Computational Investigations of Maximum Flow Algorithms

by
Ravindra K. Ahuja
Murali Kodialam
Ajay K. Mishra
James B. Orlin

WP #3811-95 April 1995

COMPUTATIONAL INVESTIGATIONS OF MAXIMUM FLOW ALGORITHMS

Ravindra K. Ahuja, Murali Kodialam, Ajay K. Mishra, and James B. Orlin

ABSTRACT

The maximum flow algorithm is distinguished by the long line of successive contributions researchers have made in obtaining algorithms with incrementally better worst-case complexity. Some, but not all, of these theoretical improvements have produced improvements in practice. The purpose of this paper is to test some of the major algorithmic ideas developed in the recent years and to assess their utility on the empirical front. However, our study differs from previous studies in several ways. Whereas previous studies focus primarily on CPU time analysis, our analysis goes further and provides detailed insight into algorithmic behavior. It not only observes how algorithms behave but also tries to explain why algorithms behave that way. We have limited our study to the best previous maximum flow algorithms and some of the recent algorithms that are likely to be efficient in practice. Our study encompasses ten maximum flow algorithms and five classes of networks. The augmenting path algorithms tested by us include Dinic's algorithm, the shortest augmenting path algorithm, and the capacity scaling algorithm. The preflow-push algorithms tested by us include Karzanov's algorithm, three implementations of Goldberg-Tarjan algorithm, and three versions of Ahuja-Orlin-Tarjan excess-scaling algorithms. Among many findings, our study concludes that the preflow-push algorithms are substantially faster than other classes of algorithms, and the highest-label preflow-push algorithm is the fastest maximum flow algorithm for which the growth rate in the computational time is $O(n^{1.5})$ on four out of five of our problem classes. Further, in contrast to the results of the worst-case analysis of maximum flow algorithms, our study finds that the time to perform relabel operations (or constructing the layered networks) takes at least as much computation time as that taken by augmentations and/or pushes.

1. INTRODUCTION

The maximum flow problem is one of the most fundamental problems in network optimization. Its intuitive appeal, mathematical simplicity, and wide applicability has made it a popular research topic among mathematicians, operations researchers and computer scientists.

The maximum flow problem arises in a wide variety of situations. It occurs directly in problems as diverse as the flow of commodities in pipeline networks, parallel machine scheduling, distributed computing on multi-processor computers, matrix rounding problems, baseball elimination problem, and the statistical security of data. The maximum flow problem also occurs as a subproblem while solving more complex problems such as the minimum cost flow problem and the generalized flow problem. The maximum flow problem also arises in combinatorics, with applications to network connectivity, and to matchings and coverings in bipartite networks. The book by Ahuja, Magnanti and Orlin [1993] describes these and other applications of the maximum flow problem.

Due to its wide applicability, designing efficient algorithms for the maximum flow problem has been a popular research topic. The maximum flow problem is distinguished by the long line of successive contributions researchers have made in obtaining algorithms with incrementally better worst-case complexity (see, e.g., Ahuja, Magnanti and Orlin [1993] for a survey of these contributions). Indeed, no other fundamental network optimization problem has witnessed as many incremental improvements in solution techniques as has the maximum flow problem. Some, but not all, of these theoretical improvements have produced improvements in practice. The purpose of this paper is to test some of the major algorithmic ideas developed in recent years and to assess their utility in practice.

Prior to the advent of preflow-push algorithms due to Goldberg and Tarjan [1986], Dinic's [1970] and Karzanov's [1974] algorithms were considered to be the fastest maximum flow algorithms. Subsequent developments from 1974 to 1986 included several algorithms with improved worst-case complexity, but these theoretical improvements did not translate into empirically faster algorithms. The novel concept of distance labels, in contrast to the *layered (or, referent) network* concept in Dinic's and Karzanov's algorithms, proposed by Goldberg and Tarjan [1986] led to breakthroughs both theoretically as well as empirically. Using distance labels in preflow-push algorithms, Goldberg and Tarjan [1986], and subsequently, Ahuja and Orlin [1989], Ahuja, Orlin and Tarjan [1989], Cheriyan and Hagerup [1989], and Alon [1990], obtained maximum flow algorithms with incrementally improved worst-case complexities. Some of these algorithms are also substantially faster than Dinic's and Karzanov's algorithms empirically, as the computational testings of Derigs and Meier [1989] and Anderson and Setubal [1992] revealed.

In this paper, we present the results of an extensive computational study of maximum flow algorithms. Our study differs from the previous computational studies in several ways. Whereas the previous studies focus primarily on CPU time analysis, our analysis goes farther and provides detailed insight into algorithmic behavior. It observes how algorithms behave and also tries to explain the behavior. We perform our empirical study using the *representative operation counts*, as presented in Ahuja and Orlin [1995], and Ahuja, Magnanti and Orlin [1993]. The use of representative operation counts allows us

(i) to identify bottleneck operations of an algorithm; (ii) to facilitate the determination of the growth rate of an algorithm; and (iii) to provide a fairer comparison of algorithms. This approach is one method of incorporating computation counts into an empirical analysis.

We have limited our study to the best previous maximum flow algorithms and some recent algorithms that are likely to be efficient in practice. Our study encompasses ten maximum flow algorithms whose discoverers and worst-case time bounds are given in Table 1.1. In the table, we denote by n, the number of nodes; by m, the number of arcs; and by U, the largest arc capacity in the network. For Dinic's and Karzanov's algorithm, we used the computer codes developed by Imai [1983], and for other algorithms we developed our own codes.

S.No.	Algorithm	Discoverer(s)	Running Time
1.	Dinic's algorithm	Dinic [1970]	$O(n^2m)$
2.	Karzanov's algorithm	Karzanov [1974]	$O(n^3)$
3.	Shortest augmenting path algorithm	Ahuja and Orlin [1991]	$O(n^2m)$
4.	Capacity scaling algorithm	Gabow [1985] and Ahuja and Orlin [1991]	O(nm log U)
	Preflow-push algorithms		
5.	Highest-label algorithm	Goldberg and Tarjan [1986]	$O(n^2m^{1/2})$
6.	FIFO algorithm	Goldberg and Tarjan [1986]	$O(n^3)$
7.	Lowest-label algorithm	Goldberg and Tarjan [1986]	$O(n^2m)$
	Excess-scaling algorithms		
8.	Original excess-scaling	Ahuja and Orlin [1989]	$O(nm + n^2 \log U)$
9.	Stack-scaling algorithm	Ahuja, Orlin and Tarjan [1989]	$O\left(nm + \frac{n^2 \log U}{\log \log U}\right)$
10.	Wave-scaling algorithm	Ahuja, Orlin and Tarjan [1989]	$O\left(nm + n^2\sqrt{\log U}\right)$

Table 1.1 Worst-case bounds of algorithms investigated in our study.

We tested these algorithms on a variety of networks. We carried out extensive testing using grid and layered networks, and also considered the DIMACS benchmark instances. We summarize in Tables 1.2 and 1.3 respectively the CPU times taken by the maximum flow algorithms to solve maximum flow problems on layered and grid networks. Figure 1.1 plots the CPU times of some selected algorithms applied to the grid networks. From this data and the additional experiments described in Sections 10 and 11, we can draw several conclusions, which are given below. These conclusions apply to problems obtained using all network generators, unless stated otherwise .

n	d	Shortest Aug. Path	Capacity Scaling	Dinic	PREFLOW-PUSH			EXCESS SCALING			Karzanov
					Highest Label	FIFO	Lowest Label	Excess Scaling	Stack Scaling	Wave Scaling	
500	4	0.21	0.62	0.24	0.06	0.08	0.17	0.14	0.13	0.15	0.14
1000	4	0.67	2.05	0.72	0.15	0.20	0.52	0.36	0.31	0.37	0.40
2000	4	2.09	5.84	2.19	0.33	0.49	1.60	0.94	0.75	0.93	1.19
3000	4	3.96	11.52	4.14	0.50	0.80	3.23	1.59	1.21	1.63	2.36
4000	4	7.27	20.63	7.78	0.70	1.29	6.25	2.71	1.93	2.79	4.81
5000	4	13.00	52.97	13.80	0.90	2.67	12.78	5.91	3.70	6.50	9.84
6000	4	11.47	34.52	12.11	1.05	1.84	9.24	4.05	2.78	4.14	6.99
7000	4	15.45	41.26	16.37	1.30	2.44	13.20	5.26	3.61	5.43	9.67
8000	4	19.78	62.30	21.01	1.59	2.98	17.98	6.71	4.50	7.13	13.21
9000	4	26.77	78.22	28.47	1.77	4.16	25.67	9.08	5.87	10.06	18.55
10000	4	25.64	68.45	27.52	1.78	3.74	22.88	8.91	5.79	9.33	16.43
Mean		11.48	34.40	12.21	0.92	1.88	10.32	4.15	2.78	4.41	7.60
500	6	0.41	1.03	0.45	0.09	0.11	0.32	0.20	0.17	0.21	0.23
1000	6	1.20	3.12	1.27	0.19	0.26	0.95	0.49	0.39	0.48	0.58
2000	6	3.58	8.09	3.83	0.40	0.59	2.94	1.29	0.90	1.28	1.76
3000	6	6.46	13.78	6.86	0.61	0.92	5.22	2.19	1.42	2.03	3.00
4000	6	10.76	23.65	11.45	0.87	1.51	9.21	3.54	2.29	3.39	5.34
5000	6	13.78	26.71	14.93	1.06	1.66	11.33	4.38	2.68	4.19	6.45
6000	6	19.22	38.36	20.30	1.32	2.20	16.54	6.11	3.63	5.92	9.43
7000	6	27.22	57.09	29.30	1.56	3.16	25.09	8.86	5.09	9.03	14.76
8000	6	34.63	76.31	37.47	1.88	3.76	32.41	10.59	6.06	10.48	18.64
9000	6	29.04	47.88	31.14	1.74	2.93	22.76	8.43	4.96	7.51	12.01
10000	6	46.79	107.92	49.81	2.30	5.15	44.33	14.58	8.11	14.91	26.03
Mean		17.55	36.72	18.80	1.09	2.02	15.55	5.52	3.25	5.40	8.93
500	8	0.51	1.38	0.55	0.11	0.13	0.40	0.23	0.19	0.22	0.22
1000	8	1.46	3.45	1.59	0.22	0.29	1.14	0.56	0.42	0.53	0.61
2000	8	4.41	8.06	4.65	0.47	0.59	3.34	1.43	0.94	1.27	1.43
3000	8	8.63	16.22	9.13	0.74	0.97	6.69	2.55	1.58	2.27	3.05
4000	8	15.20	30.68	15.93	1.04	1.73	12.89	4.74	2.73	4.55	6.43
5000	8	23.68	56.43	25.09	1.46	3.19	21.47	7.27	4.21	7.52	11.82
6000	8	26.66	45.67	28.90	1.61	2.46	22.46	7.53	4.22	7.09	10.94
7000	8	41.92	83.05	45.42	2.02	4.22	38.63	12.76	6.66	12.98	20.60
8000	8	42.94	84.73	46.51	2.12	3.78	37.47	12.00	6.46	11.77	19.42
9000	8	55.32	108.73	59.83	2.57	5.46	50.98	16.03	8.44	16.39	27.47
10000	8	68.36	149.13	72.52	2.91	6.79	64.73	20.33	10.17	21.55	32.72
Mean		26.28	53.41	28.19	1.39	2.69	23.66	7.77	4.18	7.83	12.25
500	10	0.62	1.56	0.70	0.11	0.13	0.48	0.25	0.20	0.24	0.26
1000	10	1.71	3.59	1.93	0.26	0.30	1.35	0.59	0.44	0.54	0.58
2000	10	6.11	11.37	6.42	0.58	0.76	4.82	1.84	1.19	1.69	2.18
3000	10	10.34	16.75	11.57	0.84	1.07	8.17	2.94	1.78	2.57	3.62
4000	10	17.93	33.02	18.87	1.22	1.72	14.54	4.80	2.74	4.40	6.12
5000	10	23.56	43.23	25.85	1.47	1.94	18.79	6.03	3.34	5.39	7.97
6000	10	39.72	83.89	41.46	2.01	4.03	35.53	11.28	6.03	11.56	17.08
7000	10	44.22	75.38	47.23	2.16	3.30	36.54	11.55	5.88	10.68	16.41
8000	10	59.80	121.97	63.52	2.56	5.12	52.03	16.18	8.11	15.81	25.14
9000	10	64.85	118.98	69.94	2.72	4.70	54.64	17.49	8.47	16.81	24.73
10000	10	99.24	220.78	106.80	3.41	10.08	94.28	31.02	13.65	32.08	48.50
Mean		33.46	66.41	35.84	1.58	3.01	29.20	9.45	4.71	9.25	13.87

Table 1.2. CPU time (in seconds on Convex) taken by algorithms on the layered network.

n	d	Shortest Aug. Path	Capacity Scaling	Dinic	PREFLOW-PUSH			EXCESS SCALING			Karzanov
					Highest Label	FIFO	Lowest Label	Excess Scaling	Stack Scaling	Wave Scaling	
500	5	0.41	1.71	0.39	0.11	0.15	0.27	0.21	0.21	0.23	0.33
1000	5	1.25	4.81	1.27	0.28	0.38	0.82	0.54	0.54	0.58	1.02
2000	5	3.84	15.17	3.97	0.76	1.12	2.62	1.54	1.47	1.68	3.18
3000	5	7.80	33.39	7.14	1.32	1.97	5.29	2.60	2.49	2.80	5.54
4000	5	15.89	74.02	13.82	1.98	3.14	11.67	4.50	4.01	4.93	12.37
5000	5	19.74	93.14	18.30	2.89	4.31	13.20	5.69	5.30	6.24	14.33
6000	5	26.80	110.53	24.61	3.65	5.80	21.31	7.86	7.29	8.72	20.05
7000	5	33.09	137.19	31.64	4.25	6.74	26.35	9.52	8.60	10.58	25.99
8000	5	39.07	167.13	40.24	4.88	8.11	30.13	11.36	10.26	12.82	31.61
9000	5	46.81	202.26	42.18	5.55	9.53	36.83	12.91	11.81	14.40	35.85
10000	5	67.48	283.88	57.37	6.94	11.43	52.41	16.40	14.85	18.24	51.58
Mean		23.83	102.11	21.90	2.96	4.79	18.26	6.65	6.07	7.38	18.35

Table 1.3. CPU time (in seconds on Convex) taken by algorithms on the grid network.

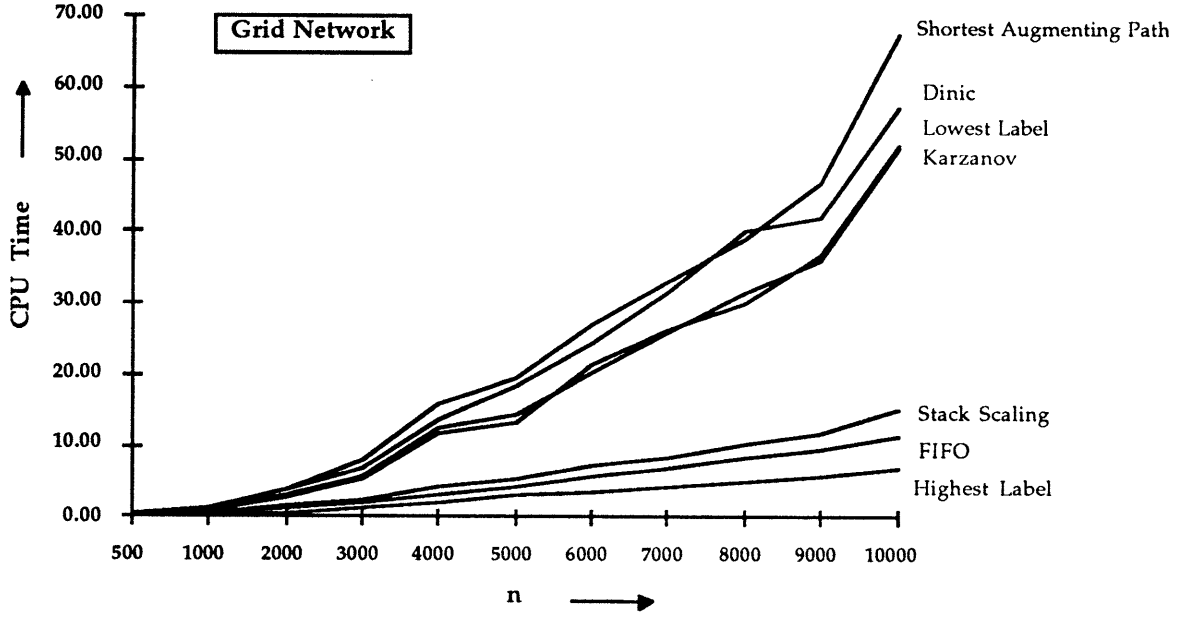


Figure 1.1. CPU time (in seconds) taken by the algorithms on grid network.

1. The preflow-push algorithms generally outperform the augmenting path algorithms and their relative performance improves as the problem size gets bigger.

2. Among the three implementations of the Goldberg-Tarjan preflow-push algorithms we tested, the highest-label preflow-push algorithm is the fastest. In other words, among these three algorithms, the highest-label preflow-push algorithm has the best worst-case complexity while simultaneously having the best empirical performance.

3. In the worst-case, the highest-label preflow-push algorithm requires $O(n^2\sqrt{m})$, but its empirical running time is $O(n^{1.5})$ on four of the five classes of problems that we tested.

4. All the preflow-push algorithms have a set of two "representative operations": (i) performing pushes; and (ii) relabels of the nodes. We describe representative operations in Section 5 of this paper. See also Ahuja and Orlin [1995]. Though in the worst-case, performing the pushes is the bottleneck operation, we find that empirically this time is no greater than the relabel time. This observation suggests that the dynamic tree implementations of the preflow-push algorithms worsen the running time in the practice, though they improve the worst-case running time.

5. We find that the number of nonsaturating pushes is .8 to 6 times the number of saturating pushes.

6. The excess-scaling algorithms improve the worst-case complexity of the Goldberg-Tarjan preflow-push algorithms, but this does not lead to an improvement empirically. We observed that the three excess-scaling algorithms tested by us are somewhat slower than the highest-label preflow-push algorithm. We find the stack-scaling algorithm to be the fastest of the three excess-scaling algorithms, but it is on the average twice slower than the highest-label preflow-push algorithm.

7. The running times of Dinic's algorithm and the shortest augmenting path algorithm are comparable, which is consistent with the fact that both algorithms perform the same sequence of augmentations (see Ahuja and Orlin [1991]).

8. Though in the worst-case Dinic's algorithm and the successive shortest path algorithm perform $O(nm)$ augmentations and take $O(n^2m)$ time, empirically we find that they perform no more than $O(n^{1.6})$ augmentations and their running times are bounded by $O(n^2)$.

9. Dinic's and the successive shortest path algorithms have two representative operations: (i) performing augmentations whose worst-case complexity is $O(n^2m)$; and (ii) relabeling the nodes whose worst-case complexity is $O(nm)$. We find that empirically the time to relabel the nodes grows faster than the time for augmentations. This explains why the capacity scaling algorithms (which decreases the worst-case running

time of augmentations at the expense of increasing the relabel time) do not improve the empirical running time over Dinic's algorithm.

2. NOTATION AND DEFINITIONS

We consider the maximum flow problem over a network G = (N, A) with N as the *node set* and A as the *arc set*. Let n = |N| and m = |A|. The *source* s and the *sink* t are two distinguished nodes of the network. Let u_{ij} denote the *capacity* of each arc (i, j) $\in$ A. We assume that u_{ij} is integral and finite. Some of the algorithms tested by us (namely, the capacity scaling and excess scaling algorithms) require that capacities are integral while other algorithms don't. Let $U = \max\{u_{ij}: (i, j) \in A\}$. We define the *arc adjacency list* A(i) of node i $\in$ N as the set of arcs directed out of node i, i.e., $A(i) = \{(i, k) \in A : k \in N\}$.

A *flow* x is a function $x : A \rightarrow R$ satisfying

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = 0 \text{ for all } i \in N - \{s, t\}, \tag{2.1}$$

$$\sum_{\{i:(i,t)\in A\}} x_{it} = v, \tag{2.2}$$

$$0 \le x_{ij} \le u_{ij} \text{ for all } (i, j) \in A, \tag{2.3}$$

for some $v \ge 0$. The maximum flow problem is to determine a flow for which its value v is maximized.

A *preflow* x is a function $x : A \rightarrow R$ satisfying (2.2), (2.3), and the following relaxation of (2.1):

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} \ge 0 \text{ for all } i \in N - \{s, t\}. \tag{2.4}$$

We say that a preflow x is *maximum* if its associated value v is maximum. The preflow-push algorithms considered in this paper maintain a preflow at each intermediate stage. For a given preflow x, we define for each node i $\in$ N - {s, t}, its *excess*

$$e(i) = \sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji}. \tag{2.5}$$

A node with positive excess is referred to as an *active* node. We use the convention that the source and sink nodes are never active. We define the *residual capacity* r_{ij} of any arc (i, j) $\in$ A with respect to the given preflow x as $r_{ij} = (u_{ij} - x_{ij}) + x_{ji}$. Notice that the residual capacity u_{ij} has two components : (i) $(u_{ij} - x_{ij})$, the unused capacity of arc (i, j), and (ii) the current flow x_{ji} on arc (j, i), which we can cancel to increase the flow from node i to node j. We refer to the network G(x) consisting of the arcs with positive residual capacities as the *residual network*.

A *path* is a sequence of distinct nodes (and arcs) i_1-i_2- ... -i_r satisfying the property that for all $1 \le p \le r-1$, either $(i_1, i_2) \in A$ or $(i_2, i_1) \in A$. A *directed path* is an "oriented" version of the path in the sense that for any consecutive nodes i_k and i_{k+1} in the walk, $(i_k, i_{k+1}) \in A$. An *augmenting path* is a directed path in which each arc has a positive residual capacity.

A *cut* is a partition of the node set N into two parts, S and $\bar{S}$ = N-S. Each cut $[S, \bar{S}]$ defines a set of arcs consisting of those arcs that have one endpoint in S and another in $\bar{S}$. An *s-t cut* $[S, \bar{S}]$ is a cut satisfying the property that $s \in S$ and $t \in \bar{S}$. Any (i, j) in $[S, \bar{S}]$ is a *forward arc* if $i \in S$ and $j \in \bar{S}$ and is a *backward arc* if $i \in \bar{S}$ and $j \in S$.

3. LITERATURE SURVEY

In this section, we present a brief survey of the theoretical and empirical developments of the maximum flow problem.

Theoretical Developments

The maximum flow problem was first studied by Ford and Fulkerson [1956], who developed the well-known labeling algorithm, which proceeds by sending flows along augmenting paths. The labeling algorithm runs in pseudo-polynomial time. Edmonds and Karp [1972] suggested two polynomial-time specializations of the labeling algorithm: the first algorithm augments flow along shortest augmenting paths and runs in $O(nm^2)$ time; the second algorithm augments flow along paths with maximum residual capacity and runs in $O(m^2 \log U)$ time. Independently, Dinic [1970] introduced the concept of shortest path networks, called *layered networks*, and showed that by constructing *blocking flows* in layered networks, a maximum flow can be obtained in $O(n^2m)$ time.

Researchers have made several subsequent improvements in maximum flow algorithms by developing more efficient algorithms to establish blocking flows in layered networks. Karzanov [1974] introduced the concept of preflows and showed that an implementation that maintains preflows and pushes flows from nodes with excesses obtains a maximum flow in $O(n^3)$ time. Subsequently, Malhotra, Kumar and Maheshwari [1978] presented a conceptually simpler $O(n^3)$ time algorithm. Cherkassky [1977] and Galil [1980] presented further improvements of Karzanov's algorithm that respectively run in $O(n^2m^{1/2})$ and $O(n^{5/3}m^{2/3})$ time.

The search for more efficient maximum flow algorithms led to the development of new data structures for implementing Dinic's algorithm. The first such data structure was suggested by Shiloach [1978], and Galil and Naamad [1980], and the resulting implementations ran in $O(nm \log^2 n)$ time. Sleator and Tarjan [1983] improved this approach by using the dynamic tree data structure, which yielded an O(nm log n) time algorithm. All of these data structures are quite sophisticated and require substantial overheads, which limits their practical utility. Pursuing a different approach, Gabow [1985] incorporated scaling technique into Dinic's algorithm and developed an O(nm log U) algorithm.

A set of new maximum flow algorithms emerged with the development of distance labels by Goldberg and Tarjan [1986] in the context of preflow-push algorithms. Distance labels were easier to manipulate than layered networks and led to more efficient algorithms both theoretically and empirically. Goldberg and Tarjan suggested *FIFO* and *highest label* preflow-push algorithms, both of which ran in $O(n^3)$ time using simple data structures and in $O(nm \log(n^2/m))$ time using the dynamic tree data structures. Cheriyan and Maheshwari [1989] subsequently showed that the highest-label preflow-push algorithm actually runs in $O(n^2\sqrt{m})$ time. Incorporating *excess-scaling* into the preflow-push algorithms, Ahuja and Orlin [1989] obtained an $O(nm + n^2 \log U)$ algorithm. Subsequently, Ahuja, Orlin and Tarjan [1989] developed two improved versions of the excess-scaling algorithms namely, (i) the *stack-scaling algorithm* with a time bound of $O(nm + (n^2 \log U)/(\log \log U))$, and (ii) the *wave-scaling algorithm* with a time bound of $O(nm + (n^2 \log U)^{1/2})$. Cheriyan and Hagerup [1989], and Alon [1990] gave further improvements of these scaling algorithms. Goldfarb and Hao [1990 and 1991] describe polynomial time primal simplex algorithms that solves the maximum flow problem in $O(n^2m)$ time, and Goldberg , Grigoriadis, and Tarjan [1991] describe an $O(nm \log n)$ implementation of the first of these algorithms using the dynamic trees data structure.

Empirical Developments

We now summarize the results of the previous computational studies conducted by a number of researchers including Hamacher [1979], Cheung [1980], Glover, Klingman, Mote and Whitman [1983, 1984], Imai [1983], Goldfarb and Grigoriadis [1988], Derigs and Meier [1989], Anderson and Setubal [1992], Nguyen and Venkateswaran [1992], and Badics, Bodos and Cepek [1992].

Hamachar [1979] tested Karzanov's algorithm versus the labeling algorithm and found Karzanov's algorithm to be substantially superior to the labeling algorithm. Cheung [1980] conducted an extensive study of maximum flow algorithms including Dinic's, Karzanov's and several versions of the labeling algorithm including the maximum capacity augmentation algorithm. This study found Dinic's and Karzanov's algorithms to be the best algorithms, and the maximum capacity augmentation algorithm slower than both the depth-first and breadth-first labeling algorithms.

Imai [1983] performed another extensive study of the maximum flow algorithms and his results were consistent with those of Cheung [1980]. However, he found Karzanov's algorithm to be superior to Dinic's algorithm for most problem classes. Glover, Klingman, Mote and Whitman [1983, 1984] and Goldfarb and Grigoriadis [1988] have tested network simplex algorithms for the maximum flow problem.

Researchers have also tested implementations of Dinic's algorithm using sophisticated data structures. Imai [1983] tested Galil and Naamad's [1980] data structure, and Sleator and Tarjan [1983] tested their dynamic tree data structure. Both the studies observed that these data structures slowed down the original Dinic's algorithm by a constant factor. Until 1985, Dinic's and Karzanov's algorithms were widely considered to be the fastest algorithms for solving the maximum flow problem. For sparse graphs, Karzanov's algorithm was comparable to Dinic's algorithm, but for dense graphs, Karzanov's algorithm was faster than Dinic's algorithm.

We now discuss computational studies that tested more recently developed maximum flow algorithms. Derigs and Meier [1989] implemented several versions of Goldberg and Tarjan's algorithm. They found that Goldberg and Tarjan's algorithm (using stack or dequeue to select nodes for pushing flows) is substantially faster than Dinic's and Karzanov's algorithms. In a similar study, Anderson and Setubal [1992] find different versions (FIFO, highest label, stack, and highest label) to be best for different classes of networks and queue implementations to be about 4 times faster than Dinic's algorithm.

Nguyen and Venkateswaran [1992] report computational investigations with 10 variants of the preflow-push maximum flow algorithm. They find that FIFO and highest-label implementations together with periodic global updates have the best overall performance. Badics, Boros and Cepek [1992] compared Cheriyan and Hagerup's [1989] PLED (Prudent Linking and Excess Diminishing) algorithm and Goldberg-Tarjan's algorithm with and without dynamic trees. They found that Goldberg-Tarjan's algorithm outperformed PLED algorithm. Further, Goldberg-Tarjan's algorithm without dynamic trees was generally superior to the algorithm with dynamic trees; but they also identify a class of networks where the dynamic tree data structure does improve the algorithm performance.

In contrast to these studies, we find that our implementation of the highest-label preflow-push is consistently superior to all other implementations of the preflow-push algorithm on the five problem classes considered in our study. We find the highest-label preflow-push algorithm to be about 7 to 20 times faster than Dinic's algorithm and about 6 to 8 times faster than Karzanov's algorithm for large problem sizes. Our study also provides insights into these and several other algorithms not found in other computational studies.

4. NETWORK GENERATORS

The performance of an algorithm depends upon the topology of the networks it is tested on. An algorithm can perform very well on some networks and poorly on others. To meet our primary objective, we need to choose networks such that an algorithm's performance on it can give sufficient insight into its general behavior. In the maximum flow literature, no particular type of network has been favored for empirical analysis. Different researchers have used different type of network generators to conduct empirical analysis. We performed preliminary testing on four types of networks : (i) *purely random networks* (where arcs are added by randomly generating tail and head nodes; the source and sink nodes are also randomly selected); (ii) *NETGEN networks* (which are generated by using the well-known network generator NETGEN developed by Klingman et al. [1974]); (iii) *random layered networks* (where nodes are partitioned into layers of nodes and arcs are added from one layer to the next layer using a random process); and (iv) *random grid networks* (where nodes are arranged in a grid and each node is connected to its neighbor in the same and the next grid).

Our preliminary testing revealed that purely random networks and NETGEN networks were rather easy classes of networks for maximum flow algorithms. NETGEN networks were easy even when we generated multi-source and multi-sink maximum flow problems. For our computational testing, we wanted relatively harder problems to better assess the relative merits and demerits of the algorithms. Random

layered and random grid networks appear to meet our criteria and were used in our extensive testing. We give in Figure 4.2(a) an illustration of the random layered network, and in Figure 4.2(b) an illustration of the random grid network, both with width (W) = 3 and length (L) = 4. The topological structure of these networks is revealed in those figures. For a specific value of W and L, the networks have (WL + 2) nodes. A random grid network is from the parameters W and L; however, a random layered network has an additional parameter d, denoting the average outdegree of a node. To generate arcs emanating from a node in layer l in a random layered network, we first determine its outdegree by selecting a random integer, say w, from the uniform distribution in the range [1, 2d -1], and then generate w arcs emanating from node i whose head nodes are randomly selected from nodes in the layer $(l + 1)$. For both the network types, we set the capacities of the source and sink arcs (i.e., arcs incident to the source and sink nodes) to a large number (which essentially amounts to creating w source nodes and w sink nodes). The capacities of other arcs are randomly selected from a uniform distribution in the range [500, 10000] if arcs have their endpoints in different layers, and in the range [200, 1000] if arcs have their endpoints in the same layer.

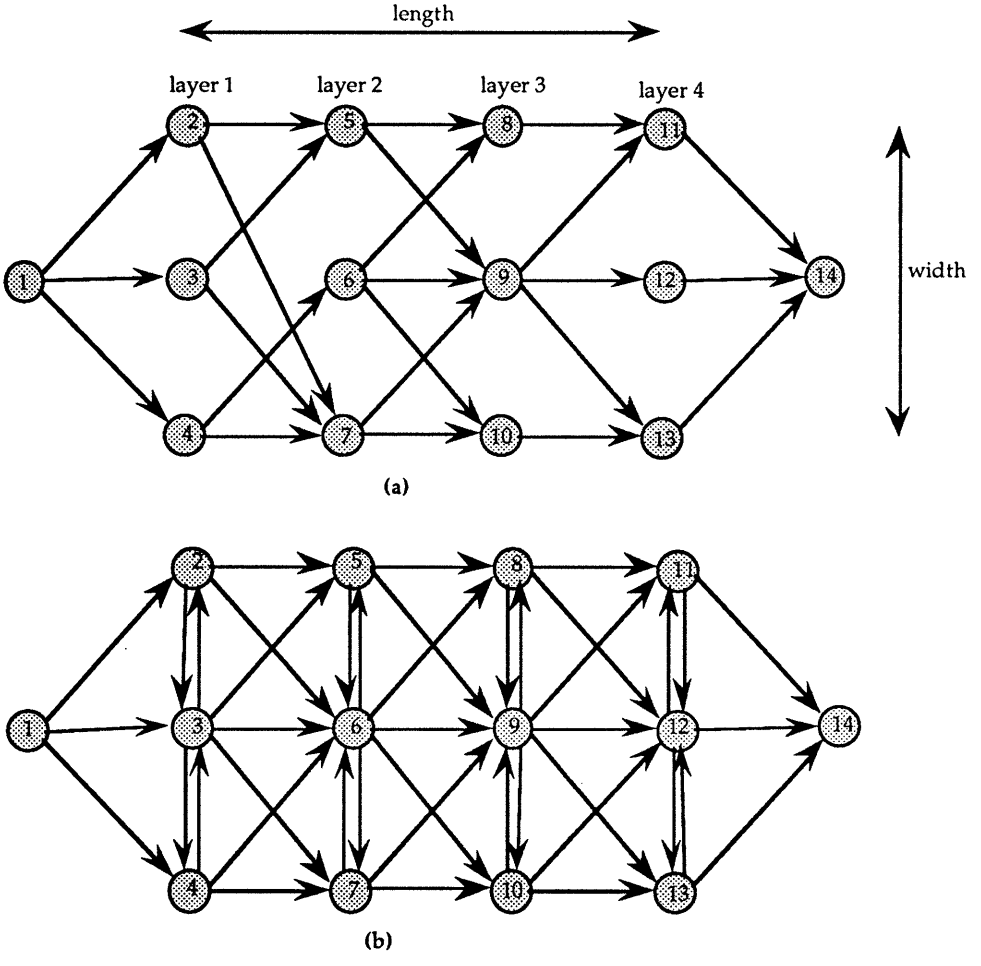


Figure 4.2 Example of a random layered network and a random grid network for width = 3 and length = 4.

In our experiments, we considered networks with different sizes. Two parameters determined the size of the networks: n (number of nodes), and d (average outdegree). For the same number of nodes, we tested different combinations of W (width) and L (length). We observed that various values of the ratio L/W gave similar results unless the network was sufficiently long (L >> W) or sufficiently wide (W >> L). We selected L/W = 2, and observed that the corresponding results were a good representative for a broader range of L/W. The values of n, we considered, varied from 500 to 10,000. Table 4.3 gives the specific values of n and the resulting combinations of W and L. For each n, we considered four densities d = 4, 6, 8 and 10 (for layered networks only). For each combination of n and d, we solved 20 different problems by changing the random number seeds.

Width (W)	16	22	32	39	45	50	55	59	64	67	71
Length (L)	31	45	63	77	89	100	109	119	125	134	141
n (approx.)	500	1,000	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000

Table 4.3 Network Dimensions.

We performed an in-depth empirical analysis of the maximum flow algorithms on random layered and grid networks. But we also wanted to check whether our findings are valid for other classes of networks too. We tested our algorithms on three additional network generators: GL(Genrmf-Long), GW(Genrmf-Wide), and WLM(Washington-Line-Moderate). These networks were part of the DIMACS challenge workshop held in 1991 at Rutgers University. The details of these networks can be found in Badics, Boros and Cepek [1992].

5. REPRESENTATIVE OPERATION COUNTS

Most iterative algorithms for solving optimization problems repetitively perform some basic steps. We can decompose these basic steps into fundamental operations so that the algorithm executes each of these operations in $\Theta(1)$ time. An algorithm typically performs a large number of fundamental operations. We refer to a subset of fundamental operations as a set of *representative operations* if for every possible problem instance, the sum of representative operations provides an upper bound (to within a multiplicative constant) on the sum of all fundamental operations performed by the algorithm. Ahuja and Orlin [1995] present a comprehensive discussion on representative operations and show that these representative operation counts can provide valuable information about an algorithm's behavior. We now present a brief introduction of representative operations counts. We will describe later in Section 8 the use of representative operations counts in the empirical analysis of algorithms.

Let an algorithm perform K fundamental operations denoted by a_1, a_2, ..., a_K, each requiring $\Theta(1)$ time to execute once. For a given instance I of the problem, let $\alpha_k(I)$, for k = 1 to K, denote the number of times

that the algorithm performs the k-th fundamental operation, and CPU(I) denote the CPU time taken by the algorithm. Let S denote a subset of {1, 2, ..., K}. We call S a *representative set of operations* if CPU(I) = $\Theta(\Sigma_{k\in S}\, \alpha_k(I))$, for every instance I, and we call each α_k in this summation a *representative operation count.* In other words, the sum of the representative operation counts can estimate the empirical running time of an algorithm to within a constant factor, i.e., there exist constants c_1 and c_2 such that $c_1\ \Sigma_{k\in S}\alpha_k(I) \le$ CPU(I) $\le$ $c_2\ \Sigma_{k\in S}\alpha_k(I)$. To identify a representative set of operations of an algorithm, we essentially need to identify a set S of operations so that each of these operations takes $\Theta(1)$ time and each execution of every operation not in S can be "charged" to an execution of some operation in S.

6. DESCRIPTION OF AUGMENTING PATH ALGORITHMS

In this section, we describe the following augmenting path algorithms: the shortest augmenting path algorithm, Dinic's algorithm, and the capacity scaling algorithm. In Section 9, we will present the computational testings of these algorithms. In our presentation, we first present a brief description of the algorithm and identify the representative operation counts. We have tried to keep our algorithm description as brief as possible; further details about the algorithms can be found in the cited references, or in Ahuja, Magnanti and Orlin [1993]. We also outline the heuristics we incorporated to speed-up the algorithm performance. In general, we preferred implementing the algorithms in their "purest" forms, and so we incorporated heuristics only when they improved the performance of an algorithm substantially.

Shortest Augmenting Path Algorithm

Augmenting path algorithms incrementally augment flow along paths from the source node to the sink node in the residual network. The shortest augmenting path algorithm always augments flow along a shortest path, i.e., one that contains the fewest number of arcs. A shortest augmenting path in the residual network can be determined by performing a breadth-first search of the network, requiring O(m) time. Edmonds and Karp [1972] showed that the shortest augmenting path algorithm would perform O(nm) augmentations. Consequently, the shortest augmenting path algorithm can be easily implemented in $O(nm^2)$ time. However, a shortest augmenting path can be discovered in an average of O(n) time. One method to achieve the average time of O(n) per path is to maintain "distance labels" and use these labels to identify a shortest path. A set of node label d(·) defined with respect to a given flow x are called *distance labels* if they satisfy the following conditions:

$$d(t) = 0, \qquad (6.1a)$$

$$d(i) \le d(j) + 1 \text{ for every arc } (i, j) \text{ in } G(x). \qquad (6.1b)$$

We call an arc (i, j) in the residual network *admissible* if it satisfies d(i) = d(j) + 1, and *inadmissible* otherwise. We call a directed path P *admissible* if each arc in the path is admissible. The shortest

augmenting path algorithm proceeds by augmenting flows along admissible paths from the source node to the sink node. It obtains an admissible path by successively building it up from scratch. The algorithm maintains a *partial admissible path* (i.e., an admissible path from node s to some node i), and iteratively performs *advance* or *retreat* steps at the last node of the partial admissible path (called the tip). If the tip of the path, node i, has an admissible arc (i, j), then we perform an advance step and add arc (i, j) to the partial admissible path; otherwise we perform a *retreat* step and backtrack by one arc. We repeat these steps until the partial admissible path reaches the sink node, at which time we perform an augmentation. We repeat this process until the flow is maximum.

To begin with, the algorithm performs a backward breadth-first search of the residual network (starting with the sink node) to compute the "exact" distance labels. (The distance label d(i) is called *exact* if d(i) is the fewest number of arcs in the residual network from i to t. Equivalently, d(i) is exact if there is an admissible path from i to t.) The algorithm starts with the partial admissible path $P := \phi$ and tip $i := s$, and repeatedly executes one of the following three steps:

advance(i). If there exists an admissible arc (i, j), then set pred(j) : = i and $P := P \cup \{(i, j)\}$. If j = t, then go to *augment*; else replace i by j and repeat *advance(i)*.

retreat(i). Update $d(i) := \min\{d(j) + 1 : r_{ij} > 0 \text{ and } (i, j) \in A(i)\}$. (This operation is called a *relabel* operation.) If $d(s) \geq n$, then stop. If i = s, then go to advance(i); else delete (pred(i), i) from P, replace i by pred(i) and go to *advance(i)*.

augment. Let $\Delta := \min\{r_{ij}: (i, j) \in P\}$. Augment Δ units of flow along P. Set $P := \phi$, i := s, and go to *advance(i)*.

The shortest augmenting path algorithm uses the following data structure to identify admissible arcs emanating from a node in the *advance* steps. Recall that for each node i, we maintain the arc adjacency list which contains the arcs emanating from node i. We can arrange arcs in these lists arbitrarily, but the order, once decided, remains unchanged throughout the algorithm. We further maintain with each node i an index, called *current-arc*, which is an arc in A(i) and is the next candidate for admissibility testing. Initially, the current-arc of node i is the first arc in A(i). Whenever the algorithm attempts to find an admissible arc emanating from node i, it tests whether the node's current arc is admissible. If not, it designates the next arc in the arc list as the current-arc. The algorithm repeats this process until it finds an admissible arc or reaches the end of the arc list. In the latter case, the algorithm relabels node i and sets its current-arc to the first arc in A(i).

We can show the following results about the shortest augmenting path algorithm: (i) the algorithm relabels any node at most n times; consequently, the total number of relabels is $O(n^2)$; (ii) the algorithm performs at most nm augmentations; and (iii) the running time of the algorithm is $O(n^2m)$.

The shortest augmenting path algorithm, as described, terminates when $d(s) \geq n$. Empirical investigations revealed that this is not a satisfactory termination criterion because the algorithm spends too much time relabeling the nodes after the algorithm has already established a maximum flow. This happens because the algorithm does not know that it has found a maximum flow. We next suggest a technique that is capable of detecting the presence of a minimum cut and a maximum flow much before the label of node s satisfies $d(s) \geq n$. This technique was independently developed by Ahuja and Orlin [1991], and Derigs and Meier [1989].

To implement this technique, we maintain an n-dimensional array called number, whose indices vary from 0 to (n-1). The value number(k) stores the number of nodes whose distance label equals k. Initially, when the algorithm computes exact distance labels using breadth-first search, the positive entries in the array number are consecutive. Subsequently, whenever the algorithm increases the distance label of a node from k_1 to k_2, it subtracts 1 from number(k_1), adds 1 to number(k_2), and checks whether number(k_1) = 0. If number(k_1) = 0, then there is a "gap" in the number array and the algorithm terminates. To see why this termination criteria works, let $S = \{i \in N: d(i) > k_1\}$ and $\bar{S} = \{i \in N: d(i) < k_1\}$. It can be verified using the distance validity conditions (6.1) that all forward arcs in the s-t cut $[S, \bar{S}]$ must be saturated and backward arcs must be empty; consequently, $[S, \bar{S}]$ must be a minimum cut and the current flow maximum. We shall see later that this termination criteria typically reduces the running time of the shortest augmenting path algorithm by a factor between 10 and 30 in our tests.

We now determine the set of representative operations performed by the algorithm. At a fundamental level, the steps performed by the algorithm can be decomposed into scanning the arcs, each requiring $\Theta(1)$ time. We therefore analyse the number of arcs scanned by various steps of the algorithm.

Retreats. A retreat step at node i scans $|A(i)|$ arcs to relabel node i. If node i is relabeled $\alpha(i)$ times, then the algorithm scan a total of $\sum_{i \in N} \alpha(i) |A(i)|$ arcs during relabels. Thus arc scans during relabels, called *arc-relabels*, is the first representative operation. Observe that in the worst-case, each node i is relabeled at most n times, and the arc scans in the relabel operations could be as many as $\sum_{i \in N} n |A(i)| = n \sum_{i \in N} |A(i)| = nm$; however, on the average, the arc scans would be much less.

Augmentations. The fundamental operation in augmentation steps is the arcs scanned to update flows. Thus arc scans during augmentations, called *arc-augmentations*, is the second representative operation. Notice that in the worst-case, arcs-augmentations could be as many as n^2m; however, the actual number would be much less in practice.

Advances. Each advance step traverses (or scans) one arc. Each arc scan in an advance step is one of the two types: (i) a scan which is later cancelled by a retreat operation; and (ii) a scan on which an augmentation is

subsequently performed. In the former case, this arc scan can be charged to the retreat step, and in the later case it can be charged to the augmentation step. Thus, the arc scans during advances can be accounted by the first and second representative operations, and we do not need to keep track of advances explicitly.

Finding admissible arcs. Finally, we consider the arcs scanned while identifying admissible arcs emanating from nodes. Consider any node i. Notice that when we have scanned $|A(i)|$ arcs, we reach the end of the arc list and the node is relabeled, which requires scanning $|A(i)|$ arcs. Thus, arcs scanned while finding admissible arcs can be charged to arc-relabels, which is the first representative operation.

Thus the preceding analysis concludes that one legitimate set of representative operations for the shortest augmenting path algorithm is the following: (i) arc-relabels; and (ii) arc-augmentations.

Dinic's Algorithm

Dinic's algorithm proceeds by constructing shortest path networks, called *layered networks*, and by establishing *blocking flows* in these networks. With respect to a given flow x, we construct the layered network V as follows. We determine the exact distance labels d in G(x). The layered network consists of those arcs (i, j) in G(x) which satisfy d(i) = d(j)+1. In the layered network, nodes are partitioned into layers of nodes $V_0, V_1, V_2, ..., V_l$, where layer k contains the nodes whose distance labels equal k. Furthermore, each arc (i, j) in the layered network satisfies $i \in V_k$ and $j \in V_{k-1}$ for some k. Dinic's algorithm augments flow along those paths P in the layered network for which $i \in V_k$ and $j \in V_{k-1}$ for each arc $(i, j) \in P$. In other words, Dinic's algorithm does not allow traversing the arcs of the layered network in the opposite direction. Each augmentation saturates at least one arc in the layered network, and after at most m augmentations the layered network contains no augmenting path. We call the flow at this stage a *blocking flow.*

Using a simplified version of the shortest augmenting path algorithm described earlier, the blocking flow in a layered network can be constructed in O(nm) time (see Tarjan [1983]). When a blocking flow has been constructed in the network, Dinic's algorithm recomputes the exact distance labels, forms a new layered network, and constructs a blocking flow in the new layered network. The algorithm repeats this process until obtaining a layered network for which the source is not connected to the sink, indicating the presence of a maximum flow. It is possible to show that every time Dinic's algorithm forms a new layered network, the distance label of the source node strictly increases. Consequently, Dinic's algorithm forms at most n layered networks and runs in $O(n^2m)$ time.

We point out that Dinic's algorithm is very similar to the shortest augmenting path algorithm. Indeed the shortest augmenting path algorithm can be viewed as Dinic's algorithm where in place of the layered network, distance labels are used to identify shortest augmenting paths. Ahuja and Orlin [1991] show that both the algorithms are equivalent in the sense that on the same problem they will perform the

same sequence of augmentations. Consequently, the operations performed by Dinic's algorithm are the same as those performed by the shortest augmenting path algorithm except that the arcs scanned during relabels will be replaced by the arc scanned while constructing layered networks. Hence, Dinic's algorithm has the following two representative operations: (i) arcs scanned while constructing layered networks; and (ii) arc-augmentations.

Capacity Scaling Algorithm

We now describe the capacity scaling algorithm for the maximum flow problem. This algorithm was originally suggested by Gabow [1985]. Ahuja and Orlin [1991] subsequently developed a variant of this approach which is better empirically. We therefore tested this variant in our computational study.

The essential idea behind the capacity scaling algorithm is to augment flow along a path with *sufficiently large* residual capacity so that the number of augmentations is *sufficiently small.* The capacity scaling algorithm uses a parameter Δ and with respect to a given flow x, defines the *Δ-residual network* as a subgraph of the residual network where the residual capacity of every arc is at least Δ. We denote the Δ-residual network by $G(x, \Delta)$. The capacity scaling algorithm works as follows:

algorithm *capacity-scaling;*
begin
    $\Delta := 2^{\lfloor \log U \rfloor}$; $x^{2\Delta} := 0$;
    while $\Delta \geq 1$ **do**
    begin
        starting with the flow $x = x^{2\Delta}$, use the shortest augmenting path algorithm to construct augmentations of capacity Δ or greater until obtaining a flow x^{Δ} such that there is no augmenting path of size Δ in $G(x, \Delta)$;
        set $x := x^{\Delta}$;
        reset $\Delta := \Delta/2$;
    end;
end;

We call a phase of the capacity scaling algorithm during which Δ remains constant as the *Δ-scaling phase.* In the Δ-scaling phase, each augmentation carries at least Δ units of flow. The algorithm starts with $\Delta = 2^{\lfloor \log U \rfloor}$ and halves its value in every scaling phase until $\Delta = 1$. Hence the algorithm performs $1 + \lfloor \log U \rfloor = O(\log U)$ scaling phases. Further, in the last scaling phase, $\Delta = 1$ and hence $G(x, \Delta) = G(x)$. This establishes that the algorithm terminates with a maximum flow.

The efficiency of the capacity scaling algorithm depends upon the fact that it performs at most 2m augmentations per scaling phase (see Ahuja and Orlin [1991]). Recall our earlier discussion that the shortest augmenting path algorithm takes $O(n^2 m)$ time to perform augmentations (because it performs O(m) augmentations) and O(nm) time to perform the remaining operations. When we employ the shortest

augmenting path algorithm for reoptimization in a scaling phase, it performs only O(m) augmentations and, consequently, runs in O(nm) time. As there are O(log U) scaling phases, the overall running time of the capacity scaling algorithm is O(nm log U).

The capacity scaling algorithm has the following three representative operations:

Relabels. The first representative operation is arcs scanned while relabeling the nodes. In each scaling phase, the algorithm scans O(nm) arcs. Overall, the arc scanning could be as much as O(nm log U), but empirically it is much less.

Augmentations. The second representative operation is the arcs scanned during flow augmentations. As observed earlier, the worst-case bound on the arcs scanned during flow augmentations is O(nm log U).

Constructing Δ-residual networks. The algorithm constructs Δ-residual networks $1 + \lfloor \log U \rfloor$ times and each such construction requires scanning Θ(m) arcs. Hence, constructing Δ-residual network requires scanning a total of Θ(m log U) arcs, which is the third representative operation.

It may be noted that compared to the shortest augmenting path algorithm, the capacity scaling algorithm reduces the arc-augmentations from $O(n^2m)$ to O(nm log U). Though this improves the overall worst-case performance of the algorithm, it actually worsens the empirical performance, as discussed in Section 9.

7. DESCRIPTION OF PREFLOW-PUSH ALGORITHMS

In this section, we describe the following preflow-push algorithms: FIFO, highest-label, lowest-label, excess-scaling, stack-scaling, wave-scaling, and Karzanov's algorithm. Section 10 presents the results of the computational testing of these algorithms.

The preflow-push algorithms maintain a preflow, defined in Section 2, and proceed by examining active nodes, i.e., nodes with positive excess. The basic repetitive step in the algorithm is to select an active node and to attempt to send its excess closer to the sink. As sending flow on admissible arcs pushes the flow closer to the sink, the algorithm always pushes flow on admissible arcs. If the active node being examined has no admissible arc, then we increase its distance label to create at least one admissible arc. The algorithm terminates when there are no active nodes. The algorithmic description of the preflow-push algorithm is as follows:

algorithm *preflow-push;*
begin
set x: = 0 and compute exact distance labels in G(x);
send $x_{sj} := u_{sj}$ flow on each arc (s, j) $\in$ A and set d(s) : = n;
while the network contains an active node **do**
begin
select an active node i;
push/relabel(i)
end;
end;

procedure *push/relabel(i);*
begin
if the network contains an admissible arc (i, j) **then**
push $\delta := \min \{e(i), r_{ij}\}$ units of flow from node i to node j
else replace d(i) by min{d(j) + 1: (i, j) $\in$ A(i) and $r_{ij} > 0$};
end;

We say that a push of δ units on an arc (i, j) is *saturating* if $\delta = r_{ij}$, and *nonsaturating* if $\delta < r_{ij}$. A nonsaturating push reduces the excess at node i to zero. We refer to the process of increasing the distance label of a node as a *relabel* operation. Goldberg and Tarjan [1986] established the following results for the preflow-push algorithm.

(i) Each node is relabeled at most 2n times and the total relabel time is O(nm).

(ii) The algorithm performs O(nm) saturating pushes.

(c) The algorithm performs $O(n^2m)$ nonsaturating pushes.

In each iteration, the preflow-push algorithm either performs a push, or relabels a node. The preflow-push algorithm identifies admissible arcs using the current-arc data structure also used in the shortest augmenting path algorithm. We observed in Section 6 that the effort spent in identifying admissible arcs can be charged to the arc-relabels. Therefore, the algorithm has the following two representative operations: (i) arc-relabels, and (ii) pushes. The first operation has a worst-case time bound of O(nm) and the second operation has a worst-case time bound of $O(n^2m)$.

It may be noted that the representative operations of the generic preflow-push algorithm have a close resemblance with those of the shortest augmenting path algorithm and, hence, with those of Dinic's and capacity scaling algorithms. They both have arc-relabels as their first representative operation. Whereas the shortest augmenting path algorithm has arc-augmentation as its second representative operation, the preflow-push algorithm has pushes on arcs as its second representative operation. We note that sending flow on an augmenting path P may be viewed as a sequence of pushes along the arcs of P.

We next describe some implementation details of the preflow-push algorithms. All preflow-push algorithms tested by us incorporate these implementation details. In an iteration, the preflow-push algorithm selects a node, say i, and performs a saturating push, or a nonsaturating push, or relabels a node. If the algorithm performs a saturating push, then node i may still be active, but in the next iteration the algorithm may select another active node for push/relabel step. However, it is easy to incorporate the rule that whenever the algorithm selects an active node, it keeps pushing flow from that node until either its excess becomes zero or it is relabeled. Consequently, there may be several saturating pushes followed by either a nonsaturating push or a relabel operation. We associate this sequence of operation with a *node examination.* We shall henceforth assume that the preflow-push algorithms follow this rule.

The generic preflow-push algorithm terminates when all the excess is pushed to the sink or returns back to the source node. This termination criteria is not attractive in practice because this results in too many relabels and too many pushes, a major portion of which is done after the algorithm has already established a maximum flow. To speed up the algorithm, we need a method to identify the active nodes that become disconnected from the sink (i.e., have no augmenting paths to the sink) and avoid examining them. One method that has been implemented by several researchers is to occasionally perform a breadth-first search to recompute exact distance labels. This method also identifies nodes that become disconnected from the sink node. In our preliminary testing, we tried this method and several other methods. We found the following variant on the "number array" method to be the most efficient in practice.

Let the set *DLIST(k)* consist of all nodes with distance label equal to k. Let the index *first(k)* point to the first node in DLIST(k) if DLIST(k) is nonempty, and first(k) = 0 otherwise. We maintain the set DLIST(k) for each $1 \leq k \leq n$ in the form of a doubly linked list. We initialize these lists when initial distance labels are computed by the breadth-first search. Subsequently, we update these lists whenever a distance update takes place. Whenever the algorithm updates the distance label of a node from k_1 to k_2, we update DLIST(k_1) and DLIST(k_2) and check whether first(k_1) = 0. If so, then all nodes in the sets DLIST(k_1+1), DLIST(k_1+2), ... have become disconnected from the sink. We scan the sets DLIST(k_1+1), DLIST(k_1+2), ..., and *mark* all the nodes in these sets so that they are never examined again. We then continue with the algorithm until there are no active nodes that are unmarked.

We also found another heuristic speedup to be effective in practice. At every iteration, we keep track of the number r of marked nodes. Wherever any node i is found to have $d(i) \geq (n-r-1)$, we mark it too and increment r by one. It can be readily shown that such a node is disconnected from the sink node.

If we implement preflow-push algorithms with these speedups, then the algorithm terminates with a maximum preflow. It may not be a flow because some excess may reside at marked nodes. At this time, we initiate the second phase of the algorithm, in which we convert the maximum preflow into a

maximum flow by returning the excesses of all nodes back to the source. We perform a (forward) breadth-first search from the source to compute the initial distance labels d'(·), where the distance label d'(i) represents a lower bound on the length of the shortest path from node i to node s in the residual network. We then perform preflow-push operations on active nodes until there are no more active nodes. It can be shown that regardless of the order in which active nodes are examined, the second phase terminates in O(nm) time. We experimented with several rules for examining active nodes and found that the rule that always examines an active node with the highest distance label leads to minimum number of pushes in practice. We incorporated this rule into our algorithms.

An attractive feature of the generic preflow-push algorithm is its flexibility. By specifying different rules for selecting active nodes for the push/relabel operations, we can derive many different algorithms, each with different worst-case and empirical behaviors. We consider the following three implementations:

Highest-Label Preflow-Push Algorithm.

The highest-label preflow-push algorithm always pushes flow from an active node with the highest distance label. Let $h^* = \max \{d(i): i \text{ is active}\}$. The algorithm first examines nodes with distance label h^* and pushes flow to nodes with distance label h^*-1, and these nodes, in turn, push flow to nodes with distance labels equal to h^*-2, and so on, until either the algorithm relabels a node or it has exhausted all the active nodes. When it has relabeled a node, the algorithm repeats the same process. Goldberg and Tarjan [1986] obtained a bound of $O(n^3)$ on the number of nonsaturating pushes performed by the algorithm. Later, Cheriyan and Maheshwari [1989] showed that this algorithm actually performs $O(n^2 m^{1/2})$ nonsaturating pushes and this bound is tight.

We next discuss how the algorithm selects an active node with the highest distance label without too much effort. We use the following data structure to accomplish this. We maintain the sets *SLIST(k)* = {i: i is active and d(i) = k} for each k = 1, 2, ..., 2n-1, in the form of singly linked stacks. The index *next(k)*, for each $0 \le k \le 2n-1$, points to the first node in *SLIST(k)* if *SLIST(k)* is nonempty, and is 0 otherwise. We define a variable *level* representing an upper bound on the highest value of k for which *SLIST(k)* is nonempty. In order to determine a node with the highest distance label, we examine the lists *SLIST(level)*, *SLIST(level-1)*, ..., until we find a nonempty list, say *SLIST(p)*. We select any node in *SLIST(p)* for examination, and set level = p. Also, whenever the distance label of a node being examined increases, we reset level equal to the new distance label of the node. It can be shown that updating SLIST(k) and updating level is on average O(1) steps per push and O(1) steps per relabel. This result and the previous discussion implies that the highest-label preflow-push algorithm can be implemented in $O(n^2\sqrt{m})$ time.

FIFO Preflow-Push Algorithm

The FIFO preflow-push algorithm examines active nodes in the first-in-first-out order. The algorithm maintains the set of active nodes in a queue called QUEUE. It selects a node i from the front of QUEUE for examination. The algorithm examines node i until it becomes inactive or it is relabeled. In the latter case, node i is added to the rear of QUEUE. The algorithm terminates when QUEUE becomes empty. Goldberg and Tarjan [1986] showed that the FIFO implementation performs $O(n^3)$ nonsaturating pushes and can be implemented in $O(n^3)$ time.

Lowest-Label Preflow-Push Algorithm.

The lowest-label preflow-push algorithm always pushes flow from an active node with the smallest distance label. We implement this algorithm in a manner similar to the highest-label preflow-push algorithm. This algorithm performs $O(n^2m)$ nonsaturating pushes and runs in $O(n^2m)$ time.

EXCESS-SCALING ALGORITHMS

Excess-scaling algorithms are special implementations of the generic preflow-push algorithms and incorporate scaling technique which dramatically improves the number of nonsaturating pushes in the worst-case. The essential idea in the (original) excess-scaling algorithm is to assure that each nonsaturating push carries "sufficiently large" flow so that the number of nonsaturating pushes is "sufficiently small". The algorithm defines the term "sufficiently large" and "sufficiently small" iteratively. Let $e_{max} = \max\{e(i): i \text{ active}\}$ and Δ be an upper bound on e_{max}. We refer to a node i with $e(i) \geq \Delta/2 \geq e_{max}/2$ as a node with *large excess*, and a node with *small excess* otherwise. Initially $\Delta = 2^{\lfloor \log U \rfloor}$, i.e., the largest power of 2 less than or equal to U.

The (original) excess-scaling algorithm performs a number of scaling phases with different values of the scale factor Δ. In the Δ-scaling phase, the algorithm selects a node i with large excess, and among such nodes selects a node with the smallest distance label, and performs push/relabel(i) with the slight modification that during a push on arc (i, j), the algorithm pushes $\min\{e(i), r_{ij}, \Delta\text{-}e(j)\}$ units of flow. (It can be shown that the above rules ensure that each nonsaturating push carries at least $\Delta/2$ units of flow and no excess exceeds Δ.) When there is no node with large excess, then the algorithm reduces Δ by a factor 2, and repeats the above process until $\Delta = 1$, when the algorithm terminates. To implement this algorithm, we maintain the singly linked stacks SLIST(k) for each k = 1, 2, ..., 2n-1, where SLIST(k) stores the set of large excess nodes with distance label equal to k. We determine a large excess node with the smallest distance label by maintaining a variable *level* and using a scheme similar to that for the highest-label preflow-push algorithm. Ahuja and Orlin [1989] have shown that the excess-scaling algorithm performs $O(n^2 \log U)$ nonsaturating pushes and can be implemented in $O(nm + n^2 \log U)$ time.

Similar to other preflow-push algorithms, the excess-scaling algorithm has (i) arc-relabels; and (ii) pushes, as its two representative operations. The excess-scaling algorithm also constructs the lists SLIST(k) at the beginning of each scaling phase, which takes $\Theta(n)$ time, and this time can not be accounted in the two representative operations. Thus constructing these list, which takes a total of $\Theta(n \log U)$ time, is the third representative operation in the excess-scaling algorithm.

We also included in our computational testing two variants of the excess-scaling algorithm with improved worst-case complexities, which were developed by Ahuja, Orlin and Tarjan [1989]. These are (i) the stack-scaling algorithm, and (ii) the wave-scaling algorithm.

Stack-Scaling Algorithm

The stack-scaling algorithm scales excesses by a factor of $k \geq 2$ (i.e., reduces the scale factor by a factor of k from one scaling phase to another), and always pushes flow from a large excess node with the highest distance label. The complexity argument of the excess-scaling algorithm and its variant rests on the facts that a nonsaturating push must carry at least Δ/k units of flow and no excess should exceed Δ. These two conditions are easy to satisfy when the push/relabel operation is performed at a large excess node with the smallest distance label (as in the excess-scaling algorithm), but difficult to satisfy when the push/relabel operation is performed at a large excess node with the largest distance label (as in the stack-scaling algorithm). To overcome this difficulty, the stack-scaling algorithm performs a sequence of push and relabels using a stack S. Suppose we want to examine a large excess node i until either node i becomes a small excess node or node i is relabeled. Then we set S = {i} and repeat the following steps until S is empty.

stack-push. Let v be the top node on S. Identify an admissible arc out of v. If there is no admissible arc, then relabel node v and pop (or, delete) v from S. Otherwise, let (v, w) be an admissible arc. There are two cases.

Case 1. *$e(w) > \Delta/2$ and $w \neq t$.* Push w onto S.

Case 2. *$e(w) \leq \Delta/2$ or $w = t$.* Push $\min\{e(v), r_{ij}, \Delta\text{-}e(w)\}$ units of flow on arc (v, w). If $e(v) \leq \Delta/2$, then pop node v from S.

It can be shown that if we choose $k = \lceil \log U/\log \log U \rceil$, then the stack-scaling algorithm performs $O(n^2 \log U/\log\log U)$ nonsaturating pushes and runs in $O(nm + n^2 \log U/\log \log U)$ time. The representative operations of this algorithm are the same as those for the excess-scaling algorithm.

Wave-Scaling algorithm

The wave-scaling algorithm scales excesses by a factor of 2 and uses a parameter L whose value is chosen appropriately. This algorithm differs from the excess-scaling algorithm as follows. At the

beginning of every scaling phase, the algorithm checks whether $\Sigma_{i \in N} e(i) > n\Delta/L$ (i.e., when the total excess residing at the nodes is *sufficiently large*). If yes, then the algorithm performs passes on active nodes. In each pass, the algorithm examines all active nodes in nondecreasing order of their distance labels and performs pushes at each such node until either its excess reduces to zero or the node is relabeled. We perform pushes at active nodes using the stack-push method described earlier. We terminate these passes when we find that $\Sigma_{i \in N} e(i) \leq n\Delta/L$. At this point, we apply the original excess-scaling algorithm, i.e., we push flow from a large excess node with the smallest distance label. If we choose $L = \sqrt{\log U}$, then the algorithm can be shown to perform $O(n^2\sqrt{\log U})$ nonsaturating pushes and to run in $O(nm + n^2\sqrt{\log U})$ time.

KARZANOV'S ALGORITHM

Karzanov's algorithm is also a preflow-push algorithm, but pushes flow from the source to the sink using layered networks instead of distance labels. Karzanov [1974] describes a preflow-based algorithm to construct a blocking flow in a layered network in $O(n^2)$ time. The algorithm repeatedly performs two operations: *push* and *balance*. The push operation pushes the flow from an active node from one layer to nodes in the next layer (closer to the sink) in the layered network and the balance operation returns the flow that can't be sent to the next layer to the nodes in the previous layer it came from. Karzanov's algorithm repeatedly performs forward and reverse passes on active nodes. In a forward pass, the algorithm examines active nodes in the decreasing order of the layers they belong to and performs push operations. In a backward pass, the algorithm examines active nodes in the increasing order of the layer they belong to and performs balance operations. The algorithm terminates when there are no active nodes. Karzanov shows that this algorithm constructs a blocking flow in a layered network in $O(n^2)$ time; hence the overall running time of the algorithm is $O(n^3)$.

The representative operations in Karzanov's algorithm are (i) the arc scans required to construct layered networks (which are generally m times the number of layered networks), and (ii) the push operations. The balance operations can be charged to the first representative operation. In the worst-case, the first representative operation takes $O(nm)$ time and the second representative operation takes $O(n^3)$ time.

A Remark on the Similar Representative Operations for Maximum Flow Algorithms

The preceding description of the maximum flow algorithms and their analysis using representative operations yields the interesting conclusion that for each of the non-scaling maximum flow algorithms there is a set of two representative operations: (i) arc-relabels; and (ii) either of arc-augmentations and arc-pushes. Whereas the augmenting path algorithms perform the arc-augmentation, the preflow-push algorithms perform arc-pushes. The scaling based methods need to include one more representative operation corresponding to the operations performed at the beginning of a scaling phase. The similarity and

commonality of the representative operations reflects underlying common structure of these various maximum flow algorithms.

8. OVERVIEW OF COMPUTATIONAL TESTING

We shall now present details of our computational testing. We partition our presentation into two parts. We first present results for the augmenting path algorithms and then for the preflow-push algorithms. Among the augmenting path algorithms, we present a detailed study of the shortest augmenting path algorithm because it is the fastest augmenting path algorithm. Likewise, among the preflow-push algorithms, we present a detailed study of the highest-label preflow-push algorithm, which is the fastest among the algorithms tested by us.

Table 8.1gives the storage requirements of the algorithms tested by us. All requirements are linear in n and m, and the largest requirment is within a facotr of 2 of the smallest requirement, assuming that $m \geq n$.

Algorithm	Storage Requirement
Shortest augmenting path algorithm	7n + 6m
Capacity scaling algorithm	7n + 6m
Dinic's algorithm	5n + 6m
Karzanov's algorithm	6n + 8m
Highest-label preflow-push algorithm	10n + 6m
FIFO preflow-push algorithm	8n + 6m
Lowest-label preflow-push algorithm	10n + 6m
Excess-scaling algorithm	10n + 6m
Stack-scaling algorithm	13n + 6m
Wave-scaling algorithm	13n + 6m

Table 8.1 Storage requirements of various maximum flow algorithms.

All of our algorithms were coded in Fortran and efforts were made to run all the programs under similar conditions of load on the computer resources. We performed the computational tests in two phases. In the first phase, we tested our algorithms on the random layered and random grid networks on the Convex mini super computer under the Convex OS 10.0.5 using the Convex Fortran Compiler V 7.0.1 in a time-sharing environment. Each algorithm was tested on these two network generators and for different problem sizes. For each problem size, we solved 20 different problems by changing the seed to the random number generator, and compute the averages of these 20 sets of data. We analyse algorithms using these averages. The CPU times taken by the programs were noted using a standard available time function having a resolution of 1 microsecond. The times reported do not include the input or output times; however, they do include the time taken to initialize the variables. Most of our conclusions are based on these tests. In the second phase, we tested the algorithms on DIMACS benchmark instances on DECSYSTEM-5000, which validated our findings

for the layered and grid networks. In Sections 9 and 10, we present our results for the first phase of testing, and in Section 11 the results of the second phase of testing.

For each algorithm we tested, we considered the following questions: (i) What are the asymptotic bottleneck operations in the algorithm? (ii) What is the asymptotic growth in the running time as the problem size grows larger? (iii) What proportion of time is spent on the bottleneck operations as the problem size grows? (iv) How does each algorithm compare to the best alternative algorithm? (v) How sensitive are the results to the network generator?

We used the representative operation counts (discussed in Section 5) to answer the above questions and provide a mixture of statistics and visual aids. The representative operation counts allow us to perform the following tasks:

(a) Identifying Asymptotic Bottleneck Operations. A representative operation is an asymptotic bottleneck operation if its share in the computational time becomes larger and larger as the problem size increases. Suppose an algorithm has two representative operations A and B. Let $\alpha_S(I) = \alpha_A(I) + \alpha_B(I)$. For the identification of the asymptotic bottleneck operations, we plotted $\alpha_A(I)/\alpha_S(I)$ and $\alpha_B(I)/\alpha_S(I)$ for increasingly large problem instances. In most cases, we identified an operation that accounts for an increasing larger share of the running time as problem sizes grew larger, and we extrapolated that this operation is the asymptotic bottleneck operation.

(b) Comparing Two Algorithms. Let $\alpha_{S1}(k)$ and $\alpha_{S2}(k)$ be the total number of representative operations performed by two algorithms AL_1 and AL_2, respectively, on instances of size k. We say that AL_1 is superior to the algorithm AL_2 if $\lim_{k\to\infty}\{\alpha_{S1}(k)/\alpha_{S2}(k)\} \to 0$. We estimate this limit by extrapolating from trends in the plots of $\alpha_{S1}(k)/\alpha_{S2}(k)$.

(c) Virtual Running Time. Suppose that an algorithm has two representative operations A and B. Then we estimate the running time of the algorithm on instance I by fitting a linear regression to CPU(I) of the form c_A $\alpha_A(I)+ c_B$ $\alpha_B(I)$, To obtain an idea of the goodness of this fit, we plot the ratio V(I)/CPU(I) for all the data points. (This is an alternative to plotting the residuals.) For all the maximum flow algorithms, these virtual running time estimates were excellent approximations, typically within 5% of the true running time.

The virtual time analysis also allows us to estimate the proportion of the time spent in various representative operations. For example, if the virtual running time of a preflow-push algorithm is estimated to be c_1(number of pushes) + c_2(number of arc-relabels), then one can estimate the time spent in pushing as c_1(number of pushes)/(virtual CPU time).

(d) Estimating the Growth Rate of Bottleneck Operations. We estimated the growth rate of each bottleneck (representative) operation in terms of the input size parameters. We prefer this approach to estimating only CPU time directly because the CPU time is the weighted sum of several operations and hence usually has a more complex growth rate. We estimate the growth rateas a polynomial $\alpha\ n^{\beta}\ d^{\gamma}$ for a network flow problem, where d = m/n. After taking logs of the computation counts, the growth rate is

estimated to be linear in log n and log d. We determine the coefficients α, β and γ using linear regression analysis. We plotted the predicted operation counts (based on the regression) divided by the actual operation counts. This curve is an alternative to plotting the residuals.

We observed that the computational results are sensitive to the network generator. In principle, one can run tests on a wide range of generators to investigate the robustness of the algorithms, but this may be at the expense of unduly increasing the scope of the study. To investigate the robustness of our conclusions, we performed some additional tests of our algorithms on the DIMACS benchmark instances. Most of our conclusions based on tests on our initial network generators extend to those classes of networks too.

9. COMPUTATIONAL RESULTS OF AUGMENTING PATH ALGORITHMS

In this section, we present computational results of the augmenting path algorithms discussed in Section 6. We first present results for the shortest augmenting path algorithm.

Shortest Augmenting Path Algorithm

In Figure 9.1, we show the CPU times taken by the shortest augmenting path algorithm for the two network generators and for different problem sizes. The figure contains five plots. Of these, four plots are for the problems generated by the random layered networks (or, simply, the layered networks) for densities d = 4, d = 6, d = 8 and d = 10, and the fifth plot is for the problems generated by the random grid networks (or, simply, the grid networks). The plot with squares on it is for the grid networks. For each problem size, we solved 20 different problems and used the averaged data for drawing these figures.

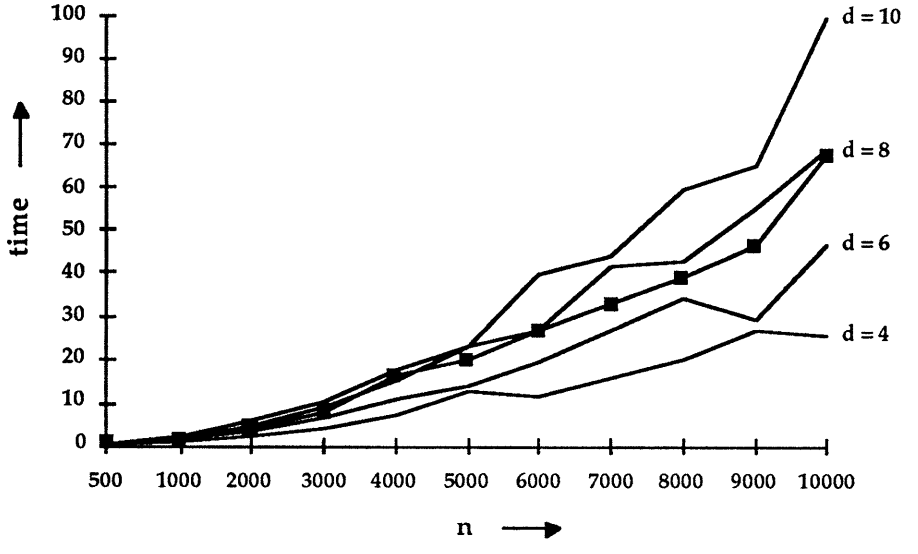


Figure 9.1. CPU Time for the shortest augmenting path algorithm.

As we observed earlier, the representative operations in the shortest augmenting path algorithm are (i) α_r, arc scans for relabels or, simply arc-relabels, and (ii) α_a, arc scans for augmentations, or, simply, arc-augmentations. To identify which one of these two operations is the asymptotic bottleneck operation, we plot the ratio $\alpha_r/(\alpha_r + \alpha_a)$ in Figure 9.2 as a function of n. Although for all of our tests on layered networks $\alpha_r \leq \alpha_a$, it appears that the plot of the ratios $\alpha_r/(\alpha_r + \alpha_a)$ have a slightly upward trend. The plots suggest that arc-relabels increase at a rate slightly higher than the arc-augmentations. In other words, empirically, the relabel time grows at a faster pace than the augmentation time. This observation contrasts with what is indicated by the worst-case analysis. The worst-case bound for the augmentation time (which is $O(n^2m)$) grows much faster than the worst-case bound for the relabel time (which is O(nm)). We also observe from these figures that as the network density increases for layered networks, the share of the relabel time in the overall time slightly decreases. We conjecture that this behavior is exhibited by the shortest augmenting path algorithm because increasing the network density causes the number of augmentations to increase at a pace faster than the number of relabels, and thus the augmentations will constitute a larger proportion of the representative operation counts.

Figure 9.2 throws plots the relative proportion of the representative operation counts within the total counts, but does not directly indicate what proportion of the CPU time is spent on these two operations. To do so, we compute the virtual running time, which is an estimate of the CPU time as a linear function of the representative operation counts, obtained using regression. We obtain the following expresion of the virtual running time (with a R^2 value equal to 0.9998).

$$V(I) = 6.7\text{x}10^{-6}\,\alpha_r(I) + 7.6\text{x}10^{-6}\,\alpha_a(I)$$

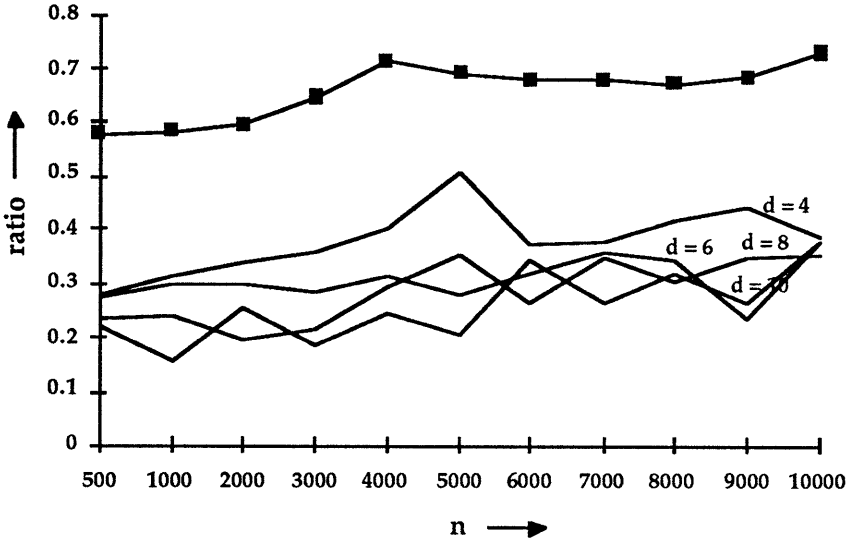


Figure 9.2. Growth rate of arc-relabels/(arc-relabels + arc-augmentations) for the shortest augmenting path algorithm.

To visualize how accurately the virtual time estimates the CPU time, we plot the ratio of V(I)/CPU(I) in Figure 9.3. We find that for all the problem instances, this ratio is between 0.95 and 1.05. To determine the time spent on the two representative operations, we plot the ratio $(6.7 \times 10^{-6}\ \alpha_r(I))/V(I)$ in Figure 9.4. We find that for the layered networksthat we tested, the relabel time is estimated to be less than the augmentation time, but for grid networks the relabel time is estimated to be greater than the augmentation time.

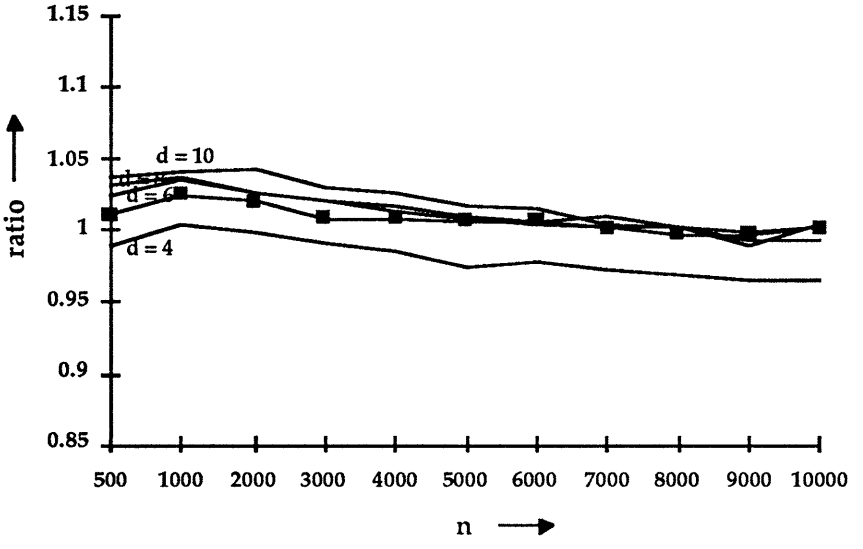


Figure 9.3. Ratio of the virtual time to the CPU time for the shortest augmenting path algorithm.

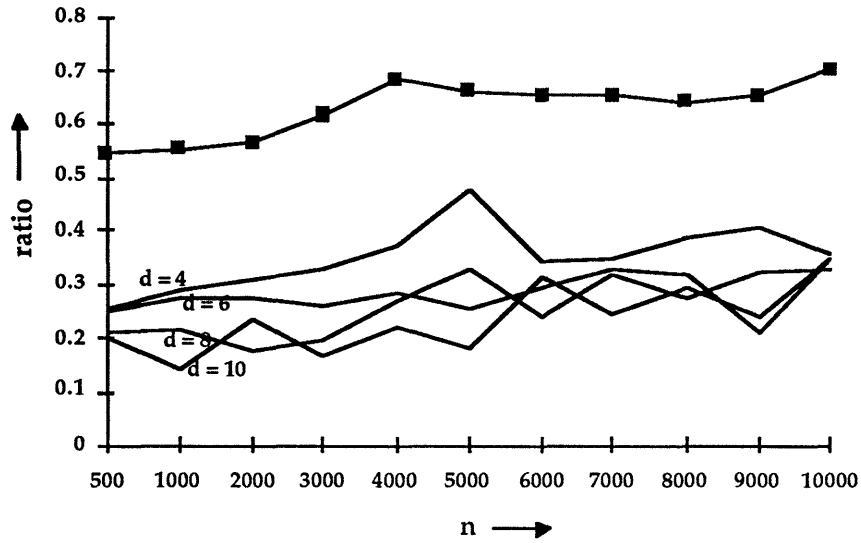


Figure 9.4. Share of relabeling time in virtual time for the shortest augmenting path algorithm.

Recall that the shortest augmenting path algorithm uses a number array to speed up its termination. To judge the usefulness of this termination criteria, we also implemented the algorithm without it. Table 9.1 shows the average number of relabels and CPU times for these two versions of the algorithm. It is evident from the table that the use of the number array characteristically reduces the running time of the shortest augmenting path algorithm, and the reductionin running time increases with problem size.

	number of relabels		CPU time (in seconds)	
n	with number array	without number array	with number array	without number array
500	1,380	47,282	0.41	3.67
1,000	4,343	294,900	1.19	22.69
2,000	12,816	1,044,614	3.88	81.17
3,000	21,102	2,054,433	6.46	162.18

Table 9.1 Use of number array in speeding up the shortest augmenting path algorithm.

We also investigated how quickly flow reaches the sink as the algorithm progresses. It may be valuable to know how quickly the flow reaches the sink for two reasons: first of all, in some applications it may be satisfactory to find a flow that is nearly maximum, and so one can terminate the algorithm early. Second, the information may be useful in the design of hybrid algorithms, which can sometimes combine the

best features of two different algorithms. In Figure 9.5, we plotted the percentage of the maximum flow that has reached the sink prior to "time p" where here time p refers to the time at which a proportion p of relabels has been carried out. (The particular network is a layered network with n = 10,000 and d = 6.) For this particular problem instance, the flow reaching the source increases almost linearly with the number of augmentations. (The amount of flow in an augmentation is roughly constant over time.) But the rate of change is quite nonlinear with the number of relabels. We observe that 90% of the total flow is sent within 10% of the total node relabels, and the remaining 10% of the total flow takes up 90% of the node relabels. In other words, the time between successive augmentations increases over time, and the final augmentations may be separated by far more relabels. Since the later part of the algorithm was taking a large proportion of the running time, we tried a variety of techniques to speed up the later part of the algorithm; however, none of these techniques yielded any significant improvement.

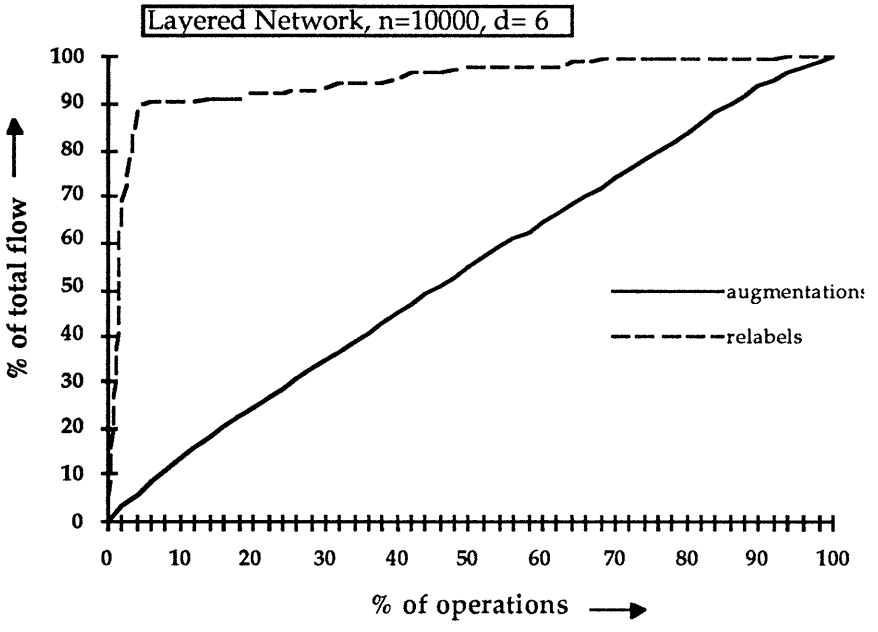


Figure 9.5. Flow sent to the sink as a percentage of total relabels and augmentations for the shortest augmenting path algorithm.

We use regression analysis to estimate the growth rate of the representative operation counts of the shortest augmenting path algorithm as a function of the problem size. For the layered and grid networks, the estimates for the arc-relabels and arc-augmentations are as follows :

	Layered Networks	Grid Networks
arc-relabels	$0.65 \times n^{1.75} \times d^{0.71}$	$0.07 \times n^{1.74}$
arc-augmentations	$0.21 \times n^{1.56} \times d^{1.34}$	$1.77 \times n^{1.54}$

We also wanted to determine how the number of relabels and augmentations vary as a function of the problem size parameters. For the layered networks, we found the estimates of relabels and augmentations to be given by

	Layered Networks	Grid Networks
relabels	$0.04 \times n^{1.72} \times d^{-0.25}$	$0.618 \times n^{1.75}$
augmentations	$0.14 \times n^{1.06} \times d^{1.36}$	$1.2 \times n^{1.04}$

To visualize the quality of these estimates, we plot the ratio of predicted to actual arc-relabels and arc-augmentations in Figure 9.6. Generally, these estimation errors are within 20%. We also conjectured that for our network types, the shortest augmenting path algorithm will perform no more than $n^{1.5}$ augmentations and will take a time bounded by $O(n^2)$. The plots shown in Figure 9.7 appear to justify our both the conjectures.

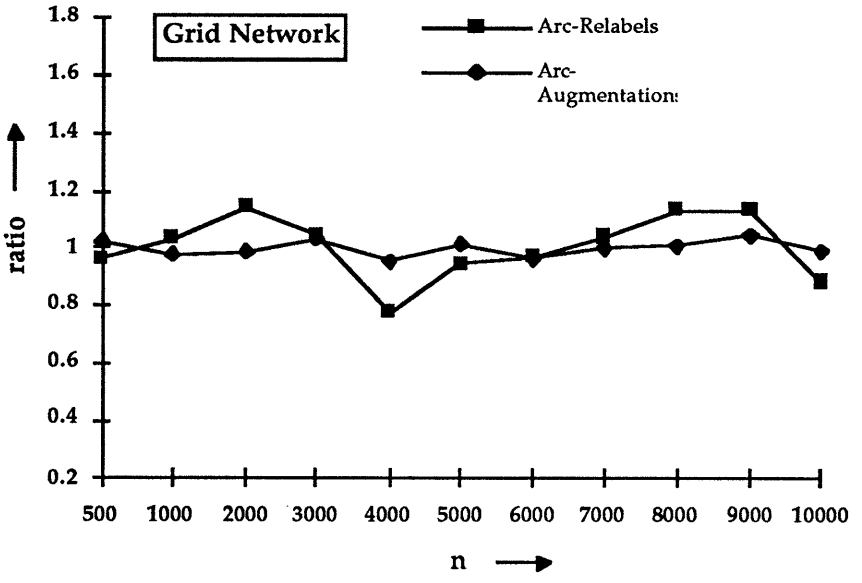


Figure 9.6. Ratio of predicted to actual arc-relabels and arc-augmentations for the shortest augmenting path algorithm.

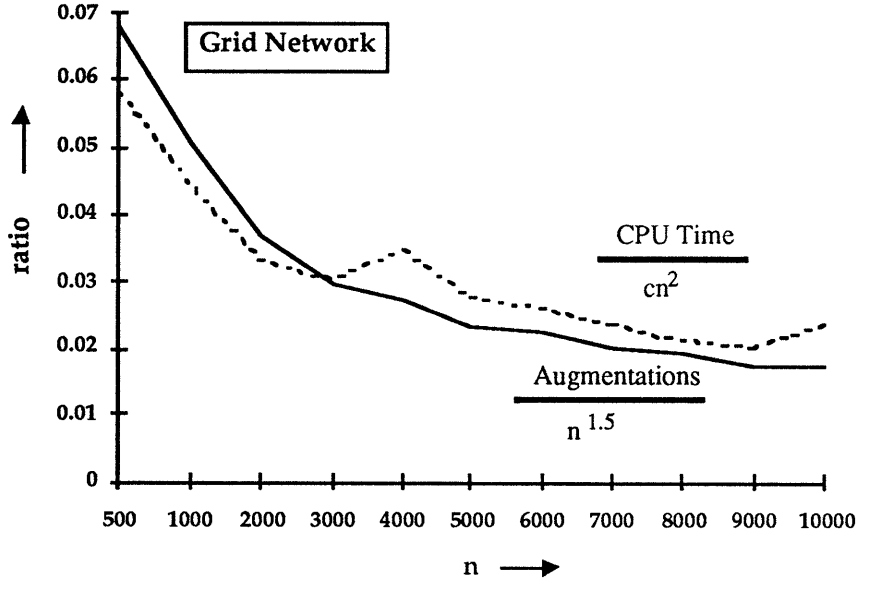


Figure 9.7. Bounding the number of augmentations and CPU time as a function of n for the shortest augmenting path algorithm.

Capacity Scaling Algorithm

The capacity scaling algorithm for the maximum flow problem improves the worst-case complexity of the shortest augmenting path algorithm by incorporating capacity scaling. In the capacity scaling algorithm, we used a scale factor of 2. Our computational tests revealed that though the capacity scaling algorithm improves the worst-case running time, it worsens the empirical running time. We shall now present a few graphs that shed insight on the comparative behavior of the shortest augmenting path algorithms and the capacity scaling algorithm. For the sake of brevity, we present results for the grid networks only, since the behavior for the layered networks has been similar.

	Shortest augmenting path algorithm	Capacity scaling algorithm
Number of augmentations	O(nm)	O(m log U)
Augmentation time	$O(n^2m)$	O(nm log U)
Number of relabels	$O(n^2)$	$O(n^2 \log U)$
Relabel time	O(nm)	O(nm log U)
Total time	$O(n^2m)$	O(nm log U)

Table 9.2 Comparative behavior of the shortest augmenting path and the capacity scaling algorithm in the worst-case.

Table 9.2 presents the worst-case comparative behavior of these two algorithms. Observe that the capacity scaling algorithm reduces the augmentation time but increases the relabel time; but overall the time decreases. We present in Figure 9.8 the ratios of the number of arc-relabels and arc-augmentations performed by the capacity scaling algorithm versus the shortest augmenting path algorithm (for layered networks with d = 8). In Figure 9.9, we give the ratios of the running times of these two algorithms (for both grid and layered networks).

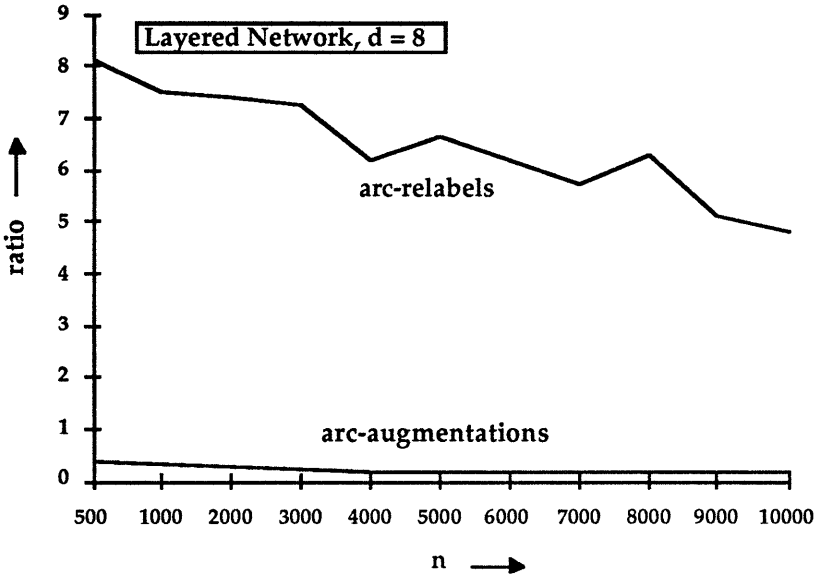


Figure 9.8. Ratios of arc-relabels and arc-augmentations for the capacity scaling algorithm and the shortest augmenting path algorithm.

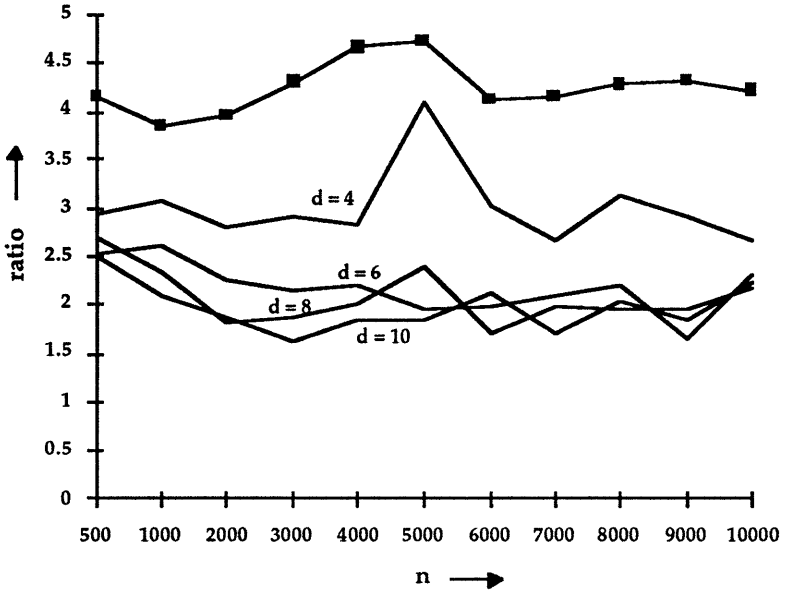


Figure 9.9. Ratio of the CPU times taken by the capacity scaling algorithm and the shortest augmenting path algorithm.

We observe from Figure 9.8 that the capacity scaling algorithm indeed performs fewer augmentations, but more relabels than the shortest augmenting path algorithm. This is consistent with the corresponding worst-case results stated in Table 9.2. Overall we find from Figure 9.9 that the capacity scaling algorithm is about 1.5 to 5 times slower than the shortest augmenting path algorithm on our test problems, depending upon the network type and network density. We also observe that for d = 6 or 8 or 10, the relative performance of the capacity scaling algorithm is much better than for d = 4. This is possibly due to the fact that the shortest augmenting path algorithm performs more augmentations for more dense problems and their contribution to the CPU time is larger.

In contrast to the worst-case results, the capacity scaling algorithm is slower empirically than the shortest augmenting path algorithm. The capacity scaling algorithm saves on the augmentation time, but increases the relabel time; overall, more time is spent. We have observed earlier that empirically the bottleneck step in the shortest augmenting path algorithm is the relabel time. Therefore, the capacity scaling algorithm is not as attractive from an empirical point of view.

The capacity scaling algorithm uses a parameter β called the *scale factor*, which denotes the factor by which the approximation of the capacity increases from phase to phase. In our computational tests we let $\beta = 2$, but we also ran a series of tests with other scale factors to investigate the effects of the scale factor on the CPU time.. As a function of the scale factor β, the worst-case time for the augmentations is $O(nm\ \beta\ \log_\beta U)$ and the worst-case time for relabels is $O(nm \log_\beta U)$ (see Ahuja and Orlin [1991]). We show in Figure 9.10, the CPU time of the capacity scaling algorithm as β varies from 2 to 10. It is evident from the figure that as β increases, the empirical running time decreases. AS β increases, the relabel time increases while the augmentation time decreases; however, the relabel time is the asymptotic bottleneck operation, and so larger scale factors tend to lead to a decreased running time. As a matter of fact, as β increases, the capacity scaling algorithm becomes more like the shortest augmenting path algorithm. (For β very large, the capacity scaling algorithm becomes the augmenting path algorithm.)

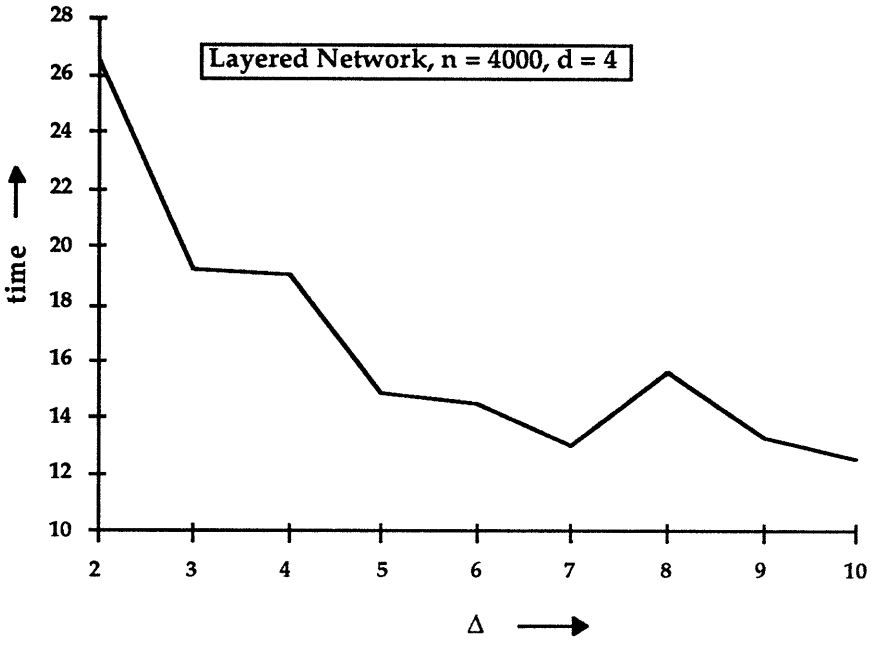


Figure 9.10. CPU time for the capacity scaling algorithm as a function of the scale factor Δ.

Dinic's Algorithm

We also conducted extensive tests of Dinic's algorithm. We noted earlier in Section 6 that Dinic's algorithm is equivalent to the shortest augmenting path algorithm in the sense that they perform the same sequence of augmentations The main difference between the two algorithms is that Dinic's algorithm constructs layered networks and the shortest augmenting path algorithm maintains distance labels. Our empirical tests yielded results that are consistent with this result and we found that for Dinic's algorithm the number of augmentations (and arc-augmentations) and the arcs scanned to construct layered networks have been comparable to corresponding numbers for the shortest augmenting path algorithm, and have almost the same growth rates. Hence we have omitted the presentations of detailed results for this algorithm. We graph the comparative running times of these two algorithms in Figure 9.11. We observe that the running times of these two algorithms are within 20% of one-another, and the two algorithms have roughly comparable CPU times.

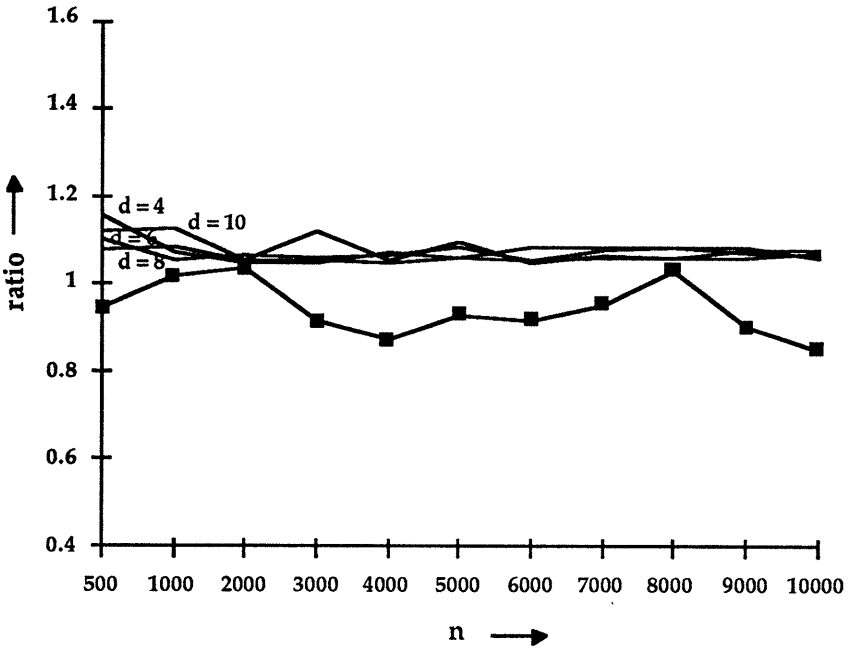


Figure 9.11. Ratio of the computational time of Dinic's algorithm and the shortest augmenting path algorithm.

Dinic's algorithm proceeds by constructing layered networks. In the worst-case, the algorithm may construct as many as n layered networks, but in practice it is rarely so and the algorithm constructs far fewer layered networks. In our tests, Dinic's algorithm constructs approximately $n^{.7}$ layered networks. Figure 9.12 plots $L/n^{0.7}$, and the plot suggests that $n^{0.7}$ is a good upper bound on the number of layered networks formed. In particular, there does not appear to be any trend where the ratio increases with increasing problem size.

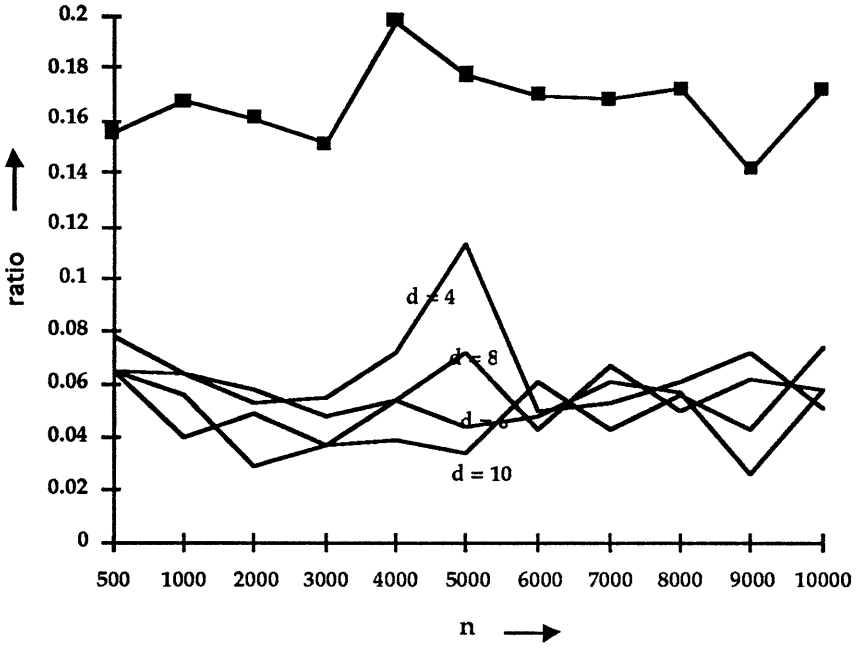


Figure 9.12. Ratio of the number of layered networks constructed by Dinic's algorithm to $n^{.7}$.

10. COMPUTATIONAL RESULTS OF PREFLOW-PUSH ALGORITHMS

In this section, we present computational results of the preflow-push algorithms described in Section 7. We tested the following preflow-push algorithms: highest-label, FIFO, LIFO, excess-scaling algorithms, and Karzanov's algorithm. Out of these, we focus more on the highest-label algorithms, which empirically is the fasterst of the algorithms that we tested.

Highest-Label Preflow-Push Algorithms

In Figure 10.1, we show the CPU times taken by the highest-label preflow-push algorithm for layered as well as grid networks. We have shown earlier that for the preflow-push algorithms, the representative operations are (i) α_r, the arc-relabels; and (ii) α_p, the pushes. To identify the asymptotic bottleneck operation, we plot the ratio $\alpha_r/(\alpha_r + \alpha_p)$ as a function of n in Figure 10.2, which shows an upward trend. This graph suggests that arc-relabels grow at a slightly faster pace than does pushes. This contrasts with the worst-case analysis where arc-relabels grow at a much slower pace than the pushes. This finding for the preflow-push algorithm is similar to our finding for the shortest augmenting path algorithm where we observed that arc-relabels take more time empirically than the arc-augmentations.

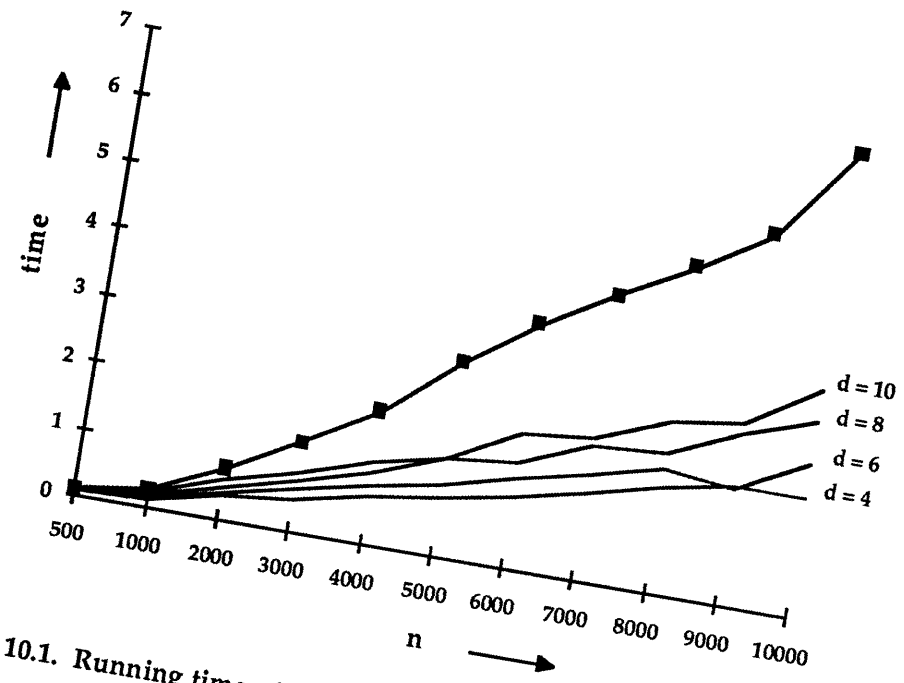


Figure 10.1. Running time of the preflow-push highest label algorithm.

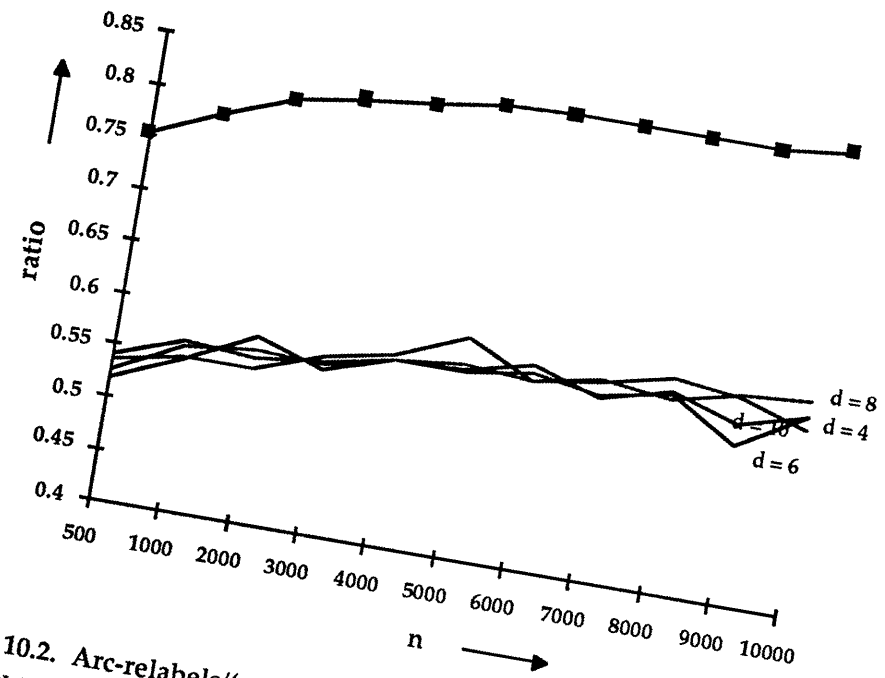


Figure 10.2. Arc-relabels/(arc-relabels+pushes) for the preflow-push highest label algorithm.

We next estimated the CPU time using regression analysis. Once again, we refer to the estimate as the virtual running time. We obtain the following expression of the running time (with R^2 equal to 0.9996).

$$V(I) = 5 \times 10^{-6} \times \alpha_r + 14 \times 10^{-6} \times \alpha_p.$$

Figure 10.3 gives the plot of V(I)/CPU(I). This plot confirms the high quality of the estimate. We next estimate the proportion of the time spent on the pushes, and we plot this graph in Figure 10.4. We find that though the highest-label preflow-push algorithm performs many more arc-relabels than the pushes (for layered networks), it spends more time performing pushes than relabels. This is because we estimate that a push takes about three times more CPU time than does a scan of an arc in a relabel.

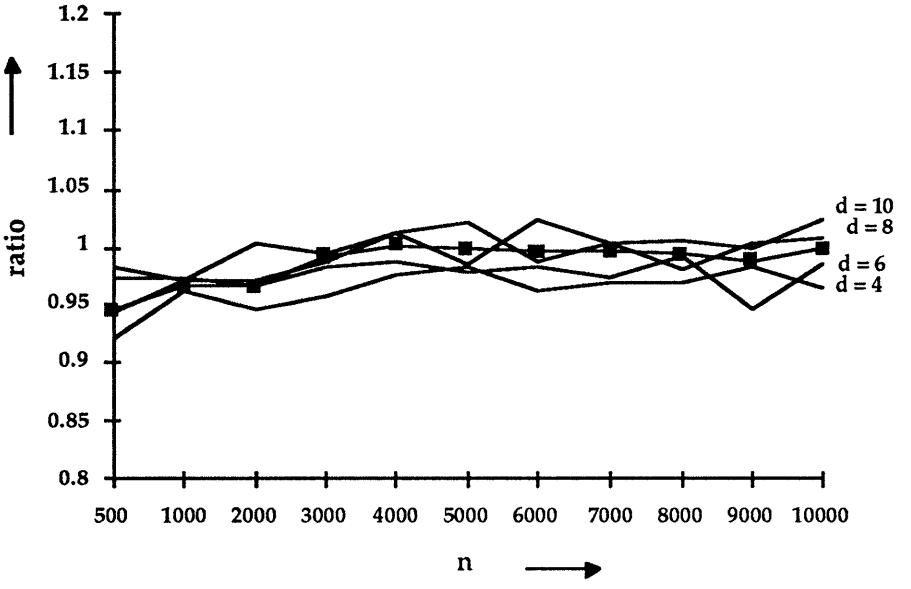


Figure 10.3. Ratio of the virtual time to the CPU time for the preflow-push highest label algorithm.

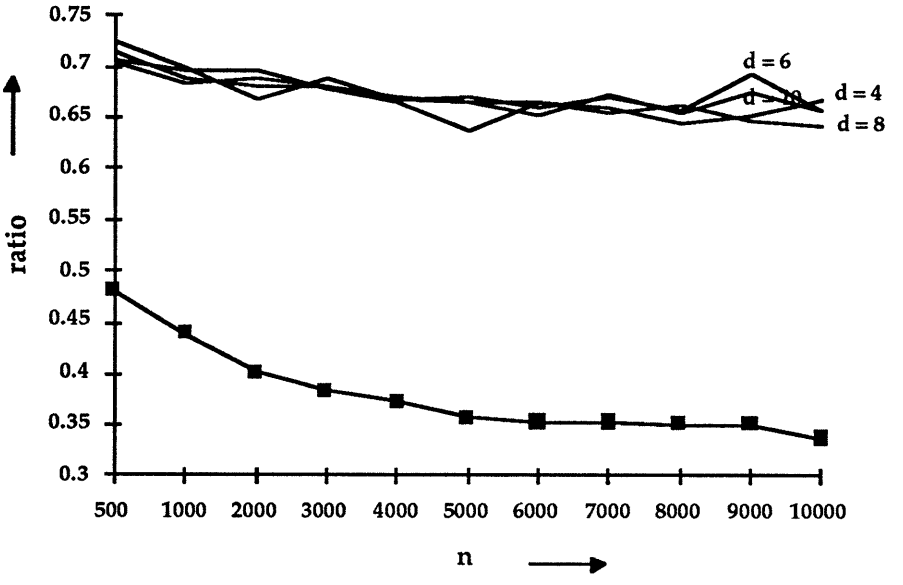


Figure 10.4. Ratio of time for pushes to virtual time for the preflow-push highest label algorithm.

We also assessed the growth rates of the two representative operation counts, arc-relabels and pushes, as a function of n and d for both the network generators. We used regression analysis to estimate these functions, which are given below.

	Layered Network	Grid Network
arc-relabels	$1.03 \times n^{1.18} d^{0.64}$	$1.20 \times n^{1.47}$
pushes	$1.46 \times n^{1.1} d^{0.64}$	$1.23 \times n^{1.28}$

Figure 10.5 plots the ratios of the estimated arc-relabels to the actual arc-relabels and the ratio of the estimated pushes to the actual pushes for the grid network. We found most ratios to be in the interval [0.9,1.1] for the grid and the layered networks. From the growth rates given in above table, we conjectured that empirically the running time of the highest-label preflow-push algorithm for reasonably dense networks will be bounded by $c\, n^{1.5}$, for some constant c. To verify this conjecture, in Figure 10.6, we plot the ratio CPU time/$(cn^{1.5})$ for $c = 10^{-6}$, and for all the network types and densities we find that these plots have the downward trend.

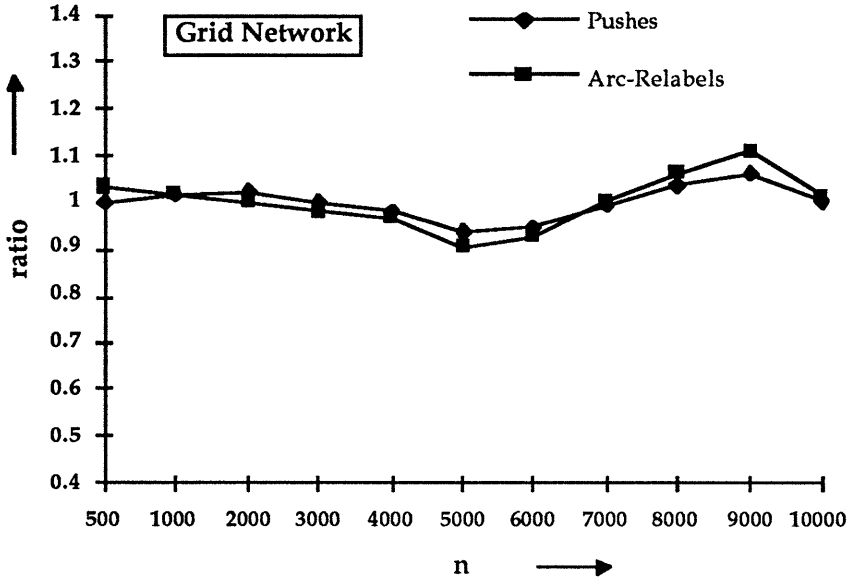


Figure 10.5. Ratio of predicted to actual arc-relabels and pushes for the preflow-push highest label algorithm.

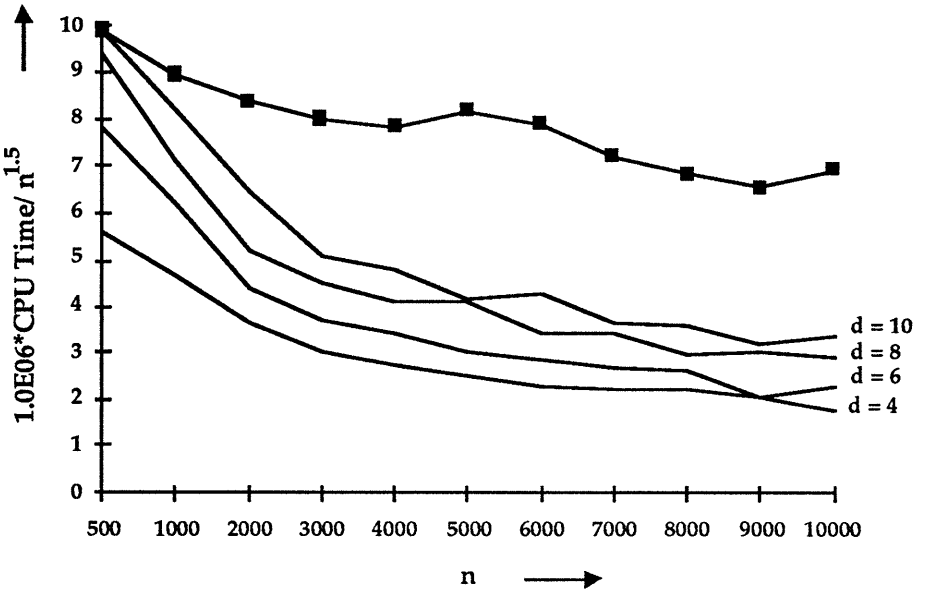


Figure 10.6. Ratio of CPU Time taken to $n^{1.5}$ for the preflow-push highest label algorithm.

As described in Section 7, we implemented a two phase version of the preflow-push algorithms. The first phase establishes a maximum preflow and the second phase converted this maximum preflow into a maximum flow. The CPU times, relabels and the pushes, whose plots are given earlier include both phases. We also investigated how these operations are split between the two phases. We show in Figure 10.7 the ratio of the second phase pushes divided by the total pushes and also the second phase relabels divided by the total relabels. Though these graphs are for the second phase of the highest-label preflow-push algorithm applied to grid networks, they also represent typical behavior of all preflow-push algorithms on grid as well as layered networks. It is evident from these figures that the pushes performed in the second phase are typically less than 5% of the total pushes, and the percentage is decreasing for larger problem sizes. In fact, we observed that the pushes performed in the second phase are typically less than n. For the layered networks, the number of relabels in the second phase were zero for all instances that we tested, and for grid networks the number of relabels in the second phase was quite low. From these observations, we can conclude that the operations performed in the second phase are rather insignificant in comparison with the first phase operations, and thus further speedups of the second phase of the preflow-push algorithm will not have much of an impact on the CPU time.

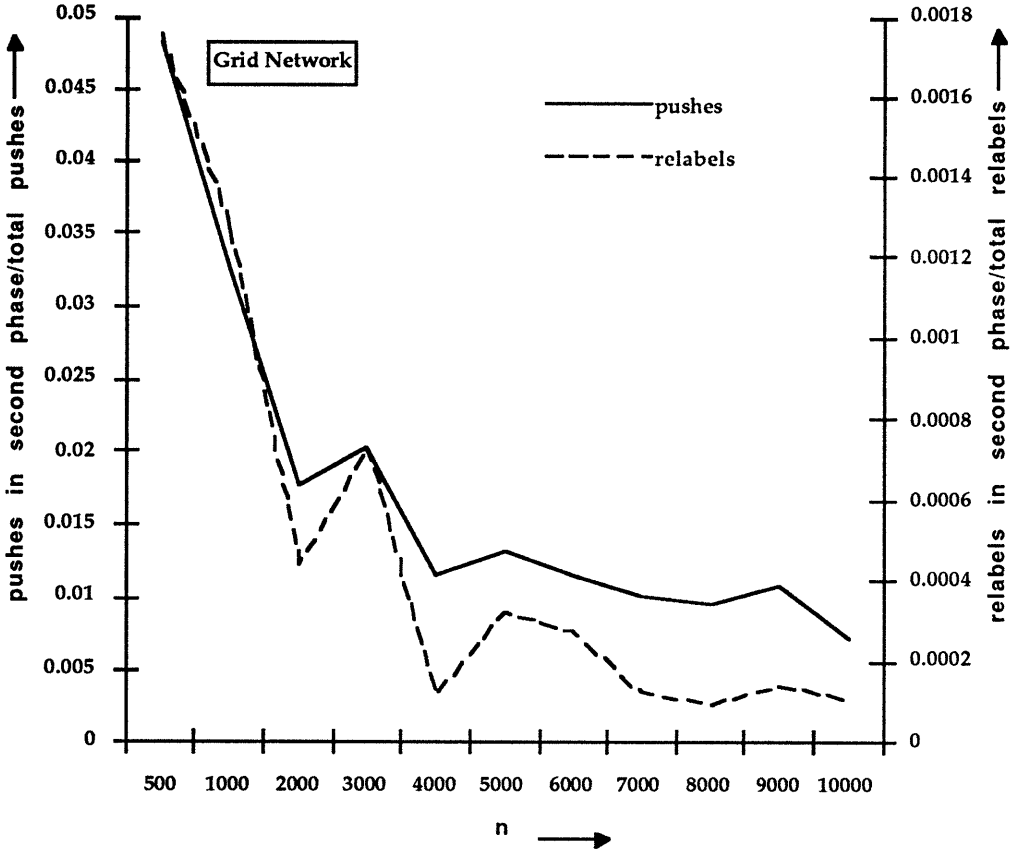


Figure 10.7. Ratio of pushes and relabels in second phase to total pushes and total relabels.

In several previous studies, researchers have investigated dynamic tree implementations of Dinic's and preflow-push algorithms. The concensus conclusion of these studies is that dynamic tree implementations are slower than the original implementations. Our computational results help to explain this phenomenon. In general, dynamic trees decrease the time for nonsaturating pushes and augmentations at the expense of increasing the time for relabels. Moreover, the dynamic tree data structure is quite intricate, and involves a large constant overhead. Our computational study reveals that for both the network types arc-relabels grow faster than the pushes and so dynamic trees increase the running time of the operation that is the bottleneck empirically.

Excess-Scaling Algorithms

We also performed computational tests of the (original) excess-scaling algorithms and its two improvements: the stack-scaling algorithm and the wave-scaling algorithm. Recall from Section 7 that all of these algorithms have the following representative operations: (i) arc-relabels; (ii) pushes; and (iii) the initialization time which is O(n log U). Figure 10.8 presents the CPU times taken by these three algorithms on grid networks. The stack-scaling algorithm performs the least number of arc-relabels and pushes and consequently ends up taking the least amount of CPU times. We suggest the following explanation for the relative superiority of the stack-scaling algorithm over the other excess-scaling algorithms. First recall that the original excess scaling algorithm performs push/relabels from a large excess node with the minimum distance label, the stack-scaling algorithm performs push/relabels from a large excess node with the highest distance label, and the wave-scaling algorithm is a hybrid of FIFO and highest-label pushing over large excess nodes. We have found earlier that the highest-label preflow-push algorithm is the fastest maximum flow algorithm. We suggest that the stack-scaling algorithm is superior to the other two scaling algorithms because highest level pushing is superior to other pushing strategies.

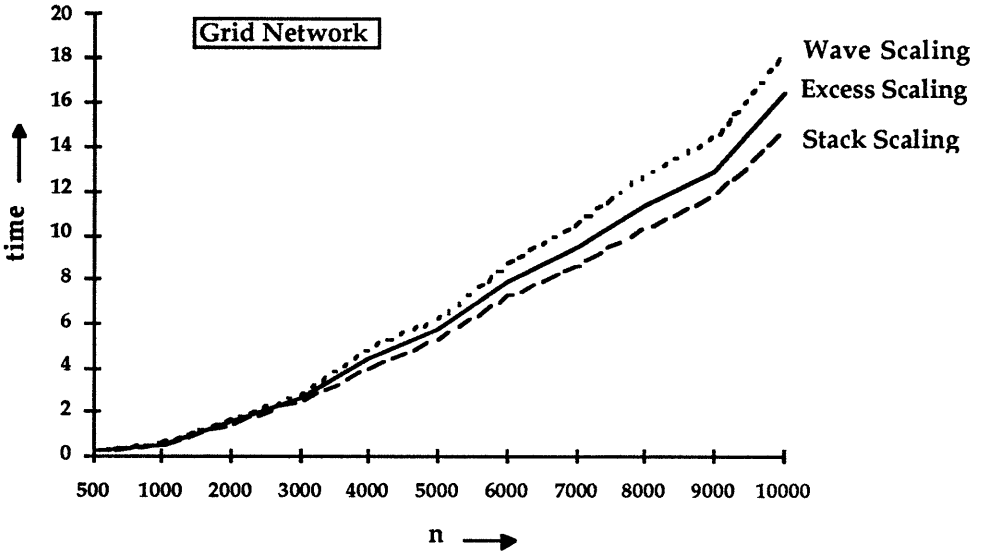


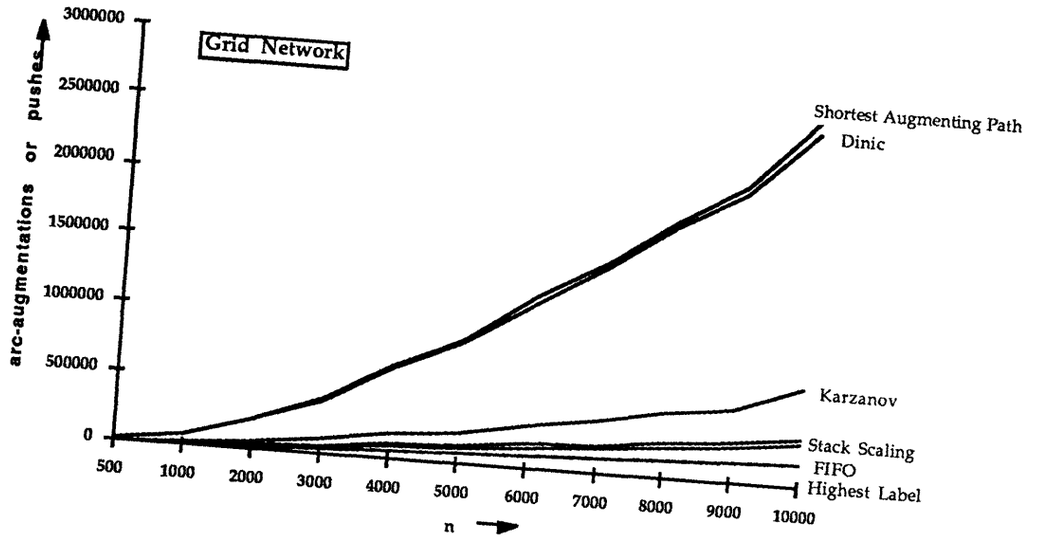
Figure 10.8. CPU times of the excess-scaling algorithms.

We also performed the virtual time analysis of the stack scaling algorithm. We found that for large sized networks the algorithm spends roughly half of the time in performing pushes on layered networks, and roughly 1/3 of the time for grid networks for the problem sizes that we tested. For grid networks, we estimate that arc-relabels grow at the rate $1.14 \times n^{1.5}$ and that the pushes grow at the rate $0.88 \times n^{1.4}$. Similar to other maximum flow algorithms, arc-relabels are estimated to grow at a faster pace than the pushes.

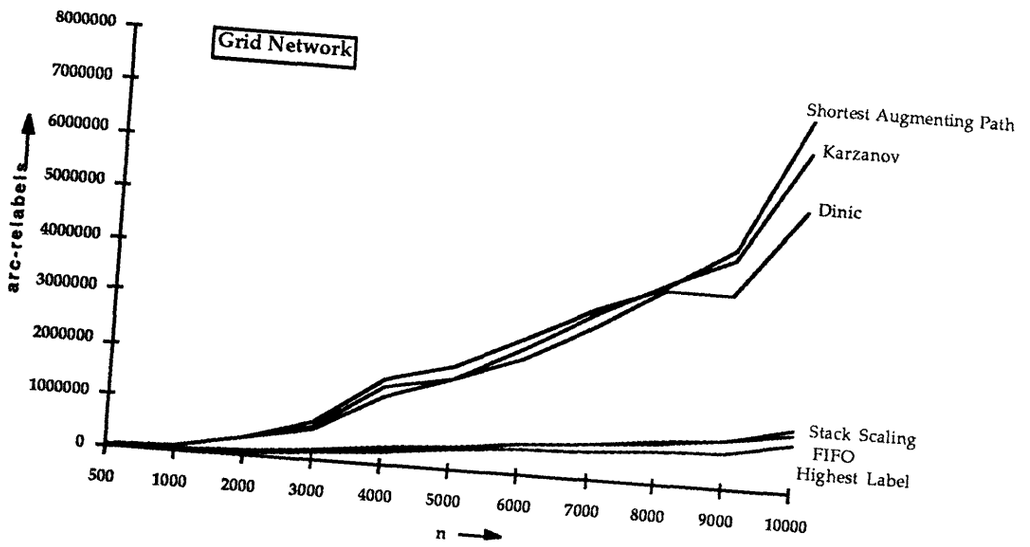
Other Preflow-Push Algorithms

In this section, we present some results about the FIFO preflow-push algorithm, the lowest-label preflow-push algorithm, and Karzanov's algorithm. We primarily focus on the comparative behavior of these algorithms with the shortest augmenting path algorithm, Dinic's algorithm, highest-label preflow-push algorithm, and the stack-scaling algorithm. Figure 1.1 gives the CPU times of these algorithms. It is evident from this figure that the highest-label preflow-push algorithm is fastest among the algorithms tested by us. The FIFO preflow-push algorithm is the second fastest algorithm for our tests. It is 1.5 to 2 times slower than the highest-label preflow-push algorithm. The performance of the stack-scaling algorithm is also attractive; it is about twice slower than the highest-label preflow-push algorithm. We find that the previous two best algorithms, namely Dinic's and Karzanov's algorithms, are substantially slower than the recently developed preflow-push algorithms. For large size grid networks, these two algorithms are about 7 to 10 times slower than the highest-label preflow-push algorithm. We can also verify from Table 1.2 that for layered networks, the performance of the highest-label preflow-push algorithms is even more attractive.

We have observed earlier that all the maximum flow algorithms tested by us have two representative operations: (i) arc-relabels; and (ii) either of arc-augmentations and pushes. To see, how the counts of these two representative operations vary for various algorithms, we plot them in Figure 10.9. We observe that the comparative growth rates of these two representative operations exhibit the same behavior as the growth rates of the computational times.



(a)



(b)

Figure 10.9. Representative operation counts for algorithms (a) arc-augmentaions, (b) arc-relabels.

To gain insight into how the pushes performed by the algorithms are split between the nonsaturating and saturating pushes, we plot in Figure 10.10 the ratio of nonsaturating pushes by total pushes. The percentage of the saturating pushes decreases as the problem size increases, but for the highest-label preflow-push algorithm, the saturating pushes are about 40% of the total pushes. The percentage of effort spent on on-saturating pushes is less for the more efficient algorithms. In particular, the highest-label preflow-push algorithm performs the fewest number of non-saturating pushes, and the shortest augmenting path algorithm performs the most.

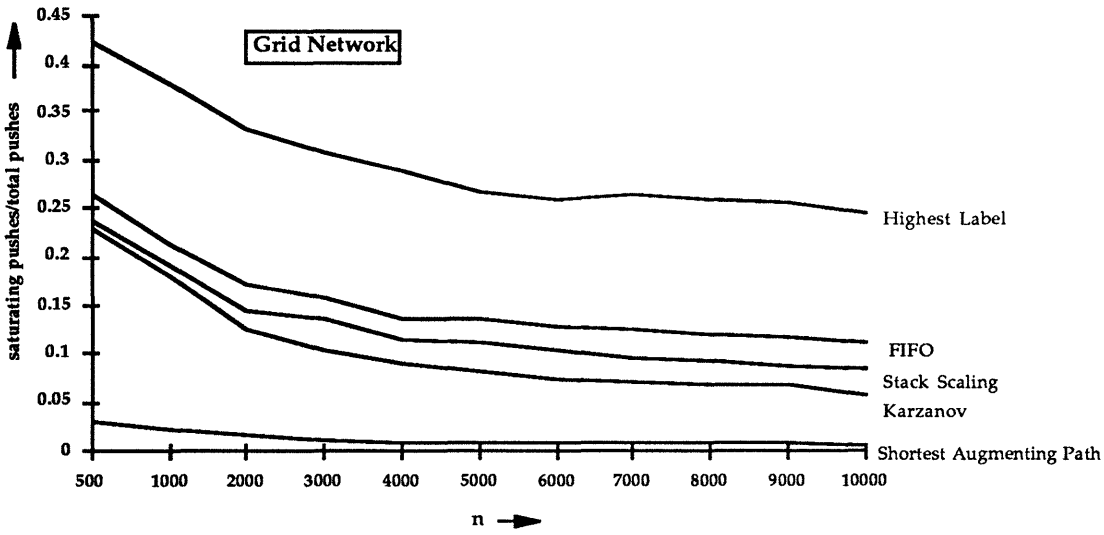
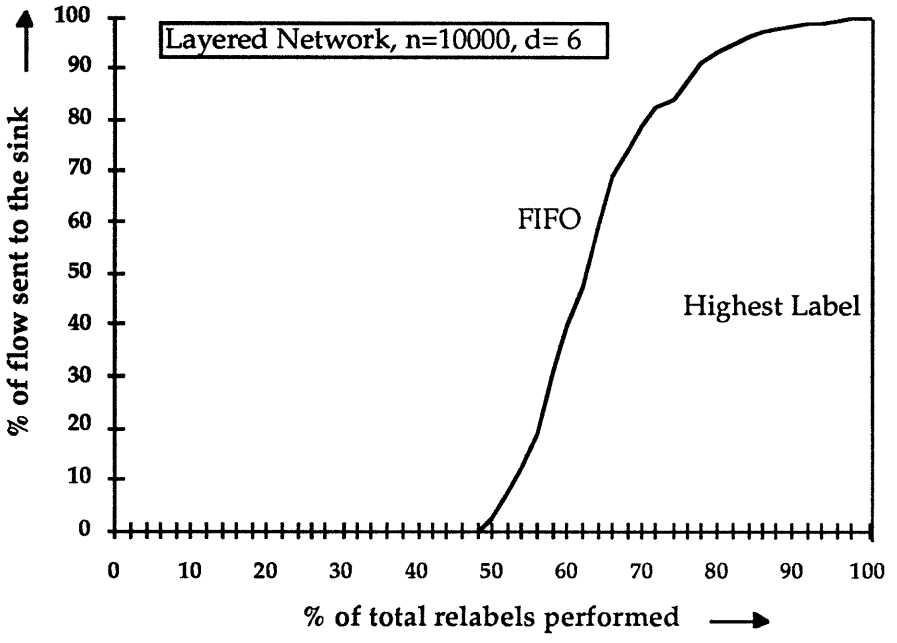
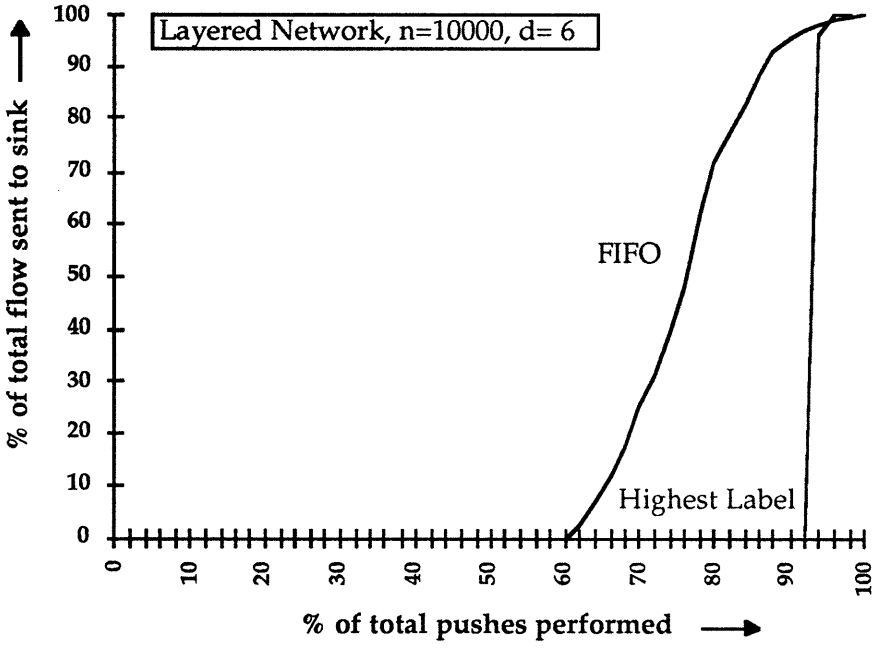


Figure 10.10. Ratio of saturating pushes to total pushes performed by the algorithms.

We also investigated how the flow reaches the sink for some preflow-push algorithms as the algorithm proceeds. Figures 10.11(a) and 10.11(b) respectively give a plot of the amount of flow reaching the sink as a function of the number of relabels and pushes. We observe that the highest-label preflow-push algorithm sends all the flow in the last 5% of the iterations. The FIFO preflow-push algorithm starts sending the flow into the sink after half of its execution is over, then sends flow at the steady pace, and slows down in the last part of the algorithm.



(a)



(b)

Figure 10.11. Flow reaching the sink as a function of (a) total relabels (b) total pushes performed.

11. COMPUTATIONAL RESULTS FOR DIMACS BENCHMARK INSTANCES

To validate our findings obtained on tests on the grid and layered networks, we also tested our algorithms on DIMACS benchmark instances; i.e., the problems used in *"The First DIMACS International Algorithm Implementation Challenge"* held during October 1991 at RUTCOR, Rutgers University, Rutgers, NJ. We tested our algorithms on three additional network generators: GL(Genrmf-Long), GW(Genrmf-Wide), and WLM(Washington-Line-Moderate). The details of these networks can be found in Badics, Boros, and Cepek [1992]. For the problems used, exact parameters for the families GL, GW, and WLM are given in Table 11.1.

In Table 11.2, we give the CPU times of the maximum flow algorithms when applied to the above benchmark instances. In Figure 11.1, we give a plot of these times. It is evident from these plots that Goldberg-Tarjan's highest-label preflow-push algorithm is again the winner among all the maximum flow algorithms and the Ahuja-Orlin-Tarjan's stack-scaling algorithm is the second best for GL and GW networks; but for WLM networks the FIFO preflow-push algorithms is the second best algorithm. In Figure 11.2, we plot the ratios of the arc-relabels by the sum of arc-relabels and arc-pushes for the highest-label preflow-push algorithm. We observe that the arc-relabels constitute a larger proportion of the total representative operations and they do seem to possess a moderately increasing trend. To assess whether the growth rate of the empirical running time of the highest-label preflow-push is $O(n^{1.5})$ for these class of instances, we plot the ratio of CPU time x $10^6/n^{1.5}$ in Figure 11.3. The GW networks appear to have a growth rate higher than $n^{1.5}$. For this class, we plot the ratio of CPU time x $10^6/n^{1.8}$ and find that $n^{1.8}$ is a reasonable bound on its growth rate. We verified that all others major conclusions we list in Section 1 for grid and layered networks are valid for the DIMACS benchmark instances too.

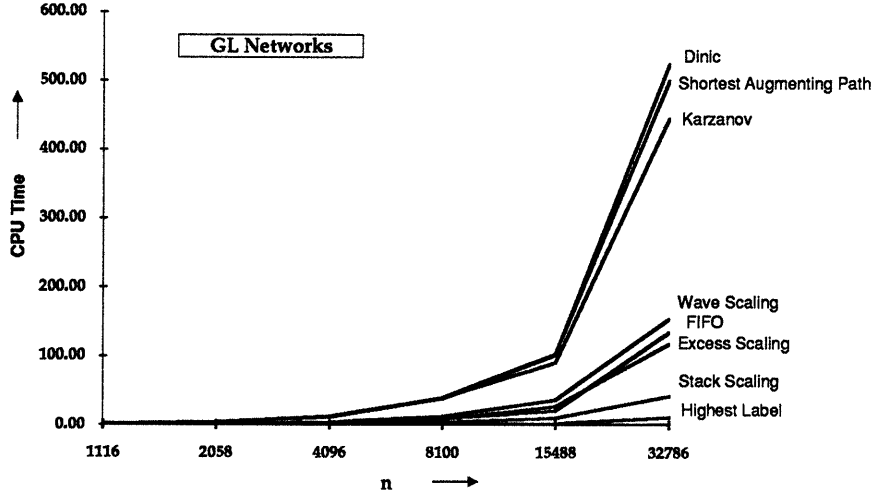
Name	Parameters	Nodes	Arcs
GL1	a=6 b=31 c_1=1 c_2=10,000	1,116	4,800
GL2	a=7 b=42 c_1=1 c_2=10,000	2,058	9,065
GL3	a=8 b=64 c_1=1 c_2=10,000	4,096	18,368
GL4	a=9 b=100 c_1=1 c_2=10,000	8,100	36,819
GL5	a=11 b=128 c_1=1 c_2=10,000	15,488	71,687
GL6	a=13 b=194 c_1=1 c_2=10,000	32,786	153,673
GW1	a=16 b=4 c_1=1 c_2=10,000	1,024	4,608
GW2	a=21 b=5 c_1=1 c_2=10,000	2,205	10,164
GW3	a=28 b=5 c_1=1 c_2=10,000	3,920	18,256
GW4	a=37 b=6 c_1=1 c_2=10,000	8,214	38,813
GW5	a=48 b=7 c_1=1 c_2=10,000	16,128	76,992
GW6	a=64 b=8 c_1=1 c_2=10,000	32,768	157,696
WLM1	6 64 4 5	258	1,236
WLM2	6 128 4 8	514	3,965
WLM3	6 256 4 8	1,026	8,071
WLM4	6 512 4 11	2,050	22,289
WLM5	6 1024 4 16	4,098	65,016
WLM6	6 2048 4 22	8,194	179,235

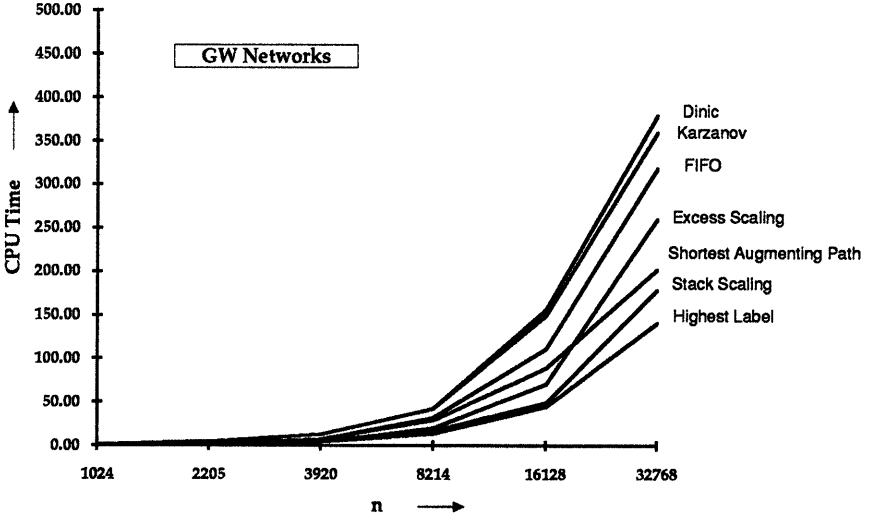
TABLE 11.1 Details of DIMACS benchmark instances.

			Shortest			PREFLOW-PUSH		EXCESS SCALING			
Network			Aug.	Capacity	Dinic	Highest	FIFO	Excess	Stack	Wave	Karzanov
Generator	NODES	ARCS	Path	Scaling		Label		Scaling	Scaling	Scaling	
GL Networks	1116	4800	1.23	4.74	1.02	0.11	0.39	0.37	0.23	0.54	1.11
	2058	9065	4.34	15.36	3.75	0.29	1.33	1.12	0.60	1.43	4.04
	4096	18368	12.02	48.08	11.94	0.69	3.37	3.20	1.54	4.20	12.12
	8100	36819	38.94	148.54	38.46	1.33	9.70	9.15	3.97	12.02	39.18
	15488	71687	90.46	335.28	101.81	2.09	20.42	27.08	10.86	36.22	102.23
	32786	153673	444.48	1646.50	523.67	11.18	134.50	117.88	42.12	153.61	499.05
GW Networks	1024	4608	0.74	5.45	1.21	0.43	0.63	0.54	0.54	0.59	1.23
	2205	10164	3.49	23.66	5.36	1.63	3.28	2.28	2.02	2.35	5.37
	3920	18256	7.41	59.59	13.38	4.16	7.91	5.48	4.83	5.88	13.22
	8214	38813	29.49	216.01	42.74	14.36	32.85	20.65	17.64	29.33	43.19
	16128	76992	89.01	596.34	156.21	45.32	111.27	70.87	49.50	-	150.05
	32768	157696	201.71	1415.74	379.82	140.91	318.09	260.24	178.25	-	359.94
WLM Networks	258	1236	0.02	0.10	0.03	0.01	0.01	0.02	0.02	0.02	0.03
	514	3965	0.08	0.42	0.10	0.03	0.03	0.05	0.05	0.06	0.09
	1026	8071	0.21	1.14	0.27	0.06	0.08	0.13	0.11	0.11	0.25
	2050	22289	1.14	3.08	1.27	0.11	0.16	0.57	0.40	0.42	1.14
	4098	65016	3.13	10.64	4.04	0.31	0.46	1.57	1.08	0.88	3.87
	8194	179235	8.59	55.74	11.58	0.87	1.28	4.00	2.70	1.77	9.94

Note: A '-' indicates that the problem was not tested.

Table 11.2. CPU Time (in seconds on DECSYSTEM-5000) taken by algorithms on the networks used in DIMACS Challenge.





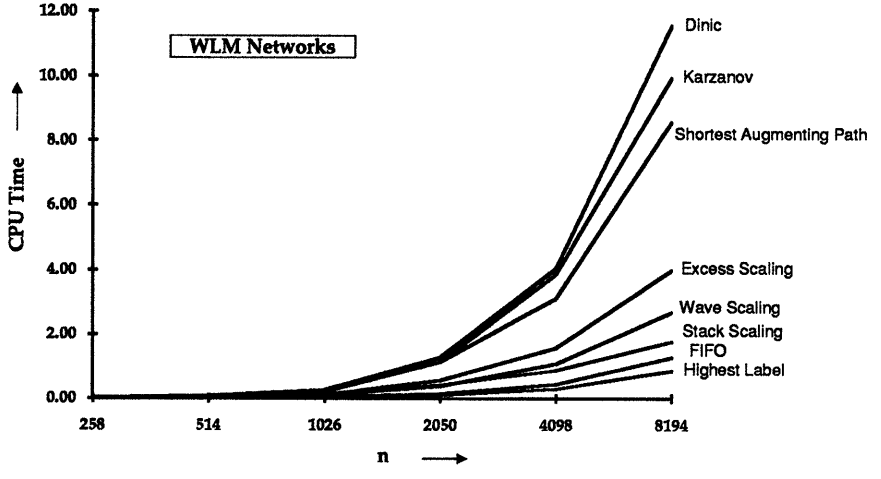


Figure 11.1. Comparison of CPU time (in seconds) taken by algorithms on the DIMACS Challenge networks.

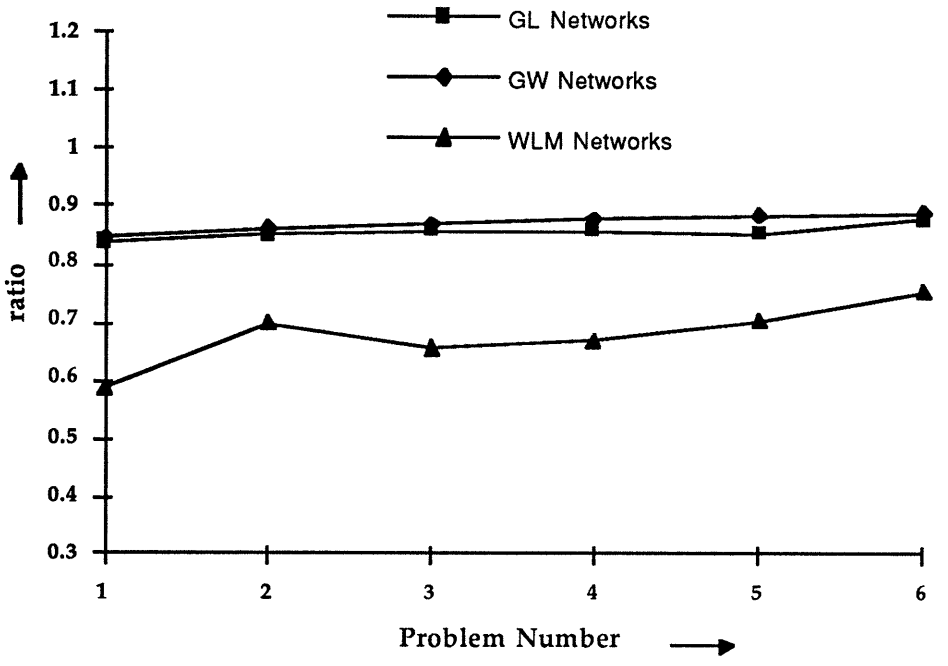


Figure 11.2. Ratio of arc-relabels to (arc-relables + pushes) for the highest label preflow-push algorithm.

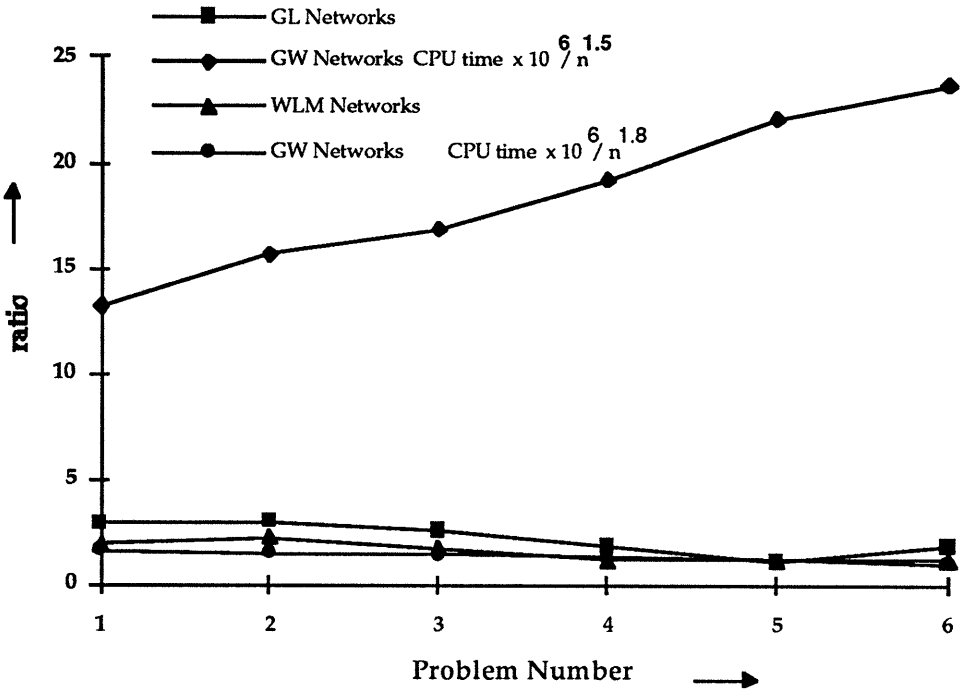


Figure 11.3 Ratio of CPU Time x 10^6 to $n^{1.5}$ for the highest label preflow-push algorithm.

REFERENCES

AHUJA, R. K., and J. B. ORLIN. 1989. A fast and simple algorithm for the maximum flow problem. *Operations Research* **37**, 748-759.

AHUJA, R. K., and J. B. ORLIN. 1991. Distance-directed augmenting path algorithms for maximum flow and parametric maximum flow problems. *Naval Research Logistics* **38**, 413-430.

AHUJA, R. K., and J. B. ORLIN. 1995. Use of Representative Operation Counts in Computational Testings of Algorithms. Sloan School Working Paper WP # 3459. Written July, 1992. Revised, March 1995.

AHUJA, R. K., J. B. ORLIN, and R. E. TARJAN. 1989. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing* **18**, 939-954.

AHUJA, R. K., T. L. MAGNANTI, and J. B. ORLIN. 1991. Some recent advances in network flows. *SIAM Review* **33**, 175-219.

AHUJA, R. K., T. L. MAGNANTI, and J. B. ORLIN. 1993. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, New Jersey.

ALON, N. 1990. Generating pseudo-random permutations and maximum flow algorithms. *Information Processing Letters* **35**, 201-204.

ANDERSON, R. J., and J. C. SETUBAL. 1992. Parallel and sequential implementations of maximum-flow

algorithms. In D. S. Johnson and C. C. McGeoch, editors, *Dimacs Implementation Challenge Workshop: Algorithms for Network Flows and Matching.* DIMACS Technical Report 92-4.

BADICS, T., E. BOROS, and O. CEPEK. 1992. Implementing a new maximum flow algorithm. In D. S. Johnson and C. C. McGeoch, editors, *Dimacs Implementation Challenge Workshop: Algorithms for Network Flows and Matching.* DIMACS Technical Report 92-4.

CHERIYAN, J., and S. N. MAHESHWARI. 1989. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal of Computing* 6, 1057-1086.

CHERIYAN, J., and T. HAGERUP. 1989. A randomized maximum flow algorithm. *Proceedings of the 30thIEEE Conference on the Foundations of Computer Science,* 118-123.

CHERKASSKY, R. V. 1977. Algorithm of construction of maximum flow in networks with complexity $O(V^2E^{1/2})$ operations. *Mathematical Methods of Solution of Economical Problems* 7, 112-115.

CHEUNG, T. 1980. Computational comparison of eight methods for the maximum network flow problem. *ACM Transactions on Mathematical Software* 6, 1-16.

DERIGS, U., and W. MEIER. 1989. Implementing Goldberg's max-flow algorithm: A computational investigation. *Zeitschrift für Operations Research* 33, 383-403.

DINIC, E. A. 1970. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady* **11**, 1277-1280.

EDMONDS, J., and R. M. KARP. 1972. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of ACM* **19**, 248-264.

FORD, L. R. Jr., and D. R. FULKERSON. 1956. Maximum flow through a network. *Canadian Journal of Mathematics* 8, 399-404.

GABOW, H. N., 1985. Scaling algorithms for network flow problems. *Journal of Computer and System Sciences* **31**, 148-168.

GALIL, Z. 1980. An $O(n^{5/3}m^{2/3})$ algorithm for the maximum flow problem. *Acta Informatica* **14**, 221-242.

GALIL, Z., and A. NAMAAD. 1980. An $O(nm \log^2 n)$ algorithm for the maximum flow problem. *Journal of Computer and System Sciences* **21**, 203-217.

GLOVER, F., D. KLINGMAN, J. MOTE and D. WHITMAN. 1983. Comprehensive computer evaluation and enhancement of maximum flow algorithms. *Applications of Management Science* 3, 109-175.

GLOVER, F., D. KLINGMAN, J. MOTE and D. WHITMAN. 1984. A primal simplex variant for the maximum flow problem. *Naval Research Logistics Quarterly* **31**, 41-61.

GOLDBERG, A. V., M. D. GRIGORIADIS, and R.E. TARJAN 1991. Efficiency of the network simplex algorithm for the maximum flow problem. *Mathematical Programming* (A) **50**, 277-290.

GOLDBERG, A. V, and R. E. TARJAN. 1986. A new approach to the maximum flow problem. *Proceedings of the 18th ACM Symposium on the Theory of Computing,* 136-146. Full paper in *Journal of ACM* **35**(1988), 921-940.

GOLDFARB, D. and M. D. GRIGORIADIS. 1988. A computational comparison of the Dinic and network

simplex methods for maximum flow. *Annals of Operations Research* **13**, 83-123.

GOLDFARB, D., and J. HAO. 1990. A primal simplex algorithm that solves the maximum flow problem in at most nm pivots and $O(n^2m)$ time. *Mathematical Programming* **47**, 353-365.

GOLDFARB, D., and J. HAO. 1991. On Strongly Polynomial Variants of The Network Simplex Algorithm forthe Maximum Flow Problem. *Operations Research Letters* **10**, 383-387.

HAMACHER, H. W. 1979. Numerical investigations on the maximal flow algorithm of Karzanov. *Computing* **22**, 17-29.

IMAI, H. 1983. On the practical efficiency of various maximum flow algorithms. *Journal of the Operations Research Society of Japan* **26**, 61-82.

KARZANOV, A. V. 1974. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady* **15**, 434-437.

KLINGMAN, D., A. NAPIER, and J. STUTZ. 1974. NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems. *Management Science* **20**, 814-821.

MALHOTRA, V. M., M. P. KUMAR, and S. N. MAHESHWARI. 1978. An $O(n^3)$ algorithm for finding maximum flows in networks. *Information Processing Letters* **7**. 277-278.

NGUYEN, Q. C., and V. VENKATESHWARAN. 1992. Implementations of the Goldberg-Tarjan maximum flow algorithm. In D. S. Johnson and C. C. McGeoch, editors, *Dimacs Implementation Challenge Workshop: Algorithms for Network Flows and Matching.* DIMACS Technical Report 92-4.

SHILOACH, Y. 1978. An $O(nI \log^2 I)$ maximum flow algorithm, Technical Report. STAN-CS-78-702. Computer Science Dept., Stanford University, Stanford, CA.

SLEATOR, D. D., and R. E. TARJAN. 1983. A data-structre for dynamic trees, *Journal of Computer and Systems Sciences* **24**, 362-391.

TARJAN, R. E. 1983. *Data Structures and Network Algorithms.* SIAM, Philadelphia, PA.