I/O and Space-Efficient Path Traversal in Planar Graphs

Craig Dillabaugh[1], Meng He[2], Anil Maheshwari[1], and Norbert Zeh[3]

[1] School of Computer Science, Carleton University, Canada
[2] Cheriton School of Computer Science, University of Waterloo, Canada
[3] Faculty of Computer Science, Dalhousie University, Canada

Abstract. We present a technique for representing bounded-degree planar graphs succinctly while permitting I/O-efficient path traversal. To represent a graph G on N vertices, each with an associated key of $q = O(\lg N)$ bits[4], we use $Nq + O(N) + o(Nq)$ bits. Using this representation, a path of length K can be traversed with $O(K/\lg B)$ I/Os, where B is the disk block size. Our structure may be adapted to represent, with similar space bounds, a terrain modeled as a triangular-irregular network to support traversal of a path that visits K triangles using $O(K/\lg B)$ I/Os. This structure can be used to answer a number of useful queries efficiently, such as reporting terrain profiles, trickle paths and connected components.

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 amount of data required to solve a given problem exceeds internal memory. Data structures and algorithms are designed to solve the problem, while minimizing 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. In addition to our own research on traversal in trees [1], the only other research that merges these techniques is on succinct EM data structures for text indexing [2, 3].

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 using a small number of I/O operations. We demonstrate practical applications of our structure by showing how it can be applied to answering queries on triangular irregular network (TIN) models.

1.1 Background

In the External Memory (EM) model [4], the number of elements in the problem instance is denoted by N. Memory is divided into a two-level hierarchy, external

[4] We use $\lg N$ to denote $\log_2 N$.

and internal memory. The external memory is assumed to have effectively infinite capacity, but accessing data elements in external memory is slow. The internal memory permits efficient operations, but its capacity is limited to $M < N$ elements. Data are transferred between internal and external memory in blocks of size B, where $1 < B < M/2$. In this paper, we assume that $B = \Omega(\lg N)$, which is true for all realistic values of N and B in an EM setting. 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.* [5] first considered 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. They identified the optimal bounds for several classes of graphs. An I/O-efficient algorithm that blocks a bounded-degree planar graph so that any path of length K can be traversed in $O(K/\lg B)$ I/Os was proposed by Agarwal *et al.* [6].

Succinct data structures for trees and graphs were originally proposed by Jacobson [7]; the goal is to represent data structures using space as near the information-theoretic lower bounds as possible, while still permitting efficient navigation. Numerous efforts have been made to improve this for graphs [8, 9]. Recently, Chiang *et al.* [10] obtained a succinct data structure for planar graphs that uses $2m + 2n + o(n)$ bits, where m and n are the number of edges and vertices, respectively, in the graph.

2 Preliminaries

A key data structure used in our graph representation is a bit vector $C[1..N]$ that supports `rank` and `select` operations efficiently. The operations $\texttt{rank}_1(C, i)$ and $\texttt{rank}_0(C, i)$ return the number of 1s and 0s in $C[1..i]$, respectively. The operations $\texttt{select}_1(C, r)$ and $\texttt{select}_0(C, r)$ return the position of the r^{th} occurrence of 1 and 0 in C, respectively. The problem of representing a bit vector succinctly to support `rank` and `select` operations in constant time under the word RAM model with word size $\Theta(\lg N)$ bits has been considered in [7, 2, 11], and these results can be directly applied to the external memory model:

Lemma 1. *A bit vector C of length N containing R 1s can be represented using (a) $N + o(N)$ bits [2] or (b) $\lg \binom{N}{R} + O(N \lg\lg N/\lg N)$ bits [11] to support the access to the bits of C, as well as* `rank` *and* `select` *operations on C, in $O(1)$ time (or $O(1)$ I/Os in external memory).*

Frederickson [12] developed a technique for decomposing planar graphs. A planar graph is divided 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. Lemma 2 summarizes Frederickson's result.

Lemma 2 ([12]). *A planar graph with N vertices can be subdivided into $\Theta(N/r)$ regions of no more than r vertices and with a total of $O(N/\sqrt{r})$ 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 $q = O(\lg N)$. We perform a two-level partitioning of G inspired by the approach in [13]. 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 vertex if it has neighbouring vertices in G that 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 vertex.

Consider some vertex $v \in G$. Based on [6], 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. The α-neighbourhood of v is the subgraph of G induced by these vertices. 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.

In our representation of G, we store each sub-region and the α-neighbourhood of each boundary vertex. When constructing the α-neighbourhood of a sub-region boundary vertex that is not a region boundary vertex, we add an additional constraint that its α-neighbourhood 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 rather are a collection of their sub-region components. Each component is stored using a succinct representation that allows efficient traversal of the component. To enable traversal of G, each vertex is assigned a unique *graph label*.

3.1 Graph Labeling

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.* [13] but uses the technique of Frederickson [12] for graph decomposition.

Each vertex v of G is assigned a unique *graph label*, in addition to possibly multiple (in the case of boundary vertices) *region labels* and *sub-region labels*. G is partitioned into a set of regions. We denote the i^{th} region by R_i. Each region R_i is subdivided into sub-regions. We denote by q_i the number of sub-regions in R_i, and denote the j^{th} sub-region of 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 of $R_{i,j}$. We say that the instance of v appearing in $R_{i,j}$ *defines* v in R_i if there is no sub-region $R_{i,h}$ in R_i, 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 vertex $v \in R_i$, v is a

defining vertex if there is no region R_h containing v, where $h < i$. 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 labels of the defining vertex.

The encoding of a sub-region $R_{i,j}$ induces a permutation of the vertex set within the sub-region. We let the position of each vertex within this permutation serve as its sub-region label. Now we discuss the assignment of region labels to the vertices within a 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 sub-region boundary vertices in R_i. We visit each sub-region $R_{i,j}$ for $j = 1, 2, \ldots, q_i$, 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 labels of their defining vertices.

The assignment of graph labels mirrors that of region labels. Let n^b be the total number of region boundary vertices over all regions in G. We 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 the 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 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:

1. A compact encoding of the graph structure of $R_{i,j}$. This 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 $|R_{i,j}|$ that stores the q-bit key for each vertex in $R_{i,j}$.

We store two arrays $\mathcal{L}_S$ and $\mathcal{L}_R$, which record for each sub-region and region, 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. We store sub-regions to disk in the following fashion: 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 in this order. Prior to writing

a sub-region we write its *sub-region offset* value, which is its size in vertices. 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 a sub-region happens to end at a block boundary, we write 0 to the block offset of the next block. Since a sub-region is no larger than a single block, a sub-region will never span portions of more than two blocks. We continue this process 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 store the α-neighbourhood for each region and sub-region boundary vertex v by encoding the subgraph G_v that comprises v's α-neighbourhood using:

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 $|G_v|$ that marks each vertex in G_v as internal or terminal.
3. A variable that records the position of v within the permutation of the vertices of G_v.
4. An array of length $|G_v|$ that stores the key associated with each vertex.
5. An array, $\mathcal{L}_\alpha$, of length $|G_v|$ that stores for each vertex w:
   (a) its graph label if v is a region boundary vertex. Otherwise,
   (b) a region offset of $\lg(A \lg^3 N)$ bits if w is interior to v's region. This offset is calculated as the difference between the graph labels of w and the first interior vertex in v's region (stored in $\mathcal{L}_R$). If w is a boundary vertex of v's region, we store w's region label in this region using $\lg(A \lg^3 N)$ bits.

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 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 $\mathcal{D}[i] = 1$ iff the vertex with graph label i is a sub-region boundary vertex that is interior to its region.

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

Proof (Sketch). Let c denote the number of bits per vertex required to store the sub-graph and boundary bit-vector (this applies to both sub-region and α-neighbourhood components). The exact value of c depends on the chosen succinct representation for the graph used to store components. We have assumed blocks of size $B \lg N$ bits, and for the sake of simplicity, we assume that $c+q$ divides $B \lg N$. Thus $(c+q)A = B \lg N$, and $A = \frac{B \lg N}{c+q}$.

When splitting G into regions and sub-regions, we let $r = A \lg^3 N$ for regions, and $r = A$ for sub-regions. By Lemma 2, this results in $\Theta\left(\frac{N}{A \lg^3 N}\right)$ regions with $O\left(\frac{N}{\sqrt{A \lg^3 N}}\right)$ region boundary vertices, and $\Theta\left(\frac{N}{A}\right)$ sub-regions with $O\left(\frac{N}{\sqrt{A}}\right)$ sub-region boundary vertices. Regions are not explicitly stored, so consider the space required to store the sub-regions. To store a single copy of each vertex, we require $N(c+q)$ bits, which accounts for all internal vertices plus the defining copies of all boundary vertices. Additionally, we need to store the $O\left(N/\sqrt{A}\right)$ duplicate boundary vertices, which requires $O\left(\frac{N}{\sqrt{A}}\right)(c+q) = o(Nq)$ bits. Storing the sub-regions also requires space for 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 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 $o(N)$ bits. Likewise, the $O(N/A)$ sub-region offsets use $o(N)$ bits. Thus, the total cost for storing the sub-regions is $N(c+q) + o(Nq)$ bits.

We let $\alpha = A^{\frac{1}{3}}$ be the size of the α-neighbourhoods for both region and sub-region boundary vertices (in fact $\alpha = A^{\frac{1}{2}-\epsilon}$, for $\epsilon > 0$, suffices for our analysis). Cumulatively, the number of bits required to store the α-neighbourhoods of all region boundary vertices is $\Theta(\frac{N}{\sqrt{A \lg^3 N}}) \cdot A^{\frac{1}{3}} \cdot (\lg N + q + c) = \Theta(\frac{N}{A^{\frac{1}{6}} \lg^{\frac{3}{2}} N}) \cdot \Theta(\lg N) = o(N)$. 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 space requirement for all α-neighbourhoods of all sub-region boundary vertices is thus $\Theta(\frac{N}{\sqrt{A}}) \cdot A^{\frac{1}{3}} \cdot (\lg(A \lg^3 N) + q + c) = o(Nq)$ bits. It is easy to show that the remaining auxiliary data structures, $\mathcal{D}$, $\mathcal{L}_R$ and $\mathcal{L}_S$, occupy $o(N)$ bits. □

3.3 Navigation

The traversal algorithm operates by loading either a sub-region or the α-neighbourhood 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 a function `step` available, 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. First we 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 (Lemma 5). Finally, we demonstrate that visiting a constant number of components guarantees a progress of $O(\lg A)$ steps along the path (Lemma 6). We omit the proofs in the rest of this paper due to space constraints.

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 into main memory.*

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 [13], and by Lemma 3.4 in their paper, conversion between these labels can be performed in $O(1)$ time. We can extend this to external memory, and prove the following lemma:

Lemma 5. *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.*

Lemma 6. *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.*

Combining Lemmas 3 and 6, we have (note that $A = \Omega(B)$):

Theorem 1. *A bounded-degree planar graph G on N vertices, where each vertex stores a key of q bits, can be represented using $Nq+O(N)+o(Nq)$ bits to support the traversal of a path of length K with $O\left(\frac{K}{\lg B}\right)$ I/O operations.*

The only comparable work to ours is that of Agarwal *et al.* [6]. In addition to storing keys, they use $O(N \lg N)$ bits to store the graph structure. In contrast, we use only $O(N)$ bits to store the graph structure, and we manage to show that the path traversal and other operations can still be performed I/O-efficiently. For very large data sets, this is an enormous saving in space.

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. 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 so 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 the face of T that corresponds to v. The new graph G' remains planar but is no longer of bounded degree. However, all the 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 to the vertices corresponding to points as *point vertices*. In a similar fashion, we refer to the edges of G as *dual edges*, and to those edges added to connect the point and triangle vertices as *point edges*. We have the following lemma that bounds the size of G':

Lemma 7. *Given the dual graph G of triangulation T with N vertices, the augmented graph G' has at most $2N + 2$ vertices.*

For the purpose of quantifying the bit cost of our data structures, we denote by $\varphi = O(\lg N)$ the number of bits required to encode the coordinates of a point. We encode G' using a succinct planar graph data structure. The encoding involves a permutation of the vertices of G'. Let the *augmented graph label*, $\ell(v)$, of the vertex v, be the position of v in this permutation. We store the point set P in an array $\mathcal{P}$ ordered by the augmented graph labels of the points. Finally, we 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, we have:

Lemma 8. *The data structures described above can represent a terrain T composed of N triangles with φ-bit point coordinates, where $\varphi = O(\lg N)$, using $N\varphi + O(N) + o(N\varphi)$ bits, so that, given the label of a triangle, the adjacent triangles and points can be reported in $O(1)$ time.*

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 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.

Lemma 9. *The space requirement, in bits, to store a component (α-neighbourhood or sub-region) representing a terrain is within a constant factor of the space required to store the terrain's dual graph.*

We have the following theorem due to Theorem 1 and Lemma 9:

Theorem 2. *A terrain T modeled as a TIN on N nodes, where the coordinates of each point can be stored in φ bits, can be represented using $N\varphi+O(N)+o(N\varphi)$ bits to support the 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 can easily extend our approach to represent a terrain using $N(\varphi+q)+O(N)+o(N(\varphi+q))$ bits to provide the same support for path traversals.

4.2 Applications on TIN Models

We now present a number of applications on TIN models represented using our data structures. In this section, we assume that we are given a starting triangle, $t \in T$, as a query parameter. We remove this assumption in Section 4.3.

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 descent, from a point p is the path on T that begins at p and follows the direction of steepest descent until it reaches a local minimum or the boundary of T [14]. In analyzing these algorithms, we measure the complexity of a path based on the number, K, of triangles it intersects. When a path intersects a vertex, we consider all triangles adjacent to that vertex to have been intersected. Given this definition, we have:

Lemma 10. *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.

Connected Component Queries: In a connected component query, 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' is in T' iff $\mathcal{P}(t) = \mathcal{P}(t')$ and there exists a path in G from t to t' consisting only of triangles that share this property.

Theorem 3. *A triangulation T with a q-bit key per triangle can be stored using $Nq + O(N) + o(Nq)$ bits such that a connected component, T', may be reported:*

(a) Using $O\left(\frac{|T'|}{\lg B}\right)$ I/Os if T' is convex.

(b) Using $O\left(\frac{|T'|}{\lg B} + h\log_B h\right)$ I/Os, and $O(h\cdot(q+\lg h))$ bits of temporary storage, if T' is non-convex and/or may contain holes. The value h denotes the number of boundary edges around all the holes and the perimeter of T'.

(c) Using $O\left(\frac{|T'|}{\lg B} + h'\log_B h'\right)$ I/Os and no additional storage if T' is non-convex and/or may contain holes. The value h' is the total number of triangles touching all the holes and the perimeter of T'.

4.3 Terrain Representation with Point Location

For the various applications that we have described, we assumed that a starting triangle in T is given as an input parameter. This is problematic because in real applications, we will typically need to locate the triangle from which we will begin reporting the result. To address this problem, we combine the succinct data structure of Bose *et al.* [13] that supports point location in internal memory with our data structures. This allows us to use $o(N\varphi)$ bits of additional storage to perform planar point location on a terrain, T, in $O(\log_B N)$ I/Os. Asymptotically this does not change the space requirement for T.

Theorem 4. *A terrain T modeled as a TIN on N nodes, where each point coordinate may be stored in φ bits, can be represented T using $N\varphi+O(N)+o(N\varphi)$ bits to support the traversal of a path crossing K faces in T with $O\left(\frac{K}{\lg B}\right)$ I/Os, and to support point location queries incurring $O(\log_B N)$ I/Os.*

References

1. Dillabaugh, C., He, M., Maheshwari, A.: Succinct and I/O efficient data structures for traversal in trees. In Hong, S.H., Nagamochi, H., Fukunaga, T., eds.: ISAAC. Volume 5369 of Lecture Notes in Computer Science., Springer (2008) 112–123
2. Clark, D.R., Munro, J.I.: Efficient suffix trees on secondary storage. In: SODA. (1996) 383–391
3. Chien, Y.F., Hon, W.K., Shah, R., Vitter, J.S.: Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In: DCC. (2008) 252–261
4. Aggarwal, A., Jeffrey, S.V.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9) (1988) 1116–1127
5. Nodine, M.H., Goodrich, M.T., Vitter, J.S.: Blocking for external graph searching. Algorithmica **16**(2) (1996) 181–214
6. Agarwal, P.K., Arge, L., Murali, T.M., Varadarajan, K.R., Vitter, J.S.: I/O-efficient algorithms for contour-line extraction and planar graph blocking (extended abstract). In: SODA. (1998) 117–126
7. Jacobson, G.: Space-efficient static trees and graphs. FOCS **42** (1989) 549–554
8. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses, static trees and planar graphs. In: FOCS. (1997) 118–126
9. Chuang, R.C.N., Garg, A., He, X., Kao, M.Y., Lu, H.I.: Compact encodings of planar graphs via canonical orderings and multiple parentheses. CoRR **cs.DS/0102005** (2001)
10. Chiang, Y.T., Lin, C.C., Lu, H.I.: Orderly spanning trees with applications. SIAM J. Comput. **34**(4) (2005) 924–945
11. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: SODA. (2002) 233–242
12. Frederickson, G.N.: Fast algorithms for shortest paths in planar graphs, with applications. SIAM J. Comput. **16**(6) (1987) 1004–1022
13. Bose, P., Chen, E.Y., He, M., Maheshwari, A., Morin, P.: Succinct geometric indexes supporting point location queries. In: SODA. (2009) 635–644
14. de Berg, M., Bose, P., Dobrindt, K., van Kreveld, M.J., Overmars, M.H., de Groot, M., Roos, T., Snoeyink, J., Yu, S.: The complexity of rivers in triangulated terrains. In: CCCG. (1996) 325–330