JOURNAL OF THEORETICAL AND APPLIED MECHANICS
46, 2, pp. 315-324, Warsaw 2008

FAST POINT LOCATION ALGORITHM ON TRIANGULAR MESHES

Michał Wichulski
Jacek Rokicki

Warsaw University of Technology, Institute of Aeronautics and Applied Mechanics, Poland
e-mail: wichulski@meil.pw.edu.pl; jack@meil.pw.edu.pl

This paper is a study of application of persistent data structures to the planar and, in part, also spatial point location. In practice, a simplified method of building persistent red-black binary search tree is considered. It corresponds to the structure of a two-dimensional cell complex. Subsequent use of the structure for searching a certain point in space is shown. The computational mesh consists of triangular (in two dimensions) or tetrahedral (in three dimensions) cells. This fact allows significant simplifications to both implementation of the total order necessary to build the search tree as well as construction of the red-black binary search tree itself. The performance of the algorithm is verified for various meshes (consisting of up to 1846197 cells). Finally, certain further directions of the research are shown.

Key words: mesh generation, Chimera mathod, point location algorithms

1. Introduction

One of the special-purpose methods in the field of numerical analysis is the so called Chimera method. It is based on dividing the computational domain into a union of overlapping subdomains. A physical problem (usually expressed in the form of Partial Differential Equations) is solved locally on each sub-domain, while the global solution is obtained by iteratively adjusting the boundary conditions on each sub-domain (see Fig. 1). The value of solution on the current mesh is used as the boundary condition for overlapping meshes in the next step of the iterative process. Effectiveness of the algorithm of localisation is one of factors limiting efficiency of such a method (especially for meshes moving one with respect to another).

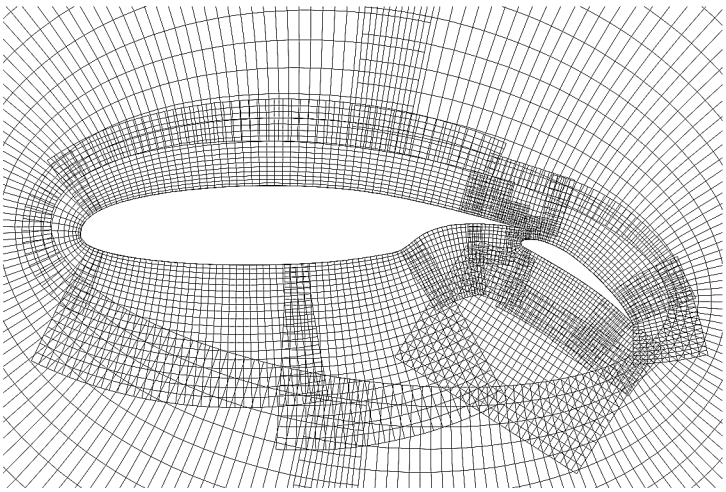

Fig. 1. An example of overlapping meshes in the Chimera method

The point location is a classic problem in computational geometry and its application scope runs out further than the Chimera method. The simplest formulation is "given an arbitrary point, find a mesh cell containing this point", this however does not describe the difficulty of finding the answer. The main goal is to minimise three factors:

- preprocessing time needed to build the data structure,
- memory used for this structure,
- query cost in which the cell containing the point is found.

The query cost of $O(N)$, characteristic for the naive method, is unacceptable. The Chimera method demands huge amounts of data to be searched and exchanged between iterations. This causes an avalanche of queries. Typical meshes of interest can contain millions of cells, and it is obvious that a brute force solution is not acceptable.

Therefore, an algorithm of cost lower than linear is sought, e.g., the average cost of one query should not exceed $O(\log^k N)$ (where $k \leqslant 2$).

The aim of the present paper is not, however, the design of a theoretically fast algorithm, but rather an efficient and robust implementation of an existing concept. Such a concept proposed by Preparata and Tamassia (1992) relied on *persistent data structures* (Discroll *et al.*, 1989) for two- and three-dimensional meshes. This type of approach will be described further.

It must be noticed that the policy of Preparata and Tamassia (1992) based on adding the *persistence* was an inspiration to find a generalisation in the treatment of the problem.

2. Preliminaries

We define a *cell* as a convex polygon, and a collection of nonoverlapping cells is called a *mesh*. In the mesh P, $V = \{v_1, v_2, \ldots, v_N\}$ denotes a set of vertices, y-coordinates of these vertices are denoted by $y_1, \ldots, y_N$, while $C = \{c_1, c_2, \ldots, c_n\}$ is a set of cells. It is assumed that the vertices are ordered in such way that $y_1 < y_2 < \cdots < y_N$. This means that two vertices never share the same y-coordinate. This may look very restrictive but in practice this condition can be easily fulfilled by an equivalent of a small rotation of the axis system (see de Berg *et al.*, 1997) for a practical algorithm dealing with degenerate cases).

First of all, one should notice that the intersection of the mesh P with the horizontal line $\lambda(y)$ at the height y forms a disjoint and a totally ordered set of intervals $R(y)$ (see Fig. 2. With the set $R(y)$, we associate a graph $G(y)$, whose vertices are the ordered intersection points of line $\lambda(y)$ with edges of the mesh P. The edges of this graph are intervals between these vertices. Such a definition guarantees a unique $1:1$ mapping from the set $R(y)$ to graph $G(y)$.

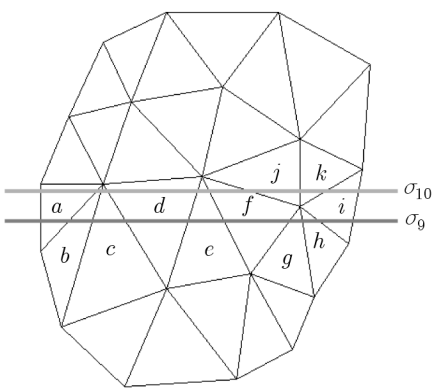

Fig. 2. Intersections corresponding to levels in the data structure

Suppose now that an arbitrary point $v_* = (x_*, y_*)$ 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 to y', the graph $G(y')$ has the same topology as $G(y_*)$ as long as the line $\lambda(y')$ does not cross the vertex.

Lemma 2.1. *For all $y', y'' \in \langle y_i, y_{i+1} \rangle$, the graphs $G(y')$ and $G(y'')$ are isomorphic $(i = 1, \ldots, N)$.*

On a particular height y, the line $\lambda(y)$ determines the set $R(y)$ with which the graph $G(y)$ and the data structure $S(y)$ are connected. The data structure $S(y)$ stores items of the set $R(y)$ which are intervals on the line $\lambda(y)$. This data structure $S(y)$ is introduced to enable the $O(\log N)$ query time. The simplest example with this property is a binary search tree. It will be shown later, however, that different data structures (red-black binary search tree) can be more efficient in a dynamic case.

Furthermore, the data structure $S(y)$ depends on the topology, and not on specific geometrical parameters. The particular interval from the set $R(y)$ forms a *key* used while constructing and accessing $S(y)$, i.e., the key depends on geometrical boundaries of the interval. But as long as the order of items of $G(y)$ does not change (none are inserted to or deleted from the set) the data structure itself does not differ. It leads to the following lemma (Preparata and Tamassia, 1992).

Lemma 2.2. *The same data structure is sufficient for searching in the 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 $y_i < y < y_{i+1}$, then it fulfills conditions of Lemma 2.1, and additionally the same edges trim the intervals. Therefore, it follows that the 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 2.2, we can assert that the data structure needs no changes at a given level. If there are n vertices in the mesh, $N-1$ levels exists. In particular, $\sigma_i = \{y : y_i < y < y_{i+1}\}$. The entities R_i, G_i and S_i can be now parameterised by the level number.

Suppose now that the point $p = (x_*, y_*)$ belongs to the level σ, i.e. $y_* \in \sigma$, while the corresponding search data structure is $S(\sigma)$. The keys present in $S(\sigma)$ are all in fact linear functions of the height y. Therefore, all localisations at the level σ are performed at the same computational cost.

One should consider now what are the differences between the levels i and $i+1$ (when the line $\lambda(y)$ crosses the vertex v_i). Obviously, the graph $G(y)$ changes its topology, while S_{i+1} is different than S_i.

An important remark, however, is that the change reduces to few basic possibilities (see Fig. 2):

a) deleting the cells for which v_i is the top vertex

b) inserting the cells for which v_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 2.3. *The number of elementary changes (see above) necessary to transform* $S(y')$ *into* $S(y'')$ *is equal to* m_i *(where* m_i *is the number of cells containing the vertex* v_i*).*

Consider an example. Two horizontal lines in Fig. 2 (light and dark gray) correspond to the levels σ_i and σ_{i+1}, $\sigma_i = \sigma_9$ and $\sigma_{i+1} = \sigma_{10}$. If the horizontal 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. the edge fj replaces fg).

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.

3. The algorithm

It is possible now to present the 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 nonintersecting segments (as described in the previous Section).

The point location algorithm consists of two queries. In the first one, y_* is located on an appropriate level, in the second, p is located between appropriate segments. The query cost is therefore at most $2 \log N$. The number of segments M is at most $N + 1$. Therefore, the size of the data structure S is proportional to $N \cdot M \sim N^2$. For large meshes ($N > 10^6$), this is fully unacceptable.

The reader may notice that the 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 overcome this problem, we can still

use the fact that the consecutive substructures, say S_i and S_{i+1}, differ only by a few elementary operations. Thus, we will attempt to add the persistence trying to store the changes to the structure, rather than keeping the structures themselves. One must observe, however, that the 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. 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 the *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 exists, called *colour*, which is necessary to preserve the balancing. For all details of the algorithms of inserting, finding and deleting an item from the tree, see Cormen (1990). 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.

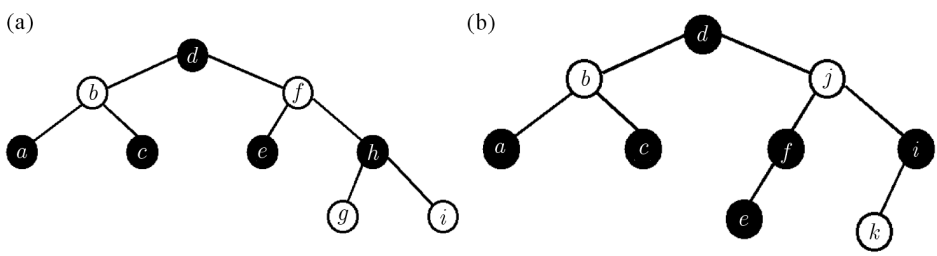

Fig. 3. The tree corresponding to the level σ_9 (a) and level σ_{10} (b)

The red-black binary search trees in Fig. 3a and 3b were built from intervals on the level σ_9 and σ_{10}, respectively (see Fig. 2). The letters in the nodes represent these intervals. Principles of construction of the red-black binary search tree follow the algorithm presented in Cormen (1990). Considering that the sets of intervals R_9 and R_{10} are totally ordered, this ordering allowed one to build the structures S_9 and S_{10}.

To add persistence to the data structure, we have chosen *the fat node method* (Discroll *et al.*, 1989). It is based on recording all changes made to a node in the node itself. As a consequence, each node contains a 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 and three are sufficient to delete the node (Discroll *et al.*, 1989). In such a way (in the most pessimistic case), there are five changes of the 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). Important thing is that η and ζ are both $O(1)$, and not $O(N)$. Similarly, the scope of changes within the tree is limited and relates to nodes neighbouring in the sub-tree. Thus, since the red-black binary search tree is balanced, the changes concern nodes located in the immediate neighbourhood of the 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 $\widetilde{S}_i$. Each node represents a cell (or strictly speaking a slice of the cell at the present level). This node has a left and right child.

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 number s_j of entities in these collections, as it was mentioned earlier, is expected to be $O(1)$. Additionally to speed up the search, the collections themselves can be ordered.

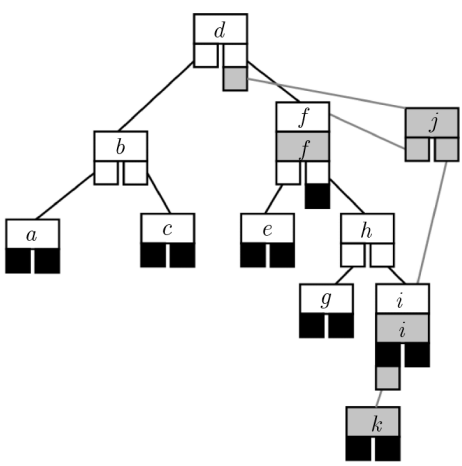

Fig. 4. The persistent binary search tree for two σ_9 and σ_{10} levels in the mesh

It remains to show that the total size M of the full date structure is now estimated as

$$M = \beta N \tag{3.1}$$

while the point location query cost C remains

$$C = \alpha \log N \tag{3.2}$$

where α and β are coefficients that depend on properties of the algorithm. This hypothesis will be verified by a numerical experiment in the next section.

4. Numerical experiment

In order to verify the presented algorithm, the following numerical experiment was performed. A sequence of meshes was generated in a pentagonal domain with N ranging from 1239 to 1846197. Each experiment consisted in the localising of one million random points. Every elementary step performed by the algorithm was counted to measure the *search cost*. The largest value for all queries was chosen for each mesh.

The obtained total cost is presented in Fig. 5 which indicates perfect logarithmic behaviour. The coefficient α, Eq. (3.2), is additionally shown in Fig. 6.

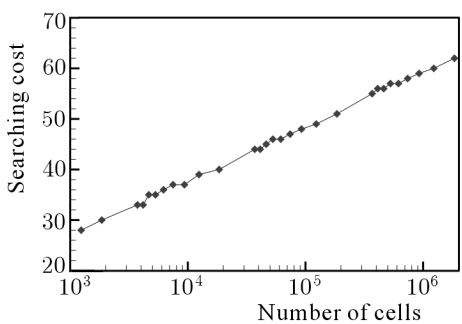

Fig. 5. The cost of the point location (measured by counting elementary operations)

Similarly, the total memory storage was measured in each experiment by making use of the Linux *ps* command. The result measured in bytes is shown in Fig. 7. One can see that this value grows linearly with the number of cells. The coefficient β, Eq. (3.1), is shown in Fig. 8. It must be noted that this result represents the size of memory used by the process, and not necessarily

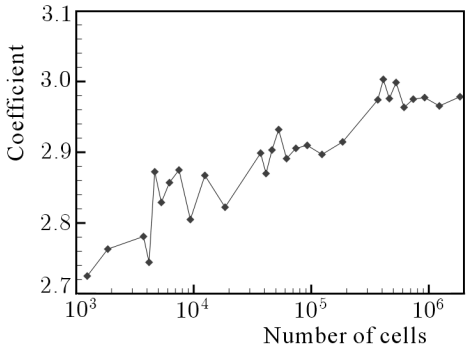

Fig. 6. The point location query cost coefficient α

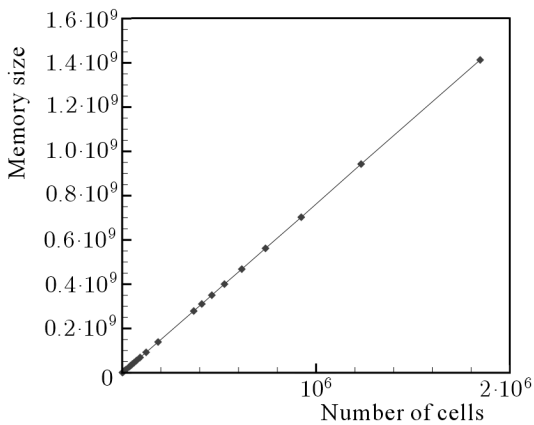

Fig. 7. Memory size of the data structure (measured in bytes)

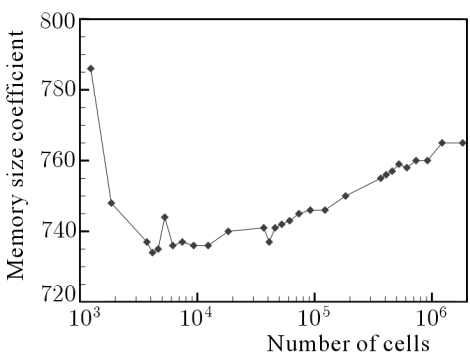

Fig. 8. The memory size coefficient β

the size of the data structure (which, we believe, is smaller). The suspected discrepancy may be caused by the memory leakage typical when the space is allocated and deallocated in a repetitive manner.

5. Concluding remarks

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

References

1. de Berg M., van Kreveld M., Overmars M., Schwarzkopf O., 1997, *Computational Geometry: Algorithms and Applications*, Springer-Verlag
2. Cormen T.H., Leiserson C.E., Rivest R.L., 1990, *Introduction to Algorithms*, The MIT Press
3. Discroll J.R., Sarnak N., Sleator D.D., Tarjan R.E., 1989, Making data structures persistent, *J. Comput. Syst. Sci.*, **38**, 1, 267-280
4. Preparata F.P., Tamassia R., 1992, Efficient point location in a convex spatial cell-complex, *SIAM J. Comput.*, **21**, 2, 267-280

Algorytm szybkiej lokalizacji punktu na siatkach trójkątnych

Streszczenie

Artykuł przedstawia wyniki badań nad zastosowaniem dynamicznych struktur danych (ang. *persistent data structures*) do planarnej i przestrzennej lokalizacji punktu w siatce obliczeniowej, z wykorzystaniem czerwono-czarnych drzew poszukiwań binarnych, które odpowiadają strukturze dwuwymiarowego kompleksu komórek. Rozważana siatka obliczeniowa składa się z komórek trójkątnych (w dwóch wymiarach) albo czworościennych (w trzech wymiarach). Ten fakt zezwala na znaczne uproszczenia realizacja totalnego porządku niezbędnego do budowy drzewa poszukiwań, jak również konstrukcji samego czerwono-czarnego drzewa poszukiwań binarnych. Wydajność algorytmu jest sprawdzone dla różnych siatek obliczeniowych (zawierających od 1239 do 1846197 komórek). Wyniki eksperymentu numerycznego potwierdzają logarytmiczny czas lokalizacji i liniowo rosnące zużycie pamięci przy wzroście rozmiaru siatki.

Manuscript received January 7, 2007; accepted for print February 7, 2008