Unified data structures for solving optimally a set of interrelated computational geometry problems

Vasyl Tereshchenko[1] and Semen Chudakov[2]

1 **Faculty of computer science and cybernetics, Taras Shevchenko National University of Kyiv, Ukraine**
vtereshch@gmail.com

2 **Faculty of computer science and cybernetics, Taras Shevchenko National University of Kyiv, Ukraine**
semen.chudakov7@gmail.com

Abstract

This paper is devoted to the development and of an efficient algorithmic model for solving a set of interrelated computational geometry problems. To do this, a unified algorithmic environment with unified data structures is created, which allows to implement complex use cases efficiently with respect to computational resources. We build the environment on the basis of the "divide and conquer" strategy. Once a convex hull is key to a set of computational geometry problems, we offer a concatenable queue data structure to maintain it. The data structure is implemented in a form of a binary tree. This allows to perform operations needed in algorithm for a set of problems in $O(\log n)$ time. Furthermore we offer a way to execute the algorithms both sequentially and in parallel. In the future the algorithmic environment can be improved to support other computational models with similar properties for solving problems. As an example, the Voronoi diagram or the Delaunay triangulation can be considered.

Lines 138

1 Introduction

Nowadays, advanced computer simulations and visualizing of complex scientific researches as well as large scale technical projects requires to simultaneously solve a set of problems. The core of this set are problems of computational geometry and computer graphics. To solve such problems it is needed to create suitable algorithmic frameworks, that would yield accurate results in real time. Existing methods (iLastic [**?**], IMARIS [**?**], ImageJ [**?**]), that are based on a set of algorithms implementations organized in a package does not result in desirable efficiency and accuracy. It is worth noting, that there are a lot of parallel algorithms designed to solve specifically certain computational geometry problems such as in [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. Every such algorithm requires its own computational resources and is executed independently from others. In such case some identical steps, such as preprocessing and building data structures, are performed several times.

Therefore, an important objective in developing the algorithmic models is to create a universal tool, which would have means to efficiently solve a set of problems. This tool should also execute identical steps of the algorithms once and be able to represent results of those steps in a form of the unified data structures. In [**?**] the notion of a unified algorithmic environment is introduced, which is based on the "divide-and-conquer" principle and takes into account the aforementioned features of the algorithms. In particular, the preprocessing and splitting the initial set of data to form the recursion tree is common for all problems and is executed only once. During the merge stage intermediate results are maintained in a

36th European Workshop on Computational Geometry, Würzburg, Germany, March 16–18, 2020.
This is an extended abstract of a presentation given at EuroCG'20. It has been made public for the benefit of the community and should be considered a preprint rather than a formally reviewed paper. Thus, this work is expected to appear eventually in more final form at a conference with formal proceedings and/or in a journal.

weighted concatenable queue. This model does not repeat computations and the intermediate results are highly reused during the algorithms, which yields good performance.

In this article we first describe how the convex hull algorithm for a static set of points is decomposed into separate stages and incorporated into our unified algorithmic environment model (UAEM). Then we provide detailed explanation on how the concatenable queue which is used in the algorithm is implemented. Finally we make complexity analysis for the algorithm and evaluate its performance.

2 Unified algorithmic environment

2.1 Algorithms stages

In this section we describe the principle of how we decompose each algorithm into distinct parts. This partition is then used to avoid repeating the computations in the algorithmic environment. The principle will be show on a convex hull algorithm which is similar to the one describe in [**?**] but operates on a static set of points.

The notion of a convex hull is simple. For a set of points S in a k-dimensional space it is a smallest convex set, that comprise S. In practice to solve such problem, means to find a subset in S, which can be a "skeleton" for the convex hull.

In the preprocessing stage, the "inner" points, which lie on a horizontal or vertical line, are removed from the set. Formally, the removal criterion is formulated as follows. For $a = (x_a, y_a)$ we denote $x(a) = x_a$, $y(a) = y_a$. Let points $a_1, a_2, ..., a_k$ lie on one horizontal line and $x(a_1) < x(a_2) < ... < x(a_k)$. Then, by the criterion, the points $a_2, a_3, ..., a_{k-1}$ must be removed. Analogously for the vertical case.

Consider an algorithm that, in a sorted array, for each group of identical elements, deletes all but the first and the last one (if there are more than two repetitions). The idea behind the algorithm is to use two pointers technique to delete repetitions by overwriting their place with other non-repeating element. This way, additional memory usage can be reduced to a constant value. The algorithm makes one pass through the array. At each step, a constant amount of work is performed to decide whether to delete the current item. Given this, the complexity of the above algorithm is $O(n)$.

To perform the preprocessing described above, it is needed to:

1. Sort points by y (if y coordinates are equal, the x coordinates are compared).
2. Delete repetitions by y coordinate using the described algorithm.
3. Sort points by x (if x coordinates are equal, the y coordinates are compared).
4. Delete repetitions by x coordinate using the described algorithm.

In the result we get a set of points for which we can apply the recursive convex hull algorithm.

At the stage of splitting the initial problem, the set of points is divided into left and right parts of equal size. Since a array-like data structure is used to store the points, this operations can be completed in $O(1)$ using the formula below:

$$M_{i,j} = \frac{i+j}{2} \tag{1}$$

The recursion stops when there are no more than 3 points in the set.

For the base case, the set of points can have 2 or 3 elements. In the first case the highest of two points forms the upper part of the hull, and the lower - the lower part. To consider the base case of 3 points, we introduce an additional notion of the tangent slope given by

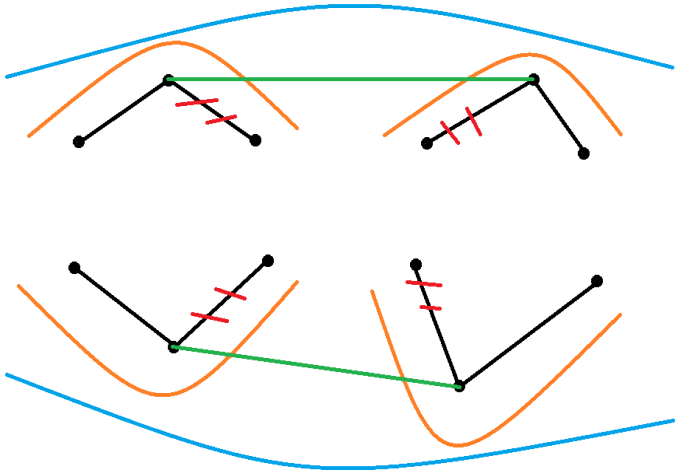

Figure 1 Merging step of two hulls

two points on the plane. For arbitrary points a_1, a_2 such that $x(a_1) < x(a_2)$ slope is denoted as λ:

$$\lambda(a_1, a_2) = \frac{y(a_2) - y(a_1)}{x(a_2) - x(a_1)} \tag{2}$$

According to this definition, the possible cases for points a_1, a_2, a_3 are given in Table **??**:

In [**?**] an algorithm for merging two convex hulls is described. The idea of this algorithm is to maintain the hull in two concatenated queues. The hull is divided into two parts. In this article it is divided into the upper and the lower sub-hulls. Then, using the two-pointer technique and the cases described in [**?**], the proper tangent line is searched. It remains to split the two queues at the found vertices that form the tangent and to merge the remaining parts. An example of performing such a procedure is shown in Fig. 1.

It remains to consider the corner cases that arise when performing the merging. The first of these cases is related to the ambiguity of the position of the utmost points in the described representation. The leftmost point of the left hull and the rightmost point of the right hull must belong to the upper parts of the view before finding the tangent line, because otherwise such tangent may be found incorrectly. An example of such incorrect search is given in Fig. 2.

To avoid such a situation, it is necessary to move the indicated points to the upper sub-hulls before merging them. For the rightmost point of the left hull and the leftmost point of the right hull we have the following cases. Similarly to the previous argument, they must be transferred to the upper parts of the hulls. And after merging these points must be transferred to the lower parts of the hull, if they do not belong to the resulting upper part of the final hull. Otherwise, the formed hull may be incorrect. An example of such case is shown in Fig. 3.

After combining the parts of the convex hulls, another corner case might take place. The search for the tangent for the upper parts of the hulls does not take into account the position of the lower parts and vice versa. As a result, the upper and lower parts of the final hull may not form a coherent structure. An example of such a situation is shown in Fig. 4.

To avoid such a situation, it is necessary to perform the step of cutting off the redundant parts of the formed lower sub-hull. Then searching for the left and right pivoting vertices in

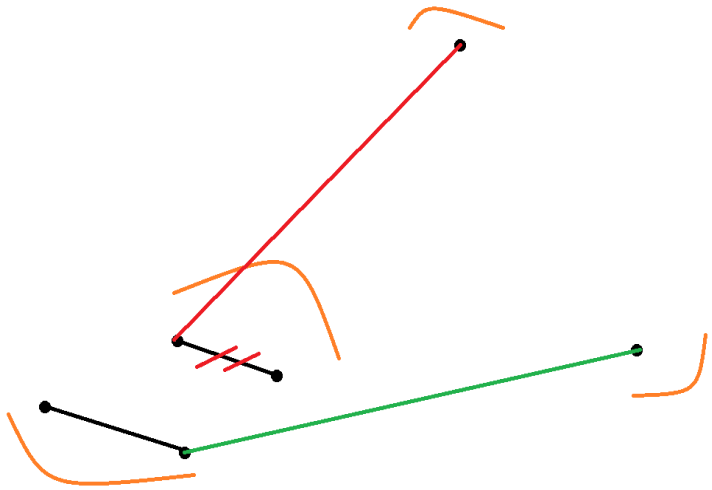

Figure 2 Example of an incorrect position of the utmost left in the left sub-hull

the concatenable queue is performed. After that, the queue is split over the found vertices. Fig. 5 shows correct convex hull.

2.2 "Divide-and-conquer" algorithm interface

The next goal of this work is to build a unified algorithmic environment. The construction of such an object requires the combination of an algorithmic database together with the necessary data structures. In fact, it is needed to create an interface of generic algorithm based on "divide-and-conquer" strategy, which will then be used on a specific input data.

We start creating the generic interface by listing its components:

- Preprocessing.
- Splitting task into sub-tasks.
- Solving sub-tasks.
- Merging obtained results.
- Checking, if a given input data is a base case for the algorithm.
- Solving the base case.

Each of these components will represent a function in the future interface. Here there is a clear separation between the input type for the algorithm and the type of the result it returns. These two types should be parameters of the algorithm model. One should also pay attention to the input and output types of functions in the interface.

A large number of computational geometry algorithms, such as computing minimal spanning tree, the Delaunay triangulation, the Voronoi diagram and the convex hull accept the list of points. However, it would be wrong to limit the input type in the algorithm interface to a list of points. The reason for this is the fact that some algorithms can use the results of other algorithms as input data. A well known example is the construction of a Delaunay triangulation based on the Voronoi diagram maintained in a special data structure.

Listing 1 shows the constructed algorithm model. IT denotes the input type and OT - output type.

References

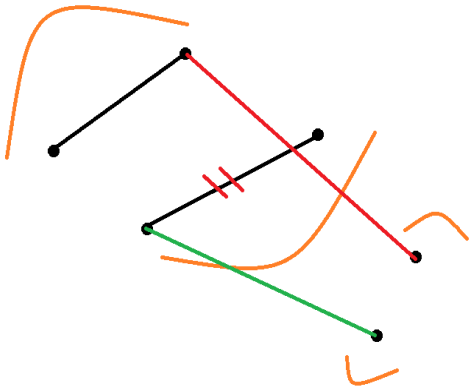

Figure 3 Example of a convex hull for a wrong position of the utmost left points of the left sub-hull

Listing 1 Algorithm model based on the "divide-and-conquer" principle

```
interface DaCAlgorithm[IT, OT]:
  boolean isBaseCase(IT input)
  int inputSize(IT input)

  OT solveBaseCase(IT input)
  IT preprocess(IT input)

  OT merge(OT first, OT second)
  Pair[IT, IT] divide(IT input)
```

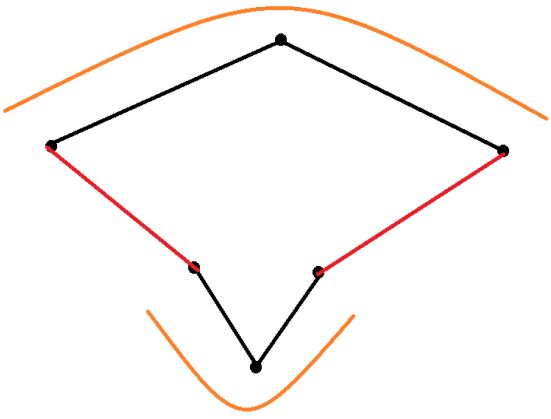

Figure 4 An example of a non-integral hull after merging along the reference lines

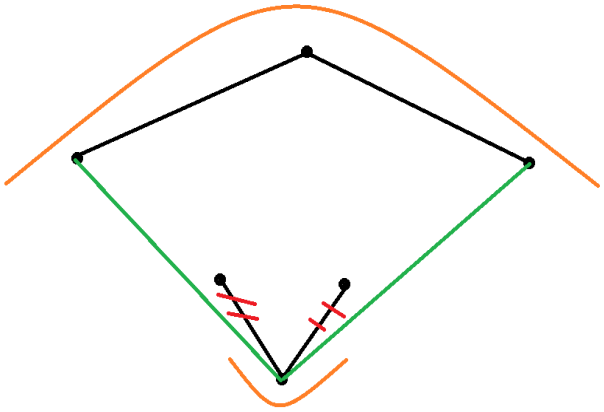

Figure 5 Correctly constructed convex hull

Figure 6 This was the logo of CCCG 2003.