SCHOOL OF OPERATIONS RESEARCH
AND INDUSTRIAL ENGINEERING
COLLEGE OF ENGINEERING
CORNELL UNIVERSITY
ITHACA, NY 14853-7501

TECHNICAL REPORT NO. 926

September 1990

A FASTER PARAMETRIC
MINIMUM CUT ALGORITHM

by

Dan Gusfield¹ and Éva Tardos²

¹Computer Science Division, University of California, Davis. Research partially supported by grant CCR-8803704 from the National Science Foundation.

²Research partially supported by grant DMS89-20550 from the National Science Foundation.

A Faster Parametric Minimum Cut Algorithm

Dan Gusfield¹
Computer Science Division
University of California, Davis

Éva Tardos²
School of Operations Research and Industrial Engineering
Cornell University, Ithaca

August, 1990

Abstract

Gallo, Grigoriadis and Tarjan [4] recently examined the problem of computing online a sequence of k maximum flows and minimum cuts in a network of n nodes, where certain edge capacities change between each flow. They showed that for an important class of networks, the k flows can be computed in $O(n^3 + kn^2)$ total time, provided that the capacity changes are made “in order”. We show how to reduce the total time to $O(n^3 + kn)$. We further reduce the total time to $O(n^2\sqrt{m})$ if k is at most $O(n)$. We also apply the ideas from Gusfield and Martel [11] to show that the faster bound holds even when the capacity changes are not “in order”. We illustrate the utility of these results by applying them to the *rectilinear layout problem*.

1 Introduction and Background

Throughout the paper, we assume familiarity with the maximum flow and minimum cut problem in general (see [16], [12] for an introduction, and [6] for a recent survey), and the Goldberg-Tarjan (GT) flow method [5] in particular. However, we briefly review some terminology and facts about the GT method.

A *network* is a directed graph $G=(V,E)$ with two distinguished vertices, a *source* s and a *sink* t , and a positive capacity $c(v,w)$ for every directed edge (v,w) . We let n denote the number of vertices and m denote the number of edges. A *flow* f in G is a function from the edges to the nonnegative reals such that the following two properties hold:

Capacity Constraint. For any directed edge (v,w) , $0 \leq f(v,w) \leq c(v,w)$.

Flow Conservation. For any vertex $v \neq s, t$, $f_{in}(v) = f_{out}(v)$.

A *preflow* is a flow except that we relax the conservation condition to allow $f_{in}(v) \geq f_{out}(v)$. If $f_{in}(v) > f_{out}(v)$, then v is said to have *excess*. The value of the excess is the difference $f_{in}(v) - f_{out}(v)$. A node other than s or t which has excess is called *active*.

Given a flow or a preflow f , the *residual graph* for f is obtained as follows: if $f(u,v) > 0$ then directed edge (v,u) is in the residual graph with capacity $f(u,v)$; if $f(u,v) < c(u,v)$, the directed edge (u,v) is in the residual graph with capacity $c(u,v) - f(u,v)$.

¹Research partially supported by grant CCR-8803704 from the National Science Foundation.

²Research partially supported by grant DMS89-20550 from the National Science Foundation.

The generic GT algorithm is a preflow algorithm, meaning that it only needs to maintain a feasible preflow until the end of the algorithm, when it becomes a (maximum) flow. During the algorithm each vertex v has an associated label $d(v)$ which is always between 0 and $2n - 1$. These d labels are called *valid* for the current preflow f if $d(s) = n$, $d(t) = 0$, and $d(u) \leq d(v) + 1$ for any edge (u, v) ($u \neq s$) in the current residual graph for f . Throughout the GT algorithm the d labels never decrease and are always valid. A directed edge (u, v) is called *admissible* if and only if it is in the current residual graph and $d(u) = d(v) + 1$.

The basic step of the generic GT algorithm is called a *node examination*. In that step an active node v is picked and all the excess is pushed along admissible edges from v to neighbors of v in the current residual graph. Any push along an edge pushes as much as possible on that edge. A push is called *saturating* if the push fills the edge to capacity, and *non-saturating* otherwise. The algorithm terminates when there are no active nodes.

For each node v the GT algorithm keeps a list $I(v)$ in arbitrary but fixed order, containing every node w such that either edge (v, w) or (w, v) is in G . Hence $I(v)$ represents all the edges (v, w) which could possibly be admissible. At any point during the algorithm there is a pointer $p(v)$ into $I(v)$. At the start of the algorithm each $p(v)$ points to the top of $I(v)$. When v is examined the algorithm finds admissible edges by searching through $I(v)$ in order, starting at $p(v)$, advancing $p(v)$ each time a new element of $I(v)$ is considered. If the bottom element of $I(v)$ is passed, then $d(v)$ is changed to $\text{Min}[d(w) + 1 : (v, w) \text{ is an edge in the current residual graph}]$, $p(v)$ is set to the top of $I(v)$, and the examination of v continues. The change of $d(v)$ always strictly increases $d(v)$ and creates at least one admissible edge. Hence, this cycling scan through $I(v)$ always results in all excess being pushed out of v during an examination of v . Note that at most one non-saturating push (the last one, if any) from v occurs during a single examination of v .

1.1 Specializations of GT

Goldberg and Tarjan discuss two specialized versions of the generic algorithm. In the FIFO version, active nodes are placed on a queue as they become active, and are picked for examination off the top of the queue. In the Max- d version, the active node picked for examination is always the one with the largest d label. The new result in this paper is based on the analysis for the Max- d version, and so we review that analysis below.

In the generic GT algorithm, the time analysis is divided between the time for all the non-saturating pushes, and the time for all-other-work. The time for all-other-work is just $O(nm)$ as follows. Each operation other than a non-saturating push or a relabeling causes $p(v)$ to advance for some v , and $d(v)$ advances every time $p(v)$ passes the bottom of $I(v)$. Since $d(v) < 2n$, and $|I(v)|$ is the sum of the in and out degrees in G of v , the total advancement of all n p pointers is bounded by $2n \sum_v |I(v)| = O(nm)$. Relabeling also only costs $O(nm)$ since a node gets relabeled at most $2n$ times, and to find the new $d(v)$, a scan of the nodes in $I(v)$ is sufficient. So the time for all-other-work is $O(nm)$.

The analysis for all-other-work in the Max- d version is the same as in the generic algorithm, except that there is additional work involved in finding an active node of maximum d label. The algorithm keeps a set A of $2n - 1$ linked lists of active nodes, each indexed by

a number from 1 to $2n - 1$. List j keeps all the active nodes whose d label is j . A is used to locate an active node of maximum d label. If there is more than one, then the node picked for examination is the first one on the list. Each push from a node v must be to a node with d label equal to $d(v) - 1$, so finding the next active node of maximum d value takes constant time. Further, updating A after a node relabeling involves only constant work, so A can be maintained and used in $O(n^2)$ time plus $O(1)$ time per push.

The time required for a non-saturating push is $O(1)$. The number of non-saturating pushes is $O(n^3)$ for both the FIFO and the Max- d algorithms. The proof (from [1]) for the Max- d case is particularly simple, and will be needed later.

Theorem 1.1 *The Max- d algorithm performs only $O(n^3)$ non-saturating pushes.*

Proof In the Max- d algorithm, at most n consecutive node examinations can occur without at least one node label increasing. To see this, note that excess is always pushed from the highest active node to a lower labeled node (since it is pushed along admissible edges). So, since each node pushes out all its excess when examined, if no node labels change during n node examinations, then all excess in the network will either be pushed forward to t or backward to s . At that point the algorithm terminates since there will be no active nodes. Each node examination can do at most one non-saturating push, so there can be at most n non-saturating pushes between node label increases. Each node label is bounded by $2n$, and node labels never decrease, so there can be at most $O(n^3)$ non-saturating pushes. $\square$

Cheriyān and Maheshwari [1] proved that the Max- d algorithm has only $O(n^2\sqrt{m})$ non-saturating pushes. We give a sketch of this improved analysis. The key point in the proof of Theorem 1.1 is that there are at most n consecutive node examinations between two relabelings. The improved analysis is based on a somewhat finer subdivision of the algorithm. A *phase* of the algorithm is a sequence of operations while the maximum label of the active nodes is unchanged. The total increase in the maximum label of the active nodes is $O(n^2)$, the same as the total increase in node labels. This implies that the number of phases is also at most $O(n^2)$.

Consider an active node v . A non-saturating push from v to some other node w pushes the excess at node v to w . We call such a flow excess traveling through a sequence of non-saturating pushes an *atom*. The path of non-saturating pushes that an atom has been pushed through before reaching its current location is called the *trace* of the atom. Each active node has exactly one atom at the node. A new atom is created when two atoms merge due to a push to a node with excess. Two new atoms are created when a saturating push leaves the examined node with excess. The number of atoms simultaneously alive is increased by saturating pushes and decreased by mergings. This implies that the number of atoms created throughout the algorithm is at most twice as much as the number of saturating pushes, which is $O(mn)$. The key observation of Cheriyān and Maheshwari is that if two atoms are at two highest labeled active nodes simultaneously then they have node-disjoint traces. This is a consequence of the choice of the active node with maximum label for examination and the use of pointers in the choice of admissible edges. Consider

an atom α at an active node v . If another atom β is at a node on the trace of α , then β is currently at a node with higher label than the label of v , and α will be merged into a new atom before node v has again the highest label among active nodes.

Theorem 1.2 *The Max-d algorithm performs only $O(n^2\sqrt{m})$ non-saturating pushes.*

Proof We analyze the number of non-saturating pushes in two parts. We consider the trace of an atom short if it has length at most $\frac{n}{\sqrt{m}}$. There are at most $O(mn)$ atoms. Each can have short trace during at most $\frac{n}{\sqrt{m}}$ non-saturating pushes. Hence the number of non-saturating pushes on atoms with short traces is at most $O(n^2\sqrt{m})$. Now consider the non-saturating pushes on atoms with long traces. All atoms pushed during a phase are at an active node with maximum label at the beginning of the phase. Therefore, all these atoms have node-disjoint traces, and hence there could be at most $\sqrt{m}$ of them in a phase. The number of phases is at most $O(n^2)$. This proves the $O(n^2\sqrt{m})$ bound on the number of non-saturating pushes on atoms with long traces. $\square$

Remark The above argument relies on the use of pointers in the selection of admissible edges. Tunçel [18] proved that Theorem 1.2 is true no matter what method is used to select admissible edges.

2 Monotone parametric flow and the GGT algorithm

In a *monotone parametric flow network* the capacities on arcs out of s and into t are functions of a real-valued parameter λ . All other arcs have fixed capacities. The capacity of any arc (s, v) is given by the capacity function $c_v(\lambda)$, where c_v is required to be a *nonnegative nondecreasing* function of λ . Similarly, the capacity of any arc (v, t) is given by the function $\hat{c}_v(\lambda)$, where $\hat{c}_v$ is a *nonnegative nonincreasing* function of λ . We do not assume that the capacity functions are explicitly known to the algorithm, but rather, the values of the functions will be obtained whenever a value for λ is specified. We let $G(\lambda)$ denote the network which results from fixing a particular value of λ and setting the parameterized edge capacities accordingly.

The fundamental result of Gallo, Grigoriadis and Tarjan [4] is the following:

Theorem 2.1 *In a monotone parametric flow network G , if the values $\lambda_1 < \lambda_2 < \dots < \lambda_k$ (or $\lambda_1 > \lambda_2 > \dots > \lambda_k$) are specified in this order, then a maximum flow and a minimum cut in each of the networks $G(\lambda_1), \dots, G(\lambda_k)$ can be computed on line in $O(n^3 + kn^2)$ total time, or in time $O(nm \log(\frac{n^2}{m}) + km \log(\frac{n^2}{m}))$ if dynamic trees are used.*

Thus for $k = O(n)$, all the flows can be done in the same worst case time as the fastest known algorithm for a single flow. This result has many applications and leads to the fastest solutions of many combinatorial problems. Numerous applications are discussed in [4] [7] [9], [11] and [8]. It is important to note that in all of these applications it is the minimum cut that is needed; the maximum flow is computed in order to find the minimum

cut. Martel [13] showed that the $O(n^3 + kn^2)$ bound can also be obtained by using either the Karzanoff or the wave versions of the Dinic maximum flow algorithm in place of the GT algorithm.

The GGT algorithm [4] (establishing Theorem 2.1) is based on the idea that the maximum flow in $G(\lambda_i)$ can be used to obtain a good initial preflow in $G(\lambda_{i+1})$ and that, if care is taken, the last d labels in $G(\lambda_i)$ are valid for this initial preflow. In detail, for each i , the initial preflow in $G(\lambda_{i+1})$ is obtained from the maximum flow in $G(\lambda_i)$ by increasing the flow in every edge (s, v) to $c_v(\lambda_{i+1})$ if $d(v) < n - 1$, by reducing the flow in every edge (v, t) to $c_v(\lambda_{i+1})$, and by leaving all other edge flows as they are in the maximum flow in $G(\lambda_i)$. Each $d(v)$ is unchanged from its last value in $G(\lambda_i)$.

It is easy to verify, by the fact that $\lambda_i < \lambda_{i+1}$ and the monotonicity of the capacity functions, that this initial flow assignment is a preflow in $G(\lambda_{i+1})$; it is also easy to verify that the d labels are valid for this preflow. After the initial preflow is set, the maximum flow in $G(\lambda_{i+1})$ is found by resuming the GT flow algorithm and running it to completion.

2.1 Main Result: Parametric flow with the Max- d version

In [4], the FIFO version of the GT algorithm was used in the above parametric method, resulting in Theorem 2.1. The new observation in this paper comes from using precisely the same parametric method as above, but using the Max- d version of the GT algorithm instead of the FIFO version.

Theorem 2.2 *In a monotone parametric flow network G , if the values $\lambda_1 < \lambda_2 < \dots < \lambda_k$ (or $\lambda_1 > \lambda_2 > \dots > \lambda_k$) are specified in this order, then a maximum flow and a minimum cut in each of the networks $G(\lambda_1), \dots, G(\lambda_k)$ can be computed on line in $O(n^3 + kn)$ total time.*

Proof As mentioned above, we use the Max- d version of the GT algorithm. However, we note two additional implementation details that were not critical in the $O(n^3 + kn^2)$ result, but are needed for the faster bound. First, when beginning the flow computation in $G(\lambda_{i+1})$, the initial position of each $p(v)$ is its ending position in the flow computation in $G(\lambda_i)$. Second, s must be at the bottom of any $I(v)$ list that it is in, and similarly for t . The first modification is needed for the time analysis below, and the second modification is needed for the correctness of the method. The reason is the following. For the correctness of the GT algorithm, whenever $d(v)$ changes there must be no admissible edges out of v . It was shown in [5] that for a single flow computation if w is a node above $p(v)$ in $I(v)$ then (v, w) is not admissible; so when $p(v)$ passes the bottom of $I(v)$ there are no admissible edges out of v . To ensure that after a capacity change any edge (v, w) is still inadmissible if w is above $p(v)$, we always put s and t at the bottom of any list they are in, since an inadmissible edge not incident with s or t is clearly still inadmissible after a capacity change.

For the time analysis for the flow, we note how the $O(n^3)$ bound for a single flow computation is affected when k flows are computed by the method described above. Again, the analysis is divided into time for non-saturating pushes, and all-other-work. In the above

parametric method, the d labels never decrease, and each is bounded by $2n$ no matter how large k is. Further, $p(v)$ is not moved when the edge capacities change. So the analysis for all-other-work *inside* the k flow computations is unchanged, and the time is again $O(nm)$. The time for making the capacity changes is certainly bounded by $O(kn)$ as is the time to set up A after each capacity change. Note that A is empty at the end of each flow computation. So all-other-work inside and between flow computations is bounded by $O(mn + kn)$.

To analyze the number of non-saturating pushes, note that *inside* any of the k flows there cannot be more than n non-saturating pushes before a d label increases, for precisely the same reason as in a single flow. However, there may be $n - 1$ non-saturating pushes, then a capacity change, and then another $n - 1$ non-saturating pushes, all without a label change. So each time the capacities change the bound on the total number of allowed non-saturating pushes increases by n . Hence the total number of non-saturating pushes over k flows is bounded by $O(n^3 + kn)$. Hence the time to compute the k flows is $O(n^3 + kn)$.

We now discuss how to find the k minimum cuts. Define S_i to be the set of nodes reachable from s in the residual graph obtained from the maximum flow in $G(\lambda_i)$; let $T_i = N \setminus S_i$. It is known [3] that S_i, T_i is the unique "leftmost" minimum s, t cut. We will find S_i, T_i in each $G(\lambda_i)$, but we cannot search naively from s since that would take $\Omega(km)$ total time. We note the following two facts. First, $S_i \subseteq S_{i+1}$ for every i [15] [4]; second, if w is in $S_{i+1} \setminus S_i$ then w must be reachable in the $G(\lambda_{i+1})$ residual graph from a node $v \in S_{i+1} \setminus S_i$ such that (s, v) is an edge in G and $c_v(\lambda_{i+1}) > c_v(\lambda_i)$ [4]. Hence the search for S_{i+1} examines some edges not previously examined in any search for S_j $j < i$, plus at most n edges previously examined. So, the time to find all the k cuts is $O(m + kn)$. $\square$

Remark If the Max- d algorithm is implemented using dynamic trees the resulting algorithm runs in $O(nm \log(\frac{n^2}{m}) + kn \log(\frac{n^2}{m}))$ time. This improves the implementation in Theorem 2.1 using dynamic trees.

Theorem 2.2 and the Remark is important when $n = o(k)$, which happens in many applications; one will be discussed in Section 4.

Next we improve the analysis of the Max- d version of the parametric flow algorithm for relatively small k by extending the Cheriyan-Maheshwari analysis. We define atoms and phases as before. The number of atoms throughout the algorithm is $O(mn + nk)$ because $2n$ new atoms can be created when a new flow computation is started. The number of phases is $O(n^2 + nk)$ since the maximum label can increase by $O(n)$ every time a new flow computation is started. Atoms at active nodes with maximum label have node disjoint traces, for precisely the same reason as in a single flow.

Theorem 2.3 *The parametric Max- d algorithm performs at most $O(n^2\sqrt{m} + n^{1.5}\sqrt{mk} + n^{1.5}k)$ non-saturating pushes.*

Proof Let ℓ be a parameter that will be selected later. We call the trace of an atom short if it has length at most ℓ . Clearly the number of non-saturating pushes on atoms with short trace is $O((nm + nk)\ell)$. As before the atoms pushed on during a phase have node-disjoint traces. Therefore, the number of non-saturating pushes on atoms with long traces during a phase is at most $O(\frac{n}{\ell})$, and the total number of such pushes is $O((n + k)\frac{n^2}{\ell})$. The

best choice for ℓ is $\sqrt{n(n+k)/(m+k)}$, which implies the bound claimed in the Theorem. $\square$

Theorem 2.3 gives an improved bound for a data structure free algorithm if, for example, $n = O(k)$. Applications with $k = O(n)$ are discussed in [4] [7] [9], [11] and [8].

3 Parametric flow for parameters given out of order

We briefly discuss how to obtain the improved time bounds presented in the previous section even when the parameters are presented online in *any order*. We discuss only the problem of finding the minimum cuts in the k problems, but the flows can also be found in the same time bound. In [4], a method was presented for completely describing the capacity of the minimum cut as a function of λ when, in addition to the monotonicity requirements, $c_v(\lambda)$ and $\hat{c}_v(\lambda)$ are known *affine* functions of λ . The method generates a sequence of λ values which are not necessarily in order. By extracting the essential ideas from the particular algorithm given for affine functions, we can generalize the method to allow any monotonic functions. Such an approach was taken in [11] where we obtained the same bounds as in Theorem 2.1 (sparse as well as dense bounds), but for λ given in any order. We will now show to achieve the faster time bounds.

We assume that initially we have two values λ_{min} and λ_{max} , with the first smaller than any other λ value, and the second larger than any other λ to be given. Define G^R as the graph obtained from G by reversing all the edges in G ; flow in G^R is assumed to be from t to s .

We start by finding a maximum flow f_{min} in $G(\lambda_{min})$ and a maximum flow f_{max} in $G^R(\lambda_{max})$ using the Max- d version of the GT algorithm in both cases. In general if $\mu_1 < \mu_2 < \dots < \mu_j$ is a sorted list of the values λ_1 to λ_j , then after processing λ_j we will have intervals $[\lambda_{min}, \mu_1], [\mu_1, \mu_2], \dots, [\mu_j, \lambda_{max}]$. We will also associate with each interval $[\mu_i, \mu_{i+1}]$ a graph G_i and two flows: a max flow in $G_i(\mu_i)$ and a max flow in $G_i^R(\mu_{i+1})$. Each of the graphs G_i is a contracted subgraph of G which has the property that for any λ in $[\mu_i, \mu_{i+1}]$ a minimum cut in $G_i(\lambda)$ is a minimum cut in $G(\lambda)$. To process the next λ_{j+1} value, we find the interval $[\mu_i, \mu_{i+1}]$ into which λ_{j+1} falls, and then find a minimum cut in $G(\lambda_{j+1})$.

The minimum cut in $G(\lambda_{j+1})$ can be found either by continuing the flow computation in $G_i(\mu_i)$ (increasing λ) or by continuing the flow computation in $G_i^R(\mu_{i+1})$ (decreasing λ). The actual method for obtaining the minimum cut in $G(\lambda_{j+1})$ is detailed in **procedure slice** in [4] (see also [11]); we will not repeat the details here³. However, procedure slice not only computes the minimum cut in $G(\lambda_{j+1})$, but it also creates two new graphs G' and G'' , and computes maximum flows in $G'(\lambda_{j+1})$ and $G''(\lambda_{j+1})$. Hence, slice establishes the needed general conditions for the new intervals $[\mu_i, \lambda_{j+1}]$ and $[\lambda_{j+1}, \mu_{i+1}]$.

Graphs G' and G'' are contracted subgraphs of G_i , and are node disjoint except for s and t . Hence one of them has at most half the number of nodes as its parent graph G_i . We call this graph an S graph and the other graph an L graph. Procedure slice always

³We should mention that procedure slice finds both the leftmost and the rightmost cuts. However, the procedure and the worst case time analysis remain correct if only the leftmost cut is used.

guarantees that any future flow on the L graph can be found by a continuation of the last flow on either G_i or G_i^R (which one is determined by procedure slice).

Although we have not described procedure slice in detail, we have stated enough facts (detailed in [4] or [11]) to establish the following.

Theorem 3.1 *In a monotone parametric flow network G , if the values $\lambda_1, \lambda_2, \dots, \lambda_k$ are specified in any order, then a minimum cut in each of the networks $G(\lambda_1), \dots, G(\lambda_k)$ can be computed on line in $O(n^3 + kn)$ total time.*

Proof We represent all the graphs used or created during the algorithm by a binary tree T of at most $2k$ nodes. The two children of the root correspond to G and G^R respectively, and each other non-root node corresponds to a graph created by one of the k calls of procedure slice. The two children of any (non-root) node v correspond to the two graphs spawned from the graph at v , by procedure slice. We consider graphs G and G^R to both be S graphs. For each S node v (a node corresponding to an S graph) there is exactly one maximal path of L nodes descending from v . We denote the length of this path by k_v . We use p_v to denote the number of nodes in the graph corresponding to node v in T .

In the time analysis, we pay $O(p^3)$ for any S graph with p nodes. That is, the desired time bound will be achieved even if the flow in every S graph is computed from scratch. We claim that the total computation time for all the S graphs is just $O(n^3)$. To see this, let the *level* of an S node v be the number of S nodes on the path from v to the root of T . Then note that the S graphs at any level i have at most $n/2^i$ nodes in total, since each S graph has at most half as many nodes at the closest S graph above it, and any two S graphs at level i share only nodes s and t . So, the total time for the maximum flows on all the S graphs at level i is $O(\sum (p_v)^3: v \text{ is an } S \text{ node at level } i) = O((1/2^i n)^3)$. Hence the total time for all the flows on S graphs is $O(n^3)$.

To analyze the time for flows on L graphs, consider a maximal path of k_v L nodes descending from an S node v . Since the flow on any L graph is a continuation of the flow on its parent graph, the cost of the flows on such a path of L graphs is $O(p_v^3 + k_v p_v)$ by Theorem 2.2. These maximal paths contain all the L graphs exactly once, and contain no S graph more than once, so the total cost for the flows on all the L graphs is $O(n^3 + kn)$.

We also have to include the time to find the correct interval when a new value of λ is presented. $O(k \log k)$ time suffices, but if we are only interested in computing maximum cuts, then the time to find all the correct intervals can be limited to $O(n \log n)$. For details see [10]. $\square$

Similarly one can prove the analogs of the other two bounds presented in the previous section.

Theorem 3.2 *In a monotone parametric flow network G , if the values $\lambda_1, \lambda_2, \dots, \lambda_k$ are specified in any order, then a minimum cut in each of the networks $G(\lambda_1), \dots, G(\lambda_k)$ can be computed on line in $O(n^2 \sqrt{m} + n^{1.5} \sqrt{mk} + n^{1.5} k)$ total time, and in $O((nm + kn) \log(\frac{n^2}{m}))$ total time using dynamic trees.*

4 An Application: The Rectilinear Layout Problem

Given m fixed points on a line, n new points must be added to that line. For each pair (i, j) of the $m + n$ points there is a given weight $w(i, j)$. The objective is to place the new points on the line to minimize $\sum_{(i,j)} [w(i, j) \times d(i, j)]$, where $d(i, j)$ is the distance between points i and j on the line. A t -dimensional version of the problem reduces to t separate one dimensional problems.

The layout problem was solved independently by Picard and Ratliff [14], Cheung [2], and Trubin [17]. Their solutions require computing $m - 1$ minimum cuts on a dense graph of n nodes, yielding an $O(mn^3)$ time bound. In general $m > n$ (in fact $m \gg n$ is likely) and so we would like to reduce the role of m . In [11] we note that the $m - 1$ minimum cut problems satisfy the requirements of the GGT model, implying immediately a time bound of $O(n^3 + mn^2)$. With Theorem 2.2 this is reduced to $O(n^3 + mn)$. In [11] we also give a method (needing the ability to process λ values out of order) with a running time of $O(n^3 + n^3 \log m)$. With Theorem 3.1, that method now has a running time of $O(n^3 + n^2 \log m)$. In summary we have

Theorem 4.1 *The rectilinear layout problem can be solved with only $O(n^3 + n^2 \log m)$ time used for the network flow computations.*

Of course, the total time for the solution must also include the time to set up the networks. There are simple ways of preprocessing the data which only take time *equal* (not just proportional) to the time to read in the data, so that after the preprocessing each network can be set up in $O(n)$ time. So the real bottleneck is the time for the network flows, and the speedup established in Theorem 4.1 is of importance.

5 Acknowledgement

Thanks to Chip Martel for continuing discussions on this general topic. The particular proof given for Theorem 3.1 was developed jointly with Chip.

References

- [1] J. Cheriyan and S.N. Maheshwari. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing*, 18:1057–1086, 1989.
- [2] T.Y. Cheung. Multifacility location problem with rectilinear distance by the minimum-cut approach. *ACM transactions on Math Software*, 6:549–561, 1980.
- [3] L.R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, New Jersey, 1962.
- [4] G. Gallo, M. Grigoriadis, and R.E. Tarjan. A fast parametric network flow algorithm. *SIAM Journal on Computing*, 18:30–55, 1989.

- [5] A. Goldberg and R.E. Tarjan. A new approach to the maximum flow problem. *JACM*, 4:136–146, 1988.
- [6] A. V. Goldberg, É. Tardos, and R. E. Tarjan. Network Flow Algorithms. Technical Report OR-860, School of Operations Research, Cornell University, 1989. To appear in *Paths, Flows and VLSI-Design* (eds. B. Korte, L. Lovász, and A. Schrijver), Springer Verlag.
- [7] D. Gusfield. Computing the strength of a network in $O(|V|^3|E|)$ time. Technical Report CSE-87-2, Computer Science Division, University of California, Davis, April, 1987.
- [8] D. Gusfield and R. Irving. *The stable marriage problem: structure and algorithms*. The MIT press, Cambridge, MA., 1989.
- [9] D. Gusfield and R.W. Irving. Parametric stable marriage and minimum cuts. *Information Processing Letters*, 30:255–259, 1989.
- [10] D. Gusfield and C. Martel. A fast algorithm for the generalized parametric minimum cut problem and applications. Revised version March, 1990.
- [11] D. Gusfield and C. Martel. A fast algorithm for the generalized parametric minimum cut problem and applications. Technical Report CSE-89-21, University of California, Davis, 1989.
- [12] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.
- [13] C. Martel. A comparison of phase and non-phase network algorithms. *Networks*, 19:691–705, 1989.
- [14] J. Picard and H. Ratliff. A cut approach to the rectilinear distance facility location problem. *Operations Research*, 26:422–433, 1978.
- [15] H.S. Stone. Critical load factors in two-processor distributed systems. *IEEE Transactions on Software Engineering*, 3:254–258, 1978.
- [16] R.E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.
- [17] V.A. Trubin. Effective algorithm for the weber problem with a rectangular metric. *Cybernetics*, 14:874–878, 1978.
- [18] L. Tunçel. On the complexity of preflow-push algorithms for maximum flow problems. Technical Report OR-901, School of Operations Research and Industrial Engineering, Cornell University, 1990.