# Merging Maps

Pavel Chalmovianský

Department of Geometry, Faculty of Mathematics and Physics
Comenius University, Bratislava

*chalmo@center.fmph.uniba.sk*
*http://www.uniba.sk/~chalmo*

The aim of the article is to describe a method of the merging several nonoverlapping maps together. It can be taken as an inverse process of cutting up the map into pieces by several axial cuts. The problem is closely related to computing the union of polygon and computing the contour of the union of polygons. One knows where the polygons can glue together.

## Introduction

GIS technologies processes usually great amount of data, however user needs often just special part of it. It is usually more flexible to work with clipped data to the area of the interest. Some useful references can be found in [1].

Our problem can be formulated as follows: *There is a collection of maps describing plots. Merge them together and create just one map containing the whole information without redundancies.* We are concerned with maps of plots describing landowners. The most stray element is the frame of each individual map, which can pass through plots and split them in a way which does not correspond to the real division of the land. Our task is to merge them together at the boundaries of each map. Such actions are required e.g. to print the whole map or to process plots as they are in reality.

The algorithm itself is not so difficult, if we would have exact matching data, however, the different sources of maps (scanning, computer aided production,...) produce small errors and should be removed by the merge process.

In fact (given by the application, where the algorithm plays the main role), we are finding the outer contour of the unified plots instead of finding their union. References on the topic can be found in [2,3].

## Basic Notions

Now we put down more exact definition of input data and used structures.

Let $M_0, \ldots, M_n$ denote maps. Each of them is of axial rectangular shape (see Fig. 1). Let $M$ be their union ($M$ is an axial polygon) and let the maps $M_i$ are arranged so that $M$ is without holes. We also assume, that in an ideal case $M_i$ and $M_j$ are overlapping just in case $i = j$. Each map contains polygonal plots. Let $P_{i,0}, \ldots, P_{i,n_i}$ denote the boundaries of the plots in $M_i$ and let $P$ denote the set of all in all maps. Each polygon is associated with the ID of the plot which is represented by the polygon. We have a restriction for ID numbers. In each $M_i$ there are no two polygons of the same ID having common intersection with respect to the open rectangle $M_i$ (see Fig. 2).

Our task is to form set $\{R_0, \ldots, R_m\}$ of plots polygons such, that $int\, R_i$ is maximal simple connected piece of the respective ID in $M$ (see Fig. 3).

Of course, brute force algorithm using general algorithm of union of the polygons is at hand, however this is very time consuming solution. We try to use the information, which is given by the arrangement of maps to produce better result.

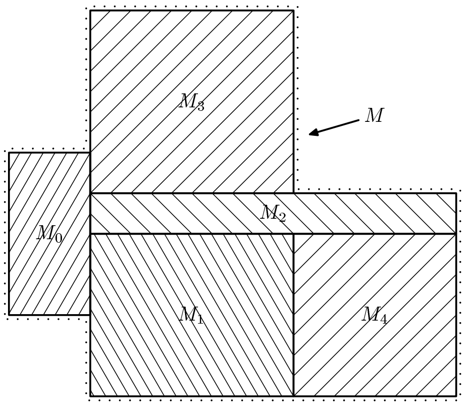


Fig.1 Arrangement of maps. Their union is $M$.

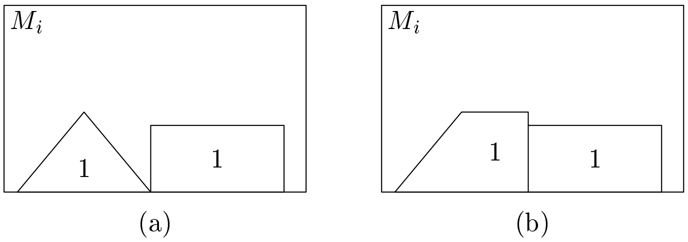


Fig.2 (a) correct numbering (b) forbidden numbering

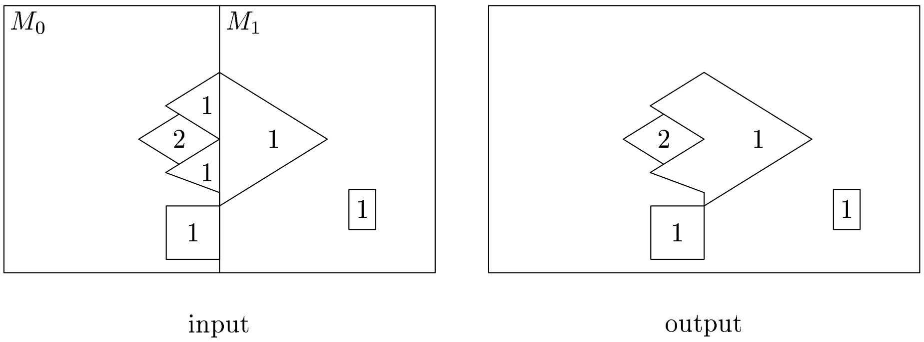


Fig.3 Merging polygons at the boundary of $M_0$ and $M_1$

## Gathering Information from Maps

Before merging polygons we make some pre-processing. The information retrieved from maps and their arrangement is as follows.

First we assign to each map the lists of its "west", "east", "south" and "north" map neighbors. Two maps are *neighbors*, if their boundaries share some segment part in common. If inaccuracies occur, it is better to add more neighbors. However, if map $M_i$ overlaps map $M_j$ at corner, map $M_i$ can occur just in one of the above lists.

Each map $M_i$ has associated the collection of points $C_i$. Then we gather the collection $S = \{S_0, \ldots, S_k\}$ of all points occurring in $C_i$, $i = 0, \ldots, n$. During this process we identify some points as the same with respect to chosen toleration and to each point in $C_i$ assign pointer to S. Further we assign to each point of map $M_i$ the flags referring whether or not they lie at the boundary of $M_i$, $i = 0, 1, \ldots, n$ (again with carefully chosen tolerance and for every boundary side of the map). Two consecutive vertices in some $P_{i,j}$ with the same boundary flag in some refer to the edge laying on boundary. These edges will be refereed as *boundary edges*. If edge lays on "north" boundary we say it has "north" *direction*. One can use the bucketing technique to efficiently solve above tasks.

## Merging Polygons

Now we have all necessary information to complete the task. The algorithm of the merging scans each polygon for boundary edges. If there is not such edge, polygon is flushed in the output. If there is one (say $f$) at the boundary of the map containing the edge, the polygon of the same ID is sought. If it is found, we try to merge them together. The following formal algorithm describes in general the way of merging the whole set of polygons $P$:

*Each polygon $P_i$ mark as not processed;*
**for** *(each not processed polygon $P_i$)* {
  *mark $P_i$ as processed and let $M_k$ is map containing $P_i$;*
  *orient $P_i$ counterclockwise;*
  **while** *(in $P_i$ is any boundary edge $f$)* {
    *let $d$ is the direction of edge $f$;*
    **do** {
      *find plot with the same ID in maps neighboring to $M_k$*
        *in direction $d$ and not processed (denote the result $T$)*
      **if** *(Found) $P_i$=*MERGE*($P_i$, $T$);*
    }
    **while** *(Found && $P_i$ and $T$ were not merged)*
  }
  *flush $P_i$ to output*
}

The main part of the previous algorithm is procedure MERGE($S$, $T$). It works in two stages. Let $T$ be oriented clockwise and $S$ be the result of the merge process. $S$ is, of course, oriented counterclockwise. Then there is a point $Q_s$ lying at the boundary of both polygons $S$ and $T$ and the vertex before in $S$ is not in $T$ and the vertex before $Q_s$ in $T$ is not in $S$ (see Fig. 4). ($Q_s$ need not be the vertex of both $S$ and $T$, but is a vertex of at least one of them.)

When we find it, we "walk" along common boundary of both $S$ and $T$, until we find the point $Q_e$ of the common boundary such, that the next vertex of $S$ is not a vertex of $T$ and vice versa. Then we cut the part between points $Q_s$ and $Q_e$ of $S$ counterclockwise and of $T$ clockwise and we connect the rests of the polygons. This is end in most cases occurring in maps, however, the common part of the boundary may have more than one component. Thus the result of the merge would not be correct in sense it is simple polygon. The second phase recognizes this situation and repairs the polygon. What we in fact must find is the outer contour of $S \cup T$.

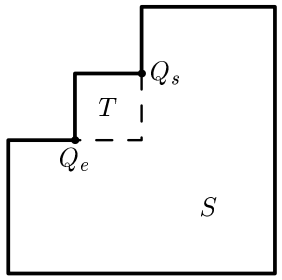


Fig.4 Startpoint and endpoint of to be removed (broken) line

Fig. 5 shows the case of necessity of cutting the parts $R'(alongS)P_s'Q''(alongT)R'$. Since the parts, where such situation can occur, lie at the boundary of some maps, we concern just at edges marked as lying at map boundary. Two edges $e_1$, $e_2$ in fact must obey following conditions:

a) $e_1 \subset S$, $e_2 \subset T$

b) bound flag of $e_1$ is directional opposite to bound flag of $e_2$

c) $e_1$ and $e_2$ are overlapping

At such edges is not our polygon simple and we must find the hole proceeding along these common edges and then around the hole until we find the last vertex at the common path. Such edges are e.g. $e_1 = P_s'P_e'$, $e_2 = Q_s'Q_e'$ and the hole is $Q_s'Q_e'P_s'Q_s'$. Then we check the orientation of the hole boundary and if it is clockwise, we cut this sequence and proceed further by restarting again the second phase of the merge process until we find counterclockwise part (it is outer contour). This removes any of the holes and common parts of the result.

At the end of the process we must obtain the topology of the polygons - two or more can lie in another one. This is done as postprocess and is not important for the algorithm. Here one must guard if some plot is lying in another one with the same ID. Then we must remove the underplot.

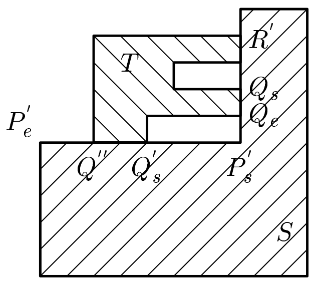


Fig. 5 Simplification of merged polygon

Now we can write in symbolic form the procedure MERGE$(S, T)$. The more detailed algorithm is very involving because of complicated "walking" along the common boundary. One must walk not only from edge to edge, but also between maps, if we have already polygon $S$ merged from several pieces.

```
Merge(S, T)
{
    orient T clockwise;
    do {
        find in T boundary edge with opposite boundary flag as f;
        if (Found) {
            try to find Q_s;
            if (Q_s is found) {
                "walk" along common boundary until Q_e is found;
                cut Q_sQ_e in S counterclockwise and in T clockwise;
                join rests of S and T;
                simplify result;
                return;
            }
        }
    }
    while (not Found);
}
```

The most time consuming part seems to be the simplification, if any occurs. Since it is almost identical to the merge process, it means just several runs of the same algorithm.

## Remarks on Complexity

Now we say something on complexity of the merge process. Usually we glue just few edges of the polygons, which have tens or hundreds of vertices. Thus we can say, that average complexity is far better than worst-case one. These edges can be easily identified after pre-processing. If there is $k$ edges at the specified boundary in $S$ and $k^*$ in $T$, there is no more than $O(kk^*)$ comparisons to find out $Q_s$. If it is found, then there is no more than $O(h)$ steps to find $Q_e$, where $h$ is the number of vertices at common boundary of two polygons. And finally this can be done proportionally to the number of components of common boundary of two polygons.

## Implementation

The whole algorithm was implemented as a part of software package for administrating landowners and now is in use for several month. It was designed in C, using Borland compiler under DOS in WIN32 mode. For completeness we state some run times for various inputs (designed at Pentium 66, 32 MB RAM, Windows 95) in Tab. 1.

The pre-processing time in the third row of the table is caused by the too central data. There was a map containing almost everything. The longer run time in the second row is caused by the low number of landowners and high number of divided plots.

In the Fig. 6 you can see input and output results of the algorithm.

The further improvement of the algorithm is sougth. We want to cover the topology of the plots – in the merge step one should also decide which plot contains another ones.

| # of points in maps | | # of plots in maps | | average # of points in plots | | Run time (in sec.) of | | # of maps |
|---|---|---|---|---|---|---|---|---|
| input | output | input | output | input | output | preprocess | algorithm | input |
| 4 058 | 3 795 | 232 | 154 | 17 | 25 | 4.34 | 2.20 | 10 |
| 8 446 | 8 088 | 1 572 | 1 418 | 5 | 6 | 10.49 | 20.60 | 10 |
| 7 160 | 6 636 | 902 | 719 | 8 | 9 | 109.91 | 4.76 | 14 |
| 2 074 | 2 026 | 55 | 44 | 38 | 46 | 1.09 | 1.10 | 6 |
| 27 024 | 24 785 | 7 071 | 5 553 | 4 | 9 | 97.87 | 52.79 | 35 |

Tab. 1 Results of running algorithm with real data

Fig. 6 Real data before and after running of the algorithm

## Acknowledgements

I must thank to my colleagues Robert Bohdal and Marek Mikita, which helped me in several particular details of the algorithm. Together with latter we programmed the algorithm in the package mentioned above.

References

[1] Okabe Atsuyuki, Boots Barry, Sugihara Kokichi: *Spatial Tessellations Concepts and Applications of Voronoi Diagrams*, John Willey & Sons, 1992

[2] O'Rourke, Joseph: *Computational Geometry in C*, Cambridge University Press 1994

[3] Preparata, Franco P., Shamos, Michael I.: *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985