I/O-Efficient Path Traversal in Planar Graphs*

Craig Dillabaugh Meng He Anil Maheshwari Norbert Zeh

October 8, 2009

Abstract

We present a technique for representing bounded degree planar graphs in a succinct fashion while permitting I/O efficient traversal of paths. We represent a graph G, on N vertices, each with an associated key of $q = O(\lg N)$ bits[1], using $Nq+O(N)+o(Nq)$ bits such that a path of length K can be traversed with $O(K/\lg B)$ I/Os, where B is the disk block size. We demonstrate that our structure may be adapted to represent, with the same space bounds, a terrain modeled as a triangular-irregular network. Within this terrain we support traversal of a path which visits K triangles in the same $O(K/\lg B)$ I/O complexity. We demonstrate that a number of useful queries - reporting terrain profiles, trickle paths, and connected components - can be performed efficiently with our data structures.

1 Introduction

External memory (EM) data structures and succinct data structures both address the problem of representing very large data sets. In the EM model, the goal is to structure data that is too large to fit into internal memory in a way that minimizes the transfer of data between internal and external memory. For succinct data structures, the aim is to encode the structural component of the data structure using as little space as is theoretically possible, while still permitting efficient navigation. As both EM and succinct data structures are essentially dealing with the same fundamental problem, it seems natural to examine how the two techniques can be used together. In addition to our own research on traversal in trees [13], the only other research of which we are aware that merges these techniques is that of Munro and Clark [9], and Chien *et al.* [7], on succinct EM data structures for text indexing.

Here we develop data structures for path traversal in planar graphs. Given a bounded degree planar graph G we wish to report a path composed of K vertices in G, such that the number of I/O operations is minimized. We demonstrate practical applications of our structure showing how it can be applied to answering queries on triangular irregular network (TIN) models of a terrain.

*This work was supported by NSERC of Canada. The work was done while the second author was at the School of Computer Science, Carleton University, Canada.

[1]In this paper we use $\lg N$ to denote $\log_2 N$.

1.1 Background

In the External Memory (EM) model [2] the number of elements in the problem instance is denoted by N. Memory is divided into a two-level hierarchy, external and internal memory. External memory is assumed to have effectively infinite capacity, but accessing data elements in external memory is slow. Internal memory permits efficient operations, but the capacity of such memory is limited to $M < N$ elements. Data elements are transferred between internal and external memory in blocks of size B, where $1 < B < M/2$. In this paper we make the additional assumption that $B = \Omega(\lg N)$. We refer to such a transfer as an I/O operation. The efficiency of algorithms in the EM model is evaluated with respect to the number of I/O operations they require.

Nodine *et al.* [19] first explored the problem of blocking graphs in external memory for efficient path traversal. Path traversal is measured in terms of *blocking speed-up* - the worst case number of vertices (path length) that can be traversed before an I/O is required. The authors identify the optimal bounds on the worst case blocking speed-up for several classes of graphs. Agarwal *et al.* [1] proposed an I/O efficient algorithm that blocks a bounded degree planar graph G such that any path of length K can be traversed in $O(K/\lg B)$ I/Os.

Succinct data structures were originally proposed by Jacobson [16]; the idea is to represent data structures using space as near the information-theoretic lower bounds as possible, while still permitting efficient navigation. For example, in a succinct graph representation adjacency queries may be performed without decompressing the structure in any way. Jacobsen presented the first succinct data structure for planar graphs, and numerous efforts have been made to improve his original result, including Munro and Raman [18], and Chuang *et al.* [8]. Recently Chiang *et al.* [6] improved on these previous results to obtain a succinct data structure for simple planar graphs of $2m + 2n + o(n)$ bits, where m and n are the number of edges and vertices, respectively, in the graph.

1.2 Our Contributions

In this paper we present the following contributions:

1. In Section 3, we describe a data structure for bounded degree planar graphs that allows traversal of a path of length K using $O(\frac{K}{\lg B})$ I/Os. This matches the I/O complexity of Agarwal *et al.* [1] but improves the space complexity from $O(N \lg N) + 2Nq$ bits to $O(N) + Nq + o(Nq)$ bits, which is considerable in the context of huge datasets. (For example, with keys of constant size the improvement in the space is by a factor of $\lg N$.)

2. In Section 4, we adapt our structure to triangular-irregular networks. We demonstrate that path traversal on terrains modeled in this manner may be performed with $O(\frac{K}{\lg B})$ I/Os, where K is the number of triangles that must be crossed to complete the traversal. If storing a point requires φ bits, then our data structure can store a terrain on N points using $N\varphi+O(N)+o(Nq)$ bits. Again our I/O efficiency matches that of [1] with similar space improvement as for bounded degree planar graphs.

3. In Section 5 we describe several practical applications that make use of our representation for triangulated terrains from the previous section. We demonstrate that reporting terrain profiles and trickle paths can be performed with the $O(\frac{K}{\lg B})$ bound on I/Os. We then show that connected component queries can be performed with the $O(\frac{|T'|}{\lg B})$ I/O bound when the region being reported is convex and is of size $|T'|$ triangles. Finally, we show how non-convex regions with holes can also be reported with $O\left(\frac{|T'|}{\lg B} + h \log_B h\right)$ I/Os where h is the number of edges on the region's boundary and on any holes it contains. To answer such queries requires an additional $O(h \cdot (\varphi + \lg h))$ bits of storage. Alternatively, we demonstrate that with no additional store we can answer the same query in $O\left(\frac{|T'|}{\lg B} + h' \log_B h'\right)$ I/Os, where h' is the total number of triangles that touch all holes in, plus the boundary of, T'.

4. Finally, in Section 6 we describe how by using $o(N\varphi)$ bits of additional storage we can perform planar point location on a terrain, T, in $O(\log_B N)$ I/Os. Assymptotically this does not change the space requirement for T.

2 Preliminaries

A key data structure used in our research is a bit vector $B[1..N]$, where $B[i] \in \{0, 1\}$, that supports efficiently the operations *rank* and *select*. The operations $\texttt{rank}_1(B, i)$ and $\texttt{rank}_0(B, i)$ return the number of 1s and 0s in $B[1..i]$, respectively. The operations $\texttt{select}_1(B, r)$ and $\texttt{select}_0(B, r)$ return the position of the r^{th} occurrence of 1 and 0, respectively. The problem of representing a bit vector succinctly to support `rank` and `select` in constant time under the word RAM model with word size $\Theta(\lg N)$ bits has been considered in [16, 9, 20], and these results can be directly applied to the external memory model. The following lemma summarizes the results of Jacobson [16] and Clark and Munro [9] (part (a)), and Raman *et al.* [20] (part (b)):

Lemma 1. *A bit vector B of length N can be represented using either: (a) $N + o(N)$ bits, or (b) $\lg \binom{N}{R} + O(N \lg\lg N / \lg N)$ bits, where R is the number of 1s in B, to support the access to each bit,* `rank` *and* `select` *in $O(1)$ time (or $O(1)$ I/Os in external memory). When $R \ll N$ the value of $\lg \binom{N}{R} + O(N \lg\lg N / \lg N)$ is $o(N)$.*

Frederickson [14] developed a technique for decomposing planar graphs. He divides a planar graph into overlapping regions which contain two types of vertices, *interior* and *boundary* vertices. Interior vertices occur in a single region and are adjacent only to vertices within that region. Boundary vertices are shared among two or more regions. Frederickson's result is summarized in the following:

Lemma 2 ([14])**.** *A planar graph with N vertices can be subdivided into $\Theta(N/r)$ regions of no more than r vertices with at most $O(N/\sqrt{r})$ boundary vertices.*

3 Graph Representation

Let G be a planar graph of bounded degree d. Each vertex in G stores a q-bit key. We assume that q may be stored using at most $O(\lg N)$ bits, but q is not necessarily related to the size of the graph and may take on a small constant value. For example, q may record colours for the vertices in the graph. We perform a two-level partitioning of G [4]. This results in a subdivision of G into regions of fixed maximum size, which are subdivided into sub-regions of smaller fixed maximum size. Within the regions and sub-regions vertices fall into one of two categories, *interior* vertices, and *boundary* vertices. A vertex is interior to a region if all of its neighbouring vertices in G belong to the same region. A vertex is a region boundary if it has neighbouring vertices in G which belong to different regions. Sub-region vertices are labeled as interior or boundary in the same fashion. Due to the two level partitioning, a sub-region boundary vertex may also be a region boundary (see Fig. 1).

Consider some vertex $v \in G$. Based on [1], we define the *α-neighbourhood* of v as follows. Beginning with v, we perform a breadth-first search in G and select the first α vertices encountered. We include the subgraph induced by these vertices to form the complete α-neighbourhood of v. Analogous to the interior and boundary vertices in a region or sub-region, we define *internal* and *terminal* vertices for α-neighbourhoods. The neighbours of an internal vertex belong to the same α-neighbourhood, while terminal vertices have neighbours external to the α-neighbourhood (see Fig. 2).

In our representation of G, we store each sub-region and the α-neighbourhood of each boundary vertex. When constructing the α-neighbourhoods of sub-region boundary vertices, we add one additional constraint, that it cannot be extended beyond the region it is interior to. Collectively we refer to the sub-regions and α-neighbourhoods as *components* of the graph. The regions are not explicitly stored, but are rather a collection of their sub-region components. Each component stores a representation of a portion of G which permits traversal within that component. To enable traversal of G each vertex is assigned a unique *graph label*. Given the graph label for a vertex, we must identify a component that contains that vertex, and identify within the component's representation which vertex corresponds to the given graph label. Furthermore, for every vertex within a component we must be able to determine its graph label.

3.1 Graph Labelling

In this section we describe the labeling scheme that enables traversal across the components of the graph. The scheme is based on Bose *et al.* [4], but uses the technique of Frederickson [14] for graph decomposition.

Each vertex $v \in G$ is assigned a unique *graph label*, in addition to possibly multiple - in the case of boundary vertices - *region labels* and *sub-region labels*. We partition G into t regions. We denote the i^{th} such region by R_i. Each R_i is then subdivided into q sub-regions. We denote by q_i the number of sub-regions in region R_i, and denote the j^{th} sub-region of region R_i as $R_{i,j}$. In partitioning G the vertices on the boundary of a sub-region (or region) appear in more than one sub-region (region). Consider a boundary vertex, $v \in R_{i,j}$. We say

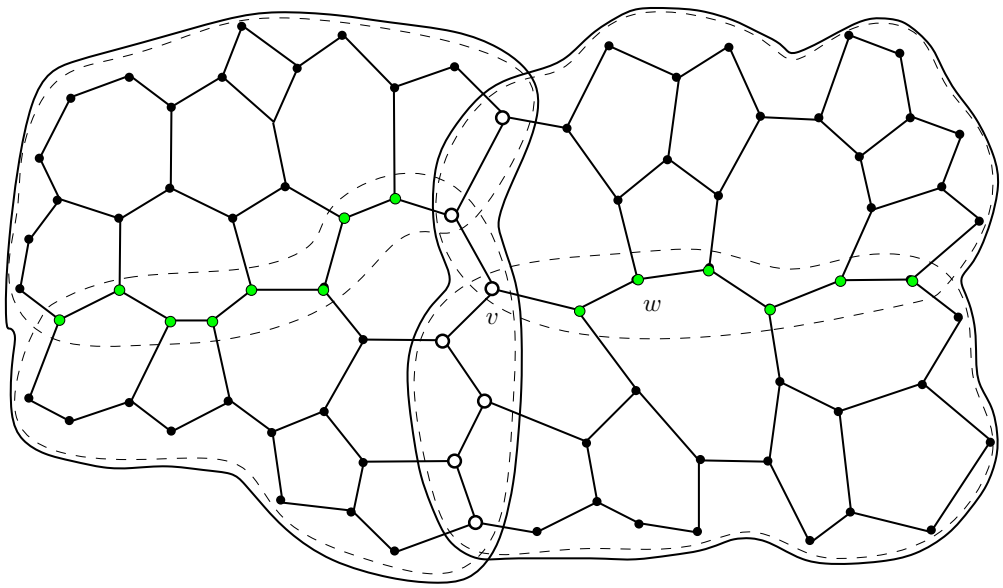


Figure 1: A graph G partitioned into two regions (delineated by solid lines), which are further subdivided into two sub-regions (delineated by dashed lines). Region boundary vertices, which are shared between the regions are marked by large hallow circles, while sub-region boundary vertices are marked by smaller, shaded circles. Regular interior vertices are marked by solid circles. The vertex v is both a region and sub-region boundary, while vertex w is a sub-region boundary, but interior to its region.

that the instance of v appearing in $R_{i,j}$ *defines* v if there is no sub-region $R_{i,h}$, such that $v \in R_{i,h}$ and $h < j$. All subsequent instances of v in any other sub-region are referred to as *duplicates*. Likewise, for a region boundary $v \in R_i$, v is a defining vertex if there is no other region R_h containing v for which $h < i$. In our labeling at the region and graph levels, our strategy is to assign defining vertices a unique label, while duplicate vertices are assigned the label of the defining vertex which they duplicate.

The encoding of a sub-region $R_{i,j}$ induces a permutation on the vertex set within the sub-region. We let the position of each vertex within this permutation serve as its sub-region label. Now consider the assignment of the vertices within the region R_i with q_i sub-regions. There are, including duplicates, a total of $\sum_{j=1}^{q_i} |R_{i,j}|$ vertices in R_i. Let n_i^b be the number of defining boundary vertices in R_i. We visit each sub-region $R_{i,j}$ for $j = 1, 2, \ldots q_i$ in turn and assign each defining vertex the next available region label from the set $\{1, \ldots, n_i^b\}$. This process is then repeated and the interior vertices are assigned labels from the set $\{n_i^b + 1, \ldots, |R_i|\}$. Duplicate vertices are assigned the label of their defining vertex.

The assignment of graph labels mirrors that of region labels. Let n^b record the total number of region boundary vertices over all regions in G. Visit each region R_i for $i = 1, 2, \ldots, t$ and assign each defining boundary vertex the next available graph label from the set $\{1, \ldots, n^b\}$. As with region labeling, we then repeat this process and assign each interior vertex the next available label from the set $\{n^b + 1, \ldots |G|\}$. This completes the labeling

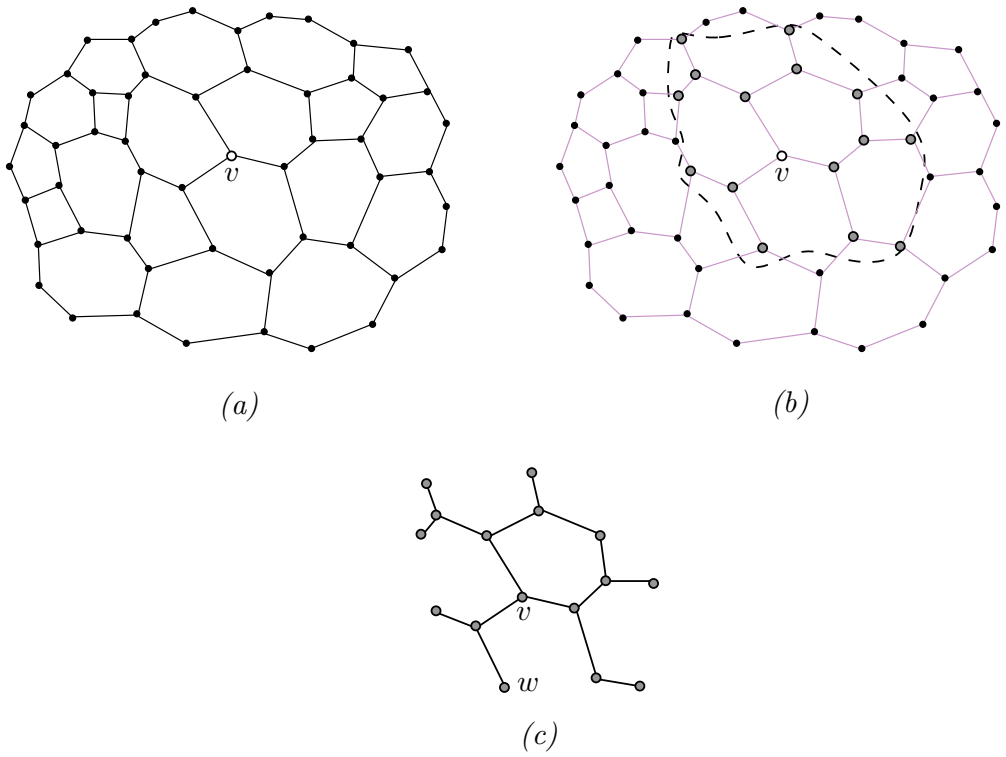


Figure 2: Creating an α-neighbourhood for a vertex v, with $\alpha = 16$. The original graph and v are shown in (a). In (b) a breadth first search is performed from v to identify the vertices in v's α-neighbourhood. Finally in (c) the α-neighbourhood subgraph, including vertices and edges is extracted. Only edges connecting vertices in the α-neighbourhood are retained. The vertex v is internal while vertex w is terminal.

procedure.

Based on this labeling scheme, observe that the graph labels assigned to all interior vertices of a region are consecutive. Likewise, the region and graph labels assigned to all interior vertices of a sub-region are consecutive.

3.2 Data Structures

We wish to have each sub-region fit in a single disk block. Denote by A the maximum number of vertices that may be stored in a block, and this becomes our maximum sub-region size. Using Lemma 2 we first divide G into regions of size $A \lg^3 N$ by setting $r = A \lg^3 N$. We further divide each region into sub-regions of size A by setting $r = A$. The sub-region $R_{i,j}$ is encoded using three data structures:

1. A compact encoding of the graph structure of $R_{i,j}$. This encoding involves a permutation of the vertices in $R_{i,j}$.

2. A bit vector, $\mathcal{B}$ of length $|R_{i,j}|$ for which $\mathcal{B}[i] = 1$ if and only if the corresponding vertex in the encoding's permutation is a boundary vertex.

3. An array of length $|R_{i,j}|$ which stores the q-bit key for each vertex.

We store two arrays $\mathcal{L}_S$ and $\mathcal{L}_R$, which record for each sub-region $R_{i,j}$, and region R_i, respectively, the graph label of the sub-region's (region's) first interior vertex.

We select A such that a sub-region of size A will fit in exactly one block in memory. However, we are only guaranteed that sub-regions will have at most A vertices, therefore some sub-regions will occupy less than a full block. We do not want to waste any bits by storing sub-regions in partially full blocks so we store sub-regions to disk in the following fashion, which is based on [13]. To begin let $\preceq_{SR}$ be a total order of the sub-regions of G. Sub-region $R_{j,k}$ comes before $R_{l,m}$ in $\preceq_{SR}$ if either $j < k$, or, if $j = k$ and $k < m$. We write the sub-regions to disk according to this order. Prior to writing a sub-region we write its *sub-region offset* value, which is the size, in vertices, of that sub-region. When we finish writing one sub-region to disk, we immediately start writing the next sub-region. If we come to a block boundary, we skip a pre-defined number of bits at the start of the next block, which we term the *block offset*, and continue writing the bits for the current sub-region. When we overrun a block in this fashion, we record the length of the overrun (in bits) in the block offset. If we are lucky and the sub-region ends at a block boundary we simply write 0 to the block offset of the next block. Since a sub-region of size A is no larger than a single block, a sub-region will never span portions of more than two blocks. We continue this process of writing sub-regions until all sub-regions have been written to disk.

Denote by Q the total number of sub-regions used to store G. To facilitate efficient lookup of sub-regions within the disk blocks we store two bit vectors:

1. Bit vector $\mathcal{B}_R[1..Q]$, where $\mathcal{B}_R[i] = 1$ iff the i^{th} sub-region in $\preceq_{SR}$ is the first sub-region in its region.

2. Bit vector $\mathcal{B}_S[1..Q]$, where $\mathcal{B}_S[i] = 1$ iff the block in which the i^{th} sub-region starts differs from sub-region $i - 1$.

We now consider the boundary vertices. There are both region and sub-region boundary vertices, and we store the α-neighbourhood for each such boundary vertex. For boundary vertex v, let G_v be the subgraph that comprises v's α-neighbourhood, which we encode as follows:

1. An encoding of the graph structure of G_v. This encoding involves a permutation of the vertices in G_v.

2. A bit array of length $|G_v|$ which marks each vertex in G_v as internal or terminal.

3. A variable which records the position of v within the permutation of the vertices of G_v.

4. An array of length $|G_v|$ which stores the key associate with each vertex.

5. An array, $\mathcal{L}_\alpha$, of length $|G_v|$ which stores:

   (a) The graph label for each vertex if v is a region boundary.

   (b) A region offset, of $\lg(A \lg^3 N)$ bits, if v is a sub-region boundary. The offset is calculated from the graph label of the first vertex in v's parent region, which is stored in $\mathcal{L}_R$.

The α-neighbourhoods of all sub-region boundary vertices are stored together in an array, $\mathcal{SR}_\alpha$. Since the size of the compact encoding of the subgraph may vary between α-neighbourhoods, we pad extra bits on neighbourhoods where necessary to ensure that the elements of $\mathcal{SR}_\alpha$ are of fixed size. This array is created by visiting each region in turn and appending the α-neighbourhoods of all sub-region boundary vertices to $\mathcal{SR}_\alpha$, ordered by region label. Additionally, we store a bit vector $\mathcal{D}$ of length N where $\mathcal{D}[i] = 1$ if the vertex with graph label i is a sub-region boundary (a 0 bit is recorded for region boundaries and interior vertices).

Lemma 3. *The data structures described above store a bounded degree planar graph G on N vertices, each associated with a $q = O(\lg N)$-bit key, in $O(N) + Nq + o(Nq)$ bits.*

Proof. First consider the space required to store the sub-regions. We denote by c the number of bits, per vertex, required to store the sub-graph and boundary bit-vector. The exact value of this constant depends on the chosen succinct representation for the graph, plus one additional bit per vertex for the boundary bit-vector. We have assumed blocks of size $B \lg N$ bits, and for the sake of simplicity, that $c+q$ divides $B \lg N$ evenly. Thus $cA + qA = B \lg N$ bits and the number of elements a sub-region can store becomes:

$$A = \frac{B \lg N}{c+q} \tag{1}$$

There are $\Theta(N/A)$ subregions with $O\left(N/\sqrt{A}\right)$ boundary vertices. To store a single copy of each vertex we require $N(c+q)$ bits, which accounts for all internal vertices plus the defining copy of each boundary vertex. We must still account for the additional space required to store the $O\left(N/\sqrt{A}\right)$ boundary vertices, which we do as follows:

$$\begin{aligned} O\left(\frac{N}{\sqrt{A}}\right)(c+q) &= O\left(\frac{N}{\sqrt{A}}\right)c + O\left(\frac{N}{\sqrt{A}}\right)q \\ &= o(N) + o(Nq) \end{aligned} \tag{2}$$

In evaluating the number of bits used to store the sub-regions we must finally consider the cost of storing the bit arrays $\mathcal{B}_R$ and $\mathcal{B}_S$, plus the block and sub-region offsets when packing the sub-regions to disk. $\mathcal{B}_R$ and $\mathcal{B}_S$ are both bit arrays of length bounded by the number of sub-regions of which there are $O(N/A)$, and as such require $o(N)$ bits by Lemma

1. Each block offset requires $\lg B$ bits and there are $O(N/B)$ blocks, so block offsets use no more than $O(N/B)\lg B = o(N)$ bits. Likewise the $O(N/A)$ sub-region offsets use no more than $O(N/A)\lg A = o(N)$ bits.

Adding the $o(N)$ bit cost of storing the structures related to block packing to the $N(c+q)$ bits required to store non-duplicate vertices, and the $o(N)+o(Nq)$ bits for duplicate vertices, we have the following total cost for storing the sub-regions:

$$N(c+q) + o(Nq) + o(N) \tag{3}$$

The array $\mathcal{L}_S$ stores the N/A graph labels, each of $\lg N$ bits, of the first interior vertex in each sub-region. Noting that $q = O(\lg N)$ bits and $B = \Omega(\lg N)$ bits the number of bits required to store $\mathcal{L}_S$ is:

$$\frac{N}{A} \cdot \lg N = \frac{N}{B\lg N/(c+q)} \cdot \lg N = O(N) \tag{4}$$

The array $\mathcal{L}_R$ which stores the graph label of the first interior vertex in each regional likewise requires $\lg N$ bits per entry, and has a total of $\Theta(N/(A\lg^3 N))$ entries. Its space requirement in bits is therefore:

$$\Theta\left(\frac{N}{A\lg^3 N}\right) \cdot \lg N = \Theta\left(\frac{N}{B\lg^2 N}\right) = o(N) \tag{5}$$

We must also account for the space required to store the α-neighbourhoods for all boundary vertices. We select $\alpha = A^{\frac{1}{3}}$ as the size of the α-neighbourhoods for both region and sub-region boundary vertices. When splitting G into regions, we set $r = A\lg^3 N$. By Lemma 2, this splits G into $\Theta\left(\frac{N}{A\lg^3 N}\right)$ regions of size less than or equal to $A\lg^3 N$ vertices, with a total of $\frac{N}{\sqrt{A\lg^3 N}}$ boundary vertices. Since labels require $\lg N$ bits each, we have a total of $\lg N + q + c$ bits associated with each vertex in the α-neighbourhood of a region boundary vertex. In each α-neighbourhood we store $A^{1/3}$ vertices. Cumulatively, the number of bits required to store the α-neighbourhoods of all region boundary vertices is:

$$\frac{N}{\sqrt{A\lg^3 N}} \cdot A^{\frac{1}{3}} \cdot (\lg N + q + c) = \frac{N}{A^{\frac{1}{6}}\lg^{\frac{3}{2}} N} \cdot \Theta(\lg N) = o(N) \tag{6}$$

For sub-regions we have $r = A$, such that G is split into $\Theta\left(\frac{N}{A}\right)$ sub-regions of size less than or equal to A vertices, with a total of $O\left(\frac{N}{\sqrt{A}}\right)$ boundary vertices. The space required by $\mathcal{L}_\alpha$ to store the labels associated with each vertex in the α-neighbourhood of a sub-region boundary vertex is $\lg(A\lg^3 N)$ bits. The total space requirement in bits for the α-neighbourhoods of all sub-region boundary vertices is:

$$\begin{aligned} \frac{N}{\sqrt{A}} \cdot A^{\frac{1}{3}} \cdot (\lg(A\lg^3 N) + q + c) &= \frac{N}{A^{\frac{1}{6}}}(\lg A + 3\lg\lg N) + \frac{N}{A^{\frac{1}{6}}}(q+c) \\ &= O(N) + o(Nq) \end{aligned} \tag{7}$$

Finally the bit vector $\mathcal{D}$ has only $\Theta(N/\sqrt{A})$ 1 bits, and is of length N, so by Lemma 1(b) can be stored in $o(N)$ bits.

To summarize we require $Nq + o(Nq) + o(N)$ bits to store the sub-regions (Eq. 3) and $O(N)$ bits for $\mathcal{L}_S$ (Eq. 4). $\mathcal{L}_R$ and the α-neighbourhoods of the region boundary vertices require only $o(N)$ bits (Eqs. 5 and 6) and as such are lower-order terms. Finally, the sub-region boundary α-neighbourhoods require $O(N)+o(Nq)$ bits (Eq. 7). Together these terms yield the final space cost of $Nq + O(N) + o(Nq)$. □

3.3 Navigation

The traversal algorithm operates by loading either a sub-region or the α-neigh-bourhood of a boundary vertex and traversing that component until a boundary vertex (in the case of a sub-region) or a terminal vertex (in the case of a α-neighbourhood) is encountered, at which time the next component is loaded from memory and traversal continues. Traversal assumes that we have available a function, `step`, which given a vertex v in G and the key value of v determines where to proceeded in the traversal. A call to the `step` function can have one of three possible outcomes, termination of the traversal; selection of a neighbour of the current vertex; or, loading a new component from memory if not all of the current vertices' neighbours are in the currently loaded component.

Now we analyze the I/O complexity. We first show that within each component, labels can be reported at no additional I/O cost (Lemma 4). We then describe how we ensure that components can be identified and loaded in $O(1)$ I/Os (Lemmas 5 and 6). Finally we demonstrate that visiting a constant number of components guarantees a progress of $O(\lg A)$ steps along the path (Lemma 7).

Lemma 4. *Given a sub-region or α-neighbourhood, the graph labels of all interior (sub-regions) and internal (α-neighbourhoods) vertices can be reported without incurring any additional I/Os beyond what is required when the component is loaded to main memory.*

Proof. The encoding of a sub-region induces an order on all vertices, both interior and boundary, of that sub-region. Consider the interior vertex at position j among all vertices in the sub-region. The position of this vertex among all interior vertices may be determined by the result of $\texttt{rank}_0(\mathcal{B}, j)$. Recall that graph labels assigned to all interior vertices in a sub-region are consecutive, and therefore by adding one less this value to the graph label of the first vertex in the sub-region, the graph label of interior vertex at position j is obtained.

For the α-neighbourhood of a region boundary vertex the graph labels from $\mathcal{L}_\alpha$ can be reported directly. For the α-neighbourhoods of sub-region boundary vertices, we can determine from $\mathcal{L}_R$ the graph label of the first vertex in the parent region. This potentially costs an I/O, but we can pay for this when the component is loaded. We can then report graph labels by adding the offset stored in $\mathcal{L}_\alpha$ to the value from $\mathcal{L}_R$. □

When we arrive at a boundary/terminal vertex, conversions between labels are necessary in order to locate the next component to load. Our labeling scheme is derived from [4], and by Lemma 3.4 in their paper, conversion between these labels can be performed in $O(1)$

operations in internal memory. Extending this to external memory we have the following lemma:

Lemma 5. *There is a data structure of $o(N)$ bits such that given a graph, G, partitioned and labeled as described above, the following operations can be performed in $O(1)$ I/O operations:*

(a) Given the graph label of vertex v, compute the region R_i to which v is interior (or which defines v if it is a region boundary), and the region label of v in R_i.

(b) Given the region label of a vertex $v \in R_i$, compute the sub-region $R_{i,j}$ to which v is interior (or which defines v if it is a sub-region boundary), and the sub-region label of v in $R_{i,j}$.

(c) Given the sub-region label of vertex v in sub-region $R_{i,j}$, compute the region label of v in R_i.

(d) Given the region label of a vertex v in region R_i, compute the graph label of v in G.

The results of the previous Lemma lead to the following Lemma.

Lemma 6. *When the traversal algorithm encounters a terminal or boundary vertex v, the next component containing v in which the traversal may be resumed can be loaded in $O(1)$ I/O operations.*

Proof. Consider first the case of a boundary vertex, v, for a sub-region. The vertex may be a region or sub-region boundary. By Lemmas 5(c) and 5(d), the graph label of v can be determined in $O(1)$ I/O operations. If v is a region boundary vertex, this graph label serves as a direct index into the array or region boundary vertex α-neighbourhoods. Loading the α-neighbourhood requires an additional I/O operation.

If v is a sub-region boundary vertex, then the region and region label can be determined by Lemma 5(a). By Lemma 5 we can determine the graph label. For the sub-region boundary vertex, v, with graph label ℓ_G, we can determine position of the α-neighbourhood of v in $\mathcal{SR}_\alpha$ in $O(1)$ I/Os by $\texttt{rank}_1(\mathcal{D}, \ell_G)$. In order to report the graph labels of vertices in the α-neighbourhood, we must know the graph label of the first interior vertex of the region, which we can read from $\mathcal{L}_R$ with at most a single additional I/O operation.

Next consider the case of a terminal node in a α-neighbourhood. If this is the α-neighbourhood of a sub-region boundary vertex, the region R_i, and region label are known, so by lemma 5(d) we can determine the graph label and load the appropriate component with $O(1)$ I/O operations. Likewise for α-neighbourhoods of a region boundary vertex the graph label is obtained directly from the vertex key.

Finally, we show that a sub-region can be loaded in $O(1)$ I/Os. Assume we wish to load sub-region $R_{i,j}$. Let $r = \texttt{select}_1(\mathcal{B}_R, i)$ mark the start of the region i. We can then locate the block, b, in which the representation for sub-region j starts by $b = \texttt{rank}_1(\mathcal{B}_S, r + j)$. There may be other sub-regions stored entirely in b prior to $R_{i,j}$. We know if $\mathcal{B}_S[r + j] = 0$ then $R_{i,j}$ is the first sub-region stored in b which has its representation start in b. If this is

not the case, the result of $\texttt{select}_1(\texttt{rank}_1(\mathcal{B}_S, r+j))$ will indicate the position of r among the sub-regions stored in b. We can now read sub-region $R_{i,j}$ as follows. We load block b into memory and read the block offset which indicates where the first sub-region in b starts. If this is $R_{i,j}$, we read the sub-region offset to determine its length, and read $R_{i,j}$ into memory, possibly reading into block $b+1$ if there is an overrun. If $R_{i,j}$ is not the first sub-region starting in block b, then we note how many sub-regions we must skip from the start of b and by reading only the sub-region offsets we can jump to the location in b where $R_{i,j}$ starts. The `rank` and `select` operations require $O(1)$ I/Os, and we read portions of at most two blocks b and $b+1$ to read a sub-region, so we can load a sub-region with $O(1)$ I/Os. □

Lemma 7. *Using the data structures and navigation scheme described above, a path of length K in graph G can be traversed in $O\left(\frac{K}{\lg A}\right)$ I/O operations.*

Proof. The α-neighbourhood components are generated by performing a breadth-first traversal, from the boundary vertex, v, which defines the neighbourhood. A total of $A^{1/3}$ vertices are added to each α-neighbourhood component. Since the degree of G is bounded by d, the length of a path from v to the terminal vertices of the α-neighbourhood is $\log_d A^{1/3}$. However, for sub-region boundary vertices the α-neighbourhoods only extend to the boundary vertices of the region, such that the path from v to a terminal node may be less than $\log_d A^{1/3}$. In later case the terminal vertex corresponds to a region boundary vertex.

Without loss of generality, assume that traversal starts with an interior vertex of some sub-region. Traversal will continue in the sub-region until a boundary vertex is encountered, at which time the α-neighbourhood of that vertex is loaded. In the worst case we travel one step before arriving at a boundary vertex of the sub-region. If the boundary vertex is a region boundary, the α-neighbourhood is loaded, and we are guaranteed to travel at least $\log_d A^{1/3}$ steps before another component must be visited. If the boundary vertex is a sub-region boundary, then the α-neighbourhood is loaded, and there are again two possible situations. In the first case, we are able to traverse $\log_d A^{1/3}$ steps before another component must be visited. In this case, by visiting two components, we have progressed a minimum of $\log_d A^{1/3}$ steps. In the second case a terminal vertex in the α-neighbourhood is reached before $\log_d A^{1/3}$ steps are taken. This case will only arise if the terminal vertex encountered is a region boundary vertex. Therefore, we load the α-neighbourhood of this region boundary vertex, and progress at least $\log_d A^{1/3}$ steps along the path before another I/O will be required.

Since we traverse $\log_d A^{1/3}$ vertices with a constant number of I/Os, we visit $O\left(\frac{K}{\lg A}\right)$ components to traverse a path of length K. By Lemma 6, loading each component requires a constant number of I/O operations, and by Lemma 4 we can report the graph labels of all vertices in each component without any additional I/Os. Thus the path may be traversed in $O\left(\frac{K}{\lg A}\right)$ I/O operations. □

Theorem 1. *Given a planar graph G of bounded degree, where each vertex stores a key of q bits, there is a data structure that represents G in $Nq + O(N) + o(Nq)$ bits that permits traversal of a path of length K with $O\left(\frac{K}{\lg B}\right)$ I/O operations.*

Proof. Proof follows directly from Lemmas 3 and 7. We substitute B for A, using Eq. 1 ($A = \Omega(B)$), as this is standard for reporting results in the I/O model. □

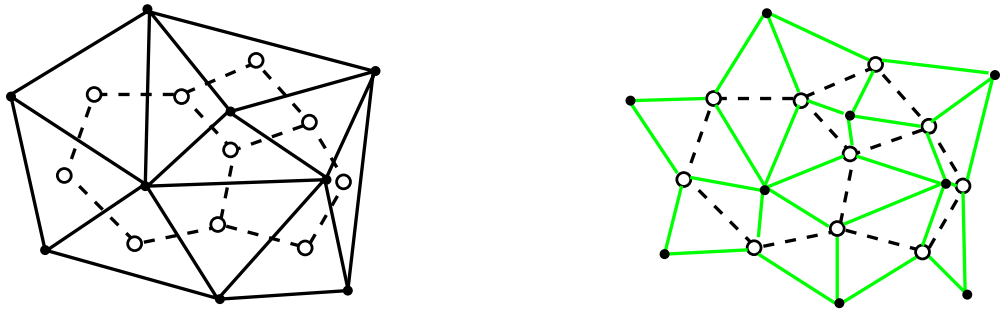
Due to the need to store keys with each vertex, its is impossible to store the graph with fewer than Nq bits. The space savings in our data structure are obtained by reducing the space required to store the actual graph representation. Agrawal *et al.* [1] do not attempt to analyse their space complexity in bits, but, assuming they use $\lg N$ bit pointers to represent the graph structure, their structure requires $O(N \lg N)$ bits for any size q. If q is a small constant, our space complexity becomes $O(N) + o(N)$ which represents a savings of $\lg N$ bits compared to the $O(N \lg N)$ space of of Agarwal *et al.*. In the worst case when $q = \Theta(\lg N)$, our space complexity is $\Theta N \lg N$ which is asympotically equivalent to that of Agarwal *et al.*. However, even in this case our structure can save a significant amount of space due to the fact that we store the actual graph structure with $O(N)$ bits (the Nq and $o(Nq)$ terms in our space requirements are related directly to space required to store keys), compared to $O(N \lg N)$ bits in their representation.

4 Representing Triangulated Terrains

Let Σ be a terrain in $\mathbb{R}^3$. Let P be a set of points on the terrain Σ with coordinates x, y, and z, where z is the elevation of the point. The triangulation T, of the point set P, is a model of Σ. By projecting T onto the x, y-plane we can view T as a planar graph with vertices being the point set P. Each triangle of T is defined by three points from P. If a point is one of the defining points for a triangle, we say that it is *adjacent* to that triangle. Two triangles are adjacent if they share a common edge (and consequently two adjacent points).

We can represent T in a compact fashion as follows. Let $G = (V, E)$ be the dual graph of T. G is a connected planar graph of bounded degree, $d = 3$, with a vertex corresponding to each triangle in T (see Fig. 3(a)). There is no vertex corresponding to the outer face, so edges along the perimeter of T do not have a corresponding edge in the dual G. Starting with G we generate an augmented planar graph $G' = (V', E')$, by adding the point set P to G such that $V' = V \cup P$. We form the edge set E' by adding an edge to E for each vertex pair (v, p) where $v \in V$ and $p \in P$ and where p is adjacent to v in T (see Fig. 3(b)). The new graph G' remains planar, but is no longer of bounded degree. However, all vertices from the original vertex set V, are still of degree at most six. In the augmented graph we refer to the vertices corresponding to triangles as *triangle vertices* and the vertices corresponding to points as *point vertices*. In a similar fashion, we refer to the edges of G as *dual edges* and those edges added to connect the point and triangle vertices as *point edges*.

For purposes of quantifying the bit cost of our data structures we denote by $\varphi = O(\lg N)$ the number of bits required to represent a point in our data structures. Most applications of terrain models do not require that a key be stored with the triangles, so we need not assume there is a q bit key associated with each dual vertex (triangle). We show that the space used by keys in our graph structure is effectively the same as the space used by the point set in our triangulation, but that if we wish to maintain keys we can do so without a significant



(a) *(b)*

Figure 3: Representation of a triangulation as a planar graph. (a) The dual graph G of a triangulation, with vertices as hallow circles and edges as dashed lines. Edges in the triangulation are shown as solid lines. (b) The augmented graph G' is shown where the point set P has been reinserted (solid dots), and newly added edges are shown as solid lines.

increase in the size of our data structures.

Lemma 8. *Given the dual graph, G of triangulation T with N vertices, the augmented graph G' has at most $2N + 2$ vertices.*

Proof. The vertex set V' includes the vertices V, of which there are exactly N, plus a vertex for each point in P. We prove by induction that there are at most $N + 2$ points in P and thus at most $2N + 2$ vertices in V'. For the base case we have a terrain T with a single triangle in which case $N = 1$ and $|P| = 3 = N + 2$. Now assume that $|P| \leq N + 2$ holds for all terrains of N triangles. Let G_N be the dual graph of a triangulation T with N triangles and $|P_N|$ points. The dual graph G_{N+1} is created by adding a single triangle to T. This new triangle has three adjacent points, however, since G_{N+1} is connected, at least one dual edge is added connecting a vertex $v \in G_N$ with the new face $w \in G_{N+1}$. In T this dual edge represents the fact that v and w are adjacent, and two of the points adjacent to w are already in P. Therefore, adding a new face adds at most one additional point to P and $|P_{N+1}| \leq (N + 1) + 2$. □

We encode G' using a succinct planar graph data structure. The encoding involves a permutation of the vertices of G'. Let $\ell(v)$, the *augmented graph label* of v, be the position of a vertex v in this permutation. We then store the point set P in an array $\mathcal{P}$ ordered by the augmented graph label, $\ell(p)$ of each point $p \in P$. Finally, create a bit vector π of length $|V'|$, where $\pi[v] = 0$ if v is a triangle vertex and $\pi[v] = 1$ if v is a point vertex. To summarize this structure we have the following Lemma.

Lemma 9. *The data structures described above can represent a terrain T composed of N triangles with $\varphi = O(\lg N)$ bit point coordinates, using $N\varphi + O(N) + o(N\varphi)$ bits, such that given the label of a triangle, the adjacent triangles and points can be reported in $O(1)$ time.*

Proof. Since the augmented planar graph is simple we can encode it with $2|E'|+2|V'|+o(|V'|)$ bits using [6]. By Lemma 8 if $|V| = N$ then $|V'| \leq 2N + 2$ which bounds the number of vertices. Each triangle vertex is connected by an edge to at most 3 other triangles, so there are at most $\frac{3}{2}$ triangle vertices. Additionally in total there are $3N$ point edges connecting the triangle vertices with point vertices so in total there are no more than $\frac{9}{2}N$ edges. Thus the augmented graph can be encoded using at most $13N + o(N)$ bits.

In [6] adjacency and degree queries can be performed in $O(1)$. We must still demonstrate that given a label we can identify the corresponding triangle vertex, and show that for a vertex we can distinguish between point vertex and triangle vertex neighbours. We assign to each triangle a unique graph label as follows. Consider the set of triangles in T, the graph label of each triangle corresponds to that of its dual vertex $v \in V$. In G' each of these vertices has an augmented graph label. The graph label of dual vertex v, and therefore the corresponding triangle, is equal to the rank of v's augmented graph label among all vertices from the set V.

Given the graph label of a triangle vertex t, t is located in G' by $\texttt{select}_0(\pi, t)$. Conversely, given the augmented graph label of a face vertex $f \in G'$, we can report the graph label of f by $\texttt{rank}_0(\pi, f)$. To report the triangles adjacent to f, simply check G' to find all neighbours of f. Since $d \leq 6$, this takes constant time. For a neighbour w the value of $\pi[w]$ indicates if w is a face or point vertex. We can recover the coordinates of a point vertex w by $\mathcal{P}[\texttt{rank}_1(\pi, w)]$.

The array $\mathcal{P}$ stores at most $N + 2$ points. If each point requires φ bits then $\mathcal{P}$ requires $N\varphi$ bits. Finally, by Lemma 1(b) we can store the bit array π such that `rank` and `select` can be performed in constant time with $N + o(N)$ bits. □

4.1 Compact External Memory TIN Representation

In this section we extend our data structures for I/O efficient traversal in bounded degree planar graphs (Section 3) to terrains. We thereby obtain a terrain representation that is compact, but which also permits efficient traversal. We represent T by its dual graph G. Since the dual graph of the the terrain (and subsequently each component) is a bounded degree planar graph, it can be represented with the data structures described in Section 3. Each component is a subgraph of G for which we generate the augmented subgraph, as described above. To represent a terrain we must store the augmented subgraph which includes points from the point set P adjacent to the triangles represented by the vertices in the sub-regions and α-neighbourhoods. In this structure, key values are optional as they are unnecessary in representing the terrain.

Lemma 10. *The space requirement, in bits, to store a component (α-neighbour-hood or sub-region) representing a terrain is within a constant factor of the space required to store the terrain's dual graph.*

Proof. We first consider the case of α-neighbourhoods. To store a terrain we require additional space to store: the points in P adjacent to the component's triangles, more space to store the augmented graph which includes more vertices and edges, and the bit-vector π. By Lemma 8 the points array is of size at most $(\alpha + 2)\varphi$ bits. Since the αq-bit array of keys

is no longer needed, storing the points incurs no additional space over that used to store the dual graph in our construction in Section 3. However, if the keys are needed we require $\alpha(\varphi + q)$ bits to store both the points and the key values. For the graph representation, again by Lemma 8, in the augmented graph we at most double the size of the vertex set and add no more than 3α edges. The graph encoding is a linear function of the number of edges and vertices and therefore the number of bits required for the graph encoding increases by a only constant factor. Finally for π we require fewer than two bits per vertex, so we can add this cost to the constant cost of representing the graph.

For sub-regions the analysis is more complex. A sub-region may be composed of multiple connected components, thus we cannot assume that there will be at most $A+2$ point vertices in the augmented subgraph. By Lemma 2 there are no more than $\Theta(N/\sqrt{A})$ boundary vertices in G. Since each connected component in a sub-region must have at least one boundary vertex, this bounds the number of distinct connected components in all sub-regions. Each sub-region is composed of at most $\Theta(\sqrt{A})$ individual components, so in the worst case there are no more than $\sqrt{A}(\sqrt{A}+2) < 2A$ point vertices. Thus, the total number of vertices in the augmented graph is less than $3A$ which adds at most an additional $6A$ edges. As demonstrated with the α-neighbourhoods, the additional storage for all data structures used to represent a sub-region increases by only a constant factor. □

One important feature of our representation is that each triangle stores very limited topological information. Consider the information available to some triangle $t \in T$ in the augmented dual graph, G'. We can determine the - up to three - triangles adjacent to t, and the three points adjacent to t, but we have no information about how these are related. For example, let w, x, and y be the points adjacent to t, and let t' be some triangle adjacent to t. We cannot directly determine if t and t' are adjacent along edge $\overline{wx}$, $\overline{xy}$, or $\overline{wy}$. The only way to acquire this information is to actually visit each of t's neighbours and thereby construct locally the topological information. Fortunately the need to visit the neighbours of a triangle in order to reconstruct its topological information will not increase the I/O costs of path traversal in the TIN. If t corresponds to an interior vertex (in a sub-region), or an internal vertex (in an α-neighbourhood), then all neighbours of t are represented in the current component. If t corresponds to a boundary (sub-region) or terminal (α-neighbourhood) vertex then the traversal algorithm already requires that a new sub-region, or α-neighbourhood, be loaded. The newly loaded component will contain the necessary neighbour information to reconstruct the topology of t. Thus, at no point during a traversal is it necessary to load a component merely to construct the topology of a triangle in an adjacent or overlapping component.

For terrains modeled using the TIN structure we have the following theorem due to Theorem 1 and Lemma 10.

Theorem 2. *Given a terrain T, where each point coordinate may be stored in φ bits, there is a data structure that represents T in $N\varphi + O(N) + o(N\varphi)$ bits, that permits traversal of a path which crosses K faces in T with $O\left(\frac{K}{\lg B}\right)$ I/O operations.*

For the case in which we wish to associate a q bit key with each triangle we also have the following theorem.

Theorem 3. *Given a terrain T, where each point coordinate may be stored in φ bits, and where each triangle has associated with it a q bit key, there is a data structure that represents T in $N(\varphi+q)+O(N)+o(N(\varphi+q))$ bits, that permits traversal of a path which crosses K faces in T with $O\left(\frac{K}{\lg B}\right)$ I/O operations.*

Proof. In our proof to Lemma 10 we demonstrated that the number of triangles (dual vertices with an associated q bit key) and points (of φ bits) are within a constant factor of each other in the worst case. Assuming this worst case does occur we are effectively assuming each triangle is associated with a $\varphi+q$ bit key in our data structures. This yields the desired space bound. □

5 Applications on TIN Models

In this section we apply our data structures for TIN models to answering several queries on triangulated surfaces. We begin with two trivial and closely related queries, reporting terrain profiles and trickle paths. These are presented to highlight the fact that even traversing a simple path on a triangulated terrain can produce useful results. We then present a slightly more complex application of our data structures in reporting connected components on a terrain. In this section we assume that for all queries we given a starting triangle, $t \in T$, as a query parameter. In Section 6 we show how to remove this assumption using $o(N\varphi)$ extra bits, such that queries take a start point p as a parameter and the triangle t containing p can be located efficiently.

5.1 Terrain Profiles and Trickle Paths

Terrain profiles are a common tool for GIS visualization. The input is a line segment, or chain of line segments possibly forming a polygon, and the output is a profile of the elevation along the line segment(s). The trickle path, or path of steepest ascent, from a point p, is the path on T that begins a p and follows the direction of steepest descent until it reaches a local minimum or the boundary of T [10]. Both queries simply involve traversing a path over T, with the fundamental difference being that in the terrain profile the path is given, whereas in reporting the trickle path the path is unknown beforehand and must be determined based on local terrain characteristics.

In analyzing these algorithms we measure the complexity of a path based on the number of triangles it intersects, which we denote by K. When a path intersects a vertex we consider all triangles adjacent to that vertex to have been intersected. Given this definition we have the following result for terrain profile and trickle path queries:

Lemma 11. *Let T be a terrain stored using our representation, then:*

(a) Given a chain of line segments, S, the profile of the intersection of S with T can be reported with $O(\frac{K}{\lg B})$ I/Os.

(b) Given a point p the trickle path from p can be reported with $O(\frac{K}{\lg B})$ I/Os.

Proof. The chain S contains of i segments denoted $s_0, s_1, \ldots s_i$. For reporting an elevation profile start at the endpoint of s_0, and let $t \in T$ be the triangle which contains this endpoint. We assume that t is given as part of the query. We calculate the intersection of s_0 with the boundary of t to determine which triangle to visit next. If at any point in reporting the query the next endpoint of the current segment s_j falls within the current triangle we advance to the next segment s_{j+1}. This procedure is repeated until the closing endpoint of s_i is visited. This query requires walking a path of exactly K triangles through T.

For the trickle path we are given triangle t and some point interior to t. The trickle path from p, and its intersection with the boundary of t, can be calculated from the coordinates of the point vertices adjacent to t. If the path crosses only the faces of triangles, but does not follow an edge or cross a point vertex, then by Theorem 2, it requires $O(K/\lg B)$ I/O's to report a path crossing K triangles. When an edge is followed, visiting both faces adjacent to that edge no more than doubles the number of triangles visited. The only possible problem occurs when the path intersects a point vertex. Such cases require a walk around the point vertex to determine which triangle (or edge) the path exits through, or if the point vertex is a local minima. In this case, the path must visit each triangle adjacent to the point vertex through which it passes, so all triangles visited during the cycle around a point vertex are accounted for in the path length K. □

5.2 Connected Component Queries

In this section we describe how connected component queries can be reported using our data structures. Recall that we denote by T a triangulation and by G the dual of T. The augmented dual graph is denoted G'. Let $\mathcal{P}$ be some property of a triangle in the triangulation T. For triangle t this property, denoted $\mathcal{P}(t)$ may either be stored as a key value, or be something that can be calculated locally, such as slope or aspect.

5.2.1 Problem Statement

We are given a convex terrain T and a triangle $t \in T$ and wish to report all triangles in the connected component $T' \subset T$ that share a common attribute or property $\mathcal{P}(t)$ (the property of t). A triangle $t' \in T$ is in T' if and only if $\mathcal{P}(t) = \mathcal{P}(t')$ and there exists a path in G from t to t' that consists of triangles for which all triangles share the same property/attribute.

5.2.2 Background

One simple technique for reporting a connected component is to start with t and perform a depth first search in G reporting all triangles $t' \in T$ for which $\mathcal{P}(t') = \mathcal{P}(t)$. However, the standard depth first search algorithm requires that we be able to mark triangles (vertices in

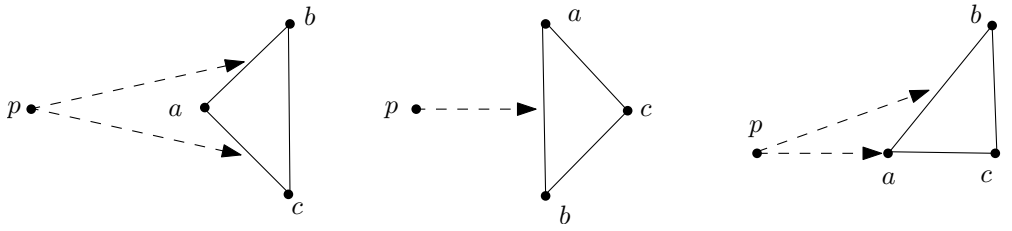


Figure 4: Visible and hidden edges on a triangle. In the figure on the left both $\overline{ab}$ and $\overline{ac}$ are visible to p. In the centre figure only $\overline{ab}$ is visible, as is the case in the figure on the right since p, a, and c are co-linear.

G) as visited to prevent vertices from being revisited by different branches of the algorithm's execution. Such an approach is infeasible with our data structures due to the duplication of the vertices of G, which would require that we mark a vertex and all duplicates whenever a vertex is visited.

To deal with this problem we employ a technique proposed by Gold and Maydell [15] that imposes a tree structure on the dual of a triangulation. This technique was later modified by De Berg *et al.* [12] to deal with planar subdivisions. In this technique we create a tree from the graph G, which includes the vertices of G and a subset of the edges of G. This tree, denoted G_T, can be rooted at an arbitrary vertex in G. The tree is not explicitly created, rather the parent-child relationships between vertices can be calculated locally at each vertex (which corresponds to a triangle in T). Since G_T is a tree rooted at t the depth-first search does not risk visiting any vertex more than once.

Before proceeding with a detailed description of how G_T is generated we will state some conventions, and define some terms. To avoid confusion between edges occurring in the triangulation and in the dual graph we refer to an edge connecting points x and y in T by $\overline{xy}$. We denote the edge in G connecting vertices u and v by (u, v).

Let $\triangle abc$ be a triangle, and p a point, in $\mathbb{R}^2$. We say that an edge $\overline{ab}$ is *visible* to p if, and only if, we can draw a line segment from p to any point on $\overline{ab}$, excluding the endpoints a and b, without intersecting any part of $\triangle abc$. If this condition does not hold we say that the edge $\overline{ab}$ is *hidden* (see Fig 4).

Let p and q be points and ℓ be a line in $\mathbb{R}^2$. Relative to p, we say that q is *behind* ℓ if the line segment pq intersects ℓ at a point other than q (in which case q is on ℓ). If pq does not intersect ℓ then the points are *in front* of ℓ relative to each other.

To generate G_T we start with the triangle $t_s \in T'$. We choose an arbitrary point s interior to t_s and build G_T based on the relationship of each triangle to this point. Consider an arbitrary vertex $v \in G$. Unless v is t_s we select from among its neighbours a unique parent triangle. Any neighbour u of v which selects v as its parent is considered a child of v. If v has neighbours which are neither its parent, nor a child, they are not considered adjacent to v. The tree G_T includes all vertices $v \in G$ and for each vertex the directed edge $(v, parent(v))$. Note that each edge in G_T is the dual of an edge in the triangulation T. If (v, u) is a parent edge in G then we also refer to the the edge $\overline{xy}$ in T adjacent to the triangles corresponding to v and u as a parent edge.

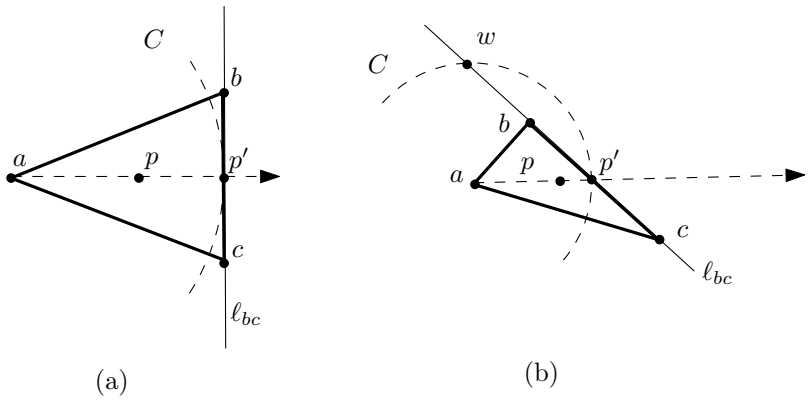


Figure 5: Proof for Observation 1.

In order to select the parent of a vertex in G_T we apply the two following rules in order. By these rules the selected parent edge is always visible to s in our terminology. Let p_1, p_2, and p_3 be the points adjacent to triangle t, where $\texttt{dist}(s, p_1) < \texttt{dist}(s, p_2) < \texttt{dist}(s, p_3)$ (if points are equidistant we break ties first by x and then by y coordinate). To determine the parent apply the following rules:

1. Consider the line segments $\overline{sp_1}$, $\overline{sp_2}$, and $\overline{sp_3}$. If one of these segments intersects an edge of t then the triangle adjacent to t along that edge is the parent of t, otherwise,

2. Let the edge $\overline{p_1p_2}$ correspond the the parent edge in G_T.

We wish to prove that G_T is a tree and that it includes as nodes every vertex in G. We will do this by showing that there is a path to every vertex $v \in G$, and that G_T contains no cycles. But first we make the following observations.

Observation 1. *Given a triangle $\triangle abc$ and a point p interior to it, for each vertex $v \in \{abc\}$, the Euclidean distance between v and p is less than the maximum Euclidean distance between v and the other two vertices.*

Proof. Without loss of generality let a be the vertex under consideration. We extend a ray from a through p. This ray intersects the edge bc at a point p'. Let C be the circle, centred at a of radius $\texttt{dist}(a, p')$. Now consider the line ℓ_{bc} which is an extension of the edge $\overline{bc}$. If ℓ_{bc} touches C at p' but does not intersect it then both $\texttt{dist}(a, p) < \texttt{dist}(a, p') < \texttt{dist}(a, b)$, and $\texttt{dist}(a, p) < \texttt{dist}(a, p') < \texttt{dist}(a, c)$ (see Fig. 5(a)). Otherwise assume, without loss of generality that $\texttt{dist}(a, b) < \texttt{dist}(a, p)$. In this case ℓ_{bc} must intersect C at some a point w and point b must lie between p' and w (see Fig.5(b)). In this case vertex c lies outside of C and therefore $\texttt{dist}(a, d) < \texttt{dist}(a, d') < \texttt{dist}(a, c)$. □

Observation 2. *If ℓ is a line containing points p, u, and u' then for an arbitrary point s if $\texttt{dist}(s, p) < \texttt{dist}(s, u)$ and u lies between u' and p on ℓ then $\texttt{dist}(s, u) < \texttt{dist}(s, u')$.*

Proof. Consider the triangle $\triangle spu'$. The vertex u is on the interior of edge $\overline{pu'}$ and thus on the interior of of $\triangle spu'$. By Observation 1 $\texttt{dist}(s,u) < \texttt{dist}(s,u')$. □

Observation 3. *If T is the triangulation of a convex region no edge on the boundary of T will be selected as the parent edge of a triangle $t \in T$.*

Proof. Let $\triangle xyz$ be a triangle in T with parent edge $\overline{xy}$ which is on the outer boundary of the triangulation T of a convex region. The edge $\overline{xy}$ is visible from s, otherwise it will not be selected as parent. Let a be a point in $\triangle xyz$ for which the line segment $\overline{sa}$ intersects $\overline{xy}$. Clearly we can select some such point since $\overline{xy}$ is visible to s. Since a is in $\triangle xyz$ we also know that a is in T, and as the region is convex the line segment $\overline{sa}$ is contained entirely in T [11]. This is a contradiction since we have claimed that $\overline{sa}$ intersects $\overline{xy}$ on the boundary of T. □

We have assumed that the domain of the triangulation T is convex. Due to Observation 3 this ensures that there is no branch of G_T that could re-enter T along the boundary after leaving. This requirement is not problematic since a non-convex triangulation can be made convex by adding a linear number of triangles. With this assumption we know prove that G_T is both connected and contains no cycles, and is therefore a tree.

Lemma 12. *For every triangle $t \in T$ there exists a path in G_T from t to the triangle t_s at which G_T is rooted.*

Proof. We prove the lemma by contradiction. Assume there exists a disconnected component, K of one or more triangles in T such that no path in G_T from t_s to any triangle $t \in K$ exists. The boundary of K consists of a set of triangle edges which form a polygon enclosing K. None of these edges on the boundary may be parent edges, otherwise the path from s_t would simply extend across that parent edge.

Let C be the chain of maximal length (measured by number of edges) along the boundary of K, for which all edges in C are visible to s. We label the vertices on C clockwise from v_0 to v_n. Each consecutive pair of vertices $\overline{v_{i-1}v_i}$ on C is a visible edge on a triangle contained in K. Consider this set of triangles. Each triangle in this set has its parent edge selected by rule 2, since rule 1 only selects parent edges for a triangle with a single visible edge. For the triangle adjacent to edge $\overline{v_{i-1}v_i}$ let the third vertex be v'_i. For each triangle $\triangle v_{i-1}v_iv'_i$ we know that either $\texttt{dist}(s,v'_i) < \texttt{dist}(s,v_i)$ or $\texttt{dist}(s,v'_i) < \texttt{dist}(s,v'_{i-1})$, in order that v'_i be on the parent edge. Consider the first triangle in the set, $\triangle v_0v_1v'_1$ and the rays $\overrightarrow{sv_0}$ and $\overrightarrow{sv_1}$. By definition v_0 is the first vertex on the chain so v'_1 does not lie above $\overrightarrow{sv_0}$, otherwise $\triangle v_0v_1v'_1$ would intersect the boundary of K (see Fig. 6). Thus $\overline{v_1v'_1}$ must lie below $\overrightarrow{sv_1}$. Since triangles in T cannot intersect each other, we can apply the same logic to the next triangle $\triangle v_1v_2v'_2$ on the chain to show that v'_2 must lie below $\overrightarrow{sv_2}$ and so on until we reach $\triangle v_{n-1}v_nv'_n$. Since C is of maximal length the boundary of K does not extend below $\overrightarrow{sv_n}$ so that $\overline{v_nv'_n}$ cannot be the parent edge of $\triangle v_{n-1}v_nv'_n$.

It may be the the case that the boundary of K encircles s such that C is convex polygonal chain around s. In this case let v_0 be the vertex on C which maximizes $\texttt{dist}(s,v_0)$. The fact that $\overline{v_{i-1}v_i}$ and $\overline{v_iv'_i}$ are both visible to s, implies that $\texttt{dist}(s,v_{i-1}) > \texttt{dist}(s,v'_i)$ by

Observation 1. Thus $\texttt{dist}(s, v_0) > \texttt{dist}(s, v_1) > \ldots > \texttt{dist}(s, v_n)$. However since C is a closed polygonal chain $v_0 = v_n$, which is impossible (see Fig. 6). □

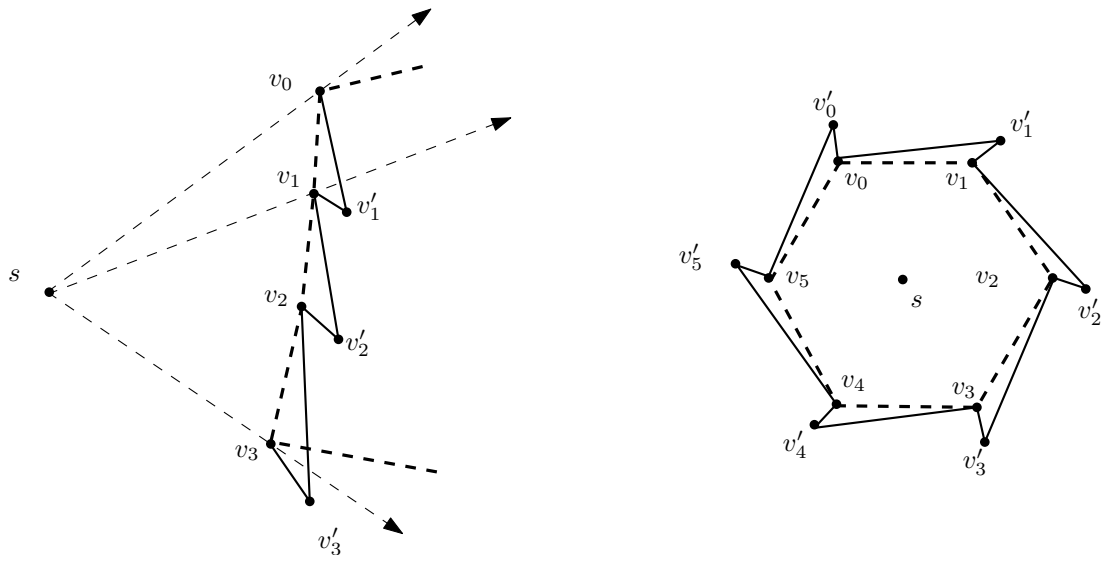


Figure 6: Proof that G_T is connected. On the left the chain C is shown. The boundary of K is the thick dashed line. On the right proof that the boundary of K cannot form a cycle around s.

Lemma 13. *There are no cycles in G_T.*

Proof. Assume that G_T contains a cycle C. In G_T a cycle corresponds to a chain of triangles, each the parent of the other. C cannot include the triangle t_s since t_s does not have a parent. By Lemma 12 G_T is connected, so there exists in G_T a path from t_s to some triangle $w \in C$. Such a path is a chain of parent edges leading from w to t_s. Without loss of generality assume that $u \notin C$ is the parent of w on P (it may be that $u = t_s$). Triangle w now has two parent edges, a parent on C and a parent on P, which is impossible because each triangle has a unique parent edge (see Fig. 7).

□

Lemma 14. *Given the triangulation T of a convex region, and some triangle $t_s \in T$ then G_T rooted at t_s is a tree.*

Proof. This proof follows directly from Lemmas 12 and 13. □

Since G_T is a tree we can report a connected component T' by selecting any triangle $t \in T'$ as t_s, selecting any point interior to t_s as s, and performing a depth first traversal on G_T rooted at t_s. We expand the traversal from any triangle $t \in T'$ for which $\mathcal{P}(t) = \mathcal{P}(t_s)$. Since each triangle contains locally all the information needed to determine its own parent/child relationships, the algorithm needs to remember the previously visited triangle to make a decision regarding where the traversal will proceed to next.

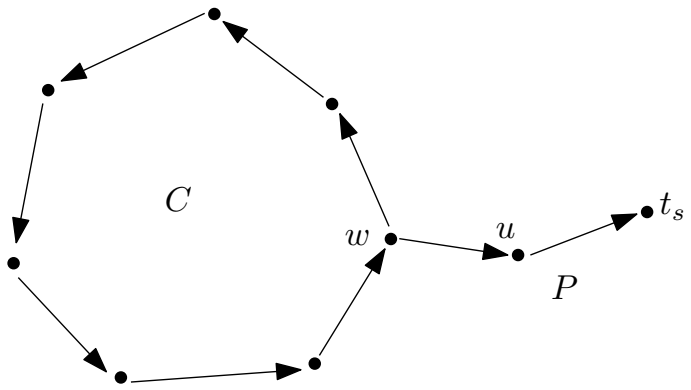


Figure 7: G_T does not contain any cycles.

Lemma 15. *Let $T' \subset T$ be a connected, convex component. Given $t_s \in T'$ we can report all triangles in T' with $\left(\frac{|T'|}{\lg B}\right)$ I/Os.*

Proof. We select t_s as the start triangle and let s be the mid-point of t.

We first demonstrate that the subtree of G_T connecting the triangles of T' to t_s, which we denote $G_{T'}$, is still a tree (it is not split into subtrees by the boundary of T'). If we apply the same logic in Observation 3 to the boundary of T' it is clear that the boundary of the convex region T' contains no parent edges. Assume some branch in G_T leaves T' and re-enters at some edge along the boundary of T'. This must occur at a parent edge, but this is impossible. Thus $G_{T'}$ is still a connected tree.

Starting with t_s as the root we perform a depth-first search in G_T, terminating the search (locally) along any branch when a triangle t' is encountered for which $\mathcal{P}(t') \neq \mathcal{P}(t_s)$. The depth-first traversal is equivalent to a walk along a path which visits each reported triangle at most three times, and each unreported triangle once. Thus, if the component contains $|T'|$ triangles the traversal is equivalent to performing a walk of length at most $4|T'|$. By Theorem 2 we can report the connected component with $O\left(\frac{|T'|}{\lg B}\right)$ I/Os. □

It can be easily shown that Lemma 15 can also apply to any star-shaped region so long as the point s is selected such that it falls in the kernel of the region. We also have the following Corollary for rectangular window queries.

Corollary 1. *Given a terrain T, a query window W, and a triangle $t \in T$ which intersects the query window, the set of triangles which intersect the query window, T_W, can be reported with $O\left(\frac{|T_W|}{\lg_d A}\right)$ I/Os.*

Proof. First note that the query window problem is equivalent to reporting a connected component where the selection property is that a triangle intersects the query window W. Care must be taken with triangles that intersect the query window boundary. We must guarantee that no triangle has a parent edge in G_T corresponding to an edge in T that lies completely outside the window W. It is easy to verify that given triangle $\triangle abc$, and the

point s, that any rectangular query window W containing s and intersecting $\triangle abc$ intersects (or contains) at least one of the visible edges of $\triangle abc$. Thus we modify rule 2, such that if rule 2 is applied we do not assign as the parent edge an edge entirely outside W. There is no need to modify rule 1 as in this case there is only one visible edge, so it must intersect W. With this modification reporting query windows is equivalent to reporting any convex connected component and the I/O complexity is $O\left(\frac{|T_W|}{\lg B}\right)$ I/Os. □

5.2.3 General Connected Components

In this section we present an algorithm that permits us to traverse components that are non-convex and possibly contain holes. By hole we refer to a connected component of triangles for which $\mathcal{P}(t) \neq \mathcal{P}(t_s)$ and for which all boundary edges on the hole border on a triangle in T'. In the connected component T', there exists a path in G_T from t to t_s. This implies that triangles in T' must be edge adjacent. For holes the opposite holds, such that two holes that touch at a vertex may be considered a single hole, and a hole touching the boundary of T' may be considered part of the boundary.

We select a triangle, t_s within T' and wish to report the connected component containing all triangles in T'. Let u, and v be vertices in G corresponding to the adjacent triangles t_u and t_v in T for which $\mathcal{P}(t_u) = \mathcal{P}(t_s) \neq \mathcal{P}(t_v)$. Let e be the edge in T that corresponds to the undirected edge (u, v) in G. The edge e is on the boundary (or perimeter) of the region T', or on the boundary of one of the holes in T'. We divide boundary edges into three classes with respect to G_T (see Fig.8):

1. e is a *wall* edge if neither the directed edge (u, v) nor the directed edge (v, u) is present in G_T.

2. e is an *entry* edge if the directed edge (v, u) is present in G_T.

3. e is an *exit* edge if the directed edge (u, v) is present in G_T.

We report the triangles in T' using the algorithm $DepthFirstTraversal$ shown in Fig. 9. We assume that we are given the starting triangle t_s. $DepthFirstTraversal$ performs a standard depth-first traversal in G_T with one important modification. If a branch of the algorithm's execution terminates at an entry boundary edge the function $ScanBoundary$ (Fig. 10) is called, which traverses the entire chain of boundary edges and recursively calls $DepthFirstSearch$ at each triangle encountered adjacent to an exit edge. The search structure $\mathcal{V}$ ensures that no boundary edge is scanned more than once. Whenever a entry edge is visited during the $ScanBoundary$ or $DepthFirstSearch$ processes it is added to the search structure $\mathcal{V}$ if it is not already present. If an entry edge is encountered which is already in $\mathcal{V}$ execution of $ScanBoundary$ is halted. Likewise when $DepthFirstTraversal$ encounters an entry edge already in $\mathcal{V}$ it does not invoke $ScanBoundary$. Figure 11 demonstrates how $DepthFirstSearch$ and $ScanBoundary$ operate.

Lemma 16. *Given a triangle $t_s \in T$ the algorithm described above reports the all triangles $t \in T'$ even when T' has holes or a non-convex boundary.*

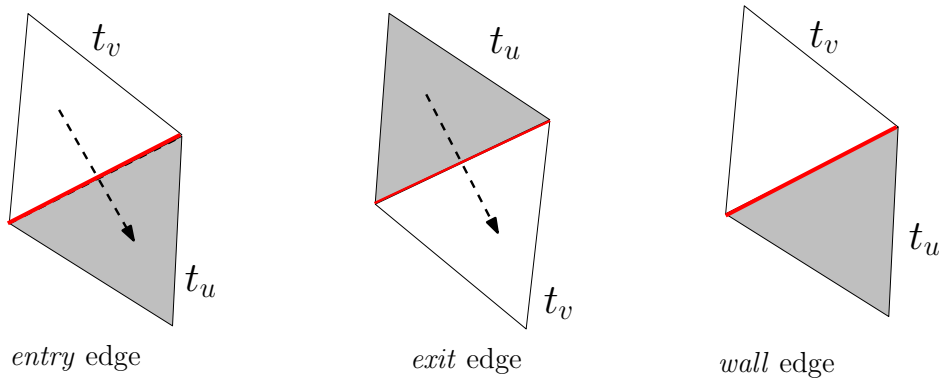


Figure 8: The three types of boundary edges. The boundary edge is drawn in red and the dashed arrow represents the directed edges (v, u) and (u, v) in G_T. The shaded region represents the triangle that belongs to T', which is t_u in each case.

Proof. Let v be a vertex in G corresponding to some triangle $t \in T'$ which is not reported by our algorithm. Since v is not reported the path $v \rightsquigarrow t_s$ in G must have one or more subpath that cuts through a hole or through the boundary of T'. Let $w \rightsquigarrow u$ be the closest such subpath to t_s. Let $w' \in T'$ be the child of w in G_T, and let $u' \in T'$ be the parent of u in G_T. The edge (w', w) corresponds to an exit edge while (u, u') corresponds to an entry edge. By definition the path $u' \rightsquigarrow t_s$ is contained wholly in T' and thus the depth-first search commencing at t_s reaches u'. If the entry edge corresponding to (u, u') has not previously been visited the *ScanBoundary* algorithm is invoked. Since *ScanBoundary* walks the entire chain of boundary edges around the boundary of the hole/component, the exit edge corresponding to (w', w) will be visited during this call to *ScanBoundary* unless it is blocked when *ScanBoundary* encounters a previously visited entry edge (this may occur if one of the recursive calls to *DepthFirstSearch* made at exit edges during *ScanBoundary* encounters the same boundary). However, if *ScanBoundary* is blocked from visiting (w', w) in such a fashion then *ScanBoundary* must have been called from the blocking entry edge. Let $s_0, s_1, \ldots, s_i$ be a sequence of such blocking calls to *ScanBoundary* along the boundary chain between (u, u') and (w', w). Clearly the last such call, s_i will result in (w', w) being visited. This same argument can be applied to any other subpaths leaving T' on $v \rightsquigarrow t_s$, and as such v is visited by *DepthFirstTraversal*. □

We evaluate the I/O efficiency of our algorithm with respect to two values. The first is the number of triangles in T' which we denote $|T'|$, and the second is the total number of boundary edges around all holes and the boundary of T'. We denote by h the total number of boundary edges. Thus I/O efficiency of our algorithms are summarized in the following lemma.

Lemma 17. *The algorithms DepthFirstTraversal reports a connected component T' with h boundary edges using $O\left(\frac{|T'|}{\lg B}\right) + O(h \log_B h)$ I/Os.*

```
Algorithm DepthFirstTraversal(t_s, s)
1.  t ← t_s
2.  do
3.      if(t has unvisited children)
4.          c ← next unvisited child of t
5.          if(P(c) ≠ P(t))
6.              e ← boundary edge corresponding to (c, t)
7.              if(e is an entry edge and e ∉ V)
8.                  ScanBoundary(e, t)
9.              endif
10.         else
11.             t ← c
12.         endif
13.     else
14.         t ← parent of t
15.     endif
16. until(t has no more unvisited children and t = t_s)
```

Figure 9: Algorithm $DepthFirstTraversal$.

Proof. Performing a the depth-first traversal is equivalent to a walk of length at most $4|T'|$, which we know can be performed in $O\left(\frac{|T'|}{\lg B}\right)$ I/Os. We must also account for the length of the paths traversed by calls to $ScanBoundary$. Any triangle in T' may be visited at most three times since it may be adjacent to no more than three different boundary chains. Thus the total additional length of the walks associated with calls to $ScanBoundary$ is bounded by $3|T'|$, which increases the length of the path traversed by a constant.

We must also account for the number of I/Os required to maintain and query the data structure $\mathcal{V}$. There are h boundary edges and in the worst case most of these edges may be entry edges. Using a B-tree or similar data structure to store $\mathcal{V}$ enables insertions and queries to be performed in $O(\log_B h)$ time. An entry edge may be added to $\mathcal{V}$ a single time, and be subsequently visited at most one additional time. Thus the total cost to maintain and query $\mathcal{V}$ is $O(h \log_B h)$. □

Finally we must account for the space used to store the search structure $\mathcal{V}$. If we store as the structure in a B-Tree we can assume that for each record we must store a φ-bit point (as each edge can be uniquely identified by the coordinates of its mid-point) in addition to a $\lg h$ bit pointer. Thus the space for this structure is $O(h \cdot (\varphi + \lg h))$ bits. In theory the size of this structure could be larger than T, but in most realistic scenarios it will be significantly smaller.

To summarize we have the following theorem to summarize our results for convex and general connected components. The space bound is from Theorem 2, plus the space for $\mathcal{V}$ for general connected components, while the I/O bounds are obtained from Lemmas 15 and

```
Algorithm ScanBoundary(e, t, s)
1.  e' ← next boundary edge in counterclockwise direction from e
2.  do
3.      if(e' is an entry edge)
4.          if(e' ∈ V)
5.              break
6.          else
7.              add e' to V
8.          endif
9.      endif
10.     if(e' is an exit edge)
11.         select triangle t adjacent to e'
12.         DepthFirstTraversal(t, s)
13.     endif
14. until(e' == e)
```

Figure 10: Algorithm *ScanBoundary*.

17.

Theorem 4. *A triangulation T, with φ-bit keys per triangle, may be stored using $N\varphi + O(N) + o(N\varphi)$ bits such that a connected component T' may be reported using $O\left(\frac{|T'|}{\lg B}\right)$ I/Os if T' is convex. If T' may be non-convex or have holes then the query requires $O\left(\frac{|T'|}{\lg B} + h \log_B h\right)$ I/Os, plus an additional $O(h \cdot (\varphi + \lg h))$ bits of storage, where h is the number of boundary edges around holes and the perimeter of T'.*

5.3 Connected Components without Additional Storage

One drawback with our technique for reporting connected components is that we must store the search structure $\mathcal{V}$ in order to complete the traversal. In this section we present a revised version of the algorithm that removes the need for this additional data structure at the cost of performing additional I/O operations.

Our technique is based on the constant memory algorithm of Bose and Morin [5], for the more general case of subdivision traversal in planar subdivisions. This paper itself is a refinement of the algorithm presented by De Berg *et al.* [12]. The strategy in both papers is to identify a single entry edge on each face. When an edge is visited while reporting the edges of a face, a check is made to determine if it is the (unique) entry edge for an adjacent face. If the edge proves to be an entry edge then the adjacent face is entered. The resulting traversal is a depth-first traversal of the faces of the subdivision. In both papers (Morin and Bose, De Berg *et al.*) the subdivision is assumed to be represented as a doubly connected edge list, or similar structure.

Bose and Morin [5] select entry edges based on a total order $\preceq_p$, on the edges of T. The position of each edge, e in $\preceq_p$, is determined based on the edge's key. The key is a 4-tuple of properties that can be calculated locally for each edge based on the edge's geometry. As with our structure the value of an edge's key depends on the value of a starting point s contained in one of the faces. To determine if an edge is the entry point of a face (that may be a hole boundary or the component boundary), we perform the following *both-ways* search. Starting at edge e the boundary is scanned in both directions until either, (a) an edge e' on either scan is encountered with a lower key than e in $\preceq_p$, or (b) the scans meet without having found any such edge. In case (b) the edge e is then selected as the entry edge for the boundary.

The fundamental result of Bose and Morin is that for a subdivision with n vertices, all faces (including edges and vertices) can be reported with $O(n \log n)$ steps. Their approach can also be directly applied to reporting a connected component of the subdivision with $O(h \log h)$ time where the h is the number of vertices in the component.

In our paper all faces are triangles, so using their technique to find the entry edge of a triangle (face) $t \in T$ can be performed in constant time. Where their 'search both ways' technique proves useful in our setting is in dealing with holes and the exterior boundary of the component. A hole, the interior of which will not be reported, may consist of one or many faces (triangles) but we treat a hole as if it were a single face. As we do not store a doubly-connected edge list, we cannot walk the edges of the hole as they do. Rather, we must walk the set of triangles that touch the boundary of the hole. This includes all triangles in T' that:

1. have an edge that is part of the boundary of a hole,

2. have as one of their defining points a point that lies on the boundary of a hole.

Let h' denote the number of triangles that must be visited to walk the boundary of all holes in, plus the external boundary of, T'. A triangle can be adjacent to at most three different holes so $h' = O(|T'|)$. Let H be some hole in our component. In order that we can apply the analysis of Bose and Morin directly in our setting, we conceptually add zero length *psuedo-edges* to H at any point on the boundary of H that is adjacent to a triangle t which touches H at a point, but which does not share an edge with H (see Fig. 12). With respect to the key values in $\preceq_p$, we set the value of a key for a pseudo-edge to ∞. Since the entry edge in any hole is the edge of minimum key value, no psuedo-edge will ever be selected as the entry edge. Given this definition of a psuedo-edge, the value h' can also be considered the sum of real and psuedo-edges over all holes and the exterior boundary of the component T'.

The following lemmas summarize results for Bose and Morin that can now be applied directly to our setting:

Lemma 18. *For a connected component T' requiring h' triangles to be visited in order to walk the boundary of all holes plus the exterior boundary of T', the both ways search technique can find entry edges for all holes (and the exterior) by visiting at most $O(h' \log h')$ steps, or $O(h' \log_B h')$ I/Os.*

Proof. Theorem 1 in Bose and Morin [] states that a planar subdivision of faces with n vertices can be traversed in $O(n \log n)$ time. By adding zero-length pseudo-edges in our setting we effectively make the set of holes and the exterior boundary equivalent to faces with h' vertices on their boundaries. Thus using the two way search can locate entry edges with $O(h' \log h')$ steps. Since we can travel $O(\log B)$ steps during such searches before incurring an I/O we can perform all such searches in $O(h' \log_B h')$ I/Os. □

Applying the both-ways search technique presented above requires only minor modifcations to our algorithms. In the *DepthFirstTraversal* algorithm (Fig. 9) at line 7 rather than check if $e \in \mathcal{V}$, we perform the both-ways search to determine if e is the unique entry edge. If this is true we then perform *ScanBoundary* starting with e. The only alteration to the *ScanBoundary* algorithm (Fig. 10) is that we can omit steps 3 through 9 since following the both-ways search we know that e is the unique entry edge for the boundary or hole.

To summarize we have the following theorem:

Theorem 5. *A triangulation T, with φ-bit keys per triangle, may be stored using $N\varphi + O(N) + o(N\varphi)$ bits such that a connected component T' may be reported using $O\left(\frac{|T'|}{\lg B} + h' \log_B h'\right)$ I/Os, where h' is the total number of triangles that touch all holes in, plus the boundary of, T'.*

6 Terrain Representation with Point Location

For the various applications that we have described in the previous section (5), we assume that a starting triangle in T is given as an input parameter to the problem. This is problematic because in real applications we will typically need to locate the triangle from which we will begin reporting the result. In this section we describe how to extend our data structures to answer point location queries efficiently.

Our point location structure is based on that of Bose *et al.* [4] as our data structures share the same two-level partition scheme. Their construction relies on the point location structure of Kirkpatrick [17] for which there is no known external memory version. Rather we use the structure of Arge *et al.* [3] which uses linear space and answers vertical ray shooting queries in $O(\log_B N)$ I/Os.

We want to design a point location structure which given a query point p_q will return the label of the sub-region containing p_q. Since we are operating in the I/O model, we can afford to perform an exhaustive search of the sub-region to locate the exact triangle containing p_q. We use a two level search structure. The first level allows us to locate the region containing p_q, while the second allows us to locate the sub-region containing p_q. As the the structure is effectively the same at each level, we will only describe the structure for locating the region in detail.

Consider the set of region boundary vertices used to partition T. These correspond to a subset of the triangles in T. We create a planar subdivision by selecting these boundary triangles and removing all triangles corresponding to the internal vertices of the regions.

We associate with each edge in this subdivision the region immediately below it. If the area below the edge is not part of T, or is perhaps a hole in T, then we mark the edge with invalid value. It may be the case for some edge that the area below it is part of a boundary triangle and therefore belongs to two or more regions. In this case we can make an arbitrary selection of any region containing that triangle. We then build the persistent B-Tree structure of Arge *et al.* [3] on this subdivision. To determine which region the query point p_q belongs to we perform a vertical ray shooting query from p_q to report the first edge encountered in our subdivision above that point. Since we associate with each edge the region below it, the result of this query yields the region containing p_q.

For each region we then build a similar structure using the sub-region boundary vertices. Thus given the region containing p_q we can then locate the sub-region containing this point by performing a second search.

Lemma 19. *By augmenting our terrain data structure with a structure using an additional $o(N\varphi)$ bits, point location queries can be answered in $O(\log_B N)$ I/Os.*

Proof. In the top level, region finding, data structure there are $O\left(\frac{N}{\sqrt{A \lg^3 N}}\right)$ region boundary vertices. For each such vertex we insert at most three edges into our search structure. Associated with each edge we must store 2φ bits for the endpoints, plus a region label requiring $\lg\left(\frac{N}{A \lg^3 N}\right)$ bits. Thus the total space in bits required by this structure will be:

$$\begin{aligned} O\left(\frac{N}{\sqrt{A \lg^3 N}}\right) \cdot \left(2\varphi + \lg\left(\frac{N}{A \lg^3 N}\right)\right) &= O\left(\frac{N\varphi}{\sqrt{A \lg^3 N}}\right) + O\left(\frac{N}{\sqrt{A \lg^3 N}}\right) \cdot \lg\left(\frac{N}{A \lg^3 N}\right) \\ &= o(N\varphi) + o(N) \end{aligned} \tag{8}$$

For the sub-region search structures we will consider their space cumulatively. We add $O\left(N/\sqrt{A}\right)$ edges each requiring 2φ bits for the endpoints and $\lg\left(\lg^3 N\right)$ bit references to the $\lg^3 N$ sub-regions within the region. The space required for this structure is then:

$$O\left(\frac{N}{\sqrt{A}}\right) \cdot \left(2\varphi + \lg\left(\lg^3 N\right)\right) = O\left(\frac{N\varphi}{\sqrt{A}}\right) + O\left(\frac{N}{\sqrt{A}} \cdot \lg\lg\left(N\right)\right) \tag{9}$$

The first term this equation is clearly $o(N\varphi)$. Recall that $A = (B \lg N)/(c + \varphi)$ and that $B = \Omega(\lg N)$. Thus for the second term of Eq.(9) we have:

$$\begin{aligned} O\left(\frac{N}{\sqrt{A}} \cdot 3 \lg\lg N\right) &= O\left(\frac{N}{\sqrt{\frac{B \lg N}{c+\varphi}}} \cdot \lg\lg\left(N\right)\right) \\ &= o(N\varphi) \end{aligned} \tag{10}$$

The total space we use is thus bounded by $o(N\varphi)$ bits. The I/O complexity stems from the fact that we perform two point location queries on data structures that answer the query in $O(\log_B N)$ I/Os where each structure is of size less than N. □

Theorem 6. *Given a terrain T, where each point coordinate may be stored in φ bits, there is a data structure that represents T in $N\varphi + O(N) + o(N\varphi)$ bits, that permits traversal of a path crossing K faces in T with $O\left(\frac{K}{\lg B}\right)$ I/Os, and which supports point location queries with $O(\log_B N)$ I/Os.*

Proof. The terrain data structure requires $N\varphi + O(N) + o(N\varphi)$ bits by Theorem 2. By Lemma 19 adding point location requires only $o(N\varphi)$ bits so the overall space bound is the same and queries can be answered in $O(\log_B N)$ I/Os. □

References

[1] Pankaj K. Agarwal, Lars Arge, T. M. Murali, Kasturi R. Varadarajan, and Jeffrey Scott Vitter. I/O-efficient algorithms for contour-line extraction and planar graph blocking (extended abstract). In *SODA*, pages 117–126, 1998.

[2] Alok Aggarwal and S. Vitter Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[3] Lars Arge, Andrew Danner, and Sha-Mayn Teh. I/o-efficient point location using persistent b-trees. In Richard E. Ladner, editor, *ALENEX*, pages 82–92. SIAM, 2003.

[4] Prosenjit Bose, Eric Y. Chen, Meng He, Anil Maheshwari, and Pat Morin. Succinct geometric indexes supporting point location queries. In *SODA*, pages 635–644, 2009.

[5] Prosenjit Bose and Pat Morin. An improved algorithm for subdivision traversal without extra storage. In D. T. Lee and Shang-Hua Teng, editors, *ISAAC*, volume 1969 of *Lecture Notes in Computer Science*, pages 444–455. Springer, 2000.

[6] Yi-Ting Chiang, Ching-Chi Lin, and Hsueh-I Lu. Orderly spanning trees with applications. *SIAM J. Comput.*, 34(4):924–945, 2005.

[7] Yu-Feng Chien, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In *DCC*, pages 252–261, 2008.

[8] Richie Chih-Nan Chuang, Ashim Garg, Xin He, Ming-Yang Kao, and Hsueh-I Lu. Compact encodings of planar graphs via canonical orderings and multiple parentheses. *CoRR*, cs.DS/0102005, 2001.

[9] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *SODA*, pages 383–391, 1996.

[10] Mark de Berg, Prosenjit Bose, Katrin Dobrindt, Marc J. van Kreveld, Mark H. Overmars, Marko de Groot, Thomas Roos, Jack Snoeyink, and Sidi Yu. The complexity of rivers in triangulated terrains. In *CCCG*, pages 325–330, 1996.

[11] Mark de Berg, Marc J. van Kreveld, Mark H. Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2 edition, 2000.

[12] Mark de Berg, Marc J. van Kreveld, René van Oostrum, and Mark H. Overmars. Simple traversal of a subdivision without extra storage. *International Journal of Geographical Information Science*, 11(4):359–373, 1997.

[13] Craig Dillabaugh, Meng He, and Anil Maheshwari. Succinct and I/O efficient data structures for traversal in trees. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *ISAAC*, volume 5369 of *Lecture Notes in Computer Science*, pages 112–123. Springer, 2008.

[14] Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.*, 16(6):1004–1022, 1987.

[15] Christoper M Gold and Uri M Maydell. Triangulation and spatial ordering in computer cartography. In *Proceedings Canadian Cartographic Association third annual meeting*, pages 69–81, 1978.

[16] Guy Jacobson. Space-efficient static trees and graphs. *FOCS*, 42:549–554, 1989.

[17] David G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[18] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *FOCS*, pages 118–126, 1997.

[19] Mark H. Nodine, Michael T. Goodrich, and Jeffrey Scott Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.

[20] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA*, pages 233–242, 2002.

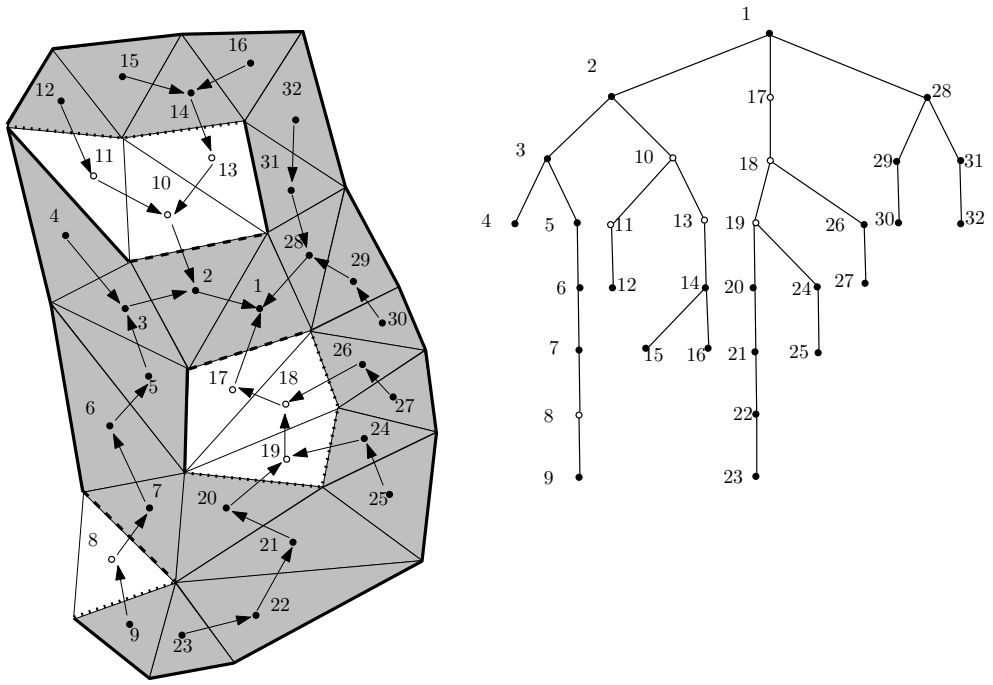


Figure 11: On the left a triangulation T and connected component T' (shaded) with a non-convex boundary and holes is shown. The original tree G_T is shown on the right with vertices labelled by preorder number for the entire triangulation T. Hallow vertices correspond to vertices removed from G_T by holes and/or concavities in the boundary of T'. The parent-child relationships, and the vertex labels are also shown on T. In T boundary edges are thick lines, with entry edges being dashed, and exit edges dotted. The original call to $DepthFirstTraversal$ first encounters the entry edge (8,7), from which $ScanBoundary$ will visit exit edges (9,8), (12,11) and (14,13), in that order, before terminating back at $(8, 7)$. This initial call to $ScanBoundary$ results in subtrees rooted at vertices 9, 12, and 14 being reported. $ScanBoundary$ is not invoked from entry edge (10,2) as this edge is added to $\mathcal{V}$ during the invocation of $ScanBoundary$ from $(8, 7)$. $ScanBoundary$ is however called from $(17, 1)$, which results in entry edges $(26, 18)$, $(24, 19)$ and $(20, 19)$ being visited and the subtrees rooted at vertices 26, 24 and 20 being reported.

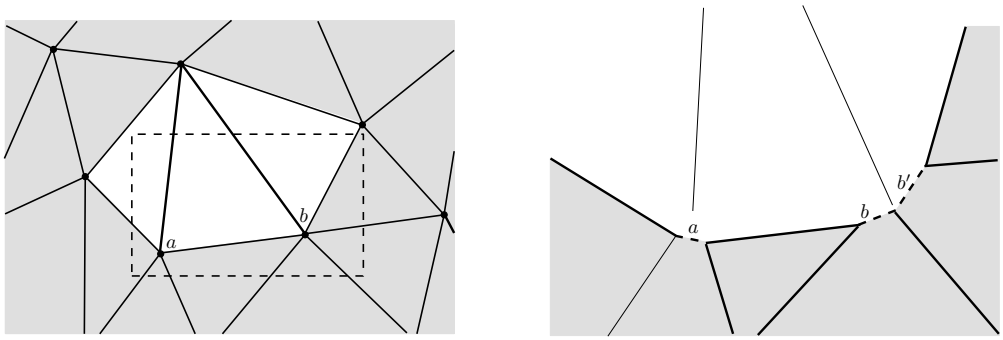


Figure 12: The figure on the left shows a portion of a component (grey) with a hole (white). The dashed box indicates the detailed area shown in the figure on the right. The right hand figure shows conceptually how additional edges are added to the hole boundary, corresponding to triangles in the component. At the point marked a a single pseudo-edge is added since there is only one non-edge adajacent triangle adjacent to this point. The point b has two non-edge adjacent triangles so two pseudo-edges b and b' are added to the hole boundary.