The Dynamic Vertex Minimum Problem and Its Application to Clustering-type Approximation Algorithms

Harold N. Gabow[1] and Seth Pettie[2,*]

[1] Department of Computer Science, University of Colorado at Boulder, Boulder, CO 80309, USA hal@cs.colorado.edu
[2] Department of Computer Science, University of Texas at Austin, Austin TX 78712, USA seth@cs.utexas.edu

Abstract. The dynamic vertex minimum problem (DVMP) is to maintain the minimum cost edge in a graph that is subject to vertex additions and deletions. DVMP abstracts the clustering operation that is used in the primal-dual approximation scheme of Goemans and Williamson (GW). We present an algorithm for DVMP that immediately leads to the best-known time bounds for the GW approximation algorithm for problems that require a metric space. These bounds include time $O(n^2)$ for the prize-collecting TSP and other direct applications of the GW algorithm (for n the number of vertices) as well as the best-known time bounds for approximating the k-MST and minimum latency problems, where the GW algorithm is used repeatedly as a subroutine. Although the improvement over previous time bounds is by only a sublogarithmic factor, our bound is asymptotically optimal in the dense case, and the data structures used are relatively simple.

1 Introduction

Many approximation algorithms are applications of the primal-dual algorithm of Goemans and Williamson (GW) [12]. (This algorithm is rooted in the approach proposed by Agrawal, Klein and Ravi [2].) This paper determines the asymptotic time complexity of the GW clustering operation on metric spaces. Although our improvement is a sublogarithmic factor, the issue is important from a theoretic viewpoint. Also our algorithm uses simple data structures that will not incur much overhead in a real implementation.

Aside from operations involving a problem-specific oracle, the only difficulty in implementing the GW algorithm is the clustering operation which determines the next components to merge. Goemans and Williamson's original implementation [12] uses time $O(n^2 \log n)$. This was improved to $O(n(n + \sqrt{m \log \log n}))$ [9] and $O(n\sqrt{m} \log n)$ [15]. Here and throughout this paper n and m denote the number of vertices and edges in the given graph, respectively. Regarding time

* This work was supported in part by Texas Advanced Research Program Grant 003658-0029-1999, NSF Grant CCR-9988160 and an MCD Graduate Fellowship.

bounds one should bear in mind that many applications of the GW algorithm are for the dense case $m = \Theta(n^2)$ (see below). We improve the above bounds to $O(n\sqrt{m})$. Cole et. al.[5] present a modified version of the GW clustering algorithm that runs in time $O(km \log^2 n)$. Here k is an arbitrary constant, and the approximation factor of the modified algorithm increases (i.e., worsens) by the small additive term $O(1/n^k)$. This time bound is a substantial improvement for sparse graphs. Still in the dense case this algorithm is slower than the original GW implementation (and slightly less accurate).

One reason that dense graphs arise is assumption of the triangle inequality. Because of this our algorithm gives the best known time bound $O(n^2)$ for these applications of the GW algorithm [12]: 4-approximation algorithm for the exact tree, exact path and exact cycle partitioning problems; 2-approximation algorithm for the prize-collecting TSP (see [14] for the TSP time bound); and finally the 2-approximation for minimum cost perfect matching (whose primary motivation is speed). The GW approximation algorithms for prize-collecting Steiner tree and TSP, on metric spaces, are used as subroutines in several constant factor approximation algorithms for the minimum latency problem and the k-MST problem. For minimum latency these include [3, 10] and the best-known 10.77-approximation algorithm of [7]. For k-MST they include [4], and the best-known 3-approximation of [7] for the rooted case and 2.5-approximation of [1] for the unrooted case. In all of these algorithms multiple executions of the GW algorithm dominate the running time, so our implementation improves these time bounds too (although the precise time bounds are higher).

For the approximation algorithm for survivable network design [9], [18], [8] the given graph can be sparse but there are additional quadratic computations. Our algorithm achieves time $O((\gamma n)^2)$ when the maximum desired connectivity is γ. (The bound of [9] is $O((\gamma n)^2 + \gamma n\sqrt{m \log \log n})$. For $\gamma = 2$ Cole et. al. avoid the quadratic computations and achieve time bound $O(km \log^2 n)$ for k as above.) For other applications of the GW algorithm where the time is dominated by clustering (generalized Steiner tree problem, prize-collecting Steiner tree problem, nonfixed point-to-point connection, and T-joins [12]) our bound is $O(n\sqrt{m})$, which improves the previous strict implementations of GW and improves the algorithm of Cole et. al. [5] in very dense graphs (but of course not in sparse graphs).

We obtain our results by solving the "dynamic vertex minimum problem" (DVMP). This problem is to keep track of the minimum cost edge in a graph where vertices can be inserted and deleted. Ignoring the graph structure it is obvious that $O(m \log n)$ is the best time bound possible, for m the total number of edges. Taking advantage of the graph structure we achieve time $O(n^2)$, for n the total number of vertices. This result immediately implies a time bound of $O(n^2)$ for GW clustering. It gives all the dense graph time bounds listed above. Our solution to DVMP is based on an amortized analysis of a system of binomial queues.

We apply the DVMP algorithm to implement the GW clustering algorithm on sparse graphs in time $O(n\sqrt{m})$. We actually solve the more general "merging minimum problem" (MMP). This problem is to keep track of the minimum cost edge in a graph whose vertices can be contracted. The costs of edges affected by the contraction are allowed to change, subject to a "monotonicity property." We solve the MMP using our DVMP algorithm and a grouping technique.

Section 2 gives our solution to the DVMP. This immediately implements GW clustering for dense graphs. Section 3 of the complete paper [11] solves the MMP; this section is omitted here because of space limitations.

2 Dynamic Vertex Minimum Problem

The *dynamic vertex minimum problem (DVMP)* concerns an undirected graph G where each edge e has a real-valued cost $c[e]$. The graph is initially empty. We wish to process (on-line) a sequence of operations of the following types:

`Add_vertex`(v)	:	add v as a new vertex with no edges.
`Add_edge`(e)	:	add edge e with cost $c[e]$.
`Delete_vertex`(v)	:	delete vertex v and all edges incident to v.
`Find_min`	:	return the edge currently in G that has smallest cost.

This section shows how to support `Add_vertex`, `Add_edge` and `Delete_vertex` operations in $O(1)$ amortized time, and `Find_min` in worst-case time linear in the number of vertices currently in G. For convenience we assume the edge costs are totally-ordered. If two edges actually have the same cost we can break the tie using lexicographic order of the vertex indices.

Our high-level approach is to store the edges incident on each vertex in a heap. Since the DVMP only involves the edge of globally minimum cost, these heaps ignore important information: An edge that is not the smallest in one of its heaps is not a candidate for the global minimum, even if it is the smallest in its other heap. We capture this principle using a notion of "active" and "inactive" edges (defined precisely below). This allows us to economize on the number of heap operations.

2.1 The Algorithm

This section presents the data structure and algorithm, and proves correctness of the implementation.

Our data structure is a collection of heaps. Our heaps are a variant of the standard binomial queue data structure [16]. As usual, each heap is a collection of heap-ordered binomial trees. However our structure differs from the standard definition in two ways. First and most importantly, one heap is allowed to contain an arbitrary number of binomial trees of a given rank. Second, a lazy strategy is used for deletion: Heap elements are marked when they get deleted (elements are initially unmarked). The root of a heap is never marked.

The data structure consists of $|V(G)|$ heaps, one for each vertex. We denote by $H(u)$ the heap associated with $u \in V(G)$. Elements of $H(u)$ correspond to edges incident on u. An edge $\{u, v\}$ appears in both $H(u)$ and $H(v)$; we differentiate these two elements with the notation (u, v) and (v, u), respectively. Let $twin(u, v)$ be synonymous with (v, u). $H(u)$ may contain marked elements (u, x) (for edges previously incident to u that have been deleted). But as already mentioned, such marked elements are never tree roots.

Each heap $H(u)$ is a collection of binomial trees divided into two groups: the *active* trees and the *inactive* trees. We sometimes refer to a whole tree by its root. Hence an *active root* is the root of an active tree. As usual the *rank* of an element e, denoted $rank(e)$, is the number of children it has, which is also the logarithm (base 2) of the size of the subtree rooted at e. The following invariant characterizes the active and inactive trees.

DVMP Invariant

(i) For all elements e, $rank(e) \leq rank(twin(e)) + 1$.
(ii) Consider a tree root f. If $rank(f) > rank(twin(f))$ then f is inactive. If $rank(f) \leq rank(twin(f))$ and f is inactive then $twin(f)$ is either active or a nonroot.
(iii) Consider a vertex u. $H(u)$ contains at most one active root per rank k, denoted $r_u(k)$ if it exists. At all times $s_u(k)$ points to the element with minimum cost among $\{r_u(k), r_u(k+1), \ldots\}$.

To better understand (ii) consider elements f and $twin(f)$ that are both roots. If the twin roots have equal rank then at least one of them is active. If the twins have unequal rank then the smaller rank element is active while the larger one is not.

Here is some motivation for the DVMP Invariant. The purpose of (i) is that the ranks of an item and its twin should not differ too much. To maintain the first part of (iii) we will sometimes merge two active trees of the same rank. This increments the rank of the resulting tree's root by one. If this happened too often the ranks of an item and its twin could differ by an arbitrary amount, violating (i). To avoid this when a root's rank is one more than that of its twin (ii) makes the root inactive. We never merge inactive trees. Hence we do not violate (i).

One consequence of the DVMP Invariant is that the minimum cost edge is easy to find:

Lemma 1. *If $\{u, v\}$ is the edge with minimum cost in G then either $s_u(0)$ or $s_v(0)$ points to it.*

Proof. Because $\{u, v\}$ is of minimum cost, (u, v) and (v, u) must be roots in $H(u)$ and $H(v)$, respectively. DVMP Invariant (ii) implies that either (u, v) or (v, u) is active. DVMP Invariant (iii) implies that in general, $s_u(0)$ points to the minimum element in an active tree of vertex u. Hence either $s_u(0)$ or $s_v(0)$ points to $\{u, v\}$. (The fact that nonroot elements can be marked, i.e., deleted, has no affect on this argument.) ∎

The procedure for `Find_min` follows directly from Lemma 1. We simply take the minimum cost element pointed to by $s_u(0)$, over all $u \in V(G)$.

The `Add_vertex`, `Add_edge` and `Delete_vertex` operations are implemented in a lazy fashion: They perform the least amount of work necessary to restore the DVMP Invariant. Each of these operations makes use of the routines `Activate`(e) and `Deactivate`(e). The purpose of these routines is to change a tree root $e = (u, v)$ from the inactive to the active state, or the reverse. Both these changes of state may violate DVMP Invariant (iii): Making a root e active may create two active roots of rank $rank(e)$. Making root e active or inactive may make $r_u(rank(e))$ or $s_u(0), \ldots, s_u(rank(e))$ out-of-date. The routines `Activate`(e) and `Deactivate`(e) repair these violations, as follows.

`Deactivate`$(e = (u, v))$

It is assumed that e is an active root. Furthermore deactivating e will not violate DVMP Invariant (ii).

1. Move the tree rooted at e to the set of inactive trees in $H(u)$.
2. If $r_u(rank(e)) = e$, set $r_u(rank(e)) =$ **nil**.
3. Update $s_u(0), \ldots, s_u(rank(e))$.

`Activate`$(e = (u, v))$

It is assumed that e is an inactive tree root and $rank(e) \leq rank(twin(e))$.

1. Remove the tree rooted at e from the set of inactive trees in $H(u)$.
2. Let $cur := e$.

   *The following loop sets $r_u(rank(cur)) = cur$ unless $r_u(rank(cur)) \neq$ **nil**, in which case merging is necessary.*
3. LOOP {
4.   Let $k := rank(cur)$.
5.   If $r_u(k) =$ **nil**
6.   Let $r_u(k) := cur$.
7.   Update $s_u(0), \ldots, s_u(k)$.
8.   EXIT THE LOOP.
9.   Otherwise, merge the trees rooted at cur and $r_u(k)$, and let cur be the resulting root.
10.   Set $r_u(k) :=$ **nil**.
11.   If $rank(cur) > rank(twin(cur))$
12.   `Deactivate`(cur).
13.   `Activate`$(twin(cur))$ if $twin(cur)$ is an inactive root.
14.   EXIT THE LOOP.
15. }

It is clear that `Deactivate` works correctly. `Activate` is more complicated because of the tail recursion in Step 13. Its correctness amounts to the following fact.

Lemma 2. `Activate` *eventually returns with the DVMP Invariant intact.*

Proof. We will prove by induction that each time Step 5 of `Activate` is reached,

(i) $rank(twin(cur)) \geq k = rank(cur)$;
(ii) the only possible violation of the DVMP Invariant is part (iii), specifically, $H(u)$ can contain two active nodes of rank k, cur and $r_u(k)$, and the values $s_u(0), \ldots, s_u(k)$ can be incorrect.

In the inductive argument we will also note that the lemma holds whenever `Activate` returns.

For the base case of the induction note that activating e (in Step 1) cannot introduce a violation of DVMP Invariant (i)–(ii). Hence the first time Step 5 is reached, only DVMP Invariant (iii) for u can fail and inductive assertion (ii) holds. Assertion (i) follows from the corresponding inequality on ranks in the entry condition of `Activate`.

Now consider Step 5. If $r_u(k) = \mathbf{nil}$, Steps 6–7 restore DVMP Invariant (iii). Then `Activate` returns with the DVMP Invariant holding, as desired. So suppose $r_u(k) \neq \mathbf{nil}$, i.e., there is a previously existing active tree of rank k.

The inequality $rank(f) \leq rank(twin(f))$ holds for both $f = cur$ (by inductive assumption) and $f = r_u(k)$ (by DVMP Invariant (ii)–(iii)). Step 9 merges trees cur and $r_u(k)$ and makes cur point to the new tree root, which has rank $k+1$. The previous inequality (along with DVMP Invariant (i)) shows that now $rank(cur)$ is equal to either $rank(twin(cur))$ or $rank(twin(cur)) + 1$. In the first case, since cur is active DVMP Invariant (ii) permits $twin(cur)$ to be active or inactive. Hence the algorithm proceeds to the next execution of Step 5 with inductive assertions (i)–(ii) intact. So the first case is correct.

In the second case DVMP Invariant (ii) requires that cur be inactive and $twin(cur)$ be active if it is a root. Step 12 makes cur inactive and `Deactivate` restores DVMP Invariant (iii). (Note the entry condition to `Deactivate` is satisfied.) The recursive call of Step 13 fixes up DVMP Invariant (ii) in its Step 1. (Again note the entry condition to `Activate` is satisfied.) Then Step 5 is reached with inductive assertions (i)–(ii) intact. This completes the induction.

It remains to show that `Activate` eventually returns. This is clear, since every time it reaches Step 5 the number of trees in the data structure has decreased (in the merge of Step 9). ∎

Now consider the remaining operations `Add_vertex`, `Add_edge` and `Delete_vertex`. `Add_vertex` is trivial. The routines for `Add_edge` and `Delete_vertex` are given below. `Delete_vertex` works in a lazy fashion: We mark heap elements when they get deleted. We keep a marked element in the heap as long as possible, i.e., until all its ancestors become marked.

`Add_edge`$(\{u, v\})$
1. Create rank 0 nodes (u, v) and (v, u).
2. Put (u, v) and (v, u) in the inactive sets of $H(u)$ and $H(v)$, respectively.
3. `Activate`(u, v).

`Delete_vertex`(u)

1. For each element $e \in H(u)$, mark $twin(e)$ as deleted.
2. For each tree root $f = twin(e)$ marked in Step 1,
3.   `Deactivate`(f) if it is active.
4.   For each unmarked element g that is in the tree of f and has all its ancestors marked,
5.   Designate g an inactive tree root (remove all its marked ancestors).
6.   If $twin(g)$ is an inactive root,
7.   If $rank(g) \leq rank(twin(g))$, `Activate`$(g)$
8.   Else `Activate`$(twin(g))$.

It is obvious that `Add_edge` is correct, so we turn to `Delete_vertex`. Step 1 can discard all the nodes of $H(u)$ since they are no longer needed. Step 4 finds the nodes g by a top-down exploration of the tree rooted at f. Step 2 ensures that every new tree root is found. Step 5 causes DVMP Invariant (ii) to fail if $twin(g)$ is an inactive root. In that case if g and $twin(g)$ have equal ranks one of them should be active; if they have unequal ranks the smaller rank element should be active. However all the rest of the DVMP Invariant is preserved in Step 5. Thus Steps 6–8 restore DVMP Invariant (ii). (Note the entry conditions to `Activate` and `Deactivate` are always satisfied.) We conclude that `Delete_vertex` works correctly.

We close this section with the final details of the data structure. Each value $twin(e)$ is represented by a pointer, so nodes (u, v) and (v, u) point to each other. The set of inactive trees in each heap $H(u)$ is represented as a doubly-linked list. Now almost every step of the four routines takes constant time. The exceptions are first, the updates of s_u: Step 3 of `Deactivate` takes $O(rank(e)+1)$ time and Step 7 of `Activate` takes $O(k+1)$ time. (We compute $s_u(i)$ in $O(1)$ time using the value of $s_u(i+1)$). Second, the top-down search in Step 4 of `Delete_vertex` amounts to $O(1)$ time for the root f plus $O(1)$ time for each binomial tree edge that is explored (and removed).

2.2 Timing Analysis

This section proves the claimed time bounds. It is immediate that the worst-case bounds for `Find_min` and `Add_vertex` are $O(n)$ and $O(1)$ respectively, where n is the current number of vertices. We show below that `Add_edge` and `Delete_vertex` use $O(1)$ amortized time.

To start off we charge each edge of G $O(1)$ time to account for work in its creation and destruction, specifically Steps 1–2 of `Add_edge` plus the possible time for marking the edge in Steps 1–2 of `Delete_vertex`. It is easy to see that the remaining work performed by our algorithm is linear in the number of comparisons between edge costs. (Specifically, the time for `Deactivate` is dominated by the comparisons to update s_u in Step 3; `Activate` is dominated by the comparison to merge binomial trees (Step 9) and the comparisons to update s_u (Step 7); in `Delete_vertex` each remaining unit of work corresponds to the destruction of a binomial tree edge, i.e., a previous comparison.) It therefore suffices to bound the number of comparisons.

We do this using the accounting method of amortized analysis [6, 17]. Define 1 credit to be the work required for 1 comparison. We will maintain the following invariant after each operation:

Credit Invariant

(i) Every heap element has $C = O(1)$ credits.
(ii) Every root of rank k has an additional $k + 4$ credits.
(iii) Every nonroot of rank k with a marked parent has an additional $2k + 5$ credits.

The precise value of the constant C will be determined below.

Note that each call `Deactivate`(e) uses $rank(e) + 1$ credits. We will require that each call `Activate`(e) is given $rank(e) + 1$ credits. Using this credit system the amortized cost of `Add_edge` is $2(C + 4) + 1$: $C + 4$ credits per rank 0 element created plus 1 credit for the call to `Activate`. Thus `Add_edge` takes $O(1)$ amortized time as desired.

Amortized Cost of `Activate` When `Activate`(e) is called $rank(e) + 1$ credits are available to be spent. We maintain that at each iteration of the loop (Step 3) at least $k + 1$ credits are available, for $k = rank(cur)$. This is clearly true for the first iteration.

Suppose in Step 5, $r_u(k) = \textbf{nil}$. The only remaining comparisons in this call to `Activate` are for updating $s_u(0), \ldots, s_u(k)$. We pay for these with the $k + 1$ available credits.

Suppose now $r_u(k) \neq \textbf{nil}$. Step 9 merges cur and $r_u(k)$, two rank k trees, producing a rank $k + 1$ tree, also denoted cur. The merge changes one root into a nonroot, releasing $k + 4$ credits. We use one credit to pay for the comparison of the merge; additionally the new rank $k + 1$ root requires one more credit. This leaves a total of $(k + 1) + (k + 2) = 2k + 3$ credits available.

Suppose Steps 12–14 are executed. We pay for `Deactivate`(cur) in Step 12 with $k + 2$ credits. (Actually we could save a comparison in `Deactivate`, since it need not update $s_u(rank(e))$.) Since $rank(twin(cur)) = k$, we can pay for the call to `Activate`$(twin(cur))$ in Step 13 (if necessary) with the remaining $k + 1$ credits.

Finally suppose the 'If' statement in Step 11 fails. The loop returns to Step 3 with $2k + 3 \geq k + 2$ available credits, as called for.

Amortized Cost of `Delete_vertex` Consider an operation `Delete_vertex`(u) and an element $(u, v) \in H(u)$ with rank k. DVMP Invariant (i) ensures that $rank(v, u) \leq k + 1$. When Step 1 marks (v, u) Credit Invariant (iii) requires credits to be placed on the children of (v, u). Let us temporarily assume this has been done and discuss the rest of `Delete_vertex` before returning to this issue.

In Step 3 a possible operation `Deactivate`(v, u) requires at most $k+2$ credits, paid for by the $k+4$ credits on (v, u).

Now consider an unmarked element g as in Step 4. The cost of discovering g and processing it in Steps 4–6 has already been associated with merge comparisons. In addition if $rank(g) = j$ we need $2j+5$ credits: Credit Invariant (ii) requires $j+4$ credits when g becomes a root (Step 5), plus we need at most $j+1$ credits to pay for the call to `Activate` for g or its twin (Steps 7–8). Credit Invariant (iii) for g gives the $2j+5$ needed credits.

It remains only to explain how Credit Invariant (iii) is maintained when (v, u) is marked in Step 1. (v, u) has at most $k+1$ children, one child of each rank $i = 0, \ldots, k$. So Credit Invariant (iii) requires a total of at most $\sum_{i=0}^{k}(2i+5) = (k+1)(k+5)$ credits. We consider two cases, depending on whether or not (u, v) is a root of $H(u)$.

$H(u)$ contains at most $|H(u)|/2^{k+1}$ rank k nonroot elements (u, v), since the parent of such a nonroot has 2^{k+1} descendants. So the total cost associated with deleting all nonroots (u, v) of all ranks k is bounded by

$$\sum_{k=0}^{\infty} \frac{|H(u)| \cdot (k+1)(k+5)}{2^{k+1}}.$$

Recall that $\sum_{k=0}^{\infty} \frac{1}{2^k} = \sum_{k=0}^{\infty} \frac{k}{2^k} = 2$ and $\sum_{k=0}^{\infty} \frac{k^2}{2^k} = 6$. Hence the above sum is at most $|H(u)|(6+12+10)/2 = 14|H(u)|$. We pay for this by taking 14 credits from each element of $H(u)$, assuming $C \geq 14$ in Credit Invariant (i).

Next consider a rank k root (u, v). The elements in the binomial tree of (u, v) now have a total of $k+4+(C-14)2^k$ credits by Credit Invariant (i)–(ii). Choosing $C = 18$ makes this quantity at least $(k+1)(k+5)$, because $k^2+5k+1 \leq 2^{k+2}$ for every $k \geq 0$. (This inequality follows by induction using base case $k \leq 1$. For the inductive step we use the identity $2k+6 \leq 2^{k+2}$ for every $k \geq 1$.) Hence we can pay the cost associated with (u, v).

Theorem 1. *The dynamic vertex minimum problem can be solved in amortized time $O(1)$ for each* `Add_vertex`*,* `Add_edge` *and* `Delete_vertex` *operation and worst-case time $O(n)$ for each* `Find_min` *when the graph contains exactly n vertices.*

We close with three remarks. First, the constant C in the analysis can be lowered because the algorithm performs unnecessary comparisons. For instance in `Delete_vertex`(u), various executions of `Activate` may update $s_v(\cdot)$ values for the same vertex v. But only one update is sufficient.

Second, an actual implementation of this data structure will probably be more efficient if we modify the DVMP Invariants slightly. Invariant (i) can be changed to $rank(e) \leq c_1 \cdot rank(twin(e)) + c_2$, for constants c_1, c_2. This allows DVMP Invariant (ii) to be relaxed so there are fewer activates and deactivates. This speeds up the algorithm in practice. Theorem 1 remains valid.

Finally, if necessary we can ensure that the space is always $O(m)$, for m the current number of edges. The idea is to reconstruct the data structure whenever there are too many marked elements.

References

1. Arya, S., Ramesh, H.: A 2.5-factor approximation algorithm for the k-MST problem. Inf. Proc. Letters **65** (1998) 117–118
2. Agrawal, A., Klein, P., Ravi, R.: When trees collide: An approximation algorithm for the generalized Steiner problem on networks. SIAM J. Comp. **24** (1995) 440–456
3. Blum, A., Chalasani, P., Coppersmith, D., Pulleyblank, B., Raghavan, P., Sudan, M.: The minimum latency problem. Proc. 26th Annual ACM Symp. on Theory Comput. (1994) 163–171
4. Blum, A., Ravi, R., Vempala, S.: A constant-factor approximation algorithm for the k-MST problem. J. Comp. Sys. Sci. **58** (1999) 101–108
5. Cole, R., Hariharan, R., Lewenstein, M., Porat, E.: A faster implementation of the Goemans-Williamson clustering algorithm. Proc. 12th Annual ACM-SIAM Symp. on Disc. Alg. (2001) 17–25
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. 2nd edn. McGraw-Hill, NY (2001)
7. Garg, N.: A 3-approximation for the minimum tree spanning k vertices. Proc. 37th Annual Symp. on Foundations Comp. Sci. (1996) 302–309
8. Goemans, M., Goldberg, A., Plotkin, S., Shmoys, D., Tardos, E., Williamson, D.: Improved approximation algorithms for network design problems. Proc. 5th Annual ACM-SIAM Symp. on Disc. Alg. (1994) 223–232
9. Gabow, H.N., Goemans, M.X., Williamson, D.P.: An efficient approximation algorithm for the survivable network design problem. Math. Programming B **82** (1998) 13–40
10. Goemans, M.X., Kleinberg, J.: An improved approximation ratio for the minimum latency problem. Proc. 7th Annual ACM-SIAM Symp. on Disc. Alg. (1996) 152–158
11. Gabow, H.N., Pettie, S.: The dynamic vertex minimum problem and its application to clustering-type approximation algorithms. Univ. of Colorado TR# CU-CS-928-02 (2002)
12. Goemans M.X., Williamson, D.P.: A general approximation technique for constrained forest problems. SIAM J. Comp. **24** (1995) 296–317
13. Goemans M.X., Williamson, D.P.: The primal-dual method for approximation algorithms and its application to network design problems. in Approximation Algorithms for NP-hard Problems, D.S. Hochbaum (ed.) (1997) 144–191
14. Johnson, D.S., Minkoff, M., Phillips, S.: The prize collecting Steiner tree problem: Theory and practice. Proc. 11th Annual ACM-SIAM Symp. on Disc. Alg. (2000) 760–769
15. Klein, P.: A data structure for bicategories, with application to speeding up an approximation algorithm. Inf. Proc. Letters **52** (1994) 303–307
16. Sedgewick, R.: Algorithms in C++. 3rd edn. Addison-Wesley, Reading, MA (1998)
17. Tarjan, R.E.: Amortized computational complexity. SIAM J. on Algebraic and Discrete Methods **6** (1985) 306–318
18. Williamson, D.P., Goemans M.X., Mihail, M., Vazirani, V.V.: An approximation algorithm for general graph connectivity problems. Combinatorica **15** (1995) 435–454