Use of Representative Operation Counts in Computational Testings of Algorithms

July 1992 WP #3459

Ravindra K. Ahuja*
Indian Institute of Technology

James B. Orlin*
Massachusetts Institute of Technology

* see page bottom for complete address

Ravindra K. Ahuja
Department of Industrial & Management Engineering
Indian Institute of Technology
Kanpur - 208 016, INDIA

James B. Orlin
Sloan School of Management
Massachusetts Institute of Technology
Cambridge, MA 01239

USE OF REPRESENTATIVE OPERATION COUNTS IN COMPUTATIONAL TESTINGS OF ALGORITHMS[*]

Ravindra K. Ahuja, and James B. Orlin

The purpose of mathematical programming is insight, not numbers.
- A.M. Geoffrion

Hundreds of books catalog optimization algorithms of one kind or another. In addition, hundreds of research papers address applications of optimization algorithms. And yet, experimentation in optimization is a topic that has not received nearly the attention it merits.
- Golden, Assad, Wasil, and Baker

ABSTRACT

In the mathematical programming literature, researchers have conducted a large number of computational studies to assess the empirical behavior of various algorithms and have utilized the CPU time as the primary measure of performance. Generally, CPU times are not a good measure of an algorithm's performance because they are implementation dependent, hard to replicate, and they do not provide much insight about an algorithm's behaviour. In this paper, we illustrate the notion of "*representative operation counts*" that can complement the conventional CPU time analysis and can allow us (i) to identify the asymptotic bottleneck operations in an algorithm, (ii) to estimate its running time for different problem sizes and on various computers without actually implementing it, and (iii) to obtain a fairer comparison of several algorithms. We believe that our ideas can be easily incorporated in an empirical study and can yield valuable insight about algorithms' behaviour.

[*] This paper appears as a chapter in our forthcoming book, Ahuja, Magnanti, and Orlin [1993].

1. INTRODUCTION

The field of mathematical programming is concerned with developing the most "efficient" algorithms for a variety of optimization problems. The notion of efficiency involves all the various computing resources needed for executing an algorithm. However, since time is often a dominant computing resource in practice, we have used computational time as the primary measure for assessing algorithmic efficiency. One important way to measure the computational time of an algorithm is through its worst-case analysis. Worst-case analysis provides upper bounds on the number of steps that a given algorithm can take on *any* problem instance. For a variety of reasons, the worst-case analysis of algorithms is a popular method for judging algorithmic efficiency, and this approach has stimulated considerable research. However, worst-case analysis can be overly pessimistic since it permits "pathological" instances to determine the performance of an algorithm, even though they might be exceedingly rare in practice. Often the empirical behavior of an algorithm is much better than suggested by its worst-case analysis. Consequently, the researchers typically rely upon the empirical testing of an algorithm to assess its performance in practice.

In the operations research literature, researchers have conducted a large number of computational studies to assess the empirical behavior of various algorithms and to determine the "best" algorithms in practice. A typical study tests more than one algorithm for a specific optimization problem and generally consists of the following steps: (i) write a computer program (often in FORTRAN) for each algorithm to be tested; (ii) either use pseudorandom problem generators to generate random problem instances with selected combinations of input size parameters (e.g., the number of variables and constraints) or obtain some real life instances of the problem; (iii) run computer programs and note the CPU (central processing unit) times for the different algorithms on these problem instances; (iv) declare the algorithm that takes the least amount of CPU time as the "winner". If different algorithms are faster for different input size parameters, then report this fact as well.

The existing literature on computational testing has a tendency to over rely on CPU time as the primary measure of performance. The CPU times depend greatly upon subtle details of the computational environment and the test problems such as: (i) the chosen programming language, compiler, and computer; (ii) the implementation style and skill of the programmer; (iii) network generators used to generate the random test problems; (iv) combinations of input size parameters; and (v) the particular programming environment (for example, the use of the computer system by other users). Because of the multiple sources of variabilities, CPU times are often difficult to replicate, which is contrary to the spirit of scientific investigation. Another drawback of the use of CPU time analysis is that CPU time is an aggregate measure of empirical performance, and does not provide much insight about an algorithm's behavior. For example, an algorithm generally performs some fundamental operations repeatedly, and a typical CPU time analysis does not help us to identify its "bottleneck" operations. Identifying bottleneck operations of an algorithm can provide useful guidelines for where to direct future efforts to understand and subsequently improve an algorithm.

The spirit of worst-case analysis is to identify theoretical bottlenecks in the performance of any algorithm and to provide upper bounds on the computation counts of these bottleneck operations as a measure of the algorithm's overall behavior. Borrowing this point of view for computational testing, we might attempt to measure the empirical performance of an algorithm (or its computer implementation) by counting the number of times the algorithm executes each of these bottleneck operations while solving each instance of the problem. That is, we would conduct computer experiments to obtain an actual count of bottleneck operations instead of providing a theoretical upper bound on this number. This approach suggests that in analyzing the empirical behavior of an algorithm, we need not count the number of times it executes each of the (possibly thousands) of lines of code, but instead can focus on a relatively small number of lines that are "summary measures" of the algorithm's empirical behavior. Even for the relatively complex algorithms, we have generally found that we need to keep track of the computation counts of at most 3 or 4 operations or lines of code.

For example, we shall show in Section 4 that in the network simplex algorithm for the minimum cost flow problem, the following three operations dominate all other operations: (i) the number of arcs scanned by the algorithm to identify entering arcs; (ii) the number of arcs in the pivot cycles; and (iii) the number of nodes in the subtrees whose potentials change. Even though any implementation of the network simplex algorithm algorithm might contain hundreds of lines of code, we need to keep track of only three fundamental operations in order to identify the bottleneck operations, and in order to estimate the running time to within a constant factor.

We will soon formalize this notion of "representative operation counts" in computational testing; however, let us first summarize some of the advantages of this approach over the more common approach of using only CPU time analysis.

1. Representative operation counts allow us to identify the asymptotic bottleneck operations of the algorithm, i.e., the operations that progressively take a larger share of the computational time as the problem size increases. (The improvement in the asymptotic bottleneck operation has the maximum impact on the running time of the algorithm for sufficiently large problem sizes.)

2. Representative operation counts provide more guidance and insight for comparisons of two algorithms that are run on different computers, and even permits us to compare algorithms implemented with different computer languages.

3. Representative operation counts facilitate the determination of lower and upper bounds on the asymptotic growth rate in the computation time as a function of the problem size.

4. We can use statistical methodologies to estimate the CPU time on a computer as a linear function of the representative operation counts. We refer to this estimate of the CPU times as the *virtual CPU time*. The virtual CPU time permits researchers to carry out experiments on different computers, but estimate the running times as if all of the experiments had been carried out on a common computer.

This type of asymptotic empirical analysis complements worst-case analysis, which is so popular in the operatins research literature. The empirical analysis using representative operation counts allows us to identify the actual empirical behavior of the algorithm for sufficiently large problem sizes. Just as worst-case analysis ignores constant factors in the running time, empirical analysis using representative operation counts ignores constant factors in the running time, and instead focuses on the dominant term in the computations. This paper explains the underlying ideas for this type of empirical analysis and illustrates it using a well known network simplex algorithm for solving the minimum cost flow problem.

We remark that the field of computational testing is very broad, and we cannot do justice to it in a short chapter. Rather than treat the wide range of topics of importance in computational testing (such as how to select test problems, how to conduct an experimental design, what type of statistical tests are most appropriate, and what data needs to be reported), we will focus on the use of representative operation counts as an aid in the analysis of computational experiments. We believe that it is quite easy to include the counting of representative operations in any computational testing, and the potential payback for keeping track of the added information is substantive.

2. RELATED RESEARCH

This paper focusses on using representative operations counts to analyse the empirical behavior of an algorithm. The idea of using representative operation counts in the empirical analysis of algorithms is quite old, and probably dates back to the origins of computational experiments. Nevertheless the literature on computational testing for mathematical programming has historically used CPU time as its primary measure of computational effort, and has used representative operation counts in a rather limited way. For example, most computational experiments on the network simplex algorithm have kept track of the number of pivots, but have not kept track in any systematic way of the work per pivot. The thesis of McGeoch [1986] is an excellent reference that de-emphasizes the focus on CPU time in favor of representative operation counts. The computational studies by Johnson [1990] and Bentley [1990] provide excellent illustrations of how to use both CPU times and representative operation counts to analyze empirical behavior of algorithms. The purpose of writing this paper is to popularize this idea and enhance its adaptability among researchers engaged in the empirical testings of algorithms.

In this chapter, we have focused on empirical running time as measured both by representative operation counts and by CPU times. Other important measures of performance for an algorithm include: (1) ease of implementation, (2) robustness, (3) reliability, and (4) accuracy of the solutions. The papers by Crowder and Saunders [1980], Hoffman and Jackson [1982], and Greenberg [1990] discuss these measures of performances.

Reporting of the computational results is an integral part of any empirical study of algorithms. The most comprehensive references on the reporting of computational experiments in mathematical programming is Crowder, Dembo, and Mulvey [1978, 1979]. They provide guidelines for what should be reported in a research paper, as well as provide help on how to conduct appropriate computational experiments. Jackson and Mulvey [1978] have summarized the reporting of computational experiments within the mathematical programming literature, largely detailing how poor the reporting had been up to that time. More recently, Jackson, Boggs, Nash, and Powell [1989] have provided updated guidelines for conducting and reporting computational experiments.

Often, statistical methodologies can provide analysis and insight that is unavailable through other means. Some papers on computational testing that provide details on statistical methodologies are due to Bland and Jensen [1985], and Golden, Assad, Wasil and Baker [1986]. (The paper of Golden et al. [1986] is an excellent paper for illustrating a variety of other important topics such as the comparison of different algorithms using multiple criteria, and the different issues that one faces in developing a system for commercial use.) Moreover, statistical methodologies such as *variance reduction* can often increase the power of the analysis and reduce the number of experiments needed to obtain conclusive results. We refer the reader to McGeoch [1992] for excellent illustrations of variance reduction as well as other pointers to the literature.

Algorithm animation is another technique for gaining insight about an algorithm. In his dissertation, Brown [1988] provides an excellent treatment of this topic. Algorithm animation techniques view the progress of an algorithm on a single instance as a sequence of snapshots. For example, consider the greedy algorithm for finding the minimum spanning tree joining n points in the plane. This algorithm creates the minimum spanning tree one arc at a time, maintaining a forest at each intermediate stage until it ultimately creates the minimum cost spanning tree. Using animation, we could construct a sequence of figures showing the forest at intermediate stages. Viewing a motion picture consisting of a series of these snapshots might reveal additional insights into the algorithm. Alternatively, we could see how the total length of the forest is increasing as a function of computer time, or we could see how the size of the components decreases as a function of computer time. The papers by Bentley and Kernighan [1990], and Bentley [1990] discuss a simple language that facilitates the construction of these animations.

3. Representative Operation Counts

In order to formalize our notion of the counting of operations performed by an algorithm, let us first back up and consider what a computer does in executing a computer program. Suppose that A is a computer program for solving some problem, and that I is an instance of the problem. The computer program consists of a finite number of lines of computer code, say $a_1, a_2, \dots, a_K$.

Each line of code gives either one or a small number of instructions to the computer. The instruction might tell the computer to carry out an arithmetic operation on a register or to move data from one memory location to another; or it might be a control instruction informing the computer which line of code it should execute next. For convenience, we assume that the program is written so that each line of the code gives O(1) instructions to the computer, and that each instruction requires O(1) units of time. We assume that the fastest computer operation requires 1 time unit. (Although these assumptions are reasonable, they do imply some restrictions; for example, we do not permit lines of code that tell the computer to add two vectors as would be allowed in some high level languages such as APL. Rather, we would require that the adding of vectors be carried out as a loop that sums the two vectors one component at a time.) The assumption that each operation executed by the computer requires a comparable amount of time seems reasonable in practice with the notable exceptions of input-output, caching and paging.

The preceding discussion implies that executing any line of code requires O(1) time units, and at least 1 time unit. Therefore, each line of code requires $\theta(1)$ time units since its execution time is bounded from both above and below by a constant number of units. (If $f : Z^+ \rightarrow R^+$ and $g : Z^+ \rightarrow R^+$, we say that $f(n) = \theta(g(n))$ if $c_1 f(n) \le g(n) \le c_2 f(n)$ for all $n \ge n_0$ and some constants c_1, c_2, n_0.) Suppose that the computer code we are investigating has K lines of code. For a given instance I of the problem, let $\alpha_k(I)$, for k = 1 to K, be the number of times that the computer executes line k of this computer program. Let CPU(I) denote the CPU time of the computer program on instance I. The preceding discussion implies the following lemma.

Lemma 1. *$CPU(I) = \theta(\sum_{k=1}^{K} \alpha_k(I))$.* ♦

Lemma 1 states that we can estimate the running time of an algorithm to within a constant factor by counting the number of times it executes each line of code. However, counting each line of code is unnecessarily burdensome. As we shall soon see, it really suffices to count a relatively small number of lines of code. For example, consider the following fragment of code:

```
for i = 1 to 3 do
begin
    A(i) := A(i) + 1;
    B(i) := B(i) + 2;
    C(i) := C(i) + 3;
end;
```

We need not count the number of times the algorithm executes the statement "B(i) := B(i) + 2", since it executes this statement whenever it executes the statement "A(i) := A(i) + 1". Similarly, we need not count the number of times the algorithm executes the statement "C(i) := C(i) + 3". Moreover, it appears that regardless of the size of the problem solved, the algorithm modifies each A, B, and C three times during the execution of the "for loop". So, in this case, it would suffice to keep track of just the number of times that the algorithm executes the "for" loop .

Note that we have treated the number of iterations of the do loop as a constant. Suppose, instead, that the first line of the do loop were "**for** i = 1 to 10 **do**." Should we also treat the 10 as a constant? What if the first line of the do loop were "**for** i = 1 to 10000 **do**." Should we continue to treat the 10000 as a constant? In answering these questions, we might invoke two rules of thumb. First, we should determine the representative operation counts after expressing the algorithm as a pseudocode with all the problem parameters treated explicitly as parameters and not as constants. For example, suppose that U denotes an upper bound on arc capacities for network flow problems. Then, although in most practical situations we can bound log U as being less than 16 (i.e., $U \le 2^{16}$), in pseudocode we should use the term "log U" rather than the constant 16. We would not treat log U as a constant. Second, as a rule, we should not treat the number of iterations of a "do loop" or a "while loop" or any other loop as a constant since frequently the number of times that the program will call the loop is a problem parameter rather than a constant.

We now formalize the notion we have been suggesting; that is, keeping track of a small number of lines of code, which we call the "representative operation counts." Let S denote a subset of {1, . . . , K}, and let a_S denote the set $\{a_i : i \in S\}$. We say that a_S is a representative set of lines of code of a program if for some constant c,

$$\alpha_i(I) \le c\,(\Sigma_{k \in S}\, \alpha_k(I)),$$

for every instance I of the problem and for every line a_i of code. In other words, the time spent in executing line a_i is dominated (up to a constant) by the time spent in executing the lines of code in S. With this definition, we have the following corollary to Lemma 1

Property 2. *Let S be a representative set of lines of code. Then* $CPU(I) = \theta(\Sigma_{k \in S}\, \alpha_k(I))$.

Proof. By Lemma 1, $CPU(I) = \theta(\Sigma_{k=1}^{K}\, \alpha_k(I))$. Moreover, for each line of code not in S, $\alpha_i(I) \le c\,(\Sigma_{k \in S}\, \alpha_k(I))$, and so $\Sigma_{k=1}^{K}\, \alpha_k(I) = \theta(\Sigma_{k \in S}\, \alpha_k(I))$. ♦

This methodology identifies a set of representative lines of code, and during the empirical investigations keeps track of the representative operation counts for each instance solved. Sometimes, the selected representatives will not refer specifically to one line of code. The representative operation might be described over several lines of code. Rather than use the expression "the computation counts for a set of representative lines of code" we will refer to "the computation counts for a set of representative operations" or more briefly as the *representative operation counts.*

In the next section, we discuss various uses of representative operation counts. We point out that in addition to noting representative operation counts for various instances solved, we might keep track of some other counts that might be helpful in assessing an algorithm's behavior. For example, in the network simplex algorithm, for each instance solved we might note the number of major iterations that the algorithm performs, such as the number of pivots of the network simplex algorithm.

At first glance, we might suspect that determining a representative set of operations could be difficult, and that the set might be quite large. But our experience suggests that it is generally quite easy to determine representative sets, and often there are a wide number of possible correct choices. In addition, the representative sets are typically quite small. We now give examples of representative sets for several popular algorithms. We point out that the representative sets, we indicate for the algorithms, are not unique. In some cases, we have suggested additional operation counts that might be of value in empirical investigations. Most of the algorithms analysed by us are network flow algorithms because they are simple and illustrate our ideas nicely; however, they do require some familiarity with network notation and network flow algorithms. For additional details of these network flow algorithms, we refer the reader to Ahuja, Magnanti, and Orlin [1993]. We, however, provide some essentials definitions next. We consider a directed network G = (N, A), where N represents the node set and A denoted the ar set. Let $n = |N|$ and $m = |A|$. Each arc (i, j) has an associated cost c_{ij} and an associated capacity u_{ij}. Let C denote the largest arc cost and U denote the largest arc capacity in the network.

Dijkstra's algorithm for the shortest path problem (Version I). In the shortest path problem, we wish to obtain a shortest path from a specified source node s to every other node in the network. Dijkstra's algorithm (see, for example, Lawler [1976]) can be used to solve this problem provided all arc lengths are nonnegative. Dijkstra's algorithm maintains a distance label d(i) for each node i in the network, which is either temporary or permanent. At each iteration, the algorithm identifies a temporary labeled node i (*node selection step*) by scanning all temporary labeled nodes, makes it permanent, and scans the arcs in its adjacency list A(i) to update the distance labels of adjacent nodes (*distance update step*). The algorithm terminates when all nodes are permanently labeled. A natural set of representative operations for Dijkstra's algorithm are : (i) the number of temporary labeled nodes scanned during the node selection step; and (ii) the number of arcs scanned to update distance labels. But notice that the operations performed by Dijksta's algorithm in the node selection operations is $n + (n-1) + (n-2) + \ldots\ldots + 1 = \Omega(n^2)$, and the number of operations performed in the distance update operations is $\sum_{i \in N} |A(i)| = \Omega(m)$. Thus the running time of the algorithm is same for same sized problems irrespective of the problem instance. Consequently, keeping track of the representative operation counts will not give us any additional insight about the algorithm.

Dijkstra's algorithm for the shortest path problem (Version II). In this version of Dijkstra's algorithm, we wish to detrmine a shortest path from a specified source node s to another specified sink node t, where c_{ij} denotes the length of arc (i, j) In this case, Dijkstra's algorithm works as earlier in Version I, except that we terminate the algorithm as soon as the sink node is permanently labeled. Observe that in this version, the two representative operation counts will differ from instance to instance and keeping track of them may provide useful information about the algorithm. In addition, it might be useful to keep track of the number of nodes permanently labeled before termination, and the number of arc scans when the distance labels strictly decrease.

Label correcting algorithm for the shortest path problem. Label correcting algorithms are an important class of algorithms to solve the shortest path problem with arbitrary arc lengths. The generic label correcting algorithm maintains a distance label d(i) with each node i, which is the length of some path from the source node to node i. If, in addition, these distance labels satisfy the optimality conditions $d(j) \leq d(i) + c_{ij}$ for each arc $(i, j) \in A$, then they represent the shortest path distances; otherwise they are non-optimal. The generic label correcting algorithm also maintains a set LIST of nodes satisfying the property that whenever an arc (i, j) violates its optimality condition, then node i is in the LIST. At each iteration, the label correcting algorithm removes a node i from LIST, scans every arc (i, j) in its adjacency list A(i) one by one, and if the arc satisfies $d(j) > d(i) + c_{ij}$, then it updates $d(j) = d(i) + c_{ij}$, and adds node j to LIST. The algorithm terminates when LIST is empty.

The generic label correcting algorithm performs the following operations repeatedly: (i) it removes nodes in LIST (*removals*); (ii) it scans adjacency lists of nodes removed from LIST (*arc scans*); and (iii) adds arcs to LIST (*additions*). Using the facts that the number of removals equals the number of additions, and that whenever a node i is added we must scan every arc (i, j) in A(i), we conclude that arc scans are the representative operation for this algorithm. Consequently, the number of arcs scanned can be used to assess the empirical behavior of the algorithm. In addition, it might be useful to keep track of the number of additions to the LIST.

Labeling algorithm for the maximum flow problem. The labeling algorithm starts with a zero flow and proceeds by identifying augmenting paths from the source node s to the sink node t and sending flows along these paths. (An augmenting path is a directed path along which positive amount of flow can be sent from the source node to the sink node.) The algorithm identifies an augmenting path using a search method that works in the following manner. The method maintains a set, LIST, of labeled nodes (i.e., the nodes reachable from the source) that are yet unexamined. Initially, LIST contains the singleton node s. Subsequently, at each iteration, the method removes a node i from LIST, scans arcs in its adjacency list A(i) to label more (unlabeled) nodes and adds them to the LIST. The method terminates when either the sink node t is labeled or LIST becomes empty. In the former case, we have identified an augmenting path; in this case, we augment a maximum possible flow along this path, unlabel all nodes, and reapply the search method. In the latter case, the current flow is maximum and the labeling algorithm terminates.

The labeling algorithm performs the following operations repeatedly: (i) removing nodes from LIST; (ii) scanning arcs in the adjacency lists of nodes removed from LIST; (iii) adding nodes to LIST; (iv) sending flows along augmenting paths; and (v) unlabeling the labeled nodes. It can be easily shown that the representative operation for the labeling agorithm is the arcs scanned while examining labeled nodes, as this operation dominates all other operations that the labeling algorithm performs. In addition to this representative operation, we would probably want to count the number of augmentations. This additional information might provide insight for comparing the labeling algorithm to other maximum flow algorithms. For example, we could check whether the labeling algorithm requires more augmentations than other augmenting path algorithms.

Bubble sort algorithm. The bubble sort algorithm is a well known algorithm to sort the elements of an array α of n elements in nondecreasing order. This algorithm proceeds by performing passes over the array α. In the l^{th} pass, it compares the two elements $\alpha(k)$ and $\alpha(k+1)$ for each $1 \le k \le l-1$ one by one, and if the two elements are out of order (i.e., $\alpha(k) > \alpha(k+1)$), it swaps their positions. The algorithm terminates when during an entire pass, no two elements are found to be out of order. The representative operation in this algorithm is the number of comparisons required to check whether the adjacent elements are out of order. In addition, we might keep track of the number of passes performed by the bubble sort algorithm.

Network simplex algorithm. The network simplex algorithm is a well known algorithm to solve the minimum cost flow problem. The network simplex algorithm is an adaptation of the revised simplex algorithm and maintains a basic feasible solution at each step, which corresponds to a spanning tree in the underlying network. For a basic feasible solution with a corresponding tree T, the agorithm scans through the arc list to identify a nontree arc (k, l) violating its optimality conditions. Adding arc (k, l) to the current spanning tree creates a unique cycle W. The algorithm identifies this cycle and augments a maximum possible flow along this cycle. Doing so saturates at least one arc, say (p, q), in the cycle W. The algorithm replaces the arc (p, q) by the arc (k, l), and updates the node potentials (or, simplex multipliers) of a subtree T_2 of T and the tree indices. The network simplex algorithm repeats this process until all arcs satisfy the optimality conditions.

The network simplex algorithm has the following set of representative operations: (i) the number of arcs scanned by the algorithm to identify entering arcs; (ii) the number of arcs in the pivot cycles; and (iii) the number of nodes in the subtrees whose potentials change. The network simplex algorithm performs many other operations. We do not need to keep track of these operations because these three operation counts dominate them. For example, the algorithm performs some initialization operations that we have ignored. Further, to update the tree indices requires $\theta(|W| + |T_2|)$ time, which is dominated by operations (ii) and (iii). This set of representative operations is strikingly compact considering the fact that implementations of the network simplex algorithm are typically quite intricate. In addition to these representative operations, we might keep track of other operations. For example, we would most likely want to keep track of the number of pivots. Moreover, if we were concerned about the effects of degeneracy, we would also keep track of the number of degenerate pivots performed by the algorithm.

Euclid's algorithm. Euclid's algorithm finds the greatest common divisor of two numbers, say α and β. We first order the numbers so that the larger number occurs first. Suppose that α is larger than β. Then, at each iteration, we replace the larger number α by the smaller number β and replace the smaller number β by α mod β. For example, if the original numbers are (102, 93), then the numbers after the first iteration will be (93, 9). We repeat these operations until the smaller number becomes zero. The larger number at this point is the greatest common divisor of α and β.

A set of representative operations for this algorithm is the single operation of α mod β. However, it is possible that one may modify this set. For example, if α and β are quite large, then we may store these numbers in several words. In that case, the set of representative operations will be the number of word operations involved in the divisions.

4. APPLICATION TO NETWORK SIMPLEX ALGORITHM

In this section, we illustrate the use of representative operation counts using the network simplex algorithm for the minimum cost flow problem. We provide experiments based upon the network simplex algorithm using the first negative pivot rule for the selection of the entering arc. The first negative pivot rule scans the arc list in a wraparound fashion and selects the first arc violating its optimality condition as the entering arc. The code for the network simplex algorithm was developed by the authors. To generate minimum cost flow problems, we used the well known network generator NETGEN developed by Klingman, Napier and Stutz [1974]. For additional codes and computational testings of the network simplex algorithm, we refer the readers to the studies of Glover, Karney and Klingman [1974], Mulvey [1978], Bradley, Brown and Graves [1977], Grigoriadis [1986], and Chang and Chen [1989].

The computational time that the network simplex algorithm requires to solve a minimum cost flow algorithm depends on a number of different parameters, including: the number of nodes, the number of arcs, the network generator, the size of the cost and capacity data, and the number of supply nodes. Of these factors, to illustrate the approach discussed in this chapter, we have chosen to focus on the number of nodes and the number of arcs. We conducted experiments on networks with 1,000, 2,000, 4,000, 6,000 and 8,000 nodes. For each choice of n nodes, we created networks with 2n, 4n, 6n, 8n, and 10n arcs. Setting $d = m/n$, we have considered networks with $d = 2, 4, 6, 8$ or 10. Thus, the largest network size we considered had 8,000 nodes and 80,000 arcs. For each specific setting of n and m, we solved 5 different problems generated from NETGEN and computed the average of these 5 problem instances. (The averages have a lower variability of outcomes than the individual tests, and so the resulting graphs and charts reveal patterns more clearly.) For each problem that we solved, we noted the following values:

(i) α_E : arcs scanned in selecting an entering arc (summed over all pivots);

(ii) α_W : number of arcs of the pivot cycles (summed over all pivots);

(iii) α_P : node potentials modified (summed over all pivots);

(iv) p : the number of pivots, further decomposed into the number of degenerate pivots and the number of nondegenerate pivots; and

(v) τ : the CPU time to execute the algorithm (times noted on a HP9000/850 computer under a multiprogramming and multisharing environment.)

We computed the averages of these values over five problem instances for each parameter setting. Figure 1 gives these averages for the network simplex algorithm with the first eligible arc pivot rule.

S. No.	n	m	d	α_E	α_W	α_P	p	$\alpha_E+\alpha_W+\alpha_P$	CPU time
1	1000	2000	2	41590	183540	237599	4721.4	462729	4.46
2	2000	4000	2	129125	831593	1356773	15358	2317491	21.62
3	4000	8000	2	406115	3989374	8319325	51936.2	12714814	119.7
4	6000	12000	2	863573	10770317	25045320	116303	36679210	345.64
5	8000	16000	2	1453942	21438864	54392393	197005	77285199	699.88
6	1000	4000	4	87883	284830	403699	10275.6	776412	7.31
7	2000	8000	4	248760	1101807	2090021	29879.8	3440588	31.26
8	4000	16000	4	765758	5171901	12341060	99858	18278719	160.7
9	6000	24000	4	1475919	12277824	33993925	197858	47747668	420.96
10	8000	32000	4	2419120	24614630	77921501	335296	104955251	853.21
11	1000	6000	6	141475	351763	518265	14372.2	1011503	9.33
12	2000	12000	6	379171	1253222	2353123	38837.4	3985516	35.42
13	4000	24000	6	1152766	5655431	15215392	124249	22023589	188.55
14	6000	36000	6	2329115	14249457	42841133	259742	59419705	505.79
15	8000	48000	6	3639711	26157289	90280080	416123	120077080	978.57
16	1000	8000	8	207114	380015	598636	16586	1185765	10.7
17	2000	16000	8	529747	1452365	3012190	47116.8	4994302	42.61
18	4000	32000	8	1534355	5919861	13833227	139434	21287443	200.67
19	6000	48000	8	2918853	13917221	48131763	273089	64967837	518.08
20	8000	64000	8	4532492	25791836	99057203	437558	129381531	1033.05
21	1000	10000	10	266541	410304	739070	18634.8	1415915	12.39
22	2000	20000	10	710258	1555660	3305765	53425.4	5571683	48.08
23	4000	40000	10	1975386	5844629	17922439	145303	25742454	208.3
24	6000	60000	10	3902416	15193494	48615853	312355	67711763	584.34
25	8000	80000	10	5942548	26724636	103238910	487624	135906094	1064.45

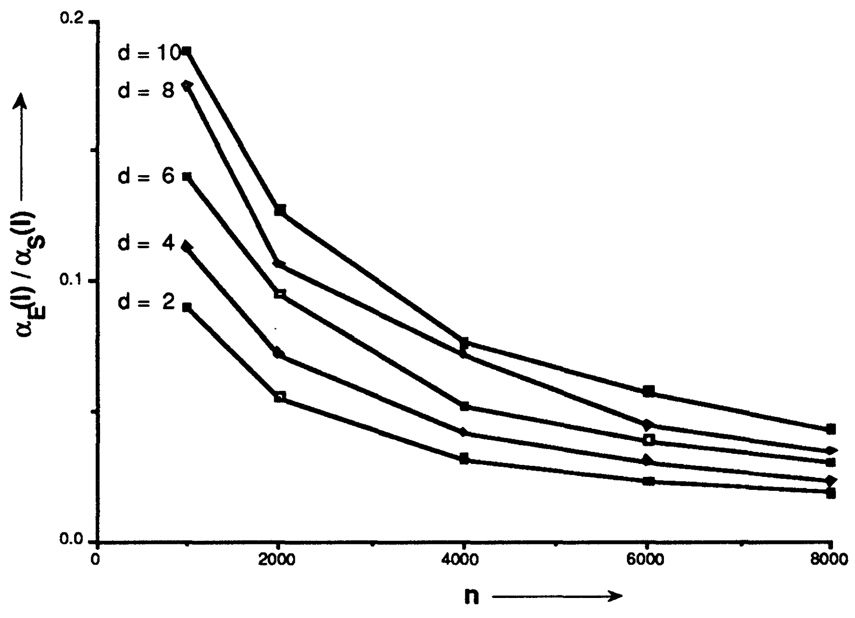
Figure 1. A table of computation counts for the network simplex simplex algorithm with the first entering pivot rule.

Identifying Asymptotic Bottleneck Operations

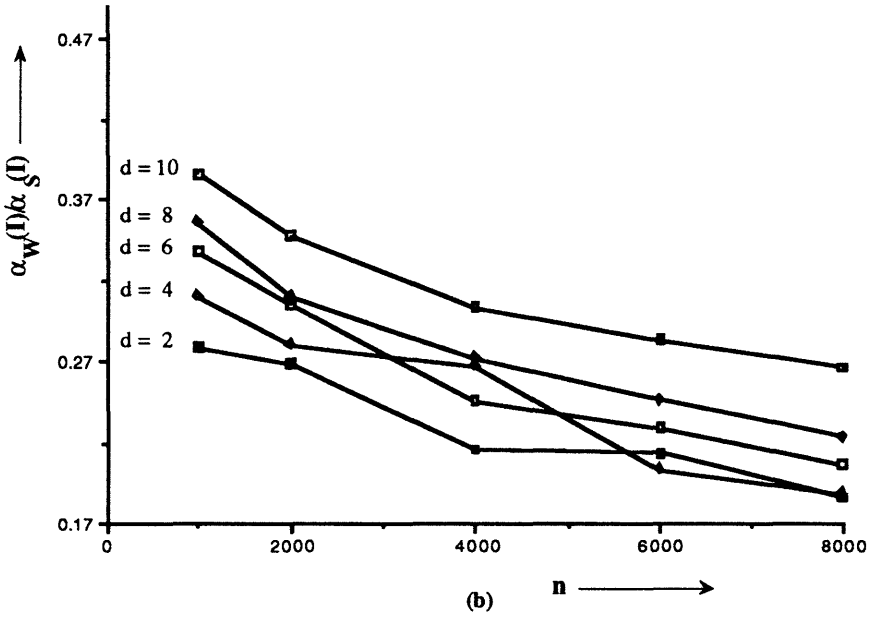
We consider an operation to be a "bottleneck operation" for an algorithm if the operation consumes a significant percentage of the execution time on at least some fraction of the problems tested. We refer to an operation as an *asymptotic nonbottleneck operation* if its share in the computational time becomes smaller and approaches zero as the problem size increases. Otherwise, we refer to an operation as an *asymptotic bottleneck operation.* The asymptotic bottleneck operations are important because they determine the running times of the algorithm for sufficiently large problem sizes.

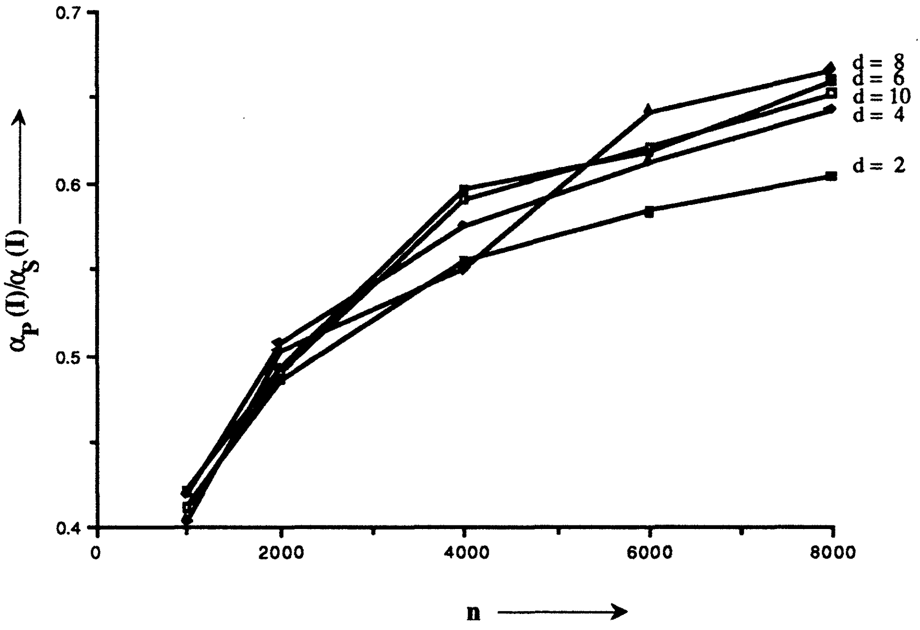
We have earlier shown that the representative operation counts provide both upper and lower bounds on the number of operations an algorithm performs; therefore, the set of representative operations must contain at least one asymptotic bottleneck operation. There is no formal method for determining asymptotic bottleneck operations using computational testing, unless we are willing to impose further assumptions on the behavior of an algorithm on large problems. After all, it is theoretically possible that an algorithm behaves one way for problems of sufficiently large size, and it behaves totally differently for small size problems. Nevertheless, there are procedures that seem quite effective in practice at determining asymptotic bottlenecks, even if they are not totally rigorous.

We illustrate how to find an asymptotic bottleneck operation for the network simplex algorithm. Let $\alpha_S(I) = \alpha_E(I) + \alpha_W(I) + \alpha_P(I)$. Then, we simply plot $\frac{\alpha_E(I)}{\alpha_S(I)}, \frac{\alpha_W(I)}{\alpha_S(I)}, \frac{\alpha_P(I)}{\alpha_S(I)}$ for increasingly larger problem instances I and look for a trend. Figure 2 gives these plots for the network simplex algorithm with the first eligible pivot rule. In these plots, we use the number of nodes as a surrogate for the problem size and give a plot for each different density d = m/n. This choice helps us to visualize the effect of n and m on the growth in representative operation counts. These plots suggest that updating node potentials is an asymptotic bottleneck operation in the algorithm.



(a)





(c)

Figure 2. Identifying asymptotic bottleneck operations.
(a) Plot of the ratio of arc scan operations and total number of operations.
(b) Plot of the ratio of flow update operations and total number of operations.
(c) Plot of the ratio of potential update operations and the total number of operations.

Estimating Growth Rates of Bottleneck Operations

How should we estimate the growth in the computational time (i.e., CPU time) as the problem size increases? Instead of directly estimating the growth in the computational time, we estimate the growth in the asymptotic bottleneck operations. By focusing only on the asymptotic bottleneck operations, we eliminate the contribution of the nonbottleneck operations, and therefore, the estimates should be more reliable. In the notation of the previous section, the asymptotic running time is proportional to $\lim_{|I| \to \infty} \max(\alpha_1(I), \alpha_2(I), \ldots, \alpha_K(I))$. In the manner that we have selected the representative operations, the asymptotic running time is also proportional to $\lim_{|I| \to \infty} \max(\alpha_i(I) : i \in S)$. There is no need to consider the nonbottleneck operations in estimating the asymptotic running time. The CPU time is a linear function of the terms $\alpha_i(I)$ for i = 1 to K, and thus includes information from many of the nonbottleneck operations. For this reason, the CPU time is a much "noisier" estimator of the asymptotic running time.

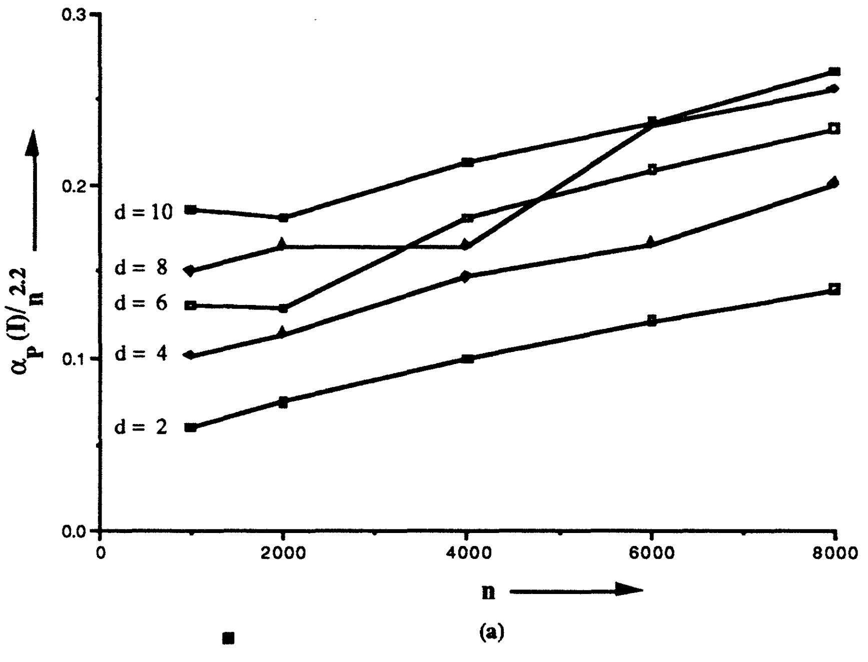
We describe a simple approach for estimating the growth rates of a bottleneck operation, which in our illustration is α_P, the number of potential updates. Our approach consists of the following steps:

Step 1. Determine an appropriate functional form for estimating the counts for the operation α_P. In our example, we choose the functional form to be n^γ for some choice of a growth parameter γ. (A more common approach would be to choose a function of both n and m; however, we will consider each network density separately, and for each network density d = m/n, the ratio of m to n is fixed. Therefore, a function of m and n reduces to a function of n.)

Step 2. Select a candidate lower bound n^l and a candidate upper bound n^u on the growth rate using one of several possible methods.

Step 3. Evaluate (using some methodology) whether n^l and n^u are really the lower and upper bounds on α_P. If not, then return to Step 2 and repeat the process.

For our illustration, we guessed a candidate lower bound of $n^{2.2}$ on the growth rate of the number of potential updates α_P and guessed a candidate upper bound of $n^{2.8}$. Figure 3 gives a plot of $\frac{\alpha_P}{n^{2.2}}$ and $\frac{\alpha_P}{n^{2.8}}$. In Figure 3(a), we find that the function $\frac{\alpha_P}{n^{2.2}}$ has an increasing trend with the problem size, suggesting that that $n^{2.2}$ is indeed a lower bound on the number of potential updates. Further, we find in Figure 3(b) that the function $\frac{\alpha_P}{n^{2.8}}$ has a decreasing trend suggesting that $n^{2.8}$ is an upper bound on the number of potential updates. If we want more refined lower and upper bounds, then we carry out the technique further to see if $n^{2.4}$ is a valid lower bound or if $n^{2.6}$ is a valid upper bound. The polynomial estimates for the lower and upper bounds are significantly different because our method for estimating lower and upper bounds typically is quite conservative, and underestimates the lower bound and overestimates the upper bound.



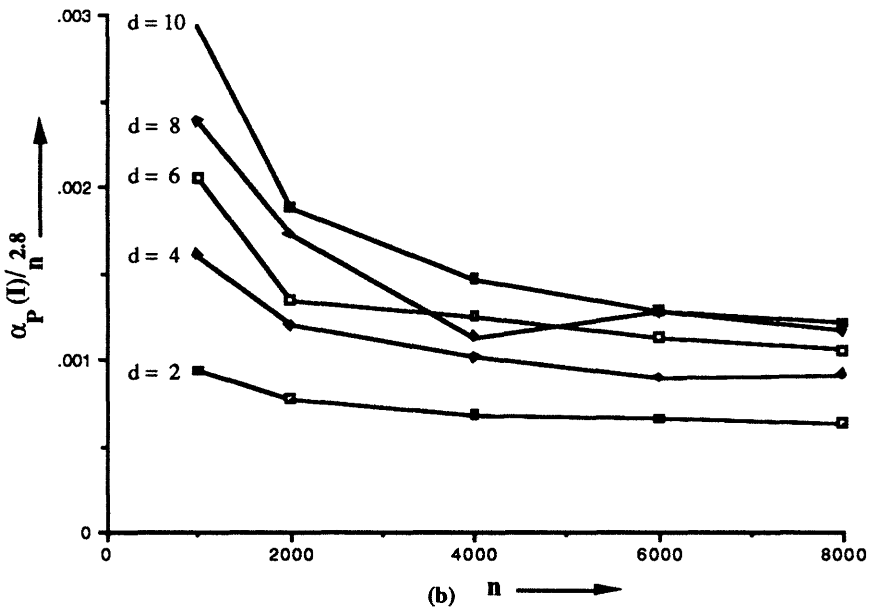


Figure 3. Determining asymptotic growth rates of the bottleneck operations.

We emphasize here that our approach is intended primarily for gathering insight into the upper and lower bounds on function growth, and is not a rigorous approach. For example, the upper bound on growth relies on the user of the methodology recognizing that a certain function has an increasing trend. However, "increasing trend" is not a rigorously defined concept, and depends to an extent on a judgment call of the user.

In addition to the lack of rigor, the methodology, as we have used it, is perhaps too conservative, and can be strengthened through careful statistical analysis. To illustrate, in the previous example, we estimated the growth in the number of potential updates as being between $n^{2.2}$ and $n^{2.8}$. The gap between these bounds is very large, and a careful statistical analysis should be able to narrow the gap considerably.

Comparing Two Algorithms

Suppose that we want to compare two different algorithms $\mathbf{AL}_1$ and $\mathbf{AL}_2$ for solving the same problem and are interested in knowing which algorithm performs better asymptotically. We can apply the same methodology for assessing asymptotic bottleneck operations to compare different algorithms. Let $\alpha_S^1(k)$ and $\alpha_S^2(k)$ be the total expected number of operations performed by the algorithms $\mathbf{AL}_1$ and $\mathbf{AL}_2$ on instances of size k. We say that the algorithm $\mathbf{AL}_1$ is asymptotically superior to the algorithm $\mathbf{AL}_2$ if

$$\lim_{k \to \infty} \frac{\alpha_S^1(k)}{\alpha_S^2(k)} = 0.$$

Virtual Running Times

In Section 1, we have already mentioned some difficulties that arise when we use CPU times as an empirical measure for computational investigations. Here we mention yet one more disadvantage. The CPU times force unnecessary rigidity in the testing of an algorithm. To collect the CPU times for an algorithm, we should conduct all of our tests on the same machine using the same compiler. Moreover, if the machine is a time sharing machine, then we should ideally solve all the test problems when the machine has a similar workload. Moreover, the CPU times for large instances might be dominated by "paging," i.e., the memory management of the secondary storage space.

We can overcome some of these drawbacks using virtual times instead of CPU times. (We point out the virtual time is not directly related to the virtual memory.) The virtual running time of an algorithm is a linear estimate of its CPU time obtained by using its representative operation counts. The term "virtual time" appears in the thesis of Brown [1988], and is used in a similar way in this paper. For example, the virtual running time V(I) of the network simplex algorithm to solve instance I is given by

$$V(I) = c_5 \alpha_E(I) + c_6 \alpha_W(I) + c_7 \alpha_P(I),$$

where c_5, c_6 and c_7 are constants selected so that V(I) is the best possible estimate of the CPU times given by the algorithm's actual running time CPU(I) on the instance I. One plausible way to determine the constants c_5, c_6 and c_7 is to use (multiple) regression analysis. To do so, we consider the points (CPU(I), $\alpha_E(I)$, $\alpha_W(I)$, $\alpha_P(I)$) generated by solving various individual instances and use regression analysis to determine the constants c_5, c_6 and c_7 that minimizes the expression $\sum_I (\mathrm{CPU}(I) - V(I))^2$.

In our regression analysis, we generated each of the points (CPU(I), $\alpha_E(I)$, $\alpha_W(I)$, $\alpha_P(I)$) by taking an average of 5 problem instances; we used the averages reported in Figure 1, and so we found the best linear fit to the 5-point averages. Using regression analysis, we found that for the network simplex algorithm with the first eligible pivot rule, we could estimate the virtual running time of an instance I as follows:

$$V(I) = (\alpha_E(I) + 2\alpha_W(I) + \alpha_P(I))/69000.$$

To obtain an idea of the goodness of this fit, in Figure 4 we plot the ratio $\frac{V(I)}{CPU(I)}$ for all the data points. We find that in 15 out of 25 cases, the error is less than 3%. In each case, the error is at most 7%. So for this example, we can use the virtual running time instead of the CPU time with remarkably little loss of accuracy.

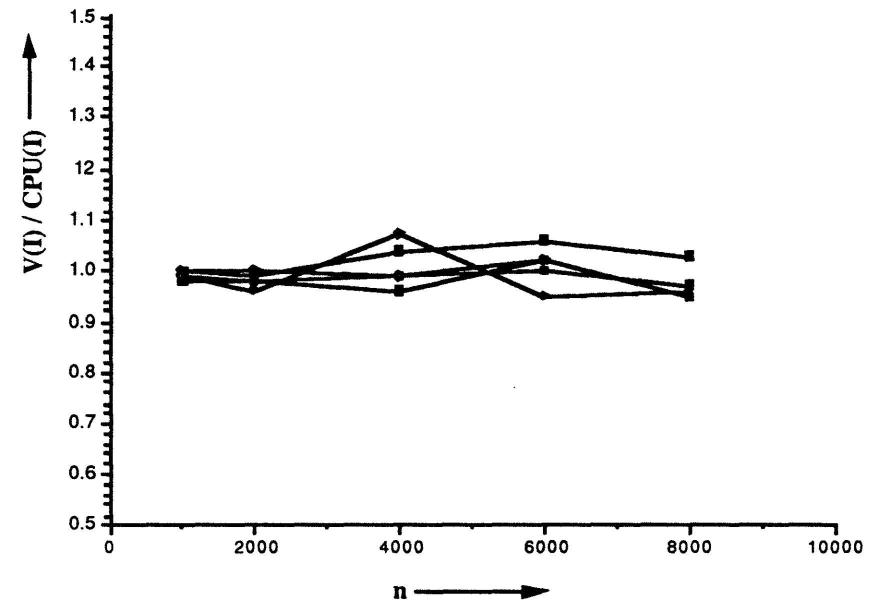


Figure 4. Determining how well the virtual running time estimates the CPU time.

Using virtual running times has several advantages. First, the virtual running time helps us to assess the proportion of time that an algorithm spends on different representative operations. As an immediate consequence, it helps us to identify the bottleneck operations for different sizes of problem instances, and not only the asymptotic bottleneck operation. For example, we estimated the virtual running time of the network simplex algorithm as V(I) = $(\alpha_E(I) + 2\alpha_W(I) + \alpha_P(I))/69000$. To estimate the proportion of time spent of potential updates, in Figure 5 we plot $\alpha_P(I)/(\alpha_E(I) + 2\alpha_W(I) + \alpha_P(I))$. As we can see, for small problems, the potential updates is not the bottleneck operation; as n increases, however, the proportion of the time taken by the potential updates increases as well.

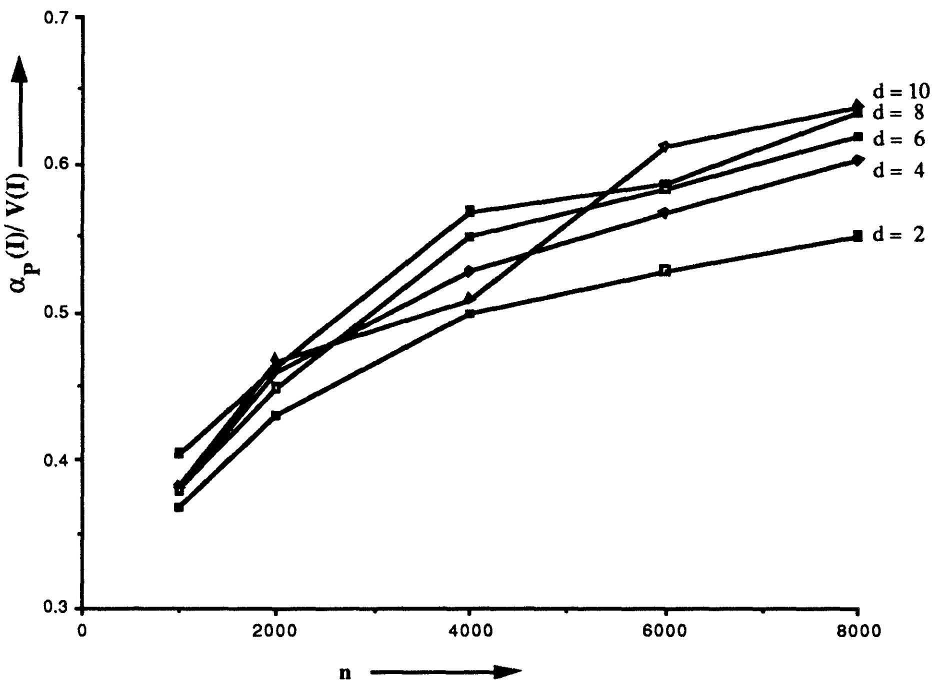


Figure 5. Identifying the percentage of the virtual running time accounted for by the potential updates.

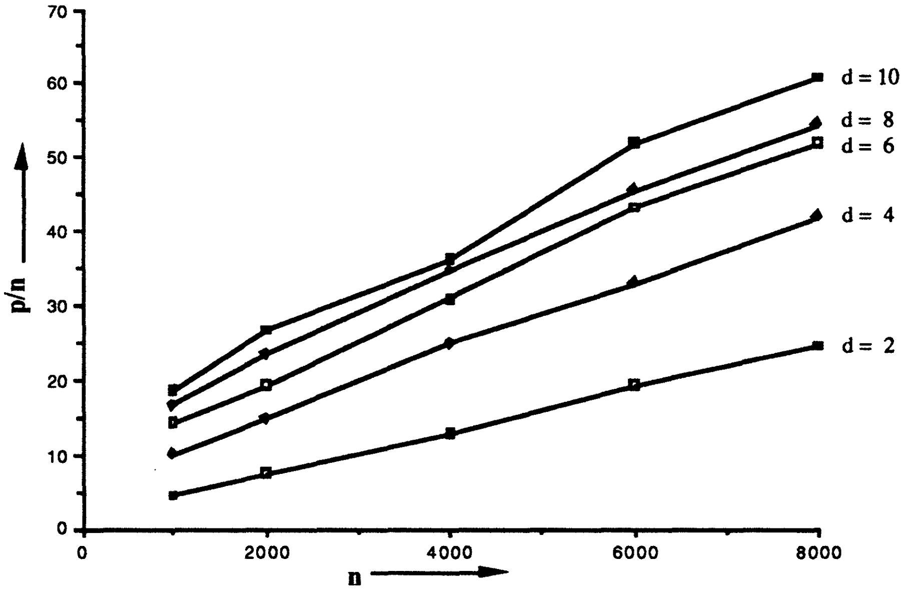
A second advantage of the virtual running time is that it is particularly well suited for situations in which the testing is carried out on more than one computer. This situation might arise for several reasons: for example, several users might be conducting computational experiments at different sites; or the same user might wish to conduct additional tests after upgrading from an old computer. This situation is very common in the research literature because authors often conduct additional experiments at the suggestion of a referee or editor. When we move from one computer system to another, the representative operation counts remains unchanged. Using the representative operation counts of the previous study and the constants of the new computer system (obtained in the regression estimate for the virtual running time), we can obtain the virtual running times for all problems of the previous study measured in terms of the new computer system.

The third advantage of the virtual running times is that it permits us to eliminate the effect of "paging" and "caching" in determining the running times. When a computer program executes a large program whose data does not fully fit into the computer's primary memory (e.g., RAM), it stores part of the data in the secondary memory (e.g., disk) with higher retrieval times. As a result, large programs run slower. In the virtual running time analysis, if we evaluate the constants using small sized problems (that fully execute in the primary memory), then the virtual running times for large sized problems would be as though the program is entirely run in the primary memory. Thus the virtual running times can be used to estimate the running time of an algorithm on a computer with sufficiently large primary memory.

A natural question is whether the constants used in the virtual running time are robust, i.e., do they give an accurate estimate of the CPU times for all possible problem inputs. Again, our use of virtual CPU times as stated here is not fully rigorous, but our limited experience so far has suggested that they do seem to be robust. In practice, we can also measure the robustness of the constants using statistical analysis.

Additional Insight of Algorithms

We can also use computation counts to gain additional insight concerning an algorithm. For example, we can use the preceding analysis to estimate the number of pivots performed by the network simplex algorithm. In the linear programming literature, it is widely believed that for most pivoting rules, the number of pivots is typically proportional to m and rarely more than 3m. Let us try to verify whether this result is valid for the first eligible pivot rule as applied to our randomly generated problems. Notice that this rule would imply that if we plot the ratio of (number of pivots)/n for different network densities d, then we should obtain nonincreasing functions. The plots given in Figure 6(a) indicate that this conjecture is not true and the number of pivots, for a fixed d, is not bounded by a linear function of n. However, if we plot (number of pivots)/n^2 for different network densities, then as indicated by Figure 6(b), we do obtain nondecreasing functions. Therefore, the growth rate of the number of pivots is bounded from above by n^2 and bounded from below by n.



(a)

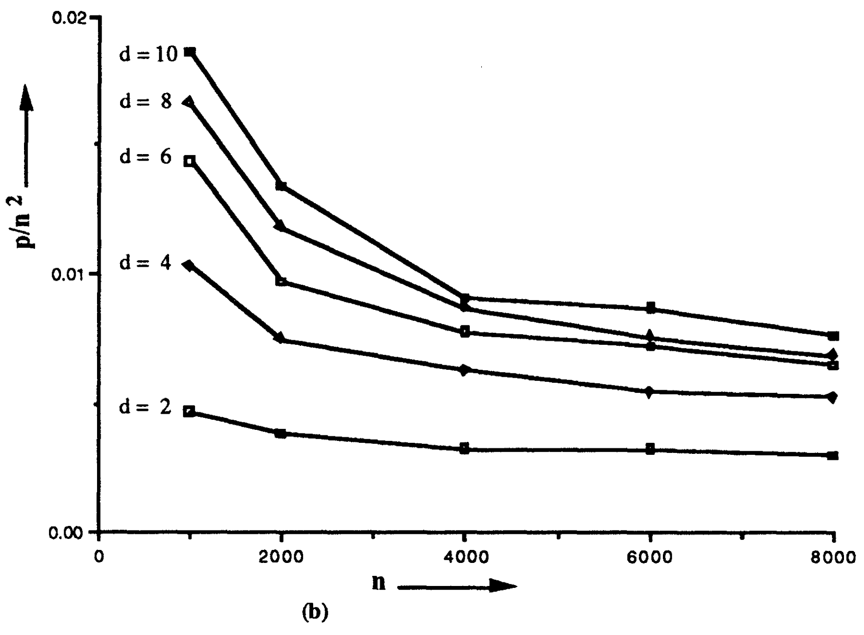


Figure 6. Determining lower and upper bounds on the growth rate of the number of pivots.

We might also be interested in determining what percentages of pivots are degenerate as the network size grows. The data given in Figure 1 indicates that the ratio of degenerate pivots to total pivots varies between 70% and 90%. We might plot a few more graphs to gain additional insight about the network simplex algorithm. For example, we might plot the average size of the tree whose node potentials change during the pivot operation (i.e., $\alpha_p(I)/p$). This plot would give us a good estimate of the running time per pivot since the potential updates is the bottleneck operation, at least for the first entering pivot rule.

In addition, we might be interested in the convergence results of the network simplex algorithm, i.e., how quickly does the network algorithm converge to the optimal objective function value? Does the algorithm quickly obtain a solution with a near optimal objective function value and then slow down, or does it slowly approach the optimal objective function value and then converges rapidly. We could, in principle, provide a partial answer to this question by selecting a few sufficiently large instances and plotting the objective function values as a function of the number of pivots.

5. LIMITATIONS

The methodology discussed in this paper is broadly applicable to sequential algorithms. We can always determine a representative set of operations since the set of all operations is itself a representative set. In addition, our experience suggests that the representative set of operations is typically quite small. Consequently, our methodology can be used to obtain valuable information about an algorithm's behaviour and the comparative performance of different algorithms. However, our methodology has some limitations and one should be cautious in deriving conclusions. Here we list some of the important limitations.

1. Our methodology ignores constants (except while computing the virtual running times). The methodology typically does not help us to measure speedups in the algorithm obtained by improved coding. As an extreme example, an improvement by a factor of 10 is quite significant in practice, but may be ignored by our methdology. Indeed, many empirical studies within the operations research literature are concerned with making minor improvements in the algorithm or in the data structure so that the running times improve by a constant factor. For such situations, we may also keep track of the CPU times and the analysis based on CPU times can be used to evaluate such improvements.

2. Our methodology is not appropriate for analysing parallel algorithms. Although, we can keep track of the number of representative operations in a parallel algorithm, these numbers can not necessarily be used to estimate CPU times because many of these operations are performed in parallel. Communication overhead is often the bottleneck operation in parallel algorithms, and it is quite difficult to estimate this overhead using only computation counts.

3. In certain circumstances, our methodology can lead to incorrect conclusions. Let we illustrate a situation in which this approach will underestimate the asymptotic growth rate of the bottleneck operation. Suppose that the actual growth rate of a bottleneck operation is $h(n) = n^2 + 1000n$ and we have data only for instances $n \leq 2{,}000$. Suppose we conjecture that the growth rate of the function is $n^{1.7}$. When we plot the ratio $(n^2 + 1000n)/n^{1.7}$ for various values of n, we obtain the plot shown in Figure 7, which is a decreasing function of n. Therefore, using the methodology suggested earlier, we would misestimate $n^{1.7}$ as an upper bound on h(n).

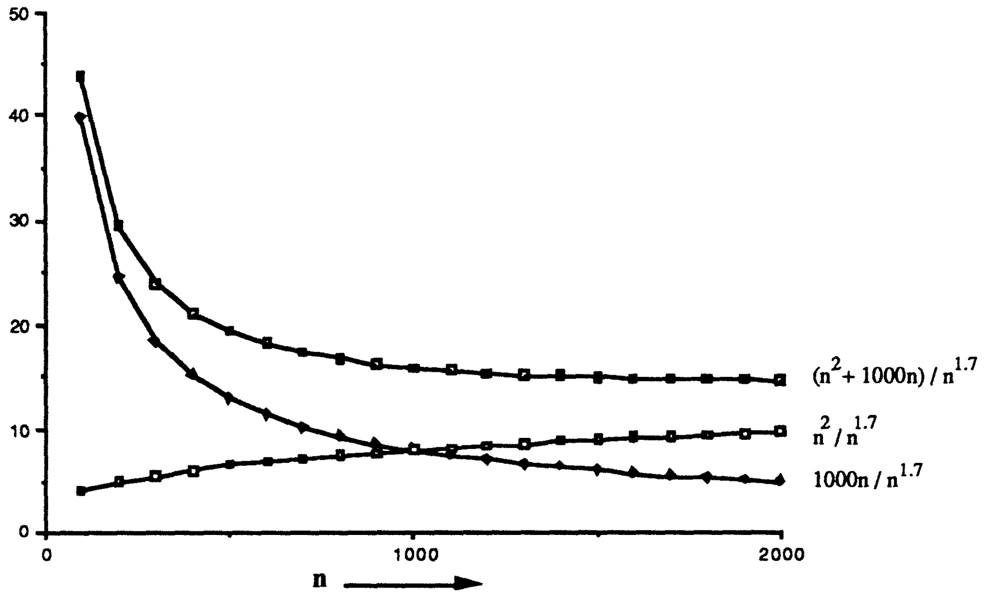


Figure 7. A limitation of the analysis using representative operation counts.

What went wrong in the previous example that would lead us to make such a significant misestimation of the running time? In our case, the growth rate had two components with very dissimilar constant terms and we did not have data for sufficiently large size problems so that the effect of the constant terms become insignificant. Had the growth function been $n^2 + 10\,n$ or had we solved problems of size $n = 100{,}000$, we would not have misestimated the running time, except possibly by a minimal amount. We anticipate that errors of this type will be rather infrequent in practice.

4. We have suggested that we can determine an expression for the virtual running time using regression analysis. In general, regression analysis will misestimate the constant terms in the expression if the representative operations counts are highly correlated. In this case, we might prefer to use alternate methods for estimating the constants, possibly including computer timings of the basic operations.

In addition to these limitations, there are a number of issues that we have either not treated or have treated in a very limited way. For example, many algorithms and heuristic are not guaranteed to get the optimal solution. In this case, one may be very interested in the deviation of the solution from optimality; i. e., one may be interested in either in the absolute error or the relative error or both. We have also not touched much on statistics, or on the selection of test problems.

6. SUMMARY

Most iterative algorithms for solving network flow problems repetitively perform some basic operations. For almost all algorithms, we can decompose these basic operations into fundamental operations so that the algorithm executes each operation in $\theta(1)$ time. An algorithm typically performs a large number of fundamental operations. We refer to a subset of fundamental operations as a set of representative operations if for every possible problem instance, the sum of representative operations provides an upper bound (to within a multiplicative constant) on the sum of all operations that an algorithm performs. We have shown that these representative operation counts might provide valuable information about an algorithm's behavior that is not captured by the CPU time. For example, the representative operation counts allow us (i) to determine the asymptotic growth rate of the running time of an algorithm independent of its computing environment; (ii) to assess the time an algorithm spends on different basic operations; and (iii) to compare two algorithms executed on different computers; and (iv) to estimate the running time of an algorithm on a computer different from the one carrying out the experiments.

The simple methodologies that we have presented for conducting empirical analysis of an algorithm using representative operation counts do not provide rigorous guarantees; nevertheless, they often provide considerable insight about an algorithm's behavior, and they typically yield far more insight than is obtainable through analyzing only CPU times. The ideas we have outlined in this chapter apply to most network algorithms and should, as well, apply to optimization algorithms for problems arising in several other application domains.

REFERENCES

AHUJA, R.K., T. L. MAGNANTI, AND J.B. ORLIN. 1993. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall, Inc., New Jersey. (In Press)

BENTLEY, J.L. 1990. Experiments on geometric traveling salesman heuristics. *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 91-99.

BENTLEY, J.L. AND B.W. KERNIGHAN. 1990. A system for algorithm animation: Tutorial and algorithm animation. Unix Research System Paper, 10th edition, Volume II, Saunders College Publishing, pp. 451-475.

BLAND, R.G., AND D.L. JENSEN. 1985. On the computational behavior of a polynomial-time network flow algorithm. *Mathematical Programming* **54**, 1-39.

BRADLEY, G., G. BROWN, AND G. GRAVES. 1977. Design and implementation of large scale primal transshipment algorithms. *Management Science* **21**, 1-38.

BROWN, M.H. 1988. *Algorithm Animation.* MIT Press, Cambridge, MA.

CHANG, M.D., AND C.J. CHEN. 1989. An improved primal simplex variant for pure processing networks. *ACM Transactions on Mathematical Software* **15**, 64-78.

CROWDER, H.P., AND P.B. SAUNDERS. 1980. Results of a survey on MP performance indicators. *COAL Newsletter, January issue*, pp. 2-6.

CROWDER, H.P., R.S. DEMBO, AND J.M. MULVEY. 1978. Reporting computational experiments in mathematical programming. *Mathematical Programming* **15**, 316-329.

CROWDER, H.P., R.S. DEMBO, AND J.M. MULVEY. 1979. On reporting computational experiments with mathematical software. *ACM Transactions on Mathematical Software* **5**, 193-23.

GLOVER, F., D. KARNEY, AND D. KLINGMAN. 1974. Implementation and computational comparisons of primal, dual and primal-dual computer codes for minimum cost network flow problem. *Networks* **4**, 191-212.

GOLDEN, B.L., A.A. ASSAD, E.A. WASIL, AND E. BAKER. 1986. Experiments in optimization. *European Journal of Operational Research* **27**, 1-16.

GREENBERG, H. 1990. Computational testing: Why, how, and how much. *ORSA Journal of Computing* **2**, 94-97.

GRIGORIADIS, M.D. 1986. An efficient implementation of the network simplex method. *Mathematical Programming Study* **26**, 83-111.

HOFFMAN, K.L., AND R.H.F. JACKSON. 1982. In pursuit of a methodology for testing mathematical programming software. In *Evaluating Mathematical Programming Techniques,* Lecure Notes in Economics and Mathematical Systems, Vol. 199, edited by J.M. Mulvey et al., Springer Verlag, New York.

JACKSON, R.H.B., AND J.M. MULVEY. 1978. A critical review of comparisons of mathematical programming algorithms and software (1953-1977). *Journal of Research of the National Bureau of Standards* **83**, 563-584.

JACKSON, R.H.B., P.T. BOGGS, S.G. NASH, AND S. POWELL. 1989. Report of the ad hoc committee to revise the guidelines for reporting computational experiments in mathematical programming. *COAL Newsletter No. 18,* 3-14.

JOHNSON, D.S. 1990. Local optimization and the traveling salesman problem. *Proceedings of the 17^{th} Colloquim on Automata, Languages and Programming*. Springer Verlag, pp. 446-461.

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

LAWLER, E.L. 1976. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston.

McGEOCH, C.C. 1986. *Experimental Analysis of Algorithms*. Unpublished Ph. D. Dissertation, Department of Computer Science, Carnegie Mellon University, Pittsburg, PA.

McGEOCH, C.C. 1992. Analysis of algorithms by simulation: Variance reduction techniques and simulation speedups. *Computing Surveys* (to apear in the June issue).

MULVEY, J. 1978. Pivot strategies for primal-simplex network codes. *Journal of ACM* **25**, 266-270.