CUCS-105-84

β-trees, γ-systems, and a Theorem on F-heaps

March 1984

Zvi Galil
Department of Computer Science
Columbia University
and
Tel-Aviv University

Thomas Spencer
Department of Computer Science
Stanford University

Contact Author: Zvi Galil

Summary of Results

This paper presents three new results on data structures which lead to improvement in two classical algorithms.

1. We consider β-trees, the analog of 2-3 trees and B-trees. These are B-trees of order β, where β is *not fixed*. We show that whenever the number of operations is m and the number of deletions and splits is n, the time complexity is $O(m \log_\beta m)$ if we take $\beta = \lceil m/n + 1 \rceil$. This bound is linear ($O(m)$) for $m = \Omega(n^{1+\alpha})$ and any $\alpha > 0$.

2. We prove that (the miraculous) F-heaps still require time $O(m + n \log n)$ (n = number of elements, m = number of operations that include decrease key) even if we allow split operations, provided they are given in advance.

3. We introduce γ-systems, a new type of a priority queue. They deal with the following setting. There are m elements inserted in n multi-inserts. Elements have keys and can die in a structured way that arises in some graph algorithms. There are n operations of "find and delete a live element with minimum key". No one of the known data structures (including F-heaps) seems capable of supporting this setting in $o(m \log m)$ time. For $\gamma = \lceil \sqrt{m/n \log^2 n + 1} \rceil$, γ-systems handle this setting in time $O(m \log_\gamma n)$. This bound is linear for $m = \Omega(n^{1+\alpha})$ and any $\alpha > 0$.

4. We improve the time bound of the algorithm that finds a maximum weighted matching in general graphs to $O(mn \log_\gamma n)$. This bound is better than the $O(mn \log(n^2/m))$ conjectured by Tarjan [**9**, p. **123**]. The latter is $O(mn)$ only for $m = \Omega(n^2)$.

5. We improve the time bound for the algorithm that finds an optimum branching to $O(m \log_\gamma n)$, answering a question by Tarjan [**8**, p. **34**].

0. Introduction.

Recent progress in the design of efficient data structures has been remarkable. This progress is documented in the new book by Tarjan [9]. Even operations manipulating trees can be implemented in amortized time $O(\log n)$; i.e., time $O(m \log n)$ for m operations. (See [9 Chapter 5].) Recently, attempts have been made to develop data structures that implement some sets of operations in (amortized) time $o(\log n)$ or even $O(1)$ time. Sometimes this is only possible when the number of operations (m) is large enough.

Example 1: Maintaining disjoint sets.

The current best bound is $O(m\alpha(m, n))$ (m = number of operations, n = number of elements). For all practical purposes $\alpha(m, n)$ = constant, and $\alpha(m, n) \leq 2$ for $m \geq n \log^*(n)$. A lower bound of $\Omega(m\alpha(m, n))$ is known for a certain family of pointer manipulating algorithms. (See [9 Chapter 2].) An important theoretical open problem is whether or not the bound can be improved to $O(m)$. A linear time algorithm for a family of important special cases was designed in [3].

Example 2: d-heaps

Johnson [5] suggested using heaps with d sons instead of 2. As a result the height of the heap was reduced to $h = \lceil \log_d m \rceil$. (We use m - - the number of operations - - to bound the number of elements in the heap. In most applications, this number is n and we have $h = \lceil \log_d n \rceil$. In graph applications $m = O(n^2)$ and the difference does not matter.) He observed that some operations (like insert, decrease key) need only time $O(h)$ while others (like delete, increase key) may need $O(dh)$ because going up the tree we need to consider all d sons of each node we traverse. As a result, if out of the m operations only n are the more expensive operations, the total time is $O(mh + nhd)$. The two terms are balanced when $d = \lceil m/n + 1 \rceil$ and give time $O(m \log_d m)$. The bound is linear (or each operation costs $O(1)$) if $m = \Omega(n^{1+\alpha})$, $\alpha > 0$.

Example 3: Fibonacci heaps (F-heaps).

Very recently Fredman and Tarjan [1] introduced F-heaps, which improve the performance of the d-heaps. They may turn out to be the ultimate data structure. F-heaps support the following setting: m operations which include find minimum, decrease key, n inserts and k deletions in time $O(m + k \log n)$. The $k \log n$ term is unavoidable because we can sort n elements using only comparisons ($m = k = n$). Hence, for $m = \Omega(k \log n)$ each operation costs $O(1)$.

Our first three results are similar to the examples above. We consider three settings and design data structures that support m operations in time $O(m)$ provided m is large enough (in terms of the other parameter n). As in Examples 2 and 3, we consider mixes of "expensive operations" and "cheap operations". When the number of cheap operations is relatively large the total time is linear in the number of operations. These settings arose in our attempt to improve the best algorithm for finding maximum weighted matching in general graphs.

1. β-trees

In the first setting we have elements from a totally ordered universe. Elements have keys - - real numbers - - associated with them. The abstract data structure supports the following operations:

— initialize a new set S to the empty set;

— insert an element a with a key x into a set S next to an element b;

— delete an element a;

— find an element in a set S with minimum key;

— concatenate S_1, S_2 (all elements in S_1 are smaller than all elements of S_2);

— split S according to an element a in S (put all elements $\leq a$ in a new set);

The number of operations is m and the number of splits and deletions is only n. Note that we actually have two orders: one defined on the elements and one on their keys.

Consider B-trees of order β [6, p. 473] or in short β-trees. These are the analog of 2-3 trees except that every internal node has at most $2\beta - 1$ sons and every internal node except for the root has at least β sons. The root has at least 2 sons, and all the leaves are at an equal distance $h = O(\log_\beta m)$ from the root. We manipulate β-trees exactly as we handle 2-3 trees. Elements are stored in the leaves and are ordered from left to right, and each internal node contains the smallest of the keys of its sons (i.e., the smallest key of an element stored in a leaf of the subtree rooted at the node).

The following two facts are immediate, since β-trees are manipulated as 2-3 trees. The β factor appears whenever we have to inspect all the sons of a node to determine its key (either because the node is new or because its key might have changed due to a deletion of one or more of its sons).

Fact 1. The deletions and splits cost $O(nh\beta)$.

Fact 2. The concatenations and inserts cost $O(mh + N\beta)$, where N is the number of splitting of internal nodes.

Lemma 1. $N = O(m/\beta + nh)$.

Proof: The lemma follows from the accounting of the full nodes in the collection of β-trees. A *full* node is one with at least β sons. (Among the internal nodes of a β-tree, only the root may be nonfull.) Initially, the number of full nodes is 0. Finally, their number is $O(m/\beta)$, because this is the bound on the number of full nodes in a collection of β-trees with m leaves. The splits and deletions may decrease the number of full nodes by at most $O(nh)$. New full nodes are generated only when we split an internal node. Hence, the number N of such splits is bounded above by $O(m/\beta + nh)$. ∎

Theorem 1 follows from Facts 1, 2 and Lemma 1.

Theorem 1. For $\beta = \lceil m/n + 1 \rceil$, β-trees support the first setting in time $O(m \log_\beta m)$.

So, β-trees are to 2-3 trees as d-heaps are to binary heaps. They lead to the same improvement in a slightly less obvious way. Note that we can include the operation decrease key without affecting the time.

2. A Theorem on F-heaps

The second setting is a special case of the first setting. We have only n elements given in advance; we have no concatenations and the splits are given in advance as follows: A forest of binary trees with n leaves is given. The leaves are the n elements and each set corresponds to a tree. (Ordering of the elements is implicit in the binary tree but we do not use it.) Splitting a set corresponds to deleting the root of the appropriate binary tree. The set is split into the set of leaves of the left and the right subtrees. This is not exactly a split according to an element, but can be interpreted in this way (split according to the rightmost leaf of the left subtree.) Note that the binary trees are not a data structure but a part of the input that specifies potential splits.

Such a setting arises when we are given a structure (e.g. a graph), where elements are basic elements of the structure (e.g. vertices), and elements belong to some sets (e.g. connected components). In this setting an element is inserted when it is given a (finite) key for the first time. We can do away with inserts if we simply initialize all keys to infinity, and when an insert occurs we replace it with a decrease key operation.

Our implementation uses F-heaps. That is, in addition to the binary tree that contains the information how to split it, each set is represented as an F-heap. For the sake of brevity we do not describe F-heaps here. We refer the reader to [1]. Out of the seven operations on F-heaps (make heap, insert, find min, delete min, meld, decrease key, and delete) we disallow only meld. Instead we allow the split operation mentioned above. As explained above, we replace each insert with a decrease key. We replace the splits by deleting the elements of the smaller half: We traverse the (left or right) subtree which has the smaller size, and delete from the F-heap all the elements in this half. The F-heap corresponding to the larger half is the remaining F-heap. The F-heap corresponding to the smaller half is initialized to be a set of (trivial) trees, where each tree consists of a single (deleted) element. This immediately yields an $O(m + n\log^2 n)$ bound. (Consider the deleted elements as new elements, and note that there are $O(n\log n)$ new elements.) A more careful consideration yields the following theorem.

Theorem 2. F-heaps handle the second setting in time $O(m + n\log n)$.

Recall that the potential φ of a collection of F-heaps is the number of trees plus twice the number of marks on non-root nodes. In the analysis below we separate the operation of delete (and decrease key) from the cascading cuts that may follow it: We assume that after such an operation a node may get a second mark. Following the operation the necessary sequence of cuts is performed. Similarly, we separate from the delete min the linking that may follow it.

Lemma 2. If we delete $k \leq n/2$ elements from an n element F-heap, then the potential of the remaining heap increases by at most $O(k\log(n/k))$.

Proof: The increase in potential is bounded by $2k$ (for the up to k additional marks) plus the number of new trees. The latter is bounded by the number of sons of the deleted elements. This number is maximal when all n elements are in one heap; the heap is full (i.e., no cuts have been made); and the k elements are at the top of the heap (those with the highest rank). Assume we delete these elements. Call a deleted element *important* if one of its sons is not deleted. The number of sons of deleted elements is bounded by k plus the number of sons of important elements. The latter was estimated in Lemma 2 of [1] to be $O(k(\log(n/k)+1))$. In our case the total is at most $O(\log(n/k))$ because $k \leq n/2$. ∎

Proof of Theorem 2: Define $t(n, m, \varphi)$ to be the maximal time needed for m operations on a collection of F-heaps with n elements and initial potential φ. A cascading cut, and a linking are not operations but must be accounted for. We write the recursive inequalities for $t(n, m, \varphi)$ for the various operations, assuming t is monotone in each variable.

For a cut, and for linking two trees in delete min (which are the two operations that do not appear in the operation count):

$$t(n, m, \varphi) \leq t(n, m, \varphi - 1) + 1.$$

For makeheap, findmin, and decrease key:

$$t(n, m, \varphi) \leq t(n, m - 1, \varphi + 1) + 1.$$

For delete and delete min in which $k \leq \log n$ new trees are formed:

$$t(n, m, \varphi) \leq t(n - 1, m - 1, \varphi + k + 2) + \log n.$$

(k for the new trees, 2 for the possible mark, and $\log n$ for the time of the operation not counting the linkings.)

For split (by Lemma 2):

$$t(n, m, \varphi) \leq \max_{\substack{1 \leq k \leq n/2 \\ m_1 + m_2 + 1 = m}} [t(k, m_1, k) + t(n - k, m_2, \varphi + ck \log(n/k)) + k].$$

An easy induction implies that $t(n, m, \varphi) = O(m + \varphi + n \log n)$. The theorem follows since initially $\varphi \leq n$. ∎

3. γ-Systems

In the third setting we have m elements with real numbered keys which are inserted in n multi-insert operations. (Each time we insert a set of elements.) Elements can die implicitly. We know whether a particular element died only if we consider that element. We have n operations of the form "find and delete a live element that has minimum key". An obvious way to implement this setting takes $O(m \log m)$ time. We do not know whether it is possible to implement this setting in time $o(m \log m)$.

Fortunately, we need only a special case of the third setting. We have a graph with n vectices and $m \leq n^2$ edges. The elements are the edges of the graph (the keys are the weights of the edges). A process *labels* vertices of the graph and occasionally shrinks a set of labeled vertices to form a new labeled vertex called a *supernode*. We also refer to vertices in the original graph that have not been shrunk as (trivial) supernodes. Consequently, each original vertex belongs to one supernode at any time. The inserted elements are all edges whose two endpoints are labeled. The dead edges are all edges whose endpoints belong to the same supernode. We call this process a *shrinking process*.

Shrinking processes turn out to be the bottleneck in the two applications mentioned below. No one of the known data structures seems to be useful for implementing a shrinking process in time $o(m \log n)$.

Let γ be an integer. Let p.q. be any efficient priority queue, say a heap. A γ-system will be a collection of p.q.'s, organized as follows. We associate p.q.'s with some subsets of (not necessarily all) vertices that belong to the same supernode. At any time the sets associated with the various p.q.'s constitute a partition of the set of labeled vertices. The p.q. associated with a set S contains edges (u, v), $v \in S$, u was labeled before v. The *size* of a p.q. is the size of its set (not the number of edges in it!). A p.q. of size s is of *rank* k if $\gamma^k \leq s < \gamma^{k+1}$. The number of ranks $r \leq \lceil \log_\gamma n + 1 \rceil$.

P.q.'s will be always *clean*; a p.q. associated with S will have no edges with both endpoints in S and for any $u \notin S$ at most one edge (u, v), $v \in S$ (namely, the edge with the minimum key). When a p.q. is generated it is *cleaned*. As a result it has only edges (u, v), $v \in S$, u in a supernode that does not contain S, and at most one edge per different supernode. However, when new supernodes are generated, this higher degree of cleanliness is not preserved. Note that an edge that is discarded in cleaning can never be a live edge with the minimum key. An easy algorithm yields:

Lemma 3. A p.q. with N elements can be cleaned in time $O(N)$.

Proof: Assume each supernode has a name i, $1 \leq i \leq n$. We scan all edges of the p.q., using an array for keeping the minimum cost edges and a list for keeping the used entries of the array. In the j-th entry of the array we store (t, e) where t is the time of the start of the cleaning and e the minimum edge (so far) among edges (u, v), $v \in S$, u in system j. The list and the time t eliminate the need for initializing the array. The cleaning is accomplished by one traversal of the heap and one traversal of the list. ∎

A γ-system Γ corresponds to the γ-ary representation of an integer $a = \sum_{i=0}^{k} a_i \gamma^i$, $0 \leq a_i < \gamma$. The system has a_i p.q.'s of rank i. The rank of Γ is k (the maximal rank of its p.q.'s) and its size s is the total size of its p.q.'s. The system satisfies the following constraint: the total number of elements in sets associated with p.q.'s of ranks $\leq i$, $(i \leq k)$ is smaller than γ^{i+1}. As a result $\gamma^k \leq s < \gamma^{k+1}$. Note that the number of p.q.'s in Γ is $O(\gamma r)$.

Our algorithm requires several γ-systems. Each system has one p.q. called the *system p.q.* which contains one element (with the minimum key) from each p.q. of the system. In addition we have one *main* p.q. which contains one element (with the minimum key) from each system.

To implement a shrinking process, we have a γ-system for each supernode. When a vertex v is labeled we generate a new system that consists of one p.q. We first put in this p.q. all edges from labeled vertices to v, and then clean it. When we look for the minimum live element we use the main p.q., the system p.q.'s and the p.q.'s of the systems for deleting dead edges until the minimum in the main p.q. is live. (We need to do this because despite the cleaning of p.q.'s there may still be dead edges in the various p.q.'s.) When we shrink a set of supernodes, we simply merge the corresponding γ-systems.

We only have to show how to implement merging of two systems (see Figure 1). Assume we merge two systems Γ_1 and Γ_2 of ranks $k_1 \leq k_2$ to form a new system Γ. We say that j is *a carry* if the sum of the sizes of p.q.'s of ranks $\leq j$ in the two systems together is at least γ^{j+1}. Let k be the highest carry, $k = -1$ if there is no carry.

Case 1 ($k_1 = k_2$): Γ consists of all queues of rank $> k$ (from either system) plus a queue of rank $k+1$ (if $k > -1$) that is the result of merging and cleaning all smaller queues.

Case 2a ($k < k_1 < k_2$): Γ contains all queues of rank $\geq k_1$ from Γ_2 and a queue of rank k_1 which is formed by merging and cleaning all of the queues from Γ_1 and all of the queues from Γ_2 with rank $< k_1$.

Case 2b ($k_1 < k_2$ and $k \geq k_1$): This is like Case 2a except that all queues of rank $\leq k$ from Γ_2 are merged and cleaned.

Fact 3. If $k_1 < k_2$, then Γ consists of old p.q.'s from Γ_2 and one new p.q. which has the smallest rank in Γ.

Theorem 3. Using γ-systems, the time bound for implementing a shrinking process is $O(mr + n\gamma^2 r^2 \log n)$. It is $O(m \log n / \log \gamma)$ for $\gamma = \lceil \sqrt{m/n \log^2 n} + 1 \rceil$.

Proof: The time for initializing systems (when vertices are labeled) is $O(m)$. When we merge two systems, the time to determine the carry is $O(\gamma r)$ for a total of $O(n\gamma r)$. The two most time consuming parts in the implementation are (1) cleaning and (2) deletion of dead edges. The first is $O(mr)$ because of Lemma 3 and the fact that each time an edge is considered in cleaning either it moves to a higher rank p.q. or to a higher rank system. The crucial lemma (Lemma 4) yields the second term in the bound given in Theorem 3. ∎

Lemma 4. The total number of deletions of dead edges is $O(n\gamma^2 r^2)$.

Proof: We consider a dead edge e in a p.q. A of rank k.

Case 1: The other endpoint of e is in a p.q. of rank $\leq k$.

Consider an *initial* rank k p.q., one that was just generated, and none of whose parts was a p.q. of rank k before. There are $\leq n/\gamma^k$ initial rank k p.q.'s. Consider the evolution of this initial rank k p.q. until it becomes a part of a p.q. of higher rank or until the end. Fewer than γ^{k+1} vertices can be in sets corresponding to p.q.'s of rank $\leq k$ in the same system. So during its lifetime the initial rank k p.q. can contribute $< \gamma^{k+1}$ case 1 deletions. Recall that each time the p.q. changes rank it is cleaned, and consequently no edge can be deleted twice.

Hence, the total number of case 1 deletions per rank $(k) \leq \gamma n$, and at most $\gamma r n$ in total.

Case 2: The other endpoint of e is in a p.q. of rank $> k$.

Here we count (differently) the number of such deletions in systems of rank k. Such case 2 edges are generated only when we merge two systems of rank k: Whenever a system of rank k is generated from lower systems it is cleaned entirely; and whenever we merge a lower system with a system of rank k (case 2 in merge) the new p.q. has smallest rank in the new system (Fact 3), and this new p.q. is cleaned. In a merge of two systems or rank k, the number of case 2 edges introduced is at most the number of p.q.'s in the new system times the number of vertices in the supernode which in turn is $O(\gamma r \gamma^{k+1})$. The number of merges of two rank k systems is $< n/\gamma^k$.

Hence, the total number of case 2 deletions per rank $(k) \leq O(\gamma^2 r n)$, and at most $O(\gamma^2 r^2 n)$ in total. ∎

Note that γ-systems are in a way a generalization of binomial queues [10]. The latter correspond to binary representation of integers. However, here our digits do not correspond to exact powers of γ (a different, more complicated, version of γ-systems may have exact powers of γ), and the size of the p.q.'s is not the number of elements in them. Had binomial queues been considered in the context of shrinking processes, they would have yielded an $O(m \log n)$ time bound.

4. Applications

We considered the settings above in an attempt to improve the maximum weighted matching algorithm for general graphs. The best algorithms were $O(n^3)$ [2, 7] and $O(mn \log n)$ [4]. Of course one could use a combined version to get $O(\min(n^3, mn \log n))$. Yet it would be better to derive *one* algorithm that dominates both. Tarjan [9, p. 123] conjectured an $O(mn \log(n^2/m))$ algorithm which equals $O(mn)$ only for $m = \Omega(n^2)$.

A similar situation still exists with network flow. The best algorithms are $O(n^3)$ and $O(mn \log n)$ and Tarjan conjectured a possible improvement to $O(mn \log(n^2/m))$. (See [9, Chapter 8].)

A similar situation existed with the shortest path problem (with non-negative lengths). The best algorithms were $O(n^2)$ and $O(m \log n)$. But using a d-heaps an $O(m \log_{\lceil m/n+1 \rceil} n)$ algorithm was developed which dominated both. (See [9 Chapter 7].) Using F-heaps it was further improved to $O(m + n \log n)$ [1].

Our first application (the motivation for this work) is the desired improvement for weighted matching.

Theorem 4. Maximum weighted matching in general graphs can be found in time $O(mn \log_\gamma n)$, $\gamma = \lceil \sqrt{m/n \log^2 n + 1} \rceil$.

Note that this time bound equals $O(mn)$ for $m \geq \Omega(n^{1+\alpha})$ and any $\alpha > 0$. The only way to get an $o(mn)$ algorithm is to use an entirely new approach to the problem.

We now sketch the proof of Theorem 4. We assume familiarity with [4]. The improved bound comes from a better implementation of the p.q.'s in [4]. Theorem 2 is used for the edges of type S-T and S-free. We have one F-heap for every T-blossom (free blossom). The split is needed for expanding T-blossoms. In the F-heap we have one element for every vertex in the blossom. The element corresponding to a vertex j is an edge (i, j) with the smallest *slack* $(\pi_{i,j})$ where i is any S-vertex. We actually have an additional operation of changing the slack of all elements in certain F-heaps. This is handled as in [4]. As a result the $p.q._2$ of [4] is maintained in time $O(m + n \log n)$.

Theorem 3 is used to maintain the edges of type S-S. The supernodes in the shrinking process are the blossoms, and the labeled vertices are the S-vertices. When a new S-blossom is generated, we first make each T-vertex an S-vertex by labeling it in the implementation of the shrinking process. Subsequently, we merge the systems corresponding to the T-vertices and to the S-subblossoms. As a result, the $p.q._1$ for edges of type S-S is maintained in time $O(m \log_\gamma n)$.

The second application is the problem of finding an optimum branching (directed spanning forest). The best algorithms for this problem have time bounds $O(n^2)$, and $O(m \log n)$ [8]. It was asked in [8] whether an $o(m \log n)$ algorithm exists. Theorem 5 shows that the answer is yes if $m = \omega(n \log^2 n)$. It yields a linear time algorithm if $m = \Omega(n^{1+\alpha})$ and $\alpha > 0$.

Theorem 5. Optimum branching can be found in time $O(m \log_\gamma n)$, $\gamma = \lceil \sqrt{m/n \log^2 n + 1} \rceil$.

The algorithm in [8] constructs a subgraph of the given directed graph by choosing one edge at a time. The optimum branching is derived from the final subgraph in linear time. At any time the subgraph is partitioned into *weak components (w.c.'s)*. These are the connected components of the underlying undirected subgraph. Each weak component is a directed tree of *strongly connected components (s.c.c.'s)*. The root of such a tree is a *root component*. The minimum cost edge incident

to a root component is chosen and added to the subgraph. If it comes from a different w.c., then the new w.c. consists of the two w.c.'s. We call the new edge a *grow edge*, and the step a *grow step*. If the edge comes from the same w.c., a cycle of s.c.c.'s is generated and all the s.c.c.'s on the cycle form a new s.c.c. The cost of all edges (i,j), j in the new s.c.c. are decremented by some amount that depends on the original s.c.c. of j. We call the new edge a *reduce edge*, and the step a *reduce step*.

In this shrinking process all the nodes are labeled initially, and the supernodes are s.c.c.'s. The γ-system that corresponds to an s.c.c. contains edges of the form (i,j), j in the s.c.c. In a reduce step we merge the corresponding systems. The only change is that the main p.q. contains only representatives from root s.c.c.'s. Consequently, in each grow step we delete an element from the main p.q. (the one that corresponds to the s.c.c. that stopped being a root). In a reduce step we delete and insert (or alternatively, change a key, possibly even increase the key) from the main p.q. The uniform change of keys is handled as in [8] (storing in a nonroot the difference between its key and the key of its father).

Acknowledgement: Bob Tarjan pointed out the possiblity that γ-systems may be applicable to the problem of finding optimum branching; Stuart Haber gave many helpful suggestions; Delores Ng typed the beautiful (TeX) manuscript; and Kerny McLaughlin produced the wonderful Figure (with the help of the MacIntosh).

References

[1] M. L. Fredman and R. E. Tarjan, "Fibonacci Heaps and Their Uses", *Manuscripts*, January 1984.

[2] H. N. Gabow, "Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs", Ph.D. Thesis, Dept. Electrical Engineering, Stanford Univ., Stanford, CA, 1973.

[3] H. N. Gabow and R. E. Tarjan, "A Linear-Time Algorithm for a Special Case of Disjoint Set Union", *Proc. Fifteenth Annual ACM Symposium on Theory of Computing*, 1983, 246-251.

[4] Z. Galil, S. Micali and H. N. Gabow, "Priority Queues with Variable Priority and an $O(EV \log V)$ Algorithm for Finding Maximal Weighted Matching in General Graphs", in *Proc. 23rd Annual IEEE Symposium on Foundations of Computer Science*, 1982, pp. 255-261.

[5] D. B. Johnson, "Efficient Algorithms for Shortest Paths in Sparse Networks", *J. Assoc. Comput. Mach.*, 24 (1977), pp. 1-13.

[6] D. E. Knuth, "The Art of Computer Programming, Vol. 3: Sorting and Searching", *Addison-Wesley*, Reading, MA, 1973.

[7] E. L. Lawler, "Combinatorial Optimization: Networks and Matroids", *Holt, Rinehart and Winston*, New York, 1976.

[8] R. E. Tarjan, "Finding Optimum branchings", *Networks*, 7 (1977), pp. 25-35.

[9] R. E. Tarjan, "Data Structures and Network Algorithms", *SIAM Publication*, 1983.

[10] J. Vuillemin, "A Data Structure for Manipulating Priority Queues", *Comm. ACM*, 21 (1978), pp. 309-314.

[11] H. N. Gabow and Z. Galil, "Efficient Implementation of Graph Algorithms using Contraction", in preparation.

Postscript

Very recently Theorems 4 and 5 were improved. Let $t(n, m) = O(m \log\log\log_{\lceil m/n+1 \rceil} n + n \log n)$. In [11], Gabow and Galil derived $nt(n, m)$ algorithm for weighted matching and a $t(n, m)$ algorithm for optimum branching. The algorithm for matching still uses Theorem 2 for the edges of type S-T, S-free. The improved algorithms use an algorithm for a special case of a shrinking process with the time bound of $t(n, m)$. Each algorithm uses it in a different way. As a result both, and in particular the one for branching, are more complicated than the algorithms above that use γ-systems. Moreover, one can show that γ-systems handle (in time $O(m \log_\gamma n)$) the most general type of shrinking process in which the graph is directed (as in branching) and vertices are labeled on-line (as in matching). None of the improved algorithms handles this case.

Figure 1. Merging two γ-systems ($\gamma = 4$) or funny arithmetic in γ-ary notation. The units for each digits are not the same; e.g. the apples are not identical (an apple's size s satisfies $4^2 \leq s < 4^3$). Consequently, even two apples (in $2b$) can generate a carry. The p.q.'s in the box on the left are merged and cleaned to yield the marked p.q. on the right.