Journal of the Operations Research
Society of Japan
Vol. 26, No. 1, March 1983

ON THE PRACTICAL EFFICIENCY OF VARIOUS MAXIMUM FLOW ALGORITHMS

Hiroshi Imai
University of Tokyo

(Received July 6, 1982; Revised December 13, 1982)

Abstract We investigate the practical efficiency of various maximum flow algorithms by a great number of systematically-designed computational experiments, where we develop and summarize the techniques useful for solving the problem in practice. The methods implemented are depth-first search, breadth-first search, Dinic's, Karzanov's, three Indians' and Galil and Naamad's. We conclude that, among the implemented methods, Dinic's method and Karzanov's are the most efficient. In these two methods, Dinic's method is so simple that we can code it quite easily, and takes rather small memory storages. Karzanov's method is superior from the viewpoint of the complexity in the worst case.

1. Introduction

The maximum flow problem is one of the fundamental problems in network flow theory together with the shortest path problem. And the algorithms for finding a maximum flow in a given network are important since they are not only applied directly to the analysis of traffic network, communication network, etc., but are often employed in the subproblems of other network problems.

The study of the maximum flow problem was begun in 1956 by Ford and Fulkerson [7] who showed the famous maximum flow minimum cut theorem and gave the labeling procedure for finding a maximum flow. And it would be considered that in 1960's the theoretical framework for this problem had been almost established (see Ford and Fulkerson [8] and Iri [15]). In about the last decade, various studies were made on the maximum flow algorithms, where, especially, the developments in recent years were striking. See Table 1.1 for the algorithms developed so far where it should be noted that the worst-case bounds given in this table were shown to be tight theoretically [10]. The latest algorithm of Sleator and Tarjan [20], [21] solves the problem in $O(|A||V|\log|V|)$ time where $|A|$ and $|V|$ denote the numbers of arcs

and vertices in the network, respectively. Their algorithm seems hard to beat and further theoretical improvement on the worst-case bound of the maximum flow algorithm would require a new approach utterly different from Dinic's [4] on which all recent algorithms are based.

Table 1.1. Various maximum flow algorithms
($|A|$: number of arcs, $|V|$: number of vertices)

Algorithms	Year	Time
Ford and Fulkerson [7]	1956	—
Dinic [4]	1970	$\|A\|\|V\|^2$
Edmonds and Karp [5]	1972	$\|A\|^2\|V\|$
Karzanov [17]	1974	$\|V\|^3$
Cherkasskii [2]	1977	$\sqrt{\|A\|}\|V\|^2$
Galil [9]	1978	$\|A\|^{2/3}\|V\|^{5/3}$
Malhotra et al. [18]	1978	$\|V\|^3$
Galil and Naamad [11]	1979	$\|A\|\|V\|(\log\|V\|)^2$
Sleator and Tarjan [20],[21]	1980	$\|A\|\|V\|\log\|V\|$

From the practical point of view, however, few researches to estimate the practical efficiency of various maximum flow algorithms have been made. So long as we know, there are only two papers which conveys computational results and evaluate the computational efficiency of the algorithms. One is Cheung's [3] which would be the first to present computational investigation of several maximum flow algorithms up to Karzanov's [17]. Cheung concluded that Dinic's algorithm is the best among the methods implemented. But the data structures and the test network that he employed are not suited well to evaluate the computational results of the algorithms and therefore his results cannot be applied directly to the practical problems. The other is Glover, Klingman, Mote and Whitman's [12] which not only studied the computational experiments of usual maximum flow algorithms extensively, but also presented elegant primal simplex methods for the maximum flow problem. They concluded that Dinic's method is the best for usual types of networks, and one of their primal simplex methods is the best for a special type of network. And they also showed that three Indians' method [18] is quite inefficient. But they did not implement Karzanov's method since Cheung concluded that Karzanov's is inferior to Dinic's, and they thought that Karzanov's algorithm requires much more memory storages than the others.

This paper studies the practical efficiency of various maximum flow algorithms. We test various algorithms for several different types of networks to estimate and characterize their practical efficiency. The methods compared are depth-first search, breadth-first search, Dinic's, Karzanov's and three Indians'. And for reference we also implement Galil and Naamad's algorithm [11] in a manner slightly different from the above methods and guess its practical efficiency.

In section 2, we define the maximum flow problem and make some preliminaries. In section 3, we first show the basic data structures for representing a given network in the computer, and then describe the examined methods with presenting the useful techniques for implementing them. The algorithms are roughly classified into two, where one is based on the labeling procedure of Ford and Fulkerson, and the other is the method using the referent networks due to Dinic. In section 4, in order to examine the computational efficiency of the above algorithms we introduce several types of test networks with describing their features and present the computational results for respective test networks.

Then we find that Dinic's method is very efficient as pointed out by the previous researchers, and moreover that Karzanov's method is no less efficient than Dinic's, and in a certain dense network, Karzanov's is really superior to Dinic's. These two methods can be easily coded by employing the data structures and detailed techniques presented in the present paper.

2. Maximum Flow Problem

Let $G=(V,A)$ be a graph with vertex set V and arc set A where two functions $\partial^+,\partial^-: A \to V$ are given such that, for $a \in A$, $\partial^+ a$ (resp. $\partial^- a$) denotes the initial vertex (resp. terminal vertex) of arc a. We further consider functions $\delta^+,\delta^-: V \to 2^A$ such that, for $u \in V$, $\delta^+ u$ (resp. $\delta^- u$) denotes the set of arcs leaving (resp. entering) vertex u. In G, two special distinct vertices s and t, called *source* and *sink*, respectively, are given. And also a *capacity* c: $A \to \mathbf{R}_+$ (nonnegative reals) is given. Then we call $N=(V,A,c,s,t)$ a *network*, where G is the *underlying graph* of N. A *(feasible) flow* f in network N is a function from A to $\mathbf{R}$ of reals satisfying

$$(2.1)\qquad \sum_{a\in\delta^+u} f(a) - \sum_{a\in\delta^-u} f(a) = \begin{cases} F \geqq 0 & (u=s) \\ 0 & (u \in V-\{s,t\}) \\ -F & (u=t) \end{cases}$$

$$(2.2)\qquad 0 \leqq f(a) \leqq c(a) \qquad (a \in A),$$

where (2.1) corresponds to the *flow conservation law*, and (2.2) corresponds to the *capacity constraint*. In (2.1), F is called a *value* of flow f. A *maximum flow* is a flow

whose value is maximum among those of feasible flows. The problem is to find a maximum flow in a given network.

A subset S of V is called a *cut* if $s \in S$ and $t \notin S$. We define a capacity $C(S)$ of a cut S by

$$C(S)=\sum_{a\in\Delta^+S} c(a), \text{ where} \tag{2.3}$$

$$\Delta^+S=\{a \mid a\in A,\ \partial^+ a\in S,\ \partial^- a\notin S\}. \tag{2.4}$$

As is well known, the maximum value of a flow is equal to the minimum capacity of a cut (see [8], [15]).

In network N with flow f, arc $a=(u,w)\in A$ is *useful from u to w* if $f(a)<c(a)$ and is *useful from w to u* if $f(a)>0$. Through this arc a, we can augment flow by $c(a)-f(a)$ from u to w, and by $f(a)$ from w to u. A *flow-augmenting path* P from u to w is a sequence $v_0, a_1, v_1, a_2, \ldots, v_{k-1}, a_k, v_k$ such that $v_0=u$, $v_k=w$, $v_i\in V-\{s,t\}$ $(1\leqq i\leqq k-1)$, $v_i\neq v_j$ $(0\leqq i<j\leqq k)$, $a_i\in A$ $(1\leqq i\leqq k)$ and, for each $i=1, \ldots, k$, arc a_i is useful from v_{i-1} to v_i. The *length* of this path P is defined to be k. If we augment flow as much as possible through this path P, there comes to be an arc a_i for some i such that a_i is no more useful from v_{i-1} to v_i. Such an arc is called a *saturated arc*.

For finding a maximum flow, usual maximum flow algorithms iterate to find a flow-augmenting path from s to t, and to push the maximum amount of flow along it until there comes to be no flow-augmenting path from s to t in the network. A flow that admits no flow-augmenting path from s to t is a maximum flow.

3. Description of Implemented Maximum Flow Algorithms

3.1. Basic data structures for representing a network

For representing a given network N in the computer, we are essentially based on the standard data structure as presented in [16]. That is, the underlying graph $G=(V,A)$ of N is represented by functions $\partial^{\pm}$ and $\delta^{\pm}$. Functions ∂^+ and ∂^- are represented by two arrays of size $|A|$. For each vertex $u\in V$, δ^+u is expressed as a one-way list having head pointer, and totally a function δ^+ is represented by an array of size $|A|$ for lists and an array of size $|V|$ for head pointers. Similarly is represented δ^-. Two arrays of size $|A|$ are employed for representing a capacity and a flow. Thus in this standard data structure N is represented by six arrays of size $|A|$ and two arrays of size $|V|$.

We make a slight modification for the above standard data structure, where an important observation is that the maximum flow algorithms manipulate arcs in both directions many times. We suppose that $|A|$ given arcs are numbered from 1 to $|A|$.

We do without δ^- but for an arc $a=(w,u)$ numbered k $(1 \leqq k \leqq |A|)$, consider that there is an arc $a'=(u,w)$ numbered $k+|A|$ and add this arc to $\delta^+ u$. And at all times we set RESCAP$[k]:=c(a)-f(a)$ and RESCAP$[k+|A|]:=f(a)$ where RESCAP, an abbreviation of "residual capacity", is an array of size $2|A|$. Apparently capacity c and flow f can be expressed by RESCAP, and the underlying graph is represented by $\partial^{\pm}$ and modified δ^+. Then N is represented by six arrays of size $|A|$ and an array of size $|V|$.

Glover et al. [12] presented tricker and more sophisticated data structures for representing a network in the maximum flow algorithms, but we do not employ them (a partition scheme in [12] may be applicable), because in many cases we must not only compute a maximum flow but also process this flow further, and the standard data structures make this process easy though the tricky one does not. And also because we believe that the further drastic improvement cannot be made so long as we employ the standard data structure as above.

3.2. Labeling procedure

Ford and Fulkerson [8] gave the labeling procedure, which finds a flow-augmenting path by labeling and scanning techniques. However their procedure is not a practical one in the sense that the order of searching vertices and scanning arcs is not explicitly given, which causes various problems. In practical cases, the last-labeled-first-scanned scheme, i.e., depth-first search method, or the first-labeled-first-scanned scheme, i.e., breadth-first search method are employed.

(a) Depth-first search method (DFS)

This method finds a flow-augmenting path by depth-first search. At the beginning, all the vertices except s are unlabeled. Starting with s, we search a labeled vertex u by scanning arcs incident to u only until an arc which is useful from u to unlabeled vertex w is found. As soon as such vertex w is found, we give the vertex w label "k" where k is the number of its arc $(1 \leqq k \leqq 2|A|)$ and begin to search w. And, t being labeled, we have found a flow-augmenting path from s to t that can be backtraced with the aid of labels. Then we first compute the maximum amount δ of flow that can be pushed from s to t through this path by backtracing it, and then augment flow by δ. After flow augmentation, all the labels except on s are deleted. This procedure is iterated until there comes to be no flow-augmenting path from s to t. The worst-case time bound of this method is not polynomial in $|A|$, $|V|$.

These operations can be done with an array of size $|V|$, element of which records the label on each vertex. We employ this method because this is the most typical and simplest in all, though there are several variants of depth-first search method which may be better.

(b) Breadth-first search method (BFS)

This method finds a flow-augmenting path by breadth-first search, that is, each flow augmentation is done through the *shortest flow-augmenting path*. Due to this property, the number of times to augment flow can be shown to be $O(|A||V|)$ [5]. Since each breadth-first search and flow augmentation can be done in $O(|A|)$ time, the total time complexity of this method is $O(|A|^2|V|)$.

We use the same type of labels and augment flow similarly as in the above depth-first search method. In this method, however, we do not delete all labels after flow augmentation, but re-use still valid labels (see [12]). At the flow augmentation, we record the saturated arc a closest to s on the present flow-augmenting path. In two vertices $\partial^+ a$ and $\partial^- a$, let u be the one closer to s and w be the other. Then we see that the labels which were set before w was labeled remains valid. Hence we remove only the labels that were set after w was labeled (including w), and restart searching from vertex u. Re-using labels practically saves the time to scan.

These operations, including re-using of labels, can be done by two arrays of size $|V|$, where one is for recording labels, and the other is for a queue to execute breadth-first search.

3.3. Methods of using the referent network

Dinic [4] gave an $O(|A||V|^2)$ maximum flow algorithm by introducing a referent network consisting of collection of the shortest flow-augmenting paths, and a maximal (or blocking) flow in the referent network. The algorithms following Dinic's improve the total time complexity by improving the method for finding a maximal flow in the referent network. Karzanov's algorithm [17] and three Indians' [18] find a maximum flow in $O(|V|^3)$ time, and Galil and Naamad's [11] does in $O(|A||V|(\log|V|)^2)$ time.

(c) Dinic's algorithm (DNC, P-DNC)

For a network $N=(V,A,c,s,t)$ with flow f, the *referent network* $\tilde{N}=(\tilde{V},\tilde{A},\tilde{c},s,t)$ is defined as follows, where we consider a function l on V such that, for $v\in V$, $l(v)$ denotes the length of the shortest flow-augmenting path from v to t in N with flow f ($l(t)=0$ and for $v\in V$ if there is no flow-augmenting path from v to t, then $l(v)=\infty$).

(3.1) $\tilde{V}=\{v|\ v\in V,\ l(v)\leqq l(s)$, there is a flow-augmenting path from s to v of length $l(s)-l(v)$ in network N with flow $f\}$.

(3.2) $\tilde{A}=\bigcup\{A^+(u)\cup A^-(u)|\ u\in\tilde{V}\}$ where

(3.3) $A^+(u)=\{a|a=(u,v)\in A,\ l(u)-l(v)=1,\ f(a)<c(a)\}$

(3.4) $A^-(u)=\{a|a'=(v,u)\in A,\ l(u)-l(v)=1,\ f(a')>0,\ a=(u,v)\}$

(3.5) $$\tilde{c}(a)=\begin{cases} c(a)-f(a) & a\in A^+(u) \text{ for } u\in\tilde{V} \\ f(a') & a\in A^-(u) \text{ for } u\in\tilde{V},\ a' \text{: reorientation of } a \end{cases}$$

Note that the referent network $\tilde{N}$ depends upon flow f as well as network N. The length of the referent network $\tilde{N}$ is defined to be $l(s)$.

As is easily seen from the definitions, the referent network is acyclic and layered, i.e., the set $\tilde{V}$ of vertices in the referent network is partitioned into $\tilde{V}_0, \tilde{V}_1, \cdots, \tilde{V}_{l(s)}$ where $\tilde{V}_i=\{v \mid v\in\tilde{V},\ l(v)=i\}$ $(0\leqq i\leqq l(s))$. A *maximal flow* $\tilde{f}$ in the referent network $\tilde{N}$ is a flow on $\tilde{N}$ such that every path from s to t contains an arc $a\in\tilde{A}$ with $\tilde{f}(a)=\tilde{c}(a)$. Note that a maximal flow is not necessarily a maximum flow. (In Fig.3.1(b), if we augment the flow through the path s, u_1, u_4, t by 1, the resulting flow is a maximal flow but not a maximum flow.)

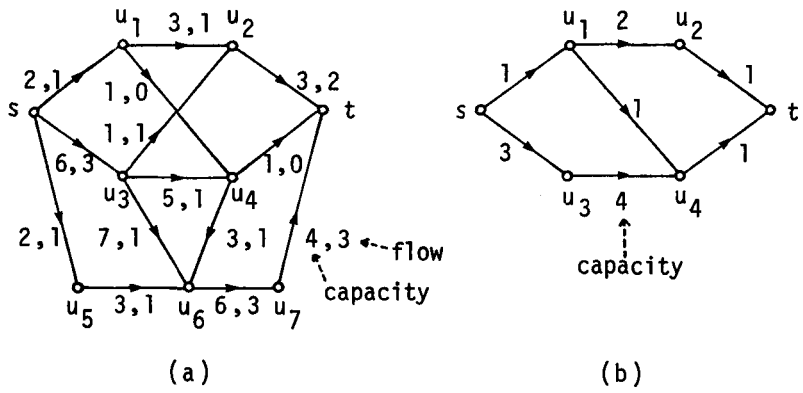

Fig. 3.1. Referent network
(a) network N with flow f
(b) its referent network $\tilde{N}$

Starting with an appropriate feasible flow f in N, which may be zero flow, Dinic's algorithm seeks for a maximum flow by iterating the following:

(i) construct the referent network $\tilde{N}$ from network N with flow f, and

(ii) find a maximal flow $\tilde{f}$ in $\tilde{N}$, and update f by adding $\tilde{f}$

until there comes to be no flow-augmenting path from s to t in N with flow f. In the above procedure, each execution of (i) and (ii) is referred to as *phase*. Since the length of the referent network can be shown to increase at each phase necessarily, and the length of the referent network is at most $|V|-1$, Dinic's algorithm finds a maximum flow by iterating (i) and (ii), i.e., executing the phase at most $|V|-1$ times.

Let us consider how we should construct and represent the referent network $\tilde{N}$ for

N with flow f. We can compute a function l by a simple breadth-first search. And, given a vertex $u \in \tilde{V}$, we can find a set of arcs leaving u in $\tilde{N}$ from (3.3) and (3.4) by using l. Crucial observation is that as long as we stretch flow-augmenting path starting from s by using arcs given in (3.3) and (3.4), vertices that are reachable from s are necessarily contained in $\tilde{V}$. Hence, in order to represent the referent network, we have only to compute and record the values of l, by which the referent network $\tilde{N}$ is represented implicitly and we can execute scanning and searching on $\tilde{N}$ straightforwardly. This is the subreferent method in [12]. This technique simplifies Dinic's method and Karzanov's much. (It costs relatively much to find and represent the referent network explicitly.)

Dinic's algorithm finds a maximal flow in a given referent network $\tilde{N}$ by depth-first search, which can be done in $O(|A||V|)$ time (therefore the total time complexity of Dinic's algorithm is $O(|A||V|^2)$). Here we employ the same type of labels as in the depth-first search method above. Of course, with incrementing flow through flow-augmenting path P, we find the vertex u to which flow can be still pushed along the part of P from s to u after flow augmentation. And in the next depth-first search step we re-use the labels given to vertices that lie on the part of P from s to u and restart searching from u.

In order to represent referent networks, we need an array of size $|V|$ for representing the function l. To execute the depth-first search, two arrays of size $|V|$ are employed where one is for recording labels and the other is for recording the arc for each vertex that should be scanned next on $\tilde{N}$.

Thus Dinic's algorithm can be implemented as easily as breadth-first search method in practice by employing the techniques presented here.

(d) Karzanov's algorithm (KARZ, (P-KARZ))

Algorithms following Dinic's improve the method to find a maximal flow in a given referent network, so that, in the following, we suppose that a referent network $\tilde{N}=(\tilde{V},\tilde{A},\tilde{c},s,t)$ is given (specifically, function l is given) and consider the method of finding a maximal flow. Also is supposed that functions $\partial^{\pm},\delta^{\pm}$ are those for $\tilde{N}$, and $\delta^{\pm}u$ denote linked lists of arcs as well as sets of arcs for vertex $u \in \tilde{V}$.

In the process of computing a maximal flow, Karzanov's algorithm admits not necessarily feasible flows, called preflows, which may not satisfy the flow conservation law (2.1). Given a function $\tilde{f}$ from $\tilde{A}$ to $\mathbf{R}$, which satisfies the capacity constraint, we define excess(u) for $u \in \tilde{V}-\{t\}$ by

$$\text{excess}(u)=\sum_{a\in\delta^- u}\tilde{f}(a)-\sum_{a\in\delta^+ u}\tilde{f}(a). \tag{3.6}$$

Then $\tilde{f}$ is a *preflow* if excess$(u) \geqq 0$ for $u \in \tilde{V}-\{s,t\}$ and excess$(s) \leqq 0$. (Note that $\tilde{f}$ is a feasible flow if excess$(u)=0$ for $u \in \tilde{V}-\{s,t\}$.) A vertex u is called an *unbalanced vertex* if excess$(u)>0$ and a *balanced vertex* if excess$(u)=0$. Source s is regarded as

the special unbalanced vertex.

In order to find a maximal flow, Karzanov's algorithm employs the following two basic operations PUSH(u) and BALANCE(u) for vertex $u\in\tilde{V}$. PUSH(u) is executed for an unbalanced vertex u, and increases flows on arcs in δ^+u until excess(u) becomes zero or all arcs in δ^+u become saturated. In increasing flows on arcs in δ^+u, arcs are chosen in the order of list δ^+u, and if the chosen arc comes to be saturated, we delete it and scan the next. Further, for executing BALANCE efficiently, we keep a stack for each vertex u in the referent network which records the history of increments of flows on arcs in δ^-u. Specifically, in PUSH(w), if flow on arc $a=(w,u)\in\tilde{A}$ is augmented by δ, we push the element (a,δ) to the stack for vertex u. An unbalanced vertex u is called *inactive* if PUSH(u) has been executed since the latest time when u became unbalanced, and *active* if not so. BALANCE(u) is executed for an inactive unbalanced vertex u, and makes excess(u) be zero by decreasing flows on arcs in δ^-u. That is, we iterate to pop the element (a,δ) from the stack for u, and to decrease flow on arc a by δ until excess(u) becomes zero. After making vertex u balanced, we delete vertex u and arcs in $\delta^{\pm}u$ from $\tilde{N}$.

On the basis of the above two basic procedures PUSH and BALANCE, Karzanov's algorithm finds a maximal flow as follows.

(i) excess(s):$=\infty$; excess(u):$=0$ for each vertex $u\in\tilde{V}-\{s,t\}$;

(ii) (PUSH step) execute PUSH(u) for each active unbalanced vertex u in the order closer to s;

(iii) if there is no unbalanced vertex in $\tilde{V}-\{s,t\}$, then halt: { *the flow is maximal* }; otherwise, consider l^* defined by $l^*=\min\{l(u)\mid u\in\tilde{V}-\{t\},\ u$: unbalanced$\}$;

(iv) (BALANCE step) execute BALANCE(u) for each inactive unbalanced vertex u with $l(u)=l^*$; return to (ii);

Due to the order of choosing vertices in (ii) and (iv), it can be shown that each vertex is balanced at most once, which guarantees that Karzanov's algorithm can find a maximal flow in $O(|V|^2)$ time.

Let us consider how we should implement Karzanov's algorithm. In order to represent referent networks, it also suffices in this case to compute and record the function l. Concerning active vertices, since they are processed in the manner like breadth-first search, we can keep them by a queue. Concerning inactive vertices, we can keep them by a stack. That is, in PUSH step, if vertex u is still unbalanced after the execution of PUSH(u), we push u into the stack, and in BALANCE step, we pop the vertex u of the stack and execute BALANCE(u) while the value of $l(u)$ does not change. These stack and queue are implemented by an array of size $|V|$ since those two sets represented by the stack and the queue are disjoint. Concerning the stacks for respective vertices u which record the history of increments of flows coming into

u in PUSH step, it takes $O(|V|^2)$ space if we record all increments simply. However, as is readily seen, we have only to record the newest increment on each arc, so that these stacks are implemented by two arrays of size $|A|$ for LIFO-lists and an array of size $|V|$ for head pointers to them. Also needed are an array of size $|V|$ expressing excess(u) and an array of size $|V|$ for recording the arc for each vertex that should be scanned next on the referent network $\tilde{N}$.

In this manner, Karzanov's method surely requires more memory storages than Dinic's, but is not so difficult to implement compared with Dinic's.

(e) Three Indians' algorithm (IND)

Malhotra, Kumar and Maheshwari [18] discovered another algorithm for finding a maximal flow in a given referent network $\tilde{N}$ in $O(|V|^2)$ time, which is conceptually simpler than Karzanov's. They introduced a *flow potential* pot(u) for each vertex u in the referent network with flow $\tilde{f}$ as the maximum extra flow that can be pushed through it, that is,

$$\text{(3.7)} \qquad \text{pot}(u)=\min\{\sum_{a\in\delta^+u}(\tilde{c}(a)-\tilde{f}(a)),\ \sum_{a\in\delta^-u}(\tilde{c}(a)-\tilde{f}(a))\},$$

where, concerning pot(s) and pot(t), we take only the first and second term in min, respectively. This algorithm works as follows.

(i) compute pot(u) for each vertex u in the referent network $\tilde{N}$ with flow $\tilde{f}$;
(ii) find a vertex u^* whose flow potential is minimum;
(iii) push flows from u^* to both s and t by pot(u^*);
(iv) delete each vertex u with pot(u) = 0 and arcs incident to u from the referent network with updating the flow potentials;
(v) if pot(s) = 0 or pot(t) = 0 then halt: { *the flow is maximal* }; else return to (ii);

In (iii), we can necessarily push flows from u^* to both s and t by pot(u^*) owing to the minimality of pot(u^*). In implementing (iii), we can push flow by the techniques like breadth-first search.

From a point of view of implementation, this algorithm has several points different from Dinic's and Karzanov's. In order to compute pot(u), we must find a set $\tilde{V}$ of vertices in the referent network explicitly, which can be done by additional breadth-first search. And also, we must manipulate arcs in the referent network in both directions extensively. Thus this method is conceptually simple, but the simplicity in the conceptual level does not apply directly to the efficient implementation.

(f) Galil and Naamad's algorithm (P-GN)

Galil and Naamad [11] discovered the algorithm to get a maximal flow in a given referent network in $O(|A|(\log|V|)^2)$ time by improving the method to find a maximal flow in Dinic's algorithm with the idea of path fragments. In the step of finding a maximal flow in Dinic's algorithm, a flow-augmenting path is found by depth-first

search and is divided into pieces of paths by deleting all saturated arcs after flow augmentation. Though these pieces of paths, called *path fragments* in [11], are still valid, Dinic's almost forgets them, and would re-discover some of them in the sequent steps. Therefore in Galil and Naamad's algorithm all the path fragments are recorded completely and used to reduce the complexity in the worst case. Each path fragment is represented by a 2-3 tree [1]. They showed the way of augmenting flows through path fragments with manipulating 2-3 trees, by which a maximal flow can be found in $O(|A|\ (\log|V|)^2)$ time.

We implement this method in a manner different from the methods above, where these are discussed in the next section.

4. Computational Results for Several Test Networks

4.1. Design of computational experiments

In this section, we investigate the practical efficiency of the maximum flow algorithms described in the last section by computational experiments. In order to compare the computational results, we must first consider what types of networks should be employed as test networks. This is because the running time for solving network problems depends not only upon the number of vertices and arcs in the network, but also upon the structure of the network itself.

In the computational experiments that have been made for the network problems so far, the so-called random networks have been employed often. Though random networks can be easily constructed in the computer, and the average behavior of the network algorithms for them may be analysed simply in some cases, they are rather different from what we think "random" networks and also from the real-life networks. Of course random networks are useful to characterize the network algorithms owing to the typical properties that random networks have. However, such computational experiments that employ only random networks as the test network should not be designed, which is insufficient to evaluate the algorithmic efficiency from the general point of view. We consider that one of the powerful methods to estimate the practical efficiency of the network algorithms is to use as test networks those obtained by processing the real-life networks such as road networks. Here "processing" means for instance changing source-sink pair, capacity and weight, adding or deleting vertices and arcs to the original ones, etc.

In this paper we consider three types of test networks, which are parametrized networks, real road networks and specially-structured networks. As parametrized networks, we consider random networks and grid networks, where we can vary the numbers of vertices and arcs. As real road networks, we provide the road networks of Tokyo metropolitan area and Kanto district. Taking account of the computational

results for the above two types of networks, we characterize the features of the methods using referent networks by employing specially-structured networks, called L-NET and T-NET. The computational results obtained for these specially-structured networks complement those for the above two types of networks.

Among the maximum flow algorithms presented in the last section, DFS, BFS, DNC, KARZ and IND are implemented by using FORTRAN (opt= 0) on HITAC M-200H (VOS3) at Computer Centre of University of Tokyo. The reported time below is the running time which does not contain the time for input and output, and is measured by subroutine CLOCK of FORTRAN on this system which is exact at least up to a millisecond. These methods are so simple that we code them without any subroutine in the main part.

On the other hand, we implement Dinic's algorithm (P-DNC) and Galil and Naamad's (P-GN) by using PASCAL (nocheck) on the same system with highly blocked structures, since we think that there would be no one who dare code P-GN without any subroutine by the fact that the program must inevitably be very long. Comparing the results of P-GN with those of P-DNC, and further, in some part, those of P-KARZ which is the PASCAL program of Karzanov's algorithm, we guess the efficiency of Galil and Naamad's algorithm. The reported time does not also contain the time for input and output, and is measured by function CLOCK of PASCAL on the same system which is exact up to a millisecond.

The memory requirements of the implemented codes as well as the rough sizes (i.e., the number of lines) of the main parts (i.e., the part except for input and output, etc.) of codes written by FORTRAN are shown in Table 4.1.

Table 4.1. Memory requirements: $c_v|V|+ c_a|A|$ and rough sizes of the implemented codes

codes	c_v	c_a	size
DFS	2	6	60
BFS	3	6	60
DNC P-DNC	4	6	75
KARZ	6	8	130
IND	10	6	220
P-GN	14	6	-

4.2. Parametrized networks

(i) Random networks

A random network $N(n,m,\hat{c})$ is constructed as follows where n, m and $\hat{c}$ are parameters: initially, n vertices are given, and then m arcs are randomly selected from possible $n(n-1)$ ones; capacities of arcs are random integers from 1 to $\hat{c}$, and source s and sink t $(s \neq t)$ are randomly chosen from n vertices.

Though this random network seems not so appropriate to use as a real-world model, we use this because Cheung and Glover et al. used the similar types of networks, and moreover, this random network has a typical property for the maximum flow problem. Intuitively and informally speaking, if a given random network is dense, the lengths of referent networks constructed from it are at most three or so, and a similar fact holds for a sparse random network. This implies that in a dense network the average running time of Dinic's and Karzanov's and three Indians' is $O(n^2)$ and similarly that the average time of breadth-first search method is $O(n^3)$.

We set $\hat{c}=100$, and consider the six cases where n and m are given by $m=10n$ for $n=250$, 500, 1000 and $m=0.2n^2$ for $n=100$, 200, 400. For each case, we solve ten different problems, i.e., the problems for ten different random networks, and the average running time is reported in Fig.4.1.

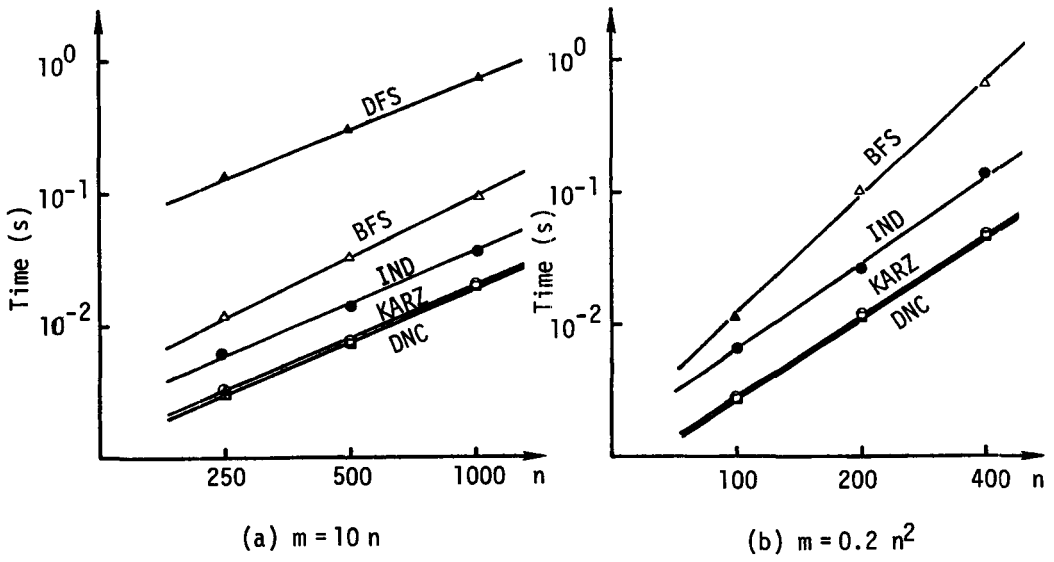

Fig. 4.1. Running time for random networks

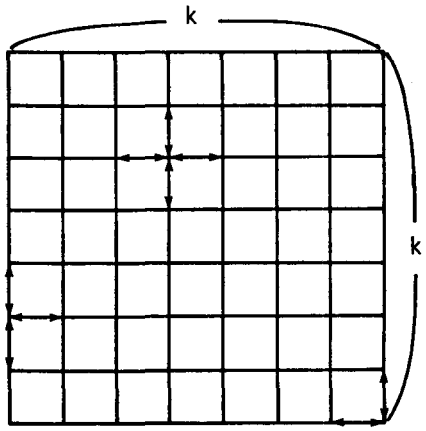

Fig. 4.2. $k \times k$ grid network

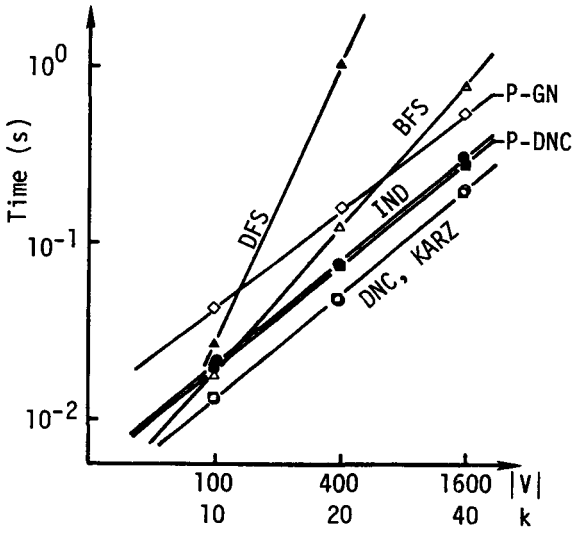

Fig. 4.3. Running time for $k \times k$ grid network

(ii) Grid networks

This network is a $k \times k$ square grid network, as depicted in Fig.4.2, where there are k^2 vertices and $4k(k-1)$ arcs, and a pair of source and sink is randomly determined and capacities of arcs are set to be random integers from 1 to $\bar{c}$. This grid network may be regarded as the model of a real traffic road network. By varying k, we can know what effect the sizes of such networks have on the running time.

We set $\bar{c} = 100$, and consider the cases of $k = 10$, 20, 40. For each case, we solve ten different problems and the average running time is shown in Fig.4.3.

4.3. Real road networks

The above two types of networks are not real-life networks but parametrized artificial ones. Here we provide real road networks of Tokyo metropolitan area and Kanto district.

Road network of Tokyo metropolitan area is shown in Fig.4.4. This network has 513 vertices and 849 arcs and capacity on each arc is set to be the width of its arc (or road). The number of ranks of capacities, valued from 50 to 500, is 13. On this network, we consider four multi-source and multi-sink problems T1, T2, T3 and T4. The problem T1 is from Yokohama to Chiba, and T2 is the reverse of T1. The problem T3 is from suburbs to Marunouchi, and T4 is the reverse of T3 (see Fig.4.4). The running time for the respective problems is depicted in Fig.4.5.

Road network of Kanto district is shown in Fig.4.6. This network consists of 6922 vertices and 10112 arcs. The number of ranks of capacities is 123. On this network,

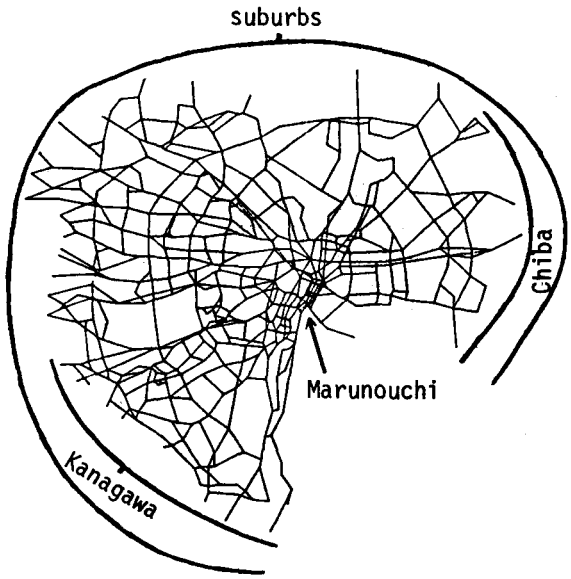

Fig. 4.4. Road network of Tokyo metropolitan area

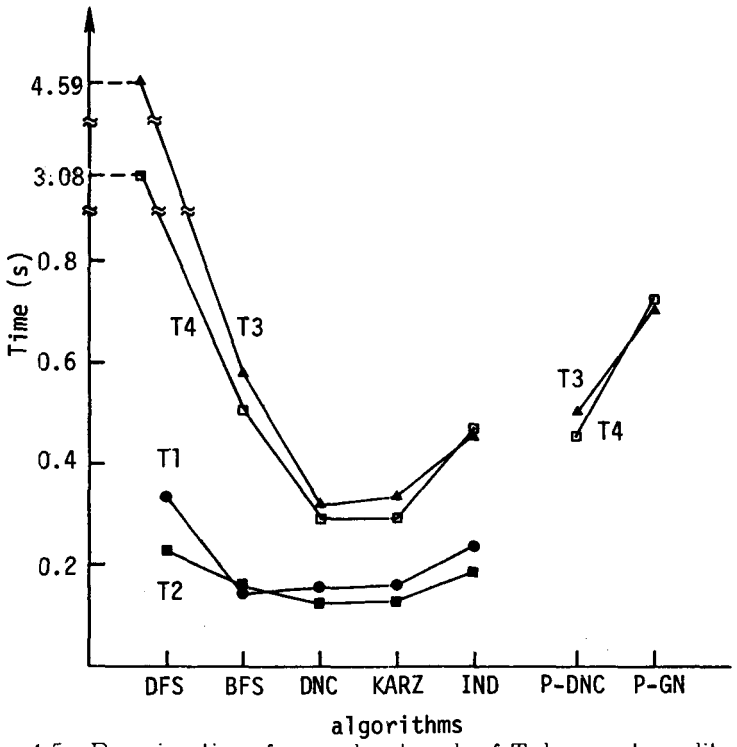

Fig. 4.5. Running time for road network of Tokyo metropolitan area

we consider two types of multi-source and multi-sink problems, K1 and K2. In the problem K1 (resp. K2), we consider five pairs of 10 (resp. 30) distinct vertices as a pair of sources and sinks which are randomly chosen. For each problem in K1 and

Fig. 4.6. Road network of Kanto district

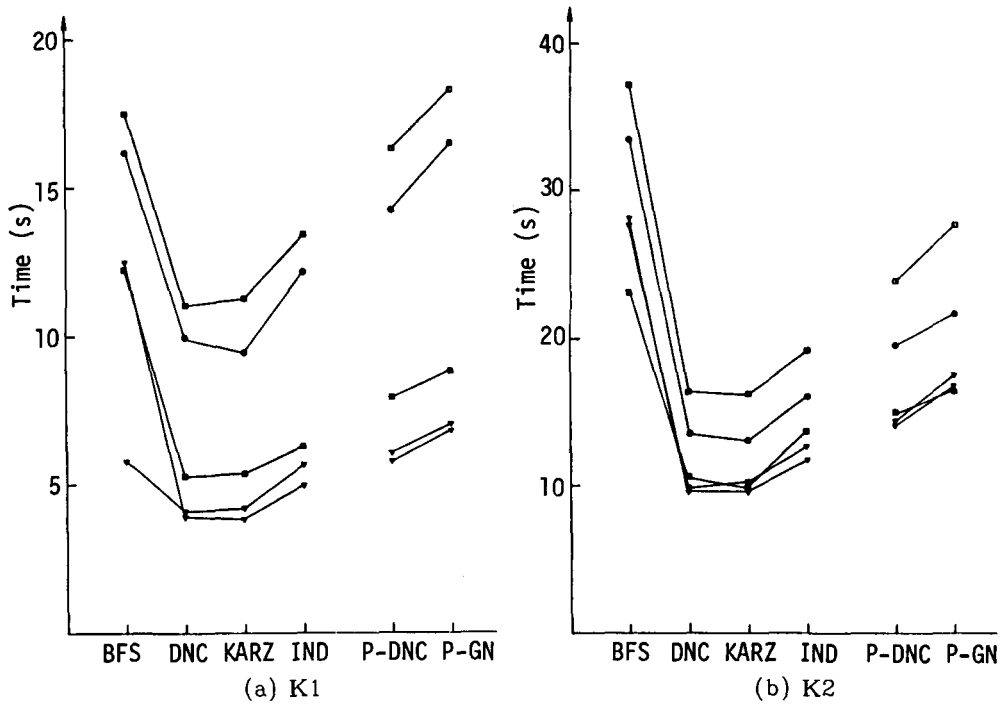

Fig. 4.7. Running time for road network of Kanto district

K2, the running time for the five problems of respective pairs of sources and sinks is shown in Fig.4.7.

4.4. Evaluation of the computational results for parametrized networks and real road networks

The above computational results show that Dinic's method and Karzanov's are the most efficient. Thus though Karzanov's algorithm seems to be much complicated at a first glance, practically it is simple enough to solve the problem quickly. Also is seen that breadth-first search method is rather effective for the small real road network.

Though Dinic's method is theoretically worse than Karzanov's and others for a dense network, we see that Dinic's method is efficient for random networks, even for dense ones. This is due to the fact that the length of the referent networks is at most three or so. In grid networks, the number of shortest flow-augmenting paths with the same length is many since grid network is so designed. But, in real road network, the number of them is not so many, because these real road networks are not so regularly structured as grid networks. Consequently, breadth-first search method is not efficient for grid networks, but may be rather well for small real road networks.

The results of depth-first search method is out of the question. Three Indians' method necessarily takes much more time than Dinic's and Karzanov's. This is mainly due to the facts noted in the description of the three Indians' method. Results of Galil and Naamad's method for grid network and real road network are poor, although these network are sparse. Apparently, manipulation of 2-3 trees takes too much time extraly.

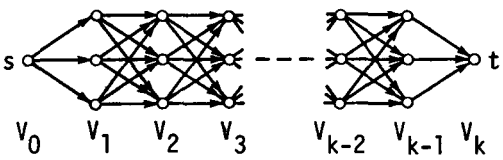

Fig. 4.8. L-NET

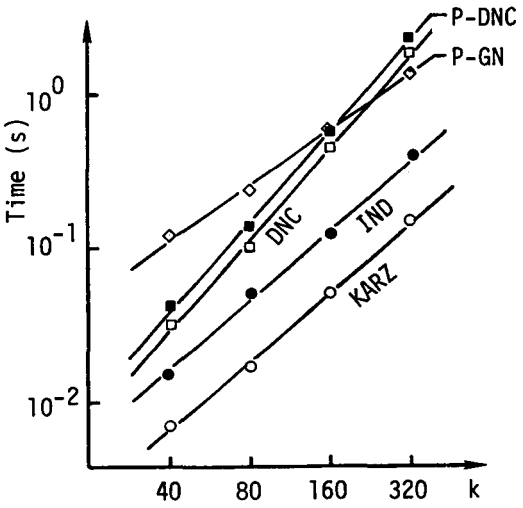

Fig. 4.9. Running time for L-NET

4.5. Specially-structured networks

(i) L-NET

The computational results of Dinic's algorithm for the above test networks are remarkably good. However, it is questionable whether Dinic's method is always the best. Reconsidering the above test networks, we see that those networks are sparse, or even if networks are dense, the lengths of referent networks constructed from it are bounded by a small constant.

In order to complement the estimation obtained from the computational results for the above test networks, we first consider the network L-NET which is defined as follows. In L-NET, two positive integers k and $\hat{c}$ are given as parameters. A set V of vertices and a set A of arcs in L-NET are given by

$$V=V_0\cup V_1\cup\cdots\cup V_k,\ V_0=\{s\},\ V_k=\{t\},\ |V_i|=3\quad(1\leqq i\leqq k-1) \tag{4.1}$$

$$A=\{(u,w)|\ u\in V_{i-1},\ w\in V_i,\ i=1,2,\ldots,\ k\}. \tag{4.2}$$

See Fig.4.8. The number of vertices is $3(k-1)+2$, and the number of arcs is $9(k-2)+6$. Capacities are random integers from 1 to $\hat{c}$. On L-NET, we test the methods using the referent networks, and consider the problem of finding a maximal flow where it should be noted that, in this case, a maximal flow is almost necessarily a maximum flow because either of sets of arcs incident to s and to t gets wholly saturated in many cases. The running time for finding a maximal flow on L-NET is reported in Fig.4.9 where we set $\hat{c}=10000$ and consider the cases of $k=40$, 80, 160 and 320.

(ii) T-NET

Next, in order to evaluate the computational efficiency of Galil and Naamad's

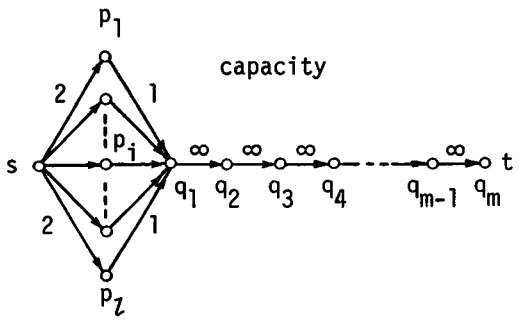

Fig. 4.10. T-NET

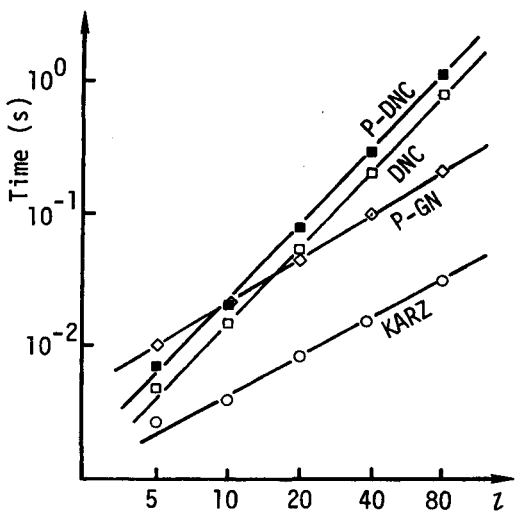

Fig. 4.11. Running time for T-NET

method, or to find out how much time is required for manipulating 2-3 trees, we provide a specially-structured network called T-NET. T-NET has two parameters l and m, and is defined as shown in Fig.4.10.

In T-NET, Dinic's finds a flow-augmenting path from q_1 to $t(=q_m)$ by scanning arcs one by one at each depth-first search, and takes $O(lm)$ time in total. Karzanov's takes only $O(l+m)$ time. Galil and Naamad's method takes $O((l+m)\log m)$ time, since this method records the path from q_1 to t at the first flow augmentation and use it in the subsequent steps. The running time is shown in Fig.11 where we set $m=10l$, and consider the cases of $l=5$, 10, 20, 40 and 80.

4.6. Evaluation of the computational results for specially-structured networks

Concerning the results for L-NET, we see that Dinic's is inefficient for L-NET, and Karzanov's method is the best. Galil and Naamad's method is faster than Dinic's in the case of $k=320$, but is not so if k is less than 160. Concerning the results for T-NET, Galil and Naamad's becomes faster than Dinic's as l is greater than 10.

Dinic's method may be much slower than the others in a specially-structured networks, but we consider that in the practical situation Dinic's would be very efficient, where its practical robustness can be informally justified as follows. Considering in detail how the theoretical worst-case bound of Dinic's algorithm is obtained, we see that its time bound is given by $O(\sum_{i=1}^{k}(|A|+l_i|\tilde{A}_i|))$ where l_i and $|\tilde{A}_i|$ are the length of, and the number of arcs of the referent network, respectively, in the ith phase, and k is the number of phases needed. Although, theoretically, we have $k=O(|V|)$, $l_i=O(|V|)$ and $|\tilde{A}_i|=O(|A|)$, which indicates that the worst-case bound of Dinic's is $O(|A||V|^2)$, we can ordinarily suppose that k and l_i are bounded by a constant, and further that, even when the original network is dense, the referent networks are not necessarily dense (probably $|\tilde{A}_i| \ll |A|$). Note that, if k and l are really bounded by a constant, then the time bound of Dinic's algorithm is $O(|A|)$!

Also seen is that Karzanov's method is very robust compared with Dinic's, as is assured from the theoretical worst-case time bound. We consider that we need not use Galil and Naamad's method in practice which is surely robust but in many cases inefficient.

5. Conclusion

Dinic's method and Karzanov's are the most efficient among the maximum flow algorithms implemented here and are recommended for the practical use. Dinic's method is very simple (easy to code even when it is compared with the labeling procedure such as breadth-first search method), and efficient enough in usual cases. If

one uses the data structures described in this paper, $4|V|+6|A|$ memory storages are needed. Karzanov's method is no less efficient than Dinic's and has the better worst-case bound. However, Karzanov's needs $6|V|+8|A|$ memory storages which is more than that of Dinic's.

Acknowledgement

The author wishes to thank Professor Masao Iri of the University of Tokyo for his constant guidance and valuable suggestions. He also wishes to thank Kazuo Murota for his many useful comments on the paper.

References

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D.: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[2] Cherkasskii, B. V.: Algorithm of Construction of Maximal Flow in a Network with Complexity of $O(V^2\sqrt{E})$ Operations (in Russian). *Mathematical Methods in Economics Studies*, Vol.7, Nauka, 1977, 117-125.

[3] Cheung, T.: Computational Comparison of Eight Methods for the Maximum Network Flow Problem. *ACM Transactions on Mathematical Software*, Vol.6, No.1 (1980), 1-16.

[4] Dinic, E. A.: Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation. *Soviet Mathematics Doklady*, Vol.11, No.5 (1970), 1277-1280.

[5] Edmonds, J., and Karp, R. M.: Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the Association for Computing Machinery*, Vol.19, No.2 (1972), 248-264.

[6] Even, S.: The Max Flow Algorithm of Dinic and Karzanov: An Exposition. *Technical Report MIT/LCS/TM-80*, Laboratory for Computer Science, Massachusetts Institute of Technology, 1976.

[7] Ford, L. R., Jr., and Fulkerson, D. R.: Maximal Flow through a Network. *Canadian Journal of Mathematics*, Vol.8, No.3 (1956), 399-404.

[8] Ford, L. R., Jr., and Fulkerson, D. R.: *Flows in Networks*. Princeton University Press, Princeton, New Jersey, 1962.

[9] Galil, Z.: A New Algorithm for the Maximal Flow Problem. *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, Ann-Arbor, Michigan, 1978, 231-245. See also: Galil, Z.: An $O(V^{5/3}E^{2/3})$ Algorithm for the Maximal Flow Problem. *Acta Informatica*, Vol.14 (1980), 221-242.

[10] Galil, Z.: On the Theoretical Efficiency of Various Network Flow Algorithms. *Theoretical Computer Science,* Vol.14 (1981), 103-111.

[11] Galil, Z., and Naamad, A.: Network Flow and Generalized Path Compression. *Proceedings of the 11th Annual ACM Symposium on Theory of Computing,* Atlanta, Georgia, 1979, 13-26. See also: Galil, Z., and Naamad, A.: An $O(EV \log^2 V)$ Algorithm for the Maximal Flow Problem. *Journal of Computer and System Sciences,* Vol.21 (1980), 203-217.

[12] Glover, F., Klingman, D., Mote, J., and Whitman, D.: Comprehensive Computer Evaluation and Enhancement of Maximum Flow Algorithms. *Research Report CCS 356,* Center for Cybernetic Studies, University of Texas, Austin, 1979. See also: Glover, F., Klingman, D., Mote, J., and Whitman, D.: An Extended Abstract of an Indepth Algorithmic and Computational Study for Maximum Flow Problems. *Discrete Applied Mathematics,* Vol.2 (1980), 251-254.

[13] Imai, H.: On the Efficient Implementation of Network Algorithms – Especially on the Maximum Flow Problem (in Japanese). Graduation Thesis, Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo, 1981.

[14] Imai, H.: Practical Estimation of Maximum Flow Algorithms (in Japanese). *Proceedings of the 1981th Fall Conference of the Operations Research Society of Japan,* E-1-3, 112-113.

[15] Iri, M.: *Network Flow, Transportation and Scheduling – Theory and Algorithms.* Academic Press, London, 1969.

[16] Iri, M., et al.: Fundamental Studies of the Techniques for Processing Network Problems in Operations Research by Computers (in Japanese). *Technical Report Series T-73-1,* Operations Research Society of Japan, 1973.

[17] Karzanov, A. V.: Determining the Maximal Flow in a Network by the Method of Preflows. *Soviet Mathematics Doklady,* Vol.15, No.2 (1974), 434-437.

[18] Malhotra, V. M., Kumar, M. P., and Maheshwari, S. N.: An $O(|V|^3)$ Algorithm for Finding Maximum Flows in Networks. *Information Processing Letters,* Vol.7, No.6 (1978), 277-278.

[19] Manabe, R.: Recent Topics on the Maximum Flow Algorithms (in Japanese). *Communications of the Operations Research Society of Japan,* Vol.25, No.12 (1980), 772-779.

[20] Sleator, D. D.: An $O(mn \log n)$ Algorithm for Maximum Network Flow. Ph.D. Thesis, Computer Science Department, Stanford University, 1980.

[21] Sleator, D. D., and Tarjan, R. E.: A Data Structure for Dynamic Trees. *Proceedings of the 13th Annual ACM Symposium on Theory of Computing,* Milwaukee, Wisconsin, 1981, 114-122.

Hiroshi IMAI: Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan 113