Summary of Results

1. We define a graph problem which we refer to as the component merging problem. Versions of the problem appear as bottlenecks in various graph algorithms. We show how to solve an important special case of the problem in time $t(m, n) = O(m \log \log \log_d n)$, where the graph has $n$ vertices, and $m$ edges, and $d = \max(m/n, 2)$. The solution makes use of a special data structure. The performance of the data structure is sped up by introducing a new algorithmic tool called packets.

2. An immediate application of the first result is an $O(t(m, n))$ algorithm for finding minimum spanning trees in undirected graphs without using F-heaps. This time bound is inferior to Fredman and Tarjan's $O(m\beta(m, n))$ algorithm [7] which uses F-heaps, where $\beta(m, n) = \min\{i|\log^i n \leq m/n\}$. However, by adding packets to their algorithm we derive an $O(m \log(\beta(m, n)))$ algorithm.

3. Using the first result, we derive an $O(n(t(m, n) + n \log n))$ algorithm for finding maximum weighted matching in general graphs. This settles an open problem posed by Tarjan [16, p.123], where the weaker bound of $O(mn \log(n^2/m))$ was conjectured.

4. Using the first result, we derive an $O(t(m, n) + n \log n)$ algorithm for finding optimum branching in directed graphs. This answers affirmatively a question by Tarjan [18, p.34] (whether an $o(m \log n)$ algorithm exists).

0. Introduction

A major goal in the field of analysis of algorithms is to identify important subproblems whose solution is used by many algorithms. One such problem is that of maintaining disjoint sets which are occasionally united. By "maintaining" we mean being able to test membership and to check whether two elements belong to the same set.

In this paper we identify another subproblem that we call the component merging problem. This is a graph problem, where vertices belong to disjoint components. These components are occasionally merged. We want to be able to answer questions about the contracted graph in which each component is contracted to a vertex. We want to do this without actually maintaining the contracted graph, which would be too expensive. Alternatively, we could maintain the original graph; but this results in inspecting many edges both of whose endpoints are in the same component. These are called dead edges or contracted edges. More specifically, we would like to be able to answer the question: what is the minimum cost edge incident to a contracted vertex. Throughout the paper we assume, without loss of generality, that there are no ties, since ties can be broken in some consistent way.

The problem looks at first as a special case of maintaining disjoint sets (the components). However, our insisting on a specific piece of information per component makes the problem harder. All the algorithms mentioned below will also maintain the components as disjoint sets. Consequently, we will know at any moment the component of any vertex.
In Section 1 we define the merging problem, and some of its versions. We then show how to solve the simplest version of the problem in time \( t(m, n) = O(m \log \log d \log n) \), where the graph has \( n \) vertices, \( m \) edges and (density) \( d = \max(m/n, 2) \). The solution is achieved by defining an appropriate data structure. Using the data structure, we obtain an \( O(m \log \log d \log n) \) algorithm. This algorithm is then sped up to \( t(m, n) \) by introducing a new algorithmic tool which we call packets.

We now describe very roughly the notion of packets. A precise description is given in Section 1. Packets are useful whenever an algorithm has \( s \) phases and in every phase we may consider all the edges. Such an algorithm has typically a time bound of \( O(ms) \). We pack a certain number \( p \) of edges into packets which are actually heaps. We then deal only with packet minima. Whenever the packet minimum is dead we purge the packet by repeatedly deleting minima until the packet minimum is alive. The total cost of the purges is \( O(m \log p) \), and except for the first phase which costs \( O(m) \), the other phases cost only \( O(m/p) \). By choosing \( p = s \) we derive a time bound of \( O(m \log s) \).

In Section 2 we sketch how an \( O(t(m, n)) \) algorithm for the minimum spanning tree problem follows immediately from our solution to the simplest version of the merging problem. This time bound is inferior to Fredman and Tarjan's \( O(m \beta(m, n)) \) algorithm [7] which uses F-heaps, where \( \beta(m, n) = \min\{i \mid \log^{(i)} n \leq m/n\} \). We denote this algorithm the FT-algorithm. However, by adding packets to the FT-algorithm we derive an \( O(m \log(\beta(m, n))) \) algorithm.

In Section 3 we sketch how to derive an \( O(n(t(m, n) + n \log n)) \) algorithm for finding a maximum weighted matching in general graphs, using the data structure of Section 1 and packets. This application is not immediate since we are not dealing with the simplest version of the merging problem.

Section 4 contains the most complicated application of our solution to the merging problem: an \( O(t(m, n) + n \log n) \) algorithm for finding an optimum branching (a directed spanning tree). The data structure of Section 1 and packets are used, but are not enough to guarantee the time bound.

In Section 5 we discuss the results and mention some open problems.

The new algorithms for matching and branching improve the best known algorithms for these problems. For each of the two problems there were two algorithms, one best for sparse graphs and one best for dense graphs. The new algorithms have the same time bound as the best algorithm for the sparse graphs (dense graphs) when the graph is very sparse (dense); i.e. when the number of edges is linear (quadratic) in the number of vertices. In all other cases the new algorithms are asymptotically better than both the sparse and dense versions.

Some of our algorithms use F-heaps [7]. An F-heap supports \( m \) operations of insert, findmin, meld, decrease key, and delete, \( n \) of which are insert and \( k \) of which are delete, in time \( O(m + k \log n) \). In our case the elements will always be edges and the keys will be their costs.

1. The Merging Problem

The (component) merging problem is defined on a graph with edge costs \( c_{ij} \). At any point the vertices are partitioned into components. The task is to keep track of the minimum cost edge incident to each component \( C \) (edge \( ij \) is incident to \( C \) if \( i \notin C, j \in C \)). This edge is referred to as the best edge for \( C \). The only operation is merge(\( C_1, C_2 \)), which replaces components \( C_1, C_2 \) by their union. (Initially every vertex is a component.)

There are several versions of the problem. The graph can be directed or undirected. (In the latter case an edge may be incident to two components.) The version is off line if the graph is given as the input and on line if during the algorithm new vertices (and edges) are added.

Assume the (final) graph has \( n \) vertices and \( m \) edges. There are two immediate solutions for all these versions of the problem. The one which contracts each component as soon as it is formed takes \( O(n^2) \). The other that uses a priority queue takes \( O(m \log n) \).

Let \( s_0 = 1, s_i = d, s_{i+1} = s_i^{1.5} = d^{1.5}, i = 1, \ldots \) The rank of a component \( r(C) \) is the integer \( r \) satisfying \( s_r \leq |C| < s_{r+1} \). The maximal rank of any component \( s \) satisfies \( s = O((\log \log d \log n)) \).

In the restricted (component) merging problem all merges are between components of equal rank: The algorithm consists of phases 0,1,2,\ldots. In phase \( r \), only components of rank \( r \) are merged. The following theorem is the basis for most of the results in this paper. It applies to directed and undirected graphs. Let \( t(m, n) = m \log \log d \log n \).

Theorem 1. Restricted merging can be done in time \( O(t(m, n)) \).

Note that for \( m = \Omega(n^{1+\alpha}) \) and any \( \alpha > 0 \), \( t(m, n) = O(m) \).

We first describe phase 0 in the basic algorithm (without packets). Let \( C \) be a component of rank \( r \). A com-
ponent \( C' \neq C \) is adjacent to \( C \) if there is an edge \( ij, i \in C', j \in C \). If \( C' \) is adjacent to \( C \), then the best edge from \( C' \) to \( C \) is the minimum cost edge \( ij, i \in C', j \in C \). Let \( C_i, i = 1, ..., k \) be the components of rank \( r \) adjacent to \( C \), ordered so the cost of their best edge is increasing. The critical set for \( C \) consists of the best edges from \( C_1, ..., C_k \) to \( C \), where \( j \) is the smallest index such that \( \left| C \right| + \sum_{i=1}^{j} \left| C_i \right| \geq s_{r+1} \) (if no such \( j \) exists, \( j = k \)). Note that the critical set for \( C \) is contained in the set of the \( \gamma_r \) best edges of \( C \), where \( \gamma_r = \left\lfloor \frac{s_{r+1}}{s_r} \right\rfloor \). Also note that \( \gamma_0 = \left\lfloor \frac{m}{n} \right\rfloor \) and \( \gamma_r = \left\lfloor \sqrt{s_r} \right\rfloor, r = 1, 2, ... \).

For each component \( C \) the algorithm maintains a set of at most \( 2\gamma_r \) edges that contains the critical set for \( C \), with a distinguished element in the set which is the best edge for \( C \). We call the set \( b-set \) (best set) for \( C \). The reason for maintaining the \( b-set \) for \( C \) is that an edge not in the critical set for \( C \) cannot become the best edge for \( C \) in phase \( r \).

By pruning a set of edges incident to a component \( C \), we mean deleting all edges coming to \( G \) except the best edge \( C \) in phase \( r \), and possibly one residual packet. When \( C \) becomes a regular packet, then the resulting packet has size at least \( p \). Prune \( C \) and, using selection, discard all but the \( p \)-best edges of \( C \) to come from the above procedure. The main difference is that in phase \( r \), for each component \( C \) the algorithm maintains a set of at most \( 2\gamma_r \) edges that contains the critical set for \( C \). These \( b-set \)s are initialized and manipulated as above. We will not mention the combine step.

The purge step repeatedly deletes the packet minimum until the packet is empty or the packet minimum is alive (it is incident to the component). During the purge each edge considered may cause an additional \( O(1) \)-time work which will be specified. The total time for deleting edges from packets is \( O(m \log p) \). The only additional time for purge is \( O(1) \) for each purged packet. In fact, it is \( 0 \) for a packet from which at least one edge was deleted.

In a prune step we are given two sets of packets \( A \) (the new) and \( B \) (the old). \( B \) is clean: its packet minima all come from different components. The prune step adds the packets of \( A \) one at a time. It purges each such a packet \( P \) until \( P \) becomes empty or one of the following two cases occurs: 1. The resulting packet minimum comes from a new component, then \( P \) is added to \( B \); 2. \( B \) has a packet \( Q \) with a packet minimum that comes from the same component. Let \( \{ P', Q' \} = \{ P, Q \} \) with \( P' \) the packet with the smaller minimum. In case \( P' = P \), \( P \) replaces \( Q \) in \( B \). Then the packet minimum of \( Q' \) is deleted and \( Q' \) is purged similarly. The resulting \( B \) is clean.

To execute the pruning efficiently we use an array of length \( n \) to store the set \( B \). The \( i \)-th entry (if it is not \( 0 \)) points to the packet minimum that comes from the \( i \)-th component. Using a linked list we eliminate the need to repeatedly initialize the array. The list contains the used entries of the array and is used to clean the array after the pruning. In all our applications \( B \) will already be represented in the array and thus the time of the prune step is only \( O(1) \). We use the prune step below we will specify the sets \( A \) and \( B \). In case \( B \) is empty we will say that we prune the set \( A \).

We now show how to improve the basic algorithm for the merging problem by using packets. Before the first phase form packets of edges by choosing \( p \equiv \left\lfloor \log \log_2 n \right\rfloor \). In phase \( r \), for each component \( C \) the algorithm maintains a set of at most \( 2\gamma_r \) packets that contains the critical set for \( C \). These \( b-set \)s are initialized and manipulated as above. The main difference is that instead of processing every edge, only packet minima are processed. The packets in a \( b-set \) will always have live minima. However, \( b-set \)s are not necessarily clean, i.e. several packet minima may come from the same component \( C' \neq C \).
We form a b-set from a set of packets as follows. We first prune the set, then we discard all but the $\gamma_r$ packets with the smallest minima. This is done in linear time by using the linear-time selection algorithm [2]. Finally we scan the packet minima to determine the best edge for $C$. The cost of forming a b-set is proportional to the number of packets considered. Note that edges are deleted from packets for the rest of the algorithm, but packets are discarded only for the duration of a phase. They may be used in later phases.

At the start of a phase we consider each component $C$, and form the b-set for $C$ from the packets of $C$. This takes $O(N)$ time, where $N = \lfloor m/p \rfloor + \lfloor n/s \rfloor$ is the bound on the number of packets (the second term bounds the number of residual packets). The total time is $O(sm/p + \sum m/s) = O(m)$. It includes the additional $O(m)$ packing time of the first phase.

In merge($C_1$, $C_2$) we combine the b-sets of $C_1$ and $C_2$ to get the one for $C_1 \cup C_2$ as follows. Let $A$ be the union of the two b-sets. If $|A| > 2\gamma_r$ we form a b-set from $A$. Otherwise, we purge the packets in $A$ and scan the resulting minima to determine the best edge. The time for merge is $O(\gamma_r)$, the number of merges is at most $n/s$, for a total of $O(m)$. Hence, the total time except for deletions from packets is $O(m)$, which implies that the additional time for maintaining disjoint sets is $O(ma(m, n))$ [16, Chapter 2]. (The $O(\log p)$ steps in delete do not spend any time in maintaining the sets.) Thus, the total time is $O(m \log p)$ as claimed in Theorem 1.

Correctness of the algorithm is immediate: the b-set (of packets) of a component $C$ contains the critical set for $C$ because edges in packets that are discarded cannot be in the critical sets later in the same phase and edges deleted from packets can never be later in any critical set. Since packets are pruned and purged, the smallest among the packet minima of the b-set of $C$ is indeed the best edge for $C$.

The above proof indicates a general phenomenon associated with the use of packets: Relevant information can be "buried" in a packet; however that information surfaces as a packet minimum by the time it is needed. In the above algorithm it is conceivable that all packets in the b-set are irrelevant except for one which contains the entire critical set. This "burying" phenomenon occurs in the algorithms of Sections 2-4.

In the next sections we will design algorithms for finding a minimum spanning tree, maximum weighted matching and optimum branching. Each will use some version of the merging problem. The (merging) components will be called in short mc's. We will use packets (of edges) not only for b-sets.

In the next sections we will need another operation which adds a set of packets $A$ to a b-set $B$ of a component $C$. We will know that all the packet minima in $B$ are still alive, so we will not need to scan them. We add $A$ to $B$ as follows. If $|A| > |B| > 2\gamma_r$ we form the new b-set for $C$ from $A \cup B$, and the add step is complete. Otherwise, we purge only the packets in $A$. We then compare the (old) best edge for $C$ with the smallest among the (new) packet minima of $A$ to derive the new best edge for $C$. The time in the first case is $O(\gamma_r)$ and is charged to the discarded packets. This cost adds $O(1)$ per packet per phase and can be ignored. The time in the second case and hence for the add step is $O(|A|)$. If $|A| = 1$ we will say that we add a packet to $B$ (in time $O(1)$).

2. Minimum Spanning Trees

The simplest application of Theorem 1 is for the problem of finding minimum spanning trees. In this case the graph is undirected and the algorithm is off line. Moreover, using an algorithm similar to Boruvka's algorithm (see [16] Chapter 6) we grow the forest in phases. In phase $r$ we only grow trees of rank $r$ by merging them with other trees of rank $r$ or (at most once) with a tree of higher rank. The restricted merging problem is used for choosing the next edge. The mc's are the sets of vertices of the trees in the spanning forest. Consequently, we derive

Corollary 1: A minimum spanning tree can be found in time $O(t(m, n))$.

The algorithm above improves [18] and [4] but not the recent PT-algorithm [7] that uses F-heaps. Our algorithm does not use F-heaps (which have a high constant coefficient) and may be better in practice. We include Corollary 1 mainly because it follows immediately from Theorem 1. The other two applications use Theorem 1 in a less obvious way. In the rest of this section we describe a modification of the FT-algorithm which uses packets. We use the same notation for a tree $T$ and its set of vertices. The best edge for $T$ is defined as for components.

Recall these facts about the FT-algorithm. The algorithm is organized in $\beta(m, n)$ phases. Each phase adds edges to a forest which is contained in the desired minimum spanning tree. At any point in the phase there is one tree $T$ that is being enlarged by adding edges incident to it. There is an F-heap for $T$, whose entries are trees $S$ adjacent to $T$, where the key of $S$ is $\min \{e_{st} | s \in S, t \in T\}$. The smallest entry in the F-heap corresponds to the best
edge for \( T \). The entry is deleted, edge \( st \) is added to the forest, and the F-heap is updated using edges incident to \( S \). Each phase has a parameter \( k \) (\( k = \lceil 2^m/n \rceil \) in the first phase). If an F-heap ever contains more than \( k \) entries, it is discarded, and another tree \( T' \) is chosen for growing. This continues until all trees \( T \) have been grown, at which point the phase ends.

The time for one phase is \( O(m + t \log k) \); here \( t \) is the number of trees at the start of the phase and the second term comes from doing \( t \) F-heap deletes. At the end of a phase each current tree is incident to at least \( k \) edges (those in its F-heap). Thus if \( t' \) is the number of trees at the end of a phase \( t'k \leq 2m \). This inequality allows the next phase to use the value \( k' = 2^k \) for its F-heap parameter \( (t' \log k' \leq 2m) \). This implies the time per phase is \( O(m) \), the number of phases is \( O(\beta(m, n)) \), and the total time is \( O(m\beta(m, n)) \).

Packets are defined as in Section 1, choosing the packet size to be \( p \equiv \beta(m, n) \). At the start of the phase, each tree has a set of packets associated with it and at most one residual packet. We prune the packets of \( T \). As a result the set packets of \( T \) is clean, i.e. all the packets have their minima leading to different trees. We charge each packet \( O(1) \) to account for the prune step. The (regular) packet minima are organized into an F-heap: an entry in the F-heap is a tree \( s \in S \), \( t \in T \). In addition, an array and a list are maintained for a fast membership test in the F-heap.

An edge is added to the forest as follows. Suppose the best edge of \( T \) is \( st \), with \( s \in S \), \( t \in T \). Then \( c_{st} \) is the smaller of the minimum key of the F-heap and packet minimum of the residual packet. Edge \( st \) is added to the forest and the data structure for \( T \) is updated for \( S \cup T \) as follows.

The entry for \( S \) is deleted from the F-heap (if it exists). As in merging components, a combine step might merge the two residual packets; then we do a prune step. The set \( B \) is the set of packets represented in the F-heap. The set \( A \) consists of the packet of \( st \) and the packets of \( S \).

We do the following additional work. When we add a new packet to \( B \) we also perform insert in the F-heap. When an element of \( B \) is replaced by one with a smaller key, we perform a decrease key in the F-heap. The cost of the pruning is proportional to the number of packets of \( S \) because the array and list for \( B \) are available and the cost of the packet of \( st \) is 0 due to the deletion of \( st \). We charge \( O(1) \) to each packet of \( S \).

The procedure is correct because only contracted edges are deleted, so the smallest cost edge in the data structure is the best edge for \( T \). It is clear that the algorithm chooses the smallest cost edge in the data structure.

The time is estimated as follows. In each phase, charge each packet \( O(1) \) (for pruning in initialization and in update), each edge deleted from a packet \( O(\log p) \), and each edge added to the forest (possibly deleted from the F-heap) \( O(\log k) \). The total charge to packets is \( O(m) \) since the number of packets in phase \( r \) is at most \( m/p+n_r \). Here \( n_r \) is the number of trees in the \( r \)-th phase which bounds the number of residual packets, and since \( n_r < n/2^r \), \( \sum n_r = O(n) \). The total charge to edges is \( O(m \log p) \) (the dominating term). Finally, to estimate the total charge to forest edges observe that at the end of each phase, each tree has \( k \) (regular) packets represented in its F-heap. Thus \( t'k \leq 2m \) or \( t' \log k' = O(m/p) \). This implies that in each phase after the first, forest edges are charged \( O(m/p) \). In the first phase \( k = \lceil 2^{m/n} \rceil \) and the charge is \( O(m) \). So the total charge is \( O(m) \). The justification for ignoring the cost of maintaining disjoint sets is as in Theorem 1.

**Theorem 2.** A minimum spanning tree can be found in \( O(m \log \beta(m, n)) \).

### 3. Maximum Weighted Matching in General Graphs

This problem is solved by various implementations of Edmonds' algorithm [5]. The best algorithm for dense graphs has \( O(n^3) \) time bound [8], [14]. The best algorithm for sparse graphs is \( O(nm \log n) \) [11]. The obvious question that arises is whether there is an algorithm that is asymptotically faster than both. Recently, the first author designed an algorithm of time complexity \( O(mn^{3/4}(\log n)^{1/2} \log N) \) if \( m \geq n\beta(m, n) \) (and a similar expression otherwise), where \( N \) is the maximal weight and the weights are all integers [9]. In this paper we are concerned with algorithms that do not depend on the type or the size of the weights.

Edmonds' algorithm has \( n \) stages. Hence, unless a new approach is invented, the best we can hope for is \( O(nm) \). Tarjan [16, p. 123] conjectured the existence of an \( O(nm \log(n^2/m)) \) algorithm. The latter bound equals \( O(nm) \) only for \( m = \Omega(n^2) \), and does not dominate \( O(nm \log n) \) when \( m = O(n^2) \), \( \beta < 2 \). Our second application of Theorem 1 is:

**Theorem 3.** Maximum weighted matching can be solved in time \( O(n(t(m, n) + n \log n)) \).

This time equals \( O(nm) \) for \( m = \Omega(n^{1+\alpha}) \) and any \( \alpha > 0 \). It dominates \( O(n^2) \) unless \( m = \Omega(n^2) \) and it dominates \( O(nm \log n) \) unless \( m = O(n) \). (The \( O(n^2) \) and
We assume some familiarity with Edmonds’ algorithm as described in [11]. In each stage in Edmonds’ algorithm vertices are labelled by \( S \) and \( T \) (outer and inner) and occasionally blossoms (odd sets of vertices) are identified, contracted and labelled by \( S \). The bottleneck in each stage is finding in the contracted graph (in the original graph) an edge between two \( S \) labelled vertices (between vertices in two different \( S \)-blossoms) of smallest cost (= slack). This leads to the following version of the merging problem, where the components are the \( S \)-blossoms. The graph is undirected; the version is online because new vertices are occasionally labeled by \( S \); and we have unrestricted merging because the choice of which components to merge is not under our control. Unlike in the statement of the merging problem we need to maintain only one minimum cost edge among all edges incident to any of the components.

We now show how to solve this version of the merging problem in time \( O((m,n) + n \log n) \). This will prove Theorem 3. Each edge \( ij \) will be represented twice as the directed edges \( ij \) and \( ji \). An edge \( ij \) is active if \( i \in C' \neq C \), \( r(C') \geq r(C) \). We use packets of size \( p \) as in Section 1. A packet is active if its packet minimum is active. For each component the algorithm maintains a b-set of (up to) \( 2p \) active regular packets, possibly one residual packet, and the best active edge for \( C \), the smallest packet minimum in the b-set. Inactive packets are not discarded because they may become active.

We also maintain two F-heaps. The first contains the best active edges of all the components. The second contains one packet minimum for each component. At the start of the phase this packet is the residual packet. The algorithm adds edges to packets. This may cause a residual packet to become a regular packet and a generation of a new residual packet. The packet minimum in the second heap is replaced only by a smaller residual packet minimum. Since we delete from the F-heaps only when we merge components, the time to maintain them is \( O(m+n \log n) \). The smaller of the minimal elements in the F-heaps is the edge needed for the matching algorithm.

Consider the time that a new vertex \( i \) and new edges \( ij \) (and \( ji \)) for different \( j \)'s are added. (The graph is given on line.) All the edges \( ji \) are active and are packed into packets (one may be residual) associated with \( i \). Then we form the b-set for component \( \{ i \} \). Each edge \( ij \) is inserted to the residual packet of the component \( C \) of \( j \). (If there is none it is added as a new residual packet \( P \).) This may result in a decrease key in the second F-heap. If, as a result, \( P \) has \( p \) edges and is active, we add it to the b-set of \( C \). The total time for packing is \( O(m) \) and is \( O(m \log p) \) for adding edges \( ij \) (using regular heaps).

Ignoring for the time being merges of components of different ranks, we actually have a restricted merging problem, where the phases are intermixed: Phase \( r \) may continue after phase \( r+1 \) has started: components of rank \( r \) “participate” in phase \( r \).

Assume we merge \( C_1 \) of rank \( r_1 \), \( l = 1,2 \), to form \( C \) of rank \( r \). Since we are interested mainly in active edges we slightly augment the purge step. After a packet is purged we check its minimum \( ij \), \( i \in C_3 \), \( j \in C \). If \( ji \) is a packet minimum of a regular packet \( P \) that was not discarded, \( r(C_3) \leq r \) (i.e. \( ji \) active) and \( P \) is not in \( B \), the b-set of \( C_3 \), we add \( P \) to \( B \). (We can use the add step since packet minima in \( B \) are still active.) The cost of the add step, including a possible decrease key in the F-heap of best edges, is \( O(1) \) and does not change the cost of the purge. Note that the merge cannot cause active packets belonging to other components to become inactive. We also use a slightly modified form step. After the pruning we consider only active packets for choosing the \( \gamma_r \) best active packets.

Details of the merge are as follows. There are three cases: 1) \( r_1 = r_2 \), \( r_1 < r_2 < r \) (\( = r_2 + 1 \)), and 3) \( r_1 = r_2 = r \). The first case is handled as in Section 1. In the second, we apply the modified form step to the set of packets of \( C \) to form the b-set for \( C \). In the third case we add the set of packets of \( C_1 \) to the b-set of \( C_2 \). (Note that packet minima in the latter are still active.)

Correctness follows as in Theorem 1. Each edge in the active critical set for a component \( C \) is either in a packet in the b-set for \( C \) or in an inactive packet of \( C \) that has not been discarded or in the residual packet of \( C \). The time analysis of Theorem 1 still holds. Case 1 above corresponds to the merges of equal rank components in Theorem 1. Case 2 corresponds to initializing \( C \) for phase \( r \). Case 3 which takes time proportional to the number of packets of \( C_1 \) (excluding costs accounted for separately) corresponds to initializing \( C_1 \) for phase \( r \).

The bottleneck handled above is not the only one in Edmonds’ algorithm. The second bottleneck is the problem of expanding T-blossoms (which does not affect the problem above). This problem can be modeled by the following problem. Given is a collection of trees. Each interior node has at least two children. Each leaf \( i \) has a cost \( c_i \). The cost of a tree is its minimum leaf cost. Some of the trees are active, some inactive. The object is to keep track of the active tree with smallest cost subject to a sequence of the following types of operations:

- activate(T) (inactive(T)) - Make T active (inactive).
decreasecost(i, c) - Decrease the cost of leaf i to the new value c.

split(T) - Delete the root B of T. This creates new trees rooted at the former children of B.

change(c) - Add c to the costs of all the leaves of all the active trees.

The active (inactive) trees are the structure trees of the free (T-) blossoms, and the cost of a leaf is the minimum slack from an S-vertex to the vertex represented by that leaf (see [11]).

There are two solutions to this problem. Both maintain the cost of each tree. The one in [12] which proves a new property of F-heaps takes $O(m + n \log n)$. The one in [9] is more complicated but has a better time bound of $O(m + n \log(m, n))$ for maintaining the cost of each tree. In the context of the present algorithm, however, the latter needs an additional F-heap to maintain the minimum cost tree. Consequently, it also costs $O(m + n \log n)$.

Since this second bottleneck is not related to the merging problem we do not describe the solutions here. The matching algorithm uses two other priority queues (see [11]). But they require only $O(n \log n)$ time.

4. Optimum Branching

The optimum branching problem is formulated as follows. Given a directed graph with a distinguished vertex 0, the root vertex, and edge costs $c_{ij}$. Find a minimum cost out-tree rooted at 0. (Other formulations such as finding a minimum cost spanning tree, i.e., out-forest, are easily reduced to this version.)

This problem is solved by implementing Edmonds' and Karp's algorithm [6], [13]. The best algorithm for dense graphs is $O(n^2)$ time, and for sparse graphs is $O(mn \log n)$ [15]. Tarjan [15 p. 34] asked whether an $O(m \log n)$ algorithm exists. Our third application of Theorem 1 answers the question affirmatively.

**Theorem 4.** Optimum branching can be solved in time $O((m, n) + n \log n)$.

This bound is linear for $m = \Omega(n^{1+\alpha})$ for any $\alpha > 0$ and is $o(m \log n)$ unless $m = O(n)$.

The efficient algorithm for branching ([6],[13]) constructs a subgraph $H$ of the given graph that contains an optimum branching. In fact, the branching can be extracted from $H$ in $O(n)$ time [3]. At any point in the algorithm, $H$ is partitioned into weak components (wc's). These are the connected components of (the undirected version of) $H$. Each wc (considered as a directed graph) consists of strongly connected components (sc's); the edges of the wc that join sc's form an out-tree when the sc's are contracted. The sc that is the root of this out-tree is a root component (rc).

The algorithm works by repeatedly choosing an rc $R$ (the choice for $R$ is arbitrary except $0 \notin R$), and adding to $H$ the minimum cost edge $ij$ incident to $R$ ($j \in R$, $i \notin R$). If $i$ and $j$ are in different wc's, this is called a grow step, and $ij$ is a grow edge. The effect is to combine the two wc's into one, and $R$ is no longer an rc. If $i$ and $j$ are in the same wc but different sc's, this is called a reduce step and $ij$ is a reduce edge. Edge $ij$ creates a cycle of sc's, $c_1, c_2, c_3, ..., c_k, S_k$,

where $c_1 = ij$, $i \in S_k$, $j \in S_1 = R$ and edge $c_k$ of $H$ is directed from $S_{i-1}$ to $S_i$, $i = 2, ..., k$. These $k$ sc's form the new rc; an edge directed to $S_i$ has its cost decreased by $c_i$. Observe that if $i$ and $j$ are in the same sc, no step is done by the algorithm. These contracted edges are the ones we wish to avoid processing.

Modelling this algorithm as a merging problem where an mc is an sc leads to the case of an off-line unrestricted merging problem on directed graphs. This is one of the hardest versions of the merging problem. An efficient solution even to the on-line unrestricted merging problem on directed graphs can be obtained by using $\gamma$-systems that were introduced in [12]. A more efficient branching algorithm can be obtained by using a definition of mc that is more amenable to restricted merging. Toward this end we organize the branching algorithm into phases, as follows. The definition of rank and phase are as in Section 1.

At the start of the r-th phase, each wc (except the wc containing 0) has rank at least $r$. The basic structure of a phase is to repeatedly choose a wc $W$ (not containing 0) of rank exactly $r$, and do grow and reduce steps (from the current rc of $W$) until the grow steps increase the rank to $r + 1$ or more (or a grow step joins $W$ to the wc containing 0). Note that this policy amounts to a depth-first search from $W$; this allows the relevant information to be organized efficiently.

This organization of the branching algorithm is modelled as a merging problem by using the following definition of mc. At the start of a phase, each wc is an mc. Thereafter two of these "initial" wc's, $W_1$ and $W_2$, are in the same mc if some (single) reduce step contracting vertices from $W_1$ and $W_2$ has been done (they must be currently in the same wc).

Consider Figure 1. At the start of the phase there are three wc's $A, B, C$ with rc's $a, b, c$ resp. Each wc is an
Figure 1.

mc at this point. Two grow edges 1 and 2 are added. The three wc's become one wc but the three mc's do not change. A reduce edge 3 is added. The only change is that sc's d, e are merged with a. The rc of the unique wc contains a, d and e. Reduce edge 4 is now added and mc's A and B are merged. As a result there is one wc \((A \cup B \cup C)\) two mc's \((A \cup B, C)\), and the rc consists of all the sc's except for c and i.

We maintain the wc's, mc's and sc's as disjoint sets. When we refer below to an edge ij, where i and j are sc's we will mean that the endpoints of the edge are in sc's i and j.

At any time, we call an sc active if it is an rc or it was an rc during the phase. So, considering the wc as a tree of sc's, each mc is a subtree, and the active sc is the root of the subtree. The key point is that throughout a phase, whenever a reduce step for a wc W causes an mc \(C_0\) to be merged, the active sc of \(C_0\) gets contracted into the new rc of W. This results from doing the search depth-first. In Figure 1, sc c remains active after it stops being an rc. If mc C is merged later into \(A \cup B\), then c will become part of the rc.

The precise formulation of the merging problem is as follows. The graph is directed, the problem is off line, and restricted (all components have the same rank). The vertices of each component are partitioned into sets called sc's, at most one of which is the active sc and the others inactive. For an inactive sc i, the best edge for i is the minimum cost edge \(ij\), where \(j\) is active in i's mc. The cost \(c_i\) of \(i\) is the cost of the best edge for \(i\).

The task is to maintain two pieces of information for each mc C:

(i) the best edge, i.e., the minimum cost edge \(ij\) incident to \(C\), where \(j\) is the active sc of \(C\);

(ii) the best sc, i.e., the minimum cost inactive sc of \(C\), and its best edge.

At the start of the phase all sc's are inactive. Besides merge, the operations are activate(i) and decreasecost(i, e). The operation activate(i), i an inactive sc, combines i with the active sc of i's mc, giving a new active sc. Similarly, merge(C1, C2) combines the active sc's of C1 into one. The operation decreasecost(i, e), \(i\) an inactive sc, \(e = ij\), \(j\) active in i's mc, is as follows: if \(c_i < c_j\) then \([c_i := c_j; \text{best edge of } i := e]\).

Defining the problem in terms of sc's, rather than vertices, is convenient because, as will be seen, the sc's of the merging data structure correspond exactly to the sc's of the branching algorithm. The techniques presented here can be used to efficiently solve a more general version of the problem that is defined in terms of vertices rather than sc's.

The branching algorithm uses the merging data structure as follows. It maintains the subgraph \(H\) of the Edmonds-Karp algorithm. As indicated above, in each phase the first time an sc enters the rc of its wc, it gets activated. (This rule implies that when a phase is initialized all rc's get activated.) Note that the best edge for an inactive sc i is the best reduce edge for i if i's mc actually contains the rc of its wc.

To do a step of the branching algorithm from an rc \(R\) in mc C, examine the best edge and best sc for \(C\), do a grow or reduce step for the one with smaller cost. In a grow step, the only action is to add the edge to \(H\); the merging component data structure is unchanged. In a reduce step with the cycle of sc's (1) the following is done.

Reduce step algorithm

1. The reduce edge is added to \(H\). The cost of an edge directed to \(S_i\) is decreased by \(c_{ij}\).

2. If any sc \(S_i, i > 1\), is the active sc of an mc \(D\), merge(C, D) is done in the merging data structure. (There may be no merges.) Let \(C'\) be the mc resulting from all these merges.

3. If any sc \(S_i\) is an inactive sc i, activate(i) is done in the merging data structure. (The sc's i come from the
4. For every edge $ij$ where $i, j$ are sc's of $C'$, $i$ inactive and $j$ active, do decreasecost$(i, ij)$ (i.e., check whether $ij$ should become the best edge for $i$).

Observe that a grow step in the branching algorithm results from choosing a best edge from the merging data structure (edges 1,2 in Figure 1). A reduce step results from a best sc or a best edge. A best sc gives a reduce step that only activates sc’s, i.e., it stays within one mc (edge 3 in Figure 1). A best edge gives a reduce step that merges mc’s and may activate sc’s, i.e., it goes across mc’s (edge 4 in Figure 1).

At this point three things remain to be done to complete the description of the algorithm:

(i) indicate how edge costs are decreased in Step 1;
(ii) solve the merging problem (for Steps 2-4);
(iii) indicate how to generate the decreasecost instructions in Step 4.

(i) Decreasing edge costs

The problem of maintaining edge costs can be stated as follows: Each vertex $i$ (of the original graph) has an offset $d_i = \sum_{s \in S_i} c_{el}$, where the summation is over all sc’s $S_i$ (over the course of the entire algorithm) that contain $i$ and get contracted in a reduce step ($c_{el}$ is from the cycle (1)). At any point in the algorithm the cost of an edge $ji$ is $c_{ji} - d_i$. Hence the problem is to maintain the offsets $d_i$.

Maintaining offsets is done using set merging techniques [1]. A forest is maintained. The vertices are all sc’s ever generated, and the children of an sc formed in a reduce step are the constituent sc’s $S_1, \ldots, S_k$. Each non-root of the forest $S_i$ has the associated cost $c_{ci}$; the offset for $d_i$ is the sum of the costs on $i$’s path to the root. This reduces the problem to the problem of maintaining disjoint sets (computing the offset is accompanied by path compression). Note that every time the algorithm references the cost of an edge, it calculates the offset.

The data structures presented in the next two subsections use packets of edges. Packets are defined as in Section 1 with one modification: all edges in a packet are incident to a common sc (rather than a common mc as in Section 1).

(ii) The merging data structure

This data structure has two parts - one for mc’s, the other for sc’s. We begin with mc’s. We use packets. The packing in the first phase and the initialization of each phase is as in Section 1 except that we use a slightly different purge step defined below. We use here a slightly different definition of critical set and of b-set by restricting attention only to edges $ij$, $j$ active.

The algorithm for merge($C_1, C_2$) is similar to Section 1, the b-set for $C_1$ and $C_2$ are combined to form the b-set for $C_1 \cup C_2$. The only difference is that vertex offsets (see (i) above) are used in computing the edge costs.

The operation activate($i$) works as follows. The packets of $i$ are added to the b-set of the component $C'$ of $i$. (Note that all packet minima in this b-set are alive.) The cost of the add step and thus of activate($i$) is proportional to the number of packets of $i$. Note that during a phase an sc is activated at most once, and hence the cost of activate can be ignored.

There is another small difference with the proof of Theorem 1. The number of residual packets per phase may be up to $n$. But the number of residual packets that are considered (and not combined) is bounded by the number of components, and the estimates of Section 1 still apply. Consequently, the time spent on this data structure is $O(t(r \log n))$.

Now we describe the sc data structure. Each mc has an F-heap with an entry for each inactive sc $i$, which is its best edge. The key for $i$’s entry is its cost $c_i$. The best sc is the F-heap minimum. The operations activate, decreasecost, and merge are the F-heap operations delete, decrease key and meld, respectively. The time in this data structure for $n$ activates, $n$ merges and $m$ decreasecosts is $O(m + n \log n)$.

(iii) The data structure for generating decreasecosts

In Step 4 of the reduce step algorithm, decreasecost instructions are generated for new reduce edges (that do not go across mc’s). A system of sets (linear lists) is maintained so that a set containing exactly these edges is available. This avoids scanning useless edges. The details are as follows.

Each inactive sc $i$ has a set of packet minima: The active set (a-set) for $i$ is $\{ij \mid ij$ is a packet minimum, $i$ and $j$ are in the same mc and $j$ is active $\}$. The algorithm maintains an sc cost $c_i$ as $\min\{c_{ij} \mid ij$ in $i$’s a-set $\}$. Observe that this is not necessarily the correct cost for $i$ (alluded to above), $\min\{c_{ij} \mid ij$ in the same mc, $j$ active $\}$. This is due to the fact that the true minimum cost edge for $i$ may be “buried” in a packet (see Section 1). However we shall see that the algorithm still computes the correct cost for $i$, when necessary. The a-sets are manipulated together with the b-sets. The mechanism for doing this is the following purge step for a packet (mentioned above). We only purge packets associated with an active sc; i.e., all $j$’s below are in the same active sc. The purge deletes
edges with both endpoints in this set.

**Purge Step:** Repeatedly delete the packet minimum until it is an edge $ij$ satisfying one of the following conditions, in which case the corresponding action is done:

- (i) $i$ and $j$ are in different $mc$'s - add the packet to the $b$-set of $j$'s $mc$,
- (ii) $i$ and $j$ are in the same $mc$, $i$ is inactive - add the packet to $i$'s $a$-set, then do $\text{decreasecost}(i, ij)$.

With this purge step, the $\text{decreasecost}$s of Step 4 of the reduce algorithm are actually done in Step 2 and 3 as follows. When $b$-sets are combined (in merge of Step 2 or in activate of Step 3) the above purge step is used possibly generating $\text{decreasecost}$s. When activate($i$) is done for a previously inactive $sc$, it is an edge $ij$.

This concludes the description of the branching algorithm. Correctness follows as in Theorem 1. The additional data structures do not affect the timing analysis: a packet can move back and forth between an $a$-set and a $b$-set and a $\text{decreasecost}$ can be done. This processing takes $O(1)$ time per packet. The time analysis is similar to that of Theorem 1. The time for deletions from packets is $O(m \log p)$ and everything else related to packets takes $O(m)$ time. The time for the F-heap for finding the best vertex is $O(n \log n)$, hence, the total time $T = O(m \log p + n \log n)$, except for computation of offsets and maintaining disjoint sets. But the latter costs $O((T, n)) = O(T)$ since $T \geq n \log n$. Consequently, the time bound of Theorem 4 follows.

Note that the $O(n \log n)$ term in the time bound is unavoidable if the Edmonds-Karp algorithm is used: Given $n$ positive numbers $x_1, \ldots, x_n$ let $M$ be the largest number. Consider the graph $G = (V, E), V = \{0, \ldots, n, n + 1\}$, $E = \{(0, n + 1), (n + 1, i), (i, n + 1), i = 1, \ldots, n\}$, with $c_{0, n+1} = M, c_{i, n+1} = 0, c_{i, n+1} = x_i$. Then no matter what arbitrary choices are made in the Edmonds-Karp algorithm, the sequence of grow/reduce steps gives the edges $x_i$ in sorted order.

This algorithm extends to the problem of finding an optimum root-constrained directed spanning tree [10]. In fact, it can be shown that such a tree can be found in $O(b(m, n) + n \log n)$ time, where $b(m, n)$ is the time to find optimum branching. Hence the bound of Theorem 4 applies.

At the time of this writing R.E. Tarjan discovered a branching algorithm that uses $O(m + n \log n)$ time [17]. It is based on the depth-first ordering and special properties of F-heaps.

5. Conclusion

This work raises the following questions:

1. Are there other algorithms that use versions of the merging problem?
2. Are there other algorithms that can be sped up by using packets?
3. Is there a simple branching theorem? Such a theorem will state the conditions under which a packet speed up is possible. It will enable the algorithm designer to devise a basic algorithm and obtain the speed up automatically. At the moment, we can state a cumbersome theorem which includes all applications in this paper.
4. Is it possible to prove an $O((t(m, n) + n \log n)$ bound for the unrestricted version of the merging problem for directed graphs?
5. Can we do better: Can we solve the restricted merging problem, minimum spanning tree, optimum branching or one stage of weighted matching in $O(m/a(m, n))$ (or even $O(m)$)?

In [7], using F-heaps, an $O(m + n \log n)$ algorithm for maximum weighted matching in bipartite graphs is described. A similar bound is conjectured for the general case. Theorem 3 is off by a factor of $\log \log \log m/n$. This factor seems too strange to be true.

References


