Algorithms for Computing Prominence on Grid Terrains

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

Mark De Berg
Department of Computer Science
TU Eindhoven, The Netherlands
mdberg@win.tue.nl

Constantinos Tsirogiannis
MADALGO*
Aarhus University, Denmark
constant@madalgo.au.dk

Abstract

In many archaeological applications it is important to measure how prominent a location on a terrain is, compared to its surroundings. Formally, the *prominence* of a location p with respect to a given window W centered at p is defined as the percentage of the terrain area within W that has a lower or equal elevation than p. Usually one wants to compute the prominence of as many locations as possible, distributed over the whole terrain. We present efficient algorithms for the case where the terrain is given as a grid $\mathcal{G}$ consisting $\sqrt{n} \times \sqrt{n}$ cells, the window W is a square consisting of $(2r + 1) \times (2r + 1)$ cells, and the goal is to compute the prominence of all cells in G. We analyse the running times of our algorithms under the assumption that neighboring grid cells do not differ too much in elevation, relative to the accuracy of the elevation measurements. Under this assumption, we obtain the following results.

(a) an algorithm that solves this problem in $O(n \log r)$ time and which performs only random accesses in memory, comparisons, and additions.
(b) an algorithm that runs in $O(n\sqrt{\log r})$ time and which uses the full functionality of the RAM computational model.

We implemented two variants of algorithm (a) and we evaluated the performance of these implementations. Our experiments show that the algorithm is very efficient in practice.

1 Introduction

Throughout the centuries people have carried out certain activities on locations with special morphological properties. For example, castles were often built on hilltops, because such locations have high visibility and are easy to defend. Today, landscape archaeologists try to analyse the morphology of landscapes in order to understand why specific locations were selected for building certain monuments. Conversely, they want to identify locations on a terrain that are potential hotspots for discovering hidden monuments. In other words, they want to identify locations that may have been important for people in the past because of their morphological properties.

*Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

To measure the importance of a specific location on a terrain, archaeologists consider a variety of properties, including visibility, reachability, and proximity to water sources. One of the most interesting properties that archaeologists consider is *prominence*. Intuitively, a location is considered prominent if it has a relatively high elevation compared to its surroundings. More formally, given a location p on a terrain and a *window* W that surrounds p, the prominence of p is defined as the fraction of the terrain area within W that has a lower or equal elevation than p [4]. A natural choice for the window W is a circular region of a given radius, centered at p. Another option is to use a square window centered at p. Archaeologists consider locations of high prominence as hotspots for monuments that reflect a higher social status [3]. Hence, evaluating prominence[1] is an important part in several archaeological case studies [3, 6] .

In recent years, archaeologists perform a considerable part of their analyses in a computer-based environment. Instead of analyzing the morphology of a terrain on site, they process digital terrain models using GIS software. The most popular digital terrain model used today is the so-called *digital elevation model* (DEM). A DEM, or *grid terrain*, represents a terrain as a regular grid of square cells on the xy-plane in which each cell is assigned an elevation. In archaeological case studies it is important to compute the prominence of as many locations as possible, distributed over the whole terrain. When using a grid terrain $\mathcal{G}$, this translates to the following problem: for every cell c in $\mathcal{G}$ we want to compute the fraction of the cells in $\mathcal{G}$ that fall within a window centered at c and that have lower or equal elevation than c.

A trivial way to do this is to explicitly compare, for each cell $c \in \mathcal{G}$, the elevation of c to the elevation of every other cell within the window centered at c. However, this approach is very inefficient in practice. Indeed, consider a square window of size 100m $\times$ 100m; windows used in archaeological case studies typically have at least this size. Since terrain data sets nowadays often have a resolution of 1m or less, this means such a window would contain 10,000 cells. Hence, the aforementioned trivial algorithm would require 10,000 comparisons for each grid cell in the terrain. Since a complete terrain dataset may consist of hundreds of millions of cells, it becomes clear that the trivial approach is infeasible. Surprisingly, to our knowledge, no other approach has been developed to compute prominence on terrains other than the trivial algorithm, thus forcing archaeologists to work on small terrains or use very coarse grids.

Our Results Inspired by the above, we designed efficient algorithms for computing the prominence of each cell on a grid terrain. In particular, we study the version of the problem where the input is a grid terrain $\mathcal{G}$ of $\sqrt{n} \times \sqrt{n}$ cells, and the window around each cell $c \in \mathcal{G}$ is a square consisting of $(2r+1) \times (2r+1)$ cells, for some given positive integer r. For this setting, we present two novel algorithms that compute the prominence of each cell in $\mathcal{G}$. We show that these algorithms are very efficient if $\mathcal{G}$ satisfies the following assumption: for any two neighboring cells $c, c' \in \mathcal{G}$ the number of distinct elevations in $\mathcal{G}$ in between the elevations of c and c' is on average a constant. Given this assumption, the first algorithm computes the prominence of all cells in $\mathcal{G}$ in $O(n \log r)$ time using only

[1]It should be stated that the term "prominence" has been used in the field of Geography to describe quite different concepts, such as the elevation difference of peaks on a terrain with respect to specific saddle points [7]. It is clear that the problem that we tackle in the present paper is not related to these alternative uses of the term.

random accesses in memory, comparisons, and additions. The second algorithm uses the full power of the word-RAM model; under the above assumption it runs in $O(n\sqrt{\log r})$ time.

We also perform an experimental evaluation of our first algorithm. (The second algorithm is rather complicated and mainly of theoretical interest.) To this end we developed two different implementations of the algorithm, which we compare to each other and to the trivial algorithm mentioned above, for various real-world terrains and window sizes. Our experiments show that, as expected, the trivial algorithm is infeasible for large window sizes. More importantly, our new algorithm performs quite well and the fastest of our two implementations scales remarkably well with respect to the window size. For instance, we can compute the prominence of all grid cells in a terrain with 116 million cells and with respect to a window of 25 million cells in less than an hour.

2 Definitions and Notation

Let $\mathcal{G}$ be a grid terrain that consists of n cells. For simplicity we assume the xy-domain of $\mathcal{G}$ is a $\sqrt{n} \times \sqrt{n}$ square grid; our results can easily be extended to terrains with rectangular domains that are not square. We assume without loss of generality that the center of grid cell $\mathcal{G}[i, j]$ has xy-coordinates (i, j). Thus $G[1, 1]$ is the bottom-left cell of $\mathcal{G}$. We refer to a collection of all cells with the same x-coordinate as a *column* of $\mathcal{G}$, and we refer to a collection of all cells with the same y-coordinate as a *row* of $\mathcal{G}$. We denote the elevation of a grid cell c by $h(c)$, and we sometimes use $x(c)$ and $y(c)$ to denote the x- and y-coordinate, respectively, of the center of the cell c. Hence, if $c = \mathcal{G}[i, j]$ then $x(c) = i$ and $y(c) = j$.

Let c be a cell in a grid $\mathcal{G}$, and let r be a positive integer. We define $W_r(c)$ to be the rectangular *window* consisting of $(2r + 1) \times (2r + 1)$ cells centered at c.[2] Now, given $\mathcal{G}$ and a value for r, the *prominence* of a grid cell c is defined as the fraction of cells within $W_r(c)$ whose elevation is at most $h(c)$. In other words, if we define

$$\text{prom}_r(c) := |\{c' : c' \in W_r(c) \text{ and } h(c') \leqslant h(c)\}|$$

as the number of cells in $W_r(c)$ with elevation at most $h(c)$, then the prominence of c is $\text{prom}_r(c)/(2r + 1)^2$. Hence, computing the prominence boils down to computing the values $\text{prom}_r(c)$, and this is what we focus on from now on.

We say that two distinct cells $G[i, j]$ and $\mathcal{G}[i', j']$ are *adjacent* (or: *neighbors*) if $|i-i'| \leqslant 1$ and $|j-j'| \leqslant 1$. For a subset R of cells in $\mathcal{G}$, we denote by $\mathcal{H}(R)$ the set of distinct elevation values of the cells in R, that is, $\mathcal{H}(R) := \{h(c) : c \in R\}$. Finally, the *contour of elevation* h_α *in* R is the subset of cells in R that have elevation h_α.

[2] More precisely, consisting of *at most* that many cells, because part of the window may fall outside the domain of $\mathcal{G}$. To simplify the description, we will from now on ignore the fact that the window may fall partially outside the grid domain. This can be ensured, for instance, by adding extra cells around $\mathcal{G}$ to obtain a grid $\mathcal{G}'$ of size $(\sqrt{n} + 2r) \times (\sqrt{n} + 2r)$, and giving these extra cells height ∞.

3 Description of Algorithms

Given a grid $\mathcal{G}$ and an integer r, we can compute the prominence of all cells in $\mathcal{G}$ using a simple range-counting approach. First, we build a 3-dimensional range tree that stores the points $(x(c), y(c), h(c))$ corresponding to the centers of all the grid cells c. Then, for each cell c we execute a range-counting query where the query range is the (partially unbounded) box $[x(c)-r : x(c)+r] \times [y(c)-r : y(c)+r] \times (-\infty : h(c)]$. The value returned by this query is the number of cells in $W_r(c)$ with elevation at most $h(c)$. Hence, if we use fractional cascading [2] in the range tree, then we can compute the prominence of all grid cells in $\mathcal{G}$ in $O(n \log^2 n)$ time in total. The amount of memory we need for storing the range tree is also $O(n \log^2 n)$.

Considering that nowadays a medium-size grid terrain easily consists of a hundred million cells or more, the described approach would require more than 200 gigabytes of memory. Therefore, this simple approach is not appropriate for the amount of data that we want to process. Next we describe a much more efficient approach.

3.1 The Coherence Assumption

We can design a more efficient algorithm for the prominence problem by taking into account the terrain morphology, and the relation between the horizontal and vertical accuracy of the elevation measurements.

In most terrains the maximum slope on the terrain surface is bounded. This implies that the elevation difference between adjacent cells is comparable to the size of the cells. For example, if we have cells of size 1m $\times$ 1m then the elevation difference between adjacent cells is typically at most, say, 2m. Moreover, the elevation values of the cells are provided with an accuracy of centimeters. In fact, from an archaeological perspective, when measuring prominence on a landscape there is no reason to compare heights with an accuracy which is higher than half a meter. Hence, we can assume that the elevation values are rounded to multiples of 50cm. Given that the elevations of adjacent grid cells do not differ by more than a constant number of meters, and given that the elevation values can be rounded such that there is a constant number of distinct elevation values per meter, we get the following assumption:

Assumption 3.1. *Let c and c' be two adjacent cells in $\mathcal{G}$, and suppose that $h(c) \leqslant h(c')$. Then the number of distinct elevation values in $\mathcal{G}$ that fall within the range $[h(c), h(c')]$ is $O(1)$, that is,*

$$|\mathcal{H}(\mathcal{G}) \cap [h(c), h(c')]| \;=\; O(1).$$

In the rest of this paper we refer to this assumption as the *Coherence Assumption*. We emphasize that the number of cells in $\mathcal{G}$ whose elevation lies in the range $[h(c), h(c')]$ can be very large; the Coherence Assumption only bounds the number of *distinct elevation values* of these cells. We also remark that the Coherence Assumption is only used in the run-time analysis of our algorithms; the assumption is not needed to guarantee their correctness.

Next we describe our algorithms for computing the prominence values of all cells on a grid $\mathcal{G}$, and we analyse it under the Coherence Assumption. Of course, some real-world terrains have steep cliffs and then the Coherence Assumption

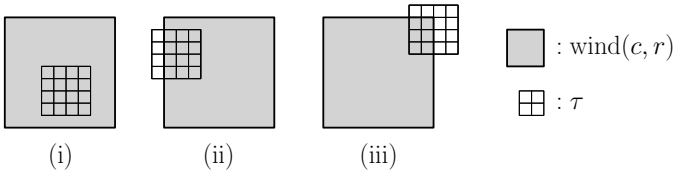

Figure 1: The possible ways that the window $W_r(c)$ of a cell c may overlap with a tile τ.

may not be satisfied. In Section 3.4 we relax the assumption without affecting the performance of the algorithms.

3.2 A practical $O(n \log r)$ algorithm

Next we describe our new algorithm *TileProminence*$(\mathcal{G}, r)$ for computing the prominence of all grid cells. Its input is a grid terrain $\mathcal{G}[1..\sqrt{n}, 1..\sqrt{n}]$ and a positive integer r that defines the window size. Its output is the value $\text{prom}_r(c)$ for each grid cell $c \in \mathcal{G}$.

The new algorithm works as follows. First, we partition $\mathcal{G}$ into a set of square *tiles*, where each tile τ consists of $r \times r$ cells. (We assume for simplicity that $\sqrt{n}$ is a multiple of r.) Note that any window $W_r(c)$ overlaps at most nine tiles. We will count, for each tile τ and each cell c such that τ overlaps with $W_r(c)$, the number of cells in $\tau \cap W_r(c)$ with elevation at most $h(c)$. Adding up these at most nine numbers for cell c then gives us $\text{prom}_r(c)$. As illustrated in Figure 1, $W_r(c)$ may overlap with a tile τ in three different ways: either (i) tile τ is fully contained in $W_r(c)$, or (ii) tile τ intersects an edge of $W_r(c)$ but it does not contain a corner of $W_r(c)$, or (iii) tile τ contains exactly one corner of $W_r(c)$. These are all possible cases, because the size of τ is less the size of $W_r(c)$ and so τ can contain at most one corner of $W_r(c)$.

Lemma 3.2. *Queries of type (i) and (ii) in a tile τ can be answered in $O(\log r)$ time per query, and the total time taken to process the entire tile τ is $O(r^2 \log r)$.*

Proof. We describe how to handle queries of type (ii) where $W_r(c)$ contains the right vertical edge of τ; handling other queries of type (ii) and handling queries of type (i) can be done in a similar manner. We process τ column by column, from left to right, meanwhile maintaining an (initially empty) dynamic balanced search tree $\mathcal{T}$. The tree $\mathcal{T}$ will store a multi-set of real numbers and it supports the following queries: given a real number h^*, return the number of elements currently stored in $\mathcal{T}$ whose value is less than or equal to h^*. Using a standard augmented balanced binary search tree, such queries can be answered in logarithmic time and insertions can be done in logarithmic time as well. To process the t-th column, $\tau[t, 1..r]$, we first insert the elevation values of all cells of this column into $\mathcal{T}$. Then, for every cell c such that $W_r(c) \cap \tau = \tau[1..t, 1..r]$ (that is, the first t columns of τ), we execute a query in $\mathcal{T}$ with the number $h^* := h(c)$. Since $\mathcal{T}$ stores at most r^2 elements, each query takes $O(\log r)$ time. Moreover, the overhead is $O(\log r)$ time per cell of τ, for inserting its elevation value into $\mathcal{T}$. □

It remains to find a way for handling queries of type (iii). Next we describe how we can perform such queries efficiently when $\mathcal{G}$ fulfils our Coherence Assumption. We show how to perform queries for the case where τ contains the top-right corner of $W_r(c)$; handling other queries of type (iii) can be done in a similar manner. We denote the the top-right cell of $W_r(c)$ by $\mathrm{tr}_r(c)$. We call c the *source* of the cell $c' := \mathrm{tr}_r(c)$, and we denote it by $\mathrm{source}_r(c')$. We use $h_{\mathrm{src}}(c')$ to denote the elevation of $\mathrm{source}_r(c')$. We number the cells in a tile τ such that $\tau[1,1]$ is its bottom-left cell.

Our approach for this case is similar to the approach used in the proof of Lemma 3.2: we scan τ column by column, from left to right, meanwhile maintaining a data structure $\mathcal{DS}$ storing information about the cells already encountered, and performing certain counting queries as we go. For each cell c' that we encounter, we wish to count the number of cells in $W_r(\mathrm{source}_r(c')) \cap \tau$ with elevation at most $h(\mathrm{source}_r(c'))$. Observe that when we scan τ and we encounter c', the cells in $W_r(\mathrm{source}_r(c')) \cap \tau$ are exactly the cells that are already encountered and, in addition, have y-coordinate less than or equal to $y(c')$. Hence, it is sufficient that our data structure $\mathcal{DS}$ stores, for each encountered cell c', the 2-dimensional point $(y(c'), h(c'))$. We call this point the *representative* of the cell c'. The operations we need are the following:

- *Insert*$(y(c'), h(c'))$: insert point $(y(c'), h(c'))$ into $\mathcal{DS}$.
- *RangeCount*(y^*, h^*): report the number of points $(y(c'), h(c'))$ in $\mathcal{DS}$ such that $y(c') \leqslant y^*$ and $h(c') \leqslant h^*$.

Our algorithm can now be stated as follows.

Algorithm ***CornerQueries***(τ)
1. **for** $i \leftarrow 1$ **to** r
2. **do for** $j \leftarrow 1$ **to** r
3. **do** $c' \leftarrow \tau[i,j]$; $c \leftarrow \mathrm{source}_r(c')$.
4. *Insert*$(y(c'), h(c'))$.
5. Add *RangeCount*$(y(c'), h(c))$ to $\mathrm{prom}_r(c)$.

Next we describe how to implement $\mathcal{DS}$ so that the operations *Insert* and *RangeCount* can be performed efficiently. One option would be to use a standard 2-dimensional range tree. Then queries and insertions take time $O(\log^2 r)$. Unfortunately we cannot use dynamic fractional cascading [5] to speed up the query and insertion time to $O(\log r)$, because counting queries do not combine with dynamic fractional cascading. Next we show that we can reduce the query and insertion time to $O(\log r)$ nevertheless. The key idea is to replace the binary search trees that form the second-level structures in standard range trees by sorted lists, and then use the fact that consecutive queries in these lists are similar (because of the Coherence Assumption) to speed up the search in these lists. Below we explain this in detail.

Recall that $\mathcal{DS}$ stores points of the form $(y(c'), h(c'))$, where c' is a cell in the tile τ. We call $y(c')$ the y-coordinate of the representative point $(y(c'), h(c'))$, and we call $h(c')$ its h-coordinate. Since the tile τ has size $r \times r$, we can assume $y(c') \in \{1, \ldots, r\}$. Let $\mathcal{P}$ be the collection of cells currently stored in $\mathcal{DS}$. (As we insert more points into $\mathcal{DS}$, the set $\mathcal{P}$ gets bigger, of course.) Then $\mathcal{DS}$ is a two-level data structure defined as follows.

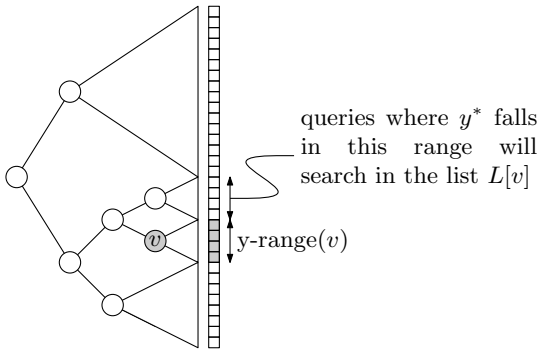

Figure 2: Illustration of the main tree of $\mathcal{DS}$. The node v will be in the subset $N(y^*)$ for all values of y^* whose search path passes through the sibling of v. To simplify the figure, only one column of the grid is shown.

- The main tree is a (static) balanced binary search tree storing the set $\{1, \ldots, r\}$ in its leaves. Each internal node v represents a range of y-values, namely the y-values stored in the leaves of the subtree rooted at v. We denote this range by y-range(v) := [y-min(v) : y-max(v)] and we use $\mathcal{P}(v)$ to indicate the subset of points $(y(c'), h(c')) \in \mathcal{P}$ such that $y(c') \in$ y-range(v).
- For each node v in the main tree, we store the set of distinct elevation values of the points in $\mathcal{P}(v)$ in a doubly-linked, sorted list $L[v]$. For each elevation value h stored in $L[v]$, we also store a counter

$$\text{count}_v[h] := |\{(y(c'), h(c')) \in \mathcal{P}(v) : h(c') = h\}|.$$

  Thus $\text{count}_v[h]$ indicates how many points in $\mathcal{P}(v)$ have elevation h.

Overall, the data structure $\mathcal{DS}$ uses $O(r^2 \log r)$ storage in the worst case. This is because each point $(y(c'), h(c'))$ that we store in $\mathcal{DS}$ is present in $O(\log r)$ sets $\mathcal{P}(v)$, and $L[v]$ stores at most $|\mathcal{P}(v)|$ elements.

For each node v in the main tree of $\mathcal{DS}$ we also store pointers $S_1(v)$ and $S_2(v)$ to certain nodes in $L[v]$, and two counters $\text{res}_1(v)$ and $\text{res}_2(v)$. These pointers and counters will help to speed up range searches. Similarly, we store two pointers $U_1(v)$ and $U_2(v)$ that will help to speed up insertions. However, before defining these pointers and counters we first explain the main idea behind introducing them.

Suppose we want to answer a query *RangeSearch*(y^*, h^*). First we search in the main tree for a collection $N(y^*)$ of $O(\log r)$ nodes v such that the sets $P(v)$ together contain exactly the points $((y(c'), h(c'))$ with $y(c') \leqslant y^*$. The nodes in $N(y^*)$ are the left children of the nodes w in the main tree where the search path to y^* turns right (plus possibly the leaf where the search ends)–see Figure 2. Observe that the values y^* for which a given node v is in $N(y^*)$ are consecutive, since these are the y-values in y-range[sibling(v)]. The next step is to count, for each node $v \in N(y^*)$, the number of points $(y(c'), h(c')) \in \mathcal{P}(v)$ with $h(c') \leqslant h^*$. This number is equal to $\sum_h \text{count}_v[h]$, where the sum is taken over all values $h \leqslant h^*$ that are stored in $L[v]$. We can compute this sum by scanning the list $L[v]$ from left to right and adding the values $\text{count}_v[h]$ of the

encountered elements, until we reach an h-value greater than h^*. However, this simple approach is not very efficient; in the worst case we have to scan all the elements in $L[v]$. The idea is therefore to not start scanning from the beginning of $L[v]$, but to start from where the previous search in $L[v]$ ended. To make this work, the already mentioned pointers $S_1(v)$ and $S_2(v)$ are used, and the counters $\text{res}_1(v)$ and $\text{res}_2(v)$. We also need a boolean AlreadySearched(v). These pointers, counters, and boolean are defined as follows.

- The boolean AlreadySearched(v) indicates whether, when handling the current column in algorithm *CornerQueries*, we already searched in the list $L[v]$. It should be initialized to `false` when we start scanning the next column in *CornerQueries*.
- The pointer $S_1(v)$ points to the element h in $L[v]$ where the previous search that was done in $L[v]$ ended.
- Let $h_1(v)$ be the h-value of the element to which $S_1(v)$ points. Then the counter $\text{res}_1(v)$ is equal to $\sum_h \text{count}_v[h]$, where the sum is taken over all values $h \leqslant h_1(v)$ that are stored in $L[v]$.
- Let y_v be the smallest y-value for which $v \in N(y_v)$. (Note that $y_v =$ y-min(sibling(v)).) Thus, the first time that we search in $L[v]$ when scanning a column is when we handle a cell c' with $y(c') = y_v$. If we already did one or more searches in $L[v]$ when handling cells in the current column—thus AlreadySearched(v) = `true`—then $S_2(v)$ points to the element h in $L[v]$ where the first search in $L[v]$ from the current column ended; otherwise $S_2(v)$ points to the element where the first search in $L[v]$ from the previous column ended.
- Let $h_2(v)$ be the h-value of the element to which $S_2(v)$ points. Then the counter $\text{res}_2(v)$ is equal to $\sum_h \text{count}_v[h]$, where the sum is taken over all values $h \leqslant h_2(v)$ that are stored in $L[v]$.

Answering a query in $\mathcal{DS}$ can now be done as follows.

Algorithm *RangeCount*(y^*, h^*)

1. Search in the main tree of $\mathcal{DS}$ to identify the set $N(y^*)$ of $O(\log r)$ nodes v such that the sets $P(v)$ together contain exactly the points $((y(c'), h(c'))$ with $y(c') \leqslant y^*$.
2. count $\leftarrow 0$
3. **for** each node $v \in N(y^*)$
4. **do if** AlreadySearched(v) = `true`
5. **then** Scan $L[v]$ starting from $S_1(v)$ until the largest h-value in $L[v]$ that is at most h^* is reached. For each of the encountered h-values, add or subtract $\text{count}_v[h]$ from res(v), depending on whether the h-values are smaller than h^* or larger than h^*.
6. Update $S_1(v)$ and $\text{res}_1(v)$ by setting $S_1(v)$ to the element in $L[v]$ where the scan ended, and setting $\text{res}_1(v)$ to res(v).
7. **else** Do the same, but now using $S_2(v)$ and $\text{res}_2(v)$ instead of $S_1(v)$ and $\text{res}_1(v)$.
8. $S_1(v) \leftarrow S_2(v)$; $\text{res}_1(v) \leftarrow \text{res}_2(v)$
9. AlreadySearched(v) $\leftarrow$ `true`
10. count $\leftarrow$ count + res(v)
11. **return** count

The correctness of *RangeCount* follows directly from the definition of the $\text{res}_1(v)$ and $\text{res}_2(v)$. For example, suppose $S_1(v)$ points to an element h_1 in $L[v]$ and

$h_1 > h^*$. Then

$$\sum_{h \leqslant h^*} \text{count}_v(h) = \text{res}_1(v) - \sum_{h^* < h \leqslant h_1} \text{count}_v(h),$$

which is exactly what the algorithm computes when in Step 5. Next we prove that the algorithm runs in the desired time bound.

Lemma 3.3. *Under the Coherence Assumption, each call to* RangeCount *in* CornerQueries(τ) *runs in $O(\log r)$ time, except for the calls when handling cells in the first column which take $O(r)$ time.*

Proof. Suppose we are handling a cell $c' = \tau[i, j]$ in *CornerQueries* and consider some node $v \in N(y^*)$, where $y^* := h_{\text{src}}(c')$. First assume AlreadySearched(v) = `true`. Then the previous search in $L[v]$ must have happened when we handled $c'' = \tau[i, j-1]$ and with $y^* := h_{\text{src}}(c'')$. Now, since c' and c'' are neighbors, the cells $\text{source}_r(c')$ and $\text{source}_r(c'')$ are neighbors as well. By the Coherence Assumption, this implies that there are only $O(1)$ distinct h-values between $h_{\text{src}}(c')$ and $h_{\text{src}}(c'')$. Hence, we scan only $O(1)$ elements in Step 5. Now assume AlreadySearched(v) = `false`. In this case we use pointer $S_2(v)$, which points to the element where the first search in $L[v]$ ended when handling the previous column. This must have been from the cell $c'' := \tau[i-1, j]$, which is again a neighbor of c'. Hence, the same argument applies. The latter argument cannot be used in the first column (where $i = 1$), but in that case $\mathcal{DS}$ contains at most r elements and so the query takes $O(r)$ time.

□

We also need to show that we can perform *Insert*-operations in $O(\log r)$ time. For this we introduce pointers $U_1(v)$ and $U_2(v)$, and a boolean AlreadyInserted(v), which are defined similarly to $S_1(v)$, $S_2(v)$, and AlreadySearched(v), except now with respect to insertions. For example, $U_1(v)$ points to the element h in $L[v]$ that was modified last by an insertion into in $L[v]$. Algorithm *Insert*$(y(c'), h(c'))$ now proceeds in much the same way as *RangeCount*:

First, we identify the nodes v for which $h(c')$ must be inserted into $L[v]$; for insertions these are the nodes v with $y(c') \in$ y-range$[v]$. Then we search in these lists for the right place to insert $h(c')$, starting from $U_1(v)$ or from $U_2(v)$ (depending on AlreadyInserted(v)). If the value $h(c')$ is already present we increment $\text{count}_v[h(c')]$, otherwise we create a new element in the list $L[v]$ with $\text{count}_v[h(c')] = 1$, and we update $U_1(v)$ (or $U_2(v)$). Finally, we need to check whether we have to increment $\text{res}_1(v)$ and/or $\text{res}_2(v)$, which can simply be done by checking if $h(c')$ is smaller than or equal to the values pointed to by $S_1(v)$ and $S_2(v)$, respectively.

The run-time analysis of the *Insert*-operation is similar to the analysis of *RangeCount*, leading to the following lemma.

Lemma 3.4. *Under the Coherence Assumption, each call to* Insert *in* CornerQueries(τ) *runs in $O(\log r)$ time, except for the calls when handling cells in the first column which take $O(r)$ time.*

We get the following theorem.

Theorem 3.5. *Let $\mathcal{G}$ be a grid terrain that consists of $\sqrt{n}\times\sqrt{n}$ cells, and let r be a positive integer. Then algorithm TileProminence computes the prominence of every cell $c \in \mathcal{G}$ with respect to the window $W_r(c)$. The algorithm uses $O(n+r^2 \log r)$ storage and under the Coherence Assumption it runs in $O(n \log r)$ time.*

Proof. According to Lemmas 3.3 and 3.4 the total time to run *CornerQueries* on on a tile τ is $O(r)$ for each of the r cells in the first column of a tile, and $O(\log r)$ for each of the remaining $O(\log^2 r)$ cells. Hence, *CornerQueries*(τ) needs $O(r^2 \log r)$ time in total, and it uses $O(r^2 \log r)$ storage for the supporting data structure $\mathcal{DS}$. By Lemma 3.2 we can also answer queries of type (i) and (ii) in $O(r^2 \log r)$ time in total. Here we need only $O(r^2)$ space for the supporting data structure. Hence, the total time to handle a tile τ is $O(r^2 \log r)$, and the storage we use is $O(r^2 \log r)$. Since the total size of all tiles is n, and the supporting data structures for a tile can be discarded after the tile has been processed, we obtain the claimed bounds.

□

3.3 An $O(n\sqrt{\log r})$ algorithm

The algorithm from the previous section computes the prominence values for a given grid using only random accesses, additions, and comparisons. Next we describe a more efficient algorithm that uses the full power of the word-RAM model.

We call the new algorithm *ContourProminence*. As in *TileProminence* we first partition $\mathcal{G}$ into square tiles consisting of $r \times r$ cells. Again, we process each tile τ separately, and for every $c \in \mathcal{G}$ such that $W_r(c)$ overlaps with τ we compute the number of cells in $W_r(c) \cap \tau$ that have elevation at most $h(c)$. Below we describe a novel way for handling these computations. Similar to the previous section, we describe how we can handle the cases where τ contains the top-right cell of $W_r(c)$; the cases where τ contains a different corner of $W_r(c)$, as well as the other ways in which $W_r(c)$ can overlap with τ, can be handled in a similar manner.

Let τ be a tile in $\mathcal{G}$, and let $\tau[i,j]$ be a cell in this tile. We denote by $P_{\mathrm{SE}}[i,j]$ the number of cells $\tau[i',j']$ for which we have $i' \leqslant i$ and $j' \leqslant j$ and $h(\tau[i',j']) \leqslant h_{\mathrm{src}}(\tau[i,j])$. Our goal is to compute for any cell $\tau[i,j]$ the value $P_{\mathrm{SE}}[i,j]$, and then add this value to $\mathrm{prom}_r(c)$, where $c = \mathrm{source}_r(\tau[i,j])$. To compute $P_{\mathrm{SE}}[i,j]$ efficiently, we can express this value in terms of the respective value of the neighboring cell $\tau[i-1,j]$, as follows. Let $\tau[i,j]$ be a cell such that $j > 1$, and suppose that $h_{\mathrm{src}}(\tau[i,j]) \geqslant h_{\mathrm{src}}(\tau[i-1,j])$. Let $\mathrm{ct}_h(i,j)$ indicate the number of cells $\tau[i',j']$ such that $i' \leqslant i$ and $j' \leqslant j$ and $\tau[i',j']$ has elevation equal to h. Also, let $\mathrm{clm}[i,j]$ denote the number of cells $\tau[i,j']$ in the i-th column of τ for which $j' \leqslant j$ and $h(\tau[i,j']) \leqslant h_{\mathrm{src}}(\tau[i,j])$. We can express the value $P_{\mathrm{SE}}[i,j]$ as follows:

$$P_{\mathrm{SE}}[i,j] = \mathrm{clm}(i,j) + \Delta(i,j) \tag{1}$$

where for $j > 1$ we have

$$\Delta(i,j) = P_{\mathrm{SE}}[i-1,j] \pm \sum_{h_{\mathrm{src}}(\tau[i-1,j]) < h \leqslant h_{\mathrm{src}}(\tau[i,j])} \mathrm{ct}_h(i-1,j),$$

where we use "+" if $h_{\text{src}}(\tau[i,j]) \geqslant h_{\text{src}}(\tau[i-1,j])$ and "−" if $h_{\text{src}}(\tau[i,j]) < h_{\text{src}}(\tau[i-1,j])$. If $j = 1$ then $\Delta(i,j) = 0$.

We can compute value $P_{\text{SE}}[i,j]$ using Equation (1) in the following way. We scan the cells in τ column-by-column from left to right, starting from cell $\tau[1,1]$. Let $\tau[i,j]$ be a cell that we are about to process, and suppose that we have already computed value $P_{\text{SE}}[i-1,j]$. Then, according to Equation (1), to compute $P_{\text{SE}}[i,j]$ we have to compute two kinds of values; first, we have to compute $\text{ct}_h(i-1,j)$ for the elevation values h that appear in τ between $h_{\text{src}}(\tau[i,j])$ and $h_{\text{src}}(\tau[i-1,j])$. Also, we have to compute $\text{clm}[i,j]$; this is equal to the number of cells in the i-th column that we have scanned so far, and which we have an elevation at most $h_{\text{src}}(\tau[i,j])$. First we present how we can compute efficiently the values of the latter kind.

To compute $\text{clm}[i,j]$ for any cell $\tau[i,j]$, we maintain a data structure L_{col} that stores the elevations of the cells on the column that we are currently processing in τ. In particular, this data structure is almost identical to the second-level list that we presented in the algorithm *TileProminence*. When processing cell $\tau[i,j]$, list L_{col} stores the elevations of the cells that we already processed in the current column, that is, L_{col} stores the elevation data for all cells $\tau[i,j']$ such that $j' < j$. When we process cell $\tau[i,j]$, we insert value $h(\tau[i,j])$ into the list, and then we perform a query with value $h_{\text{src}}(\tau[i,j])$. The result of this query is equal to number of cells that have been already processed in the current column and which have elevation at most $h_{\text{src}}(\tau[i,j])$; this number is equal to $\text{clm}[i,j]$.

The only difference with the second-level lists of the previous algorithm is that now it suffices to maintain two pointers and one counter, instead of four pointers and two counters. In particular, we maintain pointers $S[L_{\text{col}}]$ and $U[L_{\text{col}}]$, which are used in a similar way as the pointers $S_1(v)$ and $U_1(v)$ for the lists $L[v]$ used in algorithm *TileProminence*. The pointer $S[L_{\text{col}}]$ points to the element in L_{col} where the last query in L_{col} ended, while $U[L_{\text{col}}]$ points to the element in L_{col} that was modified by the last update done in this list. We also maintain a counter $\text{res}[L_{\text{col}}]$, whose value equals the result of the last query performed in L_{col}. Executing queries and insertions in L_{col} is done almost in the same manner as with the second-level lists $L[v]$ in algorithm *TileProminence*: when performing a query in L_{col}, we scan the elements of the list starting from the element pointed to by $S[L_{\text{col}}]$, and when we perform an update we start scanning from the element pointed to by $U[L_{\text{col}}]$.

To compute $P_{\text{SE}}[i,j]$ it remains to calculate $\text{ct}_h(i-1,j)$ for elevation values h with $h_{\text{src}}(\tau[i,j]) < h \leqslant h_{\text{src}}(\tau[i-1,j])$. To calculate $\text{ct}_h(i-1,j)$ we can perform a range counting query with the range $(-\infty : i-1] \times (-\infty : j]$ on the cells in τ with elevation h. In other words, we have to perform a range counting query on the contour (within τ) at elevation h.

In order to perform these queries efficiently, we use the 2-dimensional version of the range counting algorithm proposed by Chan and Patrascu [1]. The input of this algorithm is a list P of 2-dimensional points and a list Q of 2-dimensional ranges; the output is a list R storing the results of the queries in Q. If the total number of elements in P and Q is k, the running time of the algorithm is $O(k\sqrt{\log k})$. We are going to use this algorithm for computing for each contour in τ the queries that refer to this contour, and then use the results of these queries for calculating values $P_{\text{SE}}[i,j]$. Since this is an offline algorithm, the set of ranges for the queries that we want to execute should be given in advance. Therefore we proceed as follows; first, we construct a list Λ that stores the

contours in τ together with the range data for the queries that we want to perform on each contour. More precisely, for every elevation value h that is observed in τ, the list Λ stores one element λ that corresponds to the contour in τ at elevation h. Element λ in Λ stores the following information:

- The elevation $h[\lambda]$ of the contour represented by λ. The list Λ is ordered according to these values.
- A list $P[\lambda]$ of 2-dimensional points, which contains a point (i, j) for each cell $\tau[i, j]$ such that $h(\tau[i, j]) = h(h[\lambda])$.
- A list $Q[\lambda]$ containing the 2-dimensional ranges for all the counting queries that we want to perform on this contour.

To construct Λ we first have to determine the contours in τ. To this end we scan the cells τ column-by-column, alternatingly scanning bottom-to-top and top-to-bottom. In this way, we make sure that any two cells that were scanned consecutively are also adjacent on the grid. In the beginning of this scan Λ is empty. During the scan, when we process cell $\tau[i, j]$ we perform a linear search in Λ to find the element λ which has the largest value $h[\lambda]$ such that $h[\lambda] \leqslant h(\tau[i, j])$. We begin this search from the element in Λ where the last search in Λ ended. When we retrieve λ, we check if $h[\lambda] = h(\tau[i, j])$. If this is the case, then we insert point (i, j) in $P[\lambda]$, and we continue by processing the next cell in the scan. Otherwise, we create a new element λ' in Λ such that $h[\lambda'] = h(\tau[i, j])$ and $P[\lambda']$ contains only the point (i, j). To maintain that Λ is sorted, we insert the new element just after λ.

Next we determine which query ranges are to be included in each list $Q[\lambda]$. To this end we perform another scan of the cells in τ, in the same order as in the previous scan. For each cell $\tau[i, j]$ that we process in this scan, we retrieve the elements $\lambda, \lambda' \in \Lambda$ for which we have that $h(\lambda) = h_{\text{src}}(\tau[i, j])$ and $h(\lambda') = h_{\text{src}}(\tau[i - 1, j])$. Similar to what we have done so far, to retrieve these elements we search in Λ starting from the last element in the Λ that we accessed when processing the previous cell in τ. After finding elements λ and λ', for each element λ_α that appears in Λ between λ and λ' we insert range $(-\infty : i - 1] \times (-\infty : j]$ into $Q[\lambda_\alpha]$. If $h_{\text{src}}(\tau[i, j]) > h_{\text{src}}(\tau[i - 1, j])$, we also insert this range into $Q[\lambda]$, otherwise we insert the range into $Q[\lambda']$.

Having constructed Λ, we scan this list and perform the counting queries that we have accumulated for each contour in τ. During this process, we maintain a 2D array $\text{diff}[i, j]$ that we use for storing for every cell $\tau[i, j]$ the sum of the values $\text{ct}_h[i - 1, j]$. Initially every entry in this array is set to zero. More specifically, we scan the elements of Λ and for every element $\lambda \in \Lambda$ we feed $P[\lambda]$ and $Q[\lambda]$ as input to the range counting algorithm of Chan and Patrascu. This algorithm returns a list R with the results of the queries in $Q[\lambda]$. Given the output list R, we scan $Q[\lambda]$ and R simultaneously, and for every range $(-\infty : i - 1] \times (-\infty : j]$ in $Q[\lambda]$ we add the corresponding element of R to $\text{diff}[i, j]$.

After processing all the elements in list Λ, we go on and compute the values P_{SE} for all cells in τ. As mentioned, we can do this by scanning τ in a column-by-column manner starting from $\tau[1, 1]$, and for every cell $\tau[i, j]$ we calculate $P_{\text{SE}}[i, j]$ according to Equation (1) and using the values $\text{clm}[i, j]$ and $\text{diff}[i, j]$ that we have at hand.

Theorem 3.6. *Let $\mathcal{G}$ be a grid terrain that consists of $\sqrt{n} \times \sqrt{n}$ cells, and let r be a positive integer. Then Algorithm ContourProminence computes the prominence of every cell $c \in \mathcal{G}$ with respect to the window $W_r(c)$. The algorithm uses $O(n)$ storage and under the Coherence Assumption it runs in $O(n\sqrt{\log r})$ time.*

Proof. To prove the time bound, it suffices to bound the time spent for searching and inserting elements in lists L_{col} and Λ, and the time spent for all the executions of the offline range counting algorithm of Chan and Patrascu.

The searches and the insertions that we did in lists L_{col} and Λ were performed in a similar manner as with the second-level lists of algorithm *TileProminence*. Each of these operations consisted of a linear search in the list, starting from the last element that was accessed when processing an adjacent cell. In the previous section we showed that if $\mathcal{G}$ satisfies the Coherence Assumption, then each of these operations can be done in $O(1)$ time amortised. For any cell in $\mathcal{G}$ we performed a constant number of insertions and searches in L and Λ, so the total time for these operations is $O(n)$.

In order to bound the total time spent by the offline range counting algorithm, we need to bound the total number of contour points $P[\lambda]$ and ranges $Q[\lambda]$ that are stored among all elements $\lambda \in \Lambda$. There is exactly one contour point stored in Λ for every cell in τ. Also, for every cell $\tau[i,j]$ we store at most one range in Λ for every elevation value in $\mathcal{H}(\tau) \cap [h_{\text{src}}(\tau[i-1,j]), h_{\text{src}}(\tau[i,j])]$. According to Assumption 3.1, the number of these values is $O(1)$ and therefore the total amount of points and ranges stored in Λ is $O(r^2)$. Hence, the total time spent by all the executions of the offline range counting algorithm for a single tile τ in $\mathcal{G}$ is $O(r^2\sqrt{\log r})$, which yields $O(n\sqrt{\log r})$ time for processing the entire grid.

When we process a tile τ, the amount of storage needed for L_{col} and Λ is $O(r^2)$, which is $O(n)$.

□

3.4 Relaxing the Coherence Assumption

Above we described two algorithms that can compute the prominence on a grid terrain $\mathcal{G}$ very efficiently if $\mathcal{G}$ satisfies the Coherence Assumption. This assumption implies that there is no location on the input terrain around which we can observe large variations on the surface elevation. However, as already mentioned, in real-world terrains there can be cliffs and other landforms where the elevation changes abruptly. Of course, for most terrains such landforms appear in a few places, while the majority of the terrain surface has a bounded slope. To model these cases appropriately, we introduce the *Relaxed Coherence Assumption*.

Assumption 3.7. *Let $\mathcal{G}$ be a grid terrain that consists of $\sqrt{n} \times \sqrt{n}$ cells, and let $\mathcal{N}(\mathcal{G})$ be the set of all pairs of adjacent cells in $\mathcal{G}$. For any pair of cells $\{c, c'\} \in \mathcal{N}(\mathcal{G})$ let $d_h(c, c')$ denote the number of distinct elevation values in $\mathcal{G}$ that fall in the range between $h(c)$ and $h(c')$. Then*

$$\sum_{\{c,c'\} \in \mathcal{N}(\mathcal{G})} d_h(c, c') \;=\; O(n) \;.$$

The Relaxed Coherence Assumption allows pairs of adjacent cells to have a large difference in elevation, as long as this is not the case for too many pairs.

Next we prove that relaxing the Coherence Assumption in this manner does not influence asymptotic running of our algorithms.

Theorem 3.8. *Let $\mathcal{G}$ be a grid terrain consisting of $\sqrt{n} \times \sqrt{n}$ cells, and let r be a positive integer. Then algorithms TileProminence and ContourProminence compute the prominence of every cell $c \in \mathcal{G}$ with respect to the window $W_r(c)$, using $O(n+r^2 \log r)$ and $O(n)$ storage, respectively. Under the Relaxed Coherence Assumption the running times of the algorithms are $O(n \log r)$ and $O(n\sqrt{\log r})$, respectively.*

Proof. Obviously, the correctness of both algorithms is not affected by the new assumption. Next we prove the time bound for algorithm *TileProminence*. If $\mathcal{G}$ satisfies the Relaxed Coherence Assumption, this only affects the worst-case performance of insertions and queries in the second-level list of $\mathcal{DS}(\tau)$. For each of these operations the maximum number of elements that can get scanned in a single call becomes $O(r^2)$. However, according to the new version of the assumption, the total number of elements that get scanned in these lists during the entire execution of the algorithm is still $O(n)$. Therefore, the running time of the algorithm remains the same. The new assumption has no influence on the amount of storage that is used by this algorithm.

For the algorithm *ContourProminence*, relaxing the Coherence Assumption affects the time spent for handling lists L_{col} and Λ, and it also increases the amount of offline queries that we perform with the algorithm of Chan and Patrascu. Recall that for a tile τ and any two adjacent cells $\tau[i,j]$ and $\tau[i-1,j]$ we perform one 2-dimensional query for almost each elevation value in τ which falls between values $h_{\text{src}}(\tau[i,j])$ and $h_{\text{src}}(\tau[i-1,j])$. As it is the case with *TileProminence*, the total time spent during the entire algorithm for performing queries and insertions in list L_{col} is $O(n)$. Also because of the Relaxed Coherence Assumption, the total number of query ranges that we insert in Λ, and therefore the total number of offline queries that we perform, is $O(n)$. For the same reason, the asymptotical amount of storage used by the algorithm is still $O(n)$.

□

4 Experimental Evaluation

We have implemented algorithm *TileProminence* and we have evaluated thoroughly its efficiency in practice. We chose to implement this algorithm rather than algorithm *ContourProminence* because *TileProminence* uses only standard operations, and hence can be implemented in *GIS* software environments that do not support the bit-level operations needed by the algorithm of Chan and Patrascu. Next we provide details on our implementation, and later on we describe the experiments that we did in order to assess its practical performance.

In particular, we developed two different implementations of algorithm *TileProminence*; in the first implementation, we follow faithfully the description of the algorithm from Section 3.2. We refer to this implementation as the *standard* implementation. The second implementation is almost the same as the first one, except for the following detail: the secondary structures $L[v]$ associated to each node v in $\mathcal{DS}$ are implemented as arrays instead of lists. In this version of $\mathcal{DS}$, insertions and queries are performed in the same way as

described before, except that when a new entry h has to be created in $L[v]$ several existing elements may have to be shifted by one slot, to make room for h in the appropriate position in the sorted order. Depending on how much space is already allocated for $L[v]$, this could even require to build the whole array from scratch in order to have enough space for storing the new element. We call this implementation the *array-based* implementation.

At first glance, one would expect that using arrays instead of lists in `prominence_array` is very inefficient; in the worst case, in every insertion that takes place in $L[v]$ we would have to process all the elements that are already stored in the array. Yet, we chose to use this strategy due to a paradox that is observed in modern computer systems; it has been observed in practice that an array may be much more efficient than a list when handling a set of elements on which we perform a large number of linear searches, insertions and deletions [8]. This paradox can be explained from the fact that elements in an array are stored at consecutive positions in memory, while this is not usually the case for a linked list. Also, computers nowadays store data in multiple layers of cache memory. Therefore, following a link in a list is more probable to induce a cache miss than scanning several elements in an array. A cache miss results in fetching a block of data from another layer of memory. This operation is much more time-consuming than processing a set of consecutive elements in the same memory layer, hence the paradox.

Experiment settings We developed both of our implementations in C++, using the GNU g++ compiler, version 4.6.3. The experiments that we conducted were ran on a workstation with an Intel core i7-3770 CPU. This is a four-core processor with 3.40GHz per core. The main memory of this computer is 31 Gigabytes. Our implementations run on a Linux Ubuntu operating system, release 12.04.

For our experiments we used the following datasets; we used two grid datasets that each consists of 3,612×3,612 cells. The first of these datasets has grid cells of 30m resolution, and it represents a mountainous region of Mount Parnassus (Colorado, USA). The elevation values in this dataset range approximately from 1551m to 4351m. We refer to this dataset as `colorado`. The second dataset has grid cells of 2m resolution, and represents an almost flat area in the island of Funen (Denmark). The elevation range for this dataset is approximately from 17m to 108m. We refer to this dataset as `funen`. We also used one larger dataset; this dataset consists of 10,812×10,812 cells, and each cell has 10m resolution. This dataset models a region close to Smith Lake (Nevada, USA). The elevation values in this dataset range approximately from 234m to 2824m. We refer to this dataset as `nevada`. For all datasets, the elevation values are provided with an accuracy of centimeters. For our experiments, we rounded these values to multiples of 50cm; as already mentioned, in archaeological applications there is no reason to use higher accuracy. Datasets `colorado` and `nevada` were obtained from the U.S. Geological Survey (USGS) online server [9]. Dataset `funen` was created by interpolating fine-resolution point cloud data that were acquired using laser scanning over the region of Denmark.

In the first set of experiments we measured how the running times of our implementations scale with respect to the window size. More specifically, we ran our two implementations on the `colorado` and `funen` datasets, and we measured

their running time for windows of $(51 + 50k) \times (51 + 50k)$ cells, where k ranges from 0 to 19. As a point of reference, we also measured the running time of the naive prominence algorithm; this is the algorithm where we explicitly compare the elevation of each cell c in the terrain with the elevation of every cell that falls in the window around c. The results of these experiments are illustrated in Figure 3 and Figure 4. As we can see in both figures, our implementations clearly outperform the naive approach; the running time of the naive algorithm was more than three hours even for windows of 201×201 cells. In Figure 3 we see that the array-based implementation scales exceptionally well with respect to the window size, much better than the standard implementation. As we already mentioned, these two implementations differ only in the way they represent the second-level structures $L[v]$. This means that the big difference in their running times is only due to the time-consuming cache misses that take place when we scan the second-level lists in the standard implementation. Yet, when it comes to dataset `funen`, both implementations perform equally well. This possibly indicates that the standard implementation performs better on relatively flat terrains, rather than on mountainous regions. We will come back to this issue later on in this section.

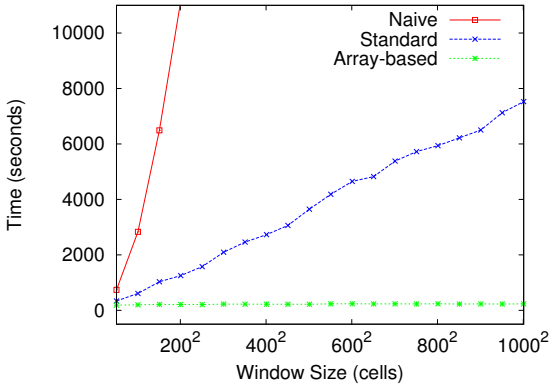

Figure 3: The running time of our two implementations and of the naive approach on the `colorado` data set, as a function of the window size.

Given the remarkable performance of the array-based implementation, we ran this program on the larger `nevada` dataset, and measured its running time for windows of $(1001+200k) \times (1001+200k)$ cells where k ranges from 0 to 20; see Figure 5. The results show again that the running time of this implementation grows very slowly as the window size increases. For the largest window size that we tried for this dataset, the array-based implementation took approximately 35 minutes to execute. This clearly shows the efficiency of this implementation.

For the second set of our experiments we measured the performance of our two implementations for different terrain sizes, given a fixed window size. For these experiments we used several terrains instances that we constructed from the `colorado` dataset in the following way: for every integer k from 1 up to 51, we extracted the region of $(1112 + 50k) \times (1112 + 50k)$ cells that contains the top-left corner of the original grid. Figure 6 shows the running times for these terrains. In this figure we also present the corresponding running times for the

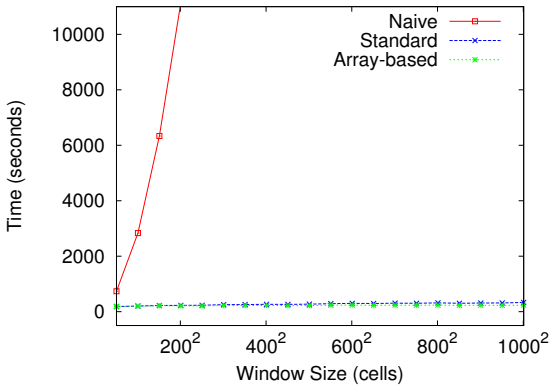

Figure 4: The running time of our two implementations and of the naive approach on the `funen` data set, as a function of the window size.

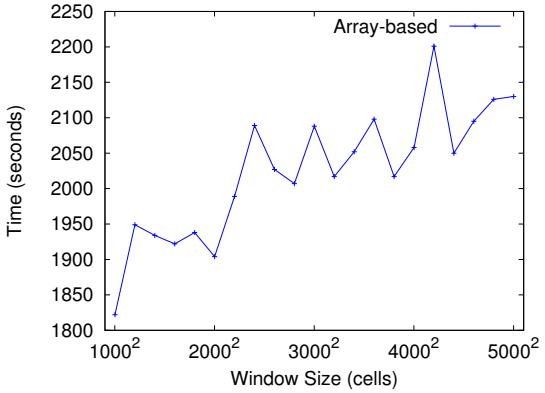

Figure 5: The running time of the array-based implementation on the `nevada` data set, as a function of the window size.

naive algorithm. As expected, the results of these experiments indicate that the running times for all programs scale almost linearly with the terrain size.

In the last experiment that we conducted we measured the average number of elements that get scanned per query and per insertion in the second-level structures $L[v]$. More precisely, for all nodes v in $\mathcal{DS}$ and for all queries that take place in the structures $L[v]$, we counted the total number of elements that get scanned during these queries, and we averaged this value over the total number of performed queries. In a similar way, we computed the average number of elements that get scanned per insertion. Note that these two numbers are the same for the standard implementation and for the array-based implementation. We evaluated the average number of scans per query and per insertion for different window sizes using the same terrain. We used the `colorado` and `funen` datasets, and for each of these terrains we measured the two averaged values for windows of $(51 + 50k) \times (51 + 50k)$ cells, where k ranges from 0 to 39. Figure 7 and Figure 8 illustrate the results of these experiments. We can make the two following observations. First, both of the measured values are quite

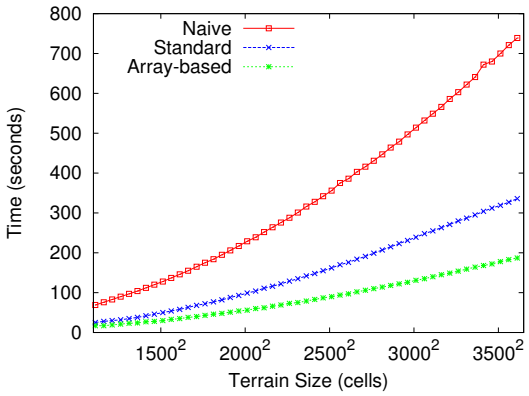

Figure 6: The running time of the two implementations of and of the naive approach, when applied to terrains of different sizes extracted from the `colorado` data set. The window used consists of 51×51 cells. Note that the terrain size is actually the square of the scale that we use on the horizontal axis.

small; for the largest window size that we tried, the average number of scans per insertion for `colorado` is approximately eight. This result is an indication that the Coherence Assumption is a realistic property that is satisfied by datasets in practice. Of course, to support this statement two datasets are not enough, and it is important to conduct experiments on terrains with different morphologies as well.

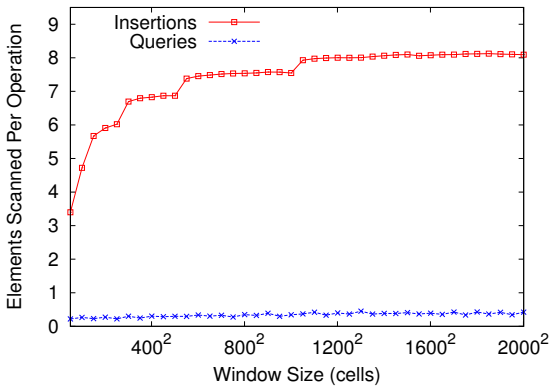

Figure 7: The average number of scanned elements per query and per insertion that our implementation has on the `colorado` data set, as a function of the window size.

The second observation that we get from Figure 7 is that, for the `colorado` dataset, the average number of scans for queries is extremely small, and evidently smaller than for insertions. In fact, for all the window sizes that we used, this value was approximately equal to 0.25. This gap between the two measured values can be possibly explained as follows. When processing a tile τ in $\mathcal{G}$, for any node v in $\mathcal{DS}$ the list $L[v]$ stores the elevations of some of the cells in τ.

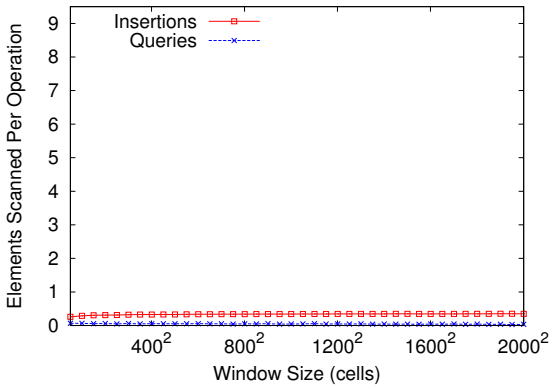

Figure 8: The average number of scanned elements per query and per insertion that our implementation has on the `funen` data set, as a function of the window size.

When we do an insertion in τ we do a linear search using an elevation value that corresponds to some cell $c \in \tau$. However, when we do a query in $L[v]$ we do a search using an elevation value that corresponds to a cell c' that falls outside τ. As c' can lie outside τ, value $h(c')$ may often fall outside the elevation range $\mathcal{H}(\tau)$. For this reason, it is possible that almost all queries performed in $L[v]$ end either at the beginning, or at the end of $L[v]$. For the `funen` dataset, the gap between the two measured values is very small. This is probably because the elevation range (and thus the number of distinct elevation values) for this terrain is small. The small number of list scans performed for both insertions and queries explains why the standard implementation performs much better for this dataset; fewer scans in a list result in fewer cache misses.

5 Conclusions and Future Work

We presented two algorithms for computing the prominence of the cells of a grid terrain $\mathcal{G}$ with respect to a square window centered around each cell. We proved that both algorithms are efficient if $\mathcal{G}$ satisfies the assumption that adjacent grid cells have a small difference in elevation, relative to the elevation accuracy on the terrain. We also developed two implementations for one of the algorithms that we presented, and we showed that one of the implementations has a remarkable performance in practice. An interesting topic for future research is to examine how our algorithms perform when the input terrain does not fit entirely in main memory. Also, it would be interesting to design efficient algorithms for computing prominence on terrains using a window of a different shape, for instance a window that is a circle.

References

[1] T.M. Chan and M. Patrascu. Counting Inversions, Offline Orthogonal Range Counting, and Related Problems. In *Proc. 21st ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 161–173, 2010.

[2] B. Chazelle and L.J. Guibas. Fractional Cascading: I. A Data Structuring Technique. *Algorithmica*, 1:133–162, 1986.

[3] G.L. Christopherson Using ARC/GRID to Calculate Topographic Prominence in an Archaeological Landscape. In *Proc. 23rd Annual ESRI International User Conference*, 2003.

[4] M. Llobera. Building Past Landscape Perception with GIS: Understanding Topographic Prominence. *Journal of Archaeological Science*, 28:1005–1014, 2001.

[5] K. Mehlhorn and S. Näher. Dynamic Fractional Cascading. *Algorithmica*, 5:215–241, 1990.

[6] J. De Reu, J. Bourgeois, P. De Smedt, A. Zwertvaegher, M. Antrop, M. Bats, P. De Maeyer, P. Finke, M. Van Meirvenne, J. Verniers and P. Crombé. Measuring the Relative Topographic Position of Archaeological Sites in the Landscape, a Case Study on the Bronze Age Barrows in Northwest Belgium. *Journal of Archaeological Science*, 38:3435–3446, 2011.

[7] G. Sinha. *Delineation, Characterization and Classification of Topographic Eminences.* PhD Thesis, SUNY University of Buffalo, 2008.

[8] Fun with C++: std::list vs. std::vector Performance. `http://yaserzt.com/blog/archives/615`.

[9] United States Geological Survey, the National Map Viewer Webpage. `http://nationalmap.gov/viewer.html`.