Computing Floods Caused by Non-Uniform Sea-Level Rise

Lars Arge* Yujin Shin* Constantinos Tsirogiannis*

Abstract

Predicting floods caused by the rise of the sea level is a critical task for preventing large scale catastrophes. Such predictions can potentially be made using a forecast of the sea level and a detailed model of the terrain. However, since available terrain datasets can easily exceed the size of the main memory of a standard computer, I/O (rather than internal computation time) can often become the bottleneck when computing such predictions. Thus to perform predictions efficiently we need an *I/O-efficient* approach, which minimizes the transfer of data blocks between main memory and disk.

Given a terrain raster T and a sea-level forecast raster S of N cells each, we examine the problem of computing the water level of the induced flood for each cell in T. We introduce an I/O-efficient algorithm for this problem that uses $O((N/B)\log_{M/B}(X/B))$ I/Os after $O((N/B)\log_{M/B}(N/B))$ I/Os of preprocessing, where X is the number of local minima in T, and M and B are the size of main memory and data block, respectively. When $X < M$ (which holds in practice) our algorithm requires optimal $O(N/B)$ I/Os after preprocessing.

We have implemented our algorithm and put considerable effort into engineering it. We present experiments that illustrate the efficiency and practicality of the algorithm, which is so efficient that work is underway to incorporate our results in the forecast services of the Danish Meteorological Institute.

1 Introduction

Many times in the past, the rise of the sea level has led to large floods with disastrous consequences. Modeling such floods before they happen can help to take measures that would reduce their negative impact. This is a complex task because the height of the sea level is not uniform across different geographic locations. More than that, the local sea-level height may change rapidly through time.

Recent advances in meteorology have made it possible to produce regular and accurate predictions of the sea level across large regions. For example, nowadays the Danish Meteorological Institute provides a forecast of the sea level within Danish territorial waters. The institute releases a new set of forecasts every six hours, containing the hourly forecasts of the expected sea-levels for the next five days. These forecasts are very accurate; in 2015, the predictions of the sea level were on average 6.61 cm higher than the actual level. By combining such forecasts with the very detailed and accurate terrain data that has been acquired in recent years, including for Denmark, there is a potential for producing very accurate flood predictions. However, doing so is challenging, not just because a set of forecasts must be accurately processed before the next set of forecasts is released, but also because the data involved typically exceeds the size of the main memory of a standard computer. When a dataset is too large to be stored in main memory, it is instead stored on disk. To process the entire dataset, the data is transferred in blocks between disk and main memory, and a single block transfer can be a million times slower than a typical CPU operation. Thus, if data is not handled appropriately such time-consuming block transfers can result in a huge performance bottleneck.

In this paper we consider the problem of computing the areas in risk of flooding based on detailed terrain data and accurate sea-level forecast data, and develop so-called I/O-efficient algorithms for this problem that try to minimize the number of block transfers. While previous such algorithms only considered uniform sea-level rise, our new algorithms can handle the much more realistic case of non-uniform sea-level rise forecasts. We have engineered and implemented the algorithms utilizing properties of typical terrain and forecast data to obtain software that are very efficient in practice.

1.1 The non-uniform sea-level flooding problem.

In order to formally describe the problem that we consider, we need a few definitions. A terrain is often represented using a *raster*, that is, as a two-dimensional array with N cells, with each cell v storing an elevation value $h(v)$. Thus, a raster divides the domain of a geographic region into square cells of equal size, and each cell is associated with an elevation value. In the theoretical part of this paper, we consider that the input consists of two rasters covering the same geographic region: A

*MADALGO, Department of Computer Science, Aarhus University, Denmark. Supported in part by the Danish National Research Foundation and Innovation Fond Denmark. MADALGO is the Center for Massive Data Algorithmics, a center of the Danish National Research Foundation. Email:{large,yujinshin,constant}@madalgo.au.dk.

terrain raster T storing the elevation of the earth surface, and a *forecast* raster S storing the elevation of the sea level. We denote the elevation functions of T and S by h_T and h_S, respectively. We say that a raster cell is *undefined* if there is no elevation value assigned to it. A cell in T is undefined when the corresponding location falls within a region covered by sea, and a cell in S is undefined when it is covered by terrain. We call a cell in T that has a defined value a *terrain cell*, and a cell with a defined value in S a *water cell*. For simplicity, we assume that all cell heights are distinct, and that T and S have the same resolution. Later on in this paper, we show how to remove these assumptions.

For two cells u and v in the same raster, 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. For a cell u in T and a cell v in S, we say that u and v are adjacent if v is adjacent to the cell u' in S that covers the same region as u in T. A terrain cell is called a *boundary* cell when it is adjacent to a water cell in S. We say that a boundary cell u in T is *directly flooded* by a water cell w in S if w is adjacent to u and $h_T(u) < h_S(w)$. For a cell u in T, we say that u is *flooded* by a water cell w, and that w is a *potential source* of u, when there exists a path $p = v \rightsquigarrow u$ of adjacent cells in T such that each cell t in p has $h_T(t) < h_S(w)$ and v is directly flooded by w. We call p a *flood path* of u. For a flooded cell u, we define its *flood height* as $f(u) = \max_{w \in S_u} h_S(w) - h_T(u)$ where S_u is the set of potential sources of u. If u is flooded by w and $h_S(w) - h_T(u) = f(u)$, that is, w is the highest potential source, then we call w the *flood source* of u.

Given input rasters T and S, the *non-uniform sea-level flooding* problem consists of computing an output raster F covering the same geographic region as T and S, and containing the flood height of each cell.

1.2 Previous results. We use the *I/O-model* by Aggarwal and Vitter [1] to design and analyze our algorithms. In this model, the computer has two levels of storage, namely main memory and disk. The main memory can store at most M items of data, whereas the disk has unlimited size. Given a dataset of $N > M$ elements, the entire dataset is initially stored on disk but computations can be performed only on items that are stored in main memory. To process the entire dataset, there is the need to transfer data between main memory and disk. Data is transferred in blocks of B items. The transfer of a single block of data is called an I/O-operation, or simply an I/O. The performance of an algorithm is measured by the number of I/Os it performs. Algorithms that are designed based on this model are called I/O-efficient algorithms.

Previously, a large number of results on I/O-efficient algorithms have been obtained. The number of I/Os required to read N items is $\Theta(\text{scan}(N)) = \Theta(N/B)$, while the number of I/Os required to sort N items is $\Theta(\text{sort}(N)) = \Theta((N/B)\log_{M/B}(N/B))$ I/Os [1]. Note that for all realistic values of N, M, and B, we have that $\text{scan}(N) < \text{sort}(N) \ll N$. Below we shortly review the I/O-efficient algorithms that have been obtained for modeling water flow and floods on terrains, and are relevant to this paper. See [15, 8] for further results.

To model how water flows on a terrain T, each cell u is normally assigned a *flow direction* indicating to which one of its 8 neighbors water will flow, namely the lowest neighbor of u that has a lower elevation than u. The *flow accumulation* of a cell u is the number of cells from which we can reach u by following the flow directions on T. The flow accumulation of each cell in T can be computed in $O(\text{sort}(N))$ I/Os [12].

A cell which has no lower neighbors is called a *sink*, and sinks are not assigned flow directions. Each sink u defines a unique *watershed* μ, which is the set of cells from which we can reach u by following the flow directions on T. Note that each cell belongs to exactly one watershed. For a watershed μ, we denote its sink by d_μ. The *watershed decomposition* raster W of T, where each cell is labeled with the sink of its watershed in T, can be compute in $O(\text{sort}(N))$ I/Os [9].

We say that a terrain cell u is a *saddle* cell if, as we traverse its neighbors in clockwise order, the cells of lower elevation than u form two or more non-connected sequences. For simplicity, we assume that all saddle cells have exactly two sequences of lower neighbors. Later in this paper we discuss how to remove this assumption. A terrain cell u defines a *depression*, that is the maximal connected component of all terrain cells that have the same or smaller elevations than u and that are connected to u. A depression β_1 is *maximal* if every depression $\beta_2 \supset \beta_1$ contains strictly more sinks than β_1. A maximal depression β that contains exactly one sink is called an *elementary depression*.

When modeling flow over a terrain T, it is often assumed that once water flows into a depression it never flows out again, that is, water stays in watersheds and depressions do not fill up and spill into other depressions in T. Therefore, most flow modeling approaches first remove depressions by *completely filling* the terrain [7, 6]. Intuitively, completely filling a terrain corresponds to raising the terrain by conceptually pouring water on it until all depressions are filled and a steady state (where there is a non-increasing path from each cell to the boundary of the terrain) is reached. More formally, each cell u in T is raised to the height corresponding to the *lowest height path* p between u and a boundary

cell v. This path is defined as the set of cells in T that connect u and a boundary cell v such that the maximum elevation encountered in the path is minimized. Raising a terrain T in such a way can be performed in $O(\text{sort}(N))$ I/Os [10].

In our problem, if all defined cells in the forecast raster S have the same value h_r, that is, if the sea-level rises *uniformly* with the same amount, then a cell u in T is flooded with flood height $h_r - h_{T'}(u)$ if and only if $h_{T'}(u) < h_r$ where T' is the terrain obtained by filling the depressions in T as described above [2]. Thus, in this case F can easily be computed in $O(\text{sort}(N))$ I/Os. However, this approach does not work for our more general problem where the sea-level rise is *non-uniform*, because the raising process only corresponds to considering the lowest height path p from a cell u to a boundary cell v in T; this path might not be a flood path for u (e.g. if v is directly flooded by w in S but $h_S(w) < h_T(u)$) but another flood path could still exist. Thus, the variant of the problem that we consider is much harder than the uniform sea-rise previously considered, since in our variant one needs to consider all paths to the boundary and not only the lowest height path.

Motivated by previous methods that require filling depressions on terrains, our algorithm will rely on the so-called *merge tree* that captures the essential topology (the nesting of depressions) of a terrain [3, 2]. The merge tree $\mathcal{M}$ represents the nested topology of maximal depressions in T. The tree contains exactly one node for each maximal depression in T; we denote the maximal depression corresponding to a node x in $\mathcal{M}$ by β_x. Each elementary depression is represented by a leaf node, and a node x is the parent of a node y when $\beta_y \subset \beta_x$ and there exists no maximal depression β_z such that $\beta_y \subset \beta_z \subset \beta_x$. Each node x corresponds to a cell s_x in T, defined as follows: If x is a leaf node, then s_x is the sink of β_x. For a node x that is the parent of nodes y and z, cell s_x is the saddle cell where β_y and β_z are merged. Refer to Figure 1. Since we have assumed that a saddle cell has exactly two non-connected sequences of lower neighbors, an internal node x has exactly two children. Thus $\mathcal{M}$ is a binary tree. Each maximal connected region of defined cells in T is represented by a distinct merge tree. For simplicity, we assume that the terrain cells in T form a single connected continent, and thus that T is represented by a single merge tree $\mathcal{M}$. Agarwal et al. [16] described how to compute the merge tree $\mathcal{M}$ in $O(\text{sort}(N))$ I/Os. In our algorithm, we will assume that each node x in $\mathcal{M}$ is labeled with its rank (using post-order numbering) in the depth first search (DFS) traversal of $\mathcal{M}$. We call this value the *DFS rank* of x. The DFS ranks of the tree nodes can be computed in $O(\text{sort}(N))$ I/Os [17].

1.3 Our results. In this paper we develop the first I/O-efficient algorithm for the non-uniform sea-level flooding problem, which as discussed above is much more practically relevant and significantly harder than the previously considered uniform sea-level flooding problem. We have engineered and implemented the algorithm utilizing properties of typical terrain and forecast data, and demonstrate the practical performance of our new algorithm on real-world data.

As we describe in Section 2, our algorithm consists of two parts. The first part is preprocessing, which involves handling only T and uses $O(\text{sort}(N))$ I/Os. After that, the non-uniform sea-level flooding problem can be solved for a given forecast raster S using $O((N/B)\log_{M/B}(X/B))$ I/Os, where X denotes the number of sinks of T (that is, the number of leaves in the merge tree $\mathcal{M}$ of T). While $X = \Theta(N)$ in the worst case, such that our algorithm also uses $O(\text{sort}(N))$ I/Os after preprocessing, X is much smaller than N in practice. For example, the publicly available detailed raster of Denmark [4] (where each cell represents a 0.4 by 0.4 meter region) contains approximately one trillion cells, whereas the number of sinks (after removing all depressions with volume less than 1 m^3, which is customary in flood computations) is approximately 24 million. Furthermore, under certain realistic assumptions, we describe how we can improve our algorithm further. Under the (often made) assumption that a constant number of rows of T fit in memory, our algorithm can be improved to use only $O(\text{scan}(N) + \text{sort}(X))$ I/Os. Under

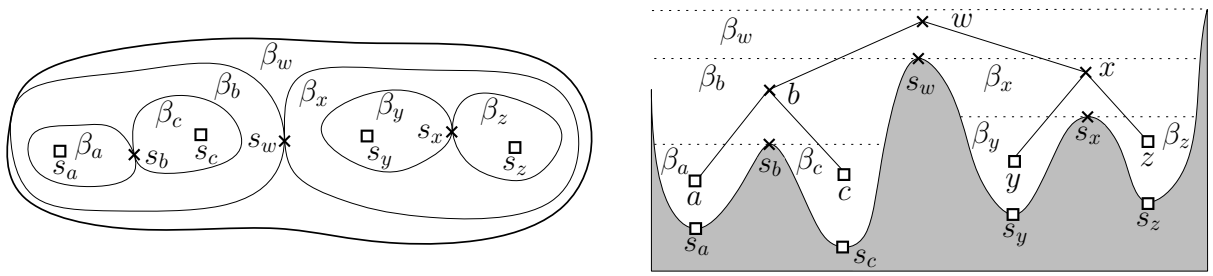


Figure 1: On the left, a terrain viewed from above. On the right, the same terrain viewed from the side along with the merge tree. Crosses and squares represent saddles and sinks, respectively.

the practically realistic assumption that $X < M$ we can adjust our algorithm to use optimal $O(\text{scan}(N))$ I/Os after preprocessing.

Our algorithm builds on the insight that after computing the flood heights of all sinks of T, it is easy to compute the flood heights of all cells in each watershed of T independently. To compute the flood heights for all sinks, we perform two sweeps of the merge tree $\mathcal{M}$ of T. Thus, the preprocessing basically consists of computing the watershed decomposition raster W and merge tree $\mathcal{M}$ of T. When handling a given forecast S we then perform three stages: In stage 1 we solve the non-uniform sea-level flooding problem for each sink independently, considering only flooding from boundary cells (if existing) of the watershed corresponding to this sink; in stage 2 we then compute the flood heights for all sinks using the merge tree; in stage 3 we finally find the flood height of all cells in each watershed independently. Note that only the first and last step works on the large rasters T (and S) and the watershed raster W, whereas the second step can be performed completely in memory when $X < M$. Overall, our algorithm is inspired by a recent algorithm by Arge et al. [13] for computing flood risk from non-uniform rain events. This algorithm also uses two sweeps of the merge tree.

In Section 3, we describe the practical implementation of our I/O-efficient algorithm and present experiments with real-world data, including a comparison with the previous uniform sea-level flooding algorithm [2]. The development of our algorithm was heavily motivated by the availability of detailed terrain data and accurate sea-level rise forecasts for Denmark, as well as by discussions with the Danish Meteorological Institute on how to practically handling the problem of producing realistic flood risk maps from their very accurate forecasts. Thus, the definition of the non-uniform sea-level flooding problem was heavily influenced by these discussions. While our algorithm uses ideas from the previous non-uniform rain flood risk algorithm [13], we put considerable effort into engineering the algorithm in order to obtain an algorithm that is able to process the Danish terrain and forecast data in less than 6 hours. An algorithm that uses more than 6 hours would be useless in practice, since a new forecast appears every 6 hours. Obtaining such an algorithm is not trivial, since as we describe in Section 3 a simple scan of two rasters of the size of Denmark terrain model (approximately 3.7 TB) and output of a result of the same size (without actually performing any computation) takes 4 hours and 7 minutes on our relatively powerful computational platform. Thus, in Section 3 we not only describe the practical implementation of our algorithm under the assumption fulfilled by the Danish datasets that $X < M$, but we also present our further engineering efforts to handle a single Danish forecast in 4 hours and 13 minutes, which is very close to the practically optimal 4 hours and 7 minutes. Some of our engineering efforts exploited properties of the data to improve the running time by moving a large part of the computation to the preprocessing step (only performed once), while at the same time handling the fact that real terrain and forecast rasters T and S do not have the same resolution (which creates a number of practical challenges, e.g. in relation to inconsistent boundaries). In fact, we arrive at an algorithm where we avoid working on the large terrain raster T in the first stage of the algorithm, such that in practice I/Os only have to be performed in stage 3.

Finally, it should be noted that our algorithm and its practical implementation can relatively easily be extended to handle a number of concurrent forecasts, like the 120 forecasts released from the Danish Meteorological Institute every 6 hours, without much overhead. In fact, our successful development and engineering of the non-uniform sea-level flooding algorithm has resulted in work being underway to incorporate our results in the forecast services of the Danish Meteorological Institute, such that not only sea-level rise forecasts but also flood risk forecasts are released every 6 hours for Denmark.

2 The algorithm

In this section we describe our I/O-efficient algorithm for the non-uniform sea-level flooding problem. As mentioned, one key insight behind our algorithm is that after finding the flood source for each sink c in T, the flood heights of all cells in the watershed of sink c can easily be computed independently. Recall that the flood source of a cell u is the water cell w of the largest elevation such that there exists a path on the terrain connecting u and w, where all cells on the path have a lower elevation than w. The key insight is then expressed in the following lemma.

LEMMA 2.1. *Let u be a flooded cell in T and let μ be the watershed that u belongs to. The flood source of u is the same as the flood source of the sink d_μ of μ.*

Proof. Let w be the flood source of u, and w' be the flood source of d_μ. Let $p : u \rightsquigarrow d_\mu$ be the non-ascending path that we get if we start from u and follow the flow directions on the terrain. Note that all cells on p have lower elevation than w, since u is flooded by w. If $h_S(w) < h_S(w')$, then u is flooded by w', since we can construct a flood path between u and w' by concatenating the flood path between w' and d_μ with the reverse of p. If $h_S(w) > h_S(w')$, then d_μ is flooded by w, since we can construct a flood path from d_μ to w

by concatenating the flood path between w and u with p. □

As discussed in the introduction, Lemma 2.1 allows us, after a preprocessing step that constructs the watershed decomposition raster W and merge tree $\mathcal{M}$ of T independent of a forecast S, to design a three stage algorithm for computing the flood height of each cell in T given a forecast S: In stage 1 we find the flood source for each sink d_μ independently (that is, a potential source), considering only flooding from boundary cells (if existing) of μ. In stage 2 we then compute the flood sources for all sinks using two controlled sweeps of the merge tree $\mathcal{M}$. Finally, in stage 3 we find the flood source and flood heights of all cells in each watershed independently, that is, we construct the output raster F.

Below we show how stage 1 and 3 can be performed in $O((N/B)\log_{M/B}(X/B))$ I/Os in the general case, in $O(\text{scan}(N) + \text{sort}(X))$ I/Os under the assumption that a constant number of rows of T fit in memory, and in $O(\text{scan}(N))$ I/Os under the assumption that $X < M$. This is also the overall complexity of our algorithm, since we also show how stage 2 can be performed in $O(\text{scan}(X))$ I/Os. That is, if $X < M$, stage 2 does not require any I/Os at all.

2.1 Preprocessing. In the preprocessing step, we construct the watershed decomposition W and merge tree $\mathcal{M}$ of T. We can construct both of these structures in $O(\text{sort}(N))$ I/Os using existing I/O-efficient algorithms [16, 9]. We also label $\mathcal{M}$ and W so that each watershed in W can be identified with the leaf node of the corresponding sink in $\mathcal{M}$. In particular, each watershed in W is labeled by the DFS rank of the corresponding leaf node in $\mathcal{M}$. This labeling can be performed in $O(\text{sort}(N))$ I/Os using an algorithm of Chiang et al. [17].

2.2 Stage 1. In stage 1 we find the flood source for each sink d_μ independently (that is, a potential source), considering only flooding from boundary cells (if existing) of μ. More precisely, given T, W, and S, we want to find the highest water cell w such that w is adjacent to a boundary cell of μ and d_μ is flooded by w. The output of this stage is the flood sources sorted by the DFS rank of the respective sinks (the leaf nodes corresponding to the sinks) in $\mathcal{M}$.

Our algorithm will be based on the following lemma, which can be proved similarly to Lemma 2.1.

LEMMA 2.2. *Let u be a terrain cell and let μ be the watershed that u belongs to. If u is flooded by a water cell w, then the sink d_μ is flooded by w.*

Based on Lemma 2.2, to compute a potential source for every sink whose watershed μ is adjacent to water, we simply need to consider the boundary cells of μ and their adjacent water cells. To do so, we scan T, S, and W simultaneously and process the cells as follows: For each terrain cell u that we process, we check if u is a boundary cell (i.e. it is adjacent to at least one water cell). If it is, then we create a record for each adjacent water cell that floods u. Each record created for u is labeled with the DFS rank of the watershed that u belongs to. After the scan we sort all created records according to their DFS ranks, and extract the potential source for each rank, which is the highest water cell that floods a terrain cell associated with that rank. Thus the stage is computed in $O(\text{sort}(N))$ I/Os.

Given that T has X sinks, and X can be much smaller than N, we can actually perform stage 1 slightly more efficiently: We process a chunk of X consecutive cells at a time, with each chunk extracted while scanning the raster. From each chunk that we process, we extract the flood sources that appear in this chunk, and we use these to update the potential sources that we have computed from all processed chunks so far. There are $O(N/X)$ chunks of X cells, each of which requires $O(\text{sort}(X))$ I/Os, therefore we use $O((N/B)\log_{M/B}(X/B))$ I/Os in total.

If a constant number of rows of T fit in main memory, we can improve our algorithm further. We scan T, S, and W maintaining a constant number of their rows in main memory. For each scanned watershed μ, we find a potential source and maintain it in main memory. Since each watershed μ is a connected region in T, it is guaranteed that the flood source of d_μ is found by the time μ is completely scanned (the number of watersheds being considered at the same time is bounded by the number of cells in the rows in memory at a given time). After the scan we sort the potential sources based on the DFS rank of the corresponding sinks. This requires $O(\text{scan}(N) + \text{sort}(X))$ I/Os.

If $X < M$, we can improve the algorithm further by simply scanning T, S, and W while maintaining all watersheds in main memory and sort them according to their DFS ranks. This requires $O(\text{scan}(N))$ I/Os.

2.3 Stage 2. In stage 1, we computed a potential source for each flooded sink whose watershed is adjacent to water. Given these potential sources and the merge tree $\mathcal{M}$ of T, in stage 2 we compute the actual flood sources of all sinks, again sorted by the DFS rank in $\mathcal{M}$. Our algorithm works by sweeping $\mathcal{M}$ twice in a controlled manner. In the first sweep, for each node x in $\mathcal{M}$ we find a potential source w for the corresponding cell s_x by only considering the potential sources of the sinks

represented by the leaves in the subtree of x. We do this by sweeping $\mathcal{M}$ in DFS order (bottom-up); by the time we visit a node x, all of its descendants have already been processed. In the second sweep, for each node x in $\mathcal{M}$ we find the flood source of s_x based on the flooded sinks among all the leaves in $\mathcal{M}$. We do this by sweeping $\mathcal{M}$ top to bottom, in reverse DFS order, while scanning the result of the first sweep. Thus, after completing the second sweep, we have then found the flood sources for all sinks in DFS order (since they correspond to the leaves in $\mathcal{M}$).

Upward sweep. In the first sweep, for each node x in $\mathcal{M}$ we find a potential source for cell s_x by examining the respective maximal depression β_x. More specifically, for each node x we find the highest potential source w for s_x such that, in stage 1, cell w was found to flood a sink in the subtree of x. We call this cell the *subtree source* of x. To compute the subtree sources we need the following lemma.

LEMMA 2.3. *Let x be a node in $\mathcal{M}$ and let u be a cell in a maximal depression $\beta_y \subset \beta_x$. If u is flooded by a water cell w such that $h_S(w) > h_T(s_x)$, then s_x is flooded by w.*

Proof. Since u belongs to the maximal depression of s_x, there exists a path $p : s_x \rightsquigarrow u$ in β_x such that for each cell t in p we have $h_T(t) \leq h_T(s_x) < h_S(w)$. Hence, we can get a flood path between w and s_x by concatenating p with the flood path between u and w. □

From Lemma 2.3 it is easy to infer that the subtree source of x, if it exists, is the maximum elevation subtree source among the children of x. Based on this observation, we can compute the subtree sources of all nodes in $\mathcal{M}$ as follows: We scan $\mathcal{M}$ in DFS order while maintaining an I/O-efficient stack Q storing the subtree sources of the visited nodes in $\mathcal{M}$. When we visit a leaf node x in the tree, we set the subtree source of x to be the potential source extracted for sink s_x during stage 1 (if such a source was found). We then push this subtree source on Q. When we visit an internal node x in the tree, we pop the elements of Q that store the subtree sources of the children of x, and compute from these the subtree source of x; this is done by selecting the popped subtree source w with the largest elevation and checking if $h_S(w) > h_T(s_x)$. Then we push the computed subtree source on Q. Since the tree is traversed in DFS order, it is guaranteed that the top elements of the stack store the entries that correspond to the children of the newly visited node x. During the sweep, we also construct a list that stores the subtree sources of the tree nodes in the order they are traversed. This list will be used in the downward sweep.

From the above description, the upward sweep can be performed using $O(\text{scan}(X))$ I/Os for traversing $\mathcal{M}$ and $O(X)$ stack operations, each of which requires $O(1/B)$ amortized I/Os, hence $O(\text{scan}(X))$ I/Os in total.

Downward sweep. Unlike the upward sweep, in the downward sweep for each node x we consider the entire merge tree $\mathcal{M}$ to find the actual flood source of s_x. We need the following lemma.

LEMMA 2.4. *Let x be a node in $\mathcal{M}$ and let s_x be the terrain cell represented by x. If cell s_x is flooded, then the flood source of s_x is also the flood source for all cells in a maximal depression $\beta_y \subset \beta_x$.*

Proof. Since $\beta_y \subset \beta_x$ every cell in β_y has a lower elevation than s_x. Therefore, there exists a flood path between the flood source of s_x and any cell in β_y. □

Let x be a node in $\mathcal{M}$ and y be a child node of x in $\mathcal{M}$. Using Lemma 2.4, we can easily conclude that if s_x is flooded then the flood source of s_y is the same as the flood source of s_x; otherwise the flood source of s_y is the same as with the subtree source of s_y (if existing). Thus, we can compute the flood source for the nodes in $\mathcal{M}$ as follows: We scan $\mathcal{M}$ top to bottom, in reverse DFS order, while simultaneously scanning in reverse order the list of subtree sources produced in the upward sweep, and maintaining a stack Q containing the flood sources of the nodes that were already visited during the sweep. When we visit a node x, if the parent of x is flooded we pop the flood source of the parent from Q and set is as the flood source of x. Otherwise, we set the flood source of x to be its already computed subtree source. We then push the computed flood source of x on Q, once for each child of x in $\mathcal{M}$. The output of this stage consists of the flood sources of the sinks in T, sorted by the DFS rank of the corresponding nodes in $\mathcal{M}$.

During this second sweep, we perform $O(X)$ stack operations each requiring $O(1/B)$ amortized I/Os, and use $O(\text{scan}(X))$ I/Os for scanning the merge tree and the upward sweep result simultaneously. This leads to $O(\text{scan}(X))$ I/Os in total for the downward sweep.

2.4 Stage 3. In stage 2 we computed the flood sources of all sinks, sorted by their DFS rank in $\mathcal{M}$. In this stage we compute the flood source and flood heights of all cells in each watershed independently, that is, we construct the output raster F.

Let c be a cell in a watershed μ. Using Lemma 2.1, we know that if c is flooded then its flood source is the same as the flood source of sink d_μ. Based on this, we can construct the output raster F as follows: We simultaneously scan W, T, and the flood sources found

in stage 2, processing a chunk of X consecutive cells from each raster at a time. Since there are at most X flood sources found from stage 2, we can compute the correct flood heights of the cells in the chunk. To do so, we sort the X cells in the chunk by the DFS rank of the corresponding sinks and scan the sorted cells and the flood sources simultaneously to compute the flood heights. Once the flood heights are computed for the cells in the chunk, we sort them based on the xy-position of corresponding cells in T and write the resulting chunk to F. There are $O(N/X)$ chunks and for each chunk we perform $O(\text{sort}(X))$ I/Os, which results in $O((N/B)\log_{M/B}(X/B))$ I/Os in total.

If a constant number of rows of T fit in main memory, we can improve the algorithm as follows: Let d_μ be a sink and let $y_{\max}(d_\mu)$ denote the maximum y-coordinate (minimum row index) where a cell in watershed μ appears. Note that the values of $y_{\max}$ can be easily computed by scanning T and W, since each watershed is a connected region in T. We sort the flooded sinks found from stage 2 in increasing order of their $y_{\max}$ values. Then we scan T and W starting from the top row, while using the sorted flooded sinks to compute the flood heights and write them to F. Sorting the flooded sinks requires $O(\text{sort}(X))$ I/Os, and we use $O(\text{scan}(N))$ I/Os in the remaining process. Therefore, this requires $O(\text{sort}(X) + \text{scan}(N))$ I/Os in total.

If $X < M$, we improve the algorithm by loading all flooded sinks in main memory. Then we scan W and T, and for each cell in T we find the flood source of the associated sink in main memory, using $O(\text{scan}(N))$ I/Os in total.

3 Implementation and experiments

As mentioned in the introduction, development of the I/O-efficient algorithm described in the previous section was heavily motivated by the availability of detailed terrain data and accurate sea-level rise forecast for Denmark, as well as by discussions with the Danish Meteorological Institute. Thus we have engineered and implemented the algorithm using properties of the Danish terrain and forecast data. In this section we describe our engineering and implementation efforts, and the results of experimentation on real-world data.

3.1 Denmark data. The terrain model of the entire terrain surface of Denmark published by the Danish Geodata Agency in 2015 is very detailed and accurate [4]. The raster terrain has a resolution of just 0.4 m, and consists of almost one trillion cells arranged in 881,201 rows and 1,128,794 columns. The size of the raster is approximately 3.7 TB. When using such detailed terrain models for flood risk analysis, small depressions of less than 1m^3 volume are often removed (filled), since they are most likely the result of noise that appears in data acquisition. Removing such small depressions from the Denmark model results in a terrain with approximately $X = 24$ million sinks. In our experiments we used the model with small depressions removed, and we refer to this dataset as `denmark`. Note that for any reasonable computer we have $X < M$, such that all sinks can be stored entirely in main memory.

The sea-level forecast data for the Baltic and the North sea released by the Danish Meteorological Institute [5] (based on complicated model runs) has a much coarser resolution than the `denmark` dataset. It consists of only 60 rows and 101 columns, where each cell has a width of 4523.59 m and a height of 5934.36 m. The size of one forecast raster is just 8 KB. In our experiments we used data from December 6, 2013, where the storm named Bodil hit Denmark. We refer to this raster as `bodil`. Note that the largest difference of the sea-level across the raster is an impressive 2.7 m.

3.2 Engineering and implementation. We implemented our algorithm closely following the description provided in Section 2 under the assumption that $X < M$, which holds for realistic terrains and definitely for `denmark`. Thus the implemented algorithm uses $O(\text{sort}(N))$ I/Os for preprosessing and then only $O(\text{scan}(N))$ I/Os for each sea-level forecast. Yet as mentioned, we removed several assumptions that were made to simplify the theoretical description. These assumptions were removed without affecting the I/O-efficiency of the algorithm. We also engineered several aspects of the algorithm in order to exploit properties of `denmark`, most of which holds for general terrain and forecast data, in order to obtain software that are very efficient in practice. Some of our modifications not only exploited properties of the data, but also handle the fact that real terrain and forecast rasters do not have the same resolution.

Removing assumptions. In Section 1 we made three simplifying assumptions about the input data. First, we assumed that all elevations in T are distinct such that flow directions can easily be assigned to all but sink cells. If not all elevations are distinct, T may contain flat regions and it is unclear how to assign flow directions to these regions. To handle such situations, we used the routing algorithm described by Arge et al. [10]. This algorithm requires $O(\text{sort}(N))$ I/Os and assigns flow directions to the cells of a flat region so that water flows down to a lower elevation cell adjacent to that region. Second, we assumed that T consists of a single connected component of defined cells, and thus is represented by

a single merge tree. If sea regions divide T into more than one connected components, then we need several merge trees to represent the topology of T. In that case, we obtain a single tree by creating a virtual root and connecting it to the root of each smaller tree. We can do this in $O(\text{scan}(X))$ I/Os by traversing the forest of smaller trees in DFS order. Finally, to make the merge tree a binary tree, we assumed that each saddle cell is adjacent to exactly two lower maximal depressions. We can remove this assumption using a technique described by Arge et al. [13]; we traverse the tree and split each node with $k > 2$ children into $k - 1$ nodes with two children each. This can be performed in $O(\text{scan}(X))$ I/Os.

Engineering. In the definition of the non-uniform sea-level problem in Section 1, we assumed that the terrain T and sea-level forecast S have the same resolution and line up perfectly. This is not the case in practice, such as in the case of the `denmark` and `bodil` datasets that have very different resolutions, where the main issue in terms of our algorithm is that not all areas corresponding to undefined cells in T (in particular cells adjacent to boundary cells in T) are defined in the coarser S. We engineered our algorithm to handle this, while at the same time exploiting the fact that S is much smaller than M to significantly improve its practical performance.

To tackle the issue of different resolutions, we add an additional preprocessing step, where we for each boundary cell u in T compute a set D_u of defined cells in a forecast raster, considered capable of flooding u directly. We call these water cells the *candidates* of u. In Appendix A, we describe an intuitive and practical approach for computing candidate cells for each boundary cell in T. This computation is required only once for the given terrain and forecast rasters, since each new forecast raster has exactly the same set of undefined cells.

Now let N_S denote the number of cells of the forecast raster S and let E denote the sum of the number of candidate cells for all watersheds boundary cells in T. Note that since a water cell can be a candidate for multiple boundary cells, E can be larger than N_S. However, in practice E is small. For our datasets `denmark` and `bodil`, E is approximately 81 MB. Thus, using that $E + N_S + X << M$, that is, that an entire forecast S, the candidates of all boundary cells of T, and all sinks of T can be stored in memory, we can significantly improve the time our algorithm needs to handle an event by moving part of stage 1 to the preprocessing stage. Recall that in stage 1 we compute for each sink d_μ in T the highest water cell that directly floods any of μ's boundary cells. As described above, our algorithm reads the terrain raster T and watershed decomposition raster W in step 1, which is time consuming since these rasters are very large. However, we engineered our algorithm to avoid reading these rasters altogether by adding a preprocessing step, where we for each sink d_μ compute a set of all pairs (u, w) where u is a boundary cell for the watershed μ and $w \in D_u$. The number of such pairs is E and they can easily be computed in a scan of W while keeping all candidates in memory. After having computed the pairs once and for all during preprocessing, stage 1 can now easily be computed entirely in main memory using the pairs and the given forecast S. Since stage 2 does not need any I/Os (since $X < M$), stage 3 is then the only part of our algorithm where the large (size N) terrain and watershed decomposition rasters are used. Thus effectively we have moved a large part of the computation to the preprocessing step, produced an algorithm that only scans the input (T, S, and W) exactly once and writes the output (F) to handle a given forecast. As described below, this is very significant in practice, since reading and writing these large rasters are the practically expensive parts of the computation.

3.3 Experiments. To illustrate the practical efficiency of our algorithm, we experimented with the `denmark` and `bodil` datasets. For the experiments we used a computer with 125 GB of main memory and a 3.50 GHz Intel Xeon E5-1650 processor. During the execution of our algorithm, we used only 64 GB of memory for the preprocessing stage, and only 4 GB during the rest of the execution. We used 18 disk drives of 5.5 TB and 5600 RPM each to store temporary files. To increase throughput, the disks were used in a RAID 6 configuration, which effectively meant that 16 of the disk were striped together to simulate a single disk with a large blocksize, while the remaining 2 disk were used to create redundancy. The operating used system was Ubuntu version 16.04.

To process the `denmark` and `bodil` datasets our algorithm used 3 days 19 hours 35 minutes on preprocessing. Hereafter processing of forecast `bodil` took only 4 hours and 13 minutes. During the forecast processing, the first two stages of our algorithm took less than 10 seconds, while the rest of the time was spent during stage 3, which as discussed above is the only part of the forecast algorithm that works on large rasters of size 3.7 TB. More precisely, it scans two rasters T and W of this size and writes the output raster F.

To further investigate the performance of our algorithm, we implemented a simple program that scans T and W and outputs a dummy raster of size N without actually performing any computation. This simple pro-

gram took 4 hours and 7 minutes (which corresponds to processing of approximately 785 MB per second). In other words, the time spent by our algorithm on other things than reading and writing the large rasters is negligible. More importantly, while our algorithm can process a forecast in close to the practically optimal time, making it interesting in a setting where the Danish Meteorological Institute releases a new forecast every 6 hours, the algorithm would have been practically useless in this setting if we had not eliminated the scans of the two large inputs in step 1.

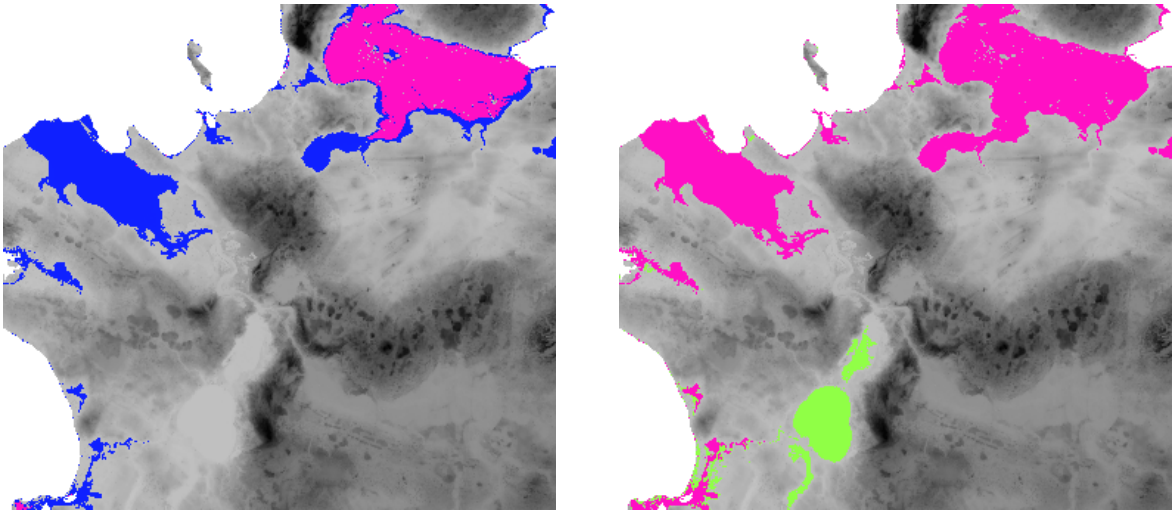
Since most of the time of our algorithm is spent on reading the input and writing the output, we also implemented a version of the algorithm working on compressed data (using the Deflate algorithm [14]). This reduces the size of `denmark` to approximately 422 GB and the size of the corresponding watershed decomposition raster to only approximately 18 GB. Using this implementation, handling of a forecast took 5 hours and 11 minutes, and the corresponding running time with no computation was 4 hours and 45 minutes. In other words, the time gained by having to read and write less data is more than offset by the time needed to compress and uncompress data.

Comparison with uniform sea-level algorithm. One key goal of our algorithm development and engineering was to process a forecast from the Danish Meteorological Institute in less than 6 hours. Since we fulfilled this goal, and since it is hard to obtain large detailed terrain datasets (along with forecast data) for other areas, just like experimenting with Danish forecast data from other times than the `bodil` data would yield the same running time, we did not perform further experiments to investigate the running time of our algorithm.

To evaluate the quality of the output of the algorithm, we would have liked to compare the output to data from actual flooding events. However, obtaining such reliable and large scale data proved hard. In the lack of better options, we instead compared the output of our algorithm to that produced by the algorithm of Danner et al. [2], which computes floods based on the assumption that all water cells have the same elevation. We refer to this algorithm as `UniformFlood`. Algorithm `UniformFlood` computes for every cell c in T the minimum value t such that c gets flooded when the sea level rises uniformly by t meters. Therefore, in a single execution `UniformFlood` computes flood predictions over many scenarios of sea-level rise, yet in all these scenarios the sea level has uniform elevation.

We ran an implementation of `UniformFlood` on `denmark` using 64 GB of memory (the same amount as for the preprocessing of our algorithm). We did so because `UniformFlood` can be viewed as a preprocessing routine whose output encodes all flood scenarios of uniform sea-level rise. In this favorable setting, the implementation of `UniformFlood` took 1 day, 11 hours and 1 minutes.

To compare the output of our algorithm with the one produced by `UniformFlood`, we extracted from the output of `UniformFlood` the regions that appear as flooded for certain values of a uniform sea-level rise. More specifically, we considered the highest and lowest sea-level values that appear in `bodil`. For each of these values, we extracted the parts of the terrain that appear flooded according to `UniformFlood`. The outputs are presented in Figure 2, together with the output of our algorithm. We see that `UniformFlood` either overestimates or underestimates the size of the flooded regions on the terrain. For example, Sjælland (the largest island of Denmark) is surrounded by water cells whose elevations range from 0.2 m to 2.0 m. For these two values, the flooded regions produced by `UniformFlood` differed substantially from the output of our algorithm. For the largest value of uniform sea-level rise, we see that `UniformFlood` overestimates which regions could get flooded, since it produces flooded areas even on coastal parts where the sea level is very low. In particular, in such flooded areas there is no flood path connecting any terrain cell u to a water cell with elevation of 2.0 m. Similarly, for the smallest value, some coastal regions are not flooded in the output of method `UniformFlood`, even though these regions have lower elevations than the adjacent water cells.



Comparison with uniform sea level 0.2 m. Comparison with uniform sea level 2.0 m.

Figure 2: Compaison of output of our algorithm with the output of `UniformFlood`. In pink are areas flooded in the output of both algorithms, in blue areas flooded only in the output of our algorithm, and in green areas flooded only in the output of `UniformFlood`.

References

[1] A. Aggarwal and J. S. Vitter. *The Input/Output Complexity of Sorting and Related Problems.* Communications of the ACM, 31(9):1116–1127, 1988.

[2] A. Danner, T. Mølhave, K. Yi, P. K. Agarwal, L. Arge, and H. Mitasova. *TerraStream: from Elevation Data to Watershed Hierarchies.* In Proceedings of the 15th Annual ACM International Symposium on Advances in Geographic Information Systems, page 28. ACM, 2007.

[3] Andrew Danner. *I/O Efficient Algorithms and Applications in Geographic Information Systems.* PhD thesis, Dept. of Comp. Sci., Duke University, 2006.

[4] Dainsh Geodata Agency. Elevation Model of Denmark:Terræn (0.4 meter grid). `http://eng.gst.dk`, 2015.

[5] Dainsh Meteorological Institute. Sea-Level Forecast. `http://dmi.dk`, 2016.

[6] J. F. O'Callaghan and D. M. Mark. *The Extraction of Drainage Networks from Digital Elevation Data.* Computer vision, graphics, and image processing, 28(3):323–344, 1984.

[7] J. O. Domingue and S. K. Jenson. *Extracting Topographic Structure from Digital Elevation Data for Geographic Information-System Analysis.* Photogrammetric Engineering and Remote Sensing, 54(11):1593–1600, 1988.

[8] Jeffrey Scott Vitter. *Algorithms and Data Structures for External Memory.* Foundations and Trends in Theoretical Computer Science, 2(4):305–474, 2008.

[9] L. Arge, A. Danner, H. Haverkort, and N. Zeh. *I/O-Efficient Hierarchical Watershed Decomposition of Grid Terrain Models.* In Progress in Spatial Data Handling, pages 825–844. Springer, 2006.

[10] L. Arge, J. S. Chase, P. Halpin, L. Toma, J. S. Vitter, D. Urban, and R. Wickremesinghe. *Efficient Flow Computation on Massive Grid Terrain Datasets.* GeoInformatica, 7(4):283–313, 2003.

[11] L. Arge, K. G. Larsen, T. Mølhave, and F. van Walderveen. *Cleaning Massive Sonar Point Clouds.* In Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems, pages 152–161. ACM, 2010.

[12] L. Arge, L. Toma, and J. S. Vitter. *I/O-Efficient Algorithms for Problems on Grid-Based Terrains.* Journal of Experimental Algorithmics (JEA), 6:1, 2001.

[13] L. Arge, M. Rav, S. Raza, and M. Revsbæk. *I/O-Efficient Event Based Depression Flood Risk.* In Proceedings of the 19th Workshop on Algorithm Engineering and Experiments (ALENEX), 2017.

[14] L Peter Deutsch. DEFLATE compressed data format specification version 1.3. `https://tools.ietf.org/html/rfc1951`, 1996.

[15] Lars Arge. *External Memory Data Structures.* In Handbook of massive data sets, pages 313–357. Springer, 2002.

[16] P. K. Agarwal, L. Arge, and K. Yi. *I/O-Efficient Batched Union-find and its Applications to Terrain Analysis.* In Proceedings of the 22nd Annual Symposium on Computational Geometry (SoCG), pages 167–176. ACM, 2006.

[17] Y. -J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. *External-Memory Graph Algorithms.* In ACM-SIAM Symposium on Discrete Algorithms (SODA), volume 95, pages 139–149, 1995.

A Appendix: Computing candidate cells

In practice we (in contrast to the definition of the non-uniform sea-level flooding problem) encounter situations where the input sea-level forecast raster S has a coarser resolution than the terrain raster T. As described in Section 3, we in such cases need to compute a set D_c of defined cells in S for each boundary cell c in T (*candidate cells for* c) considered capable of flooding c directly.

To describe how the candidate cells are defined and computed, we assume for simplicity that although T and S have different resolution they line up perfectly, that is, that there are no partial overlaps between cells in S and T: For any cell c in T and any cell r in S, the interior of the region represented by c is either fully contained in r, or it lies entirely outside r. It is easy to modify our definition and algorithm to remove this assumption. We say that a cell c in T is *adjacent* to a cell r in S if the two corresponding geographic regions share a point or the interior of c lies inside r; if c falls entirely inside r, then we say that c is *enclosed* in r.

Now when T and S have the same resolution (as in the definition of the non-uniform sea-level flooding problem), the candidate cell(s) of a boundary cell c in T is simply the adjacent cell(s) in S. When they do not have the same resolution we cannot simply just let the candidate cell(s) of c be the adjacent cell(s) in S because the adjacent cell(s) might not have defined value(s). This is because the different resolutions lead to different coastlines in S and T. Refer to Figure 3. Therefore, we may need to find another set of candidate cells in S, not adjacent to c, that we consider to be able to directly flood c. Intuitively, we will define the candidate set of c to be the closest defined cell(s) in S, where we measure distance as the number of cells in S on a path p from a cell r in S adjacent to c; in other words, we intuitively want to assign r a sea-level height value of the closest defined cell(s) in S. However, defining and computing the candidate/closest cell(s) is complicated by the fact that we only want to consider paths p in S that do not cross the terrain, that is, cells that are defined in T. This in particular means that boundary cells in T enclosed in the same cell r in S could be assigned different candidates, e.g. if there from one boundary cell c_1 only is a non-terrain crossing path to one of the four sides of S while there from another boundary cell c_2 only is a non-terrain crossing path to another of the four sides of S, in which case c_1 and c_2 should likely be assigned different candidates (closest cells) of S.

To formalize the above intuition, we define an undirected graph G that captures the allowed paths on S (paths that do not cross the terrain T). Consider a cell r in S. If r is defined, then all boundary cells in T enclosed in r should have r as their candidate. If r is undefined and there are boundary cells in T enclosed in r, then we want to compute the candidates for each connected region R of undefined cells in T enclosed in r, and assign the candidates to all boundary cells of T adjacent to a cell in R. To do so, we construct $G = (V, E)$ as follows: V consists of a vertex corresponding to each connected region of undefined cells in T that are enclosed by the same cell in S; if the cell in S is defined, then the corresponding vertices are marked as source vertices. Two vertices in V are connected with an edge in E, if the corresponding regions of undefined cells in T are enclosed in two adjacent cells in S and there exist two cells in T, one in each region, that are adjacent (that is, if the regions are connected in T and span two adjacent cells in S). With this definition of G, the candidates of any boundary cell in T adjacent to a region corresponding to vertex u in G should be equal to the cells in S corresponding to the source vertices with the closest distance to u in G. Thus, what remains is to compute the closest source vertices for every non-source vertex G.

Figure 3: Illustration of the different coastlines in the `denmark` and `bodil` datasets. Defined cells in `denmark` are black, defined cells in `bodil` are gray, and regions that are undefined in both rasters are white.

To compute the closest source vertices we use an algorithm similar to the Bellman-Ford shortest path algorithm. For each vertex v, we maintain a set that contains the currently extracted candidates for v, and the distance of these candidates to v. We denote the current set of candidates for v by $S(v)$, and their distance from v by $d(v)$. Initially, for each source vertex v we set $d(v) = 0$ and $S(v) = \{w\}$ where w is the cell in S the connected region in T corresponds to v is enclosed in. For

each non-source vertex v, we initially set $d(v)$ to infinity and $S(v) = \emptyset$. We consider multiple rounds, and in each round we scan all the edges in G. When we encounter an edge $e = \{u, w\}$ in a certain round such that $d(u) < d(w)$, then we perform the updates $S(w) = S(u) \cup S(w)$ and $d(w) = d(u) + 1$. Since in each round we process every edge exactly once, and since after the i'th round $d(v)$ is either i or infinity for each vertex v, such an update can happen only in two cases, namely when just before the update $d(w)$ is infinite or $d(w) = d(u) + 1$. In the first case, we get $S(w) = S(u)$ and in the second case $S(w)$ increases its set of candidates without changing $d(w)$. We repeat the rounds where we process all edges until we encounter a round where no update happens (at most $|V|$ times). After that, for each vertex v we find the boundary cells in T adjacent to the region corresponds to v and associate those with the candidates stored in $S(v)$.

The correctness of our algorithm follows almost immediately from the correctness of the Bellman-Ford algorithm. To analyze the algorithm we assume that each connected region has a constant number of candidates. To construct G, we find the graph vertices by computing connected components. This can be done in $O(\text{scan}(N))$ I/Os using existing I/O-efficient algorithms [11, 12]. We then perform a scan to construct the graph edges. This process requires $O(\text{scan}(N))$ I/Os. To extract the candidates for each node v, we scan all edges G in rounds and update $S(v)$ and $d(v)$. This takes $O(k \cdot \text{scan}(|G|) + \text{scan}(N))$ I/Os in total, where $|G|$ is the combinatorial size of G and k is the length of the longest shortest path between any source and non-source node in G.