I/O-Efficient Batched Union-Find and Its Applications to Terrain Analysis*

Pankaj K. Agarwal[1] Lars Arge[2,1] Ke Yi[1]

[1]Department of Computer Science, Duke University, Durham, NC 27708, USA.
{pankaj,large,yike}@cs.duke.edu

[2]Department of Computer Science, University of Aarhus, Aarhus, Denmark.
large@daimi.au.dk

Abstract

Despite extensive study over the last four decades and numerous applications, no I/O-efficient algorithm is known for the union-find problem. In this paper we present an I/O-efficient algorithm for the batched (off-line) version of the union-find problem. Given any sequence of N mixed union and find operations, where each union operation joins two distinct sets, our algorithm uses $O(\text{SORT}(N)) = O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os, where M is the memory size and B is the disk block size. This bound is asymptotically optimal in the worst case. If there are union operations that join a set with itself, our algorithm uses $O(\text{SORT}(N) + \text{MST}(N))$ I/Os, where $\text{MST}(N)$ is the number of I/Os needed to compute the minimum spanning tree of a graph with N edges. We also describe a simple and practical $O(\text{SORT}(N)\log(\frac{N}{M}))$-I/O algorithm, which we have implemented.

The main motivation for our study of the union-find problem arises from problems in terrain analysis. A terrain can be abstracted as a height function defined over $\mathbb{R}^2$, and many problems that deal with such functions require a union-find data structure. With the emergence of modern mapping technologies, huge amount of data is being generated that is too large to fit in memory, thus I/O-efficient algorithms are needed to process this data efficiently. In this paper, we study two terrain analysis problems that benefit from a union-find data structure: (i) computing topological persistence and (ii) constructing the contour tree. We give the first $O(\text{SORT}(N))$-I/O algorithms for these two problems, assuming that the input terrain is represented as a triangular mesh with N vertices.

Finally, we report some preliminary experimental results, showing that our algorithms give order-of-magnitude improvement over previous methods on large data sets that do not fit in memory.

1 Introduction

In the *union-find* problem, we want to maintain a partition of a set $U = \{x_1, x_2, \ldots\}$ (the universe) and a *representative* element of each set in this partition under a sequence Σ of $\text{UNION}(x_i, x_j)$ and $\text{FIND}(x_i)$ operations: $\text{UNION}(x_i, x_j)$ joins the set containing x_i and the set containing x_j, and provides a new representative element for the new joined set; $\text{FIND}(x_i)$ returns the representative element of the set containing x_i.

*Work on this paper is supported by ARO grant W911NF-04-1-0278. P.A. and K.Y. are also supported by NSF under grants CCR-00-86013, EIA-01-31905, CCR-02-04118, and DEB-04-25465, by ARO grant DAAD19-03-1-0352, and by a grant from the U.S.–Israel Binational Science Foundation. L.A. is also supported by an Ole Rømer Scholarship from the Danish National Science Research Council.

In the *on-line* version of the problem, Σ is given one operation at a time, whereas in the *batched* (or *off-line*) version, the entire sequence Σ is known in advance. The union-find problem is a fundamental algorithmic problem because of its applications in numerous problems across different domains, from programming languages to graph and geometric algorithms, and from computational topology to computational biology. In many of these algorithms only the batched version of the problem is required. See [12, 14, 18, 22] for a sample of applications of the batched union-find problem.

The main motivation for our study of the union-find problem arises from terrain modeling and analysis. A terrain can be abstracted as a height function defined over $\mathbb{R}^2$, and there is a rich literature on the study of such functions. We are interested in two broad problems in terrain analysis, namely flow and contour line analysis. A key step in terrain flow analysis is to modify the height function so that "small" depressions on the terrain (sinks) disappear. We use the notion of *topological persistence*, introduced in [18], to address this problem. In contour line analysis, the key notion is the *contour tree* [12, 34, 39]. Most existing topological persistence and contour tree algorithms rely on efficient data structures for the batched union-find problem.

With the emergence of high-resolution terrain mapping technologies, huge amount of data is being generated that is too large to fit in memory and has to reside on disks. Existing algorithms cannot handle such massive data sets, mainly because they optimize CPU running time while optimizing disk access is much more important. Motivated by these factors we consider the batched union-find problem in the *I/O-model* [4] (also known as the *external memory model*). We develop I/O-efficient algorithms for the problem and use them to develop I/O-efficient topological persistence and contour tree algorithms.

Related results. In the I/O model, the machine consists of an infinite-size external memory (disk) and a main memory of size M. A block of B consecutive elements can be transferred between main memory and disk in one *I/O operation* (or simply *I/O*). Computation can only occur on elements in main memory, and the complexity of an algorithm is measured in terms of the number of I/Os it uses to solve a problem. Many fundamental problems have been solved in the I/O model. For example, sorting N elements takes $\textsc{sort}(N) = \Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os, and permuting N elements takes $\textsc{perm}(N) = \Theta(\min\{N, \textsc{sort}(N)\})$ I/Os. Please refer to the surveys by Vitter [40] and Arge [5] for other results. Most relevant to this paper, many planar (batched) geometric problems can be solved in $O(\textsc{sort}(N))$ I/Os using the so-called distribution sweeping technique [13], and most basic problems on trees can be solved in $O(\textsc{perm}(N))$ I/Os [13, 41].

The study of the union-find problem (in internal memory computation models) started early in the sixties [24] and continues until today. After a few initial results, Tarjan [35] proved that an on-line sequence of N union and find operations can be performed in time $O(N\alpha(N))$, where $\alpha(N)$ is the inverse Ackermann function. This bound is tight in the worst case [21, 36]. A simpler proof of the upper bound and various generalizations have been proposed in [33, 37]. In the pointer-machine model the batched version of the problem also has a lower bound of $\Omega(N\alpha(N))$ [36], but it can be solved in linear time in the RAM model [22]. However, despite of its importance, no I/O-efficient algorithms have been developed for the union-find problem. Note that the naive use of the RAM algorithms in the I/O model results in $O(N\alpha(N))$ and $O(N)$ I/O-algorithms, in the on-line and batched case, respectively. For realistic values of N, B, and M, $\textsc{sort}(N) \ll N$, and the difference in running time between an algorithm performing N I/Os and one performing $\textsc{sort}(N)$ I/Os can be very significant.

Topological persistence (see Section 4 and [17, 18]) for its definition) of a height function is a measure of its topological attributes. Over the last few years, it has been successfully applied to a variety of problems, including topological simplification [17, 18], identifying features on a surface [3], and removing topological noises [11]; see also [42]. Efficient algorithms for computing persistence in internal memory are developed in [17, 18, 43]. The efficient algorithms for computing the topological persistence of a two-dimensional height function rely on a union-find data structure, so no I/O-efficient algorithm is known for this problem.

Contour trees are widely used to represent the topological changes in the contours of a height function (see Section 5 and [12, 34, 39] for a formal definition). Van Kreveld et al. [39] gave an $O(N \log N)$-time algorithm for constructing the contour tree for a piecewise-linear height function on $\mathbb{R}^2$ represented by a mesh with N vertices. It was later extended to 3D by Tarasov and Vyalyi [34], and to arbitrary dimensions by Carr et al. [12]. The last two algorithms rely on a union-find data structure. No I/O-efficient algorithm is known for computing contour trees even in two dimensions.

Our results. Our main result is the first I/O-efficient algorithms for the batched union-find problem. In Section 2, we present an algorithm that uses $O(\text{SORT}(N))$ I/Os on a sequence of N union and find operations, provided that none of the union operations is *redundant*, that is, for each $\text{UNION}(x_i, x_j)$, x_i and x_j are in different sets. This is optimal in the worst case since there is a simple $O(N/B)$-I/O reduction from permutation[1]. An interesting feature of our algorithm is that it reduces the problem to two geometric problems. If redundant union operations are allowed, we describe an algorithm that uses $O(\text{SORT}(N)+\text{MST}(N))$ I/Os, where $\text{MST}(N)$ is the number of I/Os needed to compute the minimum spanning tree of a graph with N edges. Currently the best bound for $\text{MST}(N)$ is $O(\text{SORT}(N) \log \log B)$ I/Os [6] by a deterministic algorithm or expected $O(\text{SORT}(N))$ I/Os by a randomized algorithm [13].

Using our union-find algorithm we develop the first I/O-efficient algorithms for topological persistence and contour trees. In Section 4, we describe an $O(\text{SORT}(N))$-I/O algorithm for computing topological persistence of a terrain represented as a triangular mesh with N vertices. The algorithm is obtained by simply plugging our union-find algorithm to the previous internal memory algorithm of [17, 18]. In Section 5, we describe an $O(\text{SORT}(N))$-I/O algorithm for computing the contour tree of a terrain represented as a triangular mesh with N vertices. For this problem it is not enough to just apply our union-find algorithm to the previous internal memory algorithms. We prove a property of contour trees (cf. Lemma 6), which is interesting in its own right. Using this property, we reduce the problem to a geometric problem and design an I/O-efficient algorithm for this problem. As in [12], our algorithm extends to higher dimensions.

While theoretically I/O-efficient, our $O(\text{SORT}(N))$ and $O(\text{SORT}(N) + \text{MST}(N))$-I/O algorithms for the batched union-find problem are probably too complicated to be of practical interests, therefore in Section 3 we present a simple divide-and-conquer algorithm that requires $O(\text{SORT}(N) \log(\frac{N}{M}))$ I/Os. It does not have to invoke an MST algorithm to handle redundant union operations. We have implemented this algorithm, and in Section 6 we present the results of a few preliminary experiments with the algorithm when used in some of its applications. These experiments show that our algorithm gives order-of-magnitude improvement over previous methods on large data sets. We also apply our techniques to the so-called *flooding* problem, a key step in flow analysis on terrains.

2 I/O-Optimal Batched Union-Find

Let $U = \{x_1, x_2, \ldots\}$ be the universe of elements, and let $\Sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_N \rangle$ be a sequence of union and find operations. Let the *time stamp* of the t-th operation σ_t be t. Assume without loss of generality that each element in U has been used in at least one operation of Σ. In this section we describe an I/O-efficient algorithm for the batched union-find problem, assuming that all union operations in Σ are non-redundant. At the end of the section we will discuss how to handle redundant union operations.

Our algorithm consists of two stages. In the first stage, we compute a total ordering on the elements of U so that as Σ is performed, each set consists of a contiguous sequence of elements in this ordering and

[1]If $N < \text{SORT}(N)$, which is true only for extremely large inputs, there is a gap between $O(\text{SORT}(N))$ and $\Omega(\text{PERM}(N))$. However, in this case we can obtain an $O(\text{PERM}(N)) = O(N)$-I/O algorithm by simply using the $O(N)$ RAM algorithm of [22].

each union operation joins two adjacent sets. More precisely, if sets A and B are joined together by any union operation in Σ, then there exist some $x_i \in A$ and $x_j \in B$ such that x_i and x_j are adjacent in this ordering. We call this special case the *interval union-find* problem. In the second stage, we solve an instance of the batched interval union-find problem. We formulate each of these two subproblems as a geometric problem and present an $O(\text{SORT}(N))$-I/O algorithm for each of them. We also note that the assumption on non-redundant union operations is needed only for the first stage.

2.1 From union-find to interval union-find

Let the *union graph* $G(\Sigma)$ be a weighted undirected graph whose vertices are the elements of U. There is an edge e between x_i and x_j with *weight* $\omega(e) = t$ for each $\text{UNION}(x_i, x_j)$ operation with time stamp t. If all union operations are non-redundant, $G(\Sigma)$ is a forest. For simplicity we just assume it is a tree; if not, we connect them by edges of weight ∞. This can be done using $O(\text{SORT}(N))$ I/Os by computing the connected component of th forest $G(\Sigma)$ [13]. We pick an arbitrary node of $G(\Sigma)$ as its root and call this rooted tree the *union tree*, denoted by T. As a convention, when we refer to an edge (u, v) in a rooted tree, u is always the parent of v.

A union tree T' on the same set of nodes as T is *equivalent* to T if for any t, the connected components of the forest formed by the edges of T and T' with weight at most t are the same. In order to reduce the union-find problem to the interval union-find problem, we first transform T into an equivalent union tree T', with the property that the weights along any leaf-to-root path are increasing. Then we show how a certain in-order traversal of T' defines an ordering of the elements of U that results in Σ being an interval union-find instance.

Transforming T to T'. We transform T into T' by repeatedly applying the following "short-cut" operation, which simulates the *path compression* technique: For any $w \in T$, let v be w's parent and u be v's parent. If $\omega(u, v) < \omega(v, w)$, we promote w to be a child of u, by removing the edge (v, w) and adding the edge (u, w) with $\omega(u, w) = \omega(v, w)$. It is easy to see that the new tree resulted after this operation is equivalent to the old tree, since at the time $\text{UNION}(v, w)$ is issued, u and v are already in the same set. When this procedure stops, we obtain a tree T' in which the weights along any leaf-to-root path are increasing.

In order to perform all these short-cut operations efficiently, for each node v, we need to find its ancestor in T that becomes the parent of v in T'. We cast this problem in a geometric setting. We construct an Euler tour on T, which starts and ends at r and traverses every edge of T exactly twice, once in each direction (see Figure 1(a)). Such a tour can be constructed in $O(\text{SORT}(N))$ I/Os [13]. For each edge $e \in T$, we map it to a horizontal line segment whose y-coordinate is $\omega(e)$, and whose left (resp. right) x-coordinate is the position (index) where e appears for the first (resp. second) time in the Euler tour. Note that the x-spans of these segments are nested. Refer to Figure 1(a) and (b). These segments can be easily constructed in $O(\text{SORT}(N))$ I/Os given the Euler tour.

Abusing notations, we will also use e to denote the segment that corresponds to the edge $e \in T$. Let E be this set of segments. Let $\gamma(e)$ be the shortest segment in E that lies above e and whose x-projection contains that of e. We claim that if $e = (u, v)$ and $\gamma(e) = (w, z)$, then (z, v) is an edge of T', i.e., z is the parent of v in T'. Indeed, the second condition ($\gamma(e)$ above e) ensures that v cannot be shortcut to w, and the first condition ($\gamma(e)$ being the shortest) and the third condition (the x-projection of $\gamma(e)$ containing that of e) ensure that z is the lowest ancestor of v that satisfies the second condition . If $\gamma(e)$ does not exist for some $e = (u, v)$, we know that v must be a child of the root r. Refer to Figure 1(c).

Thus it suffices to compute $\gamma(e)$ for each e. We next observe that since the segments in E are nested, the x-span of a segment e contains that of another segment e' if and only if the x-span of e contains the

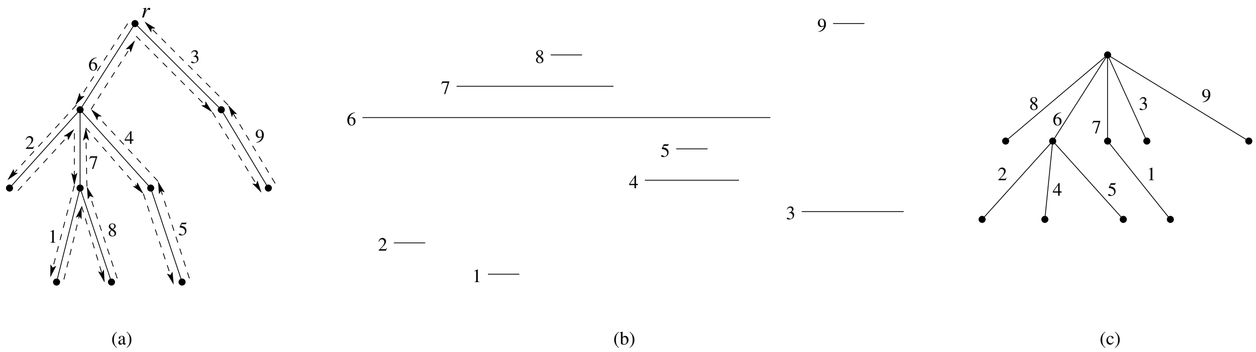


Figure 1: (a) The union tree T, with weights (time stamps) associated with each edge. The dashed lines show the Euler tour. (b) A horizontal segment is built for each edge e of T, with $\omega(e)$ as its y-coordinate and the positions of e in the Euler tour as its left and right x-coordinates. (c) The equivalent union tree T' after transformation.

x-coordinate of the left endpoint of e'. Let P be the set of left endpoints of the segments in E. For a point $p \in P$ and a subset $X \subseteq E$, let $\gamma(p, X)$ denote the shortest segment of X that is above p and whose x-span contains the x-coordinate of p. The problem now reduces to computing $\gamma(p, E)$ for each $p \in P$. We use the distribution sweeping technique [25] to solve this problem. We first sort E and P by y-coordinates. We then divide the plane into $m = M/B$ vertical slabs $L_1, \ldots, L_m$, each of which contains roughly the same number of segment endpoints of E. Let $P_i = P \cap L_i$, let E_i be the set of segments with at least one endpoint inside L_i, and E_i^* the set of segments that completely span L_i. Note that for any point $p \in P_i$, $\gamma(p, E)$ is the shorter segment of $\gamma(p, E_i)$ and $\gamma(p, E_i^*)$. We compute the sets E_i, P_i, and the segment $\gamma(p, E_i^*)$ for each $p \in P_i$, by a sweep-line algorithm, and compute $\gamma(p, E_i)$ for each $p \in P_i$ and for $1 \leq i \leq m$ recursively. We sweep the plane in the $(-y)$-direction starting from $y = \infty$. For each slab L_i, we maintain in memory the shortest segment λ_i among the segments swept so far that completely span L_i. When the sweep line reaches a point $p \in P$, we determine the slab L_i that contains p, set $\gamma(p, E_i^*)$ to λ_i, and add p to P_i. When the sweep line reaches a segment e of E, we add e to E_i if L_i contains at least one of the endpoints of e, and update λ_j if e spans L_j and e is shorter than λ_j. We recursively solve the problem for each (E_i, P_i). The sweep can be implemented by a scan of P and E. Since the depth of recursion is $O(\log_{M/B} \frac{N}{B})$, the overall cost is $O(\textsc{sort}(N))$ I/Os.

Traversing T'. To obtain an ordering of the elements of U that results in Σ being an interval union-find instance, we perform a weight-guided, in-order traversal on T' starting from the root. When we reach a node u of T' that has children $u_1, \ldots, u_k$ where $w(u, u_1) < \cdots < w(u, u_k)$, first we recursively visit the subtree rooted at u_1, then visit u, and then recursively visit the subtrees rooted at $u_2, \ldots, u_k$. To perform this traversal I/O-efficiently we first order each node's children in weight order, then compute an Euler tour. In the tour a leaf of T' appears once, and an internal node with k children appears $k + 1$ times. It is easy to verify that, for any internal node u, if we delete all of u's appearances but the second from the tour, then the desired order is obtained. To do so we first associate with each appearance of u with a node id and its position in the Euler tour, then sort them by the node id, and finally do a scan to remove all but the second appearance of each internal node.

Lemma 1 *We can convert a sequence of N union and find operations without redundant union operations into an instance of the batched interval union-find problem in $O(\textsc{sort}(N))$ I/Os.*

Proof: It only remains to show that this ordering indeed produces an instance of the interval union-find problem. Let u be any node of T', and let $u_1, \ldots, u_k$ be u's children where $w(u, u_1) < \cdots < w(u, u_k)$. For any $1 \le i \le k$, when the union operation corresponding to (u, u_i) is to be performed, in the sequence Σ, all union operations corresponding to the whole subtree of u_i must have already been performed, since they have smaller weights than $\omega(u, u_i)$. If $i = 1$, then u must have not been joined with any other node at the time. So the union operation corresponding to the edge (u, u_1) joins u_1's subtree and the singleton set $\{u\}$, which is immediately after u_1's subtree in the ordering. If $i \ge 2$, then all the subtrees of $u_1, \ldots, u_{i-1}$ must have been joined together with u, and u_i's subtree is immediately after these subtrees in the ordering, so the union operation corresponding to the edge (u, u_i) also joins adjacent sets. $\square$

2.2 Solving interval union-find

We solve the interval union-find problem by formulating it as another geometric problem. Let $x_1 < x_2 < \cdots$ be the sequence of ordered elements of U. For each union operation $\sigma_t = \text{UNION}(x_i, x_j)$ in Σ, we create a horizontal line segment with y-coordinate t, x-coordinate x_i and right x-coordinate x_j. For each find operation $\sigma_t = \text{FIND}(x_i)$ in Σ, we create a query point (x_i, t). For $\sigma_t = \text{FIND}(x_i)$ we return the smallest element of the set as its representative at time t that contains x_i. For each query point q, we consider the union of the x-projections of all the segments lower than q. Each interval in the union corresponds to a set at time t (when $\text{FIND}(x_i)$ was performed), so we need to return the left endpoint of the interval that contains x_i.

We solve this geometric problem by solving a series of *batched orthogonal ray-shooting* problems, in which we are given a set of horizontal segments and a set of query points, the goal is to find the first segment hit by a ray shooting upwards from each query point. This is a special case of the *endpoint dominance* problem [9] and can be solved in $O(\text{SORT}(N))$ I/Os. First, from the left endpoint of each segment e we shoot a ray downwards, and collect all the segments whose rays do not hit any other segments. Second, from the left endpoints of these segments we shoot upwards. These rays stop at the first horizontal segments they hit, or go to infinity. Please refer to Figure 2 where we draw these rays in dashed lines. Finally, from each query point $q = (x_i, t)$ we shoot a ray downwards and hit a horizontal segment. If the ray does not hit any segment we simply return x_i as the answer to q; otherwise, we shoot another ray leftward from q and hit one of the vertical line segments constructed in the second step. Then the x-coordinate of this vertical segment is returned as the answer to q. It is not difficult to verify that we indeed return the correct representative element for each query point. The whole query answering process is just four instances of the batched orthogonal ray-shooting problem, thus can be completed in $O(\text{SORT}(N))$ I/Os.

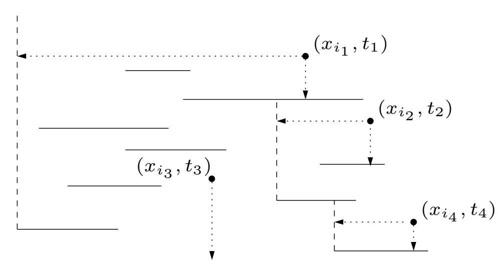


Figure 2: Answering find queries $\sigma_t = \text{FIND}(x_i)$. If no segment lies below a left endpoint, it is connected to the segment lying immediately above it by a dashed vertical line.

Lemma 2 *Given a sequence of N union and find operations possibly with redundant union operations, the batched interval union-find problem can be solved in $O(\text{SORT}(N))$ I/Os.*

Combining Lemmas 1 and 2, we obtain the main result.

Theorem 1 *Given a sequence of N union and find operations with no redundant union operations, the batched union-find problem can be solved in $O(\text{SORT}(N))$ I/Os.*

Remark 1 In some applications, certain elements are required to be the representative elements of the sets returned by the find operations. For example, each element may be weighted, and the minimum-weight element of a set is required to be the representative of the set (see Section 4 and 5). Our algorithm as described above may pick arbitrary elements as the representatives, but such a requirement can be satisfied by running our algorithm twice, followed by a post-processing step in $O(\text{SORT}(N))$ I/Os. We first answer all the find queries by returning the minimum element in the ordering produced by our reduction from union-find to interval union-find (Section 2.1); then by solving the interval union-find problem symmetrically we can also return the maximum elements in the same ordering. Now for each find query we have both the minimum and maximum elements of the set, returning the minimum-weight element in the set becomes a range-minimum query. This batched range-minimum problem can be easily solved in $O(\text{SORT}(N))$ I/Os using distribution sweeping [25].

By running our algorithm and the $O(N)$ RAM algorithm of [22], the bound in Theorem 1 is in fact $O(\text{PERM}(N))$. This is tight in the worst case.

Theorem 2 *Given a sequence of N union and find operations with no redundant union operations, any deterministic batched union-find algorithm has to spend $\Omega(\text{PERM}(N))$ I/Os in the worst case.*

Proof: We can prove the $\Omega(\text{PERM}(N))$-I/O lower bound by a simple reduction from permutation. The idea is to build a sequence of union and find operations such that the results to the find queries exactly form the desired permutation. The argument holds for any deterministic batched union-find algorithm that does not examine the content of the elements, i.e., the algorithm's behavior is determined only by the union and find operations, not the content of the elements. Consider the following $O(\frac{N}{B})$-I/O reduction from permuting. Let $x_1, x_2, \ldots, x_N$ be a set of N elements. Let $\sigma : [1, N] \rightarrow [1, N]$ be a permutation and we want to move x_i to position $\sigma(i)$. We create N special elements $y_1, \ldots, y_N$, and y_i can be created directly from i. We generate two sequences of union-find operations: $\text{UNION}(x_1, y_{\sigma(1)})$, $\text{UNION}(x_2, y_{\sigma(2)})$, $\ldots$, $\text{UNION}(x_N, y_{\sigma(N)})$, $\text{FIND}(y_1)$, $\text{FIND}(y_2)$, $\ldots$, $\text{FIND}(x_N)$, and $\text{UNION}(y_{\sigma(1)}, x_1)$, $\text{UNION}(y_{\sigma(2)}, x_2)$, $\ldots$, $\text{UNION}(y_{\sigma(N)}, x_N)$, $\text{FIND}(y_1)$, $\text{FIND}(y_2)$, $\ldots$, $\text{FIND}(x_N)$. We feed the two sequences to any union-find algorithm $\mathcal{A}$. For any i, the answer to $\text{FIND}(y_{\sigma(i)})$ must be x_i in one of the two sequences, since the two union-find operation sequences look identical to $\mathcal{A}$. So we can scan the results returned by $\mathcal{A}$ in parallel, and construct the desired permutation in $O(\frac{N}{B})$ I/Os. $\square$

2.3 Handling non-redundant union operations

So far we have assumed that there are no redundant union operations in Σ. If this assumption does not hold, $G(\Sigma)$ is not a forest. We first compute the MST (minimum spanning forest in general) on the union graph $G(\Sigma)$, and delete all edges that are not in the MST. From Kruskal's algorithm [14] we know that only the union operations corresponding to the MST edges of $G(\Sigma)$ are non-redundant. For a general graph with N edges, computing the MST takes $O(\text{SORT}(N) \log \log B)$ I/Os by a deterministic algorithm [6] or expected $O(\text{SORT}(N))$ I/Os by a randomized algorithm [13]; if the graph is planar, computing the MST takes deterministic $O(\text{SORT}(N))$ I/Os [13]. So we have the following.

Theorem 3 *Given a sequence Σ of N union and find operations possibly with redundant union operations, the batched union-find problem can be solved in by a deterministic algorithm with $O(\text{SORT}(N) \log\log B)$ I/Os or a randomized algorithm with expected $O(\text{SORT}(N))$ I/Os. If the union graph of Σ is planar, the bound is deterministic $O(\text{SORT}(N))$ I/Os.*

In the on-line union-find problem, redundant union operations are not an issue, since we can always issue two find operations to before issuing a union operation to make sure it is non-redundant. However, in the batched case it is a little annoying that we have to state different bounds depending on whether there are redundant union operations or not, due to the lack of an $O(\text{SORT}(N))$-I/O MST algorithm. Note that in internal memory, Gabow and Tarjan [23] also assumed that there were no redundant union operations, because at that time, the linear-time MST algorithm [20] (assuming integer weights) was not developed yet. It is believed that an $O(\text{SORT}(N))$-I/O deterministic MST algorithm does exist, and we will be able to not worry about redundant union operations when such an algorithm is developed.

2.4 Semi-on-line union-find

The *semi-on-line union-find* problem is another variant of the union-find problem where all the union operations are given in advance, but the find queries appear in an on-line fashion. Suppose we are given a sequence of union operations, each with a time stamp. A $\text{FIND}(x, t)$ query asks for the representative of the set containing a query element x at a specific time t, and we want to build a data structure that supports these queries in an on-line fashion. Such a structure that allows queries in the past is also called a *multiversion* or *partially persistent* structure [10, 16], and it is useful in situations where we are not only interested in the data in their latest version, but also their development over time.

We can solve this variant by combining our algorithm and the persistent B-tree [10]. Observe that in our batched union-find algorithm, we in fact have not used the information of the find queries until the very last two steps, when we shoot a ray downwards from the query point, and then shoot to the left. Such *orthogonal ray-shooting* queries can be handled by a *persistent B-tree* [10] in worst-case $O(\log_B N)$ I/Os per query. Thus to solve the semi-on-line union-find problem, we can run our batched union-find algorithm until we have generated those horizontal and vertical segments of Figure 2, and then build two persistent B-trees on these segments, respectively. Since a persistent B-tree takes linear space and can be constructed in $O(\text{SORT}(N))$ I/Os [38], we have the following.

Theorem 4 *With $O(\text{SORT}(N))$ I/Os, we can build a linear-size data structure for the semi-on-line union-find problem such that a find query at any specific time can be answered in $O(\log_B N)$ I/Os.*

3 A Practically Efficient Union-Find Algorithm

In this section we present a simple divide-and-conquer algorithm that bypasses the need of an MST algorithm. It uses $O(\text{SORT}(N) \log(\frac{N}{M}))$ I/Os, but has a much smaller hidden constant, and can be easily implemented. The input to a recursive call is a sequence Σ of union and find operations. The recursive call outputs the answers of all $\text{FIND}(x_i)$ queries in Σ and returns a set R of $(x, \varrho(x))$ pairs, one for each element x involved in any operation in Σ, where $\varrho(x)$ is the representative of the set containing x after all union operations in Σ are performed. The basic idea behind a recursive call, outlined in Figure 1, is the following.

Algorithm 1 Recursive call $\text{UNION-FIND}(\Sigma)$.

input: a sequence Σ of union and find operations;
output: a set R of $(x, \varrho(x))$ pairs for each element x involved in Σ.

if Σ can be processed in main memory **then**
    Call an internal memory algorithm;
else
    Split Σ into two halves Σ_1 and Σ_2;
    R_1 = $\text{UNION-FIND}(\Sigma_1)$;
(a) For $\forall(x, \varrho(x)) \in R_1$, replace all occurrences of x in Σ_2 with $\varrho(x)$;
    R_2 = $\text{UNION-FIND}(\Sigma_2)$;
(b) For $\forall(x, \varrho(x)) \in R_1$, if $\exists(y, \varrho(y)) \in R_2$ s.t. $y = \varrho(x)$, replace $(x, \varrho(x))$ with $(x, \varrho(y))$ in R_1;
    return $R_1 \cup R_2$.

If Σ fits in main memory, we use an internal memory algorithm; otherwise we split Σ into two halves Σ_1 and Σ_2. We solve Σ_1 recursively. Before solving Σ_2 recursively, we use the element-representative set R_1, returned by the recursive call for Σ_1, to pass on "information" to Σ_2 about how the sets are joined in Σ_1. We do so by replacing each element x involved in any operation in Σ_2 with $\varrho(x)$ if $(x, \varrho(x)) \in R_1$ (line (a)). When the second recursive call on Σ_2 finishes, we need to return the complete and correct element-representative set to the upper level calls. All element-representative pairs in R_2 are correct, but some in R_1 might get updated. We update each $(x, \varrho(x)) \in R_1$ with $(x, \rho(y))$, if there exists some $(y, \varrho(y)) \in R_2$ such that $\varrho(x) = y$ (line (b)). Finally we return the union of R_1 and R_2.

Both line (a) and (b) can be performed by a constant number of sort and scan steps in very similar manners. For example in line (a), we first sort R_1 by x, and Σ_2 by the first operand of the UNION or the operand of the find operations. Then we scan the two lists in parallel, replacing the elements of Σ_2 by the corresponding $\varrho(x)$ in R_1. Similarly we can replace the second operand of the UNION operations. Finally we sort Σ_2 back into its original ordering according to the time stamps. With a little care in the implementation, we can reduce the number of sorts (of $|\Sigma|/2$ elements) to two. The same algorithm also works for line (b). Therefore we spend $O(\text{SORT}(|\Sigma|))$ I/Os in each call (excluding the recursive calls). Since there are $O(\log(\frac{N}{M}))$ levels of recursion, the total cost is $O(\text{SORT}(N)\log(\frac{N}{M}))$ I/Os.

Applications to minimum spanning trees and connected components. Since this practical algorithm allows redundant union operations, it leads to a simple I/O-efficient version of Kruskal's algorithm for computing the MSTs. Given a graph G, we sort all edges of G by weight, and construct a union-find sequence by issuing two FIND and one UNION operations on its two endpoints for each edge in order. After solving this batched union-find problem we check the two FIND results for each edge e. If they are different, we declare that e is in the MST. This simple algorithm runs in deterministic $O(\text{SORT}(N)\log(\frac{N}{M}))$ I/Os. Previously the only known practical external memory algorithm for MST [15] was randomized with expected $O(\text{SORT}(N)\log(\frac{N}{M}))$ number of I/Os.

Similarly we can easily reduce the connected components problem to the batched union-find problem. Given a graph G, we simply issue a union operation for each of its edges, followed by a find query for each of its vertices. Previously known algorithms take $O(\text{SORT}(N)\log\log B)$ I/Os [30] (deterministic) or $O(\text{SORT}(N))$ I/Os [13] (randomized), but are not practical.

4 Topological Persistence

Let $\mathbb{M}$ be a triangulation of $\mathbb{R}^2$ in which a height is associated with every vertex of $\mathbb{M}$. $\mathbb{M}$ defines a piecewise-linear height function $f : \mathbb{R}^2 \to \mathbb{R}$. It will be convenient to assume $f(u) \neq f(v)$ for all vertices $u \neq v$ in $\mathbb{M}$. There are three types of *critical* vertices on $\mathbb{M}$: *minima*, *saddles*, and *maxima*, distinguished by the number of connected components in u's *lower link* $\mathrm{Lk}^-(u)$—the set of vertices adjacent to u whose heights are smaller than $f(u)$ and the set of edges connecting them. Refer to Figure 3. For ease of representation we assume that each saddle is *simple*, i.e., $\mathrm{Lk}^-(u)$ for each saddle u consists of exactly two connected components. When $\mathrm{Lk}^-(u)$ has more than two connected component (e.g. the rightmost in Figure 3), we unfold it into a number of simple saddles [17].

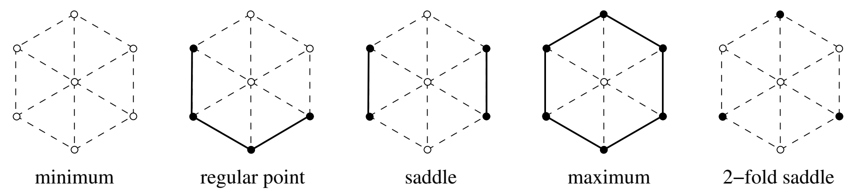


Figure 3: Classification of a vertex based its lower link. The lower link is marked black.

Roughly speaking, *topological persistence* of f (or $\mathbb{M}$) is defined as follows. Suppose we sweep a horizontal plane in the direction of increasing values of f, and keep track of $\mathbb{M}_h = \{x \in \mathbb{R}^2 \mid f(x) < h\}$. A component of $\mathbb{M}_h$ starts at a minimum and ends at a saddle when it merges with another, older component. A hole of $\mathbb{M}_h$ starts at a saddle and ends when it is closed off at a maximum. Based on this observation, Edelsbrunner et al. [18] propose a scheme to pair a critical point that creates a component or a hole with the one that destroys it. For each minimum-saddle or maximum-saddle pair, the absolute difference between their heights is defined as the *persistence* of that pair of critical vertices. See [17, 18] for a formal definition.

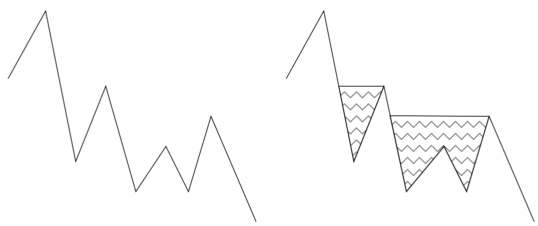
We first sketch the algorithm in [17, 18], and then describe how to modify it. We focus on the minimum-saddle pairs, as the other case is symmetric. We sort all vertices by their height. We sweep a horizontal plane bottom-up and maintain the connected components of $\mathbb{M}_h$ when the sweep plane passes through $f(x) = h$. For each component of $\mathbb{M}_h$, we maintain its minimum-height vertex as its representative. When we pass a minimum, a new component is created. When we pass a saddle, the two components adjacent to the saddle get merged if they were different. If so, between the two representatives of the two components, we choose the lower one to continue to represent the merged component, while the other is paired with the saddle. Passing regular points or maxima does not change the set of components.

After the initial sorting, the sweep can be implemented using a batched union-find algorithm, as follows. When passing a regular vertex u, we pick an arbitrary vertex v in $\mathrm{Lk}^-(u)$ and issue a $\textsc{Union}(u, v)$ operation. When passing a saddle u, we first issue two find operations: For each of the two connected components of $\mathrm{Lk}^-(u)$, an arbitrary vertex v in the component is chosen and a $\textsc{Find}(v)$ is issued. Next for each such v, we issue the union operation $\textsc{Union}(u, v)$. After sweeping over all vertices, we invoke our batched union-find algorithm on the resulting sequence Σ. Using the remark following Theorem 1, we can use the vertex of the lowest height as the representative of the component it belongs to. Finally, we scan the results of the find queries. If the answers to the two FIND queries of a saddle are different, we pair the higher one with the saddle. It can be checked that the union graph is planar, so we conclude the following.

Theorem 5 *Topological persistence of a height function defined by a triangular mesh in $\mathbb{R}^2$ with N vertices can be computed using $O(\textsc{sort}(N))$ I/Os.*

Automated terrain flooding. Flow analysis of a terrain is a central problem, which models how water flows and how river networks form when water is uniformly poured on a terrain. Almost all existing flow models proposed in the literature [27, 32] assume that once water flows into a minima, it never flows out. This is of course only true if the minima corresponds to the ocean or some big lake. However, on a mesh constructed from the elevation data of a terrain, there are numerous minima (often called *sinks*), due to either measurement noises or real small pits on the terrain. As a result, various sink-removal techniques have been proposed to remove all the spurious sinks before computing the actual flows.

Currently, the most popular sink-removal method is *flooding* [27], which has been used in many commercial and open-source GIS systems such as ArcInfo [19] and GRASS [7, 26]. The idea of flooding is to simulate uniformly pouring water on the terrain until all sinks are filled and a steady-state is reached. Refer to Figure 4. In the end, the only remaining sink is the "outside", which corresponds to the ocean. Finally, all the flooded vertices are raised to its *raise elevation*, which is the height of the water at that vertex. This procedure produces a sink-less terrain, on which various flow models can be applied. An $O(\text{SORT}(N))$-I/O algorithm for flooding is given in [7].



(a) (b)

Figure 4: (a) The terrain. (b) Flood the terrain until a steady-state is reached.

One serious problem with this flooding method is that it floods all sinks, regardless of its importance. As such, many important geographical features, such as big lakes or river valleys that do not end up in the ocean will vanish after this flooding procedure. To preserve these features, users have to manually mark the important sinks as "real sinks" to distinguish them from the spurious sinks. This is labor-consuming, and furthermore it undermines the value of flow computation, one of whose purposes is to determine where the big lakes and rivers are.

We use persistence to measure the importance of sinks on a terrain. For a user-specified threshold τ, we declare all sinks with persistence greater than τ to be the real sinks, while the rest should be removed. The user can change the threshold to control the smallest feature size to be preserved. Thus we get an automated way for flooding that extracts features on the terrain based on their persistence. In Section 6, we show the results of this automated flooding procedure when applied on some real terrains.

Finally, by using the minimum-saddle pairings produced as a by-product of the persistence algorithm, we also developed a new flooding algorithm [1] that automatically takes care of multiple real sinks. The new algorithm has the same asymptotic I/O-bound as the previous algorithm [7], but is much simpler to implement and faster in practice. In particular, it does not need a planar separator [28], which is very complicated to compute. In fact the previous algorithm [7] has only been implemented on a grid representation of the terrain. The new flooding algorithm works on both a grid and a triangular mesh with the same implementation.

5 Contour trees

Let $\mathbb{M}$ be a mesh as defined in Section 4. A *contour* in $\mathbb{M}$ is a connected component of the level set of f at some height h (i.e., $\{x \in \mathbb{R}^2 \mid f(x) = h\}$). As we vary h, the contours vary continuously but their structure changes only at the critical vertices of $\mathbb{M}$. If we increase the height, then a new contour appears at a minimum, a contour disappears at a maximum, two contours join at a saddle, or a contour splits into two at a saddle. The *contour tree* is a graph (actually, a tree) that tracks these changes. Its nodes are the critical vertices in $\mathbb{M}$. Each edge (u, v) in the tree corresponds to a contour that is created at v and destroyed at u. As in [12, 39], we add each regular vertex w to the contour tree by splitting the edge (u, v) to (u, w) and (w, v) if the contour corresponding to (u, v) contains w. Let $\mathcal{C}$ denote the resulting (augmented) contour tree.

We follow the same approach as in [12]. We first construct the so-called *join* and *split* trees of $\mathbb{M}$ and then merge them to construct $\mathcal{C}$. The join tree $\mathcal{J}_\mathbb{M}$ represents all joins in $\mathcal{C}$, and the split tree $\mathcal{S}_\mathbb{M}$ represents all splits in $\mathcal{C}$. For a height h, let $\mathbb{M}_h = \{x \mid f(x) < h\}$. $\mathcal{J}_\mathbb{M}$ is a tree in which (u, v), with $f(u) > f(v)$, is an edge if v is the highest vertex in a connected component of $\mathbb{M}_{f(u)}$ that contains a vertex of $\mathrm{Lk}^-(u)$. The tree $\mathcal{S}_\mathbb{M}$ is defined symmetrically by negating the function f. For any edge $(u, v) \in \mathcal{J}_\mathbb{M}$ with $f(u) > f(v)$, we declare u to be the parent of v, so $\mathcal{J}_\mathbb{M}$ is a tree growing downward with the highest node being the root. Similarly for any edge in the split tree, we declare the lower node to be parent, and it becomes a tree growing upward with the lowest node being the root.

The topological persistence algorithm can be adapted to compute $\mathcal{J}_\mathbb{M}$. We sweep a plane bottom-up and maintain the connected components of $\mathbb{M}_h$. When the sweep plane reaches $f(u)$, for each $v \in \mathrm{Lk}^-(u)$, we connect u with the highest node in the connected component of $\mathbb{M}_h$ that contains v. Duplicate edges are removed. This can be done using our batched union-find algorithm in the same way as the topological persistence case, except that here we require the highest node to be the representative of a component, rather than the lowest. The split tree $\mathcal{S}_\mathbb{M}$ is built symmetrically. As in the persistence case, the union graph is planar, so the algorithm takes $O(\textsc{sort}(N))$ I/Os.

Since the definition of lower links applies to any graph in which height is associated with its vertices, the notion of join and split trees can be extended to $\mathcal{C}$ (i.e., replace $\mathbb{M}$ with $\mathcal{C}$ verbatim in the definition) and the construction algorithm makes sense even when applied to $\mathcal{C}$ instead of $\mathbb{M}$. The following lemma is proved in [12].

Lemma 3 ([12]) *The contour tree $\mathcal{C}$ and the mesh $\mathbb{M}$ have the same join tree and split tree.*

Carr et al. [12] define a linear-time algorithm to construct $\mathcal{C}$ from $\mathcal{J}_\mathbb{M} = \mathcal{J}_\mathcal{C}$ and $\mathcal{S}_\mathbb{M} = \mathcal{S}_\mathcal{C}$ using the following observation. A node v is *qualified* if it is a leaf of $\mathcal{J}_\mathcal{C}$ (resp. $\mathcal{S}_\mathcal{C}$) and has only one child in $\mathcal{S}_\mathcal{C}$ (resp. $\mathcal{J}_\mathcal{C}$), and we call the edge (u, v) a *qualified edge*, where u is the parent of v in $\mathcal{J}_\mathcal{C}$ (resp. $\mathcal{S}_\mathcal{C}$). A qualified node must be a leaf of $\mathcal{C}$, and any leaf of $\mathcal{C}$ must also be a qualified node. For a qualified node v, let $\mathcal{J}_\mathcal{C} \ominus v$ (or $\mathcal{S}_\mathcal{C} \ominus v$) be the tree obtained after deleting the node v and the edges adjacent to it; if v is not a leaf, connect u to the only child of v by an edge.

Lemma 4 ([12]) *A qualified edge (u, v) is an edge of $\mathcal{C}$. If v is a qualified, then $\mathcal{J}_\mathcal{C} \ominus v$ and $\mathcal{S}_\mathcal{C} \ominus v$ are the join and split trees of $\mathcal{C} \ominus v$.*

Using this lemma, Carr et al. [12] repeatedly delete a qualified node from $\mathcal{J}_\mathcal{C}$ and $\mathcal{S}_\mathcal{C}$, and add the corresponding qualified edge to $\mathcal{C}$. Figure 5(a)–(d) illustrate one such iteration, and Figure 5(e) shows the final contour tree constructed. In internal memory, each step takes $O(1)$ time, but we cannot afford even one

I/O per step. It is difficult to batch the steps together since the set of qualified nodes depends on the choices of which qualified nodes to delete in previous steps. We therefore present a different characterization of contour tree edges, formulate the problem as a geometric problem, and solve it using $O(\text{SORT}(N))$ I/Os.

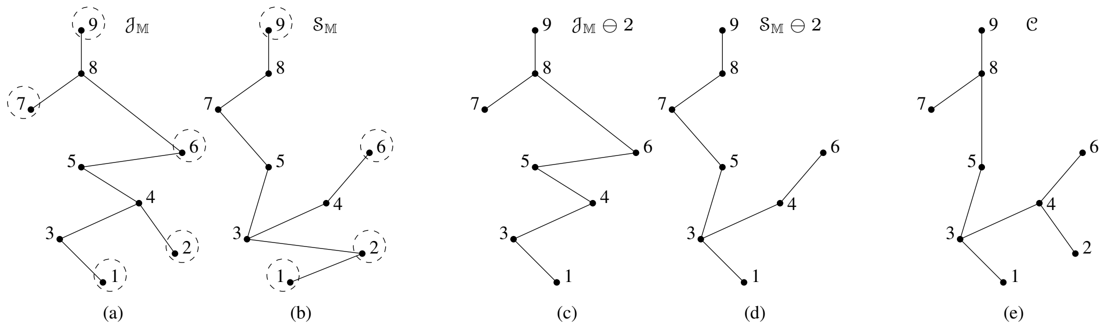


Figure 5: Construction of a contour tree from the join tree and the split tree using Carr et al. [12]: (a) The join tree $\mathcal{J}_{\mathbb{M}}$ of a mesh $\mathbb{M}$, where the number beside each node is its height. (b) The split tree $\mathcal{S}_{\mathbb{M}}$. All qualified nodes are circled. (c)-(d) One step of Carr et al. [12]: node 2 is removed and the edge between node 2 and node 4 is added to $\mathcal{C}$. (e) The final contour tree $\mathcal{C}$.

As a convention, we say a node is both an ancestor and a descendant of itself. For any edge $e = (u, v) \in \mathcal{J}_{\mathbb{M}}$, let $\sigma(e)$ be the highest node that is a descendant of v in $\mathcal{J}_{\mathbb{M}}$ and at the same time an ancestor of u in $\mathcal{S}_{\mathbb{M}}$.

Lemma 5 *$\sigma(e)$ exists for any $e \in \mathcal{J}_{\mathbb{M}}$.*

Proof: First, by simple induction, we can show that for any node u, the subtree of $\mathcal{J}_{\mathbb{M}}$ (resp. $\mathcal{S}_{\mathbb{M}}$) rooted at u contains exactly the set of nodes in the connected component of $\mathbb{M}$ that contains u, when the sweep plane just passes u. Now consider any $e = (u, v) \in \mathcal{J}_{\mathbb{M}}$. During the construction of $\mathcal{J}_{\mathbb{M}}$, u is connected to v because when the sweep plane is at u, v is the highest node in the connected component of $\mathbb{M}$ that contains some vertex w in u's lower link. w must be in the already constructed subtree of $\mathcal{J}_{\mathbb{M}}$ rooted at v, and it also might be v itself. Next consider the construction of $\mathcal{S}_{\mathbb{M}}$. As we pass w, we connect w with the lowest node w' in the connected component of $\mathbb{M}$ that contains u, since u is in w's lower link if f is negated. This connected component contains exactly the set of nodes in the already constructed subtree of $\mathcal{S}_{\mathbb{M}}$ rooted at w', therefore w must be an ancestor of u in $\mathcal{S}_{\mathbb{M}}$. Thus we have found a candidate w for $\sigma(e)$. $\square$

The following lemma characterizes the edges of $\mathcal{C}$.

Lemma 6 *For each edge $e = (u, v) \in \mathcal{J}_{\mathbb{M}}$, $(u, \sigma(e))$ is an edge of $\mathcal{C}$.*

Proof: Please refer to Figure 6(a). Let $e = (u, v)$ be any edge of $\mathcal{J}_{\mathbb{M}}$. We know that $\sigma(e)$ is the highest node that is both a descendant of v in $\mathcal{J}_{\mathbb{M}}$ and an ancestor of u in $\mathcal{S}_{\mathbb{M}}$. Let P_1 be the set of nodes on the path of $\mathcal{J}_{\mathbb{M}}$ from u to $\sigma(e)$, and P_2 the set of nodes on the path of $\mathcal{S}_{\mathbb{M}}$ from $\sigma(e)$ to u. u and $\sigma(e)$ are not included in P_1 and P_2. Note that P_1 and P_2 are disjoint, otherwise we would have chosen a different $\sigma(e)$.

We follow the procedure in [12], i.e., repeatedly delete qualified nodes one by one except u until $\sigma(e)$ becomes qualified. This strategy always works since at any time, the remaining part of the contour tree yet to be constructed has at least two leaves. So there are at least two qualified nodes, and we can always find one other than u to remove.

Next we prove that when $\sigma(e)$ becomes qualified, P_1 must be empty, i.e., u is the parent of $\sigma(e)$ in $\mathcal{J}_{\mathbb{M}}$, and thus $(u, \sigma(e))$ is added to $\mathcal{C}$. Suppose on the contrary $P_1 \neq \emptyset$ when $\sigma(e)$ becomes qualified, and $w \in P_1$

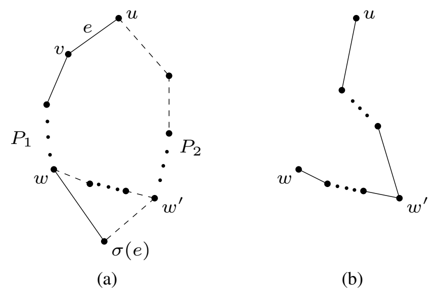


Figure 6: (a) Solid lines represent $\mathcal{J}_{\mathbb{M}}$ edges; dashed lines represent $\mathcal{S}_{\mathbb{M}}$ edges. $\sigma(e)$ is the highest node that is both a descendant of v in $\mathcal{J}_{\mathbb{M}}$ and an ancestor of u in $\mathcal{S}_{\mathbb{M}}$. We show that $(u, \sigma(e))$ is a contour tree edge. (b) Part of the contour tree $\mathcal{C}$.

is the parent of $\sigma(e)$ in $\mathcal{J}_{\mathbb{M}}$. Since $\sigma(e)$ is qualified, by Lemma 4, $(w, \sigma(e))$ is a contour tree edge. Consider the construction of $\mathcal{S}_{\mathbb{M}}$. By Lemma 3, this is the same as if we used the contour tree $\mathcal{C}$ for the sweep. When we sweep to $f(\sigma(e))$, we connect $\sigma(e)$ to the root of the already constructed subtree of $\mathcal{S}_{\mathbb{M}}$ that contains w, since w is in the lower link of $\sigma(e)$ in $\mathcal{C}$ if we negate f. So $\sigma(e)$ must be an ancestor of w in $\mathcal{S}_{\mathbb{M}}$, while it is also an ancestor of u in $\mathcal{S}_{\mathbb{M}}$. Let w' be the nearest common ancestor of u and w in $\mathcal{S}_{\mathbb{M}}$; w' cannot be $\sigma(e)$ because otherwise $\sigma(e)$ would not have been qualified. So w' must be some node in P_2. As the sweep plane reaches $f(w')$ while constructing $\mathcal{S}_{\mathbb{M}}$, w' is the node that merges the components containing u and w together, so w' must be on the path between u and w in $\mathcal{C}$; see Figure 6(b). Next consider the construction of $\mathcal{J}_{\mathbb{M}}$. When the sweep plane reaches $f(u)$, u merges into the component containing w in $\mathcal{J}_{\mathbb{M}}$, which implies that the height of all the nodes on the path in $\mathcal{C}$ connecting u and w is less than $f(u)$. However on the other hand, u and w' are still not in the same component at this point, because otherwise w' would be a descendant of u in $\mathcal{J}_{\mathbb{M}}$ and we would have chosen w' as $\sigma(e)$. This means that the path in $\mathcal{C}$ connecting u and w' must have some node whose height is greater than $f(u)$, but we just concluded that the height of all the nodes on the path from u to w is less than $f(u)$, a contradiction. $\square$

Since $\mathcal{J}_{\mathbb{M}}$ has $N-1$ edges $e = (u, v)$, and $(u, \sigma(e))$ are all different by definition. So if we can compute $\sigma(e)$ for each edge of $\mathcal{J}_{\mathbb{M}}$, we have all the $N-1$ edges of $\mathcal{C}$. In the following, we show how this maps to a geometric problem, which can be solved with $O(\text{SORT}(N))$ I/Os.

First we perform an Euler tour on $\mathcal{J}_{\mathbb{M}}$, and record for each node its first and second appearances in the tour. This gives us an interval $[x_1(u), x_2(u)]$ for each node u, such that u is an ancestor of v if and only if $x_1(u) < x_1(v)$ and $x_2(u) > x_2(v)$. We do the same with $\mathcal{S}_{\mathbb{M}}$, and compute another interval $[y_1(u), y_2(u)]$. The two intervals of a node u naturally maps it to a rectangle $R(u)$ in the plane. Note that the x-spans of these rectangles are nested, so are their y-spans, though the rectangles are not necessarily nested. Now for any edge $e = (u, v) \in \mathcal{J}_{\mathbb{M}}$, computing $\sigma(e)$ translates into the following query: Find the rectangle $R(w)$ with the shortest y-span, whose x-span is contained in that of $R(v)$, and whose y-span contains that of $R(u)$. We form a *query rectangle* $R(e)$ for each $e = (u, v) \in \mathcal{J}_{\mathbb{M}}$ using the x-span of $R(v)$ and the y-span of $R(u)$. The rectangle $R(u)$ mapped from a node is called a *node rectangle*.

We solve this batched problem using the distribution sweeping technique. During the execution, with each query rectangle $R(e)$ we store its best answer (the one with the shortest y-span) found so far. Initially we sort the top and bottom sides of all the node and query rectangles. Then we divide the plane into $\Theta(\sqrt{M/B})$ vertical *slabs*, each of which contains roughly equal number of the left and right sides of the rectangles. We call any number of contiguous slabs a *multislab*, and there are $\Theta(M/B)$ multislabs. Next we

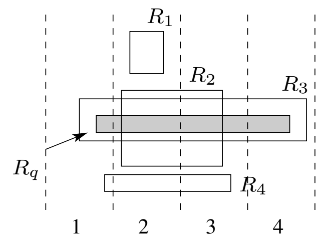


Figure 7: The unshaded rectangles $R_1, \ldots, R_4$ are node rectangles, the shaded rectangle R_q is a query rectangle. During the sweep, R_1 is pushed into the stack for multislab 2-2, R_2 into multislab 2-3, R_3 into 1-4, and R_4 into 1-3. When R_q is reached, the top element of the stacks corresponding to multislabs 2-2, 3-3, and 2-3 are checked. After the sweep, R_3 and R_q are vertically broken into three pieces: a middle piece spans multislab 2-3, a left leftover piece in slab 1, and a right leftover piece in slab 4. R_2 is broken into two pieces, and R_1 has only one piece.

sweep a horizontal line top-down. During the sweep, for each multislab L we maintain an external memory stack that stores all rectangles for which L is the minimal containing multislab (see Figure 7). Whenever we reach the top of a node rectangle, we push it into the corresponding stack; whenever we reach the bottom of a node rectangle, we pop one from the corresponding stack. Since there are $O(M/B)$ stacks, we can allocate one memory block for each stack, so that all stack operations take linear I/Os. When we reach the top of a query rectangle $R(e)$, we look at all the multislabs whose x-spans are contained in that of $R(e)$, and check the top element of each corresponding stack as a candidate to the answer of $R(e)$. During the sweep, we also vertically break each rectangle R, a node rectangle as well as a query rectangle, into at most three pieces: a middle piece that spans the maximal multislab whose x-span is contained in that of R, a left leftover piece and a right leftover piece (see Figure 7). After this rectangle is processed, we throw away its middle piece, and put the two leftover pieces into the corresponding slab. When the sweep finishes, we go into each slab and process those leftover pieces recursively. The fact that the x-spans of these rectangles are nested guarantees that it is sufficient to check the leftover pieces of a node rectangle and a query rectangle in order to determine if one's x-span contains that of the other. Note that a rectangle can generate both a left leftover piece and a right leftover piece only at one level in the recursion, thus employing standard distribution sweeping analysis yields an $O(\text{SORT}(N))$-I/O bound for the algorithm.

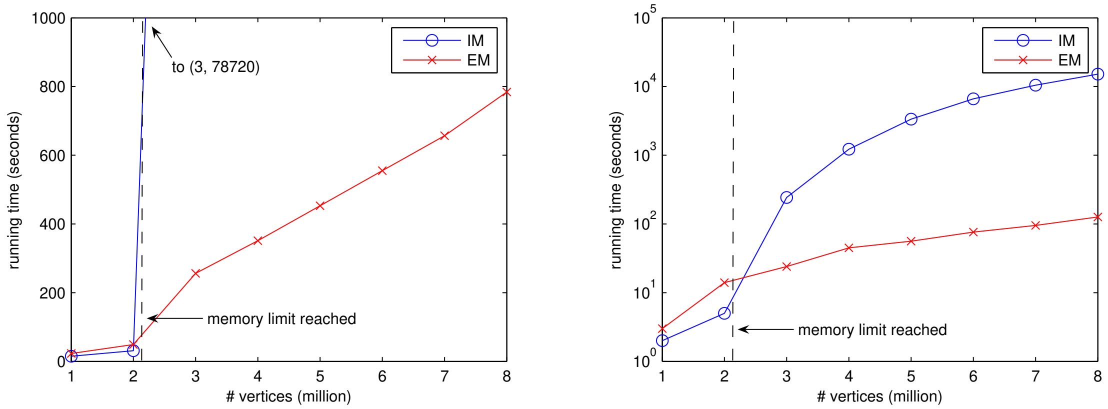
Ptting all the pieces together, we obtain the following.

Theorem 6 *The contour tree of a triangular mesh with N vertices in $\mathbb{R}^2$ can be computed using $O(\text{SORT}(N))$ I/Os.*

Remark 2 Our algorithm extends to higher dimensions, with N replaced by the number of simplices in the mesh [12], except that in $d \geq 3$ dimensions the union graph when building the join and split trees may not be planar, and the $O(\text{SORT}(N) \log\log B)$-I/O MST algorithm or the randomized MST algorithm with expected $O(\text{SORT}(N))$ I/Os needs to be used.

6 Experiments

We have implemented our practical batched union-find algorithm (Section 3) with TPIE [8]. In this section, we report some preliminary experimental results of our algorithm in the context of some of its applications. We compared with the internal memory algorithm that uses link-by-rank with path compression [14]. We could have implemented the linear-time off-line union-find algorithm [22], but it is quite complicated and



(a) (b)

Figure 8: Running time of the internal memory (IM) and the external memory (EM) union-find algorithms for (a) computing minimum spanning trees, and (b) computing topological persistence (running time is shown in log scale).

is unlikely to outperform the simple $O(N\alpha(N))$ on-line algorithm in practice. In the experiments we limit the physical memory of the test machine to 128M (unless otherwise specified) in order to obtain a large data size to memory size ratio. In the experiments with the internal memory algorithm, we relied on virtual memory to handle I/Os (swapping).

We performed three sets of experiments. We first use minimum spanning tree as a test case for the batched union-find algorithm. Next, we apply our union-find algorithm to compute the topological persistence of a terrain. Finally, we use the persistence to the so-called *flooding* problem on terrains.

Computing minimum spanning trees. We investigate the performance of our union-find algorithm when applied to Kruskal's algorithm for computing MSTs. We generated random graphs with N vertices and $5N$ edges. For each edge, we randomly pick its two endpoints, and also randomly pick its weight uniformly from $(0, 1)$. This random graph model is similar to earlier experimental studies of MST algorithms [15, 29].

We tested the internal memory algorithm (IM) and the external memory algorithm (EM) on eight graphs, with N between 1 million to 8 million. The results are shown in Figure 8(a), where the running time reported is just the time spent by the union-find algorithm. It does not include the initial sorting, which is performed by both algorithms. The 128M main memory can accommodate the data structure for a graph with roughly up to 2 million vertices. As we can see, when the memory limit is not reached, the two algorithms perform roughly the same. In fact, in this case EM is the same as IM except for an additional step to check whether the input can be handled in memory. That is why EM spends a little more time for the first two data sets. For larger graphs, the running time of IM immediately shoots up due to the random access pattern in its data structure. On the other hand, the running time of EM scales roughly linearly as the size of the input graph. On the graph with 3 million vertices, IM spends roughly 22 hours, that is, 300 times more than that of EM. We cannot run IM on the other data sets in a reasonable amount of time.

We should point out that there exist other practical external memory MST algorithms [15] that do not use the union-find algorithm; however, our focus is on investigating the performance of union-find algorithms, and a comprehensive experimental evaluation of MST algorithms is out of the scope of this paper.

Computing topological persistence on triangulated terrains. We implemented the topological persistence algorithm using both the internal memory (IM) and our (EM) union-find algorithm. We tested the performance on the elevation data of the Neuse River Basin of North Carolina, obtained from [31]. The data set, consisting of over 0.5×10^9 points, is too large for the IM algorithm, so we created eight smaller data sets by sampling 1 million to 8 million points. We converted this data into a triangular mesh using an I/O-efficient Delaunay triangulation algorithm [2].

The experimental results of these 8 data sets are shown in Figure 8(b) (note the log scale on the running time). Again the running time shown is only that of the union-find algorithms. In this set of experiments we see similar trends of both IM and EM as the MST case, except that IM deteriorates slower in this case. This is because the union and find operations generated by the topological persistence algorithm have better locality than in the MST case due to the underlying geometry. Still, the difference in running time between IM and EM gets more significant as the data set gets larger. On the last data set of 8 million points, the difference is more than two orders of magnitude. Finally, we set the physical memory to its full size (1 GB) and tried the entire Neuse River Basin data set (with 0.5 billion points). EM finishes in about 5.22 hours, while IM crashed after running for about 7 hours, because the 32-bit address space is not enough to accommodate such a large data set.

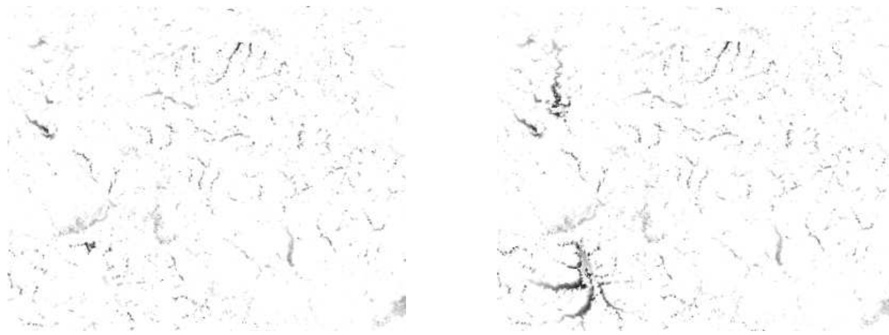
Automated terrain flooding. We computed the distribution of the persistence values of all sinks for the Neuse River Basin. The distribution is highly skewed: A few sinks have large persistence (possible real sinks), while the majority have very small persistence (spurious sinks). Based on this information, we can choose an appropriate persistence threshold to remove all the spurious sinks while preserving the main features. We applied the flooding algorithm to the terrain with a persistence threshold of $\tau = 30$. We also ran the algorithm with $\tau = \infty$, i.e., flooding all sinks, which gives the same results as the previous flooding algorithm. A portion of the original terrain, and the flooded terrains are shown in Figure 9. With a threshold of $\tau = 30$, around 99.5% of the sinks have been removed, while the major features have been preserved. On the other hand, the previous flooding procedure has eradicated some major features, and generated a few large flat areas (for example around the left-bottom corner), which are undesirable. Figure 10 shows the difference in height of the original and the flooded terrain.

(a) (b) (c)

Figure 9: (a) Original terrain. (b) Terrain flooded with persistence threshold $\tau = 30$. (c) Terrain flooded with $\tau = \infty$.

References

[1] P. K. Agarwal, L. Arge, A. Danner, H. Mitasova, T. Moelhave, J. Vahrenhold, and K. Yi. From LIDAR to watershed hierarchies. Manuscript.



(a) (b)

Figure 10: Differences in the height values between the original terrain and the one after flooding with (a) $\tau = 30$ and (b) $\tau = \infty$. Darkness is proportional to the height difference.

[2] P. K. Agarwal, L. Arge, and K. Yi. I/O-efficient construction of constrained Delaunay triangulations. In *Proc. European Symposium on Algorithms*, pages 355–366, 2005.

[3] P. K. Agarwal, H. Edelsbrunner, J. Harer, and Y. Wang. Extreme elevation on a 2-manifold. In *Proc. ACM Sympos. Comput. Geom.*, pages 357–365, 2004.

[4] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.

[6] L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP and multi-way planar graph separation. *Journal of Algorithms*, 53(2):186–206, 2004.

[7] L. Arge, J. Chase, P. Halpin, L. Toma, D. Urban, J. S. Vitter, and R. Wickremesinghe. Flow computation on massive grid terrains. *GeoInformatica*, 7(4):283–313, 2003.

[8] L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proc. European Symposium on Algorithms*, pages 88–100, 2002.

[9] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Proc. European Symposium on Algorithms*, pages 295–310, 1995.

[10] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275, 1996.

[11] P. T. Bremer, V. Pascucci, H. Edelsbrunner, and B. Hamann. A topological hierarchy for functions on triangulated surfaces. *IEEE Trans. Vis. Comput. Graphics*, 10:385–396, 2004.

[12] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. *Computational Geometry: Theory and Applications*, 24:75–94, 2003.

[13] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd Edition*. The MIT Press, Cambridge, Mass., 2001.

[15] R. Dementiev, P. Sanders, D. Schultes, and J. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *Proc. 3rd IFIP Intl. Conf. on Theoretical Computer Science*, pages 195–208, 2004.

[16] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[17] H. Edelsbrunner, J. Harer, and A. Zomorodian. Hierarchical morse complexes for piecewise linear 2-manifolds. In *Proc. ACM Sympos. Comput. Geom.*, pages 70–79, 2001.

[18] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. In *Proc. IEEE Sympos. Found. Comput. Sci.*, pages 454–463, 2000.

[19] Environmental Systems Research Inc. ARC/INFO Professional GIS, 1997. Version 7.1.2.

[20] F. W. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *Proc. 31st Annu. IEEE Sympos. Found. Comput. Sci.*, pages 719–725, 1990.

[21] M. L. Fredman and M. E. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st Annu. ACM Sympos. Theory Comput.*, pages 345–354, 1989.

[22] H. Gabow and R. Tarjan. A linear time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30:209–221, 1985.

[23] H. Gabow and R. Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18:1013–1036, 1989.

[24] B. A. Galler and M. J. Fisher. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964.

[25] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 714–723, 1993.

[26] GRASS Development Team. GRASS GIS homepage. http://www.baylor.edu/grass/.

[27] S. Jenson and J. Domingue. Extracting topographic structure from digital elevation data for geographic information system analysis. *Photogrammetric Engineering and Remote Sensing*, 54(11):1593–1600, 1988.

[28] A. Maheshwari and N. Zeh. I/O-optimal algorithms for planar graphs using separators. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 372–381, 2002.

[29] B. M. E. Moret and H. D. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. In *Proc. 2nd Workshop Algorithms Data Struct.*, volume 519 of *Lecture Notes Comput. Sci.*, pages 400–411. Springer-Verlag, 1991.

[30] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 687–694, 1999.

[31] North Carolina Flood Mapping Program. http://www.ncfloodmaps.com.

[32] J. F. O'Callaghan and D. M. Mark. The extraction of drainage networks from digital elevation data. *Computer Vision, Graphics and Image Processing*, 28, 1984.

[33] R. Seidel and M. Sharir. Top-down analysis of path compression. *SIAM Journal on Computing*, 34(3):515–525, 2005.

[34] S. P. Tarasov and M. N. Vyalyi. Construction of contour trees in 3D in $O(n \log n)$ steps. In *Proc. 14th Sympos. Comput. Geom.*, pages 68–75, 1998.

[35] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, 1975.

[36] R. E. Tarjan. A class of algorithms that require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18:110–127, 1979.

[37] R. E. Tarjan and J. V. Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31:245–281, 1984.

[38] J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proc. International Conference on Very Large Databases*, pages 406–415, 1997.

[39] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proc. ACM Annual Symposium on Computational Geometry*, pages 212–219, 1997.

[40] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[41] N. Zeh. I/O-efficient graph algorithms. *EEF Summer School on Massive Data Sets*, 2002.

[42] A. Zomorodian. *Topology for Computing*. Cambridge University Press, 2005.

[43] A. Zomorodian and G. Carlsson. Computing persistent homology. In *Proc. 20th Sympos. Comput. Geom.*, pages 347–356, 2004.