https://doi.org/10.17721/1812-5409.2020/4.9

УДК 004.021+519.16

Терещенко В. М. [1], д.ф.-м.н., проф.,
Марченко О.О. [1], д.ф.-м.н., проф.
Терещенко Я. В.[1] , аспірант,
Тара О. М.[1] , студент

Оптимальний алгоритм динамічної підтримки Діаграми Вороного для множини точок

[1]Київський національний університет імені Тараса Шевченка,83000, м. Київ, пр-т. Глушкова 4д,

e-mail: vtereshch@gmail.com,
rozenkrans17@gmail.com,
yter2016@gmail.com,
olexandr.tara@knu.ua

V.N. Tereshchenko[1], Dr. Sci., pof.
A.A. Marchenko[1], Dr. Sci., Prof.
Y.V. Tereshchenko[1], Post-graduate,
A.N. Tara[1], student

The optimal algorithm for dynamic support of the Voronoi Diagram for a set of points

[1]Taras Shevchenko National University of Kyiv, 83000, Kyiv, Glushkova st., 4d,

e-mail: vtereshch@gmail.com,
rozenkrans17@gmail.com,
yter2016@gmail.com,
olexandr.tara@knu.ua

Стаття присвячена розробці динамічної структури даних для розв'язання задач близькості на основі динамічної Діаграми Вороного. Така структура даних може бути ядром моделі єдиного алгоритмічного середовища (МЄАС) та архітектури її реалізації на основі єдиної алгоритмічної платформи, для розв'язання комплексу задач візуалізації та комп'ютерного моделювання.
Структура даних побудована на основі стратегії «розділяй та володарюй» при побудові діаграми Вороного. Подібно до оригінального алгоритму, ми зберігаємо двійкове дерево, яке представляє діаграму Вороного, але визначаємо три нові операції: вставка, видалення та балансування. Для забезпечення ефективності операцій пропонується використати червоно-чорне дерево. Загалом запропонована структура даних показує набагато кращі результати, ніж оригінальний статичний алгоритм. В порівнянні з існуючими алгоритмами, дана структура являється одночасно простою та ефективною. На базі динамічної діаграми Вороного можливо створити єдине алгоритмічне середовище для ефективного моделювання динамічних процесів.

Ключові слова: динамічні структури даних, алгоритм, розділяй та володарюй, діаграма Вороного, найближчі сусіди, модель єдиного алгоритмічного середовища

The article is devoted to the development of a dynamic data structure for solving proximity problems based on the dynamic Voronoi Diagram. This data structure can be used as the core of the common algorithmic space model for solving a set of visualization and computer modeling problems.
The data structure is based on the strategy of "divide and rule" for Voronoi diagram construction. Similar to the original algorithm, we store a binary tree that represents the Voronoi diagram, but define three new operations: insert, delete, and balance. To ensure the efficiency of operations, it is proposed to use red-black tree. In general, the proposed data structure shows much better results than the original static algorithm. Compared to existing algorithms, this data structure is both simple and efficient.

Key Words: dynamic data structures, algorithm, divide and conquer , Voronoi diagram, nearest neighbour, model of common algorithmic space

Статтю представив д.ф.-м.н., член. кор.. НАН України, проф. Анісімов А.В.

Introduction

The nearest neighbor search is a fundamental problem of computational geometry, and its dynamic version is a powerful tool for solving many problems such as computer vision, statistical classification, DNA sequencing etc. Data structure that supports points insertions and deletions and queries for the nearest neighbor can be used to implement more complex systems like an autopilot for drones that uses computer vision. Firstly, Agarwal and Matoušek's in paper [1] proposed several data structures with updates in $O(n^{\varepsilon})$ amortized time and queries in $O(log\ n)$ worst-case time, while the other supports updates in $O(log^2 n)$ amortized time queries in$O(n^{\varepsilon})$ worst-case time, for any $\varepsilon > 0$.
Also techniques of Bentley and Saxe [2] yield an$O(log^2 n)$ amortized bound for updates and queries. In work Chan [3] obtained data structure with $O(log^3 n)$ insertion time , deletions in $O(log^6 n)$ amortized time, and nearest-neighbor queries in $O(log^2 n)$worst-case time. One of the possible solution of the nearest neighbour problem is to use a dynamic Voronoi diagram. Current version of dynamic Vorionoi diagram based on modified sweepline Fortune algorithm [4]. Additionally, this data structure could be used as a kernel for MCAS [5] to construct efficient computational geometry algorithms.
Despite the results of the mentioned works, the lower estimate of the complexity of the dynamic nearest neighbor search has not yet been achieved, and therefore work in this direction has not yet been completed.

Main approach

Data structure for a dynamic nearest neighbor

An important aspect of the choice of approach was novelty: during the analysis of existing papers, no works were found that use the principle of "divide and conquer" in a dynamic version. Although in the worst case the above algorithm loses to the already known ones, it is impractical to think only about the worst case, because it is rather exceptional. The efficiency of the above algorithm is ensured by the properties of the Voronoi diagram given above, which, however, do not hold in the general case.
Properties of the Voronoi diagram:
1. The dual graph to the Voronoi diagram (in the case of Euclidean space with point sites) corresponds to the Delaunay triangulation for the same set of points.
2. In the Euclidean plane, two points are adjacent on a convex hull if and only if their Voronoi cells share an infinitely long side.
3. If the space is normalized and the distance to each site is known (for example, when the site is a compact set or a closed sphere), then each Voronoi cell can be represented as a union of segments of rows coming from the sites.
4. Under relatively general conditions Voronoi cells have the property of stability: a small change in the shape of the sites, for example, a change caused by some transfer or distortion, leads to a small change in the shape of Voronoi cells. This is the geometric stability of Voronoi diagrams. This property does not occur in the general case, even if the space is two-dimensional and the plots are points.

Having a Voronoi diagram it is very easy to find the nearest neighbor of the entry point: you need to find a cell of the diagram that contains the entry point and the vertex of this cell will be the closest point to the entry point. The diagram itself does not allow you to quickly search for a cell, but using point localization algorithms, you can perform this operation in $O(log\ n)$.
The proposed data structure is based on a dynamic version of the known algorithm for constructing a static Voronoi diagram using the principle of "divide and conquer", so for a better understanding we present a static algorithm for constructing a Voronoi diagram.

Algorithm for static Voronoi diagram using the "divide and conquer" method.

Shamos and Hoey [6] presented the first $O(n\ log\ n)$ deterministic algorithm for computing the Voronoi diagram in the plane that is optimal in worst case sense. This algorithm is significant from a theoretical standpoint not only because it was the first one but also it uses the divide-and-conquer paradigm. The 'Divide-and-Conquer' is one of the fundamental paradigms for designing efficient algorithms. In this paradigm, the original problem is recursively divided into several simpler sub-problems of roughly equal size, and the solution of the original problem obtained by merging the solutions of the sub-problems. In the

divide-and-conquer approach of Shamos and Hoey, the set of sites, S, is split up by a dividing line into subsets S_L and S_R same sizes. Then, the Voronoi diagram, $Vor(S_L)$, of subset S_L and Voronoi diagram, $Vor(S_R)$, of subset S_R are computed recursively [7].
The algorithm for static Voronoi Diagram:

INPUT: The number n > 3 of sites and the list $S = (s_1, s_2, \dots, s_n)$of the sites in increasing order with respect to x-coordinates.

OUTPUT: Voronoi diagram $Vor(S)$.
1. L
et t be the integral part of n/2 and split S into $S_L = (s_1, s_2, \dots, s_t)$ and $S_R = (s_{t+1}, s_{t+2}, \dots, s_n)$.
2. C
onstruct the Voronoi diagram $Vor(S_L)$ of $S_L$recursively.
3. C
onstruct the Voronoi diagram $Vor(S_R)$ of S_R recursively.
4. M
erge$Vor(S_L)$ and $Vor(S_R)$ into the Voronoi diagram of $Vor(S)$ i.e., $Vor(S) = Vor(S_L) \cup Vor(S_R)$ by algorithm MERGE_VORONOI.
5. R
eturn $Vor(S)$.

Dynamic Voronoi Diagram algorithm

The algorithm described in the previous section was used as a basis: the set of points sorted by x coordinate is divided by vertical lines in half recursively. We will keep such structure in a binary tree. The nodes contain a static Voronoi diagram. The leaves of the tree contain trivial diagrams no larger than three. An arbitrary vertex, except leaves, contains a diagram which is obtained by merging two diagrams contained in the two descendants of the vertex. Each vertex, except leaves, has exactly 2 descendants by construction.
The Voronoi dynamic diagram must support two basic operations: inserting and deleting a point.
Next we describe the logic of these operations.

Insertion:
```
IF Vor(S) is empty THEN root ← p
ELSE
   cur ←DESCEND(p)
   IF cur.diagram.size() < 3
      cur.diagram.insert(p)
      MERGE_UP()
   ELSE
      DIVIDE(cur.diagram + p)
      MERGE_UP()
```

In *cur* we get the minimum static Voronoi diagram with no more than 3 vertices (a tree leaf) by the DESCEND method. For it, the insertion of a point occurs through a complete rebuilding of the diagram. If *cur* contained 3 points, we split it into two diagrams of two points (3 from the original cur chart plus one that we inserted) using DIVIDE. Next is the restructuring of $Vor(S)$ from the bottom up. Here we rely on the property of the Voronoi stability diagram - inserting a single point requires a local change of the diagram, and therefore MERGE_UP doesn't rearrange the diagram completely most of the time. The DESCEND, DIVIDE, MERGE_UP will be described after the point removal algorithm.

Deletion:
```
IF Vor(S) contains p THEN
   cur ← DESCEND(p)
   IF cur.diagram.size == 3 THEN
   cur.diagram.remove(p)
   MERGE_UP()
ELSE
   cur <- cur.parent
   IF cur.left.diagram.size == 2 &&
      cur.right.diagram.size == 2 THEN
      vertices ←cur.diagram.vertices
      vertices.remove(p)
      cur ← NEW_DIAGRAM(vertices)
      MERGE_UP()
   ELSE
      DIVIDE(cur.diagram – p)
      MERGE_UP()
```

The point deletion algorithm works in such a way that a node cannot contain a diagram of size 1. Suppose that when deleting a point p we go down to a leaf containing diagrams $Vor(S_i)$ of size 2 formed from two of its descendants. If the size of $Vor(S_i)$is 3, then we simply remove the point p from $Vor(S_i)$. After that, as in the insertion algorithm, there is a merge from the bottom up using MERGE_UP.

DESCEND:
```
WHILE cur.diagram.size > 3
```

IF p is in cur.left THEN
cur ←cur.left
ELSE cur ←cur.right

To descend the tree, use the location of the point p relative to the dividing line. For example, if the point is to the left of the dividing line, then the desired trivial diagram is also to the left of the line. If the point is on the chain, it does not matter which way to go down.

MERGE_UP:
merged ← MERGE(cur.left.diagram, cur.right.diagram)
WHILE cur.diagram.dividing_chain != merged.dividing_chain DO
cur .diagram ← merged
cur ← cur.parent
merged ←MERGE(cur.left.diagram, cur.right.diagram)

The condition of completion of the merge algorithm is important: if after merging two diagrams $Vor(S_i)$, $Vor(S_j)$ the dividing chain of$Vor(S_i \cup S_j)$has not changed, the diagram will not change anymore and we can stop merging.

DIVIDE:
vertices ← diagram.vertices
vertices.sort()
cur.left ←StaticDiagram(vertices[1, 2])
cur.right ←StaticDiagram(vertices[3, 4])

An example insertion process can be seen in Fig. 1. This figure shows the resulting diagram after each insertion. Now let us look at the insertion process in depth. Fig. 2 and Fig. 3 shows the tree data structure representing the dynamic voronoi diagram. We can see that 5 point diagram is represented as 2 trivial diagrams: 3 points in the left diagram and 2 points in the right one. Since the 6th point lies within the left diagram, the DESCEND algorithm will add it to the left diagram. Now the left diagram grows from 3 points to 4. This means the resulting diagram is not trivial and we need to divide it using the DIVIDE algorithm into two trivial diagrams which are now children of the 4 point diagram.

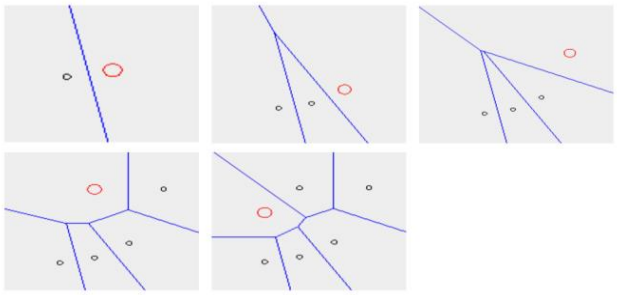

Fig. 1. Insertion of 6 points. Red circle denotes the newly added point.

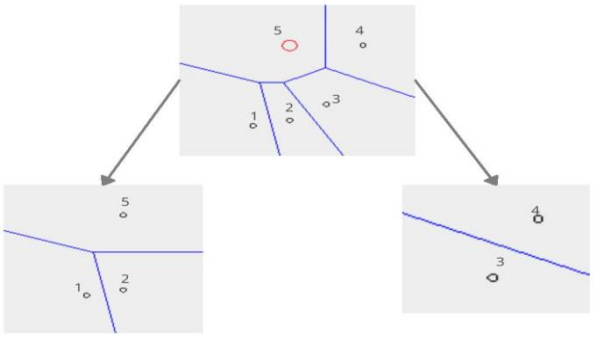

Fig. 2. Dynamic Voronoi diagram tree for first 5 points.

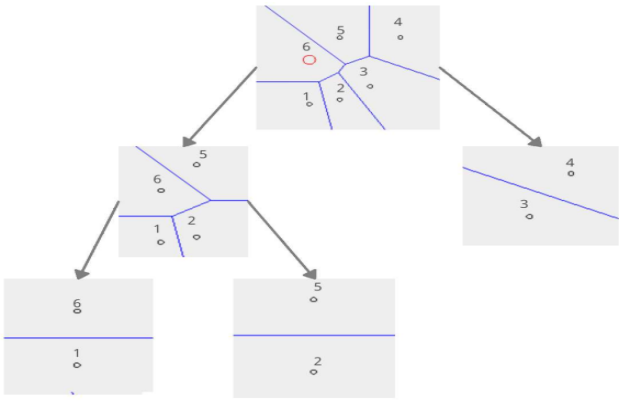

Fig. 3. Dynamic Voronoi Diagram tree for the final 6 point diagram.

Complexity analysis

Teorem. The data structure allows dynamic mainten the Voronoi Diagram, when a point is inserted or removed, for *O*(*n*) time in the worst case and *O*(log*n*) in average:
1) search for the nearest neighbor can be find in O(logn) time;
2) insert and delete a point in the data structure in O(logn) time.

Proof :

Lemma 1*(Nearest neighbour query).* Nearest neighbour query for dynamic Voronoi diagram can be done using $O(log\ n)$processing time.

Proof. To find the nearest point, you need to find the cell of the Voronoi diagram of the whole set of points and return its vertex. The Voronoi diagram of the whole set of points is located in the root of the tree, i.e. it is accessed in a constant time. Next, the search for the Voronoi cell to which the entry point belongs is performed using $O(log\ n)$localization algorithms. After that, we can immediately get the vertex of the localized cell, and hence the nearest point. Thus, the complexity of processing the search request for the nearest neighbor is $O(log\ n)$.

For optimal runtimes of insert and delete operations, an important factor is balancing the Voronoi diagram tree. Since the tree itself is binary, we can use red-black tree to keep it balanced. Next we will assume that the tree is balanced.

Lemma 2 *(Insertion and deletion)***.** Insertion and deletion for dynamic Voronoi diagram can be done using $O(n)$ processing time in worst case and $O(log\ n)$for a large amount of points.

Proof. When descending to the leaf of the tree we iterate over a logarithmic number of nodes. Each node takes a constant time. Therefore we can descend in $O(log\ n)$. The merge operation is done in linear time and in the worst case it is necessary to perform the merge all the way from the leaf to the root. Assuming the worst case scenario, we have a full binary tree with leaves containing 3-point trivial diagrams. In this case we start by merging two 3-point diagrams, then two 6-point diagrams and so on. Overall this results in:

$$\sum_{i=1}^{log_2(n)} 2^i \times 3 = 6(2^{log_2(n)} - 1) = 6(n-1)$$

which results in the computational complexity of the merge operation being $O(n)$. However, for large values, only a small part of the diagram will be updated: this follows from the stability property of the Voronoi diagram described in the first section. Therefore, time complexity for a large number of points could be reduced to $O(log\ n)$.

The deleting time complexity is the same as for insertion.

Comparison

Leading articles on the topic are Agarwal and Matushek [1] and Chan [3]. The first article describes two data structures: one supports inserting/deleting vertices in $O(n^{\varepsilon})$ on average and processing requests in $O(log\ n)$ in the worst case, and the second supports insertion/deletion in $O(log^2 n)$ on average and processes queries in $O(n^{\varepsilon})$ in the worst case, for an arbitrary $\varepsilon > 0$. The second article first describes the data structure that works in polylogarithmic time for shells in three-dimensional space, and hence for the two-dimensional case of the nearest neighbor. Chan's data structure supports insertions in$O(log^3 n)$ on average, deletions in $O(log^6 n)$ on average, and processing requests for an extreme point and the nearest neighbor in $O(log^2 n)$ in the worst case.

Thus, in the worst case, the proposed data structure behaves on par with the one proposed by Agarwal and Matushek, and on average even better than it.

Testing

We implemented a dynamic Voronoi diagram using the proposed algorithm. The implementation was done in Java with static implementation. The result is demonstrated in Fig. 4.

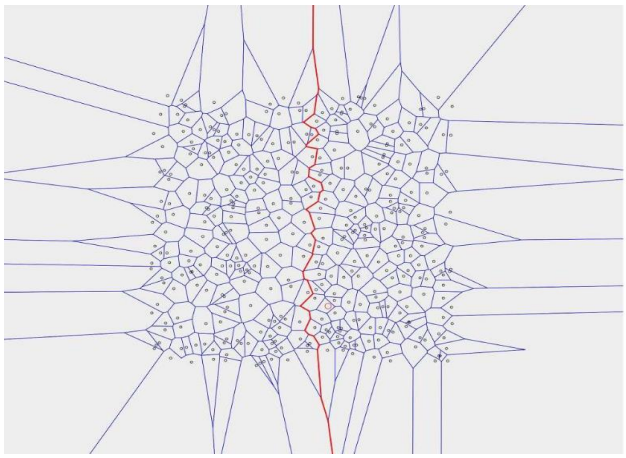

Fig. 4. Example diagram. Red line denotes the dividing chain of the root diagram.

It was tested on 1000 randomly generated set of points. The results are shown in Fig.5. The peaks on the dynamic times graph correspond to the rebalancing that is done by completely rebuilding the diagram. This process can be made faster by implementing a red-black tree data structure for the dynamic Voronoi diagram tree. The data for the yellow graph was obtained by building a new static Voronoi diagram at each point insertion.

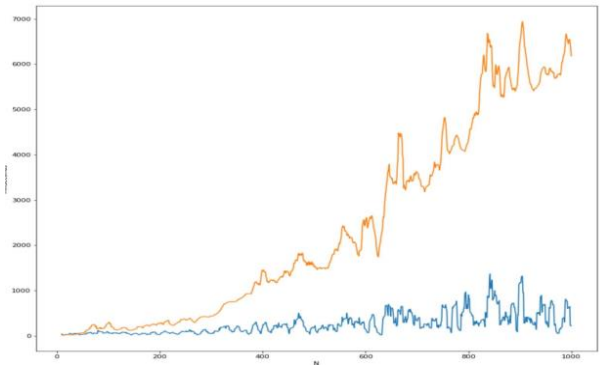

Fig. 5. Time taken to insert a point. Dynamic Voronoi diagram (blue) and Static Voronoi diagram rebuilding (yellow). 1000 points were inserted.

Conclusion

We showed a novel algorithm for the dynamic Voronoi diagram that performs significantly better than the full Voronoi diagram rebuild. It uses a well-known concept "divide and conquer" and therefore is easy to understand.

The algorithm for dynamic Voronoi diagram was implemented in Java and is now open-source. It was tested against a static Voronoi diagram.

Список використаних джерел

1. *Agarwal P. K., Matusek, J.* Dynamic half-space range reporting and its applications. Algorithmica. 13. p. 325–345., 1995
2. *Bently J. L., Saxe J. B.* Decomposable searching problems: Static-to-dynamic transformations. J. Algorithms. 1. p. 301–358., 1980
3. *Chan T. M.* A dynamic data structure for 3-D convex hulls and 2-D nearest neighbor queries. J. ACM. 57(3), # 16., 2010
4. *Fortune S. A.* sweepline algorithm for Voronoi diagrams. Algorithmica. 1987. Vol. 2, No. 1-4. P. 153–174.
5. *Tereshchenko V. N., Budjak I., Fisunenko A.* The Unified Algorithmic Platform for Solving Complex Problems of Computational Geometry, Parallel Computing Technologies. 2013. Vol. 7979. P. 424-429.
6. *Shamos, M. I., Hoey, D.* Closest-point problems. In 16th Annual IEEE Symposium on Foundations of Computer Science. October 1975. pp. 151–162.
7. *Sack J. R., Urrutia* J., Handbook of Computational Geometry, Elsevier Science, Netherlands (2000)

References

1. AGARWAL P. K., MATUSEK, J.(1995). Dynamic half-space range reporting and its applications. Algorithmica. 13. p. 325–345.
2. BENTLY J. L., SAXE, J. B. (1980). Decomposable searching problems: Static-to-dynamic transformations. J. Algorithms. 1. p. 301–358.
3. CHAN T. M. (2010) A dynamic data structure for 3-D convex hulls and 2-D nearest neighbor queries. J. ACM. 57(3), # 16.
4. FORTUNE S. A. sweepline algorithm for Voronoi diagrams. Algorithmica. 1987. Vol. 2, No. 1-4. P. 153–174.
5. TERESHCHENKO V. N., BUDJAK I., FISUNENKO A. The Unified Algorithmic Platform for Solving Complex Problems of Computational Geometry, Parallel Computing Technologies. 2013. Vol. 7979. P. 424-429.
6. SHAMOS M. I., HOEY, D. (1975). Closest-point problems. In 16th Annual IEEE Symposium on Foundations of Computer Science. October 1975. pp. 151–162.
7. SACK J. R., URRUTIA J., Handbook of Computational Geometry, Elsevier Science, Netherlands (2000)

Робота надійшла до редакції 29.11.20