DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume **00**, 0000

Performance Evaluation of the MINET Minimum Cost Netflow Solver

ISTVÁN MAROS

ABSTRACT. In this paper we give account of the experiences obtained when participating in the DIMACS International Implementation Challenge with MINET simplex based minimum cost network flow (MCNF) LP solver. MINET has some distinguishing features among network simplex solvers in pricing and in selecting the leaving arc. With some fixed setting of strategic parameters of MINET, on the given set of test problems an $O(mn)$ performance in solution time was observed. MINET was also compared with NETFLO an industry standard netflow solver. The effect of two "external" conditions, namely, code optimization and loading conditions of the computer are also analyzed and some important facts are revealed.

1. Introduction

Efficient solution of a wide variety of minimal cost network flow (MCNF) problems, due to its theoretical and practical importance, has long been a challenge for researchers in the field. The efforts resulted in a number of powerful algorithms to solve the MCNF problem. One of them is MINET, developed by the author. MINET has some distinguishing features in pricing and row selection that will be presented briefly in section 2.

The purpose of this paper is to give account of the experiences obtained participating in the DIMACS International Implementation Challenge (for details of the Challenge, see [**11**]). The nature of experiences is manifold. First, MINET was tested, as required by [**11**], on a set of predefined problems (7 problem groups of problems generated by NETGEN [**5**], and 8 problems from the original NETGEN set of 40 problems). The test results have been evaluated by statistical methods to find some relationship between problem characteristics and solution time. This discussion is presented in section 3. Next, a series of real–life problems from a specific family of problems (TRGEN) [**6**] was solved. A statistical

1991 *Mathematics Subject Classification.* 90C35.
This research was supported in part by Hungarian Research Fund OTKA 2587.

©0000 American Mathematical Society
0000-0000/00 $1.00 + $.25 per page

analysis was also carried out. The TRGEN family turned out to contain very degenerate problems. Therefore, it appeared appropriate to test a new anti–degeneracy row selection strategy [**10**] for reducing the number of degenerate iterations. The very interesting experiences obtained with this technique are also presented in section 4.

In section 5 we give account of a further category of tests when we used MINET to solve the "gte_bad" set of real–life problems that made the — otherwise excellent — RELAX–III optimizer behave very strangely producing extreme timings between 0.020 to 508.829 *sec.* MINET solved all these problems consistently within less than 0.06 *sec.*

To get an idea about the performance of MINET relative to other MCNF solvers, we were able to do some comparative testing with NETFLO. NETFLO is an industry standard MCNF solver. It has been made available for the participants of the Challenge. For comparisons we chose test problems from the original and the final DIMACS Core problems both. The results of this testing are presented and discussed in section 6.

We paid special attention to some "external" conditions that may influence the solution time, with the purpose of getting better insight into the reliability of the reported solution times. In this respect, we investigated the effect of two factors, namely code optimization and loading conditions of the computer. The issue of loading conditions is discussed in section 7, while the effect of code optimization is evaluated in section 3 where the relevant data are presented in the table of the NETGEN problems.

2. The algorithm of MINET

Experience has shown that computationally the most critical operation in simplex based MCNF algorithms is pricing (selecting the incoming edge variable). Its logical and computational simplicity are misleading (two integer additions and some indirect memory accesses per scanned nonbasic edges). It is not unusual that 50–95 percent of the total computational effort is spent on pricing.

There is another remarkable characteristic feature of MCNF algorithms, namely, on many real life problems, they tend to make a large number of degenerate steps, thus slowing down the convergence to an optimal solution.

In designing and developing MINET, we tried to address mainly the above two issues, in addition to the most important efficiency considerations [**7**], [**8**].

MINET is a primal simplex based minimum cost network flow solver. It was originally planned to be a module in a hierarchically designed special–purpose optimization algorithm. On the basis of its successful performance it was taken out of that framework and has been further developed as a stand–alone MCNF optimizer.

Efficiency of network LP systems heavily depends on the data structure applied. As a good compromise between extreme possibilities, MINET keeps track of the basic spanning tree (T) using the following list functions:

- $p(i)$ predecessor of node i,
- $s(i)$ thread successor of node i,
- $t(i)$ cardinality of subtree headed by node i,
- $f(i)$ last node in subtree headed by node i,
- $id(i)$ directed node identifier of the ith basic arc.

MINET is equipped with a newly designed column selection (pricing) technique, called NETPRI, described in [**9**], that can take advantage of a recognized structure of a problem. The purpose of this technique is to reduce the overall computational effort spent on pricing. It is controlled by 3 parameters. By special setting of them, it is possible to reproduce most of the known pricing strategies.

Due to the fact that randomly generated problems usually do not have any recognizable structure, this sophisticated pricing feature of MINET was not used during the experiments. The present fixed setting of the parameters reproduces the *least recently considered (LRC)* entering rule [**2**].

The most important parts of the main algorithmic steps of MINET, as used during the testing, can be summarized as follows.

Step 0. *Initialization.* Create all–logical basis. Set principal pricing pointer $p = m$ (the number of edges) and set up pricing vector.

Step 1. *Pricing.* Start pricing at $p+1 \pmod m$, price the nonbasic arcs and stop at the first improving arc. Set pointer p to the index of this arc. If no improving arc can be found, stop (solution optimal, or problem infeasible).

Step 2. *Pivot.* Trace the loop created by the entering arc in the T basis tree and, simultaneously, do the ratio test. Start from both endpoints of this arc and move upwards in the direction of the root of T. Move always on the smaller subtree. Stop ratio test immediately if a 0 ratio is encountered. The corresponding arc will leave the basis.

Step 3. *Updating.* Update solution, T, and pricing vector. Go to step 1.

Step 2 ensures that the "smaller subtree" will always be as small as possible, thus updating work will be reduced to a minimum.

Logically the most complicated part of the program is the updating of the basic spanning tree. In step 2, there is an effort to minimize this work. In–depth investigation has shown that, after all, updating is a less decisive factor in the running time of MINET than pricing. Table 5 and model (4.1) in section 4 show an example that, by Spearman's rank correlation, there may be a perfect positive relationship between the number of scanned variables and solution time. The same is usually not true for the relation between the number of iterations (which, in most cases, is a good estimate of the tree updates) and solution time. Similar observations have been made on a large number of other problems.

MINET has been used to solve large scale real–life problems for a couple of years. During this period it encountered a large number of very degenerate

problems. Such problems caused MINET to make many degenerate steps. In an effort to try to decrease the number of them, a new anti–degeneracy row selection procedure was introduced [**10**] and experimentally implemented in MINET. The idea behind it can briefly be sketched as follows.

During degenerate iterations the value of the objective function does not change if we do not want to violate the feasibility bounds of the basic and the out–going variables. On the other hand, the objective can be improved if we relax the feasibility bounds of these variables in every iteration and allow some or all of them to go slightly more and more infeasible. Namely, this always allows us to make a positive step, at least to the extent of the relaxation of the feasibility bounds. By this principle, at least a *minimum forced step* can be made at each iteration. It is important that the value of the objective function will always change, therefore, a past basis cannot come up again, thus cycling will not occur. At the same time, the ever growing infeasibility may cause troubles. However, if it is small enough, it can be kept under control.

Though a minimum forced step corresponds to an otherwise degenerate iteration, experience has shown that by this technique usually less time is spent on degenerate vertices (this procedure generates different pivot sequence), therefore the overall efficiency of the algorithm increases. Moreover, since we know that the solution of the MCNF problem will be an integer point, it can be expected that the relaxed optimal solution will be in the close neighborhood of the integer optimum. It means that when the original feasibility bounds are restored after "relaxed optimum" has been achieved, very few if any additional iterations will have to be made to obtain integer optimum.

It is clear that the benefits of this procedure are only hopes based on some heuristic reasoning. At the same time, its weak points are evident. While with the traditional feasible pivot rule we can minimize the size of the "smaller" subtree that is to be updated at every change of the basis, with the new method this will not always be possible. Another point of degradation is the necessary use of floating point arithmetic for the solution vector instead of the more efficient integer arithmetic. These arguments suggest that the tradeoff between advantages and disadvantages of this procedure may be problem dependent. Problem family TRGEN analyzed in section 4 proved to belong to the profitable category.

MINET is written in portable FORTRAN–77. It was developed on an IBM PC compatible computer in DOS environment. This fact influenced the design of the data structure. Uploading MINET into a UNIX environment was formally straightforward. However, since it was done mechanically, some inherited structural weaknesses are present in MINET/UNIX. While much attention was paid to rectifying the sizes of integer arrays and variables (which is an issue of consideration on PCs to save memory and speed up computations), the basic data structure was not redesigned. MINET is an in–core program. The incidence matrix of the network is stored in a {*from(j), to(j)*} pair of arrays. The basis is maintained as a set of list functions outlined above.

The computational testing of MINET was performed on a Sun Sparc Station 1+ at RUTCOR, Rutgers University. The computer is linked up with 5 other similar Sun Sparc Stations, and a larger number of terminals have access to any computer in the system. These facts will be important when we discuss the effect of the loading conditions of the system on the running time. The operating system is SunOS 4.1.1 (UNIX), the identifier of the Fortran compiler is $f77$.

3. Experiences with NETGEN problems

NETGEN is a widely accepted problem generator that has been available to the scientific community since 1974 [**5**]. It is able to generate different types of network problems, among them MCNF instances.

Two series of experiments with NETGEN were requested by [**11**]. One of them was with a newly designed set of generation parameters, the other one was with eight selected MCNF problems of the original set of 40 problems in [**5**].

3.1. New NETGEN problems.

The description how to carry out the first category of computational tests for the DIMACS Challenge with 7 classes of MCNF problems can be found in [**11**], Section 7, Minimum Cost Flows.

Unfortunately, the version of NETGEN obtained from DIMACS to generate these 7 classes of problems appeared to have some bugs. Therefore I used a revised version received directly from Cornell University, Ithaca, NY. This instance of NETGEN was able to generate most of the required test problems. In one case, however, I experienced that even this version did not work properly. Namely, in class 5, NETGEN–Lo–Supply the only requested size that was accepted by the generator was 128×1024 resulting in an actual size of 128×1039. Any further attempt with very different, but larger sizes in this class made NETGEN stop at the very beginning of the run with a message of *STOP 9000*. This is a certain error message that has many meanings.

I did not carry out testing in class 6 that is apparently identical with class 4 (NETGEN–Hi–Hi $\equiv$ NETGEN–Med–Supply).

In each class, 7 different problem sizes were chosen. NETGEN generated problems identical or very close to the required sizes. The number of nodes was taken to be powers of 2, as requested.

In Table 1, times in each line were obtained as average solution times (in seconds) of 10 problems generated with different random seeds. Standard deviations are also provided. Solution times do not include input. Running times were observed at low load conditions (Saturday night) at RUTCOR, Rutgers University. The problem classes are identified in the same way as in [**11**], like NETGEN–Lo–Lo, etc. MINET was compiled by the $f77$ compiler in two different ways: without and with code optimization ($-O4$ option). The table contains solution times with both versions. Index 1 refers to data obtained without, while index 2 with code optimization. Column "Ratio of Times" shows the ratio of the performance of the optimized and the non–optimized code.

Problem	Nodes (n)	Edges (m)	Average time–1	Standard dev–1	Average time–2	Standard dev–2	Ratio of Times
NETGEN–Lo–Lo: sources=n/8, sinks=n/8							
N11	128	1024	0.24	0.03	0.15	0.02	0.625
N12	256	2048	0.74	0.06	0.44	0.04	0.595
N13	512	4096	2.50	0.14	1.51	0.08	0.604
N14	1024	8196	8.99	0.40	5.28	0.24	0.587
N15	2048	16388	33.18	1.47	20.50	0.90	0.618
N16	4096	32807	132.58	4.17	84.15	2.67	0.635
N17	8192	65627	536.94	12.76	335.39	7.80	0.625
NETGEN–Lo–Hi: sources=n/8, sinks=n/2							
N21	128	1024	0.28	0.04	0.16	0.02	0.571
N22	256	2068	0.89	0.05	0.49	0.03	0.551
N23	512	4123	3.10	0.22	1.73	0.12	0.558
N24	1024	8259	11.23	0.68	6.31	0.37	0.562
N25	2048	16541	43.31	1.69	26.14	0.95	0.604
N26	4096	33065	170.57	4.19	108.65	2.65	0.637
N27	8192	66155	681.42	7.40	424.28	4.44	0.623
NETGEN–Hi–Lo: sources=n/2, sinks=n/8							
N31	128	1024	0.31	0.03	0.18	0.02	0.581
N32	256	2048	1.01	0.04	0.56	0.02	0.554
N33	512	4096	3.40	0.15	1.91	0.07	0.562
N34	1024	8192	12.41	0.35	6.98	0.20	0.562
N35	2048	16384	49.57	1.02	28.54	0.60	0.576
N36	4096	32768	188.42	3.20	119.92	2.02	0.636
N37	8192	65536	736.44	11.65	460.11	7.07	0.625
NETGEN–Hi–Hi: sources=n/2, sinks=n/2							
N41	128	1025	0.39	0.03	0.22	0.02	0.564
N42	256	2048	1.30	0.07	0.73	0.03	0.562
N43	512	4124	4.62	0.22	2.55	0.12	0.552
N44	1024	8262	16.53	0.43	9.19	0.23	0.556
N45	2048	16524	62.18	2.43	37.48	1.43	0.603
N46	4096	33021	243.64	4.61	153.11	2.97	0.628
N47	8192	66119	902.75	11.46	556.43	7.84	0.616
NETGEN–Lo–Supply: supply=100							
N51	128	1039	0.31	0.04	0.17	0.03	0.548
NETGEN could not generate larger problems in this class							
NETGEN–Med–Supply: supply=10^5 $\equiv$ NETGEN–Hi–Hi							
NETGEN–Hi–Supply: supply=10^8							
N71	128	1025	0.39	0.02	0.22	0.01	0.564
N72	256	2048	1.31	0.09	0.73	0.05	0.555
N73	512	4124	4.63	0.19	2.52	0.11	0.579
N74	1024	8262	17.40	0.45	9.55	0.24	0.533
N75	2048	16524	72.25	2.72	40.90	1.58	0.581
N76	4096	33018	291.25	5.69	183.94	3.63	0.638
N77	8192	66112	1244.80	22.39	810.69	13.52	0.604

TABLE 1.

The above data provide good basis for statistical analysis of solution times of MINET as a function of problem characteristics. Among several possible models, we found that the simple linear regression model described solution times very accurately if it was defined as follows:

$$T = a + bx, \tag{3.1}$$

where x is the problem size expressed as mn, i.e., the size of the incidence matrix. Coefficients a and b vary by problem class.

During the regression analysis the unit of the problem size for all 7 observations in each class was defined as $mn \times 10^{-6}$ which resulted in x_i values in the range of 0.13 to 541.9 such that $x_i \approx 4x_{i-1}$ (for $i = 2, \ldots, 7$).

For the 5 different classes we determined the correlation coefficients, standard errors, and 95 % confidence intervals of b for the "Average time–2" values. They are summarized in Table 2.

Problem	Correlation coefficient (r)	Standard error of the estimate	Confidence interval for b
NETGEN–Lo–Lo:	0.999998	0.260936	(0.622452, 0.625220)
NETGEN–Lo–Hi:	0.999978	1.060852	(0.777804, 0.788968)
NETGEN–Hi–Lo:	0.999940	2.035211	(0.846901, 0.868520)
NETGEN–Hi–Hi:	0.999686	5.689803	(0.997979, 1.057886)
NETGEN–Hi–Supply:	0.999713	52.514020	(1.222111, 1.775359)

TABLE 2.

The confidence intervals for b are included to give a better characterization of the goodness of fit. For the first four classes these intervals are narrow, while class 5 has a relatively wider confidence interval.

The above data show that the performance of MINET can well be estimated by the (3.1) model within the experimental range of $0.13 \leq x \leq 542$.

It is instructive to investigate the predicting power of this model. For this purpose we did the following. We applied the (3.1) model for only the first 5 observations and used the resulting regression line for predicting the remaining 2 data points. The findings are presented in Table 3.

The relatively small sample (number of observations is equal to 5 in each class there) appears to support the applicability of the (3.1) linear model. Consequently, we can expect that the regression model based on 7 observations will behave similarly beyond the experimental region, therefore, it can be used for at least short range predictions. And this is approximately the same what other forecasting models can promise.

Problem	i	x_i	Observed T_i	Predicted $\hat{T}_i$
Netgen–Lo–Lo	6	134	84.15	81.68
	7	538	335.39	326.35
Netgen–Lo–Hi	6	135	108.65	104.23
	7	542	424.28	417.02
Netgen–Hi–Lo	6	134	119.92	113.83
	7	537	460.11	455.13
Netgen–Hi–Hi	6	135	153.11	151.81
	7	542	556.43	607.84
Netgen–Hi–Supply	6	135	183.94	165.95
	7	542	810.69	665.02

TABLE 3.

It is important to note that the validity of the model and results are restricted to the present pricing strategy of MINET on the NETGEN problems.

3.2. Old NETGEN problems.

The second group of NETGEN experiments was carried out with 8 problems from the NETGEN test set of [**5**] selected by DIMACS. In Table 4 they are identified by their original sequential number. Running times were obtained at low load conditions of the computer system. The table contains solution times with two versions of MINET. Index 1 refers to data obtained without, while index 2 with code optimization. Times are given in seconds. Column "Time Ratio" shows the ratio of the performance of the optimized code to the non–optimized one.

Problem	Nodes	Edges	Iterations	Time–1	Time–2	Time Ratio
11	400	1500	5095	2.37	1.36	0.574
20	400	1416	1468	0.74	0.37	0.500
27	400	2676	2323	0.99	0.54	0.545
28	1000	2900	4503	3.32	1.85	0.557
32	1500	4342	6594	6.07	3.41	0.562
36	8000	15000	34461	118.48	76.78	0.648
38	3000	35000	54105	75.73	45.93	0.606
40	3000	23000	41595	48.46	33.86	0.699

TABLE 4.

This set of NETGEN problems consists of structurally different MCNF problems. Therefore, the (3.1) regression analysis is not applicable here since (3.1) was designed to characterize the behavior of MINET on a structurally similar set of problems.

In Tables 1 and 4 we included a column (Ratio of Times) that shows the ratio of solution times with and without code optimization. This serves to evaluate the gain that can be obtained by using the $-O4$ code optimizing option of the $f77$ compiler. The mean value of this figure is 58.8% with a small standard deviation. It means that the gain in solution time that can be obtained by code optimization is 41.2% which is very considerable.

4. Experiences with TRGEN problems

4.1. The TRGEN real–life problems.

As a second category of testing, I generated some real–life problems based on the trailer subproblems for "Traffic scheduling via Benders decomposition" [**6**]. These problems are based on 4 parameters, say p_1, p_2, p_3, p_4. The size of a generated problem is:

\# of nodes : $2(p_1 + p_2)$,

\# of edges : $(p_1 + p_2)^2 + p_2$.

p_3 and p_4 play role in generating the cost coefficients.

I submitted the FORTRAN version of this generator to DIMACS to make it available for other interested researchers.

The test results with MINET (optimized code, low loading conditions) can be found in Table 5 which contains two additional columns: "Degenerate iterations" and "Scanned variables". The term "Scanned variables" refers to the total number of scalar products computed in search for an incoming variable throughout the algorithm. In column "Problem" the four generation parameters are also given for reproducibility.

Here again, we can investigate if there is a significant linear relationship between problem size (measured in mn) and running time. A regression analysis based on the linear model of (3.1) for the above data gives a correlation coefficient $r = 0.950881$, while the 95% confidence interval for b is $0.102864 \leq b \leq 0.209920$. This relatively wide interval is caused by the seemingly irregular behavior of TR8. If we leave it out from the investigations then values of $r = 0.997614$ and $0.141881 \leq b \leq 0.169919$ are obtained which indicates the existence of a close linear relationship.

Table 5 suggests an additional interesting investigation. We can test whether there is a significant linear relationship between solution time (T) and the number of scanned variables (x) measured in units of 10^6, i.e.,

$$T = a + bx \tag{4.1}$$

Problem & Gen. params	Nodes	Edges	Iterations	Degenerate iterations	Scanned variables	Sol. time seconds
TR4 50,150,200,48	400	40150	16171	15797	325348	3.33
TR5 75,150,200,48	450	50775	19633	19209	462524	4.31
TR6 100,150,200,48	500	62650	22488	22014	588475	5.32
TR7 50,250,300,48	600	90250	38484	37867	735998	8.57
TR8 100,250,300,168	700	122750	58949	58372	4119085	25.18
TR9 150,250,400,168	800	160250	68029	67305	3385990	22.71
TR10 200,300,600,168	1000	250300	101709	100871	6026781	39.07

TABLE 5.

For this model we obtain $r = 0.996183$ and $5.400 \leq b \leq 6.609$. This means that under the present setting of strategic parameters of MINET, solution time is a linear function of the number of scanned variables.

Similarly, we can compute the Spearman rank correlation coefficient for these data. Doing so, we obtain a value of $r_s = 1.00$ that suggests a perfect positive relationship between the two variables.

Unfortunately, values based on model (4.1) that cannot be computed in advance, therefore this model cannot be used to estimate solution time. However, this information can be useful for researchers who study the behavior of algorithms with the purpose of trying to improve the performance of them.

4.2. Effects of an anti–degeneracy row selection strategy.

The very large number of degenerate iterations (up to 99 % of the total number of iterations) that appears in Table 5 suggests that solution times could be reduced if some "cheap" anti–degeneracy strategy could be found. Though in the framework of simplex based MCNF algorithms degenerate steps cannot completely be avoided (for detailed analysis see [**3**]), there seems to be a chance for achieving some improvement. For the TRGEN problems we tried the anti–degeneracy row selection Version–1 described in [**10**] and obtained the results displayed in Table 6 (MINET, optimized code). Row "Old" presents data with the feasible (traditional) row selection, row "New" with Version–1. In a "New" row a degenerate iteration indicates the number of minimum forced steps. If any small step larger than this was made, it was not counted degenerate.

Problem	Method	Iterations in Phase–I	Deg. itns in Phase–I	Iterations (Total)	Deg. itns (Total)	Sol. time in *sec*
TR4	Old	12798	12598	16171	15797	3.33
	New	399	0	3675	0	4.04
TR5	Old	14398	14173	19633	19209	4.31
	New	449	0	3601	0	3.37
TR6	Old	16623	16373	22488	22014	5.32
	New	499	0	4113	0	4.81
TR7	Old	32948	32648	38484	37867	8.57
	New	599	0	3952	0	6.97
TR8	Old	36773	36423	58949	58372	25.18
	New	699	0	6521	0	11.66
TR9	Old	43098	42698	68029	67305	22.76
	New	799	0	8899	2	15.16
TR10	Old	65748	65248	101709	100871	39.07
	New	999	0	12664	4	25.59

TABLE 6.

The improvement in the number of iterations seems to be considerable, while the solution times do not show a proportional decrease. This is partly due to the rough experimental implementation of the simplest version of the new row selection procedure, and partly to the fact that with this procedure we usually get a much larger "smaller subtree" that is to be updated at every change of the basis, as outlined in section 2. This latter can be improved by some refinements and also by the implementation of versions 2 and 3 proposed in [**10**]. However, it is inevitable that the computational cost of an iteration will be higher than with the traditional row selection technique. This is the price to be paid for the reduction in the number of iterations. The tradeoff may be problem dependent.

5. GTE–instances

The third category of our experimentation was a set of recently discovered real–life problems identified as GTE–instances [**12**]. They have been obtained from a multi–commodity model. The problems are small (49 nodes, 520 edges) and differ from one another in the capacities and demands.

The remarkable feature of these problems is that they are apparently "bad" for Bertsekas–Tseng's RELAXT–III algorithm making it to produce a large variation in solution times. This is why this family of problems received the `bad` suffix. The range of the running times is 0.020 *sec* through 508.829 *sec*. Clearly, this is a challenge for the simplex based network solvers to see how efficiently and with what variance in running time they can solve this family of problems.

MINET showed a very stable performance on the `gte_bad` problems, as it can be seen in Table 7. Solution times never exceeded 0.06 *sec*, which is still in the magnitude of the resolution of the clock in the test environment of MINET. The extension of problem names after the period gives the solution time in milliseconds obtained by RELAXT–III. The last column contains times obtained by MINET (optimized code).

Problem	Iterations	Degenerate iterations	Scanned variables	Sol. time in *sec*
gte.20	121	117	2741	0.04
gte.60	128	44	2461	0.03
gte.200	154	48	2465	0.03
gte.510	226	116	3943	0.04
gte.1160	248	138	3571	0.06
gte.6830	211	108	3301	0.04
gte.15100	139	87	2609	0.03
gte.15710	163	71	2869	0.04
gte.35620	192	47	3129	0.04
gte.60090	290	147	4229	0.06
gte.298300	135	39	2554	0.03
gte.451760	178	68	3721	0.05
gte.469010	131	41	2852	0.05
gte.508829	127	38	3041	0.03

TABLE 7.

6. Comparison of MINET and NETFLO

NETFLO is an industry standard MCNF solver developed by R. V. Helgason and J. L. Kennington. The FORTRAN version of NETFLO has been made available for the participants of the DIMACS Challenge at DIMACS `/netflow/mincost/solver-1` together with some hints how to use it for timing. (Actually, this is the only MCNF solver available there.) This version of NETFLO is dated February 1988.

To enable the comparison, one little change had to be made in NETFLO. Since NETFLO makes some preprocessing during input, it was necessary to include it in timing. MINET does not generate any detailed solution output after the optimality condition is fulfilled. To ensure comparability, we changed NETFLO to stop after optimality condition is detected, included input for timing of MINET, and used the run time system *logtime* provided in the above referenced subdirectory of DIMACS for timing. Software tool *logtime* separates *user time* and *system time* and gives them in seconds with an accuracy of one decimal for a program run under its supervision.

For the comparison, we used seven of the eight selected NETGEN problems. The smallest of them (#11) is an assignment problem that could not be converted for NETFLO by the supplied conversion routine, therefore it was not considered here.

In the other rounds of comparisons we used problem classes from the final DIMACS Core problems [**12**]. First, problems generated by the `GGRAPH` generator, producing grid graphs, were used, then several different classes generated by the `GOTO` generator, producing capacitated transportation problems on a Grid–On–TOrus network, were solved. The GOTO problems are acknowledgedly hard.

For the tests, both MINET and NETFLO were compiled with the recommended $-O$ optimization feature of the $f77$ compiler.

Results with the selected NETGEN problems are summarized in Table 8.

			MINET		NETFLO	
Problem	Nodes	Edges	Iterations	User time	Iterations	User time
20	400	1426	1468	1.9	1338	2.6
27	400	2676	2323	3.3	1900	5.3
28	1000	2900	4503	5.0	3565	7.7
32	1500	4342	6594	8.1	5704	15.2
36	8000	15000	34461	85.3	31246	186.0
38	3000	35000	54105	79.7	25077	333.7
40	3000	23000	41595	54.6	23267	195.7

TABLE 8.

There are a few things in Table 8 that deserve attention. First, the two codes give different sequences of problems if sorted by solution time. Second, NETFLO tends to make considerably less iterations than MINET. Third, solution times of MINET get significantly smaller than that of NETFLO as problem size increases.

The GGRAPH problems can be generated by 5 parameters: `X`: Grid height, `Y`: Grid width, `MAXCAP`: Capacities are uniform on [0, `MAXCAP`], `MAXCOST`: Costs are uniform on [0, `MAXCOST`], `SEED`: Seed of the random number generator.

For all of the GGRAPH problems the values `MAXCAP`$= 10^4$ and `MAXCOST`$= 10^4$ were fixed. In Table 9, under problem generation parameters, we give the values for `X`, `Y`, and `SEED`.

The last two GGARPH problems in Table 9 differ only in the value of the `SEED`. For this set we ignore regression analysis because the problems are structurally inhomogeneous. Nevertheless, it is interesting to observe that both solvers show a smooth performance in the sense that solution times increase smoothly with problem size. NETFLO is always faster by a narrow margin.

Generation parameters	Nodes	Edges	MINET Iterations	MINET User time	NETFLO Iterations	NETFLO User time
32 32 270001	1026	2048	2674	3.5	1869	3.0
16 128 270001	2050	3984	4988	8.8	2282	5.9
128 16 270001	2050	4208	5762	9.3	6260	8.9
64 64 270003	4098	8192	13645	23.3	10168	18.8
256 16 270001	4098	8432	12290	25.8	13210	22.5
80 80 270004	6402	12800	19539	42.2	21384	40.8
512 16 270001	8194	16680	23815	77.1	33306	75.2
512 16 270011	8194	16680	22811	75.9	34968	74.4

TABLE 9.

The GOTO problems are generated by following 5 parameters: `N`: Number of nodes, `M`: Number of arcs, `MAXCAP`: Maximum capacity (some arcs may be generated with capacities greater than this to ensure feasibility), `MAXCOST`: Maximum arc cost, `SEED`: Seed for random number generator.

We investigated three groups of GOTO problems. Each of them contained structurally similar problems on which both codes obeyed the (3.1) model. Therefore, we could use regression analysis for comparison. It turned out that the computed values of the correlation coefficients for MINET and NETFLO in the groups were quite the same that established a basis for comparison of the regression lines.

The first group of GOTO problems is referenced as Grid–Density–8, (GD–8) and has the feature that $M = 8N$. For the generated test problems the last three parameters, in all cases, were fixed: `MAXCAP`=16384, `MAXCOST`=4096, and `SEED`=270001. Results of the test are summarized in Table 10.

Nodes	Edges	MINET Iterations	MINET User time	NETFLO Iterations	NETFLO User time
128	1024	1409	1.6	2138	1.8
256	2048	3188	3.8	5450	4.7
512	4096	6860	11.8	16496	16.5
1024	8192	11674	41.9	30003	54.8
2048	16384	28759	108.1	67287	182.7
4096	32768	75322	329.3	342391	905.9

TABLE 10.

The second group of GOTO problems is called Grid–Density–16 (GD–16),

and is characterized by the relation $M = 16N$. The value of the last three parameters, in all cases, were fixed in the same way as in Grid–Density–16. Results of the test are presented in Table 11.

		MINET		NETFLO	
Nodes	Edges	Iterations	User time	Iterations	User time
128	2048	2858	3.2	5750	4.5
256	4096	6812	8.4	14265	11.0
512	8192	14948	24.2	31305	30.4
1024	16384	29753	77.0	72282	109.3
2048	32768	70538	239.4	172723	408.3
4096	65536	143122	1042.4	460458	1759.5

TABLE 11.

The third group of GOTO problems is referenced as Grid–Increasing–Density (GDI), and has the feature that $M = \lceil N^{3/2} \rceil$. The rest of the parameters is as above. Results of the test can be found in Table 12.

		MINET		NETFLO	
Nodes	Edges	Iterations	User time	Iterations	User time
128	1449	2040	2.2	2725	2.5
256	4096	6812	8.4	14265	11.0
512	11586	19490	37.0	48493	50.7
1024	32768	60980	166.1	190027	297.4
2048	92682	199899	832.7	680137	1953.9

TABLE 12.

¿From Tables 10–11–12 we can compute linear regression data for the three GOTO groups by (3.1). The obtained values of r and b are summarized in Table 13.

Group	Parameter	MINET	NETFLO
GD–8	r	0.996360	0.998699
	b	2.406176	6.731219
GD–16	r	0.999618	0.999803
	b	3.856446	6.540692
GDI	r	0.999784	0.999698
	b	4.352572	10.330800

TABLE 13.

Based on the near equality of the correlation coefficients (r) of MINET and

NETFLO in Table 13, we can compare the slopes (b). This suggests that on the GOTO problems MINET is 2 to 3 times faster than NETFLO.

7. Running times versus loading conditions of the computer

The most widely accepted measure of performance of optimization algorithms is running time. Other measures fail to express the total computational effort needed to solve a problem. One such example is the number of iterations which can heavily be influenced by the work per iteration, therefore, the meaning of an iteration can be different from algorithm to algorithm.

The running time of a program was believed to be proportional to the real computational effort required to solve a problem and therefore suitable for comparing the performance of different algorithms. Standards have been set up for reporting computational experiences (c.f., [**1**] or [**4**]).

Timing on PCs is relatively simple and accurate. For comparability of results, it is more or less sufficient to specify the type of the processor and clock rate, together with the *solution time* which is usually equal to the *residence time*.

Workstations, however, run under UNIX operating system that provides multiprocessing, multitasking, and networking. Therefore, in any short time interval, a larger number of programs may run in the computer. This immediately explains why the question of timing accuracy was raised by the organizers of the First DIMACS International Algorithm Implementation Challenge [**11**].

Clearly, under UNIX environment, residence time can be anything depending on the nature of the concurrently running processes, therefore it cannot be used for timing. However, most of the systems offer time measuring procedures that can separate *system time* and *user time*.

We investigated with MINET the following question: How accurately can solution times be reproduced if the same program solves the same problem over and over again?

MINET uses the *etime* routine of the UNIX FORTRAN library. Its first parameter is supposed to measure user time.

First, the performance of *etime* was tested when the Spark 1+ was run in a stand alone mode. The problem to be solved was chosen from the GGRAPH family and identified as gg2 with 2027 nodes and 4050 arcs. MINET solved gg2 three consecutive times in 7.82, 7.73, and 7.79 seconds.

When two identical copies (say, A and B) of MINET were loaded and both worked on gg2 concurrently, the following running times were obtained at tree different attempts:

Run	Copy–A	Copy–B
1	7.80	7.87
2	7.75	7.89
3	7.84	7.88

When the experiment was repeated with three copies of MINET running con-

currently on gg2, solution times provided by *etime* were 7.62, 7.70, and 7.67 seconds.

These figures seem to justify the belief that *etime* really returns the user time of a program, and not residence time or something else.

However, when MINET was tested on the same gg2 problem under everyday multiprogramming, multitasking, and networking circumstances, we observed a rather erratic behavior.

The method of testing was the following. Within a selected time interval MINET was run to solve the same problem repeatedly. The lengths of the intervals were different, and the intervals were selected on different days of the week and different hours of the day. For each interval the mean ($\bar{X}$) of the solution times is computed, and also the $t_{\max}/t_{\min}$ ratio in percents. For all the three problems a "Total" row is also created displaying the total sample mean and the $t_{\max}/t_{\min}$ ratio for the sample. Problem characteristics are summarized below. gg2 and gg3 are from the GGRAPH family, while cap1 was generated by capt, also from [**12**].

Problem	# of nodes	# of arcs
gg2	2027	4050
gg3	4098	8192
cap1	514	33025

The results obtained with gg2, gg3, and cap1 are summarized in Tables 13, 14, and 15, resp.

Time Interval	Observations (t_i)	mean ($\bar{x}$)	$t_{\max}/t_{\min}$ %
2 min	7.91 7.96 7.95	7.940	100.63
5 min	7.59 7.77 7.66 7.62 7.68 7.58	7.650	102.51
5 min	8.26 8.08 8.27 8.07 8.19	8.174	102.48
2 min	9.68 9.82 9.60	9.700	102.29
5 min	10.96 10.95 7.57 7.58 7.88 7.59	8.757	144.78
	Total	8.357	144.78

TABLE 14.

Time Interval	Observations (t_i)	mean ($\bar{x}$)	$t_{\max}/t_{\min}$ %
5 min	25.09 25.61 25.17 25.07 24.03	24.994	106.58
6 min	24.05 24.06 24.91 25.08 24.19	24.458	104.28
8 min	31.43 30.13 25.18 27.13 25.23	27.820	124.82
5 min	34.30 23.98 24.01 24.88	26.793	143.04
	Total	25.975	143.04

TABLE 15.

Time Interval	Observations (t_i)	mean ($\bar{x}$)	$t_{\max}/t_{\min}$ %
3 min	8.14 8.17 8.20 7.85	8.090	104.46
3 min	9.70 9.51 9.46 9.32 9.88	9.574	106.01
3 min	9.89 8.18 8.17 9.20	8.860	121.05
	Total	8.898	125.99

TABLE 16.

The obtained figures do not look very encouraging. Running times seem to vary too much. A bit more careful analysis reveals some important details.

Within the first four rows of `gg2` (Table 14) we can see a uniform behavior of running times. The same uniform figures cannot be observed between rows. The mean values of 7.940, 7.650, 8.174, and 9.700 differ nearly 27%. Then comes row 5 with a very inhomogeneous row performance and globally extreme values that also define the $t_{\max}/t_{\min}$ ratio for the whole sample.

The same can be said if we look at the observed data for `gg3` (Table 15). The only difference is that in this case the variable behavior can be seen in two rows.

Though in the third series of observations `cap1` (Table 16) did not show such a more than 40% deviation, it is still too variable with its 26%.

It is our belief that if a single counterexample can be found then this is sufficient to reject the unconditional reliability of time measuring in UNIX. What we presented here is definitely more than just a single counterexample. Therefore, if we imagine a long test run including the serial solution of hundreds of problems in a blind batch mode then we really cannot be certain about the comparability of such solution times.

Having seen the above instances several questions can be asked. What is the real solution time? How can we compare algorithms on the basis of reported running times? We think that nowadays, when a 25% performance improvement of an algorithm is considered significant, these questions should be addressed correctly.

Our purpose here is to point out the above problems. Answering them may require a considerable effort. At the same time, it appears inevitable that such a study must be conducted.

8. Summary

Based on the quick overview of the above experiences, it can be concluded that MINET shows a relatively stable and more or less predictable performance with the present setting of the available strategic parameters.

In the discussion — among others — we addressed two important issues that may influence the comparability of reported running times and, therefore the

relative efficiency of codes. It turned out that the optimizing capability of a compiler can improve timings by more than 40%. It also became evident that time measuring under UNIX operating system can be rather unreliable giving an at least 40% wide range of solution times. The former observation suggests that information on code optimization must always be provided, while the latter necessitates an in–depth analysis of time measuring in UNIX.

9. Acknowledgments

The author is indebted to the two anonymous referees for their helpful comments.

References

1. Crowder, Dembo, and Mulvey, *Reporting Computational Experiments in Mathematical Programming*, Mathematical Programming **15** (1978), 315–329.
2. Cunningham, W. H., *Theoretical Properties of the Network Simplex Method*, Mathematics of Operations Research **4** (1979), 196–208.
3. Gal, T., Geue, F., *A New Pivoting Rule for Solving Various Degeneracy Problems*, Operations Research Letters **11** (1992), 23–32.
4. Greenberg, H. J., *Computational Testing: Why, How, and How Much*, ORSA Journal on Computing **2** (1990), 94–97.
5. Klingman, D., Napier, A., Stutz, J., *NETGEN—A program for generating large–scale (un)capacitated assignment, transportation and minimum cost flow problems*, Management Science **20** (1974), 814–821.
6. Love, R. R., *Traffic scheduling via Benders decomposition*, Mathematical Programming Study **15** (1981), 102–124.
7. Maros, I., *MINET a Fast Network LP Solver*, IIASA Working Paper WP–87–50, International Institute for Applied Systems Analysis, A–2361 Laxenburg, Austria, 1987.
8. Maros, I., *MINET Fast Network LP Solver, Description and User's Guide for V2.00*, IIASA Working Paper WP–88–6, International Institute for Applied Systems Analysis, A–2361 Laxenburg, Austria, 1988.
9. Maros, I., *A Structure Exploiting Pricing Procedure in the Primal Network Simplex Algorithm,*, RUTCOR Research Report, RRR #18–91, RUTCOR, Rutgers University, 1991.
10. Maros, I., *A Practical Anti–Degeneracy Row Selection Technique in Network Linear Programming*, RUTCOR Research Report, RRR #4–92, 1992.
11. McGeoch, C. C., *The First DIMACS International Algorithm Implementation Challenge: The Core Experiments*, DIMACS, August, 1991.
12. McGeoch, C. C., *The First DIMACS International Algorithm Implementation Challenge: The Benchmark Experiments*, DIMACS, December, 1991.

Computer and Automation Institute, Hungarian Academy of Sciences, H–1518 Budapest, P.O.Box 63., Hungary

Current address: RUTCOR, Rutgers Center for Operations Research, Rutgers University, P.O.Box 5062, New Brunswick, NJ 08903

E-mail address: maros@rutcor.rutgers.edu and h107mar@ella.hu