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

An Empirical Study of Min Cost Flow Algorithms

R. G. BLAND, J. CHERIYAN, D. L. JENSEN AND L. LADÁNYI

ABSTRACT. Extensive computational testing of four minimum cost network flow codes over several distributions of sparse networks is conducted. Regression analyses yield models of mean execution times that predict well over the specified distributions.

1. Introduction

There is considerable practical and theoretical interest in algorithms for network flows. Several new algorithms with attractive theoretical properties have been presented in recent years (see the surveys [**1, 15**]). Little is known about the empirical behavior of the recent algorithms; only a few detailed computational studies have appeared, e.g., [**6, 8, 11**]. This study focuses on the computational behavior of codes implementing four min cost flow algorithms based on cost scaling (see [**8**]), successive approximations (see [**17**]), the network simplex method (see [**18**]) and relaxation (see [**6**]). Our main goal is to model mean execution time as a function of the input parameters over several populations (distributions) of network instances. A second goal, related to the first, is to study the inherent variability of individual execution times. Techniques from regression analysis are applied [**12**].

The analysis of the main experiment concerns estimation of the responses of four fixed codes to variations in certain key problem characteristics over specific input distributions. Since run-times can be sensitive to both variations in the codes and in the population from which inputs are drawn, one should resist broad interpretations of the experiment that reach for conclusions on the relative

1991 *Mathematics Subject Classification.* Primary 90B10; Secondary 68Q20.

The authors were supported in part by AFOSR, NSF and ONR through NSF grant #DMS-8920550.

The second author was supported in part by the Lucille and David Packard Fellowship of Éva Tardos.

efficacy of the four algorithms, as opposed to the fixed codes, or conclusions concerning wider classes of inputs. Indeed, the data offer ample evidence of large variations in the behavior of a fixed code across different populations, and, consequently, of the need to limit interpretations of empirical data to the realm in which the computation was performed.

This study does show that it is possible to deduce accurate models of average computational behavior over an entire (carefully circumscribed) population from tests on a modest-sized sample. Such models provide insights that may be useful in improving codes, and may serve as a first step toward both more robust models and more robust codes.

The populations of networks studied here are each associated with a network generator and a random mechanism for selecting the generator inputs. The generator accepts a list of inputs, and outputs a directed graph together with a cost and a capacity for each arc, and associated demands and supplies for some of the nodes. In our experiments the generator inputs are determined by a smaller set of control variables, some of which coincide with generator inputs (e.g., the seed, the number of nodes); each of the remaining generator inputs is a fixed function of the control variables. Treating the control variables as random variables with a fixed joint distribution implicitly determines a distribution (population) of network instances. We are specifically interested in modeling computational behavior as a function of the control variables over the populations that arise from independent uniform distributions on each control variable. If accurate models can be determined, this will facilitate examination of several secondary issues: comparisons of different codes on fixed input distributions; comparisons of the behavior of a fixed code across different distributions; analysis of the inherent variability of execution time about the mean as a function of the control variables.

Section 2 discusses briefly the four codes (SCALE, MIN3, RNET 3.61 [**18**], and RELAXT-III [**6**]) and the two populations of networks on which the codes were tested in the main experiment. These populations derive from the network generators CAPT [**8**] and NETGEN [**19**]. Section 3 presents the main experiment and statistical analysis. Some additional experimentation on populations from the GRIDGEN [**20**] and GOTO [**14**] generators is outlined in Section 4. Subsequent to the completion of the main experiments we conducted additional ad hoc runs trying different versions of SCALE, MIN3 and RNET 3.61; Section 5 reports on these runs, which offer indications of how some of the codes can be speeded up on some classes of inputs. Section 5 also comments on ad hoc runs on networks of larger size or different density from those considered in the main experiment. Brief conclusions appear in Section 6.

Additional data concerning these experiments are stored in the directory pub/netflow/instances/mincost/bcjl at dimacs.rutgers.edu; the file README in that directory has additional information. Those who are interested in reproducing parts of the experiments should contact the first author for access to executable SCALE and MIN3 codes for the SUN Sparc-2 workstation.

2. The Generators and The Algorithms

The main experiments used two generators to construct instances of min-cost flow problems: the NETGEN generator of Klingman, Napier and Stutz [**19**]; and a new version of the CAPT generator of Bland and Jensen [**8**]. (See Section 4 for a brief summary of additional experiments based on the GRIDGEN generator of Lee and Orlin [**20**] and the GOTO generator of Goldberg [**14**].) Each generator accepts a list of parameter values as inputs, and outputs a directed graph together with a cost and a capacity for each arc, and associated demands and supplies for some of the nodes.

The version of NETGEN we used (supplied by DIMACS and dated 9/6/90) has fifteen input parameters. Of these, the following are of interest here: *seed*, an integer used by the pseudo random number generator; *nodes*, the number of nodes; *arcs*, the (approximate) number of arcs; *nsorc*, the number of sources; *nsink*, the number of sinks; *supply*, the amount of flow to be sent from the sources to the sinks; a range on the arc costs, *locost* ... *hicost*; a range on the arc capacities, *locap* ... *hicap*; and the percentage of capacitated arcs, *ipcap*. (Our networks had no transshipment sources or sinks, and all of the "skeleton" arcs were specified to have the maximum cost.)

In order to ensure that the network will be feasible, i.e., the given node demands and supplies and arc capacities will not be inconsistent, NETGEN first constructs a "skeleton" which is an extremely sparse subnetwork. The skeleton arcs are capacitated in such a way that the specified amount of flow may be sent from the sources to the sinks using only these arcs. After generating the skeleton, NETGEN throws in additional "random" arcs until the total number of arcs roughly equals the requested number, by visiting each non-sink node and randomly adding arcs leaving that node; no duplicate arcs are created. (It was necessary to correct one line of the NETGEN code in order to allow additional arcs entering the highest-numbered sink node.)

In the NETGEN networks of the main experiment the number of nodes was in the range 2^9 to 2^{13}, the number of arcs was in the range $2 \cdot nodes$ to $32 \cdot nodes$, the minimum arc cost, *locost*, was equal to zero, the maximum arc cost, *hicost*, was in the range 2^8 to 2^{17}, the number of sources was in the range $\frac{nodes}{2^7}$ to $\frac{nodes}{2}$, *nsink* was equal to *nsorc*, and the amount of flow, *supply*, was in the range $2^6 \cdot nsorc$ to $2^{13} \cdot nsorc$. The percentage of capacitated arcs was either zero (for about one-seventh of all the networks) or 100. For the capacitated networks, the maximum arc capacity, *hicap*, was taken to be $k \cdot \frac{supply}{nsorc} \cdot \frac{nodes}{arcs}$, where k was one of the constants 1, 2, 3, 4, 5 or 6, and for the uncapacitated networks *hicap* was set to *supply*. We chose these values of k after preliminary tests in which we computed an optimal min-cost flow and counted the number of saturated arcs (i.e., arcs whose flow was at the upper bound); for the values of k above, between 5% and 39% of the arcs were saturated in our tests. Our motivation was that if the percentage is too high, then the skeleton arcs may play a dominant role, absorbing most of the flow; if the percentage is too low, then the network becomes effectively uncapacitated. For example, in the "benchmark" NETGEN networks, even those that are capacitated have very few saturated arcs in the optimal solutions:

benchmark problem no.	capacitated arcs in the network	saturated arcs in the optimal solution
16–19	20%	0.05%
20–23	40%	0.09%
24–27	80%	0.66%

We used a version of CAPT (available from DIMACS) with eleven input parameters, of which the following are of interest here: *seed*, a real number between zero and one used by the pseudo random number generator; *nsorc*, the number of sources; *nsink*, the number of sinks; *arcs*, the (approximate) number of arcs; *avgsup*, the average supply at each source; *intlen*, which controls how tightly capacitated the network will be; and *bcost*, the maximum bit-length for the arc costs. Arc costs are independent and uniform on the integers between 0 and $2^{bcost} - 1$.

CAPT generates only transportation problems. A network generated by CAPT essentially consists of a bipartite graph, one of whose node partitions is the set of sources while the other is the set of sinks. The bipartite graph is chosen uniformly from the set of $nsorc \times nsink$ bipartite graphs having the requested number of arcs; no duplicate arcs are present. CAPT next constructs what will be a feasible flow. Consider the unit simplex in the Euclidean space whose dimension is equal to the number of arcs in the bipartite graph, (i.e., each arc has a corresponding coordinate). CAPT creates a preliminary flow by choosing a point uniformly in the unit simplex, and using the coordinates of the point to allocate the proportion of the total flow ($nsorc \cdot avgsup$) on each arc. The preliminary supply or demand at a source or sink node is given by the sum of the preliminary flows of the bipartite graph arcs incident with it. The capacity on each arc is set to the preliminary flow value on the arc plus an additive perturbation chosen uniformly from zero to *intlen*. Supplies (demands) are obtained by perturbing the preliminary values up (down) similarly.

The ranges for the parameters in our experiments with CAPT were similar to the NETGEN parameters, except that *nsorc* and *nsink* were each equal to half the total number of nodes. Roughly four-fifths of the networks were capacitated, and the maximum additive perturbation, *intlen*, was taken to be $k \cdot avgsup \cdot \frac{nodes}{arcs}$, where k was one of the constants 1.5, 2.5, 3.5, or 4.5. For the remaining one-fifth of the networks *intlen* was taken to be $10 \cdot avgsup$, and these networks were regarded to be uncapacitated.

Four codes were tested in our computational experiments: SCALE, MIN3, RNET 3.61 and RELAXT-III. The code SCALE was implemented by us; it is based on a cost scaling algorithm of Bland and Jensen (see [**8**]), which is related to earlier work by Bland and Jensen [**7**] and by Röck [**23**]. We implemented the code MIN3 mostly following the successive approximation algorithm of Goldberg and Tarjan [**17**], which combines and extends: the notion of cost scaling due, independently, to Bland and Jensen [**7**] and Roc̆k [**23**], and motivated by Edmonds and Karp [**13**]; the push-relabel approach to maximum flow due to Goldberg and Tarjan (see [**16**]); and the relaxation method of Bertsekas (see [**4**]), which is rooted in Bertsekas's auction algorithm for the assignment problem [**3**]. Tardos's use of scaling and approximation to achieve the first strongly polynomial min cost flow algorithm [**24**] also motivated [**17**]. The interested reader is referred to [**17, 5, 15, 1**] and their references for comments on the development of these min cost flow algorithms and their antecedents. RNET (version 3.61–12/79) of Grigoriadis and Hsu [**18**] is a subroutine library to implement the primal network simplex algorithm (see [**10**]). We denote by RNET 3.61 the network simplex code used in the main experiments in which our driver fixes the RNET 3.61 parameters as follows: *frq*=0.5 (default=1.0, ... ,8.0), p_0=200 (default=1), and p_1=1.5 (default). RNET 3.61's execution times can be quite sensitive to these parameters, especially *frq*. The code RELAXT-III (April 1990 – no version number) of Bertsekas and Tseng is based on their relaxation algorithm [**6**]. This code has no user-settable parameters. (Henceforth, we refer to this code as RELAX.)

Bland and Jensen (see [**8**]) presented a cost-scaling algorithm for the minimum cost flow problem in which the computational engine is a maximum flow routine. After some preliminary comparisons among implementations of several maximum-flow algorithms, we chose the algorithm of [**16**] that uses the FIFO policy and does periodic computations of exact distance labels (but does not use the dynamic trees data structure). On several families of networks, this max-flow implementation runs as fast or faster than other max-flow implementations that we have tested; for example, even on "long and thin" (e.g., $10,000 \times 10$) networks the execution time of this max-flow implementation compares favorably with that of the code of [**2**]. An earlier version of SCALE implemented by Bland and Jensen [**8**] used the MPM maximum flow algorithm [**21**]. Perhaps because of the MPM-based maximum flow routine, the earlier version of SCALE performed poorly on sparse networks, and even on dense networks the MPM routine was problematic, since the maximum flow subproblems encountered in SCALE become very sparse as the iterations progress.

We have implemented the successive approximation algorithm of [**17**] with several parameters that specify different variants of the algorithm. The main variant described in [**17**] maintains a parameter ϵ which is iteratively decreased by a scaling factor of two; each iteration executes the *refine* procedure during which the current solution, which is guaranteed to be ϵ-optimal, is manipulated to yield a solution which is (ϵ/fac)-optimal ($fac = 2$ is the scaling factor). When ϵ decreases from its initial value by a factor of nC, where n is the number of nodes and C is the maximum absolute cost, then the algorithm terminates with an optimal solution. The basic step in the refine procedure is to select any *active* node and apply a *discharge* step to it. Several different policies for selecting the node to discharge have been proposed and analyzed (in the worst case) e.g., queue (FIFO), wave, and first active [**17**]. We have implemented five policies, namely, stack, queue, q-max (a hybrid of queue and stack), wave, and first-active. We found the stack and q-max policies to be the fastest among these five policies on all networks in our preliminary tests. MIN3 uses the stack policy.

As suggested by Goldberg and Tarjan, one variant of the implemented algorithm periodically executes Bellman-Ford computations to find an ϵ within a factor of two of minimum such that the current solution is ϵ-optimal; when possible, the Bellman-Ford computation also updates the node prices (i.e., the dual solution). We have tried several heuristics for invoking the Bellman-Ford computation, including one variant that invokes it after every i refines ($i = 1, 2, 4, \ldots$), and another one that attempts to balance the overall number of operations executed by the refine computation and the Bellman-Ford computation.

On most networks we have tested, the successive approximation algorithm finds an optimal flow after executing significantly fewer refines than the worst-case bound of $\log nC$. Unfortunately, the algorithm has no direct way to check when an optimal solution has been found; this is because the node prices are only guaranteed to be ϵ-optimal with respect to the current flow. This raises the issue of detecting an optimal flow and terminating the algorithm. In each refine, a quick estimate of the total cost of the current flow is computed using 32-bit arithmetic for the computation. When two consecutive refines have the same value for the estimated total cost, we may have an optimal flow, so a Bellman-Ford computation is run to find zero-optimal node prices; if the Bellman-Ford computation succeeds, then we have an optimal flow. Also, a successful Bellman-Ford computation is rather fast. If the Bellman-Ford computation is invoked before optimality, then it runs slower and returns no useful information. MIN3 employs the variant above, which worked well on all networks tested, running faster than both (1) the variant that executes only refines (but no Bellman-Ford computations), and (2) the variant of [**17**] that periodically executes Bellman-Ford computations.

At the conclusion of each refine, MIN3 fixes the flow at its current value on all arcs on which the absolute value of the current reduced cost exceeds $n\epsilon$, thereby reducing the effective number of arcs in the remaining computation.

We considered the use of several scaling factors in our preliminary tests, and found that a factor of four usually gave the fastest execution times, and almost always gave faster execution times than a factor of two. The reported times for MIN3 use a scaling factor of four.

In addition to the preliminary testing of SCALE and MIN3 described above, some preliminary tests were performed running RNET 3.61 on NETGEN networks to determine the settings for the parameters *frq* and p_0. All tuning of the SCALE and MIN3 codes and the choice of the parameter settings for RNET 3.61 were fixed prior to the design of the main experiments, which took place prior to the DIMACS Workshop in October 1991.

The compilers used are: Gnu C (for MIN3), and SUN Fortran (for SCALE, RELAXT and RNET 3.61), all with the standard optimization level, "-O" (i.e., "big oh"). The execution times reported are the "elapsed user times" in seconds on a SUN Sparc-2 with 16 megabytes of memory, measured to an accuracy of $\frac{1}{60}$ seconds; the times for input and initialization are never included.

3. The Main Experiments

The primary goal of the experimental analysis is to model mean execution time as a function of problem characteristics for each of the several codes and input distributions discussed previously. The problem characteristics, or *control variables*, identified for study here are: n (the number of nodes); m (the number of arcs); C (the maximum absolute value of any cost); f (the total flow through the network); s (the number of source nodes); and d (a measure of tightness of the capacity constraints on the arcs – for CAPT, d denotes *intlen*, while for the other generators it denotes *hicap*.

3.1. Experiment design. Based on earlier work on statistical analysis of network flow algorithms [**8**], models of execution time of the form:

$$T = \varepsilon e^{\beta_0} n^{\beta_1} m^{\beta_2} (\ln C)^{\beta_3} f^{\beta_4} d^{\beta_5} s^{\beta_6} \tag{3.1}$$

seemed a plausible first choice. Here $\beta_0, \ldots, \beta_6$ are the parameters of the model to be estimated by regression analysis, e is the base of the natural logarithm, and the value of β_0 reflects the units of measurement of the dependent random variable T, the execution time, so that the random variable ε has mean one. Both T and ε must be regarded to be functions $T(n, m, \ln C, f, d, s)$ and $\varepsilon(n, m, \ln C, f, d, s)$ of the control variables. An important part of the analysis to follow is the question of whether the random variables $\varepsilon(n, m, \ln C, f, d, s)$ are independent and identically distributed. We will address an even more stringent *lognormality assumption*:

$$\varepsilon(n, m, \ln C, f, d, s) \text{ are i. i. d. lognormal with mean 1.} \tag{3.2}$$

For the CAPT distribution examined here s is always $n/2$, and so s is dropped from the model (3.1). It is natural to use $\ln C$ rather than C in the model, since the number of iterations executed by SCALE is linear in $\ln C$ for all networks, and MIN3 has similar behavior. Section 4 has more discussion on the model, including consideration of different choices of some of the control variables and the issue of how to measure d for uncapacitated networks.

We did not expect a model as simple as (3.1) to be sufficiently robust to portray accurately algorithmic behavior over the entire class of networks constructible by any one of the generators. Even an unabashed optimist could not reasonably hope for an accurate model much simpler than a sum of terms each of the form (3.1):

$$T = \sum_{i=1}^{k} \varepsilon_i e^{\beta_{0,i}} n^{\beta_{1,i}} m^{\beta_{2,i}} (\ln C)^{\beta_{3,i}} f^{\beta_{4,i}} d^{\beta_{5,i}} s^{\beta_{6,i}}. \tag{3.3}$$

However, (3.1) is plausible for limited ranges of the control variables (it picks out a dominant term from (3.3) over those ranges) and much easier to analyze. Moreover, statistical analysis based on (3.1) might be a productive first step toward analysis of more complicated models like (3.3).

The main experiment was designed to have four stages:

(1) preliminary screening;
(2) collection of large data sets for detailed analysis;
(3) estimation – regression analysis of the data from (2);
(4) validation – evaluation of the predictive power of the models developed in (3).

The regression analysis of the data in both stages (1) and (3) is facilitated by a logarithmic transformation of the model (3.1) to the equivalent model:

$$Y = \ln\varepsilon + \beta_0 + \beta_1 \ln n + \beta_2 \ln m + \beta_3 \ln\ln C + \beta_4 \ln f + \beta_5 \ln d + \beta_6 \ln s \tag{3.4}$$

of Y, the natural logarithm of execution time. Standard techniques of linear regression apply to (3.4). The package SAS was used for this purpose.

The screening tests were based on a half-factorial design. They indicated that when the control variables were chosen from the intervals of values previously selected (see Section 2), then the model (3.4) was sufficiently promising to continue with large-scale tests. Consequently we designed the experiment of stage (2) to generate 100 observations where each of n, m, and C was chosen uniformly from the respectives values $n \in \{$ 512, 1024, 1536, 2048, 2560, 3072, 3584, 4096, 4608, 5120, 5632, 6144, 6656, 7168, 7680, 8192 $\}$, $m/n \in \{$ 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32 $\}$, $C \in \{2^8, 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16}, 2^{17}\}$. With NETGEN s is determined by choosing uniformly $n/s \in \{2, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7\}$; with CAPT $s = n/2$. In both cases f is set by choosing uniformly $f/s \in \{2^6, 2^7, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}\}$, and d is chosen to be $k \cdot (f/s) \cdot (n/m)$ with k selected uniformly from $\{1, 2, 3, 4, 5, 6, sm/n\}$ for NETGEN and $\{1.5, 2.5, 3.5, 4.5, 10m/n\}$ for CAPT. Each observation in stage (2) consisted of a min cost flow instance solved to optimality by each of the four codes. The values of the control variables, seed, other generator input parameters, the optimal solution value, and the execution time (in seconds) for each code were recorded, along with information about the number of saturated arcs in the optimal solution, the step times and counts, etc.

3.2. Regression analysis. The regressions on the data from stage (2) based on (3.4) are summarized in Table 1 which shows the least-squares estimators b_i of β_i, $(0 \leq i \leq 6)$, and their standard errors, as well as the R^2 values for each of the eight samples (4 codes $\times$ 2 generators) of 100 instances.

Examining the third column of Table 1(b), for example, reveals that the model estimates the mean of the natural logarithm of execution time for RNET 3.61 on the NETGEN networks by

(3.5)
$$-12.87 + 0.58 \ln n + 0.92 \ln m + 0.08 \ln \ln C + 0.27 \ln f - 0.20 \ln d - 0.07 \ln s$$

which translates to an estimate

$$e^{-12.87} n^{0.58} m^{0.92} (\ln C)^{0.08} f^{0.27} d^{-0.20} s^{-0.07} \tag{3.6}$$

of the median execution time (in seconds). Point estimates of mean execution time (based on the lognormality assumption) exceed the estimates of median execution time by 0.1% to 1.6%, for the ranges of values of the control variables noted above; the mean exceeds the median by no more than 1.9% for RELAX and 0.4% for MIN3 and SCALE. On the CAPT population these percentages are smaller than for NETGEN by a factor of 2 to 4.

Figures 1–4 (a) illustrate the growth in predicted execution times with the logarithm of the number of nodes when $m = 16 \cdot n$ and the other control variables are held fixed, and Figures 1–4 (b) illustrate the growth in predicted execution times with the logarithm of the number of arcs when all other control variables are held fixed. The plots may look different if we choose different values for the control variables that are held fixed; each choice specifies a different slice of a multi-dimensional surface.

The correlation coefficient R between the vector of observations and the vector of predictions is the cosine of the angle formed by the vector of differences between the observations $Y = \ln T$ and their mean $\overline{Y}$ and the vector of differences between the predicted values

$$\widehat{Y} = b_0 + b_1 \ln n + b_2 \ln m + b_3 \ln \ln C + b_4 \ln f + b_5 \ln d + b_6 \ln s$$

and $\overline{Y}$. R^2 can be regarded to be the fraction of the variation about $\overline{Y}$ that is explained by the model. High R^2 values indicate that the model has fitted the data well. Seven of the eight R^2 values are between 0.962 and 0.993; R^2 for RELAX on the NETGEN sample is 0.914.

Because of the large number of control variables, it is difficult to portray the goodness of fit by a conventional plot of observed values versus a single control variable together with the fitted curve. Figures 5–12 (a) offer that sort of information in a slightly different form. Here, we have accumulated the effect of the entire set of control variables by plotting for each of the observations

$i = 1, \ldots, 100$ the predicted value $\widehat{Y}_i$ versus the observed value Y_i. The clustering of the pairs $(Y_i, \widehat{Y}_i)$ about the 45°-line is a clear indication of good fit for each of the samples.

The standard errors given in Table 1 allow the reader to ascertain the precision of the estimators. An estimator that is small in magnitude (relative to its standard error), e.g. the estimator b_3 associated with $\ln C$ in the RNET 3.61 model for the NETGEN population, suggests that the associated control variable may not be helpful in explaining variations in the response variable, Y. The ratio of an estimator b_i of β_i to its standard error is t_i, the t-value of the estimator. Here most of the t-values are large and have very small significance probabilities (for the test of the hypothesis that $\beta_i = 0$). The estimator b_6 associated with s tends to have small t-values, suggesting that the control variable s may be dropped from the model.

All of the analysis reported in this section is based on regressions with all of the control variables present in the model; however for any one code and distribution several of those parameters may be insignificant. Consideration of the information in Table 1 together with other data gathered in the regression analysis yields insights about which control variables are the most critical in explaining variations in execution times of a given code on a particular population. For example regressions with only n and m are sufficient to give R^2 values of better than 0.924 for MIN3 on the CAPT sample and 0.948 on the NETGEN sample. Similarly for SCALE n and $\ln C$ are enough to get $R^2 = 0.948$ on the NETGEN sample.

It is interesting to consider by how much R^2 drops when a single control variable is excluded from the model — this gives a measure of the sensitivity of the fit to the inclusion of that variable in the model. (Note that this is not the same issue as whether the estimate b_i of β_i is large in magnitude, since there may be correlations among the variables.) For example on the NETGEN sample SCALE is sensitive to n, but not to m: SCALE's R^2 value falls by 0.332 if n is dropped from the model, but by only 0.010 when m is dropped. On the CAPT sample the respective numbers are 0.093 and 0.005. SCALE is not as sensitive as one might expect to $\ln C$ — for both samples R^2 remains above 0.900 when $\ln C$ is dropped from the model.

All four codes are insensitive to s; R^2 never drops by more than 0.007 when s is excluded from the NETGEN models. (Recall that s does not appear in the CAPT models.)

MIN3 is also relatively insensitive to $\ln C$, d and f, with R^2 never decreasing by more than 0.018 on the NETGEN sample or 0.055 on the CAPT sample with the deletion from the model of any one of these three parameters; dropping all of them simultaneously still gives R^2 of 0.948 on NETGEN and 0.924 on CAPT.

On the CAPT sample, RELAX is extremely sensitive to each of d (R^2 decreases by 0.431) and f (0.237) and a little sensitive to n (0.080), but not $\ln C$ (0.000) or m (0.001). However, on the NETGEN sample it is somewhat sensitive to m (0.131) and not at all sensitive to d (0.000).

RNET 3.61 is quite sensitive to d on the CAPT sample (0.262), less so on the NETGEN sample (0.070), where it is sensitive to m (0.184). In both cases it is impervious to $\ln C$ (0.000).

For each of the four codes the estimate of the exponent of d is negative. All four codes seem to run faster, on average, as the arc capacities are loosened. This effect is less pronounced for the NETGEN population, where RNET 3.61 had the largest negative exponent on d, -0.203.

In order to evaluate further the effects of variations in the tightness of capacities, each of the 100 instances in each of the CAPT and NETGEN base tests was re-run twice – once with very loose capacities and once with the parameter k fixed at 2.5 on the CAPT instances, where it determines *intlen*, and fixed at 2 on the NETGEN instances, where it determines *hicap*. All input parameters other than k were identical in the two replicates. Figures 13–16 show clearly that all four codes ran much slower on the CAPT instances with tight capacities and all but MIN3 ran slower on the NETGEN instances with tight capacities. The effects on RNET 3.61 times are especially noteworthy – in both sets RNET 3.61 ran slower on all 100 instances with tight capacities. The percentage of saturated arcs at optimality for the instances with tight capacities ranged between 9.2% and 28.8% and averaged 24.0% on the CAPT set, and ranged between 2.3% and 32.2% and averaged 15.4% on the NETGEN set.

3.3. Analysis of residuals. Some of the statistical interpretations of the regressions depend on the lognormality assumption, which is equivalent to the assumption that the random variables $\ln \varepsilon(n, m, \ln C, f, d, s)$ are independent normals with mean zero and constant variance. In order to evaluate the accuracy of that assumption, we produced scatter plots of residuals $(Y - \widehat{Y})$ versus the different control variables; these plots do not show marked deviations from what one might expect under the lognormality assumption. More revealing are the normal probability plots of Figures 17–20. If the lognormality assumption were valid and the model were correct, the studentized residuals, i.e., the residuals divided by their standard errors, should be independent standard normals. The normal probability plots are produced by ordering the hundred studentized residuals from smallest R_{j_1} to largest $R_{j_{100}}$ and plotting each pair (R_{j_i}, Z_i), where Z_i is the inverse value of the ith percentile of the standard normal cumulative distribution function. Clustering of these pairs about the 45°-line indicates that the lognormality assumption is approximately correct. For comparison, Figures 21–22 show normal probability plots from two sets of 100 pseudorandom standard normal variates generated by SAS using sums of twelve uniform variates, and the Box-Mueller method, respectively. The probability plots of studentized residuals in Figures 21–22 do not differ appreciably from the plots of Figures 17–20.

3.4. Tests with independent data sets. In the fourth stage of the experiment we generated eight new sets of 100 observations each, as in stage (2), but the new *validation sets* are independent of the earlier *base sets* on which the regressions were performed and predictions are based. Figures 5–12 (b) show the plots of predicted values versus observed values of the logarithm of execution times for the new validation sets. These plots indicate clearly that the predictions are good. Variations about the predictions appear to be symmetric and not sensitive to the magnitude of the predicted value. The plots for the validation set do not differ substantially from those for the base set on which the estimation was carried out. The mean errors from prediction on the CAPT validation set are almost as small as on the CAPT base set. They are relatively larger for the NETGEN validation set.

One can compute confidence intervals on individual observations based on the lognormality assumption and the assumption that the model is correct. Table 2 presents counts on the number of observations out of 100 in the validation sets that fall below, within, and above specified confidence intervals. These counts provide further evidence that the model (3.1) is approximately correct over the distributions of networks from which the base and validation sets were drawn and that the lognormality assumption is approximately correct.

3.5. Lognormality assumption. The lognormality assumption underlies some of the statistical evaluations, interpretations, and applications of the regressions, for example, the evaluation of precision based on t-statistics, and the construction of confidence intervals for prediction of individual observations. Our use of confidence intervals above was in an inverse role, to validate the model and the lognormality assumption. If correct, the lognormality assumption would indicate that factors contributing to the inherent variability of execution times accumulate in a multiplicative way. This seems to be interesting in its own right, and suggests further study of other optimization problems and codes for similar behavior.

The estimates from the main experiment appear to lead to acceptable models of median execution time. However, one should be cautious in accepting the lognormality assumption as a close enough approximation to the truth to invoke formal statistical interpretations that derive from it, or to pretend that the inherent variability of execution times (separate from responses to the control variables) is understood well. In order to examine this issue further, we collected additional observations with large numbers of replicates. (In replicates all generator input parameters except the seed are held fixed.)

For each of CAPT and NETGEN we generated 2500 new observations in five sets of 500 replicates, each set with a different choice of values of the control variables. Figures 23– 24 show the normal probability plots for these observations, where residuals are measured from the sample mean of the 500 replicates. These probability plots are quite striking. The lognormality assumption appears to be accurate for all four codes on the CAPT networks, and on the NETGEN networks except for RELAX, where deviation from lognormality is apparent outside of the center of the distribution. The probability plots suggest that under the distributions of networks considered here, although (3.1) may not be quite right, there are accurate models of median and mean execution times with the same choice of control (independent) variables. The large numbers of replicates in these sets revealed some significant bias in the current model (with the estimators from our base tests). It appears that this may be due largely to inaccuracy in the way in which the model reflects the effects of capacitation; see the next section. Further study is needed. It appears from these replicate sets that the greater variability in residuals from predictions for RNET 3.61 and RELAX over SCALE and MIN3 is due to greater inaccuracy in the RNET 3.61 and RELAX models; the residuals from the sample means for each set of 500 replicates were relatively smaller for RNET 3.61 and RELAX. Indeed, the combination of faster run-times and smaller variability necessitated running RNET 3.61 and RELAX on larger (harder) replicated inputs than SCALE and MIN3 – the small variations in RNET 3.61 and RELAX runtimes on the smaller networks tended to result in large numbers of duplicate values among the 500 replicates (due to the 1/60 second accuracy in the timer).

3.6. More on the model. One should not attach too much importance to the estimates b_i of the individual coefficients β_i; it is the aggregation of their effects that is interesting. Not only are there strong (positive and negative) correlations between pairs of the b_i's, but their values depend on certain arbitrary choices on how the model is represented. Such substitutions as m/n for m or dms/nf for d do not really change the model. They result in exactly the same fit, merely transforming its representation in a way that alters the coefficients b_i in accordance with the substitution.

A variation on (3.1) in which $\ln u$ and $\ln f$ replace u and f, respectively, gives a symmetry in the treatment of u and f with C, which plays a role dual to u and f in the minimum cost flow problem. For some algorithms that are naturally symmetric in their treatment of primal and dual variables this symmetry in the model might be desirable. Regressions with $\ln u$ and $\ln f$ in place of u and f did not change the R^2 values appreciably – in some cases they increased slightly, and in others they decreased slightly. Perhaps if the experimental ranges on u and f were wider, the effects would be more noticeable.

It has been suggested that the diameter of a network may be influential on the run-times of certain minimum cost flow codes [**5**]. It might be interesting to study this over some network distributions where diameters are easy to estimate.

The inclusion in these tests of uncapacitated networks along with capacitated networks causes a difficulty in our attempt to evaluate responses of run-times to variations in capacitation. In a model of the basic form (3.1) applied to capacitated networks there are simple and natural ways of defining a parameter d that measures strictness of capacities, for example, the maximum capacity. When uncapacitated networks are also included, some convention is needed to extend the definition of d. Regarding uncapacitated networks to be capacitated with large, effectively infinite capacities leads to an assignment to the d above of a value representing effectively infinite capacity. The effect on the regression of arbitrary choices that may differ by orders of magnitude can be substantial. It seems plausible that one should avoid, if possible, assignment of a value much larger than necessary to guarantee, or, at least, make it likely that no edge with that capacity will be saturated in an optimal solution. It appears that the accuracy of the estimation based on our main experiment is improvable by a more careful treatment of this issue. Capacitation does seem to be contributing more to bias in the model than any other control variable. One could use the fraction of arcs that are saturated in an optimal solution as a good measure of capacitation. (Let *bfrac* denote this fraction.) This might be useful in evaluating how run-times respond to changes in capacitation, but not of much practical use for prediction, since it requires solving the problem (perhaps with a different code) in order to evaluate one of the independent variables. Some regressions were run using *bfrac* in place of d. It did not improve the R^2 values as much as we expected on the CAPT and NETGEN samples.

4. Some Additional Experiments

Subsequent to the main experiment we ran additional tests along the same lines using two other generators: GRIDGEN [**20**] and GOTO [**14**]. The model(3.1) fit the GRIDGEN data well with R^2 values between .95 and .99. See Table 3a and Figure 3. For the GOTO sample the model (3.1), without the control variables f and s (which are not controllable inputs with GOTO), did not fit as well here as in the samples from the three other generators, with R^2 values between 0.711 and 0.909 (see Table 3). This generator seemed to produce more difficult instances than the other three, and there is some indication that the difficulty of the GOTO networks is related to the fraction of the arcs that are saturated at optimality. When *bfrac* (which should not be regarded as a generator input) is included to extend the model (3.1) by multiplying the right hand side by $(bfrac)^\gamma$, the R^2 values go up dramatically (see the last line of Table 3). The relative behavior of the codes on the GOTO sample was rather different from what we saw in the main experiment (see Figure 4).

5. Ad Hoc Runs

While the analysis of the previous sections leads to statistically significant interpretations, the comments below are impressions based on small numbers of additional ad-hoc tests. They raise some points for further consideration.

5.1. DIMACS benchmarks. In order to facilitate run-time comparisons with other computing environments, Table 4 gives run-times of the four codes on some benchmark networks described in the DIMACS document [**9**]. In each case the seed was set to 270001.

5.2. Large sparse networks. In order to evaluate responses to density, it was necessary to design the main experiment so that the sample included a wider range of densities than one often sees in network flow experiments; this limited the number of nodes in those networks to the range 512–8,192. Later we ran the codes on sparse networks with 16,384–55,056 nodes and 32,768–165,167 arcs. There were twenty networks generated by each of CAPT and NETGEN. The total run-times for each code are summarized below.

Generator	RNET 3.61	RELAX	MIN3	SCALE
CAPT	773.79	9666.43	2505.48	5262.06
NETGEN	9210.97	1342.92	8932.87	19681.8

RNET 3.61 was the fastest on 19 of the 20 CAPT networks and RELAX was the slowest on 13 of the 20. RELAX was the fastest on all 20 NETGEN networks and SCALE was the slowest on 19 of the 20. Because these networks have far more nodes than those tested in the main experiment, there is reason to expect greater deviations here from predictions based on the main experiment. Indeed the mean errors are much larger here, except for the combinations of RNET 3.61 with CAPT and RELAX with NETGEN.

5.3. A few large, dense networks. We tested a small number of dense networks larger (in m) than those included in the main experiment. The models that fit well earlier did not predict so well on these new networks outside the population on which the model was developed. These few large, dense networks also offer further evidence that relative performance of codes may be highly sensitive to the population of test problems. We solved four networks from each of CAPT and NETGEN, with 520,000–530,000 arcs and 8,000–33,000 nodes. Cumulative times show that on the four CAPT networks RNET 3.61 ran faster than the next fastest code by a factor of four and RELAX ran slower than the next slowest by a factor of two. On the four NETGEN networks RELAX ran faster than the next fastest by a factor of 2.5 and RNET 3.61 ran slower than the next slowest by a factor 2.5. We also ran some very dense NETGEN networks on 2048 nodes, two with approximately 524,000 arcs and two with more than one million arcs. On these networks and the sparser $m > 500{,}000$ networks, uncapacitated or very loosely capacitated examples were much easier for RNET 3.61 and RELAX, and to a lesser extent, SCALE, than the more tightly capacitated examples.

5.4. Variations on the codes. Interpretation of these experiments should be viewed in terms of the specific implementations of the four algorithms. Clearly there are many modifications that might improve performance.

The cost-scaling approach has not reached a mature implementation. The version of SCALE employed here differs from the first version [**8**] primarily in the substitution of a max flow algorithm based on [**16**] for one based on [**21**].

This appears to yield substantial speed-ups on sparse networks. The choice of a scaling factor of four in SCALE was based on testing with the old max flow routine; this should be re-examined. It would also be interesting to examine arc-fixing, early termination, and pre-processing in SCALE.

With preprocessing of the cost data to reduce the maximum absolute cost one might hope to reduce the number of iterations required by SCALE and MIN3. In a small number of trials on CAPT and NETGEN networks the costs were transformed, but with no appreciable benefit for any of the codes; for relatively dense networks the preprocessing time was substantial and outweighed any improvements in run-times. There were surprising outcomes in some runs of SCALE on GOTO networks. The initial node numbers were changed from zero to minimize the absolute value of the costs incident with a given node; the nodes were processed in the order in which they were given and several passes were done. We were not able to reduce the number of scaling iterations, but the SCALE execution times (including preprocessing) dropped by factors of 6–8 on some of the larger GOTO networks ($n = 8192$, m =60,000–80,000) run. It is not yet clear how consistently such speed-ups occur on GOTO networks.

Some ad hoc testing of variations on the MIN3 code offer indications of potential for general improvements, as well as some changes that could make the code run faster on specific input classes. MIN3 only fixes an arc when the magnitude of its reduced cost exceeds $n\epsilon$, the theoretical worst-case criterion. Fixing at lower values, with provision for correcting for mistakes at the end, led to improved running times on CAPT and NETGEN networks. A scaling factor of two was found to work better on GOTO networks, whereas four seems to work better with the other generators examined here. More robust price updates than the single node update, as described in [**17**] and employed in MIN3, look promising, but we have not yet tested this. There seems to be a significant potential for improved implementations of the relatively new successive approximation algorithm.

Version 3.61 of RNET (1979) as used here was not designed for networks of the size and density of some of those in our tests. It is likely that substantial speedups could be achieved on large networks by better choices of the RNET 3.61 parameters, and by modification of the pricing routine, e.g., employing a candidate queue.

We have done some experimentation with the RNET 3.61 parameters *frq* and p_0. The frequency *frq* controls the partitioning of the edge set for partial pricing, and p_0 controls the initial costs on the artificial edges. For each of two CAPT and two NETGEN networks from the base sets we ran RNET 3.61 with several different values of *frq* and each choice of $p_0 \in \{1, 2, 3, 4, 10, 200\}$.

On the CAPT networks the choice of p_0 made little difference; for every choice of *frq* only one pass was required to drive out the artificial variables, and the variations in run-times over the six choices of p_0 never exceeded 8.5%. Run-times were sensitive to the choice of *frq*, sometimes exhibiting abrupt responses to small variations. For example, on one of the two networks the six run-times for each of *frq*=2.50, 2.22, and 2.00 ranged between 20.93-21.28, 45.61-48.15, and 22.9-23.0 seconds, respectively. General responses to *frq* were somewhat unpredictable, but in both tests the minimum run-time (among all 144 combinations of 24 different values of *frq* between .06 and 20.0 and the six choices of p_0) occurred with *frq*=2.5 and p_0=3:

Network	n	m	Base test time	Min. observed time
CAPT3-115	8,096	194,304	76.1	23.35 at *frq*=2.5 p_0=3
CAPT3-146	7,680	215,040	54.1	20.93 at *frq*=2.5 p_0=3

This prompted us to re-run all 100 networks from the CAPT validation set with *frq*=2.5 and p_0=3. It ran faster on all 100 networks by a minimum of 17.5%. The average improvement of 59.2% is almost a 2.5-fold speedup. (Approximately the same speedup was also achieved on the large sparse CAPT networks reported on above.)

On the NETGEN networks the choice of p_0 as well as *frq* made a substantial difference. Here we examined 14 values of *frq* between .06 and 4.

Network	n	m	Base test time	Min. observed time
NETG4-121	5,632	123,904	191.6	188.7 at *frq*=0.50 p_0=10
NETG4-140	8,192	180,224	348.0	291.7 at *frq*=0.25 p_0=3

In contrast with the CAPT networks, here the parameter values used in the main experiment were not far from the best observed. This is not surprising, since the choices of the RNET 3.61 parameter settings for the main experiment were based on preliminary tests on NETGEN networks. Note that our choice of $p_0 = 200$ for the main experiments essentially forces RNET 3.61 into a Big–M scheme, taking a single pass to force all of the artificial edges out of the basis. This is contrary to the intentions of RNET 3.61's authors, and it can have a disastrous effect when combined with an unfortunate choice of *frq*. However in our (limited) testing with fixed values of *frq* and varying values of p_0, the unusually high value of p_0 did not cause run-times much greater than values in the ranges they suggest (and sometimes gave faster times), as long as *frq* was not too large.

Four modifications of the RNET 3.61 pricing strategy (other than the choice of frq) were tested on a set of 100 large, sparse NETGEN networks and a few

GOTO networks. The four strategies are:

(i) the original RNET strategy – the arc list is partitioned into approximately $\frac{m \cdot frq}{100}$ blocks consisting of arcs whose indices are congruent modulo $\frac{100}{frq}$;

(ii) the original RNET strategy, but with the blocks constructed by taking sets of $\frac{100}{frq}$ arcs with consecutive indices, beginning at the top of the arc list;

(iii) the original RNET strategy, but with the blocks constructed by taking sets of $\frac{100}{frq}$ arcs with consecutive indices, beginning at the top of a random permutation of the arc list;

(iv) a candidate queue strategy (see [**22**]). Each time the queue is constructed it is filled with basic arcs and negative transformed cost arcs until it has $\frac{m}{20}$ arcs in addition to the basic arcs. The queue is refreshed after pricing for at most $\frac{m}{200}$ iterations. The number of arcs priced is set to $\frac{100}{frq}$, as above.

A variation on these strategies is to topologically sort the arcs, and scan them in groups incident with a node[**22**].

The table below displays the total runtimes with each of the four strategies for each of two settings of *frq*, 0.5 and 0.1. All times are on an IBM RS/6000 Model 550 with 256M of memory and running AIX 3.2, and with $p_0 = 2$.

frq	i	ii	iii	iv
0.5	22042.8	16857.6	16438.7	10871.2
0.1	23032.4	12292.3	12512.5	16486.0

Strategy (iii) was faster than (i) on all 200 runs. Strategy (ii) exhibited somewhat more variability than (iii). Strategy (iv) combined with $frq = .5$ appears to be superior to the other seven combinations tested on these NETGEN networks; it achieved the minimum run-time among the eight on 69 of the 100 networks, and was only about 10% slower than RELAX in total runtime on these observations (excluding one on which we were unable to get RELAX to terminate within specified limits). It would be interesting to examine all of the strategies further, particularly with a wider range of values of *frq*. In contrast with the NETGEN runs, in runs on a small number of GOTO networks strategy (iv) was the slowest by far and (iii) was the fastest.

6. Conclusions

These experiments yielded good models of mean execution time over two distributions of networks. They offer strong evidence that over those populations, for three of the four codes the inherent variability of execution times is very accurately approximated by a lognormal distribution with constant variance. It would be interesting to extend the general model (3.1) to the form (3.3) and attempt to get accurate fits to data from distributions arising from NETGEN, CAPT, etc., with a broader domain of generator inputs.

None of the four codes studied here can be regarded to be dominant over the entire set of distributions examined.

An accurate model of the average behavior of a particular code over a population of networks based on empirical data presents very different information from an estimate of worst-case complexity, and may appear to be of greater practical interest. However there are inherent limitations to such studies as these. In principle, the worst-case complexity of an algorithm over any permissible population of inputs can be estimated, and for the answer to be nontrivial the population will have to have infinite variation in the parameters of interest. However, for a sufficiently large diverse population there may not exist any simple model that accurately captures the average behavior of a particular code. Furthermore, even if an accurate and simple model exists, limitations on computational resources and on our ingenuity at generating test instances may not allow for adequate sampling of a large diverse population.

This study, like all empirical studies we know of in mathematical programming, samples from rather narrow classes of inputs. Thus we may find that what emerges is a good model of empirical behavior over a very limited population, and that may be the best we can hope for initially. Such models certainly extend our understanding of the codes under examination, and, with appropriate qualifications, the algorithms that the codes implement. If well designed, they may offer insights that can lead to more robust models and codes that are also more robust. However performance can be quite sensitive to variations in the codes and in the population of inputs. Certainly the realm of commercial applications of the minimum cost flow model does not have a sufficiently close connection to the populations that we test in this study, or any other comprehensive study of network flow algorithms that we know of, to draw fine conclusions about computation in that realm. In particular, we can construct from the generators different populations that have much more in common with one another than with the "real-world", but over which the relative merits of codes studied here shift dramatically.

When we observe consistent behavior of a code across diverse populations of networks, we may be on firmer ground in offering broader assessments, and even prognosticating on its merits for practical computation. Our success at obtaining good models for a few codes over a few (admittedly narrow) populations offers some hope in that direction.

Acknowledgments. We acknowledge with pleasure the frequent advice of David Ruppert on the statistical analysis.

References

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows*, in: Optimization (G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd,eds.), Handbooks in Operations Research and Management Science, vol. 1, North-Holland, Amsterdam, 1989, pp. 211–369.
2. T. Badics, *Maximum flow code PLED*, 1991, First DIMACS International Algorithm Implementation Challenge.
3. D. P. Bertsekas, *A distributed algorithm for the assignment problem*, unpublished LIDS working paper, March 1979.
4. D. P. Bertsekas, *Distributed asynchronous relaxation methods for linear network flow problems*, Tech. Report LIDS-P-1606, Dept. of EE&CS, M. I. T., September, 1986, Revised November 1986.
5. D. P. Bertsekas, *Linear Network Optimization*, MIT Press, Cambridge, MA, 1991.
6. D. P. Bertsekas and P. Tseng, *Relaxation methods for minimum cost ordinary and generalized network flow problems*, Operations Research **36** (1988), 93–114.
7. R. G. Bland and D. L. Jensen, *On the generality of network flow theory*, U. California-Berkeley, lecture, May 1979.
8. R. G. Bland and D. L. Jensen, *On the computational behavior of a polynomial-time network flow algorithm*, Mathematical Programming **54** (1992), 1-39.
9. DIMACS, *The First DIMACS International Algorithm Implementation Challenge: The Benchmark Experiments*, December 1991, New Brunswick, NJ.
10. G. B. Dantzig, *Linear programming and extensions*, Princeton University Press, Princeton, NJ, 1963.
11. U. Derigs and W. Meier, *Implementing Goldberg's max-flow-algorithm — a computational investigation*, ZOR - Methods and Models of Operations Research **33** (1989), 383–403.
12. N. R. Draper and H. Smith, *Applied regression analysis*, second ed., Wiley, New York, 1981.
13. J. Edmonds and R. M. Karp, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. ACM **19** (1972), 248–264.
14. A. V. Goldberg, *The Grid-On-TOrus (GOTO) generator*, 1991, First DIMACS International Algorithm Implementation Challenge.
15. A. V. Goldberg, É. Tardos, and R. E. Tarjan, *Network flow algorithms*, in: Paths, Flows and VLSI-Design (B. Korte, L. Lovász, H.J. Proemel, and A. Schrijver, eds.) Springer Verlag, 1990, pp. 101-164.
16. A. V. Goldberg and R. E. Tarjan, *A new approach to the maximum-flow problem*, J. ACM **35** (1988), 921–940.
17. A. V. Goldberg and R. E. Tarjan, *Finding minimum-cost circulations by successive approximation*, Mathematics of Operations Research **15** (1990), 430–466.
18. M. D. Grigoriadis, *An efficient implementation of the network simplex method*, Mathematical Programming Study **26** (1986), 83–111.
19. D. Klingman, A. Napier, and J. Stutz, *Netgen: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems*, Management Science **20** (1974), 814–821.
20. Y. Lee and J. B. Orlin, *Computational testing of a network simplex algorithm*, 1991, First DIMACS International Algorithm Implementation Challenge.
21. V. M. Malhotra, M. P. Kumar, and S. N. Maheshwari, *An $O(|V|^3)$ algorithm for finding maximum flows in networks*, Information Processing Letters **7** (1978), 277–278.
22. J. Mulvey, *Pivot strategies for primal-simplex network codes*, Journal of the ACM **25** (1978), 266–270.
23. H. Röck, *Scaling techniques for minimal cost network flows*, Discrete Structures and Algorithms (V. Page, ed.), Carl Hansen, Munich, 1980.
24. É. Tardos, *A strongly polynomial minimum cost circulation algorithm*, Combinatorica **5** (1985), 247–255.

Table 1. Parameter estimates for (a) CAPT (b) NETGEN. Standard errors are shown in parentheses.

CAPT	SCALE	MIN3	RNET	RELAX
R^2	0.990	0.993	0.963	0.963
b_0	-8.940	-9.728	-8.413	-12.104
	(0.165)	(0.142)	(0.341)	(0.524)
b_1: $\ln n$	0.683	0.332	0.268	1.029
	(0.023)	(0.020)	(0.047)	(0.073)
b_2: $\ln m$	0.127	0.699	0.489	-0.070
	(0.019)	(0.016)	(0.038)	(0.059)
b_3: $\ln \ln C$	1.144	0.433	0.009	0.040
	(0.046)	(0.040)	(0.095)	(0.146)
b_4: $\ln f$	0.356	0.213	0.375	0.740
	(0.010)	(0.008)	(0.020)	(0.030)
b_5: $\ln d$	-0.331	-0.159	-0.356	-0.697
	(0.007)	(0.006)	(0.014)	(0.021)

NETGEN	SCALE	MIN3	RNET	RELAX
R^2	0.976	0.979	0.962	0.914
b_0	-9.173	-9.293	-12.875	-12.442
	(0.248)	(0.250)	(0.508)	(0.555)
b_1: $\ln n$	1.126	0.771	0.584	0.593
	(0.032)	(0.032)	(0.065)	(0.071)
b_2: $\ln m$	0.130	0.519	0.929	0.591
	(0.021)	(0.021)	(0.044)	(0.048)
b_3: $\ln \ln C$	1.124	0.214	0.084	1.065
	(0.067)	(0.067)	(0.137)	(0.150)
b_4: $\ln f$	0.086	0.115	0.271	0.171
	(0.013)	(0.013)	(0.026)	(0.028)
b_5: $\ln d$	-0.032	-0.006	-0.203	-0.097
	(0.008)	(0.008)	(0.015)	(0.017)
b_6: $\ln s$	-0.079	-0.092	-0.079	-0.047
	(0.017)	(0.017)	(0.035)	(0.038)

TABLE 2. Confidence interval tallies for (a) CAPT (b) NETGEN.

CAPT Confidence Interval Results of Validation Test											
		SCALE	MIN3	RNET	RELAXT			SCALE	MIN3	RNET	RELAXT
	LO	4	3	0	3		LO	18	21	24	22
95%	IN	95	94	97	97	60%	IN	64	59	61	58
	HI	1	3	3	0		HI	18	20	15	20
	LO	7	5	2	3		LO	25	32	37	30
90%	IN	92	90	94	94	40%	IN	45	37	35	39
	HI	1	5	4	3		HI	30	31	28	31
	LO	13	14	9	9		LO	32	41	46	39
80%	IN	79	77	82	81	20%	IN	25	19	18	23
	HI	8	9	9	10		HI	43	40	36	38

NETGEN Confidence Interval Results of Validation Test											
		SCALE	MIN3	RNET	RELAXT			SCALE	MIN3	RNET	RELAXT
	LO	2	4	2	2		LO	14	11	13	17
95%	IN	93	90	93	94	60%	IN	63	66	67	59
	HI	5	6	5	4		HI	23	23	20	24
	LO	5	4	2	2		LO	22	17	19	25
90%	IN	89	88	90	94	40%	IN	47	53	51	36
	HI	6	8	8	4		HI	31	30	30	39
	LO	10	7	5	5		LO	35	31	33	38
80%	IN	79	78	82	84	20%	IN	24	29	24	17
	HI	11	15	13	11		HI	41	40	43	45

TABLE 3. (a) Parameter estimates for GRIDGEN. (b) Parameter estimates for GOTO. (c) R^2 values, when *bfrac* is included in the model. Standard errors are shown in parentheses.

GRIDGEN	SCALE	MIN3	RNET	RELAX
R^2	0.965	0.983	0.978	0.951
b_0	-9.414	-9.634	-12.003	-12.971
	(0.303)	(0.234)	(0.377)	(0.508)
b_1: $\ln n$	0.780	0.467	0.346	0.459
	(0.040)	(0.031)	(0.050)	(0.068)
b_2: $\ln m$	0.428	0.821	1.125	0.714
	(0.033)	(0.025)	(0.041)	(0.055)
b_3: $\ln\ln C$	1.142	0.109	-0.201	0.921
	(0.090)	(0.070)	(0.113)	(0.152)
b_4: $\ln f$	-0.003	-0.014	0.131	0.261
	(0.017)	(0.013)	(0.022)	(0.029)
b_5: $\ln d$	0.056	0.136	-0.107	-0.163
	(0.019)	(0.015)	(0.024)	(0.032)

GOTO	SCALE	MIN3	RNET	RELAX
R^2	0.839	0.878	0.909	0.711
b_0	-16.131	-13.862	-17.965	-25.688
	(1.426)	(1.088)	(1.027)	(2.378)
b_1: $\ln n$	0.847	0.989	0.191	-1.143
	(0.335)	(0.256)	(0.241)	(0.559)
b_2: $\ln m$	1.443	1.068	2.033	3.318
	(0.313)	(0.238)	(0.225)	(0.521)
b_3: $\ln\ln C$	0.155	0.255	0.074	3.679
	(0.365)	(0.279)	(0.263)	(0.609)
b_5: $\ln d$	0.103	0.066	0.077	0.078
	(0.049)	(0.037)	(0.035)	(0.082)
R^2 incl. *bfrac*	0.978	0.980	0.978	0.963

TABLE 4. DIMACS benchmark tests with the generators (a) GGRAPH (b) GOTO (increasing density) (c) NETGEN. For other network parameters, see [**9**, Section 4].

GGRAPH	SCALE	MIN3	RNET	RELAXT
$N = 2^{12}$ (long)	485.500	113.470	8.050	13.400
$N = 2^{13}$ (long)	1959.100	907.450	13.850	36.167
$N = 2^{14}$ (long)	6950.183	3563.580	44.700	132.050
$N = 2^{15}$ (long)	29553.200	12596.330	151.967	182.867
$N = 2^{10}$ (square)	27.200	9.270	1.167	3.083
$N = 2^{12}$ (square)	258.983	69.180	7.100	76.117
$N = 2^{14}$ (square)	2401.650	791.300	90.500	1119.900
$N = 2^{12}$ (wide)	171.583	56.320	5.933	99.233
$N = 2^{13}$ (wide)	410.250	199.030	11.967	502.900
$N = 2^{14}$ (wide)	962.133	256.980	32.217	3397.717
$N = 2^{15}$ (wide)	2324.033	691.330	70.683	16644.583

GOTO	SCALE	MIN3	RNET	RELAXT
$N = 2^{8}$	11.850	5.180	3.100	78.850
$N = 2^{9}$	47.750	31.730	15.217	513.567
$N = 2^{10}$	246.067	166.070	66.300	2441.633
$N = 2^{11}$	768.550	748.420	355.033	11735.250

NETGEN	SCALE	MIN3	RNET	RELAXT
Prob. 11	1.483	0.350	0.200	0.150
Prob. 20	3.367	0.750	0.167	0.167
Prob. 27	2.517	0.850	0.267	0.100
Prob. 28	8.867	3.670	0.500	0.333
Prob. 32	16.483	7.980	1.067	0.700
Prob. 36	206.417	64.170	12.150	14.450
Prob. 38	56.250	36.630	9.000	6.133
Prob. 40	53.117	28.470	6.617	3.650

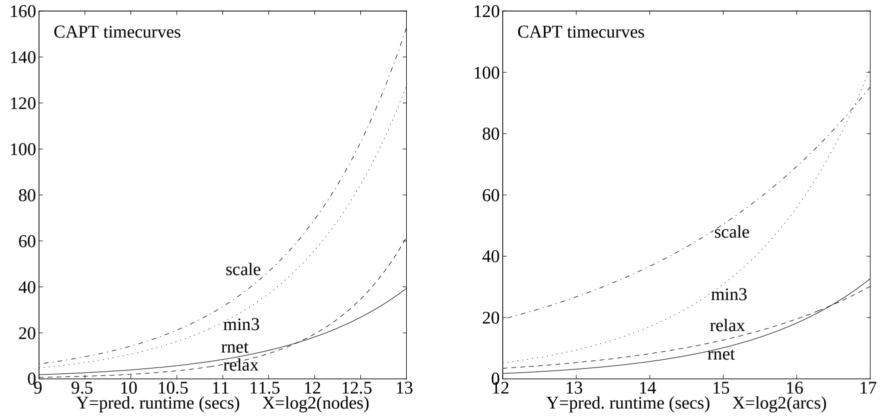

FIGURE 1. Predicted execution times (seconds) of SCALE, MIN3, RNET, and RELAX on CAPT networks: $hicost = 2^{14}$, $supply = 10^6$, $intlen = 5 \cdot supply/arcs$. (a) $nodes = 2^9, \ldots, 2^{13}$, $arcs = 16 \cdot nodes$, (b) $nodes = 2^{12}$, $arcs = 2^{12}, \ldots, 2^{17}$.

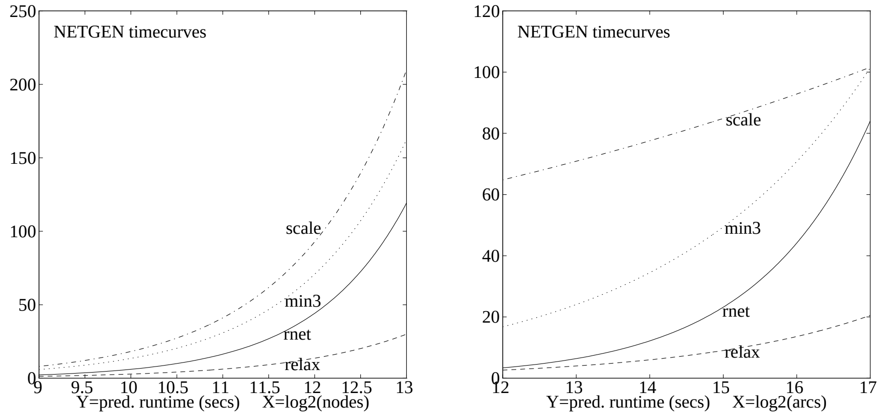

FIGURE 2. Predicted execution times (seconds) of SCALE, MIN3, RNET, and RELAX on NETGEN networks: $hicost = 2^{14}$, $hicap = 10^4$, $supply = 10^6$. (a) $nodes = 2^9, \ldots, 2^{13}$, $arcs = 16 \cdot nodes$, (b) $nodes = 2^{12}$, $arcs = 2^{12}, \ldots, 2^{17}$.

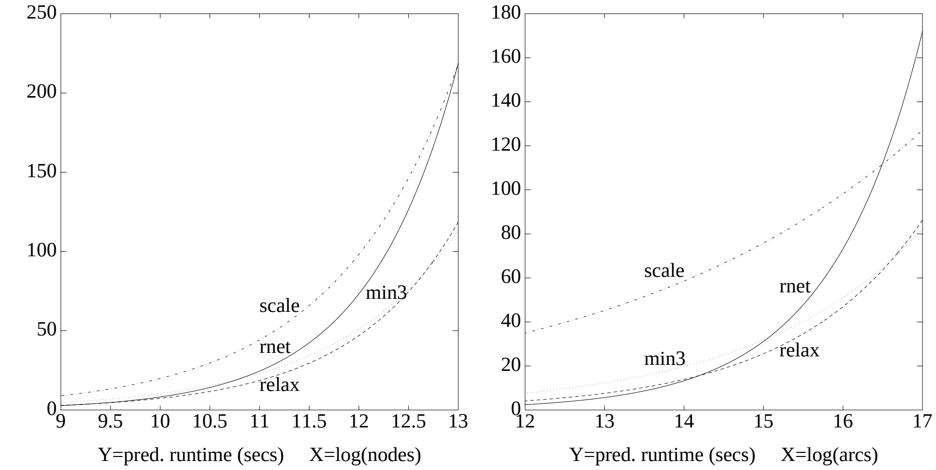

FIGURE 3. Predicted execution times (seconds) of SCALE, MIN3, RNET, and RELAX on GRIDGEN networks: $hicost = 2^{14}$, $supply = 10^6$, $hicap = 3 \cdot supply/arcs$, $width = \sqrt{nodes}$. (a) $nodes = 2^9, \ldots, 2^{13}$, $arcs = 16 \cdot nodes$, (b) $nodes = 2^{12}$, $arcs = 2^{12}, \ldots, 2^{17}$.

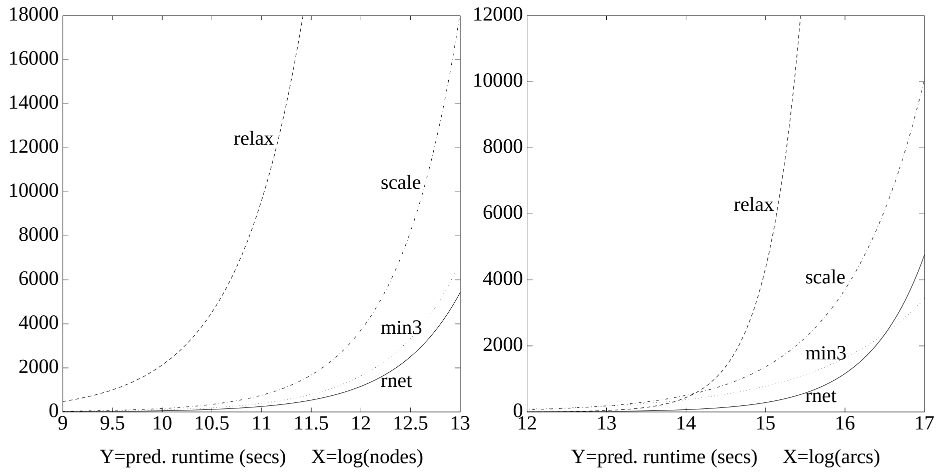

FIGURE 4. Predicted execution times (seconds) of SCALE, MIN3, RNET, and RELAX on GOTO networks: $hicost = 2^{14}$, $hicap = 10^4$. (a) $nodes = 2^9, \ldots, 2^{13}$, $arcs = 16 \cdot nodes$, (b) $nodes = 2^{12}$, $arcs = 2^{12}, \ldots, 2^{17}$.

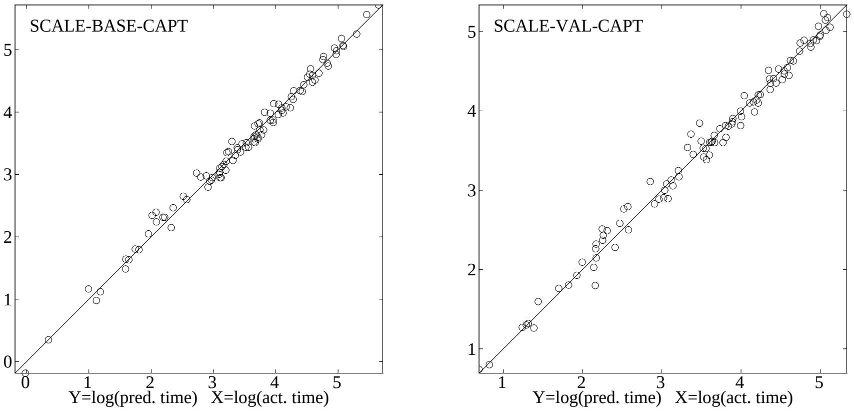

FIGURE 5. SCALE execution times on CAPT networks: Log of predicted versus log of observed execution time. (a) Base dataset. (b) Validation dataset.

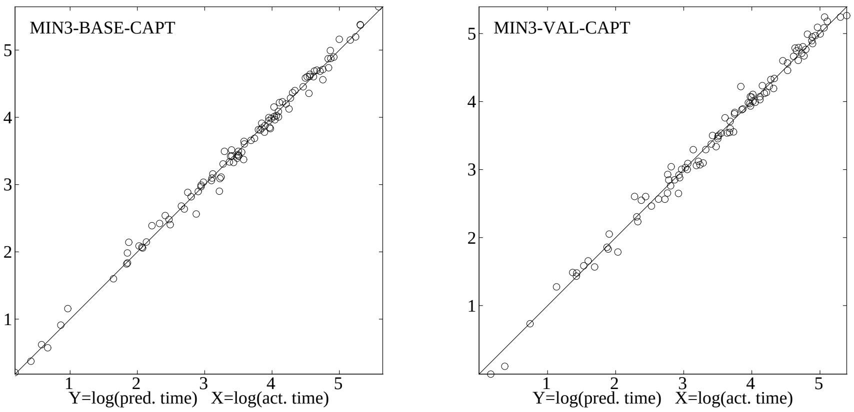

FIGURE 6. MIN3 execution times on CAPT networks: Log of predicted versus log of observed execution time. (a) Base dataset. (b) Validation dataset.

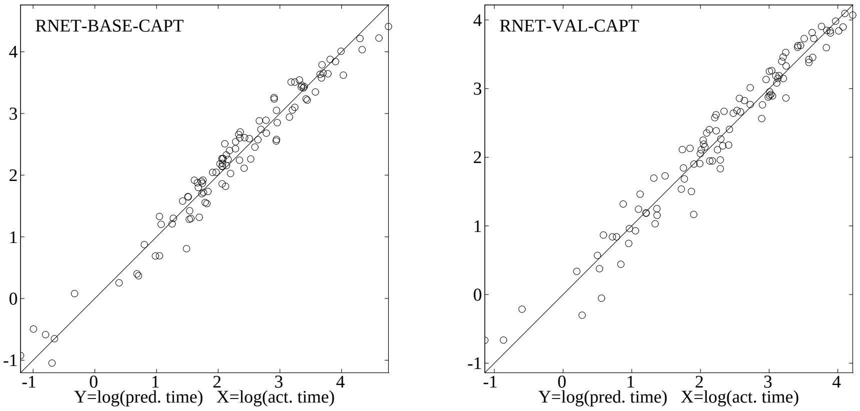

FIGURE 7. RNET execution times on CAPT networks: Log of predicted versus log of observed execution time. (a) Base dataset. (b) Validation dataset.

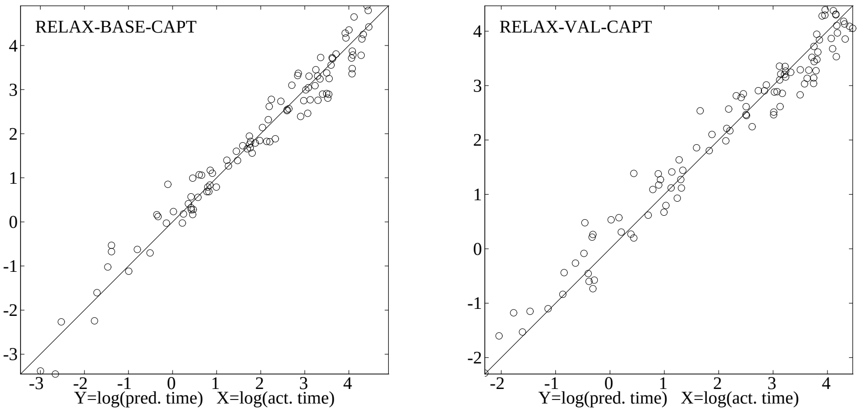

FIGURE 8. RELAX execution times on CAPT networks: Log of predicted versus log of observed execution time. (a) Base dataset. (b) Validation dataset.

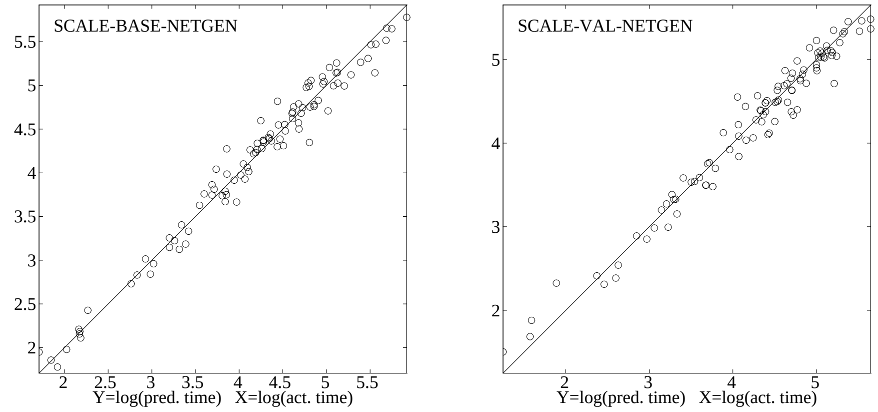

FIGURE 9. SCALE execution times on NETGEN networks: Log of predicted versus log of observed execution time. (a) Base dataset. (b) Validation dataset.

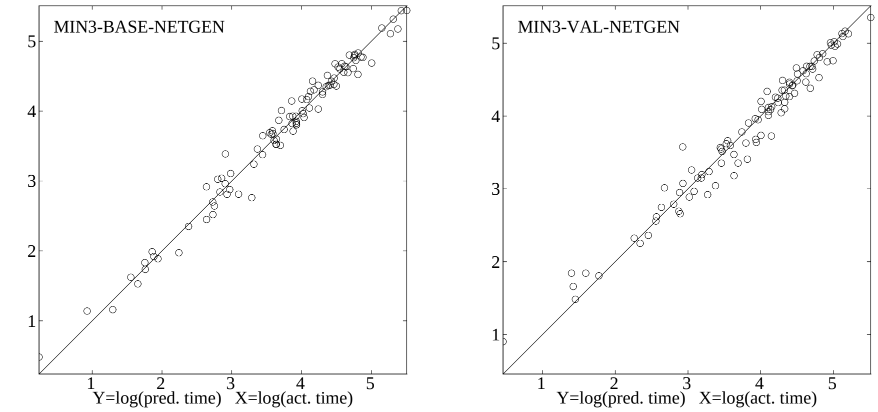

FIGURE 10. MIN3 execution times on NETGEN networks: Log of predicted versus log of observed execution time. (a) Base dataset. (b) Validation dataset.

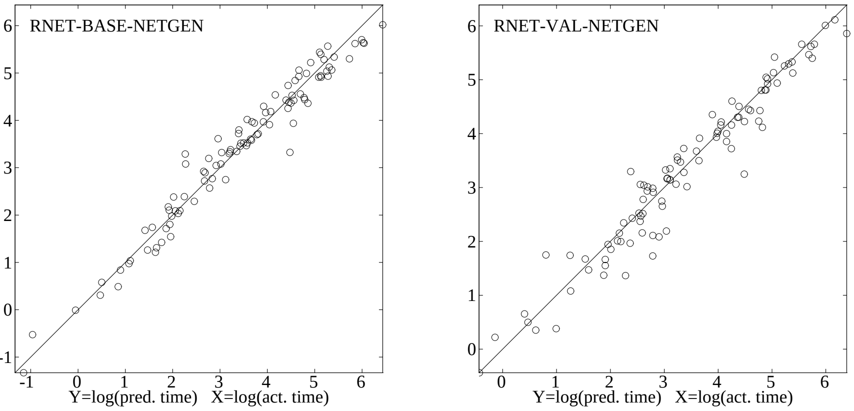

FIGURE 11. RNET execution times on NETGEN networks: Log of predicted versus log of observed execution time. (a) Base dataset. (b) Validation dataset.

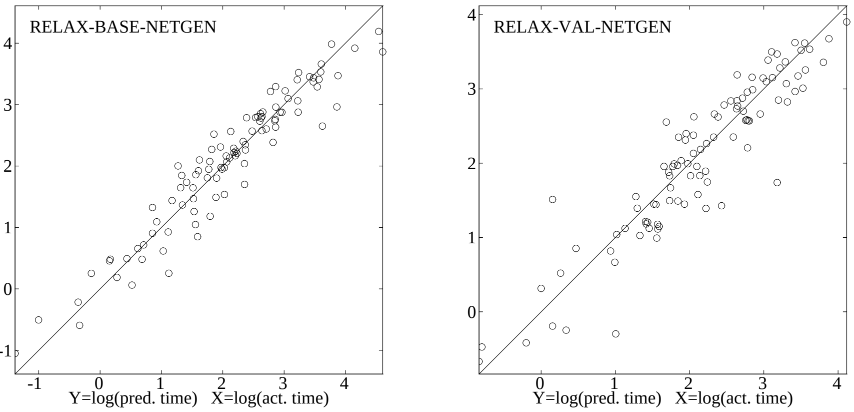

FIGURE 12. RELAX execution times on NETGEN networks: Log of predicted versus log of observed execution time. (a) Base dataset. (b) Validation dataset.

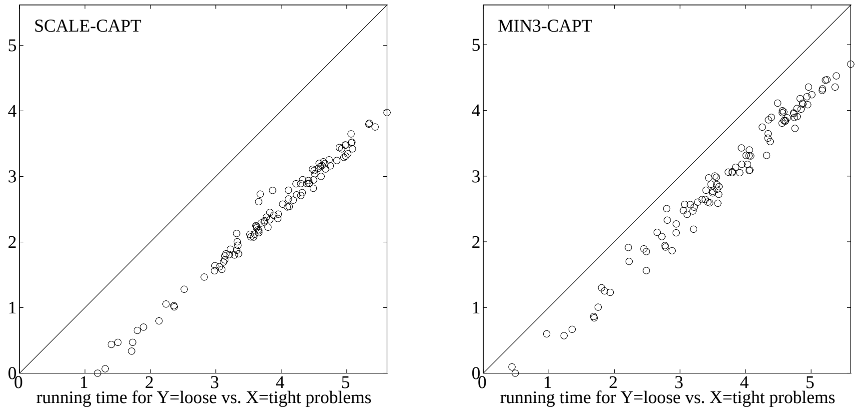

FIGURE 13. Time plots for tight and loose CAPT networks: Running times with tight capacities versus running times with loose capacities. (a) SCALE. (b) MIN3.

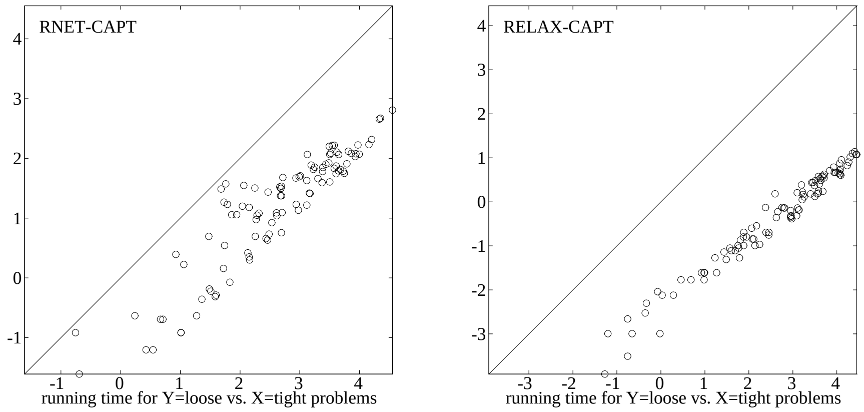

FIGURE 14. Time plots for tight and loose CAPT networks: Running times with tight capacities versus running times with loose capacities. (a) RNET. (b) RELAX.

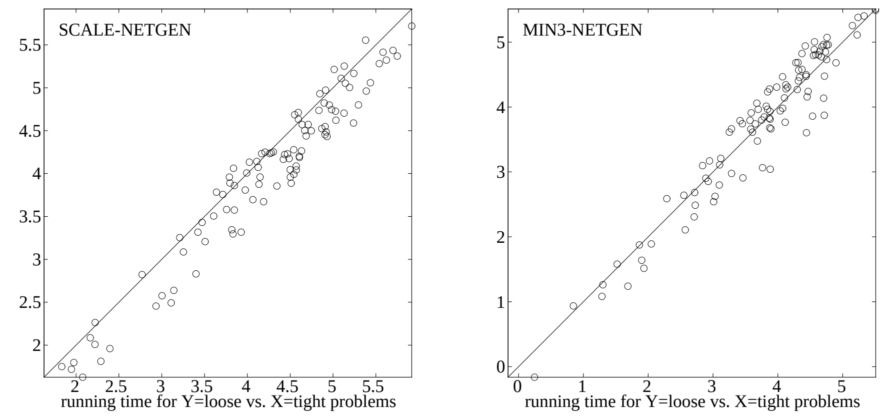

FIGURE 15. Time plots for tight and loose NETGEN networks: Running times with tight capacities versus running times with loose capacities. (a) SCALE. (b) MIN3.

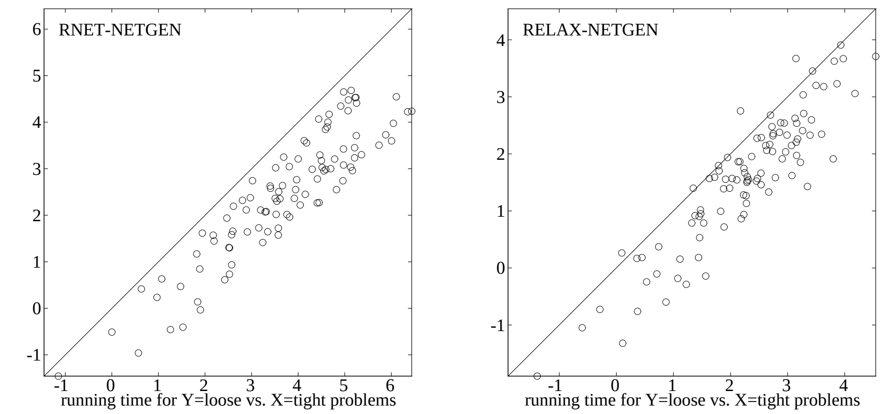

FIGURE 16. Time plots for tight and loose NETGEN networks: Running times with tight capacities versus running times with loose capacities. (a) RNET. (b) RELAX.

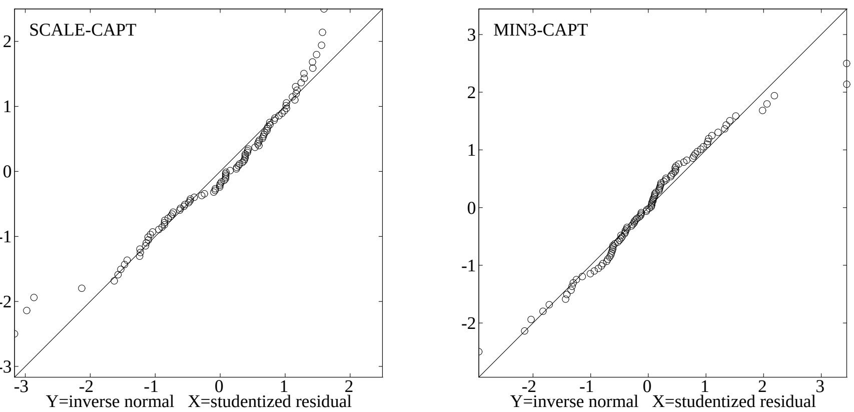

FIGURE 17. Probability plots for CAPT networks: Inverse of normal distribution versus studentized residuals. (a) SCALE. (b) MIN3.

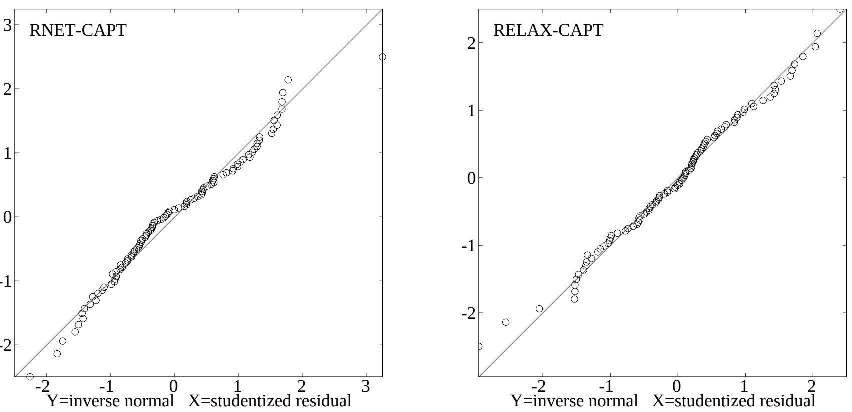

FIGURE 18. Probability plots for CAPT networks: Inverse of normal distribution versus studentized residuals. (a) RNET. (b) RELAX.

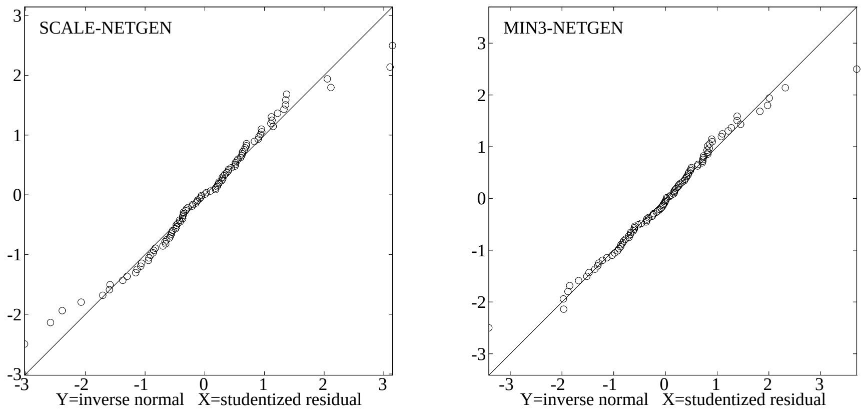

FIGURE 19. Probability plots for NETGEN networks: Inverse of normal distribution versus studentized residuals. (a) SCALE. (b) MIN3.

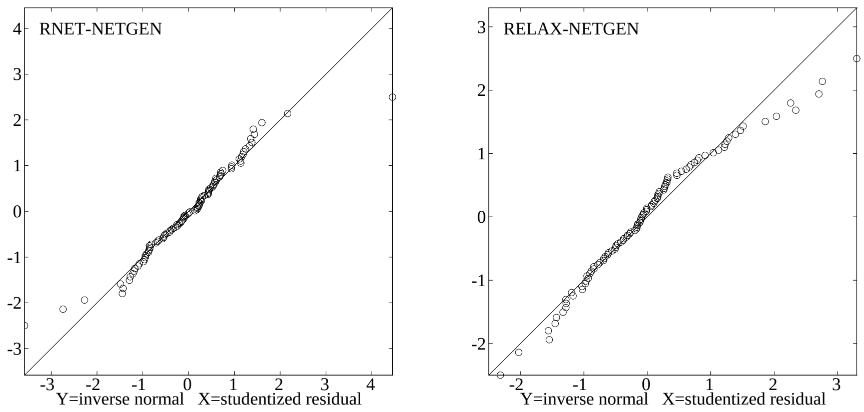

FIGURE 20. Probability plots for NETGEN networks: Inverse of normal distribution versus studentized residuals. (a) RNET. (b) RELAX.

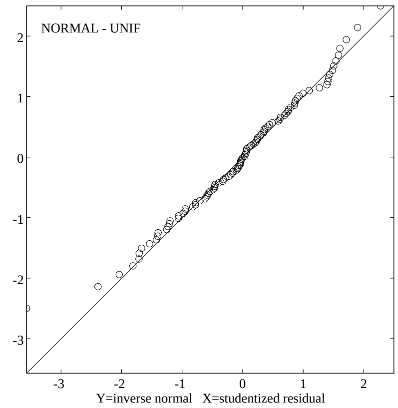

FIGURE 21. Probability plots for randomly generated normal variates created using uniform random variables. Inverse of normal distribution versus standard normal variates.

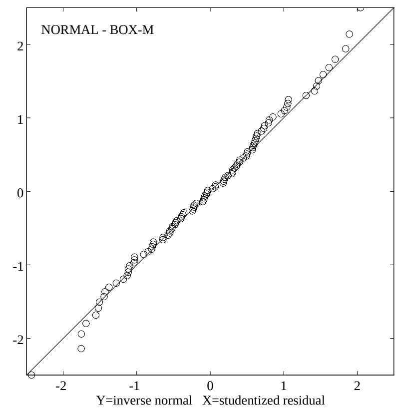

FIGURE 22. Probability plots for randomly generated normal variates created using Box-Mueller method. Inverse of normal distribution versus standard normal variates.

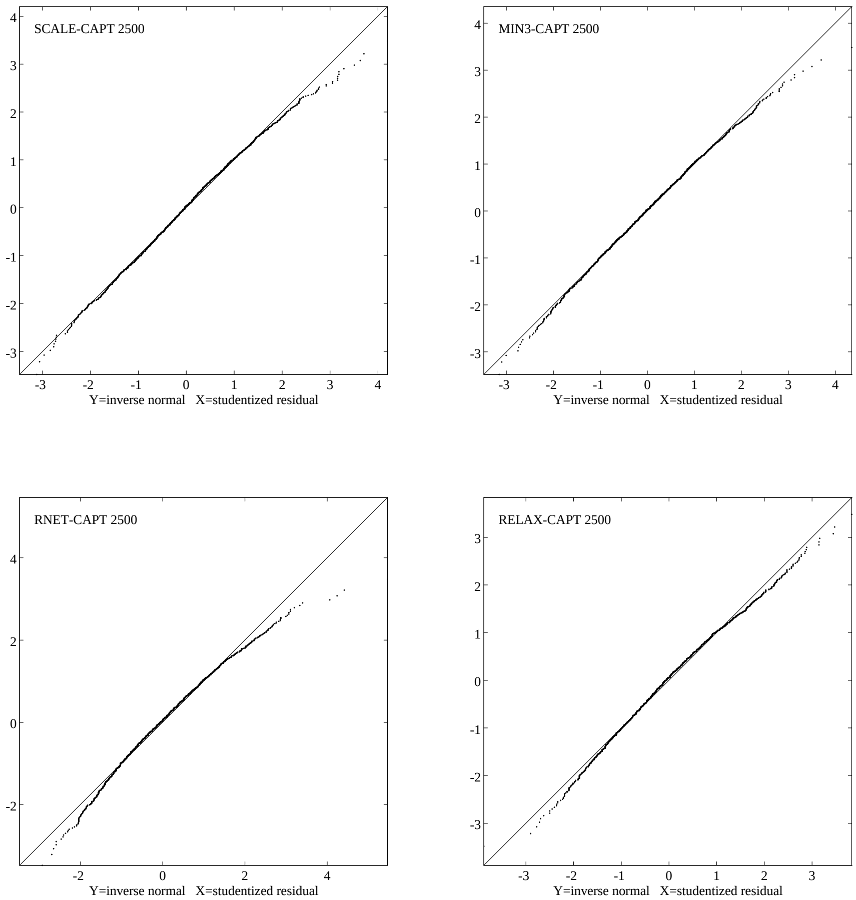

FIGURE 23. Probability plots for 2500 CAPT networks: Inverse of normal distribution versus studentized residuals. (a) SCALE (b) MIN3 (c) RNET (d) RELAX.

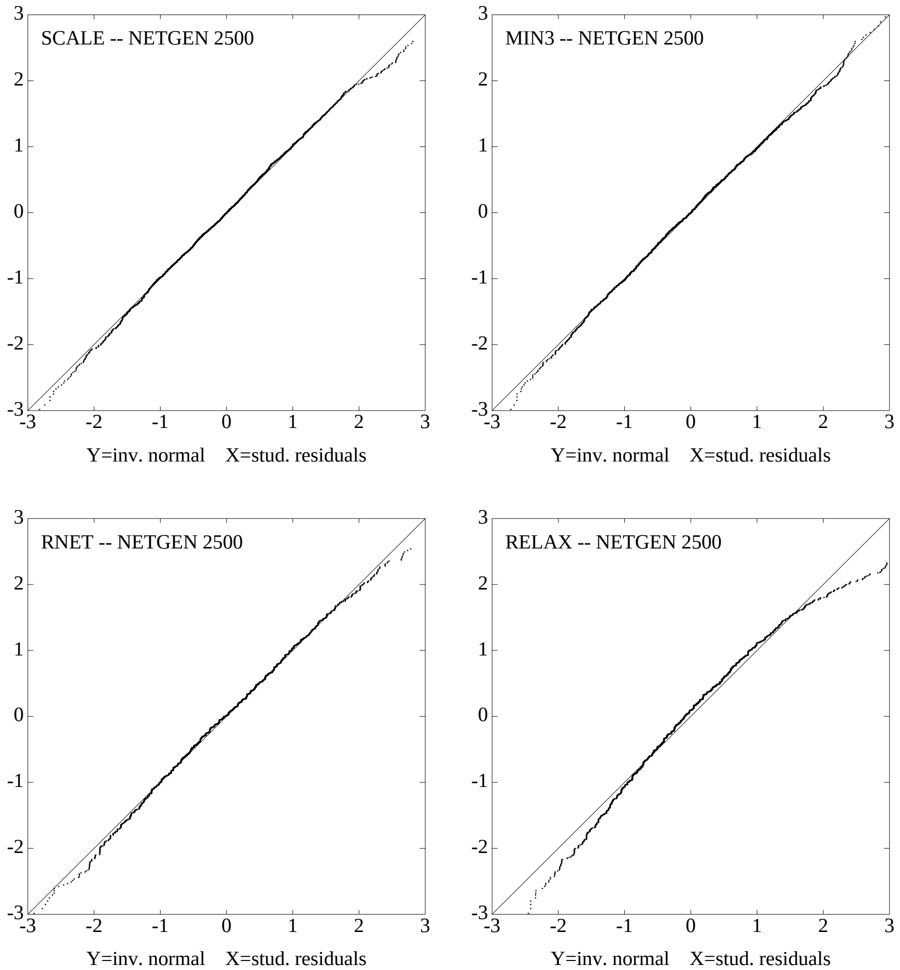

FIGURE 24. Probability plots for 2500 NETGEN networks: Inverse of normal distribution versus studentized residuals. (a) SCALE (b) MIN3 (c) RNET (d) RELAX.

School of O. R. and I. E., Cornell University, Ithaca, NY 14853–3801.
E-mail address: bland@orie.cornell.edu

Dept. of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.
E-mail address: jcheriyan@watdragon.waterloo.edu

I. B. M., T. J. Watson Research Center, Yorktown Heights, NY 10598.
E-mail address: djensen@ibm.com

School of O. R. and I. E., Cornell University, Ithaca, NY 14853–3801.
E-mail address: ladanyi@cs.cornell.edu