AN EFFICIENT COST SCALING ALGORITHM FOR THE ASSIGNMENT PROBLEM

ANDREW V. GOLDBERG
AND
ROBERT KENNEDY

COMPUTER SCIENCE DEPARTMENT
STANFORD UNIVERSITY
STANFORD, CA 94305

GOLDBERG@CS.STANFORD.EDU
ROBERT@CS.STANFORD.EDU

16 May, 1995

ABSTRACT. The cost scaling push-relabel method has been shown to be efficient for solving minimum-cost flow problems. In this paper we apply the method to the assignment problem and investigate implementations of the method that take advantage of assignment's special structure. The results show that the method is very promising for practical use.

Andrew V. Goldberg was supported in part by ONR Young Investigator Award N00014-91-J-1855, NSF Presidential Young Investigator Grant CCR-8858097 with matching funds from AT&T, DEC, and 3M, and a grant from the Powell Foundation.
Robert Kennedy was supported by the above mentioned ONR and NSF grants.

1. Introduction.

Significant progress has been made in the last decade on the theory of algorithms for network flow problems. Some of the algorithms that came out of this research have been shown to have practical impact as well. In particular, the push-relabel method [10, 15] is the best currently known way for solving the maximum flow problem [1, 7, 22]. This method extends to the minimum-cost flow problem using cost scaling [10, 16]. Earlier implementations of this method [4, 13] performed well on some problems but relatively poorly on others. A later implementation [11] has been shown very competitive on a wide class of problems. In this paper we study efficient implementations of the cost scaling push-relabel method for the assignment problem.

The most relevant theoretical results on the assignment problem are as follows. The best currently known strongly polynomial time bound of $O(n(m + n \log n))$ is achieved by the classical Hungarian method of Kuhn [20]. Here n denotes the number of nodes in the input network and m denotes the number of edges. (Implementations of the Hungarian method and related algorithms are discussed in [6].) Under the assumption that the input costs are integers in the range $[-C, \ldots, C]$, Gabow and Tarjan [9] use cost scaling and blocking flow techniques to obtain an $O(\sqrt{n}m \log(nC))$ time algorithm. Algorithms with the same running time bound based on the push-relabel method appeared in [14, 23].

In this paper we study implementations of the scaling push-relabel method in the context of the assignment problem. We use the ideas behind the minimum-cost flow codes [4, 11, 13], the assignment codes [2, 5, 3], and the ideas of theoretical work on the push-relabel method for the assignment problem [14], as well as new techniques aimed at improving practical performance of the method. We develop several CSA (**C**ost **S**caling **A**ssignment) codes based on different heuristics which improve the code performance on many problem classes. The "basic" code CSA-B does not use any heuristics, the CSA-Q code uses a "quick-minima" heuristic, and the CSA-S code uses a "speculative arc fixing" heuristic. Another outcome of our research is a better understanding of cost scaling algorithm implementations, which may lead in turn to improved cost scaling codes for the minimum-cost flow problem.

We compare the performance of the CSA codes to four other codes: SFR10, an implementation of the auction method for the assignment problem [5]; SJV and DJV, implementations of Jonker and Volgenant's shortest augmenting path method [18] tuned for sparse and dense graphs respectively; and ADP/A, an implementation of the interior-point method specialized for the assignment problem [24]. We make the comparison over classes of problems from generators developed for the First DIMACS Implementation Challenge [17][1] and on problems obtained from digital images as suggested by Don Knuth [19]. Of our codes, CSA-Q is best overall. This code outperforms ADP/A on all problem

[1]The DIMACS benchmark codes, problem generators, and other information we refer to are available by anonymous `ftp` from `dimacs.rutgers.edu`

instances in our tests, outperforms SFR10 on all except one class, and outperforms SJV and DJV on large instances in every class. Although our second-best code, CSA-S, is somewhat slower than CSA-Q on many problem classes, it is usually not much slower and it outperforms CSA-Q on three problem classes, always outperforms ADP/A, is worse than SFR10 by only a slight margin on one problem class and by a noticeable margin on only one problem class, and loses to the Jonker-Volgenant codes only on one class and on small instances from two other classes. While we use the CSA-B code primarily to gauge the effect of heuristics on performance, it is worth noting that it outperforms ADP/A in all our tests, the Jonker-Volgenant codes on large instances from all but one class, and SFR10 on all but one class of problems we tested.

This paper is organized as follows. Section 2 gives the relevant definitions. Section 3 outlines the scaling push-relabel method for the assignment problem. Section 4 discusses our implementation, in particular the techniques used to improve our code's practical performance. Section 5 describes the experimental setup. Section 6 gives the experimental results. In Section 7, we give concluding remarks.

2. Background

Let $\overline{G} = (\overline{V} = X \cup Y, \overline{E})$ be an undirected bipartite graph and let $n = |\overline{V}|$, $m = |\overline{E}|$. A *matching* in $\overline{G}$ is a subset of edges $M \subseteq \overline{E}$ that have no node in common. The *cardinality* of the matching is $|M|$. Given a weight function $\overline{c} : \overline{E} \to \mathbf{R}$, we define the weight of M to be the sum of weights of edges in M. The *assignment problem* is to find a maximum cardinality matching of maximum weight. We assume that the weights are integers in the range $[-C, \ldots, C]$. To simplify the presentation, we assume that $|X| = |Y|$, $\overline{G}$ has a perfect matching (*i.e.*, a matching of cardinality $|X|$), and every node degree in $\overline{G}$ is at least two. We can dispense with these last assumptions without any significant decrease in performance by using a slightly more complicated reduction to the transportation problem than the one described below.

Our implementation reduces the assignment problem to the *transportation problem* defined as follows. Let $G = (V, E)$ be a digraph with a real-valued *capacity* $u(a)$ and a real-valued *cost* $c(a)$ associated with each arc[2] a and a real-valued *supply* $d(v)$ associated with each node v. We assume that $\sum_V d(v) = 0$. A *pseudoflow* is a function $f : E \to \mathbf{R}_+$ satisfying the *capacity* constraints: for each $a \in E$, $f(a) \leq u(a)$. For a pseudoflow f and a node v, the *excess flow into* v, $e_f(v)$, is defined by $e_f(v) = d(v) + \sum_{(u,v) \in E} f(u, v) - \sum_{(v,w) \in E} f(v, w)$. A node v with $e_f(v) > 0$ is called *active*. Note that $\sum_{v \in V} e_f(v) = 0$.

[2]Sometimes we refer to an arc a by its endpoints, *e.g.*, (v, w). This is ambiguous if there are multiple arcs from v to w. An alternative is to refer to v as the tail of a and to w as the head of a, which is precise but inconvenient.

A *flow* is a pseudoflow f such that, for each node v, $e_f(v) = 0$. Observe that a pseudoflow f is a flow if and only if there are no active nodes. The *cost* of a pseudoflow f is given by $\text{cost}(f) = \sum_{a \in E} c(a)f(a)$. The transportation problem is to find a flow of minimum cost.

We use a slight variation of the standard reduction from the assignment problem to the minimum-cost flow problem (see *e.g.* [21]). Given an instance of the assignment problem $(\overline{G}, \overline{c})$, we construct a transportation problem instance $(G = (V, E), c, u)$ as follows. We define $V = \overline{V} = X \cup Y$. For every edge $\{v, w\} \in \overline{E}$ such that $v \in X$ and $w \in Y$, we add the arc (v, w) to E and define $c(v, w) = -\overline{c}(v, w)$ and $u(v, w) = 1$. Finally we define $d(v) = 1$ for all $v \in X$ and $d(w) = -1$ for all $w \in Y$. Note that the graph G is bipartite.

For a given pseudoflow f, the *residual capacity* of an arc $a \in E$ is $u_f(a) = u(a) - f(a)$. The set of *residual arcs* E_f contains the arcs $a \in E$ with $f(a) < u(a)$ and the reverse arcs, a^R, for every $a \in E$ with $f(a) > 0$. The *residual graph* $G_f = (V, E_f)$ is the graph induced by the residual arcs. For $a \in E$, we define $c(a^R) = -c(a)$. Note that if G is obtained by the above reduction, then for any integral pseudoflow f and for any arc $a \in E$, $u(a), f(a) \in \{0, 1\}$.

A *price function* is a function $p : V \to \mathbf{R}$. For a given price function p, the *reduced cost* of an arc (v, w) is $c_p(v, w) = c(v, w) + p(v) - p(w)$ and the *partial reduced cost* is $c'_p(v, w) = c(v, w) - p(w)$.

For a constant $\epsilon \geq 0$, a pseudoflow f is said to be *ϵ-optimal with respect to a price function* p if, for every residual arc $a \in E_f$, we have

$$\begin{cases} a \in E \Rightarrow c_p(a) \geq 0, \\ a^R \in E \Rightarrow c_p(a) \geq -\epsilon. \end{cases}$$

A pseudoflow f is *ϵ-optimal* if f is ϵ-optimal with respect to some price function p. If the arc costs and capacities are integers and $\epsilon < 1/n$, any ϵ-optimal flow is optimal.

For a given f and p, an arc $a \in E_f$ is *admissible* iff

$$\begin{cases} a \in E \text{ and } c_p(a) < \epsilon/2 \quad \text{or} \\ a^R \in E \text{ and } c_p(a) < -\epsilon/2. \end{cases}$$

The *admissible graph* $G_A = (V, E_A)$ is the graph induced by the admissible arcs.

3. The Method

First we give a high-level description of the successive approximation algorithm (see Figure 1). For a detailed presentation of the successive approximation framework and the associated proofs, see [16]. The algorithm starts with $\epsilon = C$ and $p(v) = 0$ for all $v \in V$. At the beginning of every iteration, the algorithm divides ϵ by a constant factor α and sets f to the zero pseudoflow. The iteration modifies f

procedure MIN-COST(V, E, u, c);
  [initialization]
  $\epsilon \leftarrow C$; $\forall v$, $p(v) \leftarrow 0$;
  [loop]
  while $\epsilon \geq 1/n$ **do**
    $(\epsilon, f, p) \leftarrow$ *refine*(ϵ, p);
  return(f);
end.

FIGURE 1. The cost scaling algorithm.

procedure REFINE(ϵ, p);
  [initialization]
  $\epsilon \leftarrow \epsilon/\alpha$;
  $\forall a \in E$, $f(a) \leftarrow 0$;
  $\forall v \in X$, $p(v) \leftarrow -\min_{(v,w)\in E} c'_p(v,w)$;
  [loop]
  while f is not a flow
    apply a *push* or a *relabel* operation;
  return(ϵ, f, p);
end.

FIGURE 2. The generic *refine* subroutine.

and p so that f is an (ϵ/α)-optimal flow with respect to p. When $\epsilon < 1/n$, f is optimal and the algorithm terminates. The number of iterations of the algorithm is $1 + \lfloor \log_\alpha(nC) \rfloor$.

Reducing ϵ is the task of the subroutine *refine*. The input to *refine* is ϵ and p such that there exists a flow f that is ϵ-optimal with respect to p. The output from *refine* is $\epsilon' = \epsilon/\alpha$, a flow f, and a price function p such that f is ϵ'-optimal with respect to p.

The generic *refine* subroutine (described in Figure 2) begins by decreasing the value of ϵ, setting f to the zero pseudoflow (thus creating some excesses and making some nodes active), and setting $p(v) = -\min_{(v,w)\in E}\{c'_p(v,w)\}$ for every $v \in X$. This converts the f into an ϵ-optimal pseudoflow (indeed, into a 0-optimal pseudoflow). Then the subroutine converts f into an ϵ-optimal flow by applying a sequence of *push* and *relabel* operations, each of which preserves ϵ-optimality. The generic algorithm does not specify the order in which these operations are applied. Next, we describe the *push* and *relabel* operations for the unit-capacity case.

A *push* operation applies to an admissible arc (v, w) whose tail node v is active. It consists of pushing one unit of flow from v to w, thereby decreasing $e_f(v)$ by one, increasing $e_f(w)$, and either increasing $f(v, w)$ by one if $(v, w) \in E$ or decreasing $f(w, v)$ by one if $(w, v) \in E$. A *relabel* operation applies to a node v. The operation sets $p(v)$ to the smallest value allowed by the ϵ-optimality constraints, namely

PUSH(v, w).
    send a unit of flow from v to w.
end.

RELABEL(v).
    if $v \in X$
        then replace $p(v)$ by $\max_{(v,w)\in E_f}\{p(w) - c(v, w)\}$
        else replace $p(v)$ by $\max_{(v,w)\in E_f}\{p(w) - c(v, w) - \epsilon\}$
end.

FIGURE 3. The *push* and *relabel* operations

$\max_{(v,w)\in E_f}\{p(w) - c(v, w)\}$ if $v \in X$, or $\max_{(v,w)\in E_f}\{p(w) - c(v, w) - \epsilon\}$ otherwise.

The analysis of cost scaling push-relabel algorithms is based on the following facts [14, 16]: During a scaling iteration

- no node price increases;
- for any $v \in V$, $p(v)$ decreases by $O(n\epsilon)$.

4. Implementation and Heuristics

In this section we discuss implementation issues and heuristics aimed at speeding up the method.

The efficiency of a scaling implementation depends on the choice of scale factor α. Although an earlier study [5] suggests that the performance of scaling codes for the assignment problem may be quite sensitive to the choice of scale factor, our observations are to the contrary. Spot checks on instances from several problem classes indicated that the running times seem to vary by a factor of no more than 2 for values of α between 4 and 40. We chose $\alpha = 10$ for our tests; different values of α would yield running times that are somewhat worse on some problem classes and somewhat better on others, but the difference is not drastic. We believe the lack of robustness alluded to in [5] may be due to a characteristic of the implementation of SFR10 and related codes. In particular, SFR10 contains an "optimization" that seems to terminate early scaling phases prematurely. Our codes run every scaling phase to completion as suggested by the theory.

The efficiency of an implementation of refine depends on the number of operations performed by the method and on the implementation details. We discuss the operation ordering first.

The implementation maintains the price function p and the flow f. For each node $w \in Y$ with $e_f(w) = 0$, we maintain a pointer to the unique node $v = \mu(w)$ such that $f(v, w) = 1$.

Our implementation maintains the invariant that only the nodes in X are active, except possibly in the middle of the double-push operation described below. The implementation picks an active node and applies the double-push operation to it.

The performance of the implementation depends on the strategy for selecting the next active node to process. We experimented with several operation orderings, including those suggested in [16] and [12]. Our implementation uses the LIFO ordering, *i.e.*, the set of active nodes is maintained as a stack. This ordering worked best in our tests; the FIFO ordering usually worked somewhat worse, although the difference was never drastic.

4.1. The Double-Push Operation. The *double-push* operation is similar to a sequential version of the match-and-push procedure from [14]. The operation applies to an active node v. Recall that at the beginning of a double-push, all active nodes are in X, so $v \in X$.

First the double-push operation processes v by relabeling v, pushing flow from v along an admissible arc (v, w), and then relabeling v again. If $e_f(w)$ becomes positive, the operation pushes flow from w to $\mu(w)$ and sets $\mu(w) = v$. Finally, double-push relabels w.

Lemma 4.1. *The double-push operation is correct.*

Proof. We only need to show that double-push applies the pushing operation correctly. Since immediately before the flow is pushed out of v the node is relabeled, there is an admissible arc out of v and the push is correct. If this push makes w active, then there is a second push from w to $\mu(w)$.

Consider the last double-push into w which set $\mu(w)$ to its current value. Because the network is obtained via a reduction described in Section 2, $(w, \mu(w))$ is the only residual arc out of w. So when the double-push relabeled w, $c_p(\mu(w), w)$ became ϵ. From this double-push to the current one, w and $\mu(w)$ have not been relabeled (the latter holds because $(w, \mu(w))$ was the only residual arc into $\mu(w)$ during that time period). Thus during the current push from w, $c_p(\mu(w), w) = \epsilon$, so the push is valid. ∎

Lemma 4.2. *A double-push operation decreases the price of a node $w \in Y$ by at least ϵ.*

Proof. Just before the double-push, w is either unmatched or matched.

In the first case, the flow is pushed into w and at this point the only residual arc out of w is the arc (w, v). Just before that the double-push relabeled v and $c_p(v, w) = 0$. Next double-push relabels w and $p(w)$ decreases by ϵ.

In the second case, the flow is pushed to w and at this point w has two outgoing residual arcs, (w, v) and $(w, \mu(w))$. As we have seen, $c_p(v, w) = 0$ before v's second relabeling, and $c_p(\mu(w), w) = \epsilon$. After the second relabeling of v, double-push pushes flow from w to $\mu(w)$ and relabels w, reducing $p(w)$ by at least ϵ. ∎

DOUBLE-PUSH(v).
  let (v, w) and (v, z) be the arcs with the smallest and the second-smallest reduced costs;
  $push(v, w)$;
  $p(v) = -c'_p(v, z)$;
  if $e_f(w) > 0$
    $push(w, \mu(w))$;
  $\mu(w) = v$;
  $p(w) = p(v) + c(v, w) - \epsilon$;
end.

FIGURE 4. Efficient implementation of double-push

Corollary 4.3. *There are $O(n^2)$ double-push operations per refine.*

4.2. Efficient Implementation. Suppose we apply double-push to a node v. Let (v, w) and (v, z) be the arcs out of v with the smallest and the second-smallest reduced costs, respectively. These arcs can by found by scanning the adjacency list of v once. The effects of double-push on v are equivalent to pushing flow along (v, w) and setting $p(v) = -c'_p(v, z)$. To relabel w, we set $p(w) = p(v) + c(v, w) - \epsilon$. This implementation of double-push is summarized in Figure 4.

It is not necessary to maintain the prices of nodes in X explicitly; for $v \in X$, we can define $p(v)$ implicitly by $p(v) = -\min_{(v,w)\in E}\{c'_p(v, w)\}$ if $e_f(v) = 1$ and $p(v) = c'(v, w) + \epsilon$ if $e_f(v) = 0$ and (v, w) is the unique arc with $f(v, w) = 1$. One can easily verify that using implicit prices is equivalent to using explicit prices in the above implementation. The only time we need to know the value of $p(v)$ is when we relabel w in double-push, and at that time $p(v) = -c'_p(v, z)$ which we compute during the previous relabel of v. Maintaining the prices implicitly saves memory and time. The implementation of the double-push operation with implicit prices is similar to the basic step of the auction algorithm of [2].

Our code CSA-B implements the scaling push-relabel algorithm using stack ordering of active nodes and the implementation of double-push with implicit prices mentioned above.

4.3. Heuristics. In this section we describe two heuristics that often improve the algorithm's performance.

The kth-best heuristic [2] is aimed at reducing the number of scans of arc lists of nodes in X. The idea of the kth-best heuristic is as follows. Recall that we scan the list of v to find the arcs (v, w) and (v, z) with the smallest and second-smallest values of the partial reduced cost. Let $k \geq 3$ be an integer. When we scan the list of $v \in X$, we compute the kth-smallest value K of the partial reduced costs of the outgoing arcs and store the $k-1$ arcs with the $k-1$ smallest partial reduced costs. The node prices monotonically decrease during refine, hence during the subsequent double-push operations we can first look for the smallest and the second-smallest arcs among the stored arcs whose current partial reduced cost is at most K. We need to scan the list of v again only when all except possibly one of the saved arcs have partial reduced costs greater than K.

Our code CSA-Q is a variation of CSA-B that uses the 4th-best heuristic.

The idea of the *speculative arc fixing* heuristic [8, 11] is to move arcs with reduced costs of large magnitude to a special list. These arcs are not examined by the double-push procedure but are examined as follows at a (relatively large) periodic interval. When the arc (v, w) is examined, if the ϵ-optimality condition is violated on (v, w), $f(v, w)$ is modified to restore ϵ-optimality and (v, w) is moved back to the adjacency list of v; if ϵ-optimality holds for (v, w) but $|c_p(v, w)|$ is no longer large, (v, w) is simply moved back to the adjacency list. This heuristic takes advantage of the fact that the flow is *fixed* on arcs of high reduced cost [16].

Our code CSA-S is a variation of CSA-B that uses the speculative arc fixing heuristic.

We implemented a number of other heuristics that are known to improve performance of cost scaling code for the minimum-cost flow problem [11]. Among these are: *global price updates* which periodically ensure, via a specialized shortest-paths computation, that the admissible graph contains a path from every node with flow excess to some node with flow deficit; and *price refinement* which determines at each iteration whether the current assignment is actually ϵ'-optimal for some $\epsilon' < \epsilon$, and hence avoids unnecessary executions of *refine*. Our best implementation uses neither of these strategies, however, since even taking advantage of the assignment problem's structure to simplify and speed up these heuristics, a typical price refinement iteration used more time than simply executing *refine* in our tests. The double-push operation seems to maintain a sufficiently "aggressive" price function and global price updates cannot reduce the number of push and relabel operations enough to improve the running time.

5. Experimental Setup

All the test runs were executed on a Sun SparcStation 2 with a clock rate of 40 MHz and 96 Megabytes of main memory. We compiled the SFR10 code supplied by David Castañon with the Sun Fortran-77 compiler, release 2.0.1 using the `-O4` optimization switch[3]. We compiled the DJV and SJV codes supplied by Jianxiu Hao with the Sun C compiler release 1.0, using the `-O2` optimization option. We compiled our CSA codes with the Sun C compiler release 1.0, using the `-fast` optimization option; each choice seemed to yield the fastest execution times for the code where we used it. Times reported here are Unix *user* CPU times, and were measured using the `times()` library function. During each run, the programs

[3]Castañon [5] recommends setting the initial "bidding increment" in SFR10 to a special value for problems of high density; we found this advice appropriate for the dense problem class, but discovered that it hurt performance on the geometric class. We followed Castañon's recommendation only on the class where it seemed to improve SFR10's performance.

C benchmarks *user* times	
Test 1	Test 2
2.7 sec	24.0 sec

FORTRAN benchmarks *user* times	
Test 1	Test 2
1.2 sec	2.2 sec

FIGURE 5. DIMACS benchmark times

collect time usage information after reading the input problem and initializing all data structures and again after computing the optimum assignment; we take the difference between the two figures to indicate the CPU time actually spent solving the assignment problem.

To give a baseline for comparison of our machine's speed to others, we ran the DIMACS benchmarks `wmatch` (to benchmark C performance) and `netflo` (to benchmark FORTRAN performance) on our machines, with the timing results given in Figure 5. It is interesting (though neither surprising nor critical to our conclusions) to note that the DIMACS benchmarks do not precisely reflect the mix of operations in the codes we developed. Of two C compilers available on our system, the one that consistently ran our code faster by a few percent also ran the benchmarks more slowly by a few percent (the C benchmark times in Figure 5 are for code generated by the same compiler we used for our experiments). But even though they should not be taken as the basis for very precise comparison, the benchmarks provide a useful way to estimate relative speeds of different machines on the sort of operations typically performed by combinatorial optimization codes.

We did not run the ADP/A code on our machine, but because the benchmark times reported in [24] differ only slightly from the times we obtained on our machine, we conclude that the running times reported for ADP/A in [24] form a reasonable basis for comparison with our codes. Therefore, we report running times directly from [24]. As the reader will see, even if this benchmark comparison introduces a significant amount of error, our conclusions about the codes' relative performance are justified by the large differences in performance between ADP/A and the other codes we tested.

The DJV code is designed for dense problems and uses an adjacency-matrix data structure. The memory requirements for this code would be prohibitive on sparse problems with many nodes. For this reason, we included it only in experiments on problem classes that are dense. On these problems, DJV is faster than SJV by a factor of about 1.5. It is likely that our codes and the SFR10 code would enjoy a similar improvement in performance if they were modified to use the adjacency-matrix data structure.

We collected performance data on a variety of problem classes, many of which we took from the First DIMACS Implementation Challenge. Following is a brief description of each class; details of the generator inputs that produced each set of instances are included in Appendix A.

5.1. The High-Cost Class. Each $v \in X$ is connected by an edge to $2\log_2 |V|$ randomly-selected nodes of Y, with integer edge costs uniformly distributed in the interval $[0, 10^8]$.

5.2. The Low-Cost Class. Each $v \in X$ is connected by an edge to $2\log_2|V|$ randomly-selected nodes of Y, with integer edge costs uniformly distributed in the interval $[0, 100]$.

5.3. The Two-Cost Class. Each $v \in X$ is connected by an edge to $2\log_2|V|$ randomly-selected nodes of Y, each edge having cost 100 with probability 1/2, or cost 10^8 with probability 1/2.

5.4. The Fixed-Cost Class. For problems in this class, we view X as a copy of the set $\{1, 2, \ldots, |V|/2\}$, and Y as a copy of $\{\,|V|/2+1, |V|/2+2, \ldots, |V|\,\}$. Each $v \in X$ is connected by an edge to $|V|/16$ randomly-selected nodes of Y, with edge (x, y), if present, having cost $100 \cdot x \cdot y$.

5.5. The Geometric Class. Geometric problems are generated by placing a collection of integer-coordinate points uniformly at random in the square $[0, 10^6] \times [0, 10^6]$, coloring half the points blue and the other half red, and introducing an edge between every red point r and every blue point b with cost equal to the floor of the distance between r and b.

5.6. The Dense Class. Like instances of the geometric class, dense problems are complete, but edge costs are distributed uniformly at random in the range $[0, 10^7]$.

5.7. Picture Problems. Picture problems, suggested by Don Knuth [19], are generated from photographs scanned at various resolutions, with 256 greyscale values. The set V is the set of pixels; the pixel at row r, column c is a member of X if $r + c$ is odd, and is a member of Y otherwise. Each pixel has edges to its vertical and horizontal neighbors in the image, and the cost of each edge is the absolute value of the greyscale difference between its two endpoints. Note that picture problems are extremely sparse, with an average degree always below 4. Although picture problems are an abstract construct with no practical motivation, the solution to a picture problem can be viewed as a tiling of the picture with dominos, where we would like each domino to cover greyscale values that are as different as possible.

For our problems, we used two scanned photographs, one of each author of this paper.

6. Experimental Observations and Discussion

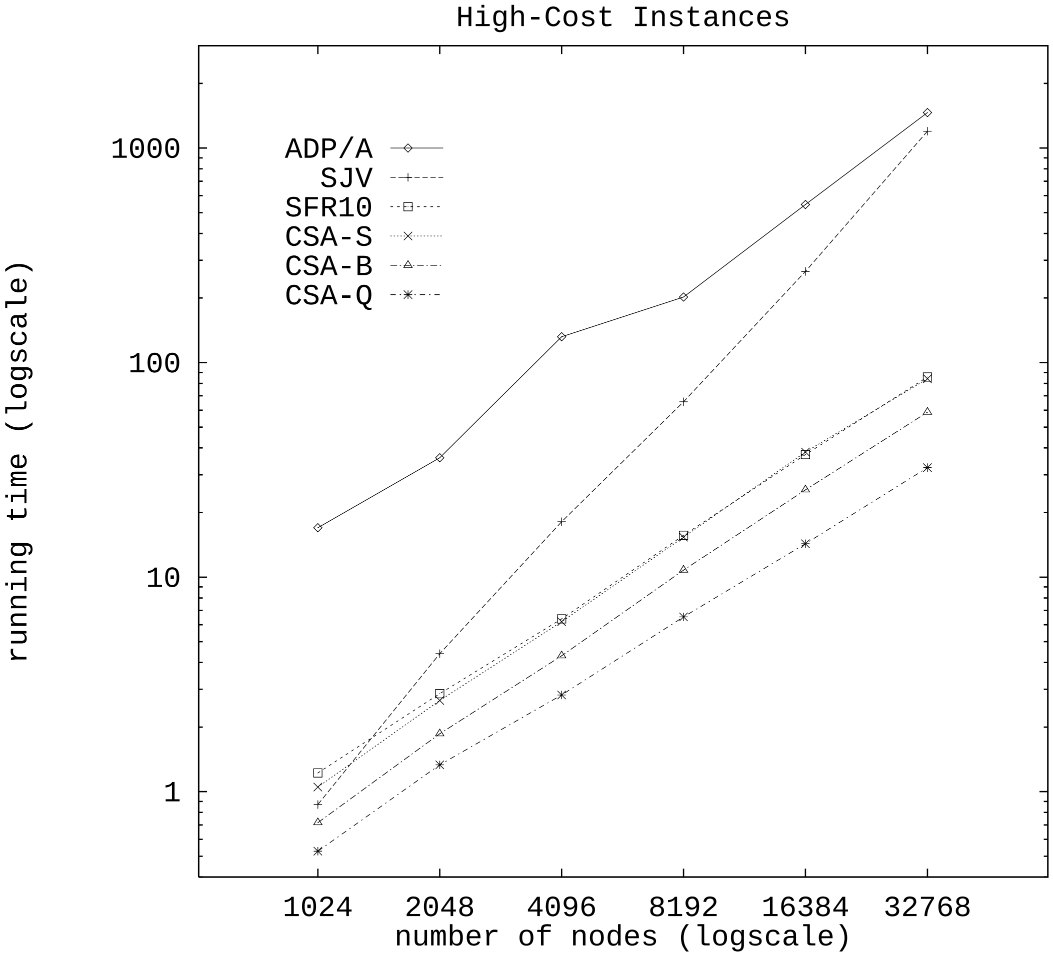
In the following tables and graphs, we present performance data for the codes. Note that problem instances are characterized by the number of nodes on a single side, *i.e.*, *half* the number of nodes in the graph.

We report times on the test runs we conducted, along with performance data for the ADP/A code taken from [24]. The instances on which ADP/A was timed in [24] are identical to those we used in our tests. We give mean running times computed over three instances for each problem size in each class; in the two-cost and geometric classes we also give mean running times computed over 15 instances and sample deviations for each sample size. We computed sample deviations for each problem class and size, and observed that in most cases they were less than ten percent of the mean (often much less). The two exceptions were the two-cost and geometric classes, where we observed larger sample deviations in the running times for some of the codes. For these two classes we also collected data on 15 instances for each problem size. The sample statistics taken over 15 instances seem to validate those we observed for three instances. All statistics are reported in seconds.

6.1. The High-Cost Class. Figure 6 summarizes the timings on DIMACS high-cost instances. The kth-best heuristic yields a clear advantage in running time on these instances. CSA-Q beats CSA-B, its nearest competitor, by a factor of nearly 2 on large instances, and CSA-Q seems to have an asymptotic advantage over the other codes, as well. The overhead of speculative arc fixing is too great on high-cost instances; the running times of CSA-S for large graphs are essentially the same as those of SFR10. SJV has the worst asymptotic behavior.

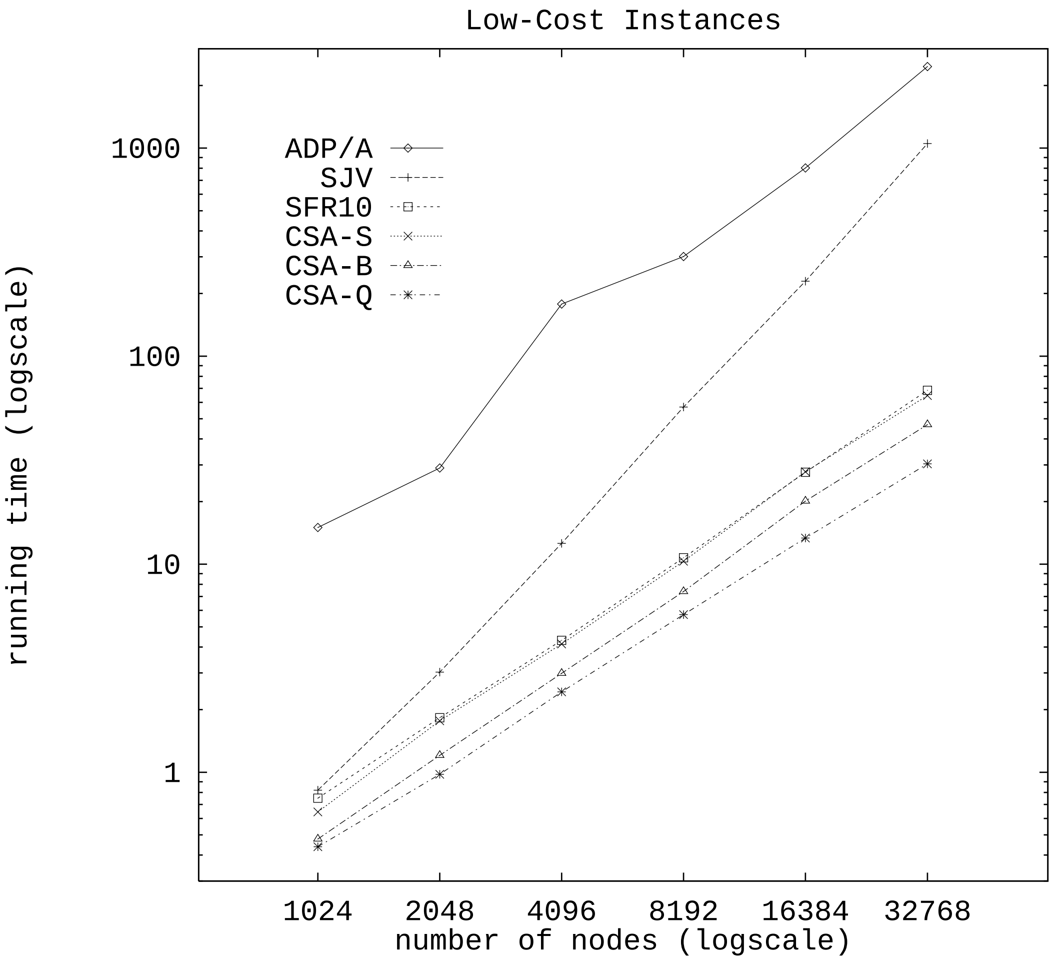
6.2. The Low-Cost Class. The situation here is very similar to the high-cost case: CSA-Q enjoys a slight asymptotic advantage as well as a clear constant-factor advantage over the competing codes. SJV has worse asymptotic behavior than the other codes on the low-cost class, just as it does on high-cost instances. See Figure 7.

6.3. The Two-Cost Class. The two-cost data appear in Figure 8 and Figure 9. It is difficult for robust scaling algorithms to exploit the special structure of two-cost instances; the assignment problem for most of the graphs in this class amounts to finding a perfect matching on the high-cost edges, and none of the scaling codes we tested is able to take special advantage of this observation. Because SJV does not use scaling, it would seem a good candidate to perform especially well on this class, and indeed it does well on small two-cost instances. For large instances, however, SJV uses a great deal of time in its shortest augmenting path phase, and performs poorly for this reason. Speculative arc fixing improves significantly upon the performance of the basic CSA implementation, and the kth-best heuristic hurts performance on this class of problems. It seems that the kth-best heuristic tends to speed up the last few iterations of *refine*, but it hurts in the early iterations. Like kth-best, the speculative arc fixing heuristic is able to capitalize on the fact that later iterations of *refine* can afford to ignore many of the arcs incident to each node, but by keeping all arcs of similar cost under consideration in the beginning, speculative arc fixing allows early iterations to run relatively fast. On this class, CSA-S is the winner, although for applications limited to this sort of strongly bimodal cost distribution, an unscaled push-relabel or blocking flow algorithm might perform better than any of the codes we tested. No running times are given in [24] for ADP/A on this problem class, but the authors suggest that their program performs very well on two-cost problems. Relative to those of the other codes, the running times of SFR10 are



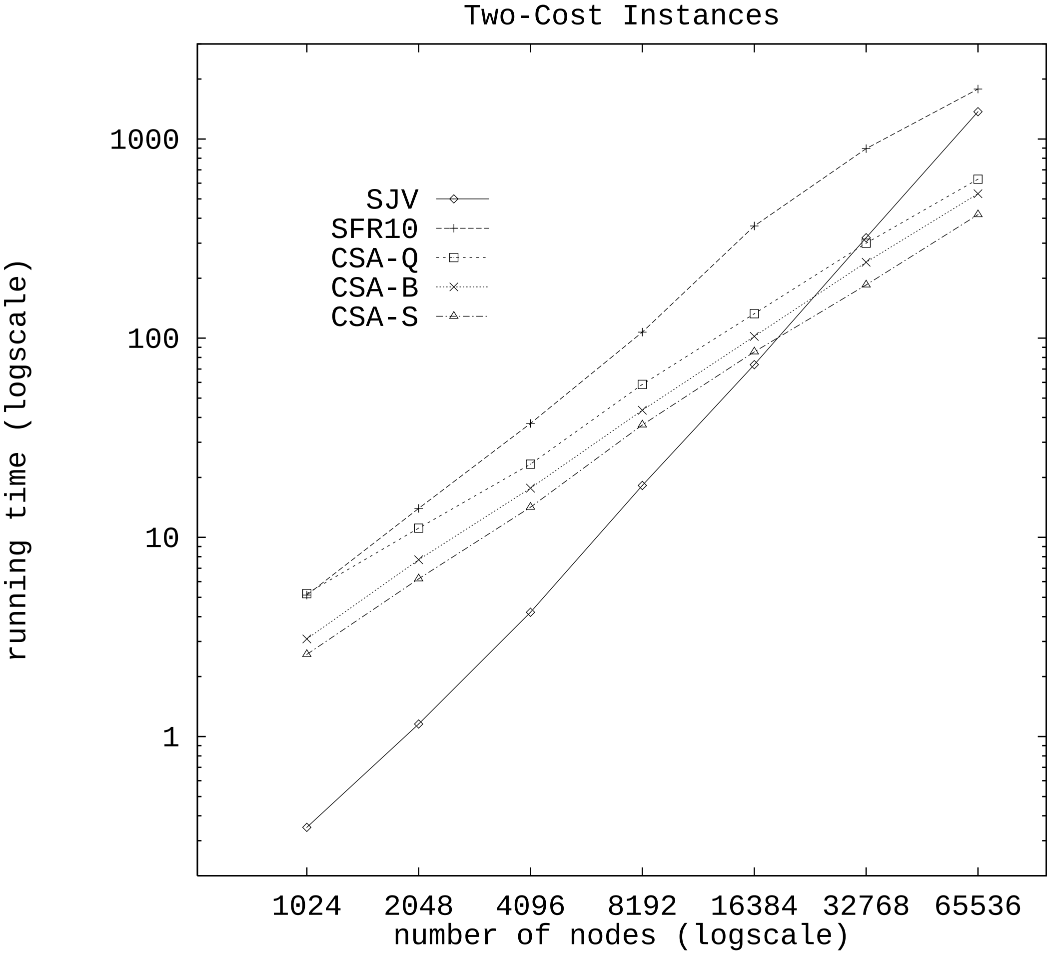
Nodes (|X|)	ADP/A time	SFR10 time	SJV time	CSA-B time	CSA-S time	CSA-Q time
1024	**17**	**1.2**	**0.87**	**0.7**	**1.1**	**0.5**
2048	**36**	**2.9**	**4.40**	**1.9**	**2.7**	**1.3**
4096	**132**	**6.4**	**18.1**	**4.3**	**6.2**	**2.8**
8192	**202**	**15.7**	**65.6**	**10.8**	**15.3**	**6.5**
16384	**545**	**37.3**	**266**	**25.5**	**38.3**	**14.3**
32768	**1463**	**85.7**	**1197**	**58.7**	**84.0**	**32.4**

FIGURE 6. Running Times for the High-Cost Class



Nodes (|X|)	ADP/A time	SFR10 time	SJV time	CSA-B time	CSA-S time	CSA-Q time
1024	**15**	**0.75**	**0.82**	**0.48**	**0.64**	**0.44**
2048	**29**	**1.83**	**3.03**	**1.21**	**1.77**	**0.98**
4096	**178**	**4.31**	**12.6**	**2.99**	**4.13**	**2.43**
8192	**301**	**10.7**	**57.0**	**7.39**	**10.3**	**5.72**
16384	**803**	**27.7**	**229**	**20.1**	**27.8**	**13.4**
32768	**2464**	**68.5**	**1052**	**46.9**	**64.6**	**30.3**

FIGURE 7. Running Times for the Low-Cost Class



Nodes	SFR10		SJV		CSA-B		CSA-S		CSA-Q	
($\|X\|$)	time	s	time	s	time	s	time	s	time	s
1024	**5.13**	0.09	**0.35**	0.00	**3.09**	0.24	**2.58**	0.15	**5.21**	0.33
2048	**14.0**	1.1	**1.16**	0.01	**7.72**	0.28	**6.19**	0.18	**11.1**	1.0
4096	**37.3**	1.1	**4.21**	0.16	**17.7**	1.1	**14.2**	1.6	**23.3**	2.4
8192	**107**	12	**18.2**	0.43	**43.4**	3.5	**36.7**	2.1	**58.6**	3.3
16384	**366**	81	**73.6**	0.58	**102**	2.8	**85.4**	3.2	**133**	7.8
32768	**894**	180	**320**	1.2	**240**	6.0	**185**	6.8	**299**	6.4
65536	**1782**	60	**1370**	5.8	**531**	15	**417**	11	**628**	25

FIGURE 8. Running Times (3-instance samples) for the Two-Cost Class

Nodes	SFR10		SJV		CSA-B		CSA-S		CSA-Q	
($\|X\|$)	time	s	time	s	time	s	time	s	time	s
1024	**5.05**	0.32	**0.35**	0.02	**3.07**	0.21	**2.56**	0.10	**4.93**	0.40
2048	**14.1**	1.1	**1.18**	0.04	**7.49**	0.37	**6.18**	0.26	**10.8**	0.73
4096	**37.4**	2.7	**4.22**	0.14	**17.7**	1.0	**14.7**	0.84	**24.1**	1.6
8192	**109**	9.8	**18.0**	0.37	**44.6**	2.5	**36.5**	1.5	**57.5**	3.2
16384	**314**	50	**73.7**	0.57	**105**	4.2	**84.1**	2.9	**130**	8.4
32768	**822**	194	**320**	2.1	**239**	8.5	**186**	4.8	**293**	15
65536	**2021**	342	**1376**	7.5	**524**	25	**426**	16	**637**	27

FIGURE 9. Running Times (15-instance samples) for the Two-Cost Class

comparatively scattered at each problem size in this class; we believe this phenomenon results from the premature termination of early scaling phases in SFR10 (see Section 4).

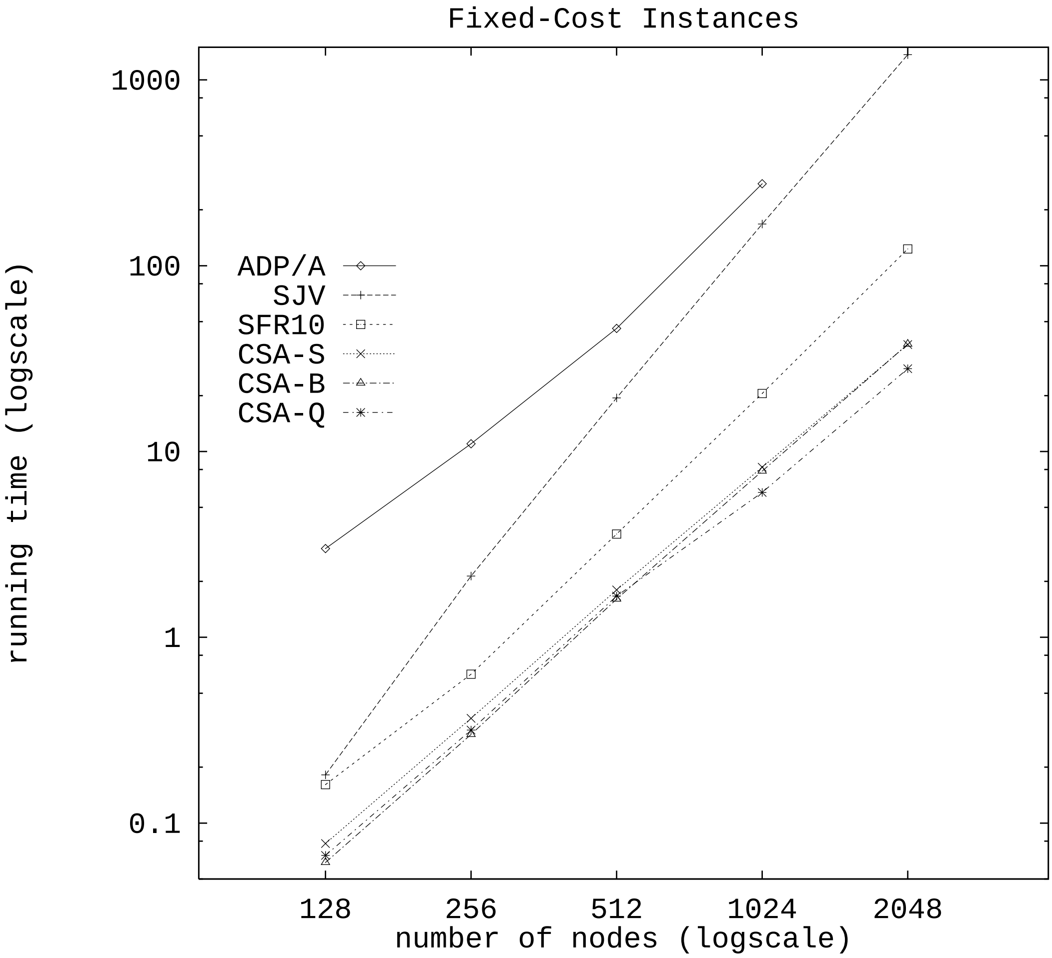
The relatively large sample deviations shown in Figure 8 motivated our experiments with 15 instances of each problem size. The sample means and deviations of the 15-instance data are shown in Figure 9, and they are consistent with and very similar to the three-instance data shown in Figure 8.

6.4. The Fixed-Cost Class. Figure 10 gives the data for the fixed-cost problem class. On smaller instances of this class, CSA-B and CSA-Q have nearly the same performance. On instances with $|X| = 1024$ and $|X| = 2048$, CSA-Q is faster on fixed-cost problems than CSA-B, or indeed any of the other codes. On smaller instances, speculative arc fixing does not pay for itself; when $|X| = 2048$, the overhead is just paid for. Perhaps on larger instances, speculative arc fixing would pay off. It is doubtful, though, that CSA-S would beat CSA-Q on any instances of reasonable size. SJV exhibits the worst asymptotic behavior among the codes we tested on this problem class.

6.5. The Geometric Class. On geometric problems, both heuristics improve performance over the basic CSA-B code. The performance of CSA-S and CSA-Q is similar to and better than that of the other codes. The Jonker-Volgenant codes seem to have asymptotic behavior similar to the other codes on this class. See Figure 11.

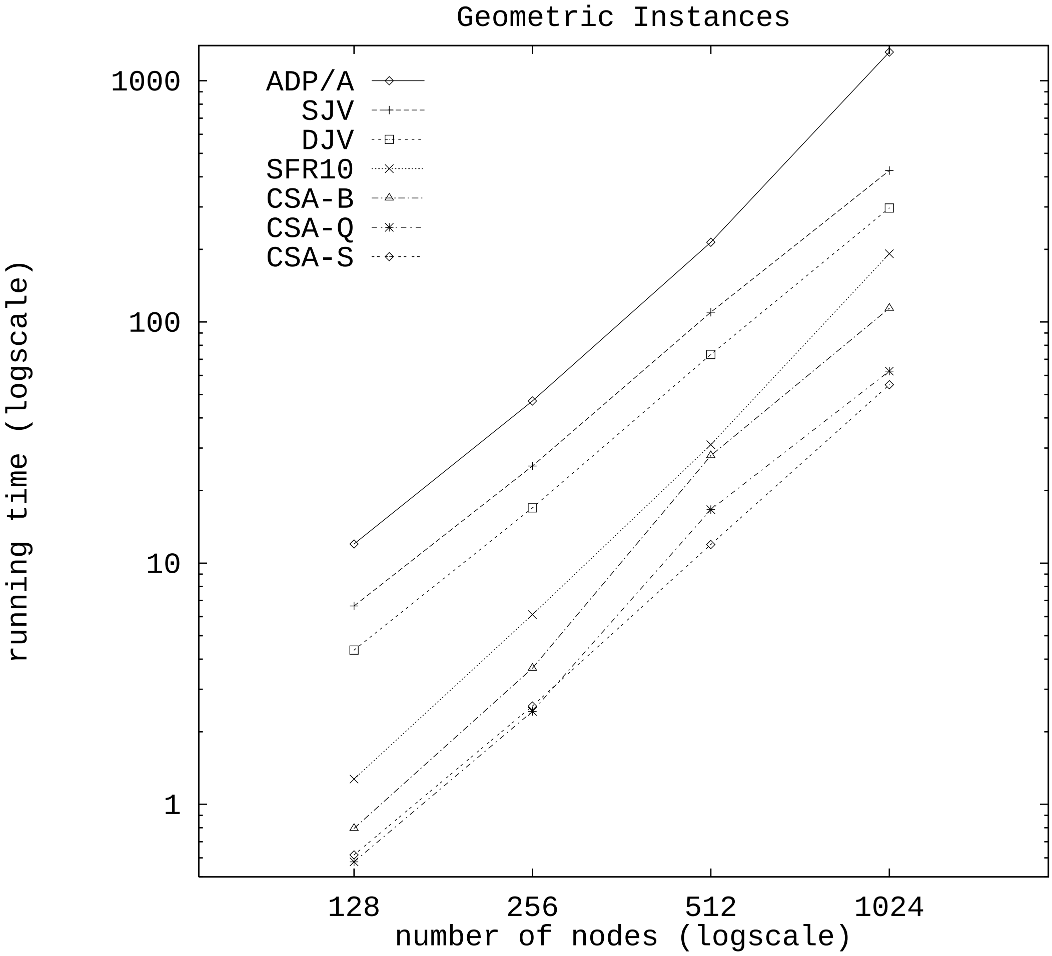
Because the sample deviations shown in Figure 11 are somewhat large compared to those we observed on most other problem classes, we ran experiments on 15 instances as a check on the validity of the data. Statistics calculated over 15-instance samples are reported in Figure 12, and they are very much like the three-instance data.

6.6. The Dense Class. The difference between Figures 12 and 13 shows that the codes' relative performance is significantly affected by changes in cost distribution. Except on very small instances, CSA-Q is the winner in this class; DJV is its closest competitor, with SJV performing fairly well also.



Nodes (|X|)	ADP/A time	SFR10 time	SJV time	CSA-B time	CSA-S time	CSA-Q time
128	**3**	**0.16**	**0.18**	**0.06**	**0.08**	**0.07**
256	**11**	**0.63**	**2.14**	**0.30**	**0.37**	**0.32**
512	**46**	**3.59**	**19.4**	**1.6**	**1.8**	**1.7**
1024	**276**	**20.5**	**168**	**7.8**	**8.2**	**6.0**
2048	**N/A**	**123**	**1367**	**37.8**	**37.6**	**27.9**

FIGURE 10. Running Times for the Fixed-Cost Class



Nodes	ADP/A		SFR10		SJV		DJV		CSA-B		CSA-S		CSA-Q	
($\lvert X \rvert$)	time	s	time	s	time	s	time	s	time	s	time	s	time	s
128	**12**	0.5	**1.27**	0.46	**6.64**	4.4	**4.36**	2.9	**0.79**	0.28	**0.62**	0.05	**0.58**	0.19
256	**47**	1	**6.12**	0.23	**25.3**	3.3	**16.9**	2.0	**3.67**	0.67	**2.56**	0.08	**2.43**	0.34
512	**214**	42	**31.0**	4.1	**110**	2.8	**73.2**	1.0	**27.9**	8.1	**11.9**	0.89	**16.7**	3.7
1024	**1316**	288	**193**	19	**424**	51	**297**	32	**114**	24	**54.9**	1.42	**62.5**	2.6

FIGURE 11. Running Times (3-instance samples) for the Geometric Class

Nodes	SFR10		SJV		DJV		CSA-B		CSA-S		CSA-Q	
($\lvert X \rvert$)	time	s	time	s	time	s	time	s	time	s	time	s
128	**1.28**	0.21	**5.96**	2.0	**3.85**	1.3	**0.78**	0.16	**0.61**	0.03	**0.57**	0.11
256	**6.21**	0.82	**26.1**	4.7	**17.5**	2.9	**3.72**	0.51	**2.63**	0.09	**2.50**	0.27
512	**35.0**	6.0	**101**	11	**68.2**	7.4	**23.2**	4.9	**11.8**	0.67	**15.1**	2.4
1024	**214**	54	**416**	38	**291**	25	**127**	27	**54.4**	2.2	**66.7**	9.7

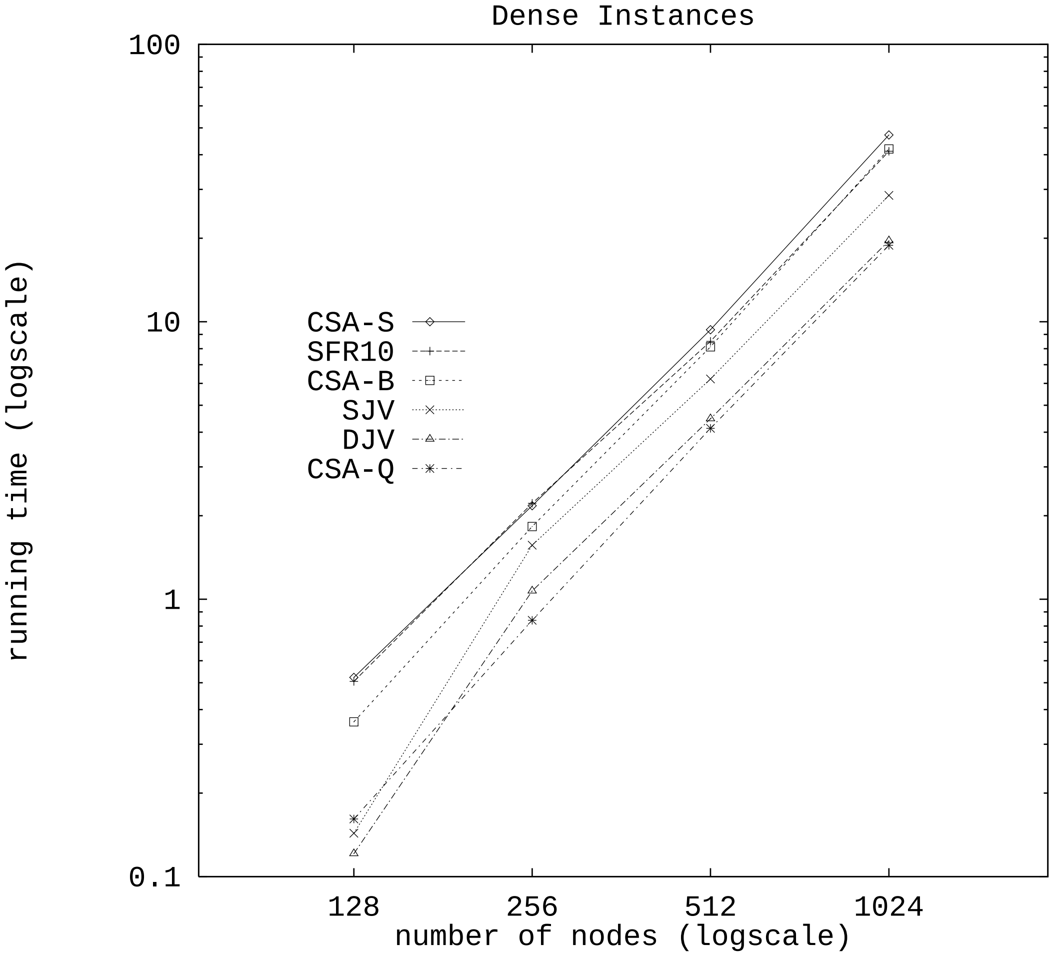
FIGURE 12. Running Times (15-instance samples) for the Geometric Class

As in the case of geometric problems, SJV and DJV seem to have asymptotic performance similar to the scaling and interior-point codes on this class.

6.7. Picture Problems. Although the pictures used had very similar characteristics, the tentative conclusions we draw here about the relative performance of the codes seem to apply to a broader class of images. We performed trials on a variety of images generated and transformed by various techniques, and found no substantial differences in relative performance, although some pictures seem to yield more difficult assignment problems than others. On the picture problems we tried, SFR10 performs better than any of the CSA implementations; we believe that the "reverse-auction" phases performed by SFR10 [5] are critical to this performance difference. We were unable to obtain times for SJV and CSA-Q on the largest problem instance from each picture, nor from SFR10 on the largest problem instance from one of the pictures because the codes required too much memory. On the second-largest instance from each picture, our experiments suggested that SJV would require more than a day of CPU time, so we did not collect data for these cases. On picture problems CSA-Q performs significantly worse than either of the other two CSA implementations. This situation is no surprise because CSA-Q performs an additional pointer dereference each time it examines an arc. In such a sparse graph, the 4 arcs stored at each node exhaust the list of arcs incident to that node, so no benefit is to be had from the kth-best heuristic.

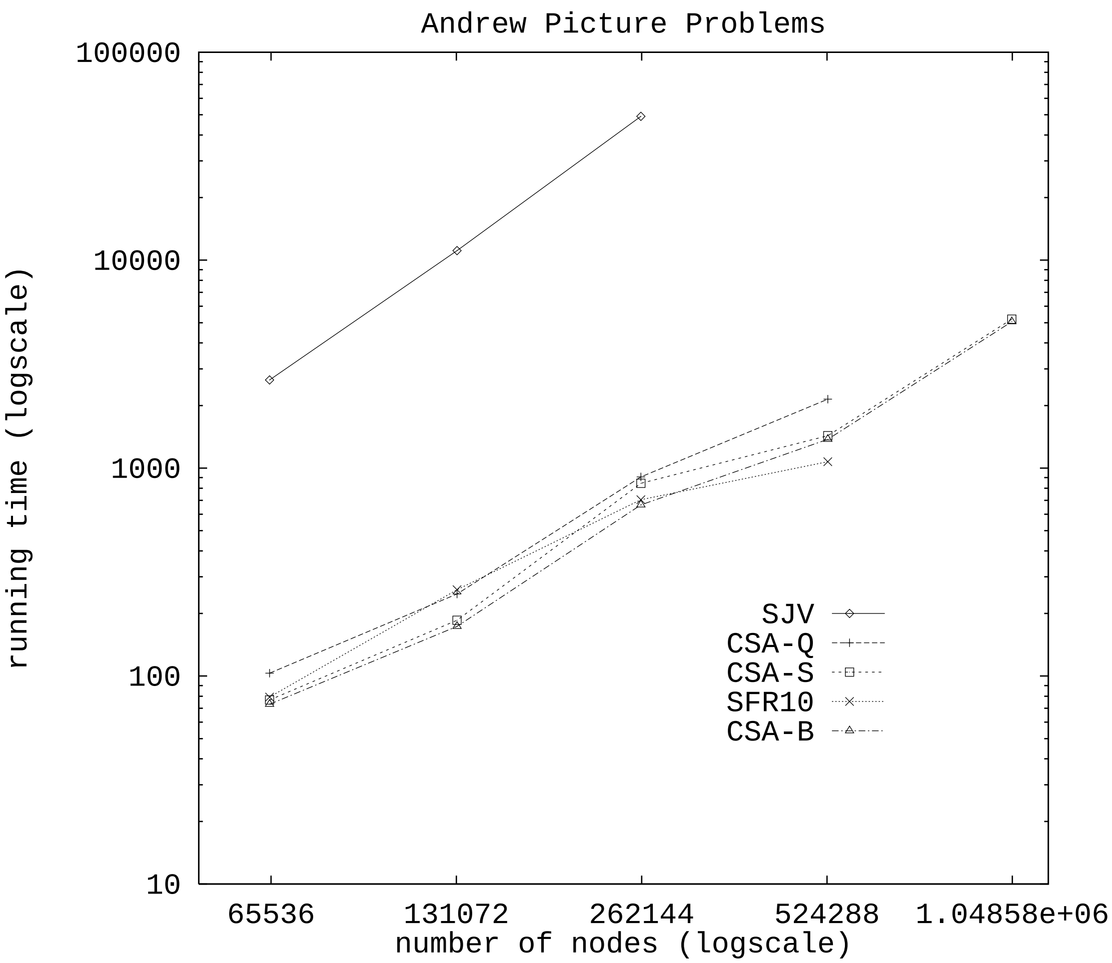
7. CONCLUDING REMARKS

Castañon [5] gives running times for an auction code called SF5 in addition to performance data for SFR10; SF5 and SFR10 are the fastest among the robust codes discussed. The data in [5] show that on several classes of problems, SF5 outperforms SFR10 by a noticeable margin. Comparing Castañon's reported running times for SFR10 with the data we obtained for the same code allows us to estimate roughly how SF5 performs relative to our codes. The data indicate that CSA-S and CSA-Q should perform at least as well as SF5 on all classes for which data are available, and that CSA-Q should outperform SF5 by a wide margin on some classes. A possible source of error in this technique of estimation is that Castañon reports times for test runs on cost-minimization problems, whereas all the



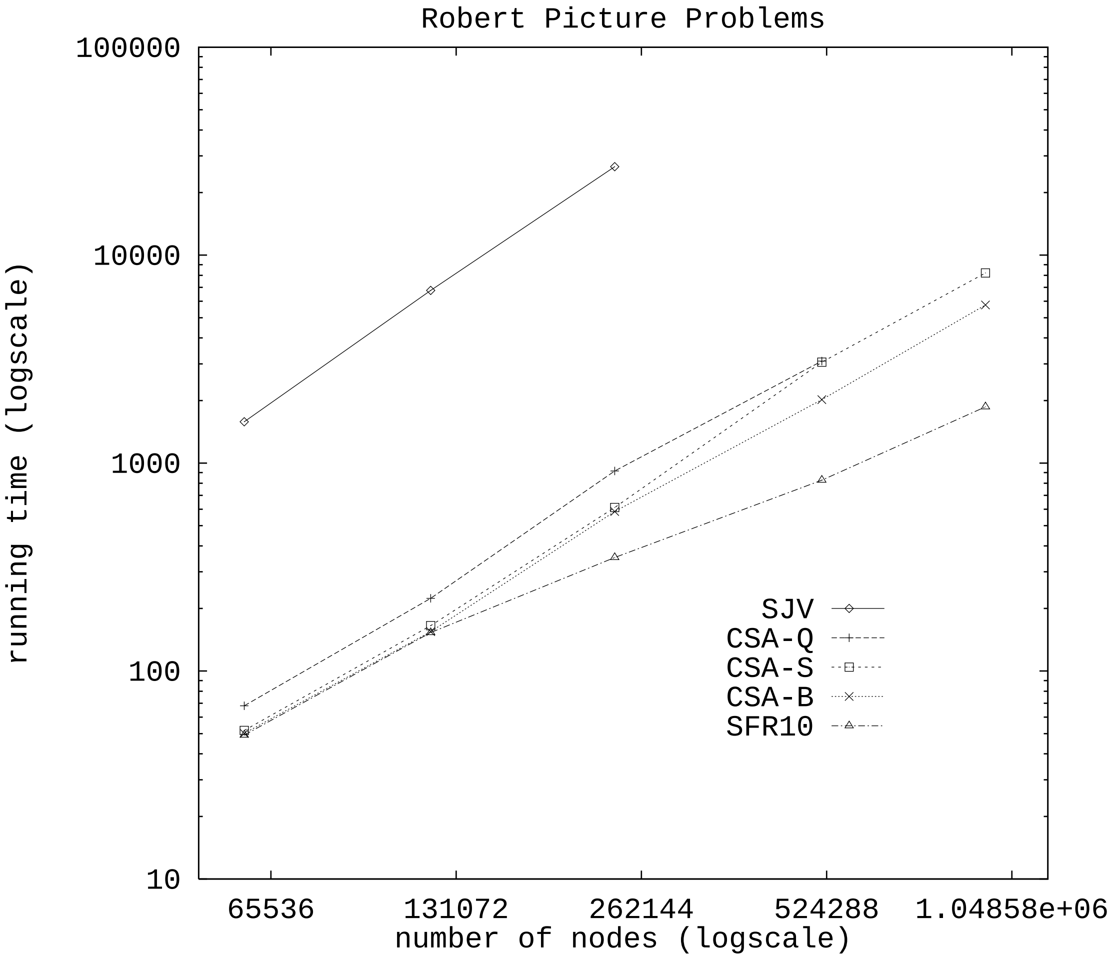
Nodes ($\|X\|$)	SFR10 time	SJV time	DJV time	CSA-B time	CSA-S time	CSA-Q time
128	**0.51**	**0.14**	**0.12**	**0.36**	**0.52**	**0.16**
256	**2.22**	**1.57**	**1.07**	**1.83**	**2.17**	**0.84**
512	**8.50**	**6.22**	**4.47**	**8.12**	**9.36**	**4.13**
1024	**41.2**	**28.5**	**19.6**	**42.0**	**47.1**	**18.9**

FIGURE 13. Running Times for the Dense Class



Nodes ($\|X\|$)	SFR10 time	SJV time	CSA-B time	CSA-Q time	CSA-S time
65158	**79.20**	**2656**	**73.23**	**103.3**	**76.70**
131370	**260.2**	**11115**	**173.2**	**248.0**	**185.5**
261324	**705.2**	**49137**	**665.1**	**907.8**	**844.8**
526008	**1073**	**N/A**	**1375**	**2146**	**1432**
1046520	**N/A**	**N/A**	**5061**	**N/A**	**5204**

FIGURE 14. Running Times for Problems from Andrew's Picture



Nodes (|X|)	SFR10 time	SJV time	CSA-B time	CSA-Q time	CSA-S time
59318	**49.17**	**1580**	**50.13**	**68.10**	**51.82**
119132	**153.1**	**6767**	**154.8**	**223.6**	**165.4**
237272	**351.4**	**26637**	**585.0**	**916.8**	**611.2**
515088	**827.8**	**N/A**	**2019**	**3095**	**3057**
950152	**1865**	**N/A**	**5764**	**N/A**	**8215**

FIGURE 15. Running Times for Problems from Robert's Picture

codes we test here (including SFR10) are configured to maximize cost. The difference in every case is but a single line of code, but while on some classes minimization and maximization problems are similar, on other classes we observed that minimization problems were significantly easier for all the codes. This difference is unlikely to be a large error source, however, since the relative performance of the codes we tested was very similar for minimization problems and maximization problems.

It is interesting that SJV is asymptotically worse than all its competitors on every sparse class, and that SJV and DJV are asymptotically very similar to their competitors on the dense classes. DJV performs very well on the uniform dense problem class, but we feel SJV provides a more genuine reference point, since the other combinatorial codes could be sped up on dense problems by replacing their central data structures with an adjacency matrix representation similar to that in DJV.

From our tests and data from [24] and [5], we conclude that CSA-Q is a robust, competitive implementation that should be considered for use by those who wish to solve assignment problems in practice.

Acknowledgments

The authors would like to thank David Castañon for supplying and assisting with the SFR10 code, Anil Kamath and K. G. Ramakrishnan for their assistance in interpreting results reported in [24], Jianxiu Hao for supplying and assisting with the SJV and DJV implementations, and Serge Plotkin for his help producing the digital pictures. The second author would like to thank Scott Burson for providing computing facilities during the development of the codes.

References

1. R. J. Anderson and J. C. Setubal. Goldberg's Algorithm for the Maximum Flow in Perspective: a Computational Study. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 1–18. AMS, 1993.
2. D. P. Bertsekas. The Auction Algorithm: A Distributed Relaxation Method for the Assignment Problem. *Annals of Oper. Res.*, 14:105–123, 1988.
3. D. P. Bertsekas. *Linear Network Optimization: Algorithms and Codes*. MIT Press, 1991.
4. R. G. Bland, J. Cheriyan, D. L. Jensen, and L. Ladańyi. An Empirical Study of Min Cost Flow Algorithms. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 119–156. AMS, 1993.
5. D. A. Castañon. Reverse Auction Algorithms for the Assignment Problems. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 407–430. AMS, 1993.
6. U. Derigs. The Shortest Augmenting Path Method for Solving Assignment Problems – Motivation and Computational Experience. *Annals of Oper. Res.*, 4:57–102, 1985/6.
7. U. Derigs and W. Meier. Implementing Goldberg's Max-Flow Algorithm — A Computational Investigation. *ZOR — Methods and Models of Operations Research*, 33:383–403, 1989.

8. S. Fujishige, K. Iwano, J. Nakano, and S. Tezuka. A Speculative Contraction Method for the Minimum Cost Flows: Toward a Practical Algorithm. The First DIMACS International Implementation Challenge, 1991.
9. H. N. Gabow and R. E. Tarjan. Faster Scaling Algorithms for Network Problems. *SIAM J. Comput.*, pages 1013–1036, 1989.
10. A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD thesis, M.I.T., January 1987. (Also available as Technical Report TR-374, Lab. for Computer Science, M.I.T., 1987).
11. A. V. Goldberg. An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm. In *Proc. 3rd Integer Prog. and Combinatorial Opt. Conf.*, pages 251–266, 1993.
12. A. V. Goldberg and R. Kennedy. Global Price Updates Help. Technical Report STAN-CS-94-1509, Department of Computer Science, Stanford University, 1994.
13. A. V. Goldberg and M. Kharitonov. On Implementing Scaling Push-Relabel Algorithms for the Minimum-Cost Flow Problem. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 157–198. AMS, 1993.
14. A. V. Goldberg, S. A. Plotkin, and P. M. Vaidya. Sublinear-Time Parallel Algorithms for Matching and Related Problems. *J. Algorithms*, 14:180–213, 1993.
15. A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988.
16. A. V. Goldberg and R. E. Tarjan. Finding Minimum-Cost Circulations by Successive Approximation. *Math. of Oper. Res.*, 15:430–466, 1990.
17. D. S. Johnson and C. C. McGeoch, editors. *Network Flows and Matching: F1rst DIMACS Implementation Challenge*. AMS, 1993.
18. R. Jonker and A. Volgenant. A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems. *Computing*, 38:325–340, 1987.
19. D. Knuth. Personal communication. 1993.
20. H. W. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Res. Logist. Quart.*, 2:83–97, 1955.
21. E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Reinhart, and Winston, New York, NY., 1976.
22. Q. C. Nguyen and V. Venkateswaran. Implementations of Goldberg-Tarjan Maximum Flow Algorithm. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 19–42. AMS, 1993.
23. J. B. Orlin and R. K. Ahuja. New Scaling Algorithms for the Assignment and Minimum Cycle Mean Problems. Sloan Working Paper 2019-88, Sloan School of Management, M.I.T., 1988.
24. K. G. Ramakrishnan, N. K. Karmarkar, and A. P. Kamath. An Approximate Dual Projective Algorithm for Solving Assignment Problems. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 431–452. AMS, 1993.

Appendix A. Generator Inputs

The assignment instances on which we ran our tests were generated as follows: Problems in the high-cost, low-cost, fixed-cost, and dense classes were generated using the DIMACS generator `assign.c`. Problems in the two-cost class were generated using `assign.c` with output post-processed by the DIMACS `awk` script `twocost.a`. Problems in the geometric class were generated using the DIMACS generator `dcube.c` with output post-processed by the DIMACS `awk` script `geomasn.a`. Picture problems were generated from images in the Portable Grey Map format using our program `p5pgmtoasn`. To obtain the DIMACS generators, use anonymous `ftp` to `dimacs.rutgers.edu`, or obtain the `csa` package (which includes the generators) as described below.

In each class except the picture class, we generated instances of various numbers of nodes N and using various seeds K for the random number generator. For each problem type and each N, either three or 15 values of K were used; the values were integers 270001 through 270003 or through 270015. For picture problems, we tested the codes on a single instance of each size.

A.1. The High-Cost Class. We generated high-cost problems using `assign.c` from the DIMACS distribution. The input parameters given to the generator are as follows, with the appropriate values substituted for N and K:

`nodes` N

`sources` $N/2$

`degree` $2\log_2 N$

`maxcost` 100000000

`seed` K

A.2. The Low-Cost Class. Like high-cost problems, low-cost problems are generated using the DIMACS generator `assign.c`. The parameters to the generator are identical to those for high-cost problems, except for the maximum edge cost:

`nodes` N

`sources` $N/2$

`degree` $2\log_2 N$

`maxcost` 100

`seed` K

A.3. The Two-Cost Class. Two-cost instances are derived from low-cost instances using the Unix `awk` program and the DIMACS `awk` script `twocost.a`. The instance with N nodes and seed K was generated using the following Unix command line, with input parameters identical to those for the low-cost problem class:

```
assign | awk -f twocost.a
```

A.4. The Fixed-Cost Class. We generated fixed-cost instances using `assign.c`, with input parameters as follows:

`nodes` N

`sources` $N/2$

`degree` $N/16$

`maxcost` 100

`multiple`

`seed` K

A.5. The Geometric Class. We generated geometric problems using the DIMACS generator `dcube.c` and the DIMACS `awk` script `geomasn.a`. We gave input parameters to `dcube` as shown below, and used the following Unix command line:

```
dcube | awk -f geomasn.a
```

`nodes` N

`dimension` 2

`maxloc` 1000000

`seed` K

A.6. The Dense Class. We generated dense problems using `assign.c`, with input parameters as follows:

`nodes` N

`sources` $N/2$

`complete`

`maxcost` 1000000

`seed` K

Appendix B. Obtaining the CSA Codes

To obtain a copy of the CSA codes, DIMACS generators referred to in this paper, and documentation files, send mail to `ftp-request@theory.stanford.edu` and use `send csas.tar` as the subject line; you will automatically be mailed a `uuencode`d copy of a `tar` file.