it is sufficient to execute $RightPass(Q^r)$ only. Then, L vanishes when t is removed from it. Notice that the original property of $L(s)$, to be the union of all shortest

paths from s to all the vertices reachable from it, is not preserved by such maintenance. However, this is not essential to DA, since it uses the shortest paths to t only. The right invariant of L is to *contain* all the augmenting paths of length ℓ (not to be their union), while there is no shorter augmenting path. Obviously, this property holds after the initializing L at the beginning of a phase. Its maintenance during the phase, from iteration to iteration, is proved like the proof of Proposition 1. Notice that the weaker invariant above implies the property of increasing the layered network's length from phase to phase as well. Therefore, the analysis of the number of phases, the number of iterations, and the running time of DA, for the version where the one-terminal layered network is used, is the same as in Section 2.3.

Even and Itai go much farther while changing DA. They admit dead-ends in L in both directions; thus, they cancel *Cleaning* and are urged to give up using *PathFinding*. They even burn their bridges behind them by finding a path from s to t in L beginning from s! They suggest another way to find an augmenting path: a search of the DFS type, combined with encountered dead-ends removal, *when discovered*, by the way. Each iteration of their version of DA is as follows.

Their DFS begins to build a path from s, edge by edge, from one layer to the next. Such path building stops either by arrival at t (a success), or at another dead-end. In the last case, DFS backtracks to the tail of the edge leading to that dead-end, while removing that edge from L as useless (i.e. not contained in any path from s to t in L). After that, DFS continues as above, building a path when possible or backtracking and removing the last edge, otherwise. This process ends either when DFS is at s and has no edge to leave it, indicating that the phase is finished, or when it arrives at t. In the latter case, an augmenting path is built. Then, a flow change is executed along that path, as at FF, while removing from L all its edges which have become saturated; recall that there is at least one such edge.

The running time of a phase, after executing the extended BFS to initialize L, is counted by Even and Itai as divided into intervals between *arrivals at dead-ends*. First, each such event causes the removal of an edge from L, so there may be at most $|E|$ such events. Secondly, there may be at most ℓ forward steps of DFS between neighboring events, which costs $O(\ell)$. Except for those steps, there may be either a single backtracking and edge removal, which costs $O(1)$, or a flow change, together with saturated edge removals, which costs $O(\ell)$. In any case, every interval between events costs $O(\ell)$, which totals $O(\ell \cdot |E|) = O(|V||E|)$ per phase, as desired.

Summarizing our discussion on the version of DA suggested by Even and Itai, we see that their use of a non-cleaned layered network, where ad-hoc cleaning is made "on route", is quite successful: not only the same $O(\cdot)$ time bounds are acquired, but also the practical effectiveness (where constants are concerned as well) is not lost. Needless to say, their version not only has its own real beauty, but is somewhat sexy running DFS on the layered network constructed by an extended BFS.

Regarding proof of the crucial property of DA that the length of the layered network increases from phase to phase, Even and Itai do not use any data structure invariant, maintained from iteration to iteration, in contrast with the original paper [4]. Instead, they explicitly prove the above property, by analyzing the situation when a layered network vanishes. They characterize such a situation by the property of the "maximal" flow accumulated *in* L during the phase being (as distinguished from "maximum"), defined as a flow such that any path from s to t in L contains at least one edge saturated by it (a "blocking" flow, in notation of [9]). Lemma 5.4 in [14] proves that after arriving at a maximal flow in the layered network of length ℓ, the currently shortest augmenting path is longer than ℓ.

It is interesting that the arguments in the proof of that Lemma are similar to those in the proof of Proposition 1. However, the reader may see clearly that the overall emphases, in the presentation of Dinic's algorithm at [14] are quite different from those of the original presentation of DA in [4], in both the algorithm iteration and the algorithm analysis. The origin of this difference is apparently the rejection of the data structure maintenance approach, occurring everywhere in the version of Even and Itai. See Section 1 for a possible explanation of this phenomenon.

4 Implementation of DA by Cherkassky

At the implementation stage, nothing extra should be left over. The program designer should have removed all the non-essential parts, not only to decrease the volume of programming work and/or decrease constant factors at the running time. This is also to reduce the probability of bugs: a simpler program has less bugs, and thus causes less risk of arriving, sometimes, at a sudden malfunction, or what is the worst, at a wrong result. The only restriction, while choosing an implementation, is the equivalence of the designed program to the original algorithm, which is required for the validity of the program.

From the 1970th, Boris Cherkassky worked on the quality implementation of various flow algorithms and on the experimental evaluation of their performance, see [13, 26]. His recommendations for the best implementation of DA are as follows:

- No layered network—neither $L(s)$ nor $\hat{L}(s,t)$—should be built. It is sufficient to compute just the layer number ("rank") $dist(v,t)$ for every vertex up to the number $\ell = dist(s,t)$. This may be done by a single run of the *usual* BFS from t on the unsaturated edges, in the inverse edge direction.[2]
- The entire phase is conducted by a single DFS from s. Any saturated edge or edge not going from a vertex of rank i to a vertex of rank $i+1$ is just skipped (instead of using the list of edges in the layered network).
- No edge removal is needed. If DFS backtracks on some edge, that edge will not participate in the remaining part of DFS automatically.
- When the outgoing edges from the current vertex, $v \neq s,t$, are exhausted (v becomes a dead-end), DFS backtracks from v. If s becomes a dead-end, DFS (and the phase) is completed.
- When arriving at t, the current DFS path is an augmenting path. The usual flow change is made along it. After this, DFS continues from the tail of the edge closest to s which has been saturated during this flow change.
- The only network data needed during the phases is the residual capacity c_f for all the edges. After the last phase, the flow is restored as the difference of capacities c and c_f.

The resulting implementation is as follows:

Implementation of DA by Cherkassky

/* *Input*: */
a flow network $G = (V, E, c, s, t)$

[2] We could change to the opposite direction, in particular, to the ranks $dist(s,v)$, for consistency with the preceding discussion. Flipping to the ranks $dist(v,t)$ of Cherkassky is done for better intuition and for consistency with the push-relabel technique, discussed in Section 5.

```
    a feasible flow f, in G (equal to zero, by default)
Initialization:
    compute ∀e ∈ E : c_f(e) = c(e) − f(e);
/* Phase Loop: */
    dowhile
    begin
        compute ∀v ∈ V : rank(v) = dist(v, t), by BFS from t on edges with c_f > 0,
          in the inverse edge direction;
        if rank(s) = ∞ then begin  f ← c − c_f;  return f; end;
        while DFS from s do
        begin
            /* P denotes the current path and x the current vertex */
            any edge (x, y) s.t. c_f(x, y) = 0 or rank(y) ≠ rank(x) − 1 is skipped;
            upon arriving at x = t  do
            begin
                Δ ← min{c_f(e) : e ∈ P};
                for edges (u, v) of P, from t to s  do
                begin
                    c_f(u, v) ← c_f(u, v) − Δ;  c_f(v, u) ← c_f(v, u) + Δ;
                    if c_f(u, v) = 0  then x ← u
                end;
            continue DFS from x;
            end;
        end;
    end;
```

In Section 5, we will see that such a simplification of DA, especially using vertex ranks instead of layered networks, has a deep influence on the following research on max-flow algorithms.

5 On Advancing Max-Flow Finding

In this section, we consider some issues related to DA itself and the development of the max-flow algorithms area from DA towards the push-relabel algorithm. Many other interesting results and even directions in the max-flow area remain thus untouched or mentioned quite briefly.

Edmonds' and Karp's Version of FF Edmonds and Karp proved the finiteness and polynomial time of FF, if only the shortest augmenting paths are used in it. The achieved time bound is $O(|V||E|^2) = O(|V|^5)$, which is higher than $O(|V|^2|E|) = O(|V|^4)$ of DA, since iterations are made in $O(|E|)$ time each, as usual. A talk on their work was given at a conference held in the middle of 1968, just half a year before the invention of DA. However, their result became known in Moscow and was told to the author only at the end of 1972, after the journal version of their paper [5] had been published.

The Running Time of DA for Special Cases The "iron curtain" worked in both directions. In January 1971, at a conference in Moscow, the author and Alexander Karzanov

reported on new time bounds of DA for specific network types [6, 7]. In [6], it is shown that if DA is executed on a network with unit capacities, the running time of each phase is bounded by $O(|E|)$ (since each edge of $\hat{L}$ is saturated after participating in a one or two augmenting paths). Hence, the total running time of DA is $O(|V||E|)$ for such a network. In [7], the flow network modeling of the bipartite $n \times n$ matching is considered. The number of phases of DA in such a modeling network is shown to be $O(\sqrt{n})$. Using the result of [6], the running time of DA for solving the bipartite $n \times n$ matching is found to be $O(\sqrt{n}m) = O(n^{5/2})$, where m is the number of pairs allowed for matching.

An algorithm for the bipartite $n \times n$ matching, similar to DA, with the same time bound of $O(\sqrt{n}m) = O(n^{5/2})$, was suggested by Hopkroft and Karp [8]. Their algorithm is the only one cited in the West. Various time bounds for DA working in networks of special types, similar to those in [6, 7], were suggested by Even and Tarjan in [11]; in particular, they showed that the number of phases of DA for a network with unit capacities is bounded as $O(\min\{|E|^{1/2}, |V|^{2/3}\})$, and thus its running time is $O(|E| \min\{|E|^{1/2}, |V|^{2/3}\})$. Still, the textbook [14] published in 1979 does not mention the results of [6, 7], when reviewing these topics.

Karzanov's Algorithm After DA, there were two "revolutions" in the research on finding the max-flow. The "preflow-push" revolution is due to Karzanov, who was the first to leave the augmenting paths idea. He observed that at each phase of DA, it is sufficient to somehow find a flow in a layered network $\hat{L}(s,t)$, such that any path from s to t contains an edge saturated by it; he calls such a flow "blocking". His algorithm (henceforth, *KA*) finds a blocking flow in $O(|V|^2)$ time, thus arriving at an $O(|V|^3)$ max-flow algorithm [9].

Here, we provide an outline of KA. Each finding of a blocking flow begins from *Pushing*. It scans vertices of $\hat{L}(s,t)$ in their BFS order, starting from s, and pushes flow from the current vertex, v, on its outgoing edges, as much as possible. If the outgoing edges have enough capacity to remove from v all the flow brought to it on incoming edges (this amount is formally considered infinity at s), then a flow balance is created, at v. Otherwise, there would be a "flow excess" at v, to be fixed farther on. After finishing *Pushing*, the flow function is infeasible, in general, with a flow excess at some vertices; such a function is called a *preflow*. Then, KA begins *Balancing*, which scans vertices with a flow excess but without outgoing unsaturated edges. *Balancing* pushes flow from such a vertex *back* on its incoming edges. Then, a flow excess is created at its preceding vertices, where outgoing unsaturated edges may exist. Then, a new *Pushing* is executed, followed by a new *Balancing*, and so on until balancing the preflow at all the vertices, except for s. The result is a desired blocking flow.

Improvements upon DA and KA Much effort was made, based on DA and KA, to lower the running time bound of max-flow finding; the aim was to acquire an $O(|V||E|)$ max-flow algorithm. Following is a brief review, cut before the push-relabel algorithm. Cherkassky first simplified KA and later suggested a "hybrid" of DA and KA running in time $O(|V|^2\sqrt{|E|})$ [12]. Galil accelerated his algorithm to run in time $O(|V|^{5/3}|E|^{2/3})$ [15]. Galil and Naaman suggested an acceleration of augmenting path finding in DA by means of storing parts of previously found augmenting paths [16] (Schiloach discovered a similar algorithm independently a bit later). Their algorithm runs in time $O(|E||V|\log^2|V|)$, which differs from the "goal time" only in a poly-logarithmic factor. Sleator and Tarjan improved their algorithm to run in time $O(|E||V|\log|V|)$ by using trees instead of just paths [18]. These low running times were achieved by using smart, but heavy, data structures.

Towards the Push-Relabel Algorithm The other max-flow research movement was slow, its explicit direction was to simplify the pushing-balancing technique of Karzanov. As seen a-posteriory, its implicit goal was the "push-relabel" revolution made finally in [21]. Let us show an example of such a development by analyzing the algorithm of Boris Cherkassky [13].[3]

One of the crucial steps taken in [13] is to cancel the goal of each phase: to construct a feasible flow in the layered network, and hence to eliminate *Balancing*. Instead, the new requirement is to achieve a flow balance at every vertex only by the end of the algorithm's execution. The main technical observation, while transforming KA, is that *Balancing* at some vertex is equivalent to *Pushing* from it in future, when it will appear in some layered network at a greater distance from the sink t. As in his implementation of DA (see Section 4), Cherkassky does not build layered networks, but just computes the vertex ranks $d(v) = dist(v,t)$.

There are steps of only the following two types in his algorithm:

Push For a vertex with a flow excess, pushing from it as much flow as possible to the vertices of a rank smaller by one.

Relabel For vertices without unsaturated edges outgoing to vertices of a rank smaller by one, recomputing their ranks.

For the reader familiar with the push-relabel approach [21], the remarkable similarity of the above steps with those of the generic push-relabel algorithm is obvious. (The names of the types are taken from [21], not from [13].)

Moreover, Cherkassky never computes new vertex ranks from scratch, except at the beginning of the algorithm—he *maintains* them. That is, he just increases the current vertex ranks as needed. This maintenance is done as that of the *global layered network*, which was sketched in [4] and given later in detail in [10] (see the next subsection). So, also the maintenance of vertex ranks is equivalent to the relabeling method of [21]. Summarizing, the algorithm of [13] is one of the possible implementations of the generic push-relabel algorithm [21].

The essential difference between the algorithm [13] and the approach of [21] is that the former is rigid, while the latter is generic, i.e. leaving freedom to choose any order for processing vertices with a flow excess. Leftovers of the phase structure of DA and KA prevented Cherkassky from canceling the division of KA into phases, inherited from DA. At each phase of his algorithm, all the vertices are processed in the BFS order from farther to closer to the sink, and after that, all vertex ranks are updated. He uses the technique of the global layered network for accelerating the algorithm only. He does not notice that it is able to maintain vertex ranks *after each elementary push* at the same total cost, so that after any push, the algorithm is ready and free to push at any other vertex requiring it. This oversight prevented the algorithm [13] from becoming generalized to the generic push-relabel algorithm.

Usually, in the modern research community, relaxations of suggested methods (which get rid of various leftovers) are made very quickly after the publication of an interesting result. However, because of the "iron curtain", the results of [13] were unknown in the West, preventing further development by Western researchers[4]. The push-relabel approach was invented by a Ph.D. student, Andrew Goldberg, in the middle of 1980s.

[3] The $O(|V|^3)$ max-flow algorithm described by Shiloach and Vishkin in [17] may also be considered as a precursor of one of the push-relabel algorithms.

[4] Ahuja et al., in [25], relates even seminal *introducing distance labels, (i.e. vertex ranks) instead of layered networks*, to the paper of Goldberg [19] published in 1985.

Perfect Maintaining the One-Terminal Layered Network For completeness, let us briefly describe the method of perfect maintaining of the one-terminal layered network $L(s)$ [10]. Note that using it may eliminate the phase borders from both the original DA and its implementation given in Section 4.

The layered network L is intialized by building $L(s)$ w.r.t. the initial flow by the extended BFS in time $O(|E|)$. After a flow change, we take care of the vertices of L that lose the last incoming edge (called "dead-ends"). At DA, such a vertex is only removed from the data structure, and now we must put it into its correct layer. After removing the saturated edges, we set the *temporary distance* from s to each dead-end v at $d'(v) = d(v) + 1$. Additionally, every outgoing edge (v, u) is removed from the list of edges incoming to u. If, as a result, u becomes a dead-end, we apply the same operation to it.

We process the dead-ends in the non-decreasing order of temporary distances d'. For each dead-end v, we check whether there is an incoming edge from the layer $d'(v) - 1$. If so, we assign $d(v) = d'(v)$ and scan its incident edges: we insert into L the edges going to v from the layer $L_{d(v)-1}$ and those going from v to the layer $L_{d(v)+1}$. Otherwise, we increase $d'(v)$ by one, to be processed once more. It may be easily shown that this processing updates L to become $L(s)$ w.r.t. the new flow. In total, each vertex is moved to the next layer at most $|V| - 2$ times, which implies that the total cost of updating L is $O(|V||E|)$ during the entire layered network maintenance.

Maintaining vertex ranks, as in [13, 21], differs from the above method in beginning from t, reversing edge directions, and removing all operations with edges.

After Inventing the Generic Push-Relabel Algorithm The algorithm with the currently best strongly polynomial (i.e. not depending on values of capacities) time bound for a max-flow finding is suggested by Goldberg and Tarjan in [21]; its running time is $O(|V||E| \log(|V|^2/|E|))$. The algorithm combines the push-relabel approach, choosing a vertex of the maximal rank at each push step, with the dynamic tree technique of [18] (changed slightly).

Interestingly, after this record bound, the development of max-flow algorithms returned to the *blocking flow techniques*, that is to dividing an algorithm execution into phases, each computing a blocking flow in a certain auxiliary network. In a couple of years, Goldberg and Tarjan achieved a result finer than that of [21], by suggesting a $O(|E| \log(|V|^2/|E|))$ algorithm for finding a blocking flow in an arbitrary acyclic graph (the need to use acyclic graphs, which generalize layered, arised when building a fast min-cost max-flow algorithm).

The above-mentioned results leave a polynomial gap between the general and the unit capacity cases: the bounds on the number of blocking flow computations in the former case is $O(|V|)$ and in the latter, $O(\min\{|E|^{1/2}, |V|^{2/3}\})$. For the case when capacities are integers bounded by U, Goldberg and Rao [27] show a blocking flow algorithm with $O(\min\{|E|^{1/2}, |V|^{2/3}\} \log U)$ phases. The auxiliary network for each phase is acyclic, instead of layered. This is implied by the vertex ranks arising from assigning to each edge a length of one or zero, depending on whether its residual capacity is below or above a certain dynamic threshold, respectively, and contracting all the zero length cycles. A blocking flow at each phase is found by the algorithm from [21], thus arriving at the total $O(\min\{|E|^{1/2}, |V|^{2/3}\}|E| \log U \log(|V|^2/|E|))$ running time. Thus the time bound for the unit capacity case is extended to the integral capacity case at the expense of a factor of $\log U \log(|V|^2/|E|)$. Note that unless U is very big, the time bound achieved in [27] for general flow networks is better than $O(|V||E|)$.

****

Acknowledgments. The author is grateful to Avraham Melkman and other colleagues for sharing the effort to clarify the explanation of Dinitz' algorithm, to Boris Cherkassky and Andrew Goldberg for useful information and comments, to Oded Goldreich for his patience and help in improving presentation of this paper, and to Ethelea Katzenell for her English editing.

References

1. G.M. Adel'son-Vel'sky and E.M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady* **3** (1962), 1259–1263).
2. L.R. Ford and D.R. Fulkerson. Flows in networks. Princeton University Press, Princeton, NJ, 1962.
3. V.L. Arlazarov, E.A. Dinic, M.A. Kronrod, and I.A. Faradjev. On economical finding of transitive closure of a graph. *Doklady Akademii Nauk SSSR* **194** (1970), no. 3 (in Russian; English transl.: *Soviet Mathematics Doklady* **11** (1970), 1270–1272).
4. E.A. Dinic. An algorithm for the solution of the max-flow problem with the polynomial estimation. *Doklady Akademii Nauk SSSR* **194** (1970), no. 4 (in Russian; English transl.: *Soviet Mathematics Doklady* **11** (1970), 1277–1280).
5. J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. of ACM* **19** (1972), 248–264.
6. E.A. Dinic. An efficient algorithm for the solution of the generalized set representatives problem. In: *Voprosy Kibernetiki. Proc. of the Seminar on Combinatorial Mathematics (Moscow, 1971).* Scientific Council on the Complex Problem "Kibernetika", Akad. Nauk SSSR, 1973, 49–54 (in Russian).
7. A.V. Karzanov. An exact time bound for a max-flow finding algorithm applied to the "representatives" problem. In: *Voprosy Kibernetiki. Proc. of the Seminar on Combinatorial Mathematics (Moscow, 1971).* Scientific Council on the Complex Problem "Kibernetika", Akad. Nauk SSSR, 1973, 66–70 (in Russian).
8. J. Hopkroft and R.M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. on Computing* **2** (1973), 225–231.
9. A.V. Karzanov. Determining the maximum flow in the network by the method of preflows. *Doklady Akademii Nauk SSSR* **215** (1974), no. 1 (in Russian; English transl.: *Soviet Mathematics Doklady* **15** (1974), 434–437).
10. G.M. Adel'son-Vel'sky, E.A. Dinic, and A.V. Karzanov. *Network flow algorithms.* "Nauka", Moscow, 1975, 119 p. (in Russian; a review in English see in [24]).
11. S. Even and R.E. Tarjan. Network flow and testing graph connectivity. *SIAM J. on Computing* **4** (1975), 507–518.
12. B.V. Cherkassky. An algorithm for building a max-flow in a network, running in time $O(n^2\sqrt{p})$. In: *Mathematical Methods for Solving Economic Problems*, issue **7** (1977), "Nauka", Moscow, 117–126 (in Russian).
13. B.V. Cherkassky. A fast algorithm for constructing a maximum flow through a network. In *Combinatorial Methods in Network Flow Problems.* VNIISI, Moscow, 1979, 90–96 (in Russian; English transl.: *Amer. Math. Soc. Transl.* **158** (1994), no. 2, 23–30).
14. S. Even. *Graph Algorithms.* Computer Science Press, Rockville, MD, 1979.
15. Z. Galil. An $O(V^{5/3}E^{2/3})$ algorithm for the maximal flow problem. *Acta Inf.* **14** (1980), 221-242.
16. Z. Galil and A. Naamad. An $O(EV\log^2 V)$ algorithm for the maximal flow problem. *J. Comput. Syst. Sci.* **21** (1980), no. 2, 203-217.
17. Y. Shiloach and U. Vishkin. An $O(n^2\log n)$ parallel max-flow algorithm. *J. of Algorithms* **3** (1982), 128-146.

18. D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.* **24** (1983), 362-391.
19. A.V. Goldberg. A new max-flow algorithm. TR MIT/LCS/TM-291, Laboratory for Comp. Sci., MIT, Cambridge, MA, 1985.
20. R.E. Tarjan. Amortized computational complexity. *SIAM J. Alg. Disc. Meth.* **6** (1985), no.2, 306–318.
21. A.V. Goldberg and R.E. Tarjan. A new approach to the maximum flow problem. In: *Proc. of the 18th ACM Symp. on the Theory of Computing*, 136–146. Full paper in *J. of ACM* **35** (1988), 921–940.
22. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* McGraw-Hill, New York, NY, 1990.
23. A.V. Goldberg and R.E. Tarjan. Finding minimum-cost circulations by successive approximation. *Mathematics of Operations Research*, **15** (1990), issue 3, 430-466.
24. A.V. Goldberg and D. Gusfield. Book Review: Flow algorithms by G.M. Adel'son-Vel'ski, E.A. Dinits, and A.V. Karzanov. *SIAM Reviews* **33** (1991), no. 2, 306–314.
25. R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, Upper Saddle River, NJ, 1993.
26. B.V. Cherkassky and A.V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica* **19** (1997), 390–410.
27. A.V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Journal of ACM* **45** (1998), 753–782.