Sea-Rise Flooding on Massive Dynamic Terrains

Lars Arge
MADALGO, Aarhus University, Denmark
large@cs.au.dk

Mathias Rav
SCALGO, Aarhus, Denmark
mathias@scalgo.com

Morten Revsbæk
SCALGO, Aarhus, Denmark
morten@scalgo.com

Yujin Shin
MADALGO, Aarhus University, Denmark
yujinshin@cs.au.dk

Jungwoo Yang
SCALGO, Aarhus, Denmark
jungwoo@scalgo.com

Abstract

Predicting floods caused by storm surges is a crucial task. Since the rise of ocean water can create floods that extend far onto land, the flood damage can be severe. By developing efficient flood prediction algorithms that use very detailed terrain models and accurate sea-level forecasts, users can plan mitigations such as flood walls and gates to minimize the damage from storm surge flooding.

In this paper we present a data structure for predicting floods from dynamic sea-level forecast data on dynamic massive terrains. The forecast data is dynamic in the sense that new forecasts are released several times per day; the terrain is dynamic in the sense that the terrain model may be updated to plan flood mitigations.

Since accurate flood risk computations require using very detailed terrain models, and such terrain models can easily exceed the size of the main memory in a regular computer, our data structure is *I/O-efficient*, that is, it minimizes the number of *I/Os* (i.e. block transfers) between main memory and disk. For a terrain represented as a raster of N cells, it can be constructed using $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ I/Os, it can compute the flood risk in a given small region using $O(\log_B N)$ I/Os, and it can handle updating the terrain elevation in a given small region using $O(\log_B^2 N)$ I/Os, where B is the block size and M is the capacity of main memory.

2012 ACM Subject Classification Information systems → Geographic information systems

Keywords and phrases Computational geometry, I/O-algorithms, merge tree, dynamic terrain

Digital Object Identifier 10.4230/LIPIcs.SWAT.2020.6

Funding The work in this paper was supported in part by the Danish National Research Foundation and Innovation Fond Denmark.

1 Introduction

Predicting floods caused by storm surges is a crucial task. Since the rise of ocean water can create floods that extend far onto land, the flood damage can be severe. By developing efficient flood prediction algorithms, we hope to minimize the damage from storm surge flooding by allowing users to plan mitigations such as flood walls and gates, or evacuation of affected areas.

Due to the advancement of remote sensing technology and meteorology, nowadays very detailed terrain models and accurate sea-level forecasts can be obtained, and these datasets can be used for designing accurate flood prediction algorithms. For example, the publicly available detailed raster terrain model of Denmark [13] (where each cell represents a 0.4 by 0.4 meter region) contains 267 billion cells. Furthermore, the Danish Meteorological Institute releases a sea-level forecast of the Danish territorial waters every 6 hours, containing 81 thousand values (each value corresponding to the forecasted sea-level in a 1 km^2 region).

Designing an efficient flood prediction algorithm is a challenging task. The algorithm must be fast enough that the computation can finish before the actual disastrous event happens or a new forecast appears, while to guarantee the accuracy of the prediction it is required to use very detailed data that is larger than the main memory in a typical computer. For example, with the terrain model and sea-level forecast datasets mentioned above, the algorithm must process the terabyte-sized terrain model and complete well within 6 hours before a new sea-level forecast is released. Existing flood prediction algorithms process the entire terrain model to compute the flood risk when a new forecast is released. Moreover, if a user modifies the terrain model to e.g. examine the effect of planned mitigations, the entire terrain model must be processed again. This is critical in practice since even a simple scan of a detailed terrain model such as the model of Denmark easily takes a few hours. However, users examine flood risk and plan flood mitigations not for the entire terrain but only for small regions of the terrain. Therefore, supporting efficient computations for a small region in the terrain would make flood prediction algorithms more practically relevant.

In this paper, we consider the problem of predicting floods from dynamic sea-level forecast data on dynamic massive terrains. The forecast data is dynamic in the sense that new forecasts can appear; the terrain is dynamic in the sense that the terrain model may be updated locally, to e.g. incorporate planned flood mitigations. We present a data structure that allows updating respectively the forecast and the terrain, and a query algorithm to report the flood height in a given query window (i.e. a small region of the terrain examined by the user). The data structure is *I/O-efficient*, meaning it can efficiently handle terrain models much larger than main memory.

The dynamic sea-level flooding problem. We use the *I/O-model* by Aggarwal and Vitter [4] to design and analyze our algorithms. In this model, the computer is equipped with a two-level memory hierarchy consisting of an internal memory and a (disk-based) external memory. The internal memory is capable of holding M data items, while the external memory is of conceptually unlimited size. All computation has to happen on data in internal memory. Data is transferred between internal and external memory in blocks of B consecutive data items. Such a transfer is referred to as an *I/O-operation* or *I/O*. The cost of an algorithm is the number of I/Os it performs.

A terrain is typically represented using a digital elevation model (DEM) as a two-dimensional array with N cells (a *raster*). Note that by storing a raster of N cells in $O(\frac{N}{B})$ tiles of size $\sqrt{B} \times \sqrt{B}$, for any $s \geq B$ a $\sqrt{s}$-by-$\sqrt{s}$ square of the raster can be read or written in $O(\frac{s}{B})$ I/Os. Each cell in a raster terrain T is either a *terrain cell*, meaning that the corresponding location is on land, or an *ocean cell.* We denote the elevation of cell u in T by $h_T(u)$; the height of an ocean cell is undefined. For two cells u and v, we say that u and v are *adjacent* (or that v *is a neighbor of* u) if u and v share at least one point on their boundary. A terrain cell is a *coastal cell* if it is adjacent to an ocean cell. Since the resolution of terrain models is typically much greater than the resolution of forecasts, we partition the coastal cells into a set C of connected *coastal regions*, each region corresponding to the coastal cells associated with a single cell of the forecast. We denote the region containing a coastal cell u by $C(u)$. A *sea-level forecast* is a function $F : C \to \mathbb{R}$ that assigns a sea-level elevation value to each coastal region. We denote the sea-level elevation of a region $R \in C$ by $F(R)$, and define a function $h_F(v) = F(C(v))$ that assigns a forecast value to each coastal cell v.

A terrain cell u is *flooded through* a coastal cell v if there exists a path $p = u \rightsquigarrow v$ of adjacent terrain cells such that each cell x in p has $h_T(x) < h_F(v)$. We call p a *flood path* of u with respect to v. Let S_u be the set of coastal cells that have flood paths to u. For a flooded cell u, we define its *flood height* as $f(u) = \max_{v \in S_u}(h_F(v) - h_T(u))$. The *flood source* of u is the region R containing the coastal cell $v \in S_u$ that has the highest sea-level value, that is, $f(u) = h_F(v) - h_T(u)$; for simplicity, we assume that the flood source is unique.

The *dynamic sea-level flooding* problem we consider in this paper consists of constructing an I/O-efficient data structure on a raster terrain T, a partition C of the coastal cells and a forecast F, and supporting the following operations I/O-efficiently, where Q_B and U is a query and an update of $\sqrt{B} \times \sqrt{B}$ cells, respectively:

- Flood-Height(Q_B): Return the flood height $f(u)$ of each terrain cell u in Q_B.
- Forecast-Update(F): Update the data structure with the new forecast F.
- Height-Update(Q_B, U): Set the heights of terrain cells in Q_B to the values given by U.

Previous work. Previously, a large number of results on I/O-efficient algorithms have been obtained. Aggarwal and Vitter [4] showed that reading and sorting N items require $\Theta(\text{Scan}(N)) = \Theta(\frac{N}{B})$ and $\Theta(\text{Sort}(N)) = \Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os, respectively. A set of N items can be maintained in an $O(\frac{N}{B})$-block search tree such that updates and queries can be performed in $O(\log_B N)$ I/Os. Refer e.g. to the surveys [6, 20].

I/O-efficient algorithms for modeling flooding on terrains have been studied extensively (e.g. [2, 5, 9, 12, 7, 10, 8, 11, 14]). A number of results have also been obtained for flooding from sea-level rise. However, to our knowledge, the problem of computing flood heights while I/O-efficiently supporting updates of sea-level forecasts and the heights of terrain cells has not been studied before.

When the sea level rises *uniformly* with the same amount h_r, that is, all coastal cells belong to the same region R and $F(R) = h_r$, then it is easy to see that there is a threshold ℓ_u for each terrain cell u so that u is flooded with flood height $h_r - h_T(u)$ if and only if $h_r \geq \ell_u$. The thresholds ℓ_u for all cells u in the terrain can be computed in $O(\text{Sort}(N))$ I/Os [7], after which the flood heights in any square of B cells can be easily reported for an arbitrary h_r in $O(1)$ I/Os.

Arge et al. [11] introduced an $O(\text{Sort}(N))$-I/O algorithm for computing flood heights when the sea-level rises non-uniformly. Their algorithm relies on the so-called *merge tree* that captures the nesting topology of depressions in T [14, 15]. Their algorithm has been incorporated into a real-time storm surge flood warning system in a pilot project between Danish Meteorological Institute (DMI) and the research spin-out company SCALGO. This system maps the extent of any flood risk resulting from the current sea-level forecast for the Danish territorial waters (updated by DMI every six hours) in full resolution on the 0.4-meter terrain model of Denmark. As part of the pilot project, the algorithm has been engineered to support efficient recomputation when a new forecast is released. However, it is unable to handle updates to the terrain without incurring $O(\text{Sort}(N))$ I/Os, which is the main motivation for the work in the present paper.

In addition to rasters, TINs are commonly used to represent terrain models. A TIN $T_\triangle$ consists of a planar triangulation of N vertices in the plane, each vertex v having an associated height $h_{T_\triangle}(v)$. The height of a point interior to a face is a linear interpolation of the face vertices, so that $h_{T_\triangle}$ is a continuous piecewise linear function. Since a raster can be triangulated into a TIN, algorithms for TIN representations can be applied to raster representations as well, but the converse is not true. However, GIS applications typically implement algorithms for rasters directly, as rasters are often easier to process with simple algorithms. Furthermore, often data, such as the terrain data that we consider, is available as rasters.

To maintain dynamic terrains, Agarwal et al. [3] presented an internal-memory so-called kinetic data structure for maintaining the so-called *contour tree* of a TIN terrain $T_\triangle$ with a time-varying height function. Whereas the merge tree represents how the depressions of T are nested, the contour tree represents the nested topology of the *contours* defined by $T_\triangle$. This result was extended to an I/O-efficient data structure by Yang [21]. He showed that for a TIN terrain with N vertices, the contour tree can be constructed in $O(\text{Sort}(N))$ I/Os and the elevation of a TIN vertex can be updated in $O(\log_B^2 N)$ I/Os.

Our results. In this paper we introduce the first data structure for the dynamic sea-level flooding problem. Our data structure can be constructed in $O(\text{Sort}(N))$ I/Os and uses $O(\frac{N}{B})$ blocks, where N is the number of cells in T. Flood-Height(Q_B) can be performed in $O(\log_B N)$ I/Os, Forecast-Update(F) in $O(\text{Scan}(F))$ I/Os, and Height-Update(Q_B, U) in $O(\log_B^2 N)$ I/Os. Note that the number of I/Os needed to update a forecast does not depend on N, and that the terrain update bound matches the update bound of Yang [21].

Our result assumes that the size of partition set C is smaller than M (which implies that the forecast F is smaller than M), and that the number of local minima and maxima in the terrain T is also smaller than M. As the number of local minima and maxima in the terrain data for Denmark (after removing all depressions and hills with volume less than 1 m^3, which is customary in flood computations) is 60 million, and the sea-level forecast contains 81 thousand values, both of the assumptions hold for the data for Denmark that we described above. It also requires the so-called *confluence assumption* [17] on the flow network that models how water flows on a raster terrain T. In such a network, a *flow direction* is assigned to each terrain cell u, which is a lower neighbor of u that water will flow to, and the confluence assumption intuitively says that flowing water quickly combines to larger flows at all scales. Formally, the *confluence parameter* γ is defined as follows: Let Q_s be a square of $\sqrt{s} \times \sqrt{s}$ cells and Q_s^3 be the square of $3\sqrt{s} \times 3\sqrt{s}$ cells that has Q_s in the center. Let $\gamma(Q_s)$ be the number of cells on the boundary of Q_s^3 reached from the boundary of Q_s when following flow directions without leaving Q_s^3. Refer to Figure 1. The confluence parameter is $\gamma = \max_{s>0} \max_{Q_s} \gamma(Q_s)$ where the maximum is taken over all squares Q_s of all sizes. The *confluence assumption* then states that γ is a constant independent of the size and resolution of the terrain model.

Our work is inspired by the work of Arge et al. [11] and Yang [21]. As described previously, Arge et al. [11] compute the flood risk by using the topological features of T encoded in the merge tree. However, this structure does not support efficient terrain updates. On the other hand, Yang [21] presented an I/O-efficient data structure for maintaining the contour tree of a dynamic TIN terrain. In Section 2 we show how a raster terrain T can be transformed into a TIN $T_\triangle$ while maintaining the topology pertaining to the dynamic sea-level flooding problem. In Section 3 we then show how the merge tree of T, which is needed when computing flood risk using the approach of Arge et al. [11], can be constructed from the contour tree of $T_\triangle$, which can be maintained under terrain updates using the data structure of Yang [21]. Note that standard techniques for triangulating a raster terrain have several issues, because they do not necessarily preserve flood paths and they do not allow the merge tree to be

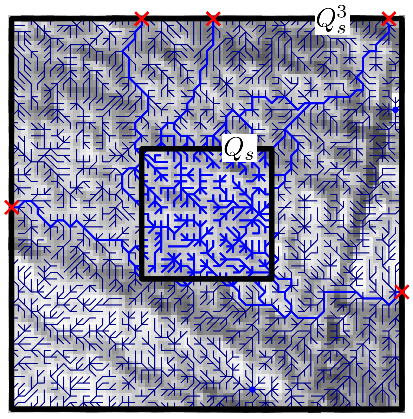


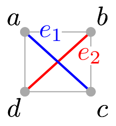
Figure 1 Confluence parameter. Hillshaded terrain shown in greyscale. Blue lines show the flow directions of the terrain. The thicker, lighter blue lines are flow directions reachable from Q_s^3. There are only 5 cells on the boundary of Q_s^3 that are reached from Q_s (red crosses), which implies that $\gamma(Q_s) = 5$ in this example.

constructed from the contour tree. Thus we believe that our transformation algorithm is of independent interest. In Section 4 we then describe the *sea-level flooding data structure* and show how the three operations are performed efficiently using the confluence assumption and the assumption on C and the number of minima and maxima.

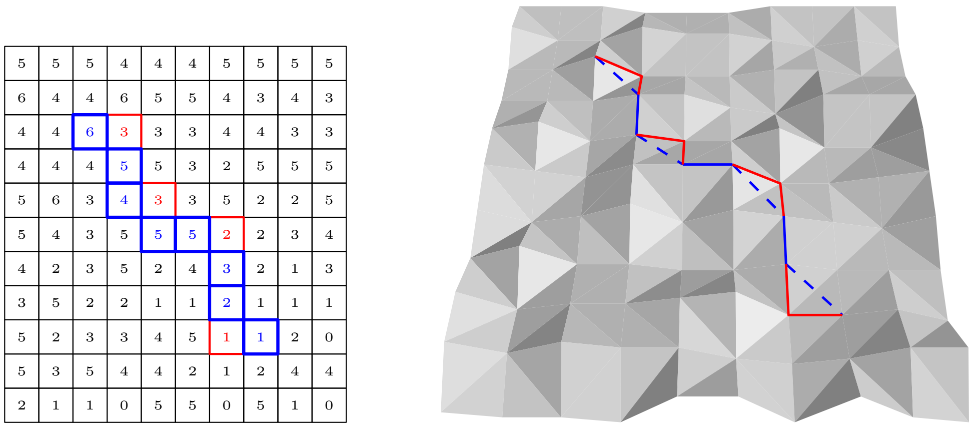
2 Reducing raster problem to TIN

Problem definition for TINs. As mentioned, a TIN consists of a planar triangulation of a set of N vertices in the plane along with a continuous height function $h_{T_\triangle}$ that is linear on each face of the triangulation. We assume that the boundary of the triangulation is a simple polygon, with the interior corresponding to land and the exterior corresponding to ocean. A subset of the vertices are *coastal vertices*, and like for raster terrains, the coastal vertices are partitioned into a set C of connected *coastal regions*. For two vertices u, v in $T_\triangle$, we say u and v are *adjacent* (or u is a *neighbor* of v) when there exists an edge in $T_\triangle$ that connects u and v. We define *flood path*, *flood height* and *flood source* on TINs as for raster terrains.

Transforming the raster terrain. The *incidence graph* G_T of a raster terrain T is a graph on the terrain cells of T, where two vertices $u_\triangle$ and $v_\triangle$ are connected in G_T if u and v are adjacent in T. If u and v are connected diagonally we call $u_\triangle$ a *diagonal neighbor* of $v_\triangle$, otherwise a *cardinal neighbor* of $v_\triangle$. Note that the natural planar embedding of G_T, where the vertex $u_\triangle$ corresponding to a cell u is placed at the center of u, is almost a triangulation, except for the intersecting edges corresponding to diagonal neighbors. We turn G_T into a



triangulation $T_\triangle$ by assigning $u_\triangle$ in $T_\triangle$ the same height as u in T, that is, $h_{T_\triangle}(u_\triangle) = h_T(u)$, and by removing a diagonal edge in $T_\triangle$ corresponding to each two-by-two square of terrain cells in T as follows: For each two-by-two square of cells a, b, c, d in clockwise order, there



5	5	5	4	4	4	5	5	5	5
6	4	4	6	5	5	4	3	4	3
4	4	6	3	3	3	4	4	3	3
4	4	4	5	5	3	2	5	5	5
5	6	3	4	3	3	5	2	2	5
5	4	3	5	5	5	2	2	3	4
4	2	3	5	2	4	3	2	1	3
3	5	2	2	1	1	2	1	1	1
5	2	3	3	4	5	1	1	2	0
5	3	5	4	4	2	1	2	4	4
2	1	1	0	5	5	0	5	1	0

Figure 2 Example of flood path preservation. On the triangulated terrain $T_\triangle$, each missing diagonal (dotted blue) is replaced by two cardinal edges (red).

are two intersecting incidence edges $e_1 = \{a, c\}$ and $e_2 = \{b, d\}$ in G_T. We triangulate the square by removing whichever edge has the higher midpoint, where the midpoint height of an edge $e = \{u, v\}$ is $s(e) = \frac{1}{2}(h_T(u) + h_T(v))$. If $s(e_1) = s(e_2)$, then we pick an arbitrary edge to remove.

▶ **Theorem 1.** *u is flooded through v in T if and only if $u_\triangle$ is flooded through $v_\triangle$ in $T_\triangle$.*

Proof. First, if $u_\triangle$ is flooded through $v_\triangle$ in $T_\triangle$, that is, there exists a flood path $p_\triangle : u_\triangle \rightsquigarrow v_\triangle$, then there is a flood path $p : u \rightsquigarrow v$ in T corresponding to $p_\triangle$, since each edge in $T_\triangle$ has a corresponding edge in G_T. Thus u is flooded through v in T.

Next, we show that if $p : u \rightsquigarrow v$ is a flood path in T, then there is a corresponding flood path $p_\triangle : u_\triangle \rightsquigarrow v_\triangle$ in $T_\triangle$. If no edge in G_T corresponding to adjacent cells in p was removed by the TIN construction, then we are done; $p_\triangle$ is the sequence of vertices that correspond to the cells in p. Otherwise, we show how to obtain $p_\triangle$ by replacing each edge e in G_T corresponding to adjacent cells in p that is not in $T_\triangle$ as follows: Since the TIN construction only removes diagonal edges, e is a diagonal edge connecting two raster cells a and c. Recall that the definition of flood path means that $\max\{h_T(a), h_T(c)\} \leq h_F(u)$. Let e' be the diagonal edge connecting cells b and d such that e and e' intersect and e' is included in $T_\triangle$. Then $s(e') \leq s(e)$, which implies that $\min\{h_T(b), h_T(d)\} \leq \max\{h_T(a), h_T(c)\} \leq h_F(u)$. Without loss of generality assume $h_T(b) \leq h_T(d)$, in which case we replace the edge e with the two edges $\{a, b\}$ and $\{b, c\}$. These edges are both non-diagonal edges and thus were not discarded when we triangulated T into $T_\triangle$, that is, we replace the adjacent cells a and c in p with cells abc. Refer to Figure 2 for an example. Since $h_T(b) \leq h_F(u)$, p is still a flood path after adding b. After handling all relevant edges this way, we have obtained a flood path p such that $p_\triangle$ is the sequence of vertices that correspond to the cells in p. ◀

3 Connecting topology of T and $T_\triangle$

3.1 Local topology and depressions

Flow directions, sinks, peaks, upper and lower sequences, and depressions. As mentioned previously, water flow on a raster terrain T can be modeled by assigning a *flow direction* on each terrain cell u in T, which is a lower neighbor of u that water will flow to from u.

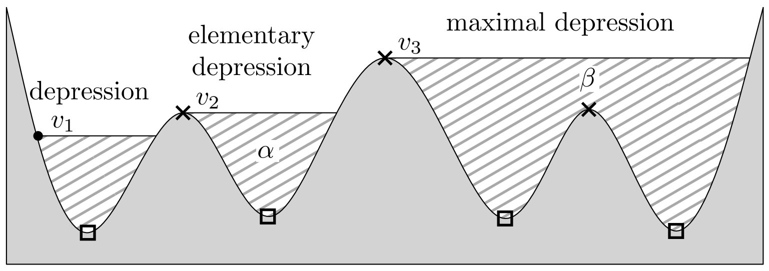


Figure 3 An example terrain seen from the side, showing cells v_1-v_3 along with depressions defined by the cells. The cell v_2 defines an elementary depression α. The cell v_3 defines a maximal, but not elementary, depression β.

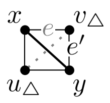
When a cell u does not have any lower neighbors, then u is not assigned a flow direction and we call it a *sink*. Similarly, a cell u is a *peak* if there is no neighbor of u that has higher elevation than u. We assume that no pair of adjacent cells have the same height. That is, a flow direction can be assigned to all terrain cells except sinks. The assumption can be removed using standard techniques [7].

As we traverse the neighbors of a cell u in T that is not a sink or peak in clockwise order, there are sequences of cells that are lower or higher than u. Each continuous sequence of lower (resp. higher) neighbors of u is called a *lower sequence* (resp. *upper sequence*) of u.

A raster terrain cell u defines a *depression* that is the maximal connected component of terrain cells containing u such that all cells v in the depression have $h(v) \leq h(u)$ [8]. Note that each depression contains at least one sink. A depression β_1 is *maximal* if every depression $\beta_2 \supset \beta_1$ contains strictly more sinks than β_1. If a maximal depression β contains exactly one sink, then we call β an *elementary depression*. Refer to Figure 3.

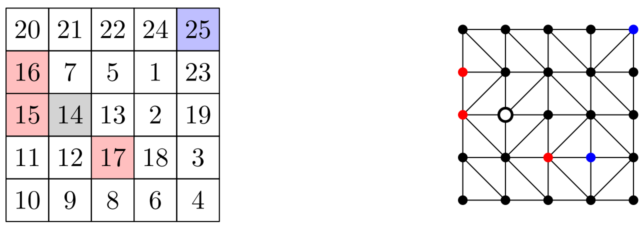
TIN construction preserves lower sequences, sinks, and depressions. On TINs, *flow direction*, *sink*, *peak*, *lower/upper sequence*, and *(maximal/elementary) depression* are defined as for rasters, but with the vertex adjacency defined by edges of the triangulation $T_\triangle$ rather than the incidence graph G_T. The following lemmas show that lower sequences, sinks, and depressions in T are preserved by our raster to TIN transformation. Note that the lemmas say nothing about peaks or upper sequences, as it is easy to verify that they are not preserved by the transformation. Refer to Figure 4 for an example.

▶ **Lemma 2.** *For any lower sequence J of any cell u in T, u has a neighbor v in J such that $u_\triangle$ and $v_\triangle$ are connected in $T_\triangle$.*



Proof. Pick any $v \in J$. If the edge $e = (u_\triangle, v_\triangle)$ is in $T_\triangle$, then we are done. Otherwise, e is removed by our TIN construction, so v must be a diagonal neighbor of u in T. Let $e' = (x, y)$ be the edge that was chosen to remain instead of e in the TIN construction. Then we have that x and y are the common neighbors of u and v and $s(e') \leq s(e)$, which implies that $\min\{h_T(x), h_T(y)\} \leq h_T(u)$. Without loss of generality assume $h_T(x) \leq h_T(y)$. Then x is a lower neighbor of u in J, and since x is a cardinal neighbor of u, $T_\triangle$ contains $\{u_\triangle, x_\triangle\}$. ◀

▶ **Corollary 3.** *A cell u has a lower neighbor in T if and only if $u_\triangle$ has a lower neighbor in $T_\triangle$.*



20	21	22	24	25
16	7	5	1	23
15	14	13	2	19
11	12	17	18	3
10	9	8	6	4

(a) Raster terrain. **(b)** Triangulated terrain.

Figure 4 Example showing that upper sequences and peaks are not necessarily preserved by our TIN construction. In (a) the raster terrain cell with height 14 has two upper sequences (red), but in the TIN terrain, the corresponding vertex (marked with a circle in (b)) has only one upper sequence. The raster terrain has only a single peak, the cell with height 25 (blue), but in the TIN terrain the vertex corresponding to the cell with height 18 is also a peak (blue).

▶ **Lemma 4.** *For any cell u in T, the cells in the depression β defined by u in T are in one-to-one correspondence with the vertices in the depression $\beta_\triangle$ defined by $u_\triangle$ in $T_\triangle$.*

Proof. First, we show that for each $v_\triangle \in \beta_\triangle$, v is in β: As $v_\triangle \in \beta_\triangle$, there is a path $p : u_\triangle \rightsquigarrow v_\triangle$ in $T_\triangle$ of vertices with height below $h_{T_\triangle}(u)$. This path corresponds to a path in T (since the TIN construction only removes edges) and therefore $v \in \beta$.

To show that the cells in β correspond to a subset of the vertices in $\beta_\triangle$, it suffices to show that the vertices corresponding to cells in β are connected in $T_\triangle$, since $\beta_\triangle$ is a maximal connected component of vertices with height $\leq h_{T_\triangle}(u_\triangle)$ (and each vertex in $T_\triangle$ has the same height as it has in T). To show this we proceed by induction in the list of cells u in T ordered by height $h_T(u)$. Suppose that for any cell u' in T with $h_T(u') < h_T(u)$, the depression defined by u' is connected in $T_\triangle$. If u is a sink, then $\beta = \{u\}$ which is trivially connected in $T_\triangle$. Otherwise, we consider the set $L = \beta \setminus \{u\}$ and make the following two observations for each connected component β' of L.

- Since β' is a depression in T defined by the highest cell in β', it follows by induction that β' is connected in $T_\triangle$.
- It is easy to see that β' contains a lower neighbor v of u and thus contains all cells in the lower sequence J containing v; by Lemma 2, it follows that $u_\triangle$ is adjacent in $T_\triangle$ to a vertex that corresponds to a cell in J.

From this it follows that $u_\triangle$ is connected to all of β' in $T_\triangle$, so β is connected in $T_\triangle$. ◀

▶ **Corollary 5.** *A depression β is a maximal (resp. elementary) depression in T if and only if $\beta_\triangle$ is a maximal (resp. elementary) depression in $T_\triangle$.*

3.2 Merge trees and contour trees

Merge tree of raster T. As mentioned, the merge tree $\mathcal{M}$ of a raster terrain T is a rooted tree that represents the nested topology of the maximal depressions [14]. Each node in $\mathcal{M}$ represents a maximal depression in T, and we refer to the maximal depression represented by a merge tree node x as β_x. Each elementary depression is represented by a leaf node, and a node y is the parent of a node x when $\beta_x \subset \beta_y$ and there exists no maximal depression β_z such that $\beta_x \subset \beta_z \subset \beta_y$.

Now consider sweeping the raster terrain with a plane of height ℓ from $-\infty$ to ∞ while maintaining the set of depressions that consist of cells with elevation less than or equal to ℓ. If the number of depressions decreases when the sweeping plane crosses a cell u, then u is called a *negative saddle*; it is easy to see that a negative saddle u has at least two lower

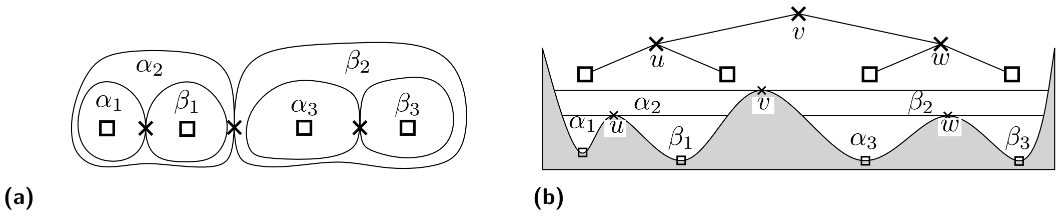


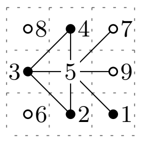
Figure 5 An example terrain with negative saddle cells u, v and w. (a) Terrain seen from above. Sinks are marked with a square and saddles are marked with a cross. The maximal depressions α_1, β_1, α_3, and β_3 are elementary. (b) Terrain seen from the side along with the merge tree $\mathcal{M}$.

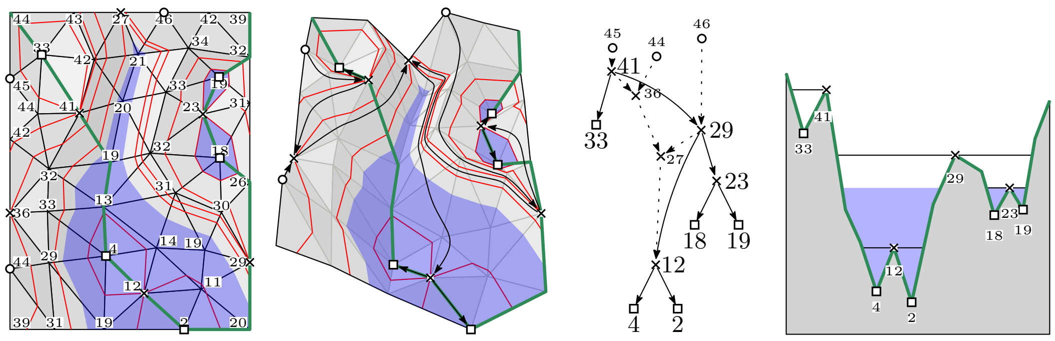
sequences. Then note that for any two maximal depressions β_x and β_y whose corresponding nodes x and y in $\mathcal{M}$ share the same parent, there exists a negative saddle where β_x and β_y are merged. Using this we associate a terrain cell to each merge tree node as follows: To a leaf node x we associate the sink in the elementary depression β_x; to an internal node x we associate the negative saddle where the maximal depressions of its children are merged. Refer to Figure 5.

Contour tree of TIN $T_\triangle$. For $\ell \in \mathbb{R}$, the *ℓ-level set* of $T_\triangle$ is defined to consist of points $x \in \mathbb{R}^2$ with $h_{T_\triangle}(x) = \ell$. A *contour* of $T_\triangle$ is a connected component of a level set of $T_\triangle$ [3]. We define a *down-contour* of $u_\triangle$ as any contour with elevation $h_{T_\triangle}(u) - \epsilon$, for a value ϵ smaller than the height difference between any pair of vertices, such that the contour intersects an edge incident to $u_\triangle$ [3]. Similarly, we define an *up-contour* as a contour with elevation $h_{T_\triangle}(u) + \epsilon$ that intersects an edge incident to $u_\triangle$.

Traversing the neighbors of a vertex $u_\triangle$ in clockwise order, we say that $u_\triangle$ is a *saddle* in $T_\triangle$ if there are multiple sequences of lower neighbors of $u_\triangle$ disconnected by higher neighbors of $u_\triangle$ [3]. For simplicity we assume that every saddle has exactly two such sequences of lower neighbors. This assumption can be removed [16]. We say that a vertex $u_\triangle$ is *critical* if $u_\triangle$ is a peak, sink, or saddle. If a saddle $u_\triangle$ has one up-contour (down-contour) and two down-contours (up-contours) then $u_\triangle$ is called a *negative (positive) saddle*. For any two contours C_1 and C_2 with level ℓ_1 and ℓ_2, respectively, we say C_1 and C_2 are *equivalent* if they belong to the same connected component of $\Gamma = \{x \in \mathbb{R}^2 \mid \ell_1 \leq h_{T_\triangle}(x) \leq \ell_2\}$ that does not contain any critical vertex. When sweeping $T_\triangle$ with a plane from $-\infty$ to ∞, an equivalence class of contours starts and ends at critical vertices. That is, the contours deform continuously as the sweeping plane changes its height, but the number of contours does not change as long as the plane varies between two critical vertices. A contour appears and disappears at a sink and a peak, respectively. Two contours merge into one at a negative saddle, and a contour splits into two at a positive saddle.

The *contour tree* $\mathcal{A}$ of a TIN $T_\triangle$ is a tree on the critical vertices in $T_\triangle$ that encodes the topological changes of the contours [3, 21]. Two critical vertices $u_\triangle, v_\triangle$ are connected in $\mathcal{A}$ if and only if an equivalence class of contours starts at $u_\triangle$ and ends at $v_\triangle$. That is, an edge $(u_\triangle, v_\triangle)$ in $\mathcal{A}$ represents the equivalence class of contour that appears at $u_\triangle$ and disappears at $v_\triangle$. Refer to Figure 6. Note that the contour tree is not a rooted tree; an internal node has two lower (higher) neighbors and one higher (lower) neighbor if it corresponds to a negative (positive) saddle.



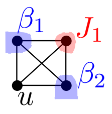


(a) From above. **(b)** In perspective. **(c)** Merge tree. **(d)** Depression structure.

Figure 6 Example TIN terrain. Sinks are marked with squares, peaks with circles, and saddles with crosses. (a, b) Everything below $\ell = 23$ is marked as blue. Contours defined by saddle vertices are marked with red lines. (b) Edges of the contour tree $\mathcal{A}$ are shown as arcs pointing downwards in height. (c) Merge tree derived from the contour tree (dashed), representing how the maximal depressions in (d) are nested.

TIN construction preserves negative saddles. Note that for raster terrains we have only defined negative saddles, and not *saddles* or *positive saddles*. The reason is, as shown in the example in Figure 6, that a cell in T with multiple lower and upper sequences can become a regular vertex in $T_\triangle$ after our transformation. However, we can show that negative saddles are preserved by the transformation. To do so we need the following lemma.

▶ **Lemma 6.** *For any negative saddle u in a raster terrain T that merges two depressions β_1 and β_2, let J_1 and J_2 be the two higher sequences of u that separate β_1 and β_2 in a clockwise traversal of the neighbors of u. Then J_1 and J_2 each contain a cardinal neighbor of u.*



Proof. Assume for contradiction that a higher sequence separating β_1 and β_2 does not contain a cardinal neighbor of u, that is, that it consists of a single diagonal neighbor. This implies that a cell in β_1 has a diagonal neighbor in β_2, which violates the definition of depressions as maximal connected components. ◀

Using Lemma 6 we can prove the following.

▶ **Lemma 7.** *A cell u is a negative saddle in T if and only if $u_\triangle$ is a negative saddle in $T_\triangle$.*

Proof. First, we show that if u is a negative saddle in T, then $u_\triangle$ is a negative saddle in $T_\triangle$. Let β_1 and β_2 be the two depressions that merge at u. By Lemma 2, $u_\triangle$ is connected in $T_\triangle$ to a lower neighbor ${u^1}_\triangle$ in β_1 and a lower neighbor ${u^2}_\triangle$ in β_2. By Lemma 6, u has two higher cardinal neighbors v^1 and v^2 separating u^1 and u^2 in a clockwise traversal of the neighbors of u. Since the construction of $T_\triangle$ only removes diagonal edges, $u_\triangle$ is connected to both ${v^1}_\triangle$ and ${v^2}_\triangle$ in $T_\triangle$. Thus $u_\triangle$ has four alternating lower and higher neighbors ${u^1}_\triangle, {v^1}_\triangle, {u^2}_\triangle, {v^2}_\triangle$ in clockwise order, so $u_\triangle$ is a saddle in $T_\triangle$. By Lemma 4, β_1 and β_2 are preserved in $T_\triangle$. Thus the down-contour of $u_\triangle$ intersecting ${u^1}_\triangle$ is distinct from the down-contour intersecting ${u^2}_\triangle$, so $u_\triangle$ is a negative saddle.

Next, we show that if u is not a negative saddle in T, then $u_\triangle$ is not a negative saddle in $T_\triangle$. If the number of lower sequences of u is less than 2, then it is easy to see that $u_\triangle$ is not a saddle in $T_\triangle$. If the number of lower sequences of u is at least 2, then these lower sequences must be from the same depression β. By Lemma 4, β is connected in $T_\triangle$ and thus $u_\triangle$ is not a negative saddle. ◀

Constructing merge tree $\mathcal{M}$ of T from contour tree $\mathcal{A}$ of $T_\triangle$. We now show how the merge tree $\mathcal{M}$ of T can be constructed from the contour tree $\mathcal{A}$ of the TIN $T_\triangle$ obtained after applying our TIN construction to T. From Corollary 3 and Lemma 7 it follows that the nodes of $\mathcal{M}$, which correspond to the sinks and negative saddles of T, are encoded in $\mathcal{A}$. In Appendix A.1 we show (Lemma 12) that if v is the highest node on the path between u and v in $\mathcal{A}$, then there is a path between u and v in $T_\triangle$ where v is the highest vertex. The following lemma is then the key to constructing the edges of $\mathcal{M}$ from $\mathcal{A}$.

▶ **Lemma 8.** *For two distinct nodes $u_\triangle$ and $v_\triangle$ in the contour tree $\mathcal{A}$ of $T_\triangle$ that correspond to negative saddles or sinks, $\beta_{u_\triangle} \subset \beta_{v_\triangle}$ if and only if there is a path $p_\mathcal{A} : u_\triangle \rightsquigarrow v_\triangle$ in $\mathcal{A}$ such that all vertices in $p_\mathcal{A}$ have height less than or equal to the height of $v_\triangle$.*

Proof. Suppose $\beta_{u_\triangle} \subset \beta_{v_\triangle}$. Then there is a path $p_\triangle : u_\triangle \rightsquigarrow v_\triangle$ in $T_\triangle$ such that all vertices in $p_\triangle$ have height less than or equal to $h_{T_\triangle}(v_\triangle)$. The edges and vertices in $\mathcal{A}$ corresponding to all contours through points on $p_\triangle$ in $T_\triangle$ form a connected subtree of $\mathcal{A}$. Thus there is a path $p_\mathcal{A} : u_\triangle \rightsquigarrow v_\triangle$ in $\mathcal{A}$ with height less than or equal to the height of $v_\triangle$.

Now, suppose $p_\mathcal{A}$ is a path in $\mathcal{A}$ from $u_\triangle$ to $v_\triangle$ such that $v_\triangle$ is the highest vertex in $p_\mathcal{A}$. By Lemma 12 there is a path $p_\triangle : v_\triangle \rightsquigarrow u_\triangle$ in $T_\triangle$ such that the highest vertex on $p_\triangle$ is $v_\triangle$. This implies that $u_\triangle$ is contained in the depression defined by $v_\triangle$, so $\beta_{u_\triangle} \subseteq \beta_{v_\triangle}$. ◀

▶ **Theorem 9.** *The merge tree $\mathcal{M}$ of T can be constructed from the contour tree $\mathcal{A}$ of $T_\triangle$.*

Proof. For each negative saddle $u_\triangle$ in $\mathcal{A}$ we define a *key descendant of* $u_\triangle$ for each child $v_\triangle$ of $u_\triangle$ as follows: If $v_\triangle$ is a sink or a negative saddle, then $v_\triangle$ is the key descendant of $u_\triangle$. Otherwise, it is a positive saddle, and the key descendant is found by following a downward path from $v_\triangle$ in $\mathcal{A}$ until reaching a negative saddle or sink; since a vertex in $\mathcal{A}$ that is not a negative saddle or sink has exactly one lower neighbor, this downward path following lower neighbors until encountering a negative saddle or sink is unique. Since $u_\triangle$ has exactly two children, it has two key descendants.

To construct $\mathcal{M}$ we start with a forest containing all the sinks of $\mathcal{A}$ (leaves of $\mathcal{M}$), and we maintain a union-find data structure that maps each vertex u of $\mathcal{M}$ to the root of u in the forest $\mathcal{M}$ constructed so far. Next, we insert the negative saddles of $\mathcal{A}$ (internal nodes) into $\mathcal{M}$ in increasing order of height using the union-find data structure as follows: When processing a negative saddle $u_\triangle$ with key descendants $v^1_\triangle$ and $v^2_\triangle$, we query the union-find data structure to obtain $v^1 = \textsc{Find}(v^1_\triangle)$ and $v^2 = \textsc{Find}(v^2_\triangle)$. From Lemma 8, v^1 and v^2 are the children of u in $\mathcal{M}$, so we insert u into $\mathcal{M}$ with children v^1 and v^2. We then update the union-find structure using $\textsc{Union}(v^1, v^2)$. When we have processed all negative saddles in this way, we have constructed $\mathcal{M}$. ◀

4 Sea-level flooding data structure

We are now ready to describe our sea-level flooding data structure. Intuitively, our structure maintains the result of a flood computation for a forecast F on a terrain T by a *flood instance* I_F, which stores for each sink cell u of T the flood source of u. When answering a query the flood height of any cell can then be computed from I_F using the observation that if a non-sink cell v is flooded by F, then the flood source of v is the same as the flood source of any sink cell u in the depression defined by v, and it is easy to compute the flood height of v from such a sink u [11]. Furthermore, we maintain the contour tree $\mathcal{A}$ of $T_\triangle$ using the data structure of Yang [21]. As required by this data structure, our data structure also maintains a so-called *descent tree* $\Pi_\downarrow$ and a so-called *ascent tree* $\Pi_\uparrow$ that store descending and ascending connectivity on $T_\triangle$, respectively. More precisely, the descent (ascent) tree is an I/O-efficient data structure that maintains a forest on the vertices in $T_\triangle$, where each vertex is connected to one of its lower (upper) neighbors in $T_\triangle$, and where each root in the forest corresponds to a sink (peak). The descent (ascent) tree $\Pi_\downarrow$ ($\Pi_\uparrow$) supports finding for a cell u, a sink (peak) that can be reached from u by a decreasing (increasing) path in $O(\log_B N)$ I/Os. Thus, $\Pi_\downarrow$ can be queried to find a sink in the depression defined by v in $O(\log_B N)$ I/Os. The ascent and descent trees also support the following operations in $O(\log_B^2 N)$ I/Os: Disconnect the subtree rooted at u from its parent, and link a root u to a vertex v. For details we refer to [21].

While Yang described how to maintain $\mathcal{A}$, $\Pi_\downarrow$ and $\Pi_\uparrow$ when updating the height of a single vertex in $O(\log_B^2 N)$ I/Os [21], we need to update heights in a square Q_B of B cells in the same bound. Yang classifies the possible changes to $\mathcal{A}$ as a result of changing the height of a vertex of $T_\triangle$ as either adding a sink or peak (*birth event*), removing a sink or peak (*death event*), or reordering saddles (*interchange event*). When updating a number of vertices in a region Q_B of $\sqrt{B}$ by $\sqrt{B}$ cells, birth and death events are conceptually simple to handle, as the involved sink or peak is either in Q_B or adjacent to a vertex in Q_B. On the other hand, an interchange event involves a saddle u in Q_B and a saddle v adjacent to u in $\mathcal{A}$, but not necessarily in the vicinity of Q_B in $T_\triangle$. Yang [21] handles such an interchange event by querying $\Pi_\downarrow$ and $\Pi_\uparrow$ with the neighbors of u and v in $T_\triangle$. However, this is a problem for our data structure, as updating the heights of B vertices in Q_B can cause $\Theta(B)$ interchange events and thus $\Theta(B)$ queries to $\Pi_\downarrow$ and $\Pi_\uparrow$, which would require $\Theta(B \log_B N)$ I/Os. In Appendix A.2 we describe (Lemma 13) how to answer the queries to $\Pi_\downarrow$ and $\Pi_\uparrow$ without I/Os, by maintaining in addition a set L of vertices of $\Pi_\downarrow$ and $\Pi_\uparrow$ in main memory. In this way, $\mathcal{A}$ can be updated without using I/Os (since we have assumed that the critical vertices, and thus $\mathcal{A}$, fits in memory).

Yang also showed how to augment his data structure such that all vertices in $T_\triangle$ are represented in $\mathcal{A}$ [21]. Intuitively, a vertex v is added to the edge representing the contour through v. The augmented data structure can be maintained in the same bounds as described above [21]. In our sea-level flooding data structure, we similarly augment the contour tree of $T_\triangle$ with the so-called *coastal minima* of $T_\triangle$ that are the coastal vertices u with no lower neighbors inside their coastal region, that is, a coastal vertex u is a coastal minimum if there is no vertex $v \in C(u)$ adjacent to u that is lower than u. We denote by $\mathcal{A}^+$ this augmented contour tree extended to represent the coastal minima. In Appendix A.3 (Lemma 14) we show that $\mathcal{A}^+$ contains enough information to determine which sinks are flooded by a forecast F. More precisely, we show that if a coastal vertex u in coastal region R floods a terrain vertex v, then there is a coastal minimum w in R that also floods v. Furthermore, we show that the number of coastal minima is bounded by the number of coastal regions $|R|$ and the number of critical vertices in $T_\triangle$.

In summary, our sea-level flooding data structure consists of the following components:

- Contour tree $\mathcal{A}^+$ containing the sinks, peaks, saddles and coastal minima of $T_\triangle$;
- Descent tree $\Pi_\downarrow$ and ascent tree $\Pi_\uparrow$ on $T_\triangle$, as well as a set L of vertices of $\Pi_\downarrow$ and $\Pi_\uparrow$;
- Terrain T, forecast F and flood instance I_F.

Space. Recall that we assume that the number of sinks, peaks and saddles in $T_\triangle$, as well as the number of coastal regions, is smaller than M. Thus $\mathcal{A}^+$ and L fit in main memory. As I_F stores a coastal region for each sink of $T_\triangle$, it also fits in main memory. By assumption, F fits in main memory. Finally, T and the descent and ascent trees are stored in external memory where they use $O(N/B)$ blocks of space [21]. It is easy to see that our data structure can be constructed in $O(\text{Sort}(N))$ I/Os [21].

Flood-Height(Q_B) query. To compute the flood height for all cells in Q_B, we first associate each cell $v \in Q_B$ with a sink in the depression defined by v as follows: Let Q_B^3 be the square of $3\sqrt{B} \times 3\sqrt{B}$ cells that has Q_B in the center. For each $v \in Q_B$, we follow flow directions from v until reaching either the boundary of Q_B^3 or a sink u. By assumption, the number of times we reach a cell on the boundary of Q_B^3 is constant. For each such cell w, we query $\Pi_\downarrow$ to find a sink u in the depression defined by w, and we associate u with the cell $v \in Q_B$ that reached w when following flow directions. After this way having associated each $v \in Q_B$ with a sink u in the depression defined by v, the flood source of each $v \in Q_B$ can be found using I_F as discussed above.

Following flow directions can be done by loading the query cells of T in Q_B^3 into main memory and computing the flow directions in $O(1)$ I/Os. Making the $O(1)$ queries to $\Pi_\downarrow$ requires $O(\log_B N)$ I/Os [21]. Then, the flood sources can be found using I_F without using any I/Os, since I_F is stored in memory. Thus, FLOOD-HEIGHT(Q_B) requires $O(\log_B N)$ I/Os in total.

Forecast-Update(F). To update the flood instance I_F given a new forecast F, we first construct $\mathcal{M}$ from $\mathcal{A}^+$ using Theorem 9. Then for each coastal minimum v in a coastal region R, we compute $h = h_F(v) - h_T(v)$. If $h > 0$, it means that v is flooded by the forecast value $F(R) = h_F(v)$. Note that this also means that all sinks in the depression defined by v are flooded. As $\mathcal{A}^+$ contains the coastal minimum v, we can then use $\mathcal{A}^+$ to find one of these sinks u by following a decreasing path in $\mathcal{A}^+$ from v until reaching a sink as follows: At a negative saddle w, we pick an arbitrary lower neighbor of w, and at a vertex w that is not a negative saddle or sink (i.e. a peak, positive saddle, or coastal minimum), w has a unique lower neighbor in $\mathcal{A}^+$. Eventually we reach a sink u and since we have followed a strictly decreasing path from v to u in $\mathcal{A}^+$ it follows by Lemma 8 that u is in the depression defined by v. Note that u is also a node in $\mathcal{M}$, and that in any instance of the sea-level flooding problem in which u is flooded with forecast value $F(R)$, the other sinks in the depression defined by v will also be flooded. After having found at most one flooded sink for each coastal minimum in this way, we can identify the remaining flooded sinks in the terrain using the algorithm by Arge et al. [11] that takes as input a list of forecast values for sinks in a merge tree $\mathcal{M}$ of T and computes the flood source and flood height for all sinks in $\mathcal{M}$. This in turn gives us the new flood instance I_F.

After the forecast F is read into memory using $O(\text{Scan}(F))$ I/Os, no further I/Os are required for FORECAST-UPDATE, since $\mathcal{A}^+$, $\mathcal{M}$ and I_F fit in main memory.

Height-Update(Q_B, U). To update the heights of the cells in Q_B we need to update $\Pi_\downarrow$, $\Pi_\uparrow$, $\mathcal{A}^+$ and I_F.

Intuitively, we update $\Pi_\downarrow$ and $\Pi_\uparrow$ by removing subtrees containing the vertices whose heights were updated, reconstructing a new forest corresponding to the new descending or ascending connectivity after the update, and then linking the forest into the structure. More precisely, let Q_B^3 and Q_B^5 be the squares of $3\sqrt{B} \times 3\sqrt{B}$ and $5\sqrt{B} \times 5\sqrt{B}$ cells, respectively, such that Q_B^3 and Q_B^5 both have Q_B in the center. We describe how the descent tree $\Pi_\downarrow$ is updated; the ascent tree can be updated in an analogous way. First, we will disconnect a number of edges on the boundary of Q_B^3 to ensure that no tree in $\Pi_\downarrow$ contains both a vertex inside Q_B and a vertex outside Q_B^5. In other words, we isolate a set of vertices V in $\Pi_\downarrow$ such that $Q_B \subset V \subset Q_B^5$, as follows: First we disconnect edges so that no vertex in Q_B is connected to a vertex on the boundary of Q_B^3 by following edges of $\Pi_\downarrow$ from each vertex in Q_B until reaching either a sink inside Q_B^3 or a vertex on the boundary of Q_B^3. By the confluence assumption, the number of times we reach a vertex on the boundary of Q_B^3 is constant. For each such vertex w, we disconnect w from its parent in $\Pi_\downarrow$. Next, we disconnect edges so that no vertex outside Q_B^5 is connected to a vertex on the boundary of Q_B^3 by following edges of $\Pi_\downarrow$ from all cells on the boundary of Q_B^5 towards Q_B^3 until we reach a sink or the boundary of Q_B^3. It is easy to show that the number of times we reach the boundary of Q_B^3 is a constant, by covering each of the four sides of Q_B^5 with five translated copies of Q_B, and bounding the number of times each copy can reach the boundary of Q_B^3 using the confluence parameter. As previously, we disconnect each vertex w reached on the boundary of Q_B^3 from its parent in $\Pi_\downarrow$. Let u in Q_B and v outside Q_B^5 be vertices that were in the same tree of $\Pi_\downarrow$ before we disconnected edges on the boundary of Q_B^3, and let w be the lowest common ancestor of u and v in $\Pi_\downarrow$ at that time. If w is outside Q_B^3, then u is no longer connected to w as we disconnected the first edge on the path from u to w that is on the boundary of Q_B^3. If w is inside Q_B^3, a similar argument shows that v is no longer connected to w. Thus we have isolated a set V of $O(B)$ vertices in $\Pi_\downarrow$ such that $Q_B \subseteq V \subseteq Q_B^5$. We can then simply reconstruct new subtrees for V in $\Pi_\downarrow$ according to the new descending connectivity resulting from the height updates in Q_B and link back the disconnected edges on the boundary of Q_B^3 into $\Pi_\downarrow$.

After updating $\Pi_\downarrow/\Pi_\uparrow$ we update $\mathcal{A}^+$ with the new topology of the terrain using the update algorithm in [21], and we use Forecast-Update(F) to update the flood instance I_F.

As we disconnect and reconnect $O(1)$ edges of $\Pi_\downarrow$ and $\Pi_\uparrow$, updating the heights of cells in Q_B can be done using $O(\log_B^2 N)$ I/Os [21]. Updating $\mathcal{A}^+$ requires querying $\Pi_\downarrow$ and $\Pi_\uparrow$ with neighbors of saddle vertices; as discussed previously (Lemma 13), this part can be handled without I/Os since $\mathcal{A}^+$ and L fit in memory, where L contains the vertices of $\Pi_\downarrow$ and $\Pi_\uparrow$ required to perform the update of $\mathcal{A}^+$. Since F is already in main memory, Forecast-Update does not require any I/Os. Thus, Height-Update(Q_B, U) requires $O(\log_B^2 N)$ I/Os in total.

▶ **Theorem 10.** *Given a terrain of N cells, a partition of the coastal cells of the terrain into a set of coastal regions C, and a forecast $F : C \to \mathbb{R}$, for which the following assumptions hold:*
- *the confluence parameter γ is constant,*
- *the number of local minima and maxima is smaller than M,*
- *$|C|$ is smaller than M,*

a data structure for the dynamic sea-level flooding problem can be constructed in $O(\mathrm{Sort}(N))$ I/Os using $O(\frac{N}{B})$ blocks of space, such that
- *Flood-Height(Q_B) can be performed in $O(\log_B N)$ I/Os,*
- *Forecast-Update(F) can be performed in $O(\mathrm{Scan}(F))$ I/Os, and*
- *Height-Update(Q_B, U) can be performed in $O(\log_B^2 N)$ I/Os.*

References

1 Pankaj K. Agarwal, Lars Arge, Gerth Stølting Brodal, and Jeffrey S. Vitter. I/O-efficient Dynamic Point Location in Monotone Planar Subdivisions. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 11–20, 1999.

2 Pankaj K Agarwal, Lars Arge, and Ke Yi. I/O-efficient batched union-find and its applications to terrain analysis. *ACM Trans. Algorithms*, 7(1):11, 2010.

3 Pankaj K. Agarwal, Thomas Mølhave, Morten Revsbæk, Issam Safa, Yusu Wang, and Jungwoo Yang. Maintaining Contour Trees of Dynamic Terrains. In *31st International Symposium on Computational Geometry*, volume 34, pages 796–811, 2015.

4 Alok Aggarwal and Jeffrey Vitter. The Input/output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

5 Cici Alexander, Lars Arge, Peder Klith Bøcher, Morten Revsbæk, Brody Sandel, Jens-Christian Svenning, Constantinos Tsirogiannis, and Jungwoo Yang. Computing River Floods Using Massive Terrain Data. In *Geographic Information Science*, pages 3–17. Springer International Publishing, 2016.

6 Lars Arge. External memory data structures. In *Handbook of massive data sets*, pages 313–357. 2002.

7 Lars Arge, Jeffrey S Chase, Patrick Halpin, Laura Toma, Jeffrey S Vitter, Dean Urban, and Rajiv Wickremesinghe. Efficient Flow Computation on Massive Grid Terrain Datasets. *GeoInformatica*, 7(4):283–313, 2003.

8 Lars Arge, Mathias Rav, Sarfraz Raza, and Morten Revsbæk. I/O-Efficient Event Based Depression Flood Risk. In *Proc. 9th Workshop on Algorithm Engineering and Experiments*, pages 259–269. SIAM, 2017.

9 Lars Arge and Morten Revsbæk. I/O-efficient contour tree simplification. In *International Symposium on Algorithms and Computation*, pages 1155–1165. Springer, 2009.

10 Lars Arge, Morten Revsbæk, and Norbert Zeh. I/O-efficient computation of water flow across a terrain. In *Proceedings of the 26th annual Symposium on Computational Geometry*, pages 403–412. ACM, 2010.

11 Lars Arge, Yujin Shin, and Constantinos Tsirogiannis. Computing Floods Caused by Non-Uniform Sea-Level Rise. In *Proc. 20th Workshop on Algorithm Engineering and Experiments*, pages 97–108. SIAM, 2018.

12 Lars Arge, Laura Toma, and Jeffrey Scott Vitter. I/O-efficient Algorithms for Problems on Grid-based Terrains. *Journal of Experimental Algorithmics*, 6:1, 2001.

13 Danish Geodata Agency. Elevation Model of Denmark:Terræn (0.4 meter grid). http://eng.gst.dk, 2015.

14 Andrew Danner. *I/O Efficient Algorithms and Applications in Geographic Information Systems*. PhD thesis, Department of Computer Science, Duke University, 2006.

15 Andrew Danner, Thomas Mølhave, Ke Yi, Pankaj K Agarwal, Lars Arge, and Helena Mitásová. TerraStream: from Elevation Data to Watershed Hierarchies. In *Proc. 15th Annual ACM International Symposium on Advances in Geographic Information Systems*, page 28. ACM, 2007.

16 Herbert Edelsbrunner, John Harer, and Afra Zomorodian. Hierarchical morse-smale complexes for piecewise linear 2-manifolds. *Discrete and computational Geometry*, 30(1):87–107, 2003.

17 Herman J. Haverkort and Jeffrey Janssen. Simple I/O-efficient Flow Accumulation on Grid Terrains. *CoRR*, abs/1211.1857, 2012. URL: `http://arxiv.org/abs/1211.1857`.

18 Robert E Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

19 Robert Endre Tarjan and Uzi Vishkin. Finding biconnected componemts and computing tree functions in logarithmic parallel time. In *25th Annual Symposium on Foundations of Computer Science*, pages 12–20. IEEE, 1984.

20 Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Foundations and Trends® in Theoretical Computer Science*, 2(4):305–474, 2008.

21 Jungwoo Yang. *Efficient Algorithms for Handling Massive Terrains.* PhD thesis, Department of Computer Science, University of Aarhus, 2015.

A Appendices

A.1 Connecting paths in $T_\triangle$ and $\mathcal{A}$

In this section we show an important relation between paths along the edges of a TIN $T_\triangle$ and paths along the edges of the contour tree $\mathcal{A}$ of $T_\triangle$. Recall that we have defined a *flood path* from a coastal vertex u to a vertex v as a path $p : u \rightsquigarrow v$ along the edges of $T_\triangle$ such that no vertex in v has height greater than $h_F(u)$. Thus, one way of showing that flood paths are preserved is by showing the more general statement that paths that stay below some height level ℓ are preserved. The following lemmas show that a path in $\mathcal{A}$ that stays below ℓ corresponds to a path along the edges of $T_\triangle$ that stays below ℓ, and vice versa.

▶ **Lemma 11.** *For any critical vertex $v_\triangle \in T_\triangle$, sink $u_\triangle \in T_\triangle$ and strictly decreasing path $p_\triangle : v_\triangle \rightsquigarrow u_\triangle$ along the edges of $T_\triangle$, there is a corresponding path $p : v_\triangle \rightsquigarrow u_\triangle$ in $\mathcal{A}$ that is strictly decreasing in height.*

Proof. Consider lowering a plane from $\ell = h_{T_\triangle}(v_\triangle)$ to $h_{T_\triangle}(u_\triangle)$. Since $p_\triangle$ is strictly decreasing in height, the plane intersects $p_\triangle$ at a single point $x(\ell)$ for all ℓ. The path p consists of edges of $\mathcal{A}$ corresponding to the contours containing $x(\ell)$ for all ℓ. ◀

▶ **Lemma 12.** *For any pair of nodes $v, w \in \mathcal{A}$ such that v is the highest vertex on the path $p : v \rightsquigarrow w$ in $\mathcal{A}$, there exists a path $p_\triangle : v \rightsquigarrow w$ along the edges of $T_\triangle$ such that the highest vertex on $p_\triangle$ is v.*

Proof. First, we observe that for each vertex $v \in \mathcal{A}$ and down-contour c of v, there exists a strictly decreasing path $\pi_\triangle(v, c)$ in $T_\triangle$ from v through c to a sink u. By applying Lemma 11, we obtain a path $\pi_{\mathcal{A}}(v, c) : v \rightsquigarrow u$ in $\mathcal{A}$. We show how to find a path in $T_\triangle$ between any pair $v, w \in \mathcal{A}$ using the paths $\pi_\triangle$, as follows: We proceed by induction in the list of contour tree nodes sorted in increasing height order. Fix $v \in \mathcal{A}$ and suppose that, for all pairs $v', w \in \mathcal{A}$ such that $h_{T_\triangle}(v') < h_{T_\triangle}(v)$ and v' is the highest vertex on the path from v' to w in $\mathcal{A}$, there exists a path from v' to w in $T_\triangle$ having v' as the highest vertex. We have to show that for any $w \in \mathcal{A}$ and path $p : v \rightsquigarrow w$ where v is the highest vertex on p, there is a corresponding path along the edges of $T_\triangle$ having v as the highest vertex. Let c be the down-contour represented by the edge of p incident to v, and let u be the sink such that $\pi(v, c)$ connects v to u in $T_\triangle$. By induction, u is connected to w in $T_\triangle$ by a path $p_\triangle$ such that w is the highest vertex. By concatenating $\pi_\triangle(v, c) : u \rightsquigarrow w$ and the reverse of $p_\triangle : w \rightsquigarrow u$, we obtain a path from v to w such that v is the highest vertex on the path. ◀

A.2 Updating $\mathcal{A}$ without using I/Os

In this section we describe how updates to $\mathcal{A}$ in HEIGHT-UPDATE can be handled without I/Os. First we give a brief description of how the data structure of Yang [21] handles updates to $\mathcal{A}$. Then we describe how our sea-level flooding data structure avoids I/Os when the contour tree is updated.

Yang describes the *dynamic forest* data structure that is an I/O-efficient data structure used to store the descent tree $\Pi_\downarrow$ and ascent tree $\Pi_\uparrow$. The data structure represents a forest of rooted trees, and for a forest of N vertices the following operations are supported: Returning the *root* of a vertex u in $O(\log_B N)$ I/Os and *linking* a root u to a vertex v or *disconnecting* a non-root u from its parent in $O(\log_B^2 N)$ I/Os. The dynamic forest is represented by its *Euler tour* [19, 18], using the observation that cutting or linking an edge corresponds to a constant number of splits and merges in the Euler tour. The Euler tour is stored in a level-balanced B-tree [1] that supports split and merge in $O(\log_B^2 N)$ I/Os. Each vertex in the forest stores a pointer to its first and last occurrences in the Euler tour, which allows cutting or linking an edge in $O(\log_B^2 N)$ I/Os.

It is straightforward to augment the level-balanced B-tree to support the *rank* operation, which returns the number of elements before a given element in the sequence in $O(\log_B N)$ I/Os. The ranks of Euler tour occurrences can be used to answer subtree queries: Given vertices u and v, u is in the subtree rooted at v if and only if the rank of the first occurrence of u is contained in the interval spanned by the ranks of the first and last occurrences of v.

Let L be the set of vertices adjacent to saddle vertices in a TIN $T_\triangle$; the size of L is at most $8X$, where X is the number of critical vertices of $T_\triangle$. Our sea-level flooding data structure stores, for each vertex v in L, the rank of the first occurrence of v in $\Pi_\downarrow$ and $\Pi_\uparrow$, and it stores for each sink (peak) u the ranks of the first and last occurrences of u in $\Pi_\downarrow$ ($\Pi_\uparrow$). As described above, from the ranks of u and v it can be determined whether v is in the subtree of u in $\Pi_\downarrow$ or $\Pi_\uparrow$.

When the data structure of Yang [21] handles an interchange event between saddles u and v in $\mathcal{A}$, that is, the event that u and v swap height order due to an update to the height of u, $\Pi_\downarrow$ ($\Pi_\uparrow$) is queried with a vertex from each lower (higher) sequence of u and v to determine the new nesting structure of contours. By answering the queries to $\Pi_\downarrow$ and $\Pi_\uparrow$ using the ranks stored in main memory, the queries require no I/Os; since this is the only case in which $\Pi_\downarrow$ or $\Pi_\uparrow$ is queried by the update algorithm for $\mathcal{A}$, our data structure may update $\mathcal{A}$ without I/Os.

During our update algorithm, $\Pi_\downarrow$ and $\Pi_\uparrow$ are updated to reflect the new descending and ascending connectivity of the vertices in the updated square Q_B. Whenever an edge is disconnected or reconnected in $\Pi_\downarrow$ or $\Pi_\uparrow$, this operation is translated into a constant number of splits and merges to the level-balanced B-trees underlying $\Pi_\downarrow$ and $\Pi_\uparrow$, and these splits and merges can cause the ranks of vertices in L to change. By using the *rank* operation on the level-balanced B-tree before and after each such split or merge operation, it is straightforward to update the stored ranks of vertices in L accordingly.

As we assume that X is less than M, L fits in main memory, and thus querying and updating L incurs no I/Os.

▶ **Lemma 13.** *For $\Pi_\downarrow$ ($\Pi_\uparrow$), the ranks of L and terrain sinks (peaks) can be maintained in internal memory by the sea-level flooding data structure such that queries to the sink (peak) reached when following edges in $\Pi_\downarrow$ ($\Pi_\uparrow$) from a vertex $v \in L$ can be answered without I/Os.*

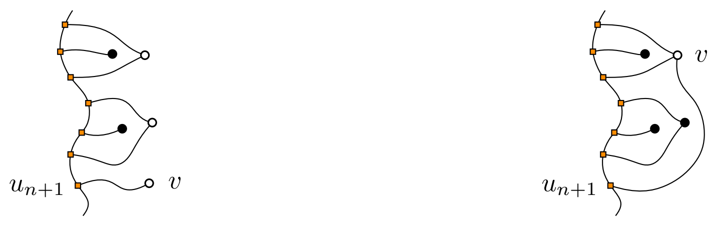
A.3 Analyzing the coastal minima

In this section we show that the contour tree augmented with coastal minima, denoted by $\mathcal{A}^+$, contains enough information to determine which sinks are flooded. Recall that a *coastal minimum* is a coastal vertex u such that no other coastal vertex in $C(u)$ is below u. We define *coastal maxima* analogously to coastal minima. A coastal minimum (maximum) is a *true coastal minimum (maximum)* if it has no lower (higher) neighbor that is a coastal vertex in any coastal region; otherwise it is a *region minimum (maximum)*.

▶ **Lemma 14.** *Let X be the number of sinks, saddles and peaks in the terrain, and let $|C|$ be the number of coastal regions.*

(i) *If a coastal vertex u in coastal region R floods a terrain vertex v, then there is a coastal minimum w in R that also floods v.*

(ii) *The number of coastal minima is $O(X + |C|)$.*

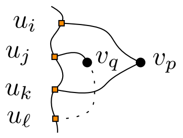


(a) $v \notin B_n$ **(b)** $v \in B_n$

Figure 7 The two cases when u_{n+1} is added. Open and closed vertices are marked with circles and disks, respectively.

Proof. First, we show (i). Suppose a coastal vertex u in coastal region R floods a terrain vertex v; we have to show that there exists a coastal minimum w in R that also floods v. Let $p : u \rightsquigarrow v$ be a flood path from u to v. Now we construct another flood path $p' : u \rightsquigarrow w \rightsquigarrow u \rightsquigarrow v$ that goes through the coastal minimum w in R reached when following a descending path in R from u until reaching a coastal minimum. Then the path $w \rightsquigarrow u \rightsquigarrow v$ is a flood path from the coastal minimum w to v.

Next, we show (ii). We define the *lowest descent path* p from a given vertex u as the path along the edges of $T_\triangle$ starting in u and ending in a sink, such that for each edge (v, w) in p, w is the lowest neighbor of v; the *highest ascent path* is defined analogously. If two lowest descent paths or two highest ascent paths p, q intersect, then they share a common suffix; as such, p and q do not cross. From each true coastal minimum, we follow the lowest descent path to reach a sink, and from each true coastal maximum, we follow the highest ascent path to reach a peak. Note that an ascending path and a descending path cannot cross, and if two paths share a common suffix, we can separate them so that all paths form a planar bipartite graph $G = (A \cup B, E)$, where A is the set of true coastal minima and maxima, and B is the set of sinks and peaks in the terrain. Let $u_1, \ldots, u_{|A|}$ be the vertices in A labeled in the order that they appear as we traverse the coastline. Each vertex $u_i \in A$ is connected to exactly one vertex in B by an edge $e(u_i) \in E$. Since any two true coastal minima are separated by a true coastal maximum and vice-versa, we assume without loss of generality that $u_1, u_3, \ldots$ are minima and



$u_2, u_4, \ldots$ are maxima. Observe that for all $n \geq 1$, u_n and u_{n+1} are connected to distinct vertices in B. Consider three vertices u_i, u_j, and u_k with $i < j < k$, such that u_i and u_k are connected to the same vertex v_p and u_j is connected to vertex v_q. Then v_q cannot be connected to any vertex u_ℓ with $k < \ell$.

To show that $|A| < 2|B|$, we consider constructing G incrementally by adding the vertices of A in the order they appear on the coastline; for each $n \geq 0$ let $G_n = (A_n \cup B_n, E_n)$ be the subgraph of G consisting of $A_n = \{u_1, \ldots, u_n\}$, $E_n = \{e(u_1), \ldots, e(u_n)\}$, and B_n being the set of B-vertices connected by E_n. For each G_n, we call a vertex $v \in B_n$ *open* if it can be reached from u_{n+1} via a path on the terrain that does not intersect any edge in G_n, and *closed* otherwise. Let C_n (O_n) be the set of closed (open) vertices of B_n. We show by induction in n that $|A_n| \leq |O_n| + 2|C_n|$, from which it follows that $|A_n| < 2|B_n|$ (since not all vertices in B_n can be closed). Initially, $A_1 = \{u_1\}$, $|O_1| = 1$ and $|C_1| = 0$. When u_{n+1} is added with an edge to a vertex $v \in B_{n+1}$, there are two cases. If $v \notin B_n$ (Figure 7a), then $|C_{n+1}| = |C_n|$ and $|O_{n+1}| = |O_n| + 1$, so

$$|A_{n+1}| = |A_n| + 1 \leq |O_n| + 2|C_n| + 1 = |O_{n+1}| + 2|C_{n+1}|. \blacktriangleleft$$

Otherwise, $v \in B_n$ (Figure 7b). The edge from u_{n+1} to v moves k vertices in O_n to C_{n+1}, so $|O_{n+1}| = |O_n| - k$, $|C_{n+1}| = |C_n| + k$. Since both u_n and u_{n+1} cannot be maxima or minima, they cannot have the same neighbor and therefore the neighbor of u_n is open in G_n and closed in G_{n+1}, which implies that $k \geq 1$. Thus it follows that

$$|A_{n+1}| = |A_n| + 1 \leq |O_n| + 2|C_n| + 1 \leq (|O_n| - k) + 2(|C_n| + k) = |O_{n+1}| + 2|C_{n+1}|. \blacktriangleleft$$