March 1992 **LIDS - P - 2098**

Appeared in Networks, Vol. 23, pp. 703-709, 1993

A Simple and Fast Label Correcting Algorithm for Shortest Paths[1]

by

Dimitri P. Bertsekas[2]

Abstract

We propose a new method for ordering the candidate nodes in label correcting methods for shortest path problems. The method is equally simple but much faster than the D' Esopo-Pape algorithm. It is similar to the threshold algorithm in that it tries to scan nodes with small labels as early as possible, and performs comparably with that algorithm. Our algorithm can also be combined with the threshold algorithm thereby considerably improving the practical performance of both algorithms.

[1] Research supported by NSF under Grants No. DDM-8903385 and CCR-9108058.

[2] Laboratory for Information and Decision Systems, M.I.T, Cambridge, Mass. 02139.

1. A NEW NODE SELECTION STRATEGY FOR LABEL CORRECTING METHODS

In this paper we propose a new label correcting method for finding shortest paths in a directed graph. The set of nodes is denoted $\mathcal{N}$ and the set of arcs is denoted $\mathcal{A}$. The numbers of nodes and arcs are denoted A and N, respectively. The nodes are numbered $1, \ldots, N$. Each arc $(i, j) \in \mathcal{A}$ has a cost or "length" a_{ij} associated with it. The length of a path $(i_1, i_2, \ldots, i_k)$, which consists exclusively of forward arcs, is equal to the length of its arcs

$$\sum_{n=1}^{k-1} a_{i_n i_{n+1}}.$$

We want to find a shortest (minimum length) path from a single origin (node 1) to all other nodes. We assume throughout that there exists a path from the origin to each other node and that all cycles have nonnegative length. This guarantees that the problem has a solution.

Most of the major shortest path methods can be viewed as special cases of a prototype shortest path algorithm given by Gallo and Pallottino [GaP86], [GaP88]. The algorithm maintains a label vector $(d_1, d_2, \ldots, d_N)$, starting with

$$d_1 = 0, \qquad d_i = \infty, \qquad \forall\ i \neq 1, \tag{1}$$

and a set of nodes V, called the *candidate list*, starting with

$$V = \{1\}. \tag{2}$$

The algorithm proceeds in iterations and terminates when V is empty. The typical iteration (assuming V is nonempty) is as follows:

Typical Iteration of the Generic Shortest Path Algorithm

> Remove a node i from the candidate list V. For each outgoing arc $(i, j) \in \mathcal{A}$, with $j \neq 1$, if $d_j > d_i + a_{ij}$, set
>
> $$d_j := d_i + a_{ij} \tag{3}$$
>
> and add j to V if it does not already belong to V.

Different algorithms are distinguished by the method of selecting the node to exit the candidate list V at each iteration. In one major class, the *label setting* or *Dijkstra* methods, the node exiting V is a node whose label is minimum over all other nodes in V. Methods that do not follow this node selection policy are called *label correcting*. There are several label setting

algorithms, which are distinguished by the data structures used to compute the minimum label node from V; see [GaP86], [GaP88], or [Ber91a]. When the arc lengths a_{ij} are nonnegative, these methods require N iterations; each node $i \neq 1$ enters and exits V exactly once. Otherwise, the number of iterations can be proportional to 2^N as shown by example in Shier and Witzgall [ShW81].

In label correcting methods, the selection of the node to be removed from V is faster than in label setting methods, at the expense of multiple entrances of nodes in V. These methods use a queue Q to maintain the candidate list V. Nodes can be inserted or removed from the queue in $O(1)$ operations; some kind of linked list data structure is needed for this. At each iteration the node removed from V is the top node of Q. The methods differ in the method for choosing the queue position to insert a node that enters V. We describe three popular methods.

a) *The Bellman-Ford method* (actually a variant of the original method of [Bel57] and [For56]). Here, the node that enters V, is added at the bottom of Q. Thus, nodes enter and exit V in first-in/first-out fashion. This method can be shown to require at most N^2 iterations and $O(NA)$ operations. Despite the generally larger number of iterations required by the Bellman-Ford method over Dijkstra's method, in practice the Bellman-Ford method can be superior because of the smaller overhead per iteration [Gol76]. Note also that for acyclic graphs the number of iterations of the Bellman-Ford method is exactly N, the same as for Dijkstra's method. However, our discussions in this paper implicitly assume that the graph is not acyclic, so that there is significant room for reduction of the number of iterations required by the Bellman-Ford algorithm.

b) *The D'Esopo-Pape method* [Pap74]. Here a node that enters V for the first time is added to Q at the bottom; a node that reenters V is added to Q at the top. The number of iterations required by this method is proportional to 2^N in worst case, even when the arc lengths are nonnegative, as shown through examples by Kershenbaum [Ker81], and Shier and Witzgall [ShW81]. Despite this fact, the D'Esopo-Pape algorithm performs very well in practice. For sparse graphs, it usually outperforms the Bellman-Ford method, and it is competitive with the best label setting methods [DGK79], [GaP88]. No definitive explanation has been given for this behavior. We will refer to the original version of the D'Esopo-Pape algorithm as the *1st version* to distinguish it from another polynomial version [$O(N^2A)$ operations] given in [Pal79], [Pal84], [GaP88], which we refer to as the *2nd version*. In the latter version the queue Q is partitioned in two disjoint queues, Q_1 and Q_2; the node exiting V is the top node of Q_1 if Q_1 is nonempty, and otherwise it is the top node of Q_2; a node that enters V for the first time is added at the bottom of Q_2; a node that reenters V is added at the

bottom of Q_1. The practical performance of the two versions is roughly comparable based on the experiments of [GaP88].

c) *The threshold algorithm* of Glover, Glover, and Klingman [GGK86]. Here the queue Q is partitioned in two disjoint queues, Q_1 and Q_2. At each iteration, the node removed from V is the top node of Q_1 and a node entering V is added to the bottom of Q_2 or to the bottom of Q_1 depending on whether its label exceeds or does not exceed a certain threshold parameter, respectively. When Q_1 becomes empty, the nodes of Q_2 whose labels do not exceed the current threshold parameter are removed from Q_2 and entered into Q_1. The threshold parameter is adjusted to a level above the minimum of the labels of nodes in Q_2 according to some heuristic scheme; see [GGK85b] and [GaP88]. The algorithm requires at most N^3 iterations and $O(N^2A)$ operations; see [GaP88], p. 20. We call the preceding algorithm the *1st version* of the threshold method to distinguish it from another and apparently less effective version, which we call the *2nd version* of the threshold algorithm. In the 2nd version, a node entering V is always added to the bottom of Q_2, regardless of whether its label exceeds the current threshold or not. When the arc lengths are nonnegative, this algorithm requires at most N^2 iterations and $O(NA)$ operations; see [GKP85a], [Ber91]. The 1st version of the threshold algorithm has performed extremely well in computational tests with randomly generated problems [GKP85b], [GaP88]. However, its performance is quite sensitive to the threshold adjustment scheme as well as to the cost structure of the problem. In particular, if the threshold is chosen too small, the method becomes equivalent to an unsophisticated version of Dijkstra's algorithm, while if the threshold is chosen too large, the method becomes equivalent to the Bellman-Ford method. Thus one may have to experiment with the threshold selection policy for a given class of problems, and even after considerable experimentation, one may be unable to find an effective adjustment scheme; as an example, in the Euclidean grid/random problems discussed in the next section it is difficult to fine-tune the threshold selection because of the large cost range. It should be noted, however, that a particular method to select the threshold, given in [GKP85b] and used in our experiments, has proved very effective for broad classes of randomly generated problems.

The new method proposed in this paper is based on the hypothesis that for many types of problems, *the number of iterations of a label correcting method strongly depends on the average rank of the node exiting V*, where nodes are ranked in terms of the size of their label (nodes with small labels have small rank). Thus, for good performance, the queue insertion strategy used should try to place nodes with small labels near the top of the queue. The performances of label setting and threshold methods are consistent with this hypothesis. We offer additional

experimental evidence for our hypothesis in the next section. For a supporting heuristic argument, note that for a node j to reenter V, some node i such that $d_i + a_{ij} < d_j$ must first exit V. Thus, the smaller d_j was at the previous exit of j from V the less likely it is that $d_i + a_{ij}$ will subsequently become less than d_j for some node $i \in V$ and arc (i, j). In particular, if $d_j \leq \min_{i \in V} d_i$, and the arc lengths a_{ij} are nonnegative, it is impossible that subsequent to the exit of j from V we will have $d_i + a_{ij} < d_j$ for some $i \in V$.

We now formally describe our algorithm. It is the label correcting method that uses the following strategy for inserting nodes in the queue Q, called *Small Label to the Front* (SLF for short):

SLF Strategy

> Whenever a node j enters Q, its label d_j is compared with the label d_i of the top node i of Q. If $d_j \leq d_i$, node j is entered at the top of Q; otherwise j is entered at the bottom of Q.

The SLF strategy can also be combined with the 1st version of the threshold algorithm. In particular, whenever a node j enters the queue Q_1, it is added to the top or the bottom of Q_1 depending on whether $d_j \leq d_i$ or $d_j > d_i$, where i is the top node of Q_1. This policy is also used when transferring to Q_1 the nodes of Q_2 whose label does not exceed the current threshold parameter; that is, when Q_1 becomes empty, the nodes of Q_2 are checked sequentially from first to last, and if a node j satisfies the test for entry into Q_1, it is inserted at the top or the bottom of Q_1 depending on whether $d_j \leq d_i$ or $d_j > d_i$, where i is the top node of Q_1. Also, whenever a node j enters the queue Q_2, it is added to the top or the bottom of Q_2 depending on whether $d_j \leq d_i$ or $d_j > d_i$, where i is the top node of Q_2. We call the corresponding label correcting method the *SLF-threshold method*.

It is also possible to consider several variations of the SLF strategy. For example, whenever the label of a node j that is already in the queue Q (or Q_1 or Q_2, in the threshold case) is decreased, one may compare the new label d_j with the label d_i of the top node of the queue, and if $d_j < d_i$, move j to the front of the queue. This requires a doubly linked list to maintain the queue and there is substantial associated extra overhead. However, based on our preliminary experiments, this leads to further reduction of the number of iterations; this is consistent with our hypothesis of correlation between number of iterations and average rank of the node exiting V. We have not experimented sufficiently with this or other related schemes to conclude whether and for what types of problems the reduction in number of iterations is worth the extra overhead.

The worst-case computational complexity of the SLF algorithms is a subject of ongoing research. It is possible to modify the SLF-threshold algorithm along the lines of the 2nd version

of the threshold algorithm to obtain an $O(NA)$ running time bound for the case of nonnegative arc lengths; however, the modified algorithm is not as practically efficient as the one described earlier, which is patterned after the 1st version of the threshold algorithm. It is an interesting open question whether the SLF and SLF-threshold algorithms are polynomial. From examples that we have constructed, we know that the worst-case complexity of these algorithms is worse than the $O(NA)$ bound of the Bellman-Ford algorithm. This makes the worst-case complexity question somewhat mute. In the next section we compare computationally the SLF and SLF-threshold algorithms with existing methods.

2. COMPUTATIONAL EXPERIMENTS

We have coded the SLF and SLF-threshold algorithms by modifying in a minimal way the codes LDEQUE and LTHRESH of Gallo and Pallottino [GaP88], which implement the 1st versions of the D'Esopo-Pape and the threshold algorithms, respectively. In summary, the results are very encouraging for our algorithms. In particular, in our experiments, the SLF algorithm is consistently faster than the D'Esopo-Pape method and requires consistently fewer iterations than the Bellman-Ford method. The SLF-threshold algorithm also requires consistently fewer iterations than the threshold algorithm, although both methods often perform so well that their effectiveness is indistinguishable. However, for problems where choosing an appropriate threshold is difficult, the SLF-threshold algorithm is significantly faster than the threshold algorithm.

We have tested the following five codes. The first three were obtained by minor modifications (a few FORTRAN statements) of the LDEQUE code of [GaP88], and the last two by minor modifications of the LTHRESH code of [GaP88].

B-F: This implements the Bellman-Ford method.

D'E-P: This implements the D'Esopo-Pape method.

SLF: This implements our SLF method.

THR: This implements the threshold method.

SLF-THR: This implements our SLF-threshold method.

Note that we have maintained intact the threshold adjustment scheme of the code LTHRESH. This scheme was suggested in [GKP85b] and is as follows:

Initially the threshold parameter, denoted *thresh*, is set at -1. When the queue Q_1 becomes

empty we update *thresh* according to

$$thresh = \begin{cases} thresh + t + 1 & \text{if } dmin \leq thresh + t + 1, \\ dmin + t & \text{otherwise,} \end{cases}$$

where

$$dmin = \min_{i \in Q_2} d_i,$$

$$t = \begin{cases} x \cdot lmax & \text{if } s \leq 7, \\ 7x \cdot lmax/s & \text{otherwise,} \end{cases}$$

$$s = \min\{A/N, 35\},$$

$lmax$ is the maximum arc length, and the value of x is chosen on the basis of the problem structure. We have used the recommended value for random graphs $x = 0.25$, and this has worked well for all types of problems tested, except for the Euclidean grid/random graphs (see below), for which smaller values of x produced a reduction of the number of iterations. However, for these problems, the optimal value of x was highly problem dependent.

We tested the five codes on several types of randomly generated single origin/all destination problems. In all cases the origin was node 1, and with the exception of the Euclidean grid/random graphs described below, all the arc lengths were integer and were chosen by a uniform distribution from the range [1,1000]. All times refer to a 25MHz Macintosh, where the programs were compiled using the Absoft compiler. Generally, the execution time is roughly proportional to the number of iterations, but the Bellman-Ford method requires less overhead per iteration than the D'Esopo-Pape and SLF algorithms, which in turn require less overhead per iteration than the threshold algorithms. The results are as follows:

NETGEN Problems

These are problems generated by the popular public domain generator NETGEN [KNS74]. The graph density was 2% in all cases ($A = 0.02 \cdot N^2$). The execution times and the numbers of iterations for the five codes are given in Table 1. It can be seen that for these problems the two threshold algorithms are much faster than the others.

Grid/Random Problems

These are problems generated by a modified version of the GRIDGEN generator of [Ber91a]. Here the nodes are arranged in a square planar grid with the origin node 1 being the southwest corner of the grid. There is a grid arc connecting each pair of adjacent grid nodes in each direction.

N	A	**B-F**	**D'E-P**	**SLF**	**THR**	**SLF-THR**
500	5000	0.117 / 992	0.100 / 995	0.083 / 750	0.066 / 517	0.050 / 513
1000	20000	0.467 / 2516	0.583 / 3066	0.383 / 1956	0.200 / 1037	0.200 / 1036
1500	45000	1.250 / 4071	1.820 / 5270	1.130 / 3184	0.533 / 1632	0.433 / 1577
2000	80000	1.983 / 5044	2.683 / 5931	2.017 / 4281	0.867 / 2066	0.717 / 2058

Table 1: Time in secs/number of iterations required to solve NETGEN problems. All arc lengths are chosen according to a uniform distribution from the range [1,1000].

N	A	**B-F**	**D'E-P**	**SLF**	**THR**	**SLF-THR**
2500	14800	0.417 / 5690	0.400 / 5004	0.333 / 4260	0.217 / 2578	0.233 / 2560
5625	33450	0.933 / 11957	0.917 / 11356	0.717 / 8568	0.500 / 5755	0.533 / 5733
10000	59600	1.933 / 23471	1.767 / 21003	1.483 / 17001	0.950 / 10275	1.017 / 10226
15625	93250	3.333 / 40231	2.750 / 31822	2.100 / 23574	1.500 / 15833	1.620 / 15776

Table 2: Time in secs/number of iterations required to solve grid/random problems. The number of nongrid arcs is $2 \cdot N$. All arc lengths are chosen according to a uniform distribution from the range [1,1000].

Also there are $2 \cdot N$ additional arcs with random starting node and random ending node. The execution times and the numbers of iterations for the five codes are given in Table 2. It can be seen that for these problems the two threshold algorithms are again much faster than the others.

Euclidean Grid/Random Problems

In these problems the nodes and arcs were generated in the same way as in the preceding grid/random problems. The length of each arc connecting grid node (i, j) to grid node (k, l) is $r \cdot e_{ij,kl}$, where $e_{ij,kl}$ is the Euclidean distance

$$e_{ij,kl} = \sqrt{(i-k)^2 + (j-l)^2},$$

and r is an integer chosen according to a uniform distribution from the range $[1, 1000]$. The execution times and the numbers of iterations for the five codes are given in Table 3. There several surprises here. First, the D'Esopo-Pape algorithm performs very poorly; we have not

seen in the literature any report of a class of randomly generated sparse problems where this algorithm exhibits such poor behavior. Second, the threshold and SLF-threshold algorithms work only slightly better than the Bellman-Ford and SLF algorithms, respectively, because the threshold adjustment scheme is not working effectively (the cost range here is very broad). We have therefore conducted some experimentation with the parameter x of the threshold adjustment scheme, and we were able to reduce the number of iterations of the threshold and SLF-threshold algorithms (see Table 4). However, the optimal value of x was highly problem dependent and varied by several orders of magnitude depending on the number of nongrid arcs, as can be seen from Table 4. Note that for this class of problems, the SLF-threshold algorithm is considerably faster than the others, except when the threshold is set to a very low value.

N	A	**B-F**	**D'E-P**	**SLF**	**THR**	**SLF-THR**
2500	14800	1.500 / 20485	6.650 / 91002	1.280 / 16472	1.470 / 21694	1.150 / 16367
5625	33450	7.280 / 96223	332.2 / 4487805	5.400 / 67828	6.420 / 92316	4.430 / 62143
10000	59600	14.60 / 187703	279.2 / 3723865	10.30 / 127625	12.73 / 178212	8.667 / 118979
15625	93250	19.97 / 255349	326.0 / 4145800	13.88 / 169516	18.00 / 250200	11.95 / 161669

Table 3: Time in secs/number of iterations required to solve Euclidean grid/random problems. The number of nongrid arcs is $2 \cdot N$. All grid arc lengths are chosen according to a uniform distribution from the range [1,1000]. The length of each nongrid arc connecting node (i, j) to node (k, l) is $r \cdot e_{ij,kl}$, where $e_{ij,kl}$ is the Euclidean distance $e_{ij,kl} = \sqrt{(i-k)^2 + (j-l)^2}$ and r is an integer chosen according to a uniform distribution from the range $[1, 1000]$.

Fully Dense Problems

In these problems all the possible $N(N-1)$ arcs are present. The computational study [GaP88] showed that high problem density favors label setting over label correcting methods. It is therefore interesting to test whether the SLF strategy increases the effectiveness of label correcting methods to the point where they can challenge the best label setting methods. We have thus compared in Table 4 the five label correcting codes with the code SHEAP of [GaP88], which is a label setting method based on a binary heap implementation. SHEAP gave the best performance for fully dense problems in the tests of [GaP88]. We have also included a comparison with AUCT-GR, which is an implementation of a version of the author's auction algorithm for shortest paths [Ber91b]. This version uses graph reduction as developed by Bertsekas, Pallottino, and Scutella' [BPS92], has complexity $O(N^2 \log N)$, and is particularly effective for dense problems. The

N	A	**Method**	$x = .25$	$x = .025$	$x = .0025$	$x = .00025$	$x = .000025$
2500	9801	THR	0.200 / 4294	0.150 / 2665	0.233 / 2501	0.800 / 2500	1.317 / 2500
		SLF-THR	0.183 / 3301	0.150 / 2564	0.267 / 2502	0.883 / 2500	1.433 / 2500
2500	14800	THR	1.470 / 21694	1.017 / 15272	0.367 / 5240	0.233 / 2674	0.567 / 2503
		SLF-THR	1.150 / 16367	0.783 / 11242	0.317 / 4067	0.267 / 2658	0.617 / 2501
2500	29800	THR	2.450 / 20249	1.733 / 14256	0.617 / 4539	0.550 / 3993	0.967 / 2500
		SLF-THR	1.800 / 14798	1.300 / 10637	0.550 / 3993	0.443 / 2607	1.017 / 2501
10000	39601	THR	2.117 / 44332	1.517 / 31733	0.733 / 14422	0.650 / 10149	1.367 / 10002
		SLF-THR	0.867 / 16710	0.817 / 15358	0.650 / 11468	0.717 / 10116	1.517 / 10001
10000	59600	THR	12.73 / 178212	11.33 / 159890	5.917 / 82770	1.233 / 14576	1.533 / 10229
		SLF-THR	8.667 / 118979	7.800 / 107661	4.783 / 64564	1.200 / 13125	1.650 / 10198
10000	99600	THR	16.67 / 130108	13.52 / 105622	7.883 / 60626	2.117 / 13893	2.550 / 10151
		SLF-THR	11.05 / 86259	10.83 / 84386	6.067 / 45971	2.000 / 12604	2.683 / 10141

Table 4: Time in secs/number of iterations required to solve Euclidean grid/random problems with the threshold and the SLF-threshold algorithms using different values of the threshold parameter x. The six problems have 1, $2 \cdot N$, $8 \cdot N$, 1, $2 \cdot N$, and $6 \cdot N$ nongrid arcs, respectively.

execution times for the seven codes are given in Table 5. Again, the D'Esopo-Pape algorithm performs poorly relative to the Bellman-Ford method, similar to the results of [GaP88]. The SLF strategy is particularly effective for these dense problems. In particular, the SLF-threshold algorithm is much faster than the threshold algorithm and slightly outperforms the heap-based label seting algorithm. However, the auction code maintains an edge over all the other codes.

Correlation of Average Rank and Number of Iterations

We mentioned earlier that the ideas of this paper are based on the hypothesis that the number of iterations of a label correcting method strongly depends on how successful the method is in selecting nodes with relatively small labels to exit V. To substantiate experimentally this

N	**B-F**	**D'E-P**	**SLF**	**THR**	**SLF-THR**	**SHEAP**	**AUCT-GR**
150	0.483 / 400	1.000 / 639	0.550 / 344	0.300 / 223	0.200 / 191	0.250	0.200
200	0.883 / 550	1.783 / 854	1.033 / 480	0.733 / 394	0.383 / 290	0.400	0.350
250	1.233 / 626	2.333 / 894	1.560 / 581	0.950 / 410	0.650 / 389	0.617	0.517
300	1.750 / 745	3.567 / 1141	2.033 / 633	1.850 / 677	0.817 / 411	0.883	0.750

Table 5: Time in secs/number of iterations required to solve fully dense problems for the label correcting methods compared with the times of the label setting code SHEAP and the auction code AUCT-GR. All arc lengths are chosen according to a uniform distribution from the range [1,1000].

hypothesis, we have recorded for each iteration the ratio

$$\frac{\text{Number of remaining nodes in } V \text{ with label smaller than } d_i}{\text{Number of remaining nodes in } V},$$

where i is the node exiting V (the ratio is defined to be zero if there are no remaining nodes in V after i exits V). The *average rank of a method for a given problem* is the sum of these ratios over all iterations, divided by the number of iterations. Thus, the average rank of a label setting method is 0 for all problems, and the closer the average rank of a label correcting method is to 0, the more successful the method is in selecting nodes with relatively small label to exit V.

Figure 1 plots the average rank as a function of the number of iterations per node for the problems of Tables 1, 2, and 5, and the five label correcting methods. The results for the problems of Table 3 were qualitatively similar, but they were not plotted because the excessive number of iterations for the D'Esopo-Pape method would extend the horizontal axis of the plot excessively. Overall the SLF-threshold method attained consistently the smallest average rank as well as the smallest number of iterations. As Fig. 1 shows, the positive correlation between average rank and number of iterations is consistent and very strong.

REFERENCES

[BPS92] Bertsekas, D. P., Pallottino, S., and Scutella', M. G., 1992. "Polynomial Auction Algorithms for Shortest Paths," LIDS Report, M.I.T., in preparation.

[Bel57] Bellman, R., 1957. Dynamic Programming, Princeton Univ. Press, Princeton, N. J.

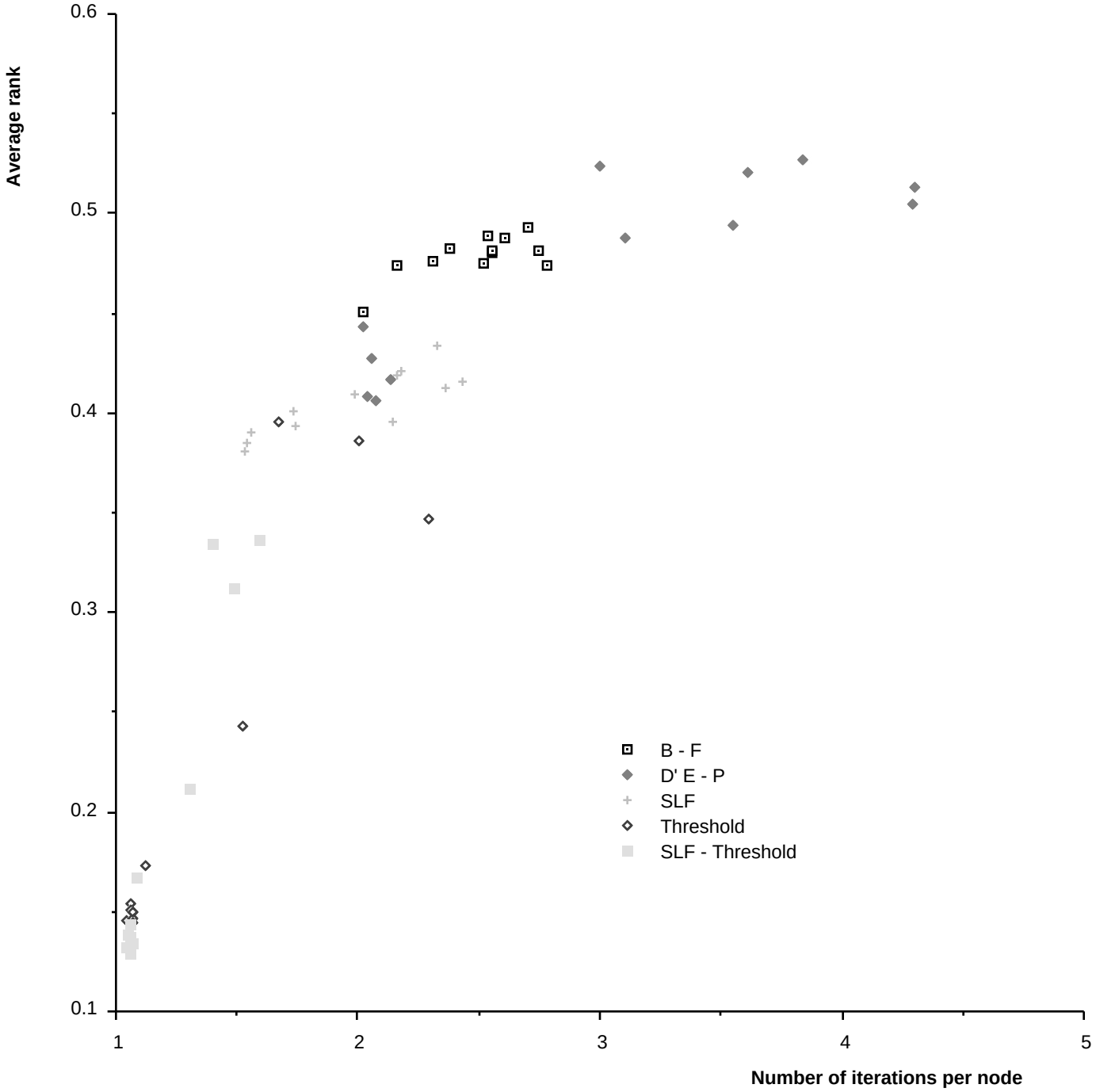

Figure 1: Plot of average rank and number of iterations per node (total number of iterations divided by the number of nodes) for the five label correcting methods tested. Each data point corresponds to a problem of Table 1 or 2 or 5.

[Ber91a] Bertsekas, D. P., 1991. Linear Network Optimization: Algorithms and Codes, M.I.T. Press, Cambridge, MA.

[Ber91b] Bertsekas, D. P., 1991. "The Auction Algorithm for Shortest Paths," SIAM J. on Optimization, Vol. 1, pp. 425-447.

[DGK79] Dial, R., Glover, F., Karney, D., and Klingman, D., 1979. "A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees," Networks, Vol. 9, pp. 215-248.

[For56] Ford, L. R., Jr., 1956. "Network Flow Theory," Report P-923, The Rand Corporation, Santa Monica, Cal.

[GKP85a] Glover, F., Klingman, D., and Phillips, N., 1985. "A New Polynomially Bounded Shortest Path Algorithm," Operations Research, Vol. 33, pp. 65-73.

[GKP85b] Glover, F., Klingman, D., Phillips, N., and Schneider, R. F., 1985. "New Polynomial Shortest Path Algorithms and Their Computational Attributes," Management Science, Vol. 31, pp. 1106-1128.

[GaP86] Gallo, G., and Pallottino, S., 1986. "Shortest Path Methods: A Unified Approach," Math. Programming Study, Vol. 26, pp. 38-64.

[GaP88] Gallo, G., and Pallottino, S., 1988. "Shortest Path Algorithms," Annals of Operations Research, Vol. 7, pp. 3-79.

[GGK86] Glover, F., Glover, R., and Klingman, D., 1986. "The Threshold Shortest Path Algorithm," Networks, Vol. 14, No. 1.

[Gol76] Golden, B., 1976. "Shortest-Path Algorithms: A Comparison," Operations Research, Vol. 44, pp. 1164-1168.

[KNS74] Klingman, D., Napier, A., and Stutz, J., 1974. "NETGEN - A Program for Generating Large Scale (Un) Capacitated Assignment, Transportation, and Minimum Cost Flow Network Problems," Management Science, Vol. 20, pp. 814-822.

[Ker81] Kershenbaum, A., 1981. "A Note on Finding Shortest Path Trees," Networks, Vol. 11, p. 399-400.

[Pal79] Pallottino, S., 1979. "Adaptation de l' Algorithme de D'Esopo-Pape pour la Determination de tous les Chemins le plus Courts: Ameliorations et Simplifications," Centre de Recherche sur les Transports, n. 136.

[Pal84] Pallottino, S., 1984. "Shortest Path Methods: Complexity, Interrelations and New Propositions," Networks, Vol. 14, pp. 257-267.

[Pap74] Pape, U., 1974. "Implementation and Efficiency of Moore - Algorithms for the Shortest Path Problem," Math. Programming, Vol. 7, pp. 212-222.

[ShW81] Shier, D. R., and Witzgall, C., 1981. "Properties of Labeling Methods for Determining Shortest Path Trees," J. Res. Natl. Bureau of Standards, Vol. 86, p. 317.