RESEARCH ARTICLE

Paralleling generalization operations to support smooth zooming: case study of merging area objects

Dongliang Peng, Martijn Meijers and Peter van Oosterom

GIS Technology, Faculty of Architecture and the Built Environment, Delft University of Technology, Delft, The Netherlands

ARTICLE HISTORY
Compiled June 27, 2020

ABSTRACT
When users zoom out on a digital map, some area objects become too tiny to be seen, resulting in visual clutters. To avoid this problem, we merge the relatively unimportant areas into their neighbors to form larger areas. We define an *event* as merging an unimportant area into its most compatible neighbor (where the neighbor gradually expands over the unimportant area). However, because of the given sequence in which the events are processed one by one, the animation duration is short. As a result, the map users experience many small shock changes. We try to produce smoother map transition by paralleling some events. We define a *step* as a set of merging events happening at the same animation duration. A step is completely processed before the next step takes place (all sequential). Because the events in a step happen parallelly, the animation duration of each event (and also the step) now is the duration sum of all the events happening sequentially. This paper shows the details of finding and processing parallel events. Our original contribution is the proposal of the parallel merging maintaining map consistency over scale transitions.

KEYWORDS
Space-scale cube, map generalization, vario-scale map, continuous generalization

---

CONTACT Dongliang Peng. Email: d.l.peng@tudelft.nl

1. Introduction

When users are reading a digital map, they expect different levels of detail (LoDs) depending on the scale. For example, they may want to see individual buildings when zooming in and see built-up areas when zooming out. That is why geographical information is dependent on the scale (Müller *et al.* 1995, Weibel 1997). In order to prepare map data for different scales, a detailed map is generalized to generate coarser data for maps at smaller scales, which is known as map generalization. Mackaness *et al.* (2016) gave a taxonomy of generalization algorithms, including selection, simplification, aggregation, and so on. Often, a multi-representation database (MRDB) is utilized to store maps at different scales and to send proper data to clients on request (e.g., Hampe *et al.* 2004). However, large and discrete changes between different map representations may confuse users, so continuous map generalization (CMG) is needed to provide smooth scale transition. Algorithms of CMG have been proposed to morph raster maps (e.g., Pantazis *et al.* 2009b,a), to morph polylines (e.g., Nöllenburg *et al.* 2008, Peng *et al.* 2013, Deng and Peng 2015, Li *et al.* 2017a), to generalize buildings (e.g., Li *et al.* 2017b, Peng and Touya 2017, Touya and Dumont 2017), to transform road networks (e.g., Šuba *et al.* 2016, Chimani *et al.* 2014), and to transit administrative boundaries (e.g., Peng *et al.* 2016).

Area objects are important features on maps. When users zoom out, some area objects become too tiny to be seen, which result in visual clutter. The clutter can be avoided by merging those tiny and relatively unimportant areas with their neighbors. For example, Haunert (2009) developed a method based on mixed-integer programming to merge area objects for a map at a certain scale. However, if zooming is realized by switching between some levels of map representations, large and discrete changes usually happen. This kind of changes may cause users to lose track of their interested area objects (van Kreveld 2001). For example, there are large and discrete changes when Figure 1a is replaced with Figure 1b. Before this replacing, Figure 1a remains (e.g., Figure 1p is the same as Figure 1a).[1] Because of the large and discrete changes, users may need a while to realize that the pink area is merged into the light-blue area. This problem can be mitigated if latter smoothly expand over the former (see Figure 1q), where users can immediately understand what happened to the pink area. To provide scale transition of small changes, van Oosterom (2005) proposed the topological Generalized Area Partitioning (tGAP) tree, where in each step the least important area is merged into its most compatible neighbor (see Figures 1d–j). Further, van Oosterom and Meijers (2014) proposed smooth tGAP to gradually apply generalization operations (such as merge, split, and simplify). Based on the smooth tGAP, a space-scale cube (SSC) is built so that the scale transition of a map can be realized smoothly by slicing the SSC (see Meijers *et al.* 2020). For example, Figure 2a is the SSC for the smooth zooming of Figures 1d–j. If slicing the SSC by a horizontal plane at $z = 250$, then we obtain Figure 1q, which is a smooth merging going on.[2] This gradual strategy allows users to better keep their context during zooming so that they do not need to re-orientate (Nöllenburg *et al.* 2008).

When users zoom on maps, they do not want to wait for too long to see the map at the desired scale. On a real map, however, many changes must be processed to arrive at the desired scale. If the changes are processed one by one (e.g., Figures 1d–j), then the animation duration for each change has to be very short. Users may still lose their

[1] See the map at https://congengis.github.io/webmaps/2020/05/merge/example-discrete-merging/.
[2] See the map at https://congengis.github.io/webmaps/2020/05/merge/example-single-merging/.

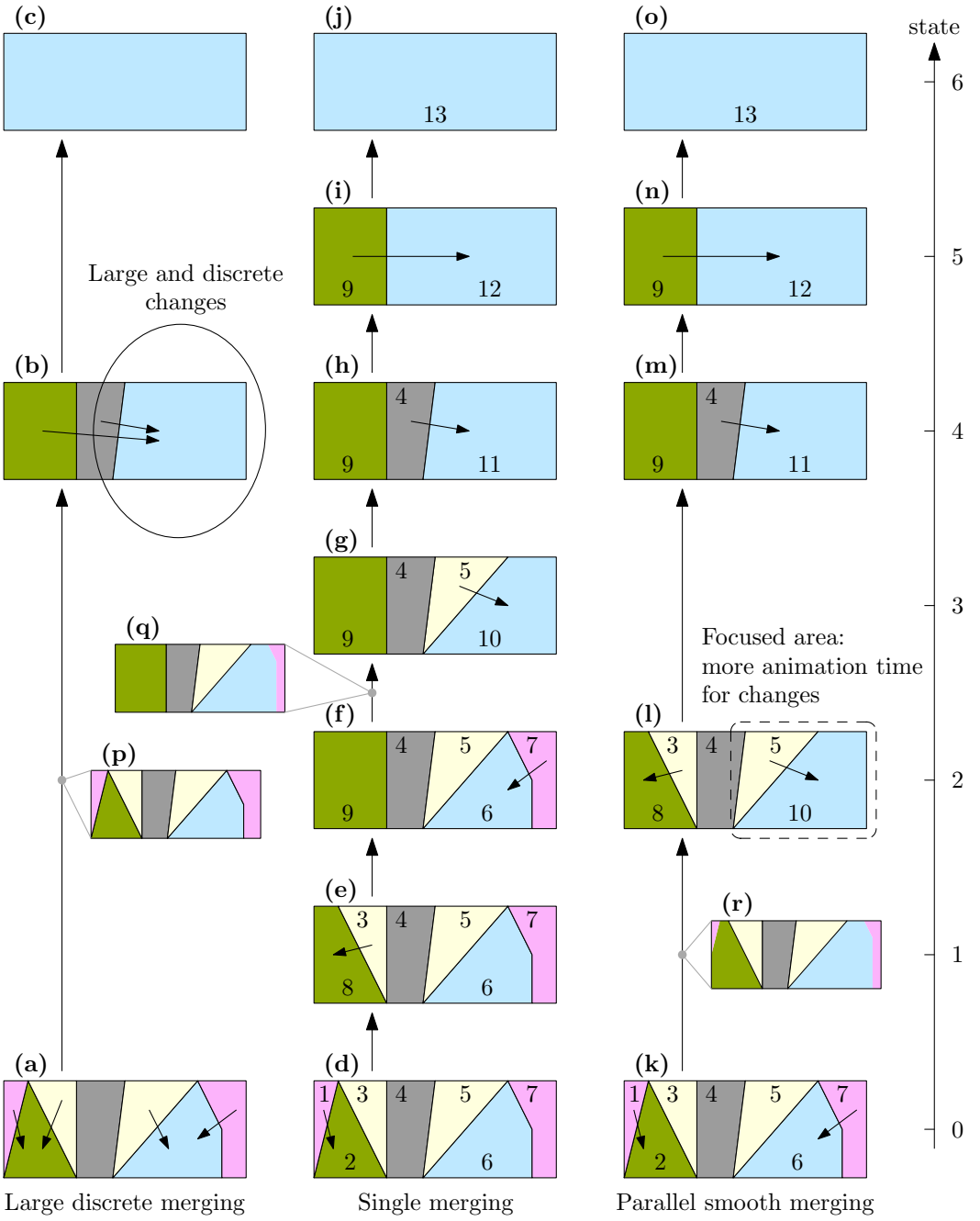
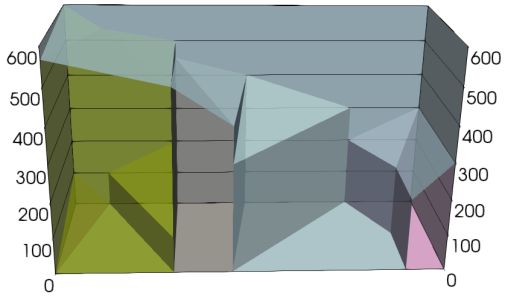
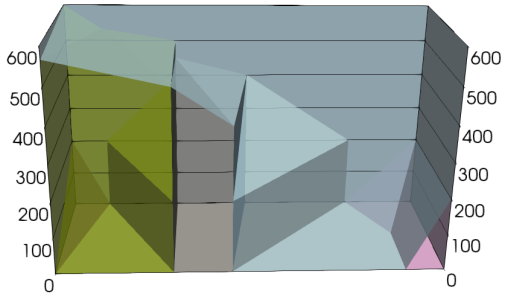


Figure 1.: A comparison of different scale-transition strategies for zooming out. Each arrow inside the subfigures indicates that an area will be merged into another one. The arrow in the right-hand side indicates the states of zooming out, which is related to the transition duration. (a–c): All changes are processed in one go. (d–j): All changes are sequenced one by one rapidly. (k–o): Changes are grouped, resulting in more animation duration for every change. The ellipse shows an example place where large and discrete change happens. The dashed polygon shows the place where a user focuses; the user has about twice of the duration to perceive the merging of the pink area, i.e., figures (k–l), comparing to figures (f–g). The numbers are ids of the faces.



(a) The SSC of the single merging shown in Figures 1d–j. If slicing the SSC with a horizontal plane at z-coordinates 0, 100, 200, 300, 400, 500, and 600, then we get Figures 1d–j. If slicing it at z-coordinate 250, then we get Figure 1q.



(b) The SSC of the parallel merging shown in Figures 1k–o. If slicing the SSC with a horizontal plane at z-coordinates 0, 200, 400, 500, and 600, then we get Figures 1k–o. If slicing it at z-coordinate 100, then we get Figure 1r.

Figure 2.: The content of each of the SSCs is stored in an OBJ file, and each of the figures is generated by displaying the OBJ file in software ParaView. The z-coordinates are 100 times of the state values in Figure 1. We did this multiplication so that the contents can be better observed; otherwise, the two SSCs will be very short when displayed in ParaView. The faces are displayed with opacity 0.8 so that we can see some faces behind. Smooth animations of zooming out are obtained by slicing the SSCs from bottom to top. Note that both SSCs have exactly same bottom and top content (also with same z-coordinates). In the left SSC, only one merging event is happening at a specific state (z-dimension), while in the right SSC multiple merging events may happen at the same state.

context because of the fast changes. For example, a user may get the impression of a shock change from Figure 1f to Figure 1h (without the intermediate state of Figure 1g) even when the three areas on the right-hand side of Figure 1f are merged smoothly. To provide users with more gradual impression, we parallel the generalization operations. For example, we should split or merge the relatively unimportant areas, and we should simplify the boundaries of the areas. If we select some of the generalization operations and process them in parallel, then each operation has more time to take place than the operations are processed one after another. Moreover, if the selected operations are fairly evenly distributed on the whole map, then there may be only a limited number of operations happening in users' focused region. We assume that it is easier for users to keep track of their interested areas when fewer operations take place longer on the focused region. As for the operations happening outside the focused region, they can be ignored because they are not interesting to map users at the moment. In this paper, we focus on paralleling merging operations.[3] For example, in the focused area, there is only one merging operation from Figure 1l to Figure 1m, whereas there are two merging operations from Figure 1f to Figure 1h; a user will have about twice of the duration to perceive the changes of the former (see duration assignment in Section 3.5).[4]

[3]See the map at https://congengis.github.io/webmaps/2020/05/merge/example-parallel-merging/.

[4]A comparison of the single merging (left) and the parallel merging (right) is at https://congengis.github.io/webmaps/2020/05/merge/example-parallel-merging/comparer.html, where the slider at the left-hand side can be moved to adjust the canvases of the two maps.

This paper is organized as follows. Section 2 reviews some related work. Our methodology is presented in Section 3. We show a case study in Section 4. Finally, Section 5 draws the conclusion and present our future work.

2. Related work

Peng (2019, Chapter 2) tried to find an optimal sequence to merge area objects based on the A* algorithm or an integer linear program. A comparison to a greedy algorithm showed that the A* algorithm improves the quality of the merging sequences in the sense of the class changes and the area compactnesses. Van Oosterom and Meijers (2014) pointed out that sequentially processing generalization operations may result in no change at some locations in a zooming duration, which seems unnatural. Therefore, they suggested paralleling the generalization operations, but no implementation, testing, or assessment of the idea was provided. Thiemann and Sester (2018) proposed a chain of operators to generalize a land-cover map. In the chain of processing area objects, they integrated cleaning, dissolving, splitting, aggregating, reclassifying, and simplifying.

Van Oosterom and Meijers (2014) explained the concept of the space-scale cube (SSC). At the bottom of the SSC is a detailed topographic map, and all the area objects extrude along the z-axis. In the SSC, an area on the map becomes a polyhedron, and the common boundary of two areas is a vertical wall. Whenever a generalization operation happens, the extrusions of the involved areas stop; then, the newly generated areas take the place and start to extrude. On this basis, the map at any scale can be generated by slicing the SSC with a horizontal plane at a corresponding z-coordinate. That is to say, the scale becomes the third dimension of the map in the SSC (e.g., Figure 2). Furthermore, they represented the smooth tGAP in the SSC. A typical example of the smooth generalization operation is that an area merges with another one by gradually expanding over it. In the SSC of the smooth tGAP, the wall starts to tilt when the expansion begins. Based on the SSC, Meijers *et al.* (2020) explained the principles of implementing a web map of area objects. They showed how to request only a part of a large dataset of a vario-scale map. They made chunks of the SSC data so that they were able to send only the chunks relevant to users' interested place. They showed how to efficiently slice the SSC to output a web map at a given scale using the GPU at the client side using WebGL in the browser. In addition to slicing the SSC with a horizontal plane, they could also slice the SSC with a curly surface to have a locally more detailed map or with a tilted surface to have a perspective view. To build an SSC, Šuba *et al.* (2014) proposed three methods to merge a pair of areas in a gradual manner, which are the *Single flat plane*, the *Zipper*, and the *Eater*. Basically, the *winner* area gradually expands over the *loser* area. We will use the *Eater* because it works for all kinds of polygons, while the other two methods have limitations for some special cases. For example, the two other methods do not work for some concave polygons. Huang *et al.* (2016) pointed out that the effort of implementing online maps had been spent mainly on preparing data on the server side. They studied the communication of map data between the server side and the client side. They proposed different strategies of assigning the work of processing map data according to the machine abilities of the clients.

Šuba *et al.* (2016) continuously generalized a planar map of road network. In each step, they process the least important face. Taking into account the local condition of the face (e.g., no compatible neighbor at the same side of the road), they may

take different decisions for the face: put it back with a higher importance, collapse it, or merge it into an adjacent face. In addition to the map generalization, they also made statistics of the number of faces, the area of faces, the number of road faces, the number of road edges, and the number of operations (merge and split) when the map scale is decreasing. These statistics can be good indications for (continuous) map generalization. Huang *et al.* (2017) utilized a matrix to guide both pruning rivers and removing vertices for a river network, where the rows and the columns respectively represent the rivers and the vertices. According to the matrix, they were able to decide which rivers and vertices should remain for a given scale. To that purpose, they proposed a method to compute how many rivers and vertices should be kept according to that given scale.

3. Methodology

In order to provide smooth merging so that map users can easily keep track of their interested areas, we merge by gradually expanding an area over another area (see Figure 1r). This expansion can be realized by slicing the space-scale cube (SSC) shown in Figure 2b. The details of slicing an SSC are illustrated in Meijers *et al.* (2020). The SSCs of Figure 2 was built based on the *Eater* of Šuba *et al.* (2014). Figure 1r shows our solution of gradually expanding an area over the other area. Our strategy of presenting the static boundary (i.e., the exterior one) is that we store the polylines and draw them with width, where the width is computed according to the map scale on the fly. We do not draw the common boundary of a pair of areas that are being merged so that it is easy for map users to identify the pairs of areas in transition.

We define an *event* as a single generalization operation, such as merging an area into a neighbor. For example, Figure 1e is obtained from Figure 1d by processing one merging event. Similarly, Figure 1l is obtained from Figure 1k by processing two merging events. We define a *step* as a set of events happening at the same animation duration. For example, Figure 1e is obtained from Figure 1d by processing a step with one merging event. Figure 1l is obtained from Figure 1k by processing a step with two merging events. In our method, a step is completely processed before the next step takes place (all sequential). We define a *state* as the point when a step starts or finishes. For example, there are seven states in the merging sequence of Figures 1d–j (i.e., states 0, 1, 2, 3, 4, 5, and 6, from bottom to top) and five states in the merging sequence of Figures 1k–o (i.e., states 0, 2, 4, 5, and 6). The value of a state is also the total number of events processed so far.

We require that the area objects involved in different merging events of the same step must not be neighbors, which makes the merging events independent from each other. There are two benefits of this independency. First, it is easy to maintain the topology of the map. When a pair of areas have been merged, we must update the common boundaries with the surrounding adjacent areas. If an adjacent area is involved in another merging event, then it is complicated to update the adjacent area's boundaries for the two merging events. Second, users can keep track of their interested areas more easily than merging several areas into a single one. In order to realize the requirement, we block the neighbors of the areas once we have found an event. We show a greedy algorithm to find the parallel merging events for each step in Section 3.1. Then, we integrate the events into the tGAP databse tables (Section 3.2), followed by integrating the events into the SSC (Section 3.3). In Section 3.4, we show how to snap the zooming

to valid states to avoid half-way merging animation as stopping halfway will result in showing slivers in a static state. In Section 3.5, we define the animation duration of zooming from one state to another state.

3.1. *A greedy algorithm*

In the greedy algorithm, we need to obtain the most compatible neighbor for a given area. There are many ways of defining the most compatible neighbor. For example, Cheng and Li (2006) proposed three choices, i.e., the neighbor has the largest size, shares the longest boundary with the least important area, or has the closest class to the least important area. Peng *et al.* (2017) proposed that the most compatible neighbor should have a close class to the least important area and the combination of the two areas should be compact; they defined the class distance based on a binary tree according to the codes of the classes. We define the importance and the compatibility as the same as van Oosterom (2005), van Putten and van Oosterom (1998). That is, the importance of an area is the multiplication of its size and its class weight. The compatibility value between a pair of areas is the multiplication of the common boundary's length and the class similarity of the two areas. Apendix A shows our implementation of computing the weight values and the class similarities.

Figure 3 shows the flowchart of our greedy algorithm. The process starts with state $s = 0$ and a detailed map of area objects, $|M_0|$. Parallel parameter r_{parallel} specifies the proportion (i.e., percentage, when multiplied by 100) of area objects that we expect to merge parallelly. As a value of percentage, r_{parallel} is in the range from 0% to 100%, which means $r_{\text{parallel}} \in [0, 1]$. Expression $|M_s|$ denotes the number of area objects of the map at state s. If there is more than one area ($|M_s| > 1$), then we start finding merging events. We first compute the number of areas that we expect to merge by

$$n_{\text{target}} = \lceil r_{\text{parallel}} \cdot |M_s| \rceil, \tag{1}$$

where the ceiling function guarantees $n_{\text{target}} \geq 1$. That is to say, we find at least one event for each step. When $n_{\text{target}} > 1$, however, we cannot always find n_{target} events because some areas may be blocked as explained before (also see Figure 4). Therefore, we use variable n_{event} to represent the number of events that actually happened within the step.

If we have not found n_{target} events ($n_{\text{event}} < n_{\text{target}}$) and there are still free areas, then we go on looking for merging events. We select the least important area a_{least} from the free areas. An area is *free* if it is not involved in an event and is not blocked. We also find a_{least}'s most compatible neighbor a_{nbr}. If area a_{nbr} is also free, we define an event of areas a_{least} and a_{nbr}. Then, we increase the number of events, n_{event}, by 1. We block the surrounding neighbors of a_{least} and a_{nbr} (see Figure 4a). If area a_{nbr} is not free, then it must be blocked because of the previously found events. In this case, we block a_{least} for now so that areas a_{least} and a_{nbr} may merge in the next step. Then, we continue to find more merging events and to block more areas (see Figure 4b).

If we have found n_{target} events or there is no free area anymore, then finding merging events of the step finishes. We parallelly merge all the pairs of areas of the found events to generate new areas, free all the blocked areas, increase state s by value n_{event}, and create map M_s based on the new areas and the freed areas. Then, finding merging events for the next step starts. This iteration of finding completes until there is only one area left on the map ($|M_s| = 1$). The merging events will be stored as records in

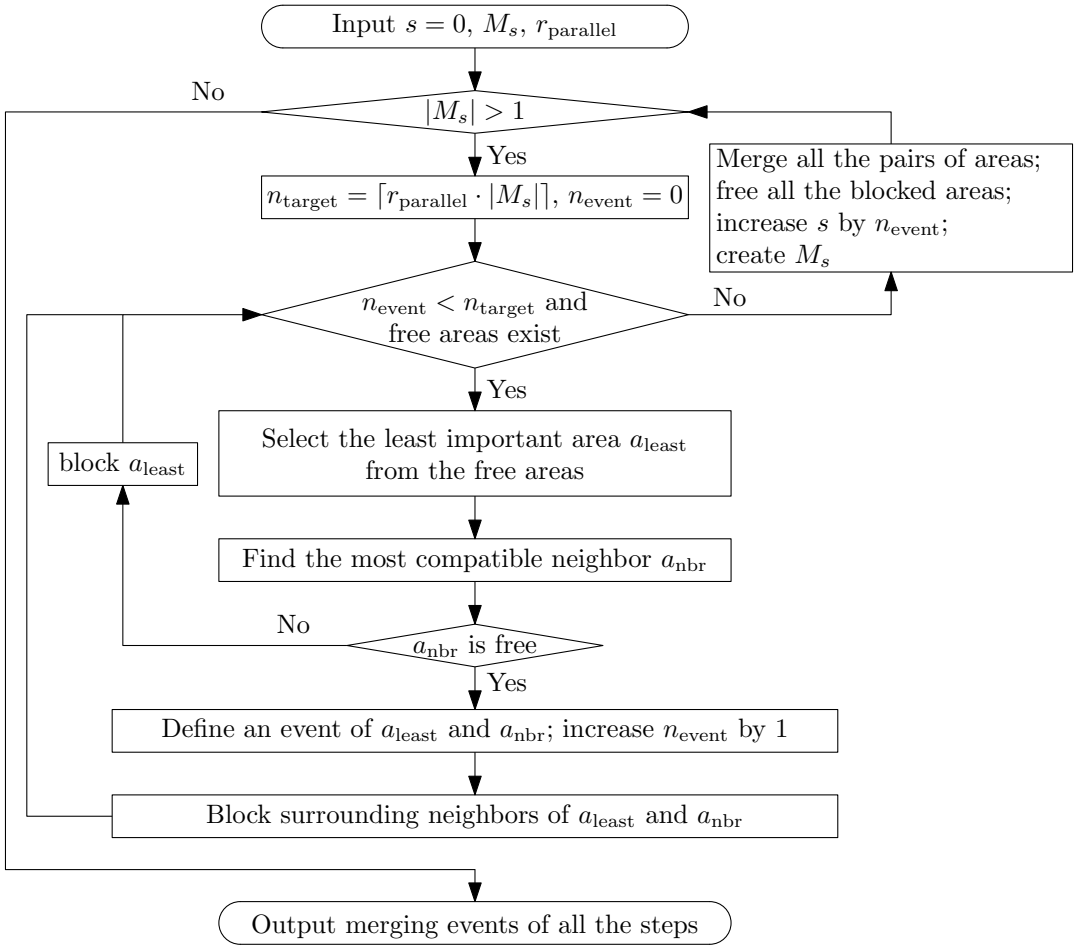


Figure 3.: The flowchart of our greedy algorithm.

Figure 4.: The process of finding parallel merging events for a step. (a) From all the free areas, the least important one is selected to merge into its most compatible neighbor. The arrow indicates the merging. Then the surrounding areas are blocked (marked by the crosses). Note that the area shares only a vertex with the least important area is not blocked. (b) Next, the least important area from the remaining free areas is selected to merge with the most compatible neighbor, and the surrounding areas are also blocked.

tGAP database tables (see Figure 5). Figures 1k–o show a sequence of four merging steps obtained by our greedy algorithm, where parallel parameter r_{parallel} is set to 0.3 (Note that this is an extremely high value, just used to explain the principle in an artificial simple example).

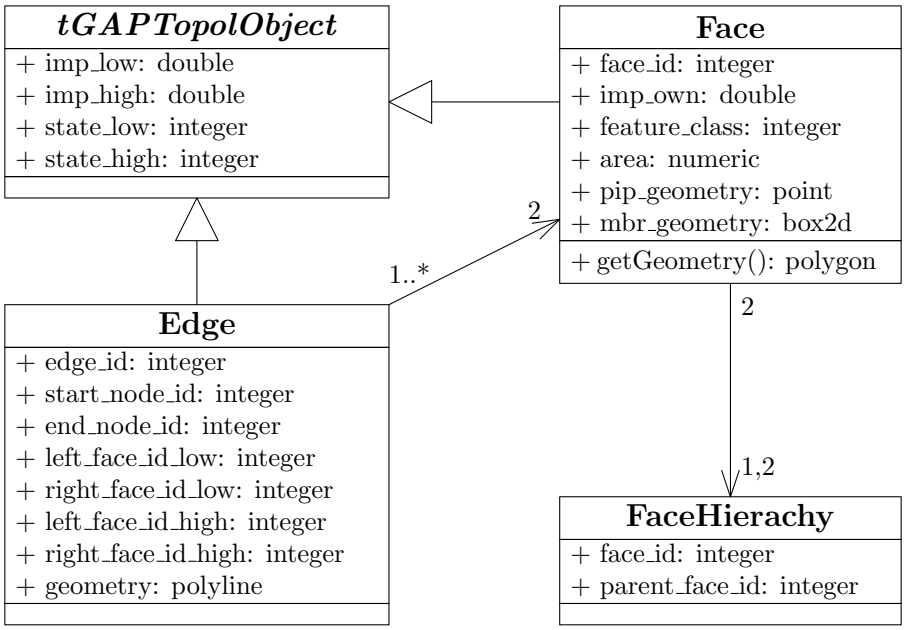


Figure 5.: The UML diagram of the classes stored in tGAP databse tables. This diagram is a slightly improved version of Meijers (2011b, p. 159). In the face table, property *pip_geometry* stores a point (usually the center) in the face (polygon). The geometry of a face can be obtained by calling function *getGeometry()*. We do not store the face geometry because we want to avoid redundancy, as the edges already stores the sequences of points.

3.2. *Integrating the parallel events into the tGAP databse tables*

Meijers (2011b, p. 159) designed three tables to record the information of faces, edges, and face hierarchies, which together form a tGAP. For example, the face table contains columns *face_id*, *imp_low*, *imp_high*, *imp_own*, *feature_class*, *area*, and *mbr_geometry*. We add columns *state_low* (s_{low}) and *state_high* (s_{high}) into the table so that it is easy to see when a face should appear or disappear (see Tables 1a and 1b, where all other columns, except face_id, are hidden). For zooming out, a face should appear, from the merging of two faces, when the slicing arrives at the low state. When the slicing arrives at the high state, the face should have been merged with another area. Comparing between the tables of single merging (Figures 1d–j) and parallel merging (Figures 1k–o), we observed some differences of the values. For example, the s_{high} values of faces 1 and 2 are changed from 1 to 2 (see Table 1). Also, the s_{low} value of face 8 is changed from 1 to 2 (see Table 1). Similarly to the face tables, the columns and records of both the edge table and the face-hierarchy table will be changed accordingly.

3.3. *Integrating the parallel events into the SSC*

Recall that we merge a pair of areas by expanding over the less important area from the neighbor. The Eater of Šuba *et al.* (2014) is used to triangulate the less important area and to tilt the triangles. Then the tilted triangles are integrated into the SSC (see Figure 2) so that we can slice the SSC to achieve smooth merging. For the case of single merging, if a pair of areas have state-high value s_{high}, then the merging animation always starts at state $s_{\text{merge}} = s_{\text{high}} - 1$ (see Table 1a). The less important

Table 1.: Some columns of the face tables. Columns s_{low}, s_{merge}, and s_{high} show the states when the faces appear, when the faces start to disappear, and when the faces completely disappear. In table (b), the different values from table (a) are underlined. Column s_{merge} is not really stored in the database. We show the column so that it is easy to see the differences between the s_{low} values and the s_{merge} values.

(a) The face table of the single merging shown in Figures 1d–j.

f_{id}	s_{low}	s_{merge}	s_{high}
1	0	0	1
2	0	0	1
3	0	1	2
4	0	4	5
5	0	3	4
6	0	2	3
7	0	2	3
8	1	1	2
9	2	5	6
10	3	3	4
11	4	4	5
12	5	5	6
13	6	—	—

(b) The face table of the parallel merging shown in Figures 1k–o.

f_{id}	s_{low}	s_{merge}	s_{high}
1	0	0	$\underline{2}$
2	0	0	$\underline{2}$
3	0	$\underline{2}$	$\underline{4}$
4	0	4	5
5	0	$\underline{2}$	4
6	0	$\underline{0}$	$\underline{2}$
7	0	$\underline{0}$	$\underline{2}$
8	$\underline{2}$	$\underline{2}$	$\underline{4}$
9	2	5	$\underline{6}$
10	$\underline{4}$	$\underline{2}$	$\underline{4}$
11	4	4	5
12	5	5	6
13	6	—	—

area completely disappears at state s_{high}. In the face table, a row will be added to record the new area, and its s_{low} value will be the previous s_{high} value. The new area takes the combined place of the pair of areas. Take 1 as an example, area 1 is merged into area 2 (Figures 1d), and area 8 is generated to take the combined place (Figures 1e). The tilted triangle is the one spans from $z = 0$ to $z = 100$ (i.e., from state 0 to state 1) in Figure 2a. In Table 1a, the s_{low} value of area 8 is 1, which is the s_{high} values of areas 1 and 2.

For the case of parallel merging, if a step consists of n_{event} events and the step finishes at state s_{high}, then the step starts at state $s_{\text{merge}} = s_{\text{high}} - n_{\text{event}}$. The reason is that if the n_{event} events would happen sequentially (i.e., single merging), then they would take place from state $s_{\text{high}} - n_{\text{event}}$ to state s_{high}. When all the events take place parallelly in the same step, each of the events can share its merging duration. As a result, each of the parallel events has more time to take place than the events would happen sequentially. In other words, for a merging step, each of the events has more time to take place if there are more parallel events.

In order to build the SSC for parallel merging, we need the s_{merge} value for each of the merging steps so that we know from which state the triangles of the less important areas should be tilted. A simple way is to add a column, say, s_{merge} into the face table during generating the tGAP, as done in Table 1. Then, the states of starting merging can be recorded into the column. However, we would like to avoid unnecessary columns to save storage. Therefore, we compute s_{merge} values based on the s_{high} values on the fly when building the SSC. As an event involves two areas, the number of events

finishing at state s_{high} can be calculated by

$$n_{\text{event}}(s_{\text{high}}) = \frac{\sum_{s \in S_{\text{high}}} [s = s_{\text{high}}]}{2},$$

where notation S_{high} denotes the set of values recorded in column s_{high} of the face table (e.g., Table 1b). Expression $[s = s_{\text{high}}]$ returns 1 if the two values are equal and returns 0 otherwise. As illustrated before, the state at which the parallel merging starts can be computed by

$$s_{\text{merge}}(s_{\text{high}}) = s_{\text{high}} - n_{\text{event}}(s_{\text{high}}).$$

Take the case of Table 1b for example, we have $S_{\text{high}} = \{2, 2, 4, 5, 4, 2, 2, 4, 6, 4, 5, 6\}$, $n_{\text{event}}(4) = 2$, and $s_{\text{merge}}(4) = 2$. Therefore, there are two merging events finishing at state 4, i.e., event of merging area 3 into area 8 and event of merging area 5 into area 10 (also see Figures 1l and 1m). The merging animation takes place from state 2 to state 4. This merging can be also observed from the two tilted triangles spanning from $z = 200$ to $z = 400$ in Figure 2b. In merging sequence of Figures 1d–j, the animation of merging area 3 into area 8 takes place from state 1 to state 2 (also see the tilted triangle spanning from $z = 100$ to $z = 200$ in Figure 2a), and the animation of merging area 5 into area 10 takes place from state 3 to state 4 (also see the tilted triangle spanning from $z = 300$ to $z = 400$ in Figure 2a). As a result, the animation duration of merging area 3 into area 8 of sequence Figures 1k–o is almost twice as that of sequence Figures 1d–j. We say *almost* because the animation duration is also dependent on the current state of the map (see Section 3.5).

3.4. *Snapping to a valid state*

For a zooming action based on the SSC, we always snap the map to a valid state. In this way, users will not see a merging operation stops half-way and will not see transition artifact (such as slivers). Take the sequence of Figure 1k–o for example, the merging animation should stop at either 1k or 1l, but not at 1r. Note that some states are not valid because of the parallel events. For example, state 1 is not valid for the sequence of Figure 1k–o. In that example, the list of valid states is $S_{\text{valid}} = [0, 2, 4, 5, 6]$. In order to snap to one of the valid states after a zooming operation, we have to communicate them to the client side. There are multiple options. The most simple one assumes that, during the creation of the parallel SSC, we can always perform the n_{target} number of events in all steps. In that case, we just have to communicate the number of areas and the ratio r_{parallel}. In case of high value ratios (e.g., $r_{\text{parallel}} > 0.01$), this assumption may be incorrect. We then have to communicate the valid states by sending them explicitly. Because this list may get rather large, we only send exceptions. Apendix B shows the details of the technique. As a result, the list of valid states S_{valid} is sent to the client side.

According to how much a user has zoomed, the target scale, say, $1 : S_{\text{t}}$ can be computed. Huang *et al.* (2016) suggested that the average density of the original map should be preserved for a smaller-scale map. Their suggestion is based on the assumption that the area density of the base map is well designed, which is reasonable. We use variable A_{real} to denote the total areal size of all the area objects in reality. Then, the size on screen at scale $1 : S_{\text{t}}$ is $A_{\text{real}}/S_{\text{t}}^2$. In order to keep the density, we

require

$$\frac{N_\mathrm{b}}{A_\mathrm{real}/S_\mathrm{b}^2} = \frac{N_\mathrm{b} - E_\mathrm{t}}{A_\mathrm{real}/S_\mathrm{t}^2}, \tag{2}$$

where parameter $N_\mathrm{b} = |M_0|$ is the number of areas on the base map, parameter S_b is the scale denominator of the base map, and variable E_t is the total number of events happening from the base map to the map at scale $1 : S_\mathrm{t}$. Equation 2 yields

$$E_\mathrm{t} = N_\mathrm{b}\left(1 - \frac{S_\mathrm{b}^2}{S_\mathrm{t}^2}\right), \tag{3}$$

In our example regarding to list of valid states $\mathrm{S}_\mathrm{valid}$, if event number $E_\mathrm{t} \leq 0$, the base map should be presented; if $E_\mathrm{t} \geq 6$ (i.e., the last value of list L_event), the map with the final single area should be presented. Otherwise, if $0 < E_\mathrm{t} < 6$, we snap event number E_t to the closest value (measured in events) of list S_valid, which is denoted by $E_\mathrm{t,snap}$. The scale denominator corresponding to event number $E_\mathrm{t,snap}$ can be computed by

$$S_\mathrm{t,snap} = S_\mathrm{b}\sqrt{\frac{N_\mathrm{b}}{N_\mathrm{b} - E_\mathrm{t,snap}}}. \tag{4}$$

where this equation is an inverse function of Equation 3. At the end of the zooming action, the map will snap to state $s_\mathrm{t,snap}$ at scale $1 : S_\mathrm{t,snap}$. Note that state $s_\mathrm{t,snap}$ always has the same value as event number $E_\mathrm{t,snap}$.

3.5. *Animation duration of a step*

When users are zooming from a scale to another scale, some steps take place to change the map from a state to another one. We define the *zooming duration* as the amount of animation time that the map reacts to one "click" of the mouse wheel. The zooming duration often is the sum of the *animation durations* of several merging steps. The animation duration of each event depends on the number of events between the two states, the *zooming factor* of the scale, and the *zooming duration*. On the one hand, the animation duration should not be too short as then the animation will be too fast. On the other hand, if the animation takes too long, the map will not be interactive, and users will be "frustrated". Meijers *et al.* (2020, Section 4.3) have introduced the zooming factor and the zooming duration. They allowed users to set the two parameters, which is also the case in this paper (see Figure 6). This section formalizes the relationship of the animation duration, the zooming duration, the zooming factor, and the number of events. In a zooming duration, there can be many merging steps, no matter single merging or parallel merging. The following calculation is based on the setting that a zooming duration is divided equally by its merging steps (Šuba (2017, Section 6.7) showed some other possible settings). In other words, the steps happen sequentially and take the same amount of animation duration. Note that the steps from different zooming durations may have different animation durations.

Let f_zoom be the zooming factor, and let t_zoom be the zooming duration. Let $1 : S_\mathrm{t,snap}$ be the snapped scale before the zooming operation, and let $1 : S_\mathrm{o}$ be the zoomed out scale (not snapped yet). For zooming out, we define the relationship between the

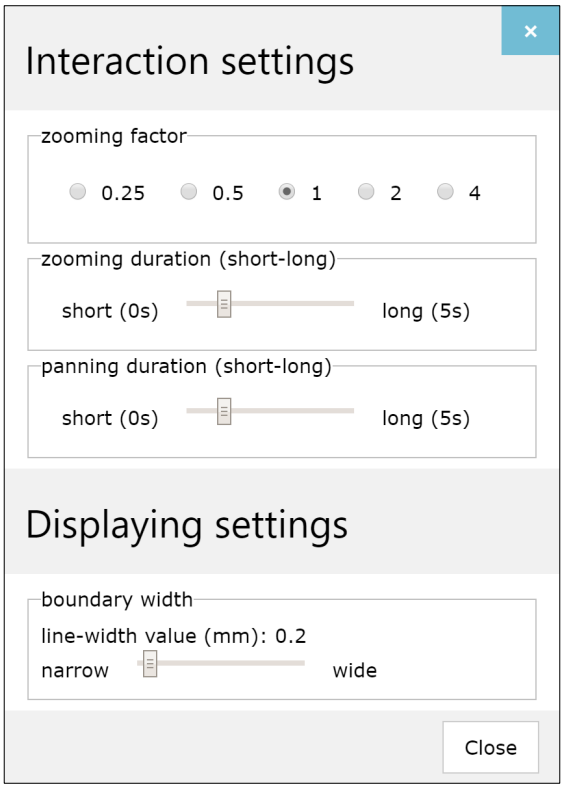


Figure 6.: Our panel of settings. Among others, one can set how much to zoom when scrolling the mouse wheel and set the zooming duration.

two scale denominators as

$$S_{\mathrm{o}} = S_{\mathrm{t,snap}}(1 + f_{\mathrm{zoom}}). \tag{5}$$

For scale $1 : S_{\mathrm{o}}$, we compute the number of events that should be processed from the base map by

$$E_{\mathrm{o}} = N_{\mathrm{b}} \left(1 - \frac{S_{\mathrm{b}}^2}{S_{\mathrm{o}}^2}\right),$$

which is according to Equation 3. As the value of E_{o} may be not an integer, we snap it to a valid state (see Section 3.4), and we have a snapped value $E_{\mathrm{o,snap}}$. Then, we compute scale denominator $S_{\mathrm{o,snap}}$ by Equation 4. According to Equation 3, we have merged $E_{\mathrm{t,snap}}$ areas when arriving at scale $1 : S_{\mathrm{t,snap}}$. The event number of zooming out from scale $1 : S_{\mathrm{t,snap}}$ to scale $1 : S_{\mathrm{o,snap}}$ is

$$N_{\mathrm{event}} = E_{\mathrm{o,snap}} - E_{\mathrm{t,snap}}.$$

Recall that zooming duration t_{zoom} is for zooming from from scale $1 : S_{\mathrm{t,snap}}$ to scale $1 : S_{\mathrm{o}}$. As the map is actually zooming to $1 : S_{\mathrm{o,snap}}$, we adjust the zooming duration to

$$t_{\mathrm{snap}} = t_{\mathrm{zoom}} \frac{N_{\mathrm{event}}}{E_{\mathrm{o}} - E_{\mathrm{t,snap}}}.$$

That is to say, the $E_{\text{o,snap}} - E_{\text{t,snap}}$ events will happen in time duration t_{snap}. If the events happen sequentially (each step consists of a single event), then the animation duration of each event is

$$t_{\text{single}} = \frac{t_{\text{snap}}}{N_{\text{event}}} = \frac{t_{\text{zoom}}}{E_{\text{o}} - E_{\text{t,snap}}}. \tag{6}$$

If we parallel these events, then we will have fewer steps and each event has more time to take place. Let n_{step} be the number of steps in a zooming duration. If we are lucky enough so that expression $r_{\text{parallel}} \cdot |M_s|$ of Equation 1 always returns an integer, then we do not need the ceiling function of Equation 1 (if we are not that lucky, the value of n_{step} will be slightly different). We have

$$N_{\text{t,snap}}(1 - r_{\text{parallel}})^{n_{\text{step}}} = N_{\text{o,snap}},$$

where $N_{\text{t,snap}} = N_{\text{b}} - E_{\text{t,snap}}$ is the number of areas at scale $1 : S_{\text{t,snap}}$, and $N_{\text{o,snap}} = N_{\text{b}} - E_{\text{o,snap}}$ is the number of areas at scale $1 : S_{\text{o,snap}}$. Then, the number of steps can be computed by

$$n_{\text{step}} = \log_{1-r_{\text{parallel}}} \frac{N_{\text{o,snap}}}{N_{\text{t,snap}}}.$$

Because the steps happen sequentially, each of the steps in the zooming duration has animation duration

$$t_{\text{parallel}} = \frac{t_{\text{snap}}}{n_{\text{step}}}, \tag{7}$$

which is also the animation duration of each of the parallel events. Putting Equations 6 and 7 together, we have

$$t_{\text{parallel}} = t_{\text{single}} \frac{N_{\text{event}}}{n_{\text{step}}}.$$

As N_{event} is larger than or equal to n_{step}, t_{parallel} is also larger than or equal to t_{single}.

When we zoom in back from scale $S_{\text{o,snap}}$ to scale S_{t}, we have

$$S_{\text{t}} = \frac{S_{\text{o,snap}}}{(1 + f_{\text{zoom}})},$$

which is the inverse function of Equation 5. We will be able to snap to scale $1 : S_{\text{t,snap}}$. We will use the same animation duration and process the same number of events and steps as we zoomed out. The difference from zooming out is that, instead of merging, areas will bubble up.

4. Case study

We have stored the result of the tGAP as a set of tables (see Section 3.2) in a PostgreSQL database. We have employed the Eater of Šuba *et al.* (2014), implemented in Python, to generate the elements (vertices, triangulated faces, and boundaries) of the

SSC (van Oosterom *et al.* 2014) and saved these elements in an OBJ file.[5] When a user visits our website to access the map, some data will be sent to the client side. On the client side, the received data will be processed by a map viewer implemented in JavaScript. The processed data and some code based on WebGL (Web Graphics Library) are submitted to GPU so that we can output the interactive map with smooth zooming by slicing the SSC.

Figures 7 shows the topographical map of this case study. The class codes and the rendering formulas are provided by Geonovum.[6] Because the base scale is 1 : 10,000, we have $S_b = 10{,}000$ for Equation 4. The maximum value of event number E_{snap} is 13,237 as there are in total 13,238 areas. When we zoom out far enough so that E_{snap} reaches its maximum value, the scale denominator will arrive at 1,150,565.1 according to Equation 4. At that moment, all the areas are merged into one single area. In each step, we want to parallelly merge some proportion of the areas. We tried three cases: 0.1%, 1%, and 10%. That is, parallel parameter $r_{parallel} = 0.001, 0.01,$ and 0.1 (see Section 3.1), which are independent of the size of the map dataset. The three versions of map can be browsed online.[7] Figure 8 shows two examples of our web map when parallel parameter $r_{parallel} = 0.01$.

Some statistics of the results when parallel parameter $r_{parallel} = 0.001$, 0.01, or 0.1 are shown in Table 2. According to column N_{step}, the number of steps decreases when the parallel parameter increases. This is reasonable because more areas will be merged in each step. As explained in Section 3.1, for each merging step we iteratively select the least important area and its most compatible neighbor to define a merging event; then, we block the neighbors of the two areas. Sometimes, a least important area is already blocked because of the previously found events. This situation happens 2,714 times in total for all the steps when parallel parameter $r_{parallel} = 0.01$ (see column $N_{blocked}$ of Table 2). Sometimes, although a least important area is free, its most compatible neighbor has been blocked because of the previously found events. This case happens 1,383 times in total for all the steps when parallel parameter $r_{parallel} = 0.01$ (see column $n_{nbr\_blocked}$ of Table 2). According to the statistics, we encounter more cases of the areas blocked when merging a larger proportion of the area objects. However, we can still reach our target number of events perfectly for settings of $r_{parallel} = 0.01$ or $r_{parallel} = 0.001$. Only when we push beyond the limit (e.g., $r_{parallel} = 0.1$), we can not reach the target number of events in a step (and we need correction information to compute the actual number of achieved events). When the target number of events cannot be met, one could also question the cartographic quality if there is hardly any free choice when generalizing.

As we can find the target numbers of merging events for all the steps when parallel parameter $r_{parallel} = 0.001$ or 0.01, the corresponding exceptions lists are empty. When $r_{parallel} = 0.1$, the exception list is $[[1, 1304], [2, 1070], \ldots, [77, 2]]$, which has 71 pairs of values. In reality, we would not use $r_{parallel} = 0.1$ (merging 10% of the current areas in every step) because it is an unrealistic high value. Using such a high value results in a multi-scale representation (because we have a few valid states or scales), whereas we would like to have representations at nearly arbitrary scales.

[5] Wavefront .obj file: https://en.wikipedia.org/wiki/Wavefront_.obj_file, accessed: Jan 14, 2020.

[6] See the details at http://register.geostandaarden.nl/visualisatie/top10nl/1.2.0/BRT_TOP10NL_1.2_beschrijving_visualisatie.xlsx, accessed: Jan 15, 2020.

[7] All of our web maps can be found at https://congengis.github.io/webmaps/2020/05/merge/.

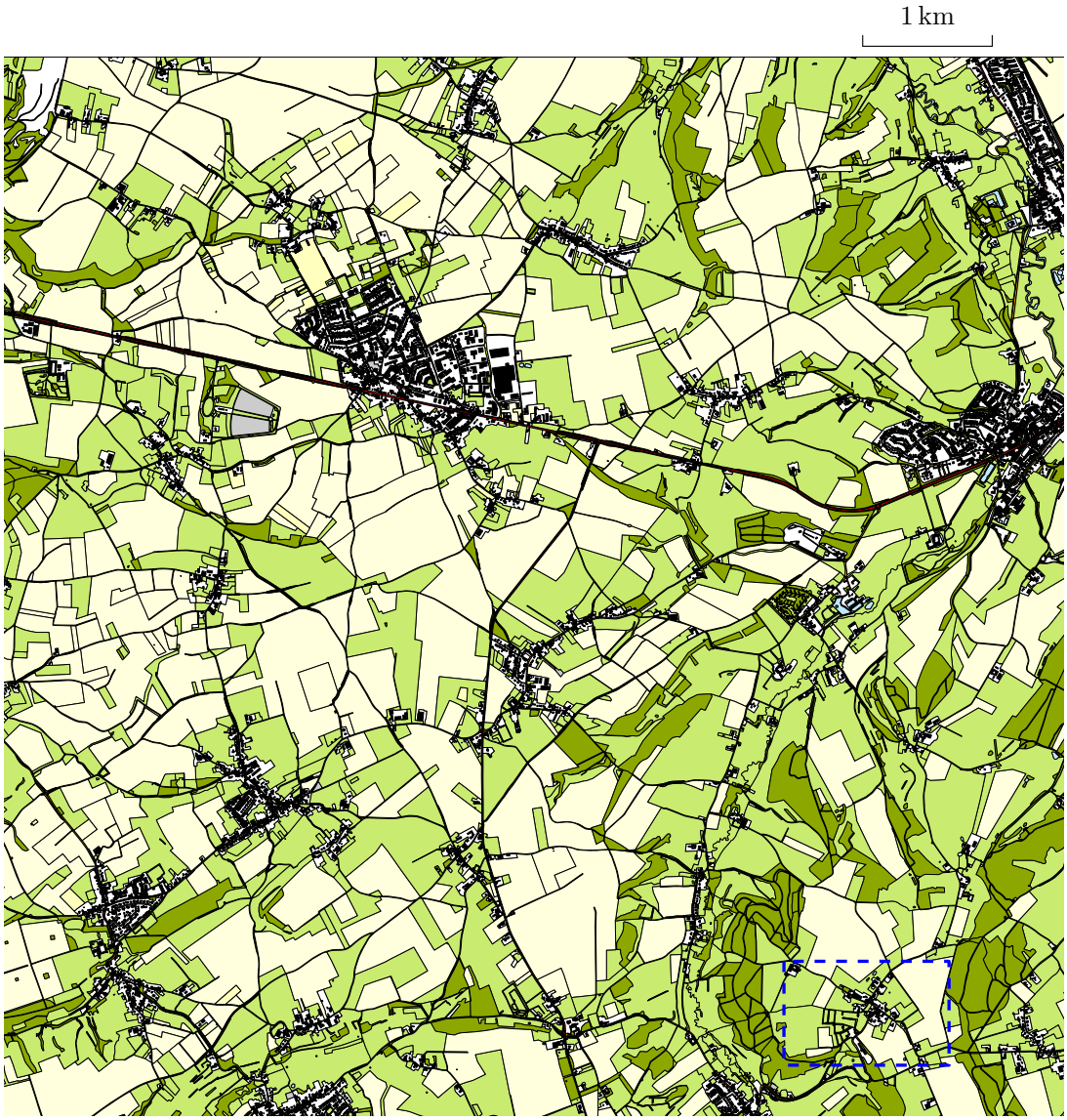
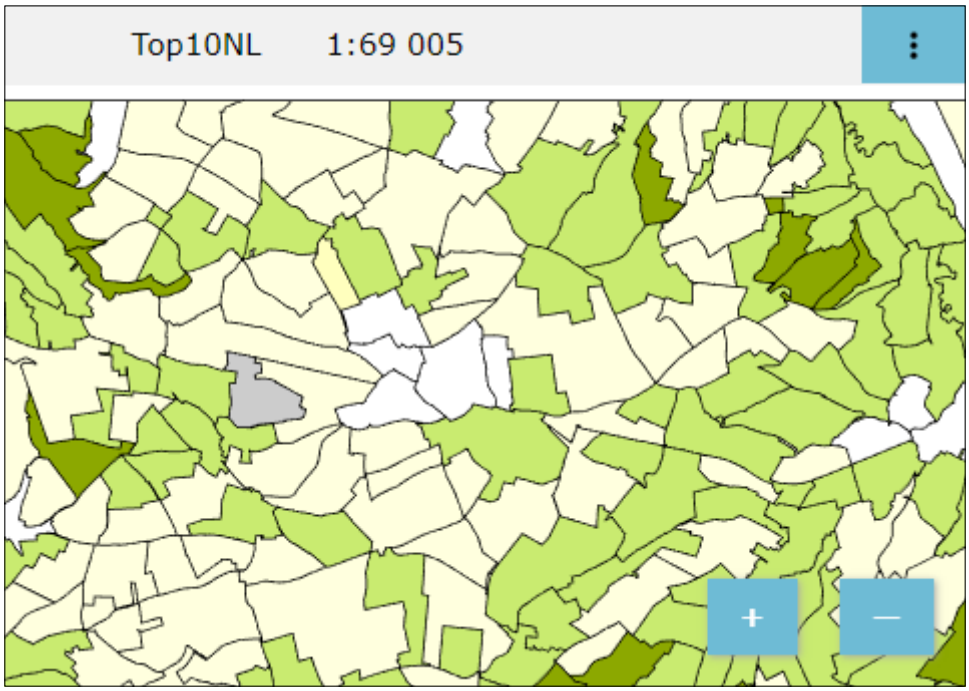


Figure 7.: The topographic map represents the place in the south of Limburg, The Netherlands. There are 13,238 parcels. The map is for scale 1 : 10,000.

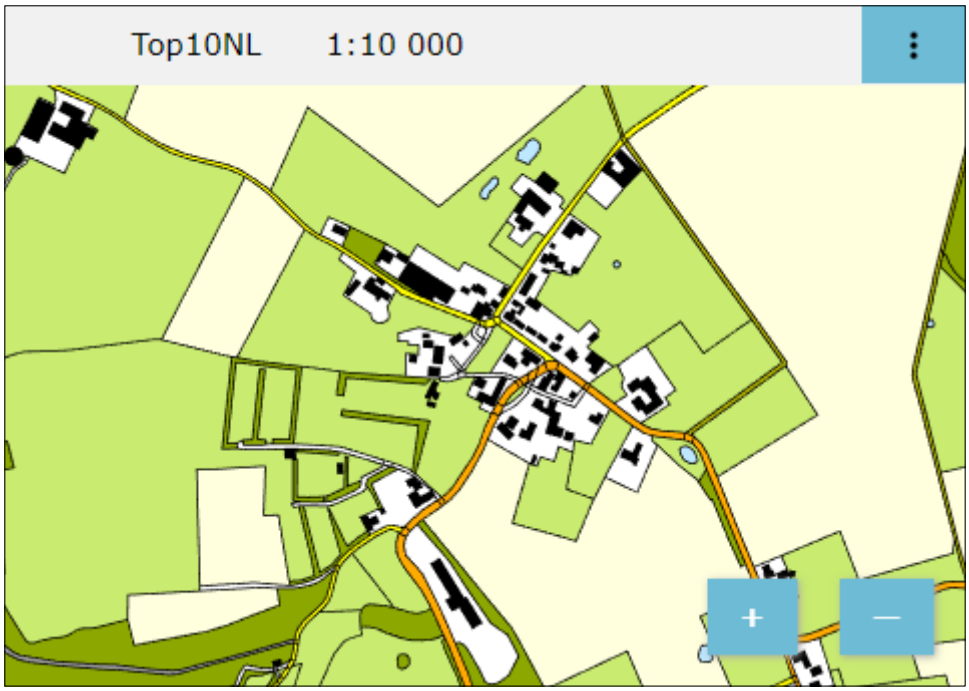
We set zooming factor $f_{\text{zoom}} = 1$ and zooming duration $t_{\text{zoom}} = 1s$ (see Section 3.5). When zooming on our web maps with different parallel parameters, we observed that the impressions of the maps based on single merging[8] and based on parallel merging with parameter $r_{\text{parallel}} = 0.001$ are almost the same. The reason is that the smooth merging happens too fast, and we cannot really see the merging animation. We get the feeling of smooth merging when $r_{\text{parallel}} = 0.01$. When $r_{\text{parallel}} = 0.1$, the smooth merging is already obvious. In order to show a better comparison of single merging and parallel merging, we put two maps together (see Figure 9), where the parallel parameter is 0.1.[9]

[8]See the web map at https://congengis.github.io/webmaps/2020/05/merge/single-merging/.

[9]See the map of comparison at https://congengis.github.io/webmaps/2020/05/merge/0.1/comparer.html.



(a) An overview map. The map is generated from the base map by parallel merging with parameter $r_{\text{parallel}} = 0.01$.



(b) A part of the base map. The displayed place is marked by the dashed rectangle in Figure 7.

Figure 8.: Two examples of our web map with different scales.

When parallel parameter r_{parallel} is set to 0.1, we noticed a problem in the north-eastern corner of the map. That is, some tiny and relatively unimportant areas stay until the scale is very small, while they should be merged much earlier. This is due to

Table 2.: Some statistics when different parallel parameters area used (i.e., $r_{\text{parallel}} = 0.001$, 0.01, and 0.1). Column N_{step} records the number of steps to transit from the base map to the map with a single area. Column N_{blocked} records the number of times when the least important area was blocked. Column $N_{\text{nbr\_blocked}}$ records the number of times when the most compatible neighbor was blocked.

r_{parallel}	N_{step}	N_{blocked}	$N_{\text{nbr\_blocked}}$
0.001	3,195	211	72
0.01	544	2,714	1,383
0.1	91	100,617	34,268

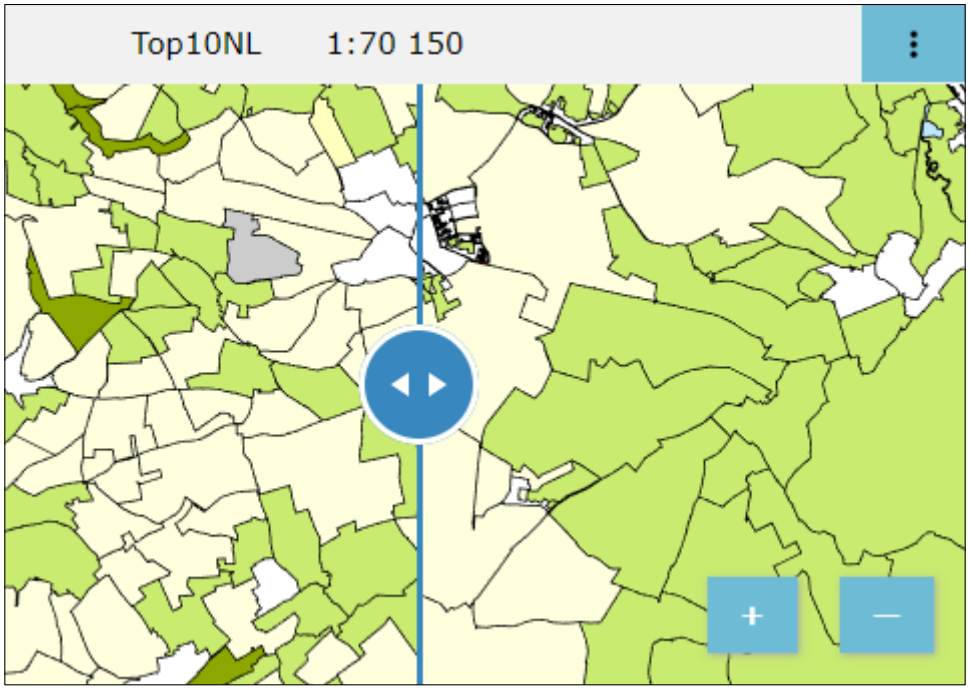


Figure 9.: The map to the left of the slider is based on single merging, and the one to the right is based on parallel merging with $r_{\text{parallel}} = 0.1$. The slider can be moved to tune the widths of the two map canvases. The two maps are very different, and we can observe some sudden changes across the slider.

the fact that there are many buildings in the northeastern corner of the map. When the buildings share the same surrounding area, they become holes of the polygon. In each step, only one of the buildings will be merged into the surrounding area because an area is allowed to merge with only one area in each step. In the meantime, the areas at other places of the map merge relatively fast because we allow 10% of the areas to be merged in each step. Fortunately, we would not need to use such a big parallel parameter in reality.

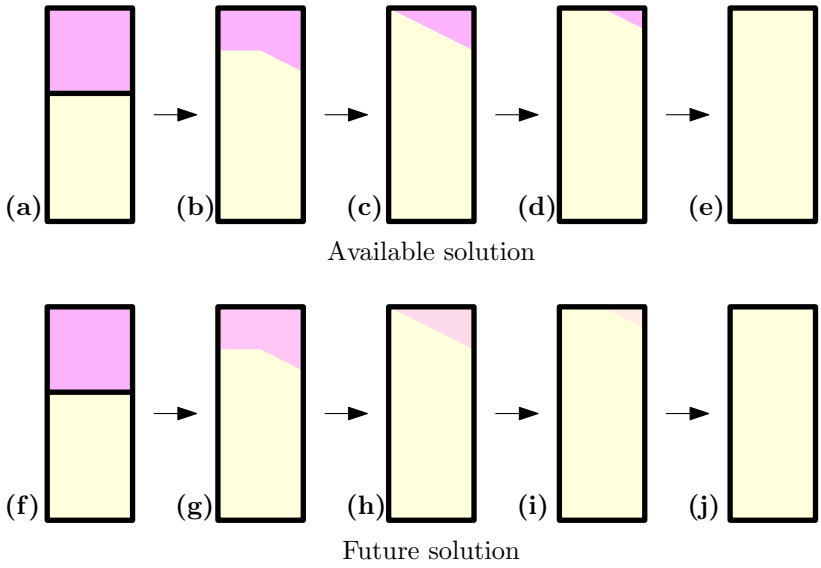


Figure 10.: Our available solution of merging animation and a possible future solution.

5. Concluding remarks

5.1. *Conclusion*

This paper investigates on paralleling generalization operations, using the merging operation as a case study. The purpose of having parallel generalization operations is to have smoother zooming experience later on (compared to the pure sequenced individual generalization events) so that users can better keep their context during zooming. This paper develops a greedy algorithm to find parallel events of merging area objects. Then, the parallel events are integrated into the tGAP and the SSC to nicely visualize the merging animations. To guarantee that the merging animations always complete for zooming, we managed to snap zooming operations to valid states. This paper also presents a recipe to define the animation duration of an event. According to our case study, the events of parallel merging can be better observed than the events of single merging. This result shows the potential that users can keep their context better and can have smoother map interaction experience when zooming on a map based on parallel-event operations.

5.2. *Future work*

Many topics related to this research need to be investigated further. Figure 10a–e shows our solution of gradually expanding an area over the other area. It may be even better if the color of the smaller area (pink) adapts to the color of the larger one (light yellow) smoothly (see Figure 10f–j).

This paper used a greedy algorithm to find parallel merging events for each step. Alternatively, it is possible to define merging steps by selecting and combining some single-event steps of a sequence found by some existing methods (e.g., the greedy algorithm of van Oosterom (2005) or the A* algorithm of Peng (2019, Chapter 2)).

Currently, the merging events distribute randomly on a map. If we are unlucky, there may be a lot of events happening in users' focused region for a zooming duration, which may cause the users to lose track of their interested objects; for another zooming duration, there may be no event happening in the focused region at all. The strategy of blocking neighbor areas in our greedy algorithm already mitigates the problem. However, it may be even better if we explicitly distribute the merging events evenly, then the workload for a user to follow the events is stable during the zooming. To this end, we could divide a map into many regions using a field-tree-like, multiple-level grid (van Putten and van Oosterom 1998) or using the road network. Then, we could find a certain number of events in each of the regions according to the regions' sizes, which should result in an even distribution of events. Finally, we could compare our current greedy algorithm and the algorithm considering even distribution.

Our current event consists of only the merging operation, it is also necessary to involve split operation because sometimes a merging operation results in an unnatural area. For example, it is weird to merge a long and thin area with one of the areas that are along it (see Haunert and Sester 2008). Therefore, such kind of long and thin areas should be split into several parts first. We may integrate a split method based on the straight skeleton (Haunert and Sester 2008) or the skeleton obtained from a constrained Delaunay triangulation (Meijers *et al.* 2016). In addition to area features, we also need to support line features for these long objects (e.g., roads, river, rail) for the smaller scales. In order to apply appropriate generalization operators for a certain scale, we need to extend and implement the framework to guide our generalization choices (Meijers *et al.* 2018).

To avoid clutter of vertices for zooming out, it is necessary to simplify the boundaries of the areas. Many existing methods could be integrated into our parallel paradigm. Meijers (2011a) proposed a method to simplify the boundaries parallelly. Their results are topologically safe. Another choice would be the method of Imai and Iri (1988), which is able to minimize the number of vertices for a given error threshold. One more choice would be to construct compatible triangulations (see Peng 2019, Chapter 3) for the two levels of topographic maps. In the SSC, we could build some tilted walls to connect the two levels of compatible triangulations. When we slice this SSC to animate a zooming action, the boundaries of the areas are morphed (moved smoothly and parallelly) between a detailed representation and a coarse representation. We can imagine that it is challenging to build the tilted walls. Furthermore, we would like to simplify point symbols and text labels in a smooth way. For this purpose, we may need to integrate the ideas of Haunert and Wolff (2017) and Schwartges *et al.* (2013).

We would also like to realize smooth zooming between a foreground map and a background map, which respectively present a thematic data and a topographical data. When a user zooms in, the foreground map should be displayed, and when the user zooms out, the background map should be displayed. Furthermore, we would like to integrate time dimension into our 3-dimensional SSC to form a 4-dimensional space-scale-time structure (a tesseract). Then, we have the opportunity to store all the updates along time in the tesseract. By "slicing" it, we will be able to output a map at any scale and any time point.

This paper develops technique for smooth zooming based on parallel merging, and we hope that it allows map users to follow the zooming more easily. A future work is to investigate how much map users benefit from our technique. We will conduct some usability tests based on the experience of Šuba (2017, Section 6.7) and Midtbø and Nordvik (2007).

Data and codes availability statement

The data and codes used in this case study can be found at *figshare* (see https://figshare.com/s/199e049e2a0f9c18d19a). The data was derived from open-access dataset TOP10NL, which is a topographic map produced by Kadaster (see https://www.pdok.nl/downloads/-/article/basisregistratie-topografie-brt-topnl).

References

Cheng, T. and Li, Z., 2006. Toward quantitative measures for the semantic quality of polygon generalization. *Cartographica*, 41 (2), 487–499. Available from: https://www.utpjournals.press/doi/abs/10.3138/0172-6733-227U-8155.

Chimani, M., van Dijk, T.C., and Haunert, J.H., 2014. How to eat a graph: Computing selection sequences for the continuous generalization of road networks. *In*: *Proc. 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACMGIS)*, Dallas, TX, USA. 243–252. Available from: http://doi.acm.org/10.1145/2666310.2666414.

Deng, M. and Peng, D., 2015. Morphing linear features based on their entire structures. *Transactions in GIS*, 19 (5), 653–677.

Hampe, M., Sester, M., and Harrie, L., 2004. Multiple representation databases to support visualization on mobile devices. *In*: *Proc. 20th ISPRS Congress*. International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, vol. XXXV (B4: IV), 135–140. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.184.3303.

Haunert, J.H., 2009. *Aggregation in map generalization by combinatorial optimization*. phdthesis. Leibniz Universität Hannover, Germany. Available from: https://dgk.badw.de/fileadmin/user_upload/Files/DGK/docs/c-626.pdf.

Haunert, J.H. and Sester, M., 2008. Area collapse and road centerlines based on straight skeletons. *GeoInformatica*, 12 (2), 169–191. Available from: https://link.springer.com/article/10.1007/s10707-007-0028-x.

Haunert, J.H. and Wolff, A., 2017. Beyond maximum independent set: An extended integer programming formulation for point labeling. *ISPRS International Journal of Geo-Information*, 6 (11).

Huang, L., *et al.*, 2017. A matrix-based structure for vario-scale vector representation over a wide range of map scales: The case of river network data. *ISPRS International Journal of Geo-Information*, 6 (7).

Huang, L., *et al.*, 2016. Engineering web maps with gradual content zoom based on streaming vector data. *ISPRS Journal of Photogrammetry and Remote Sensing*, 114, 274–293. Available from: http://www.sciencedirect.com/science/article/pii/S0924271615002646.

Imai, H. and Iri, M., 1988. Polygonal approximations of a curve—formulations and algorithms. *In*: G.T. Toussaint, ed. *Machine intelligence and pattern recognition*. Computational Morphology: A Computational Geometric Approach to the Analysis of Form, vol. 6. Elsevier, 71–86. Available from: https://www.sciencedirect.com/science/article/pii/B9780444704672500114.

Li, J., *et al.*, 2017a. Continuous scale transformations of linear features using simulated annealing-based morphing. *ISPRS International Journal of Geo-Information*, 6 (8). Available from: http://www.mdpi.com/2220-9964/6/8/242.

Li, J., Li, X., and Xie, T., 2017b. Morphing of building footprints using a turning angle function. *ISPRS International Journal of Geo-Information*, 6 (6). Available from: http://www.mdpi.com/2220-9964/6/6/173.

Mackaness, W.A., Burghardt, D., and Duchêne, C., 2016. Map generalization. *In*: *International encyclopedia of geography: People, the earth, environment and technology*. John Wiley & Sons, 1–16. Available from: http://doi.org/cx89.

Meijers, M., *et al.*, 2018. Towards a scale dependent framework for creating variable-scale maps. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLII-4, 425–432. Available from: https://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XLII-4/425/2018/.

Meijers, M., 2011a. Simultaneous & topologically-safe line simplification for a variable-scale planar partition. *In*: S. Geertman, W. Reinhardt and F. Toppen, eds. *Advancing geoinformation science for a changing world*. Berlin, Heidelberg: Springer Berlin Heidelberg, 337–358. Available from: https://doi.org/10.1007/978-3-642-19789-5_17.

Meijers, M., 2011b. *Variable-scale geo-information*. phdthesis. Delft University of Technology. Available from: http://www.gdmc.nl/publications/2011/Variable-scale_Geo-information.pdf.

Meijers, M., Savino, S., and van Oosterom, P., 2016. SplitArea: An algorithm for weighted splitting of faces in the context of a planar partition. *International Journal of Geographical Information Science*, 30 (8), 1522–1551. Available from: https://doi.org/10.1080/13658816.2016.1140770.

Meijers, M., *et al.*, 2020. Web-based dissemination of continuously generalized space-scale cube data for smooth user interaction. *International Journal of Cartography*, 0 (0), 1–25. Available from: https://doi.org/10.1080/23729333.2019.1705144.

Midtbø, T. and Nordvik, T., 2007. Effects of animations in zooming and panning operations on web maps: A web-based experiment. *The Cartographic Journal*, 44 (4), 292–303. Available from: http://doi.org/dgnjmj.

Müller, J.C., *et al.*, 1995. Generalization: State of the art and issues. *In*: J.C. Müller, J.P. Lagrange and R. Weibel, eds. *GIS and generalization: Methodology and practice*. No. 1 in GISDATA. London, UK: Taylor & Francis, Ch. 1, 3–17.

Nöllenburg, M., *et al.*, 2008. Morphing polylines: A step towards continuous generalization. *Computers, Environment and Urban Systems*, 32 (4), 248–260. Available from: https://www.sciencedirect.com/science/article/pii/S0198971508000331.

Pantazis, D., *et al.*, 2009a. Morphing techniques: Towards new methods for raster based cartographic generalization. *In*: *Proc. 24th International Cartographic Conference (ICC)*, November, Santiago, Chile. Available from: https://icaci.org/files/documents/ICC_proceedings/ICC2009/html/refer/19_5.pdf.

Pantazis, D., *et al.*, 2009b. Are the morphing techniques useful for cartographic generalization? *In*: *Urban and regional data management*. CRC Press, 195–204. Available from: http://dx.doi.org/10.1201/9780203869352.ch18.

Peng, D., 2019. *An optimization-based approach for continuous map generalization*. phdthesis. University of Würzburg. Available from: https://opus.bibliothek.uni-wuerzburg.de/frontdoor/index/index/docId/17442.

Peng, D., *et al.*, 2013. Morphing polylines based on least squares adjustment. *In*: *Proc. 16th ICA Workshop on Generalisation and Multiple Representation (ICAGM)*, August. Available from: https://kartographie.geo.tu-dresden.de/downloads/ica-gen/workshop2013/genemappro2013_submission_6.pdf.

Peng, D. and Touya, G., 2017. Continuously generalizing buildings to built-up areas by aggregating and growing. *In*: *Proc. 3rd ACM SIGSPATIAL Workshop on Smart Cities and Urban Analytics (UrbanGIS)*, November, Redondo Beach, CA, USA. ACM.

Peng, D., Wolff, A., and Haunert, J.H., 2016. Continuous generalization of administrative boundaries based on compatible triangulations. *In*: T. Sarjakoski, Y.M. Santos and T.L. Sarjakoski, eds. *Proc. 19th AGILE Conference on Geographic Information Science, Geospatial Data in a Changing World*, Lecture Notes in Geoinformation and Cartography, Helsinki, Finland. Springer, 399–415.

Peng, D., Wolff, A., and Haunert, J.H., 2017. Using the A* algorithm to find optimal sequences for area aggregation. *In*: M.P. Peterson, ed. *Proc. 28th International Cartographic Conference (ICC), Advances in Cartography and GIScience*, Lecture Notes in Geoinformation and Cartography, July, Washington DC, USA. Springer, 389–404.

Schwartges, N., *et al.*, 2013. Optimizing active ranges for point selection in dynamic maps. *In*: *Proc. 16th ICA Workshop on Generalisation and Multiple Representation (ICAGM)*. Available from: https://kartographie.geo.tu-dresden.de/downloads/ica-gen/workshop2013/genemappro2013_submission_5.pdf.

Thiemann, F. and Sester, M., 2018. An automatic approach for generalization of land-cover data from topographic data. *In*: M. Behnisch and G. Meinel, eds. *Trends in spatial analysis and modelling: Decision-support and planning strategies*. Geotechnologies and the Environment, vol. 19. Springer, Ch. 10, 193–207. Available from: https://doi.org/10.1007/978-3-319-52522-8_10.

Touya, G. and Dumont, M., 2017. Progressive block graying and landmarks enhancing as intermediate representations between buildings and urban areas. *In*: *Proc. 20th ICA Workshop on Generalisation and Multiple Representation (ICAGM)*, July. Available from: https://kartographie.geo.tu-dresden.de/downloads/ica-gen/workshop2017/genemr2017_paper_1.pdf.

van Kreveld, M., 2001. Smooth generalization for continuous zooming. *In*: *Proc. 5th ICA Workshop on Generalisation and Multiple Representation (ICAGM)*, August, Beijing, China. Available from: http://www.staff.science.uu.nl/~kreve101/papers/smooth.pdf.

van Oosterom, P., 2005. Variable-scale topological data structures suitable for progressive data transfer: The GAP-face tree and GAP-edge forest. *Cartography and Geographic Information Science*, 32 (4), 331–346. Available from: https://www.tandfonline.com/doi/abs/10.1559/152304005775194782.

van Oosterom, P. and Meijers, M., 2014. Vario-scale data structures supporting smooth zoom and progressive transfer of 2D and 3D data. *International Journal of Geographical Information Science*, 28 (3), 455–478. Available from: https://doi.org/10.1080/13658816.2013.809724.

van Oosterom, P., *et al.*, 2014. Data structures for continuous generalisation: tGAP and SSC. *In*: D. Burghardt, C. Duchêne and W. Mackaness, eds. *Abstracting geographic information in a data rich world: Methodologies and applications of map generalisation*. Lecture Notes in Geoinformation and Cartography. Cham: Springer, Ch. 4, 83–117. Available from: https://link.springer.com/chapter/10.1007/978-3-319-00203-3_4.

van Putten, J. and van Oosterom, P., 1998. New results with generalized area partitionings. *In*: *Proc. 8th International Symposium on Spatial Data Handling (SDH)*, July, Vancouver, Canada. 485–495. Available from: www.gdmc.nl/oosterom/sdh98.pdf.

Šuba, R., 2017. *Design and development of a system for vario-scale maps*. phdthesis. Delft University of Technology. Available from: https://journals.open.tudelft.nl/abe/article/view/1877.

Šuba, R., *et al.*, 2014. An area merge operation for smooth zooming. *In*: J. Huerta, S. Schade and C. Granell, eds. *Proc. 17th AGILE Conference on Geographic Information Science, Connecting a Digital Europe Through Location and Place*, Lecture Notes in Geoinformation and Cartography, Cham. Springer, 275–293. Available from: http://dx.doi.org/10.1007/978-3-319-03611-3_16.

Šuba, R., Meijers, M., and van Oosterom, P., 2016. Continuous road network generalization throughout all scales. *ISPRS International Journal of Geo-Information*, 5 (8). Available from: http://www.mdpi.com/2220-9964/5/8/145.

Weibel, R., 1997. Generalization of spatial data: Principles and selected algorithms. *In*: M. van Kreveld, J. Nievergelt, T. Roos and P. Widmayer, eds. *Algorithmic foundations of geographic information systems*. Lecture Notes in Computer Science, vol. 1340. Springer, Ch. 5, 99–152. Available from: https://link.springer.com/content/pdf/10.1007/3-540-63818-0_5.pdf.

Appendix A. Create the table of weights and the table of compatibility values

This appendix shows how to create the table of weights and the table of compatibility values in PostgreSQL. The values of the two tables are used in our greedy algorithm (see Section 3.1). Currently, we have not investigated on how to define the weight for a class, so we assign value 1 to the weights of all the classes. That is to say, the least important area is the one with the smallest size. The class similarity is defined based on the class codes as the codes indeed imply a hierarchy.

```
DROP TABLE IF EXISTS public.class_weights;
CREATE TABLE public.class_weights (
  code INTEGER,
  weight FLOAT
);

DROP TABLE IF EXISTS public.class_comp_matrix;
CREATE TABLE public.class_comp_matrix (
  code_from INTEGER,
  code_to INTEGER,
  code_dis FLOAT,
  comp_value FLOAT
);


CREATE OR REPLACE FUNCTION
  populate_class_weights(class_codes INTEGER[]) RETURNS void AS
  $$
  DECLARE
    class_code INTEGER;
  BEGIN
    FOREACH class_code IN ARRAY class_codes
    LOOP
      EXECUTE format(
        'INSERT INTO %s (code, weight)
         VALUES ($1, $2);', 'class_weights')
      USING class_code, 1;
    END LOOP;
  END;
  $$ LANGUAGE plpgsql;


CREATE OR REPLACE FUNCTION
  populate_class_comp_matrix(class_codes INTEGER[]) RETURNS void AS
  $$
  DECLARE
    code_fr INTEGER;
    code_to INTEGER;
    division_fr INTEGER;
    division_to INTEGER;
    code_dis FLOAT;
    dis_max FLOAT := 10;
    comp_value FLOAT;
    code_dis_test INTEGER[];
    code_dis_test_arr INTEGER[] := ARRAY[[1000,8], [100,6], [10,4], [1,2]];
  BEGIN
    FOREACH code_fr IN ARRAY class_codes
    LOOP
      FOREACH code_to IN ARRAY class_codes
      LOOP
        code_dis := 0;
        IF code_fr != code_to THEN
```

```
            -- code_dis_test is the inner array of code_dis_test_arr
            FOREACH code_dis_test SLICE 1 IN ARRAY code_dis_test_arr
            LOOP
              division_fr := code_fr / code_dis_test[1];
              division_to := code_to / code_dis_test[1];
              IF division_fr != division_to THEN
                code_dis := code_dis_test[2];
                EXIT;
              END IF;
            END LOOP;
          END IF;

          comp_value := (dis_max - code_dis) / dis_max;
          EXECUTE format(
            'INSERT INTO %s (code_from, code_to, code_dis, comp_value)
            VALUES ($1, $2, $3, $4);', 'class_comp_matrix')
          USING code_fr, code_to, code_dis, comp_value;
        END LOOP;
      END LOOP;
    END;
    $$ LANGUAGE plpgsql;


CREATE OR REPLACE FUNCTION
    populate_tables() RETURNS void AS
    $$
    DECLARE
      total_cost FLOAT := 0;
      class_codes INTEGER[] := ARRAY[10310, 10311, 10410, 10411, 10510, 10600,
          10700, 10710, 10720, 10730, 10740, 10741, 10750, 10760, 10780,
          12400, 12500, 13000, 14010, 14030, 14040, 14050, 14060, 14080,
          14090, 14100, 14120, 14130, 14140, 14160, 14162, 14170, 14180];
    BEGIN
      EXECUTE populate_class_weights(class_codes);
      EXECUTE populate_class_comp_matrix(class_codes);
    END;
    $$ LANGUAGE plpgsql;


SELECT populate_tables();
```

Appendix B. Communicate valid states

Section 3.4 shows how to snap to only valid states to avoid half-way merging. This appendix illustrates how to communicate valid states from the server side to the client side. By sending only the exceptions of the event number, we try to decrease the size of the sent data.

B.1. *On the server side*

On the server side, we compute the values shown in Table B1. These values are for merging sequence of Figure 1k–o (also see Table 1b), where parallel parameter r_{parallel} was set to 0.3. Note that this parameter value is extremely high, just used to explain the principle in an artificial simple example. The computation starts from step 1. At the beginning, there are 7 areas on the map, i.e., $|M_0| = 7$. According to Equation 1, our target is to parallel three events ($n_{\text{target},1} = 3$). However, only two event can be

paralleled in step 1 because some areas are blocked (see Figure 4b). Therefore, we have $n_{\text{event},1} = 2$. We require that the low state is $s_{\text{low},1} = 0$ for the first step. Then, the s_{high} value can be computed by

$$s_{\text{high},i} = s_{\text{low},i} + n_{\text{event},i}. \tag{B1}$$

That is, we have $s_{\text{high},1} = 2$ (also see the s_{high} value in the first row of Table B1). At this point, the computation for step 1 completes.

Table B1.: Some information of the merging sequence shown in Table B1. Column n_{area} shows the area number of the map at starting state s, that is, $|M_s|$ of Equation 1.

step	n_{area}	n_{target}	n_{event}	s_{low}	s_{high}
1	7	3	2	0	2
2	5	2	2	2	4
3	3	1	1	4	5
4	2	1	1	5	6

For the next step, the number of areas can be computed by

$$n_{\text{area},i+1} = n_{\text{area},i} - n_{\text{event},i},$$

where variable $n_{\text{area},i}$ denotes the area number at the low state of step i. Furthermore, the state-low value of step $i + 1$ (i.e., $s_{\text{low},i+1}$) is the same as the state-high value of step i (i.e., $s_{\text{high},i}$). Again, the target number of parallel events (i.e., $n_{\text{target},i+1}$) is computed by Equation 1, the number of actual parallel events is obtained from the greedy algorithm, and the state-high value (i.e., $s_{\text{high},i+1}$) is computed by Equation B1. The computation of all the steps starts from step $i = 1$ and finishes until only one area left on the map. As a result, we have all the values of Table B1.

Now, we have a column of n_{event} values. Among them, we record the exceptions (i.e., when value n_{event} is different from value n_{target}) with the corresponding steps in a list. The *exception list* is $[[1, 2]]$ for the example of Table B1. For the pair of values in the inner square brackets, the first one represents the step, and the second value represents the actual number of events n_{event}. The exception list, the number of faces, and the parallel parameter will be sent to the client side.

B.2. *On the client side*

When a user opens our web map, the client side receives the exception list, the number of faces, and the parallel parameter from the server side. Starting from step $i = 1$, we see if the step is in the exception list. If so, the event value of step i from the list is assigned to $n_{\text{event},i}$. If not, value $n_{\text{target},i}$ (see Equation 1) is computed and assigned to $n_{\text{event},i}$. As a result, we have the n_{event} values on the client side (see column n_{event} in Table B1). By accumulating the n_{event} values, we obtain the list of valid states $S_{\text{valid}} = [0, 2, 4, 5, 6]$.