Persistent Data Structures for Fast Point Location

Michał Wichulski and Jacek Rokicki

Warsaw University of Technology,
Institute of Aeronautics and Applied Mechanics,
Nowowiejska 24, 00-665 Warsaw, Poland
{wichulski, jack}@meil.pw.edu.pl

Abstract. The paper presents practical implementation of an algorithm for fast point location an triangular 2D meshes. The algorithm cost is proportional to $\log_2 N$ while storage is a liear function of the number of mesh vertices N. The algorithm bases on using persistent data structures was used which store only changes between consecutive mesh levels. The performance of the presented approach was positively verified for a sequence of meshes starting from 1239 up to 1846197 cells.

1 Introduction

In computational physics partial differential equations are solved numerically on discrete meshes covering the region of interest. These meshes consist of a large numbers of disjoint cells filling completely the domain Ω:

$$\bar{\Omega} = \bigcup_{i=1}^{N_\Omega} \bar{\Omega}_i,\ \Omega_i \cap \Omega_j \neq \emptyset,\ i \neq j,\ \Omega_i = \operatorname{int} \Omega_i.$$

Here we consider simplest triangular cells and the 2D domain expecting that the approach can be extended to tetrahedral cells in 3D. Edges of these cells constitute a collection:

$$E = \{e_1, e_2, ..., e_{N_E}\}.$$

We further assume that no hanging nodes exist in the mesh. Therefore each edge belongs either to two cells (internal edge) or to a single cell (boundary edge). Similarly the collection of nodes is given as:

$$P = \{p_1, p_2, ..., p_N\}.$$

Consider now an arbitrary (random) point q. The localisation problem consist in:

1. deciding whether $q \in \bar{\Omega}$ (or not),
2. if $q \in \bar{\Omega}$ finding k $(1 \leq k \leq N_\Omega)$ such that $q \in \bar{\Omega}_k$.

If by chance q belongs to an edge $e \in E$ or coincides with $p \in P$ the problem may not have a unique solution. At the moment however we are satisfied with finding an arbitrary k such that $q \in \bar{\Omega}_k$.

This problems appears in many algorithms of computational geometry. Two examples will be given:

– in the Delaunay grid generation algorithm the new point inserted into a grid has to be localised before it is accepted,
– in the Chimera overlapping mesh algorithm [4][5][6] boundary points of one mesh have to be localised on the other mesh prior to all computations (here the efficiency of localisation is especially crucial if meshes move one with respect to another).

The localisation problem can be easily solved by inspecting, all cells Ω_k, checking each time the relation $q \in \bar{\Omega}_k$. Such algorithm has an average cost $\sim N_\Omega$ which usually is not acceptable if N_Ω and the number of points to be localised are large numbers (e.g., of order $10^5 - 10^6$).

It is also relatively straightforward to present algorithm based on quad-tree (octree) approach allowing to find k_* such that

$$\|q - p_{k_*}\| \leq \|q - p_i\| \quad i = 1, 2, ..., N$$

with a cost proportional to $\log_2 N$. Such algorithm requires however the second step in which neighbourhood of p_{k_*} is searched to find the Ω_k in question (the number of cells to be traversed can be significant).

The purpose of the present paper is (basing on ideas of Preparata and Tamassia [2]) to present practical one-step algorithm of solving the localisation problem with a cost proportional to $\log_2^\kappa N$ (κ - being a small number).

Again it will be shown further, that algorithm can be straightforward if available memory is proportional to N^2. However again this is not a practical requirement and the algorithm we seek has to have:

- the cost proportional to $\log_2^\kappa N$ (κ - being a small number),
- the storage proportional to N.

2 Preliminarities

We assume that in the 2D space $p_j = (x_j, y_j)$. Suppose now that vertices (nodes) are ordered in such a way that

$$y_1 < y_2 < ... < y_N.$$

Suppose now that the graph consisting of all edges (Fig. 1) is intersected by a line $\lambda(y) = \{(x, y) : \ y = const\}$. Firstly one should notice that the result of intersection forms a totally ordered set of intervals $R(y)$. A graph $G(y)$ can be associated with $R(y)$ with vertices which are the ordered intersection points of line $\lambda(y)$. The edges of this graph are corresponding intervals of $R(y)$.

Suppose now that an arbitrary point $q = (x_*, y_*)$ is given. The point location problem is now reduced to one-dimensional search within the $R(y_*)$ set (with $O(\log N)$ query cost). If the horizontal line is shifted up or down, the graph $G(y')$ has the same topology as $G(y_*)$ as long as the line $\lambda(y')$ does not cross the vertex.

Lemma 1. *For all y', y'' such that $y_i < y'$, $y'' < y_{i+1}$, graphs $G(y')$ and $G(y'')$ are isomorphic.*

To achieve $O(\log N)$ query cost a usual binary-tree data structure $S(y)$ has to be built (Fig. 2 and Fig. 3). Other data structures are possible but it has to be noticed that in any case $S(y)$ depends only on the ordering of the intervals and therefore remains constant on each interval $\langle y_i, y_{i+1}\rangle$. The particular interval from the ordered set $R(y)$ forms a *key* used to construct and access $S(y)$. The key depends on the geometric boundaries of the interval, but as long as the elements of $G(y)$ do not change (the intervals are neither inserted nor deleted) the data structure itself has the same topology. It leads to the following lemma [2].

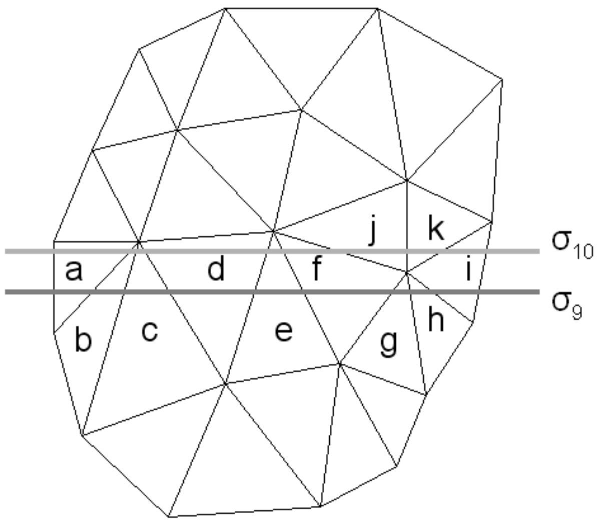

Fig. 1. Intersections corresponding to the levels in the data structure

Lemma 2. *The same data structure is sufficient for searching in subdivisions represented by graphs, which are isomorphic.*

The *keys* of the data structure $S(y)$ are parameterised by the height of the horizontal line $\lambda(y)$, because that line intersects edges of the mesh and points of that intersection give geometric coordinates of the intervals. If the line $\lambda(y)$ is such that $y_i < y < y_{i+1}$, than it fulfils conditions of Lemma 1, and additionally the same edges trim intervals. Therefore it follows that definition of the keys (that is a way of parameterising the intervals) remains unchanged. The interval between two consecutive vertices of the mesh is called now a *level*. Taking heed of Lemma 2 we can assert that data structure needs no changes at a given level. If there are N vertices in the mesh, $N - 1$ levels exists.

In particular for $y \in \sigma_i = \langle y_i, y_{i+1}\rangle$ the ordered set $R(y)$, the graph $G(y)$ and the data structures $S(y)$ can be now parametrised by a level number and are denoted respectively as G_i, R_i and S_i.

One can observe in particular that differences between G_i and G_{i+1} (as well as S_i and S_{i+1}) are quite limited The important remark is that the changes reduce to few basic possibilities (see Fig. 1):

a) deleting the cells for which p_i is the top vertex
b) inserting the cells for which p_i is the bottom vertex
c) modifying the keys in the cells for which the vertex is the middle vertex

This results in the following lemma:

Lemma 3. *The number of elementary changes (see above) necessary to transform S_i into S_{i+1} is equal to m_i (where m_i is the number of cells containing vertex p_i).*

Consider an example. Two horizontal lines on Fig. 1 (dark and light gray) correspond to the levels σ_i and σ_{i+1}. If the line λ leaves the lower level σ_i and enters σ_{i+1}, the following changes are necessary:

- cells g and h are deleted,
- cells j and k are inserted,
- the keys in f and i are modified.

This modification consists in replacing the formulas describing the edges (i.e. edge j replaces g and edge k replaces h).

Suppose that we already have the data structure S_i. The new structure S_{i+1} can be obtained from S_i by applying the changes, i.e. by adding the difference between these levels. In fact only the differences have to be stored. This observation allows to use so called persistent data structures.

3 The Algorithm

It is possible now to present point location algorithm with the query cost proportional to $O(\log N)$. The data structure S supporting this algorithm contains N substructures S_i. Each substructure S_i is (for example) a binary tree consisting of ordered non-intersecting segments (as described in the previous chapter).

The point location algorithm consists of two queries, in the first one the vertical position y_* is located on the appropriate level, in the second one the horisontal position x_* is located within appropriate segment. The query cost is therefore at most $2 \log N$. The number of segments in each substructure is at most $N + 1$. Therefore the size of the whole data structure S is proportional to N^2. For large meshes ($N > 10^6$) this is fully unacceptable.

The reader may notice, that assumption $M \sim N$ is very pessimistic, since $M \sim \sqrt{N}$ on more regular meshes. Yet, even $N\sqrt{N} = N^{3/2}$ is too large for practical purposes. In order to overcame this problem we can still use the fact, that the consecutive substructures, say S_i and S_{i+1} differ only by few elementary operations. Thus we will attempt to add persistence trying to store changes to the structure, rather than keeping structures themselves. One must observe, however, that persistence is difficult to implement if S_i is an ordinary binary tree. Every time the node is deleted or inserted, the tree needs re-balancing in order to keep the optimal height. This means that the change between S_i and S_{i+1} may concern almost all vertices. Therefore another data structure is necessary to alleviate this problem.

The possible choice is a red-black binary search tree (see Fig. 2 and Fig. 3) As in every binary search tree, the node contains three fields: the key, the left pointer and the right pointer. The search through the tree, from node to node, starts at the topmost node called a *root*. In every node the comparison (in sense of the total order) between the wanted and the current key is done and, depending on the result, a proper branch is chosen. Without going into details, one should note that in the red-black tree-node another field is added, called *colour*, which is necessary to preserve balancing. For all details of algorithms of inserting, finding and deleting an item from the tree, see [1]. It is enough to say, that as a result of re-balancing of the tree some changes may occur on the path from the root to an inserted node. The re-balancing itself is performed using rotations of the tree branches.

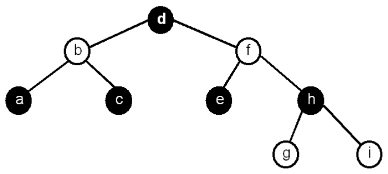

Fig. 2. The tree corresponding to the level σ_9

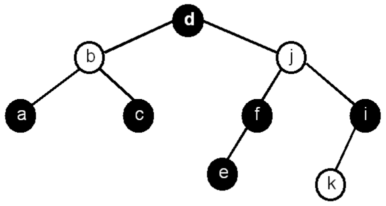

Fig. 3. The tree corresponding to the level σ_{10}

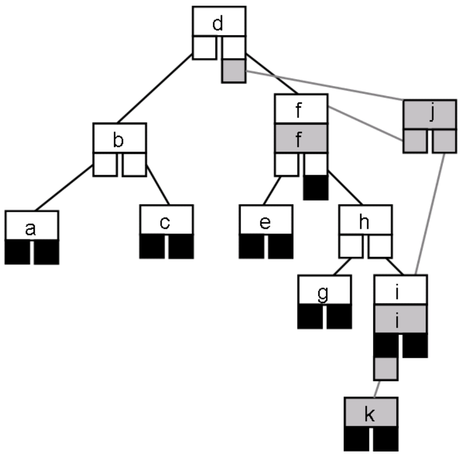

Fig. 4. The persistent binary search tree for two σ_9 and σ_{10} levels in the mesh

To add persistence to the data structure we have chosen *the fat node method* [3]. It is based on recording all changes made to a node in the node itself. As a consequence, each node contains collection of pointers, each corresponding to a different level.

Let η be the number of changes during a single operation on the tree, and let ζ be the number of nodes in which these changes occur. The rotation of a single branch, consists of three changes in the pointer fields. To insert a node no more than two rotations are necessary while three rotations are sufficient to delete a node [3]. In such a way (in a most pessimistic case) there are five changes of pointer fields for inserting a node, and seven changes for deleting the node with four and six nodes involved respectively (all from one sub-tree). The important thing is, that η and ζ are both $O(1)$, and not $O(N)$. Similarly, a scope of changes within the tree is limited and relates to the neighbouring nodes in a sub-tree. Thus, since the red-black binary search tree is balanced, the changes concern nodes located in the immediate neighbourhood of either deleted or inserted node.

As it was mentioned earlier the full data structure S can be seen as a sequence of red-black binary search trees S_i. Each node representing a cell (or strictly speaking the slice of the cell at the present level). This node has a left and right child each corresponding to neighbouring slice.

Consider now an equivalent structure consisting of all cells. Each cell has two own collections of left and right children (each child corresponding to a different level σ_i) - see Fig. 4. These collections must contain only these levels, at which the cell changes location in the red-black binary search tree. The numbers s_i^L and s_i^R of entities in these collections (of left and right children respectively), as it was mentioned earlier, are expected to be $O(1)$. Additionally to speed up the search, the collections themselves can be ordered.

The total size of the full date structure is now estimated as $\beta \cdot N$ where ($\beta \sim s_* = \max_j s_j$), while point location query cost remains $\alpha \cdot \log N$ (α does not exceed exceed $1 + \log s_*$).

4 Numerical Experiment

In order to investigate properties of the presented algorithm, the following numerical experiment was performed. A sequence of meshes was generated in the pentagonal domain with N ranging from 1239 to 1846197 (see Fig. 9). Each experiment consisted in localising of one million random points. Every elementary step performed by the algorithm was counted to measure the *search cost*. The total cost defined as a largest cost value for all queries was identified for each mesh. This value is presented in the Fig. 5 indicating the perfect logarithmic behaviour. The coefficient α (see previous chapter) is additionally shown in Fig. 6.

Similarly, the total memory storage was measured in each experiments, by using the Linux *ps* command. The result is shown in Fig. 7. One can see that this value grows linearly with the number of cells. The estimated coefficient β

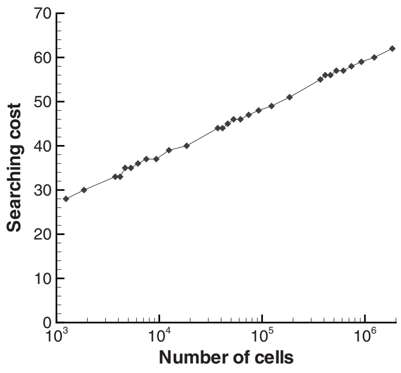

Fig. 5. The cost of the point location (measured by counting elementary operations)

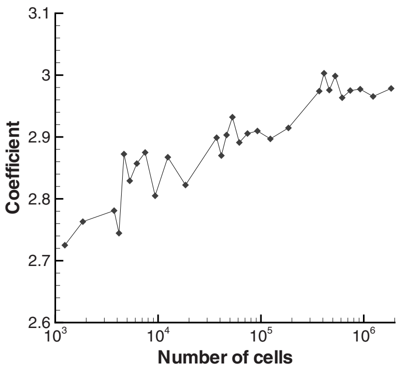

Fig. 6. The point location query cost coefficient α

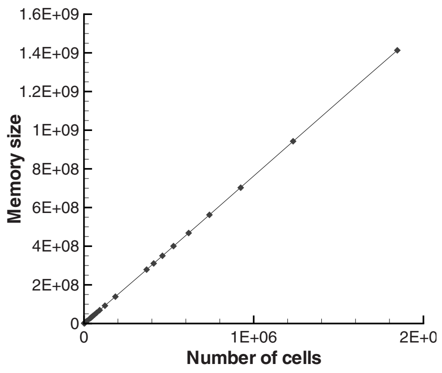

Fig. 7. The memory size of the data structure

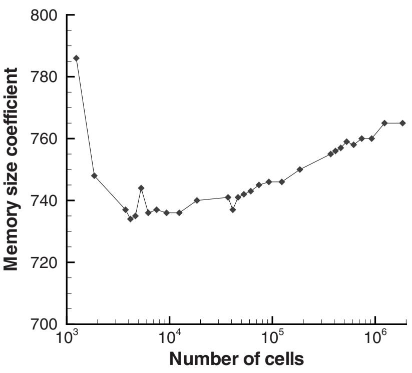

Fig. 8. The memory size coefficient β

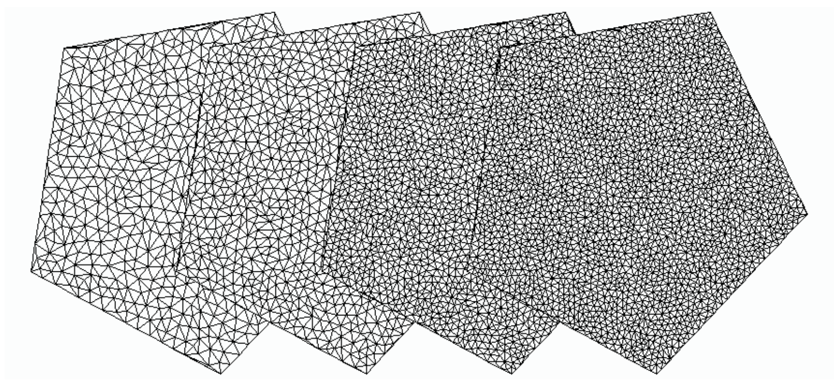

Fig. 9. Meshes used in numerical experiment

is shown in Fig. 8. It must be noted that this result represents the size of the memory used by the process, and not necessarily the size of the data structure (which we believe is smaller). It is suspected that possible discrepancy may be caused by the memory leakage typical when space is allocated and deallocated in repetitive manner.

5 Concluding Remarks

The paper presented a practical algorithm for point location problem with $\alpha \cdot \log N$ query cost and $\beta \cdot N$ memory storage. This fact was proven in numerical experiment in which both α and β were estimated. Further research is under way to extend this algorithm to 3D tetrahedral meshes.

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge (1990)
2. Preparata, F.P., Tamassia, R.: Efficient point location in a convex spatial cell-complex. SIAM J. Comput. 21(2), 267–280 (1992)
3. Discroll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. J. Comput. Syst. Sci. 38(1), 267–280 (1989)
4. Benek, J.A., Buning, P.G., Steger, J.L.: A 3-D Chimera Grid Embedding Technique. AIAA Paper 85, 1523 (1985)
5. Rokicki, J., Floryan, J.M.: Unstructured Domain Decomposition Method for the Navier-Stokes Equations. Computers & Fluids 28, 87–120 (1999)
6. Drikakis, D., Majewski, J., Rokicki, J., Żółtak, J.: Investigation of Blending-Function-Based Overlapping-Grid Technique for Compressible Flows. Computer Methods in Applied Mechanics and Engineering 190(39), 5173–5195 (2001)