Evaluating a 2-approximation algorithm for edge-separators in planar graphs

Naveen Garg [*]   *Christian Manß* [†]

Abstract

In this paper we report on results obtained by an implementation of a 2-approximation algorithm for edge separators in planar graphs [1]. For 374 out of the 435 instances the algorithm returned the optimum solution. For the remaining instances the solution returned was never more than 10.6% away from the lower bound on the optimum separator. We also improve the worst-case running time of the algorithm from $O(n^6)$ to $O(n^5)$ and present techniques which improve the running time significantly in practice.

1 Introduction

Since the seminal work of Lipton and Tarjan [4] finding small separators in planar graphs has attracted the attention of theoreticians and practitioners alike. A *separator* in a graph is a set of vertices/edges whose removal disconnects the graph into two roughly equal pieces. Finding a small separator is important because it allows one to use a divide-and-conquer paradigm for solving a host of problems on graphs [5]; the small size of the separator ensures that the amount of work done in merging the solutions obtained on the two pieces is small.

Finding a minimum separator in a graph is **NP**-hard. Leighton and Rao [3] use multicommodity flow techniques to find a separator of cost at most $O(\log n)$ times the optimum balanced cut. For planar graphs it is not known if finding the minimum separator is **NP**-hard. Rao [7] showed how to compute the optimum simple cycle separator in a planar graph (this is a separator which is a simple cycle in the dual). In a surprising result, Park and Philipps [6] showed that for planar graphs one can in polynomial time compute a cut which minimizes sparsity (the ratio of the cost of the cut to the number of vertices on its smaller side).

By picking sparse cuts repeatedly till the number of vertices accumulated is at least $n/3$ (and at most $2n/3$) one can hope to find a small separator. However it is easy to construct examples where this approach can give a separator of cost $\log n$ times the optimum. Garg, Saran and Vazirani [1] introduced the notion of net-cost (which is just the incremental cost of picking the next set of vertices) and showed that this together with ideas from [6] and [8] leads to a 2-approximation algorithm for edge separators in planar graphs.

[*]Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany. Work supported by the EU ESPRIT LTR Project No. 20244 (ALCOM-IT).

[†]Universität des Saarlandes, FB 14 Informatik, 66123 Saarbrücken, Germany.

While the algorithm in [1] has a very good performance guarantee it has a running time of $O(n^6)$. In this paper we reduce the running time to $O(n^5)$ by introducing a wrap-around technique for computing net-costs and net-weight. Besides this we introduce a host of other ideas which although do not improve the worst-case running time beyond $O(n^5)$ reduce it significantly in practice so that the empirical running times are more like $O(n^4)$ with the constant in the O-notation only about 1/80.

We also implement the algorithm and compare the cost of the solution returned to the lower bound on the optimum cost which is also provided by the algorithm. The implementation is tested on three different test-sets. The first two consist of grid graphs with sizes 10×10 to 30×30 and random edge costs chosen from the set $\{1, 2\}$ in the first and $\{1 \dots n\}$ in the second test set, where n is the number of vertices on one side of the grid. The third set is obtained from EUC_2D instances of TSPLIB [9] by computing the delaunay diagram of the given points.

In 374 out of 435 tested instances we found the optimum separator and we were never more than 10.6% away from a lower bound on the optimum separator. For problems with at least 130 vertices, the maximum error was only 2.60%. Such results have been obtained in the past for other approximation algorithms. For instance, Goemans and Williamson [10] report that the solution found by their 2-approximation algorithm for matching with triangle inequality was never more than 4% off from the optimum for a large set of Euclidean instances. Such results demonstrate that well-designed approximation algorithms perform really well in practice and their worst case bounds, which at 2 or $\log n$ are not very appealing to the practitioner, are achieved only for pathological instances which almost never arise in practice. We should point out that in our implementation of the algorithm in [1] we do not make any heuristic modifications to improve the quality of the solution; all our modifications are limited to improving the running time of the algorithm.

The paper is organized as follows: In section 2 we give a brief overview of the algorithm in [1] introducing the necessary definitions and the various components of the algorithm. In section 3, 4 and 5 we discuss our implementation of these basic components viz. computation of transfer function, dot-cycle and box-cycle respectively. In section 6 we report on the test-sets used and the results obtained and we conclude in section 7 with a discussion of these results and possible directions for future work.

2 An Overview of the algorithm

An optimum separator in a planar graph is a set of edge disjoint cycles in the dual graph.

Given a directed simple cycle, C, the side of C that lies on the right in a traversal of this cycle is the side *enclosed* by C. Let $\mathrm{cost}(C)$ be the total cost of the edges in the cycle C. Define the weight of the cycle C, $\mathrm{wt}(C)$ to be the total weight of faces enclosed by C. For $\mathrm{wt}(C) \leq n/2$, define the *sparsity* of C, $\mathrm{sparsity}(C)$, as the ratio $\mathrm{cost}(C)/\mathrm{wt}(C)$. The algorithm in [1], called DOT-BOX ALGORITHM, finds a set of directed simple cycles of small total cost and having a total weight between $n/3$ and $2n/3$.

The DOT-BOX ALGORITHM proceeds in iterations. In the i^{th} iteration it has a partial solution T_{i-1} ($T_0 = \phi$) which is a set of cycles enclosing less than $n/3$ faces. This it augments to another partial solution T_i by finding a *dot-cycle* D_i, and to a complete solution (a separator), by finding a *box-cycle* B_i. Thus in each iteration the algorithm finds a separator; the solution returned is the one with minimum cost.

For a directed simple cycle C we denote by $T_{i-1} \cup C$, the set of faces enclosed by C and the cycles in T_{i-1}. The box-cycle B_i is a simple cycle of minimum cost satisfying $n/3 \leq \mathrm{wt}(T_{i-1} \cup B_i) \leq 2n/3$.

Let G_i be the graph obtained by merging the faces enclosed by each cycle T_i into a single face. The trapped cost of a directed simple cycle C in G_i is the total cost of the cycles of T_i enclosed by C. The *net-cost* of C is then defined as

$$\text{net-cost}_i(C) = \text{cost}(C) - \text{trapped-cost}_i(C)$$

and measures the incremental cost incurred in picking C. The net-weight of C is the additional number of faces picked and the net-sparsity of C is the ratio of its net-cost to its net-weight. The dot-cycle D_i is a simple cycle of minimum net-sparsity satisfying $\mathrm{wt}(T_{i-1} \cup D_i) < n/3$.

Let k be the first iteration at which some cycle of the optimum solution is available as a box-cycle to the Dot-Box Algorithm. In [1] it is shown that if $i < k$ then $\text{cost}(T_{i-1}) \leq \text{OPT}$. For $i = k$, $\text{cost}(B_i) \leq \text{OPT}$. We compare these lower bounds on the optimum value with the solutions returned by the algorithm.

3 Computing the Transfer Function

Given an assignment of weights to the faces of a planar graph there is a way of defining values on the edges so that for every simple cycle not enclosing the infinite face the sum of the values of the edges in the cycle equals the total weight of the faces enclosed by the cycle. For cycles enclosing the infinite face this sum is equal to the total weight of the faces enclosed by the cycle minus the total weight of all faces.

We implement a linear-time procedure for computing the transfer function. We first find a spanning tree in the graph and set the transfer function values for all edges in this tree to 0. For every face in the graph we maintain the number of edges that have not been assigned any value so far; initially this is just the number of edges that are not part of the spanning tree. Next we go through the faces till we find one that has only one edge unassigned. We compute the transfer function value for this edge and update the number of unassigned edges on the other face bounded by this edge. If this number goes down to one then we consider this face next, else we go on to the next face. The infinite face is never considered. It is easy to show that we can always find a face (besides the infinite face) that has only one unassigned edge. The bound on the running time is obtained by noting that each face is considered at most twice.

4 Computing the Dot-cycle

Let t_i be the transfer function obtained by assigning faces of G_i that are cycles in T_{i-1} a weight equal to the cost of the cycle and the remaining faces a weight 0. Let C be a simple cycle not enclosing the infinite face. The net-cost of C is then given by $\text{cost}(C) - t_i(C)$. Similarly let w_i be the transfer function obtained by assigning faces of G_i a weight 0 and other faces a weight 1;

$w_i(C)$ now gives the net-weight of cycle C. By finding the cycle with minimum net-cost for each choice of net-weight in the range 0 to $n/3 - \mathrm{wt}(T_{i-1})$ we can find the minimum net-sparsity cycle satisfying the weight restriction. To consider cycles containing the infinite face we also look for cycles with minimum $\mathrm{cost}(C) - t_i(C)$ and $w_i(C)$ in the range $-n + \mathrm{wt}(T_{i-1})$ to $-2n/3$. In [1] it is shown that if the cycle obtained by the above minimization is not simple then the box-cycle found in this iteration has cost at most $\mathrm{cost}(T_{i-1})$ so that the separator found in this iteration is within twice the optimum.

Our extensions. In computing the net-weight of a cycle we sum the transfer function values on the edges of the cycle modulo $(n - \mathrm{wt}(T_{i-1}))$; this limits the net-weight to the range $[0\ldots(\text{n-wt}(T_{i-1})\text{-}1)]$. Further, every time we wrap around the right (left) of this weight range we add (subtract) $\mathrm{cost}(T_{i-1})$ to the net-cost of the cycle. We refer to this way of computing net-cost as computing *modulo* $\mathrm{cost}(T_{i-1})$. A nice aspect of this modification is that now we only need to look for minimum net-cost cycles for each value of net-weight in the range 0 to $n/3 - \mathrm{wt}(T_{i-1})$. Furthermore, the earlier argument that the "cycle minimizing the net-sparsity is simple or the separator found is within twice the optimum" continues to hold. As we point out later this way of computing net-cost and net-weight reduces the running-time for the dot-computation by a factor n.

Our procedure for computing the dot-cycle is similar to the Bellman-Ford algorithm for shortest paths. At any stage of our procedure we maintain information about the shortest paths found so far, for each weight value, from a specified starting node s to all the other nodes in the graph. This is done by having for each node v an array whose i^{th} entry is the minimum net-cost s, v path of net-weight i; the entries in the array for node s would then give us the minimum net-sparsity cycle containing s.

To update these arrays we need to keep track of those "pieces of information" which could possibly give us shorter s, v paths. This is done using a FIFO queue of "active information" each item of which is a 3-tuple of the form (node, net-cost, net-weight); an entry u, c, w in this queue would mean that node u has a path from s of net-weight w and net-cost c and that this information has not yet been passed on to the node adjacent to u. The entries in the various arrays and the queue are updated as follows. Let (u, c, w) be the top of the queue and let node v be adjacent to u (the following step is repeated for all nodes adjacent to u). The path corresponding to the entry (u, c, w) when extended to node v would have a net-weight $(w + w_i(e)) \bmod (n - \mathrm{wt}(T_{i-1}))$ and net-cost $(c + \mathrm{cost}(e) - t_i(e)) \bmod \mathrm{cost}(T_{i-1})$, where e is the edge u, v. If this net-weight entry in the array for node v has a higher net-cost then we set it to this new net-cost. Whenever we manage to update the array for a node v we add a triple to the queue (since this is "active information"). The procedure terminates when the queue becomes empty or we have broadcasted $n - 1$ times.

The wrap-around idea limits the sizes of the arrays at each node. This saves a factor n in the running time over that in [1].

5 Computing the box-cycle

Assign a zero weight to all faces enclosed by the cycles in T_{i-1} and unit weight to the other faces. Then the box-cycle is a simple cycle of minimum cost in the dual graph enclosing a total weight of at least $n/3 - \mathrm{wt}(T_{i-1})$ and at most $2n/3 - \mathrm{wt}(T_{i-1})$. Equivalently, the box-cycle is the optimum simple cycle b-balanced separator in the dual graph, for $b = (n/3 - \mathrm{wt}(T_{i-1}))/(n - \mathrm{wt}(T_{i-1}))$.

In [8] Rao shows that there is an optimum b-balanced simple cycle separator ($b \leq 1/3$), C, such that shortest paths between all pairs of vertices on C lie completely on one side of C. Assume that s is a vertex on C and f a face adjacent to v and on the side of C that does not contain the shortest paths. Compute a shortest path tree, T, rooted at s. Order the edges of T incident at any node in a clockwise manner starting from the parent edge; for the root node edges are ordered clockwise starting from the face f. We now perform a depth-first-search on T that is consistent with the above ordering. Every time a node is visited it is assigned a label which is just the time of the visit; thus a node has as many labels as its degree in T.

Traverse C starting from s and in a direction such that the face f is on the left. There is a way of picking one label for each vertex (from amongst the labels assigned to it) such that the sequence of labels obtained by the above traversal is monotonically increasing. Conversely, any path, starting and ending at s, for which we can obtain a monotonically increasing sequence of labels is a *connected circuit* — a simple cycle with "pinched portion" [8].

Our implementation. Let w_i be the transfer function corresponding to the above assignment of weights to the faces. We find the minimum cost path starting and ending at v for which there is a monotone increasing sequence of labels and such that w_i sums to at least $n/3 - \mathrm{wt}(T_{i-1})$ and at most $2n/3 - \mathrm{wt}(T_{i-1})$. From the above discussion it follows that this path corresponds to a connected circuit of cost at most that of the box-cycle. Since this connected circuit encloses the right weight we are done.

To compute this path we proceed as in the case of the dot-cycle with one modification. We now maintain a four-tuple of active information with the fourth entry now being the highest label encountered by the path so far. Thus if the entry at the top of the queue is (u, c, w, l) then we update the entries in the array of node v (v adjacent to u) only if v has a higher label than l. After updating the array we add the information to the queue, the label associated with this 4-tuple is the smallest label of v larger than l.

We compute the transfer function w_i with respect to the shortest path tree T and with f as the infinite face. Since the tree-paths between vertices on C are completely contained on the side of C not containing the infinite face, the transfer function values for all edges encountered in a clockwise traversal of C are positive. Thus we need not consider edges with negative transfer function values in the procedure above and so we can restrict the size of the array at each node to $2n/3 - \mathrm{wt}(T_{i-1})$. Furthermore, since edge costs are positive and we are searching for the cycle with minimum cost we can kill all such active information (*ie.* not add it to the queue) for which the cost of the path exceeds an upper bound on the box-cycle cost. When we find a cycle with weight in the range $n/3 - \mathrm{wt}(T_{i-1}) \ldots 2n/3 - \mathrm{wt}(T_{i-1})$ we can update the upper-bound.

When the weight of the cycle found is in the range $0 \ldots n/3 - \mathrm{wt}(T_{i-1})$ we compute the ratio of its cost and net-weight for use as an upper-bound on the net-sparsity in the dot-cycle computation. Since this ratio was for a cycle in G and not in G_i, there is a question whether it really gives us an upper-bound on the net-sparsity of the dot-cycle. However, the results in [1] show that the cycle in G with minimum net-sparsity is also a cycle in G_i. Furthermore, in computing the above ratio we used cost instead of net-cost and so it is only higher than the net-sparsity of the dot-cycle.

Note that we assumed knowledge of the vertex s and face f. Rao suggests trying for all possible choices of vertices and the faces adjacent to them; this implies that the above procedure would be repeated $2m$ times where m is the number of edges. Significant savings on this count were obtained

n	# instances	Running time	Inf. broadcast
10	60	21sec.	2.8×10^6
15	50	5min. 25sec.	51×10^6
20	40	42min.	0.4×10^9
23	28	1hr. 57min.	1.1×10^9
27	13	6hr. 17min.	3.6×10^9
30	8	13hr.	7.3×10^9

Table 1: Running times for Grids of size $n \times n$ with edge costs from $\{1, 2\}$. The running time and information broadcast are averages. All solutions were optimum.

by using the following procedure for choosing s, f. We find a spanning tree in the dual graph and for each vertex v and edge e incident to v in the tree, we pick one face adjacent to e as the infinite face. It is easy to see that one of these (vertex, face) choices would give us a vertex on the optimum box-cycle and a face which is on the side of the cycle not containing the shortest paths. This reduces the number of times we have to run the above computationally intensive procedure from $2m$ to $2n$.

6 Experimental Results

The algorithm was implemented in C++ using LEDA (Library of Efficient Data Types and Algorithms) data types and algorithms. Some of the most computationally intensive parts of the code were written in C++ (without invoking the LEDA libraries) and this resulted in some savings in the running time. All experiments were performed on a ULTRA SPARC workstation in a multiuser environment.

We ran experiments on three classes of graphs: the first two sets are $n \times n$ grids, the edge costs for the first set are chosen randomly from $\{1, 2\}$ while that for the second set are chosen randomly from the range $[1 \ldots n]$. The third set of instances are from the TSPLIB. An instance of the TSPLIB is a collection of points in the plane. A planar graph is constructed by computing the delaunay diagram of the point-set and the cost of an edge is just the Euclidean distance between its endpoints. Some programming effort was saved by assuming that the input graph is the dual of the planar graph. The results obtained are tabulated in Tables 1, 2 and 3.

7 Discussion of Results

For instances where the edge costs are small (1,2) we always get optimum solutions. The amount of active information generated for this test set also seems to behave more like $O(n^3 \log n)$. When the edge costs are from a larger range — as is the case with grid graphs with edge costs in the range $[1 \ldots n]$ and the TSPLIB instances — the amount of active information generated (and hence the running time) seem to behave more like $O(n^4)$; however the constant in the O-notation is only about 1/80. The quality of the solution for these sets of instances is again surprisingly good with

n	# instances	# optimum	max. error	Running time	Inf. broadcast
10	60	45	4.00%	27sec.	3.7×10^6
15	50	37	1.20%	8min. 20sec.	89×10^6
20	40	32	1.04%	1hr. 28min.	0.9×10^9
23	20	15	1.77%	4hr. 27min.	2.9×10^9
27	13	9	0.93%	15hr.	11×10^9
30	8	8	-	38hr.	26×10^9

Table 2: Running times and quality of solution for Grids of size $n \times n$ with edge costs from $[1 \ldots n]$. The running time and information broadcast are averages.

	# Instances	optimum	max. Error
≤ 70 nodes	2	1	10.6%
70–130 nodes	14	6	4.14%
130–200 nodes	13	8	2.60%
≥ 200 nodes	16	14	0.88%
Total	45	29	10.6%

Table 3: For TSPLIB instances the quality of solution obtained improves with increasing problem size.

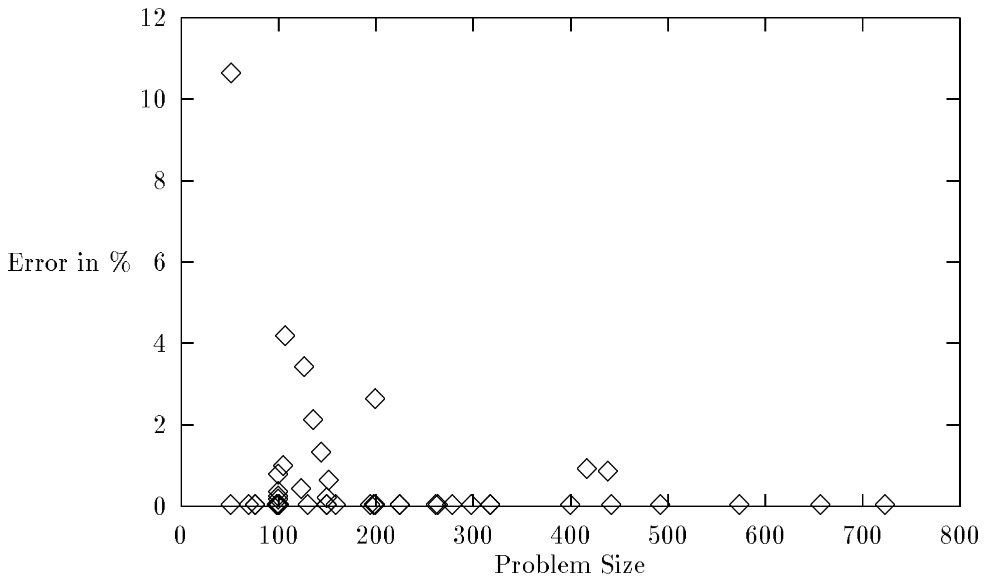

Figure 1: A plot of the quality of the solution obtained for different problem sizes from TSPLIB.

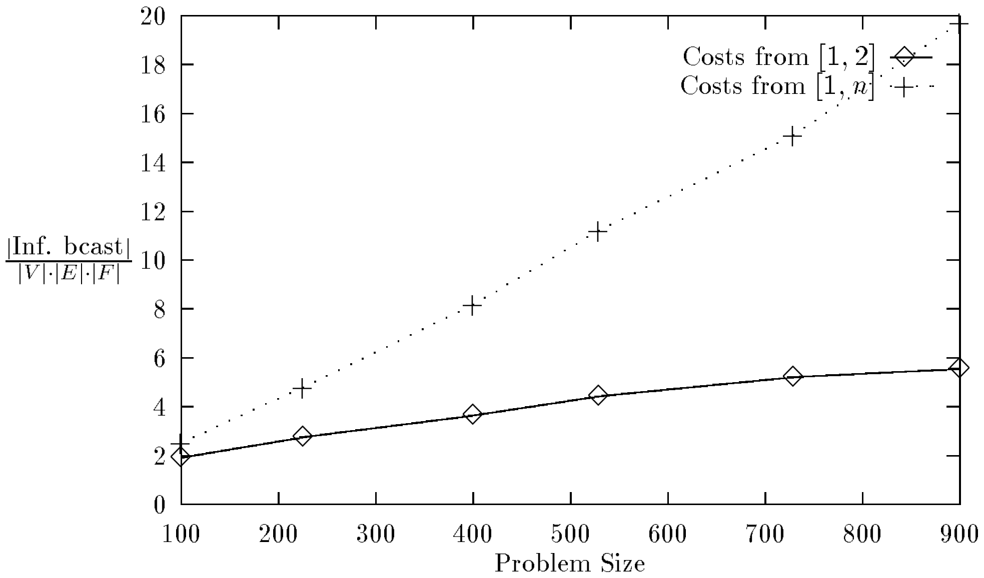

Figure 2: A plot of the average active information generated divided by $|V|\cdot|E|\cdot|F|$ for different grid sizes.

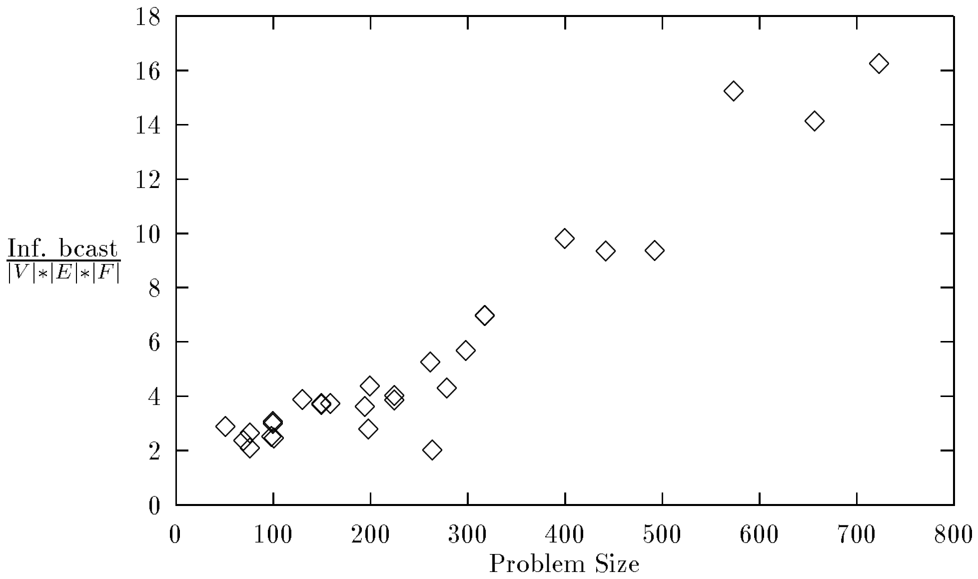

Figure 3: A plot of the average active information generated divided by $|V|\cdot|E|\cdot|F|$ for different TSPLIB instances.

algorithm finding the optimum solution in 75% of the grid and 66% of the TSPLIB instances. One drawback of the algorithm is still the fairly large running time; in spite of all the additional techniques we used to reduce running time, we obtained empirical times of $\Theta(n^4)$. A trade-off between the running time and the quality of the solution produced can be achieved by, for instance, running the dot and box-cycle only from a small set of randomly chosen starting nodes. While this shall reduce the running time we will not be able to provide any guarantees on the quality of the solution obtained. Another possible approach that we have not checked yet is to preprocess the graph in some manner so as to reduce its size.

The algorithm can be parallelized in a rather straightforward manner. The various starting nodes for the dot and box cycle computations can be distributed evenly among the various processors. The processors would have to synchronize after every dot and box cycle computation. But within one of these computations they need to exchange very little information with the other processors; every time a processor finds a dot/box cycle of lesser cost/net-sparsity than the current best it broadcasts this information to the other processors.

References

[1] N. Garg, H. Saran, and V.V. Vazirani. Finding separator cuts in planar graphs within twice the optimal. In *Proceedings, IEEE Symposium on Foundations of Computer Science*, pages 14–23, 1994. Journal version available at `http://www.mpi-sb.mpg.de/~naveen`.

[2] D.S. Johnson, L.A. McGeoch, and E.E. Rothberg. Asymptotic experimental analysis for the held-karp traveling salesman bound. In *Proceedings, ACM-SIAM Symposium on Discrete Algorithms*, 1996.

[3] F.T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with application to approximation algorithms. In *Proceedings, IEEE Symposium on Foundations of Computer Science*, pages 422–431, 1988.

[4] R.J. Lipton and R.E. Tarjan. A separator theorem for planar graphs. *SIAM Journal of Applied Mathematics*, 36(2):177–189, 1979.

[5] R.J. Lipton and R.E. Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9:615–627, 1980.

[6] J.K. Park and C.A. Phillips. Finding minimum-quotient cuts in planar graphs. In *Proceedings, ACM Symposium on Theory of Computing*, pages 766–775, 1993.

[7] S.B. Rao. Finding near optimal separators in planar graphs. In *Proceedings, IEEE Symposium on Foundations of Computer Science*, pages 225–237, 1987.

[8] S.B. Rao. Faster algorithms for finding small edge cuts in planar graphs. In *Proceedings, ACM Symposium on Theory of Computing*, pages 229–240, 1992.

[9] G. Reinelt. TSPLIB 95. `http://www.iwr.uni-heidelberg.de/iwr/cornopt/soft/TSPLIB95/TSPLIB.html`, 1995.

[10] D.P. Williamson and M.X. Goemans. Computational experience with an approximation algorithm on large-scale euclidean matching instances. In *Proceedings, ACM-SIAM Symposium on Discrete Algorithms*, pages 355–364, 1994.