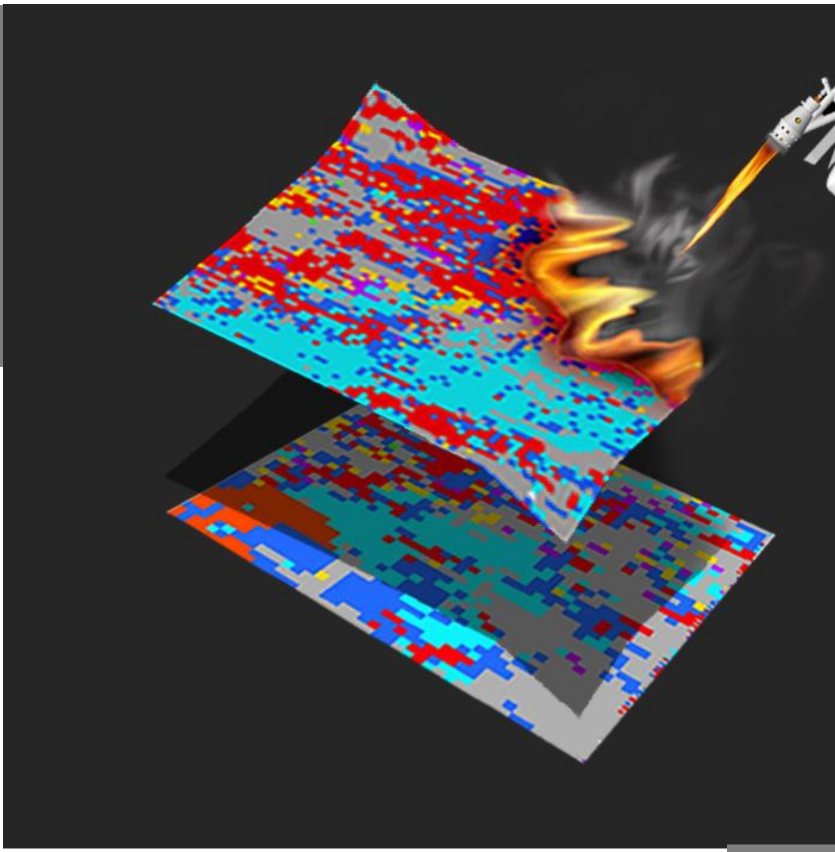
Thesis Report

Master Geo-Information Science

# Criterion-based Generalization of Raster Categorical Maps



2014

WAGENINGEN UR

For quality of life

# Criterion-based Generalization of Raster Categorical Maps

**Athanasios Strantzalis**

Registration Number: 830703-811-080

## Supervisors:

| | |
|---|---|
| Dr. Ir. Sytze de Bruin | (Centre for Geo-Information) |
| Dr. Ir. Gerard Heuvelink | (Soil Geography and Landscape / ISRIC) |

A thesis submitted in partial fulfilment of the degree of Master of Science

at Wageningen University and Research Centre,

The Netherlands.

Thesis Code Number: GRS-80436

Wageningen University and Research Centre

November 2014

Wageningen, the Netherlands

# Foreword

This report is a result of almost a year of research in cartographic generalization of raster categorical maps. I feel the need to express my gratitude to Dr. Gerard Heuvelink and Dr. Sytze de Bruin for their guidance and patience during this period. This research would not be possible without them. Furthermore, I would like to thank Eirini Simitzi for the psychological support, as well as for the creation of the image at the cover page.

Athanasios Strantzalis

17 November 2014, Wageningen

## Contents

# ABSTRACT

This research deals with a fundamental issue of Geo-information science, that of cartographic generalization and more specifically with the generalization of raster categorical data. Map generalization is the process that reduces and simplifies the details of the map; is performed mainly for visualization purposes, when a map is converted from a fine to a coarse scale / resolution.

However, during the generalization process the information and quality of the map are also reduced and errors are introduced. The challenge is to apply generalization in a way that optimizes the generalization objectives. Four generalization criteria were used, regarding the size and shape of the map entities, as well as the maintenance of the semantics and the proportion of the categorical classes. In order to obtain a generalized map that minimizes the weighted sum of the individual generalization criteria, the Simulated Annealing (SA) optimization algorithm was used.

SA is an iterative optimization algorithm that mimics the technique of the annealing process in metallurgy. In every step of the iterative procedure, a map entity is randomly selected and reclassified, while the effects of that reclassification in the map are evaluated, with the use of the defined generalization objective function. Based on the evaluation results, it is decided whether the proposed change is accepted or rejected. The iteration continues until a maximum number of iterations is reached, or the result becomes stable.

The developed algorithm was applied to two 120x120 pixels sections of the SoilGrids1km product, which is a global soil product produced by ISRIC – World Soil Information. The two selected areas were chosen in order to demonstrate behavior of the generalization at different levels of map complexity. The generalization took place at three threshold values of 5, 17 and 65 pixels for the minimum mapable unit.

The performance of the SA was evaluated based on the required computational time and on the goodness of the generalization. Additionally, the results of the developed generalization algorithm were compared with the results of a commercial generalization methodology. It was shown that when the parameters of Simulated Annealing are configured properly, the developed algorithm can outperform the commercial methodology, in terms of value of the objective function.

**Keywords**: Cartographic Generalization, Soil Map Generalization, Taxonomic Distance, Simulated Annealing Optimization, SoilGrids1km

# 1. INTRODUCTION

## 1.1 Context and Background

Map generalization is the process that reduces the details of a map and discerns regional patterns so that it becomes aesthetically pleasant, and enables users to succeed in a given task (Mackaness et al., 2007). Generalization is performed for visualization purposes, when the map needs to be converted from a fine to a coarse scale/resolution and thus, less detail is needed.

Map generalization is a challenging task primarily because the type of applied operations, their sequence and the configuration of the input parameters depend on the specific application requirements (Smirnoff et al. 2012). Thus, a large effort of human analysis of the geographic data and contextual decisions is usually required, in order one to decide what to generalize and how (Smirnoff et al., 2008).

Automated map generalization is the automated process of applying a series of operations to produce a generalized map. Automated generalization is an ongoing field of research mostly among National Mapping Agencies (NMAs), driven by the demand of maps in various scales. Despite the progress during the last years in the automated generalization domain (Steiniger and Weibel 2005), vendors still find it difficult to implement effective automated generalization solutions (Stoter, 2005).

According to Stoter et al. (2009) this occurs for three main reasons:

1. The lack of formal definitions of map specifications.
2. The lack of generic specifications among NMAs.
3. Generalization is a subjective process in which more than one ideal generalization result is often possible. This subjectivity is hard to automate.

It has been long accepted that map generalization can be divided into two sub-processes: Database or model generalization, which is the generalization of the attributes of the data, and cartographic generalization, which deals with the representation of the map entities (Mackaness et al., 2007). This research focuses on the latter.

### 1.1.1 Cartographic Generalization of Categorical Data

Much research has taken place over the years regarding automated map generalization (Föerster et al., 2007; Kazemi et al., 2004; Mackaness et al., 2007). Despite the fact that the research has mainly focused on the generalization of vector topographic data, other types of data have been explored as well. In our research however, we focus at a smaller part of the generalization spectrum, namely the cartographic generalization of gridded categorical maps.

Thematic maps are special-purpose maps that visualize a single phenomenon and they either provide information about zones where a certain characteristic is assumed to be constant, or about the distribution of some statistic (Carrao and Henriques, 2001). Categorical maps are special cases of thematic maps which they mainly visualize data regarding soil, vegetation, geology or land use (Peter and Weibel, 1999). In the literature however, no distinction is usually made between thematic and categorical data. This research deals with the generalization of soil raster maps.

Even though a large body of literature exists on surveying and (field) mapping guidelines in geology and soil science, automated generalization solutions in these fields have not been explored in depth (Steiniger and Weibel, 2005). Additionally, polygon generalization is considered as one of the most complicated and least defined steps in the generalization procedure (Smirnoff et al., 2008).An overview of commercial tools and dedicated solutions for polygon generalization (in both vector and raster format) is presented in Section 1.3.

The issue of soil map generalization was addressed in the E-soter project. E-soter was a research project funded by the European Union that took place between 2008 and 2012. The aim of the project was to develop a web-based regional pilot platform with data, methodologies and applications that make use of remote sensing imagery to validate and extend existing soil, landform and parent material data. Within E-soter an iterative procedure was developed for soil map generalization. More precisely, this iteration procedure automatically merges soil (vector) polygons, that are deemed too small for representation at the targeted level of detail, to the neighbor with the smallest semantic distance (Section 2.1.2).

Gao et al., (2004) developed a framework for soil map generalization by dividing the process into four main phases (preparation and preprocessing, database generalization, graphic generalization and evaluation of generalization results). They defined twelve generalization criteria associated with each of these phases, in order to designate the specifications of the final map, evaluate and compare different solutions.

### 1.1.2 Criteria for Cartographic Generalization

Cartographic generalization based on criteria or constraints is a domain that has been extensively researched. (Gao et al., 2004; Stoter and Baella, 2010). Criteria express the rules of the generalization or, in other words, the properties we wish the generalized output to have. Another usability of generalization criteria is that they evaluate the result after the operators have been executed (Gao et al., 2004). Since any generalization process is performed with a specific application in mind, the requirements of the user group must control the generalization process and define the generalization criteria (Molenaar 1996).

Even though the use of just one criterion similar to the human eye resolution limitation (removal of objects smaller than a certain threshold) achieves generalization effects for polygon thematic features, researchers have developed algorithms that take into account additional criteria (Li, 2007). However, even the seemingly easy problem of identifying polygons which are too small may become complex, since it may have sufficient area, yet may be too thin and belong to a less important class and therefore still be eliminated (Bader and Weibe, 1997). Additionally, removal of a map entity results in a space that must be filled by another entity and this has important semantic consequences (Edwardes and Mackaness, 2000).

For a categorical generalized map to be readable, except of elimination or aggregation of small map entities, also simplification of the shape of the polygons is required, while maintaining close similarity with the original map (Smirnoff et al., 2008). However, all these criteria cannot be fulfilled simultaneously, since in most cases the high value of one criterion will lead to a low value of another, making in this way the final map a result of a compromise between the criteria.

According to Steiniger and Weibel, (2005), several solution strategies for multi criteria problems are possible, in cases where more than one ideal situations exist.

- Solve interactively: the human decides what to do (not applicable for automated methods)
- Combine a sequence of rules with fixed order.
- combine several rules in dynamic order with respect to the situation (whereas the order is determined by a human in a pre-analysis phase)
- Use of optimization methods to search for an optimal result.

In this research an optimization algorithm has been used, in order to find the solution that satisfies the chosen criteria, in the biggest degree possible. In order to take this approach, the chosen criteria had to be quantified and cost values assigned to every one of them. Thus, generalization is considered in this research as a minimization of the total cost value related with chosen criteria, for the entire map.

### 1.1.3 Optimization Algorithms

The problem, as stated above, regards finding that map configuration that minimizes the total cost of the generalization criteria. However, finding this optimal map configuration is a challenging task, due to the fact that the number of potential map configurations is enormously large so that it is impossible to explore each and every one of them in a reasonable amount of time. In order to solve that problem, the simulated annealing (SA) optimization algorithm is used.

In general, optimization methods are used to find the best solution of a problem, in terms of minimizing or maximizing an objective function. SA is a heuristic optimization algorithm that mimics the annealing process used in metallurgy. Heuristic optimization algorithms can generate acceptable solutions, not necessarily the optimal one, within reasonable time (Pukkala and Kurttila, 2005).

Literature regarding cartographic generalization and optimization algorithms is not extensive. Ware et al., (2003) applied SA and Ware et al., (2003) a genetic algorithm, to address the issues of the shapes' displacement, exaggeration, intersections and others, for vector topographic data. Both studies showed optimization algorithms to be successful for cartographic generalization.

### 1.1.4 SoilGrids1Km

The developed generalization methodology presented in this thesis was applied to the SoilGrids1km product, which is a global soil type map generated at ISRIC – World Soil Information. The product is publicly available via a web mapping application and a tiling service, where the world is divided into 10x10 degree tiles (Figure 1.1). It is classified based on the WRB taxonomy system into 32 soil classes, in raster (GeoTIFF) format and has pixel size of 0.0083x0.0083 degrees. The actual pixel size on the ground is approximately 930x930m, however depends on the longitude.

Since no generalization has been performed to the product, it is always served with the same level of spatial detail, no matter at which scale it is shown on the computer screen. Thus, when zoomed out (at small scales) the soil map appears fragmented to the user, who cannot clearly distinguish the main soil classes of the area, as a result of the intense salt and pepper effect.

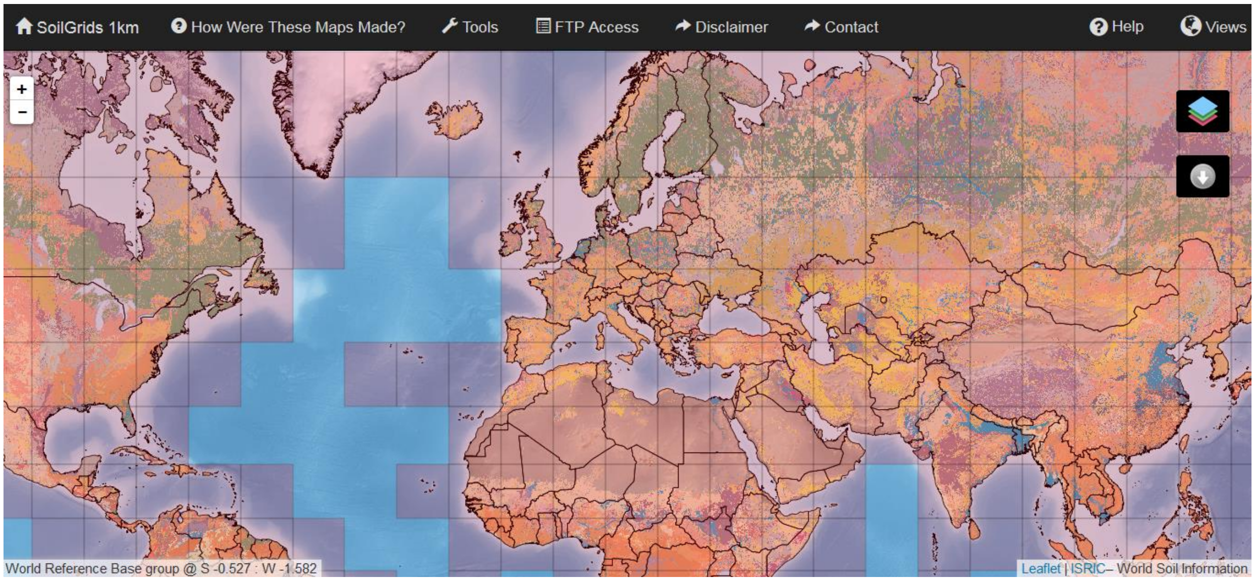

SoilGrids 1km
How Were These Maps Made?
Tools
FTP Access
Disclaimer
Contact
Help
Views
+
–
World Reference Base group @ S -0.527 : W -1.582
Leaflet | ISRIC– World Soil Information

Figure 1.1: SoilGrids1km web map application

The generalization algorithm was applied to two different sub-areas of SoilGrids1km for multiple generalization levels. The two test areas have the same size (120x120 pixels) but different level of complexity, in order to draw conclusions on the algorithm's performance with inputs with different degrees of complexity. Figures 1.2 and 1.3 present the two SoilGrids1km tiles, one in the Netherlands and one in Turkey, as well the subsets used.

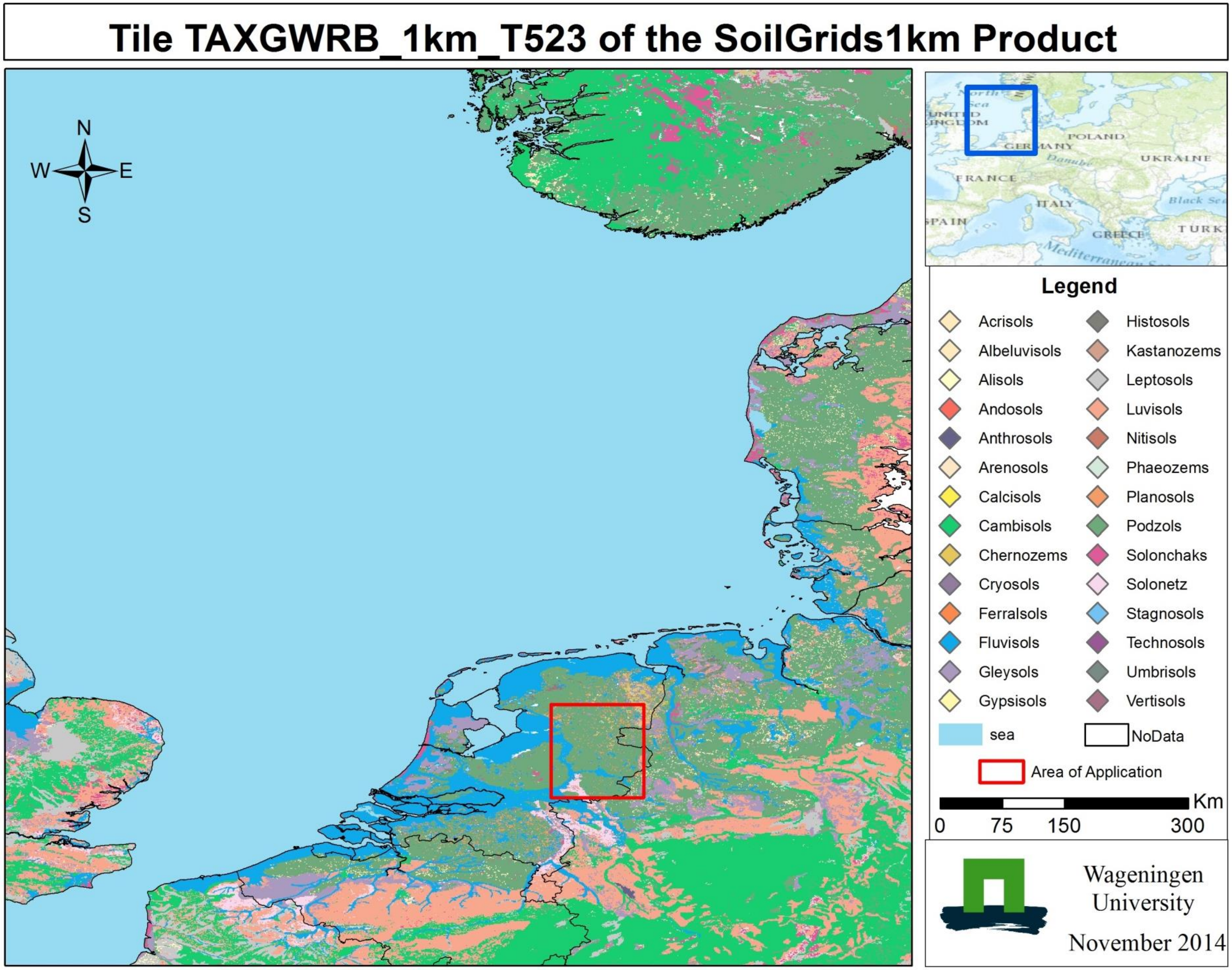


Figure 1.2: SoilGrids1km 10x10 degrees tile of the Netherlands and 120x120 pixels area of application

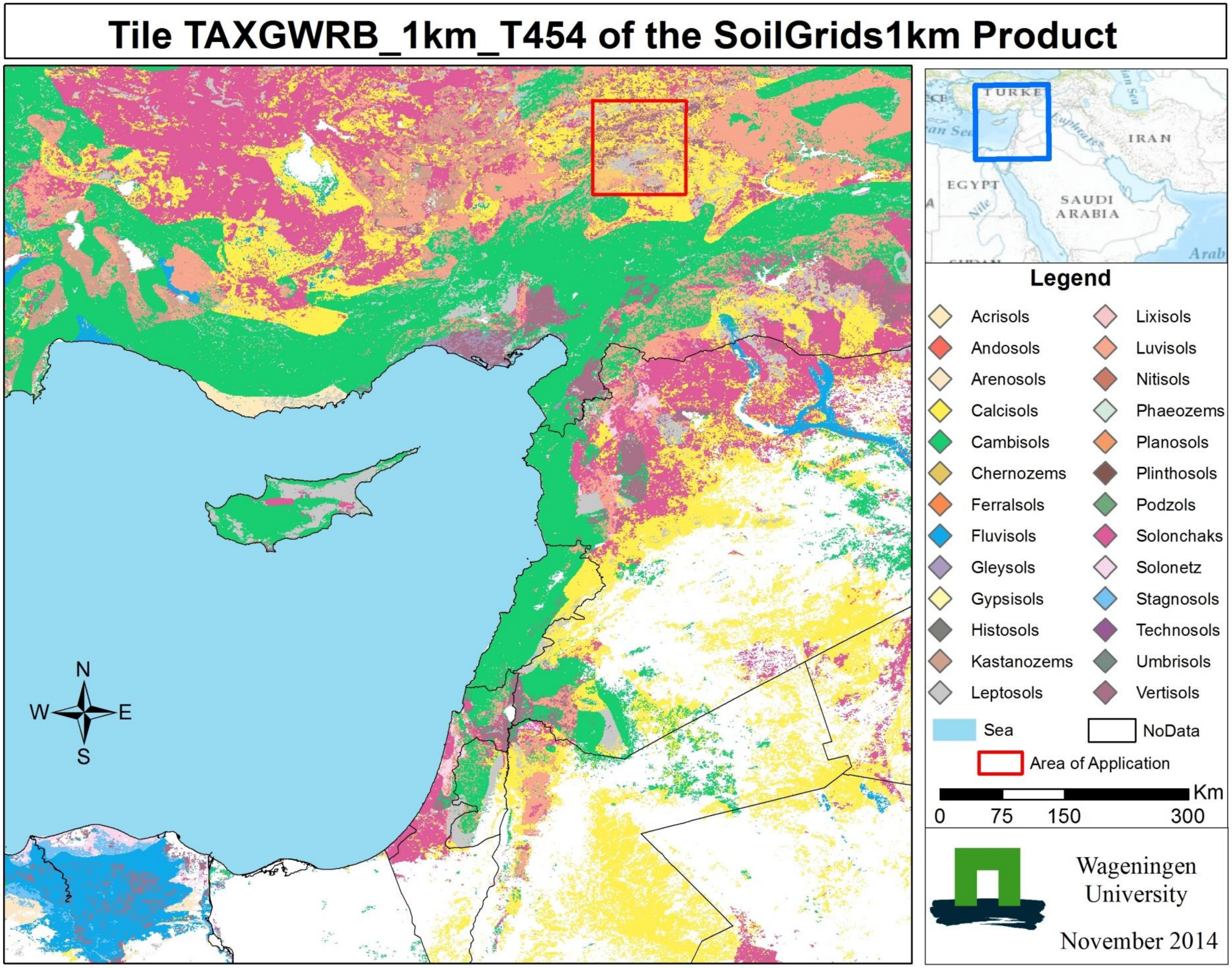


Figure 1.3: SoilGrids1km 10x10 degrees tile and 120x120 pixels area of application in Turkey

## 1.2 Research Objective and Research Questions

The general objective of this research is to develop, implement and evaluate an automated cartographic generalization procedure for categorical raster data. The research questions that need to be answered, in order to reach that objective are:

- RQ1: Which criteria are sensible to use for cartographic generalization of categorical maps and which metrics can be used for evaluation of the result?
- RQ2: How can Simulated Annealing be used for generating a near optimum result and which parameter settings are suitable?
- RQ3: How does the procedure perform in practice when applied to the SoilGrids1km map in terms of value of the objective function (goodness of the generalization) and processing time?

## 1.3 Conventional Generalization Approaches

### 1.3.1 Commercial Tools for Categorical Map Generalization

In contrast with topographic data, when one wants to generalize categorical maps the number of potential generalization operators is limited. The term operator refers to spatial functions (tools) used to produce a generalized result.  Even for vector topographic data however, operators included in commercial software in several cases  fail to meet the generalization objectives (Stoter et al., 2009).

The most commonly used generalization operator for categorical area features is geometry merging (Figures 1.4 and 1.5), where objects with small areas get united with a neighbor object, usually the one with the longest shared border or the largest area (Cheng and Li, 2006). Others potential operators include selective omission, aggregation, collapse and typification.

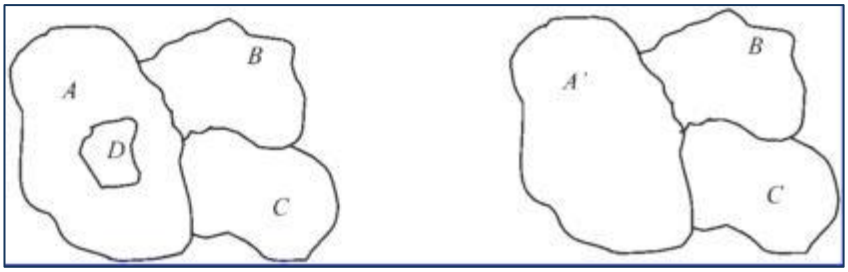


Figure 1.4: Merging (1): Small isolated area d is merged, into its neighbor area A (Cheng and Li, 2006).

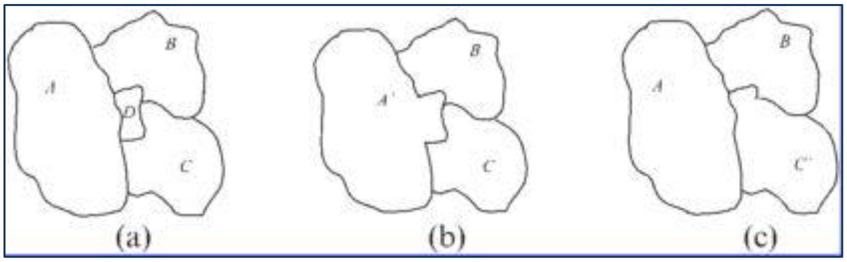


Figure 1.5: Merging (2): (a) Small area D and its neighbors; (b) Small area D is merged into A because A has the largest area; (c) Small area D is merged into C because C has the longest common boundary (Cheng and Li, 2006).

Besides the popular commercial GIS software, some dedicated solutions specifically for generalization purposes also exist. Perhaps the most notable are Radius Clarity by 1Spatial, axpand by Axes Systems, DynaGEN by Intergraph and Change/Push/Typify by the University of Hannover (Smirnoff et al., 2012). Satisfactory results in the domain of raster categorical generalization have been achieved by Cellular Automata approaches (Li et al., 2001; Smirnoffet al., 2008). Basically, a majority filter with an expanded neighborhood is applied and the level of generalization is determined by the number of iterations that the filter applies and the size of the neighborhood.

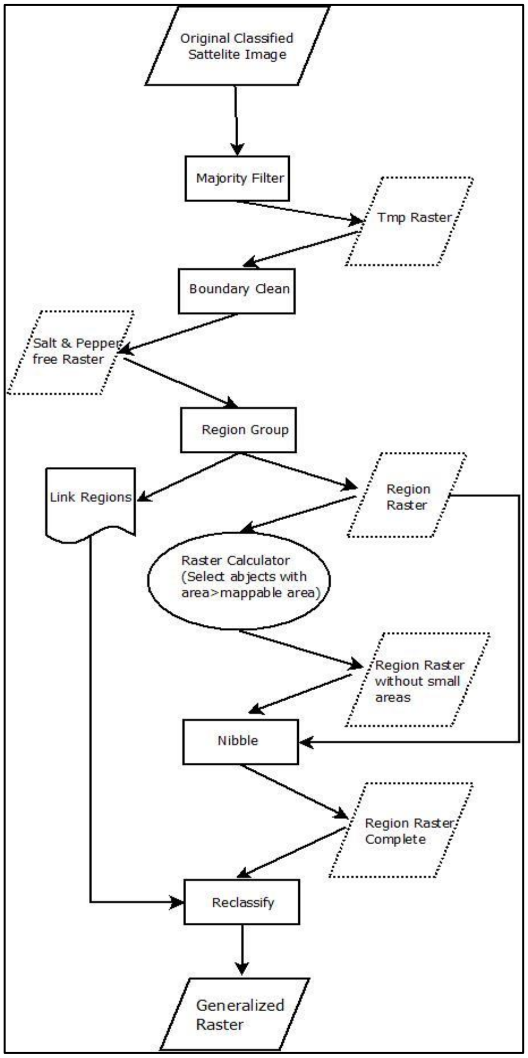

Original Classified Sattelite Image
Majority Filter
Tmp Raster
Boundary Clean
Salt & Pepper free Raster
Region Group
Link Regions
Region Raster
Raster Calculator (Select objects with area>mappable area)
Region Raster without small areas
Nibble
Region Raster Complete
Reclassify
Generalized Raster

Figure 1.6: Flowchart of the comparison generalization methodology

### 1.3.2 Generalization Methodology Chosen for Comparison

The results of the developed generalization algorithm were compared with the outputs of a conventional cartographic generalization procedure, which makes use of commercial tools. The chosen methodology for comparison is designed to deal with classified raster maps that have been derived from satellite imagery (ESRI).

Classification of satellite images usually results in maps which contain a large number of isolated and small groups of pixels. This similarity with the SoilGrids1Km product is the main reason why the specific methodology was chosen for comparison. Additionally, since this is a procedure designed for raster categorical maps, no customization is required for the implementation. Nevertheless, the objective of the commercial methodology is to eliminate groups of pixels smaller than a certain threshold, not taking into account additional criteria.

Figure 1.6 presents the flowchart of the commercial methodology. A series of generalization operators, from the ArcGIS suite, apply to the input data. More specifically, first a majority filter is used to eliminate the isolated pixels. Next, the boundaries of the shapes are smoothed and the entities with area smaller than a threshold value are identified and eliminated. Finally, the pixels that were eliminated at the previous step are filled based on the class of its closest neighbor. Figure 1.7 presents the result of the application for a classified satellite image.

The above generalization procedure was implemented in ArcGIS, in order to apply on the SoilGrids1Km, for comparison purposes. The comparison took place based on the objectives of the generalization, as defined by the generalization criteria.

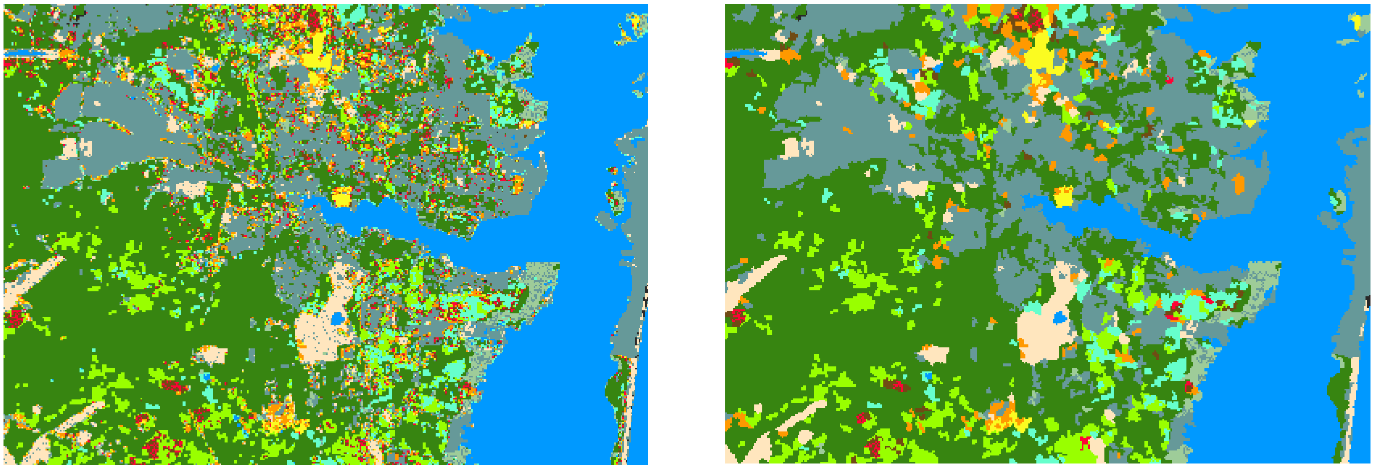


Figure 1.7: Initial not generalized raster (left) and result of the commercial generalization methodology (right).

## 1.4 Report Outline

In Chapter 2 the chosen commercial generalization methodology is presented, as well as the criteria of the generalization. In Chapter 3, the conceptual framework of the simulated annealing optimizer is demonstrated. The developed generalization methodology is presented and analyzed in Chapter 4. Next, the algorithm is first applied to synthetic datasets (Chapter 5). In Chapter 6 the methodology chosen for the configuration of the weights of the objective function is described and in Chapter 7 the algorithm is applied to the two sub-sets of the SoilGrid1km product. Finally, in Chapter 8 the results and the performance of the developed methodology are discussed and conclusions and recommendations are presented.

# 2. CRITERION-BASED GENERALIZATION

## 2.1 Generalization Criteria

The selection of the appropriate criteria is the first and the most important step of the generalization procedure, since a formalized description of specifications for the output should help to obtain a good solution (Stoter et al., 2009). Criteria express the objectives of the generalization and need to be defined and quantified. As said before, criteria not only express the specifications of the output, but they are also used for evaluation purposes.

For every chosen criterion a corresponding cost value is assigned to the output, taking into account the cases where the criteria have not been met. This cost value represents the degree in which the generalized map contains entities and properties that violate the chosen criteria and can be later used by optimization algorithms (Chapter 3) to find the solution that minimizes the sum of the criteria.

### 2.1.1 Area

The first and most straightforward criterion regards the minimum area of a map entity. This area is directly related to the scale of the output map, as well as limitations of the human eye, since the smaller the scale is, the bigger an object has to be in order to be distinguishable. Wambeke and Forbes ( 1986) presented minimum map areas according to the scale for soil maps. Their motivation was that the map units should not only be distinguishable, but also their symbol (color, shading) need to be printed clearly, which allows the users to link the unit to the legend.

Since this research deals with gridded data, a map entity is considered as a clump of pixels of the same class (hereafter called *patches*). That said, the area criterion applies to individuals patches of the raster map. The neighborhood is considered based on the rook's case (a pixel has 4 neighbors) and not based on the queen's case (a pixels has 8 neighbors).

Patches that are bigger than the minimum mapable area threshold will have a cost value of 0, resulting to no cost at all, as they do not violate the criterion. Patches smaller than this threshold however, will have a cost value that depends on their actual size and increases linearly from 0 to 1. The maximum cost is assigned to isolated pixel patches regardless the value of the area threshold. The total cost of a generalized output according to the area criterion, equals the sum of the individuals costs of all contained patches (Equation 1).

$$Area\ Total\ Cost = \sum_{patches} (Area\ Threshold - Patch\ Area\ ) \ / \ (Area\ Threshold - 1). \tag{1}$$

### 2.1.2 Taxonomic Distance

The second criterion regards the semantic similarity of the generalized result and the initial input map. Taxonomic similarity or distance among classes expresses the difference, or the distance in multivariable space, among taxonomic categories (Phillips, 2013). Basically, taxonomic distance expresses how close or similar two thematic classes are.

Semantic similarity among classes is used primarily in database generalization where the number of classes depends on the target scale (Yaolin et al., 2002). An exception to this was the cartographic generalization methodology used in the E-soter project, where the area and the taxonomic distance were the only criteria taken into account. Minasny et al. (2010) derived the semantic distances among the classes for the World Reference Base for Soil Resources (WRB) taxonomy system, which is the classification system used by SoilGrids1km. Figure 2.1 visualizes the taxonomic distance among the classes of the WRB..

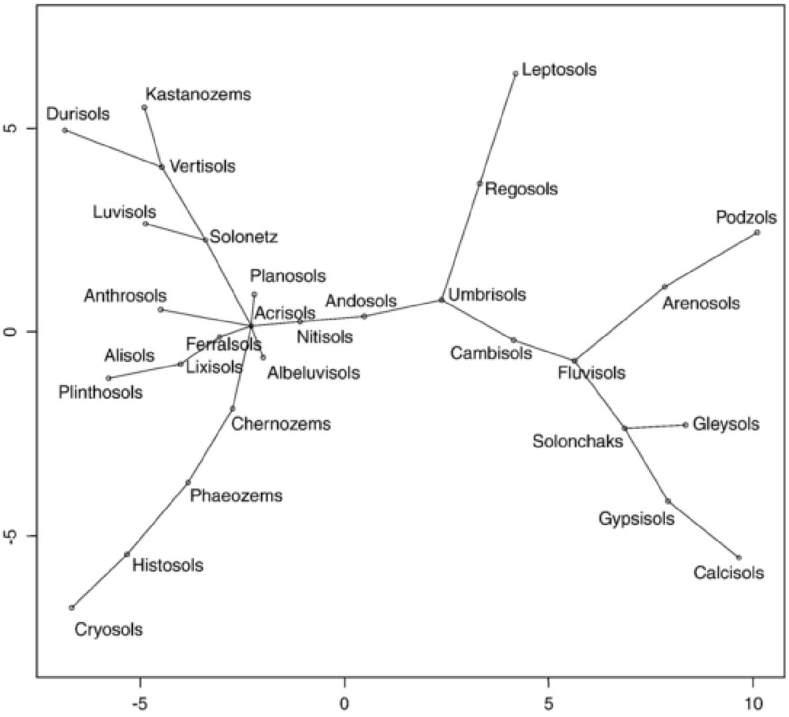


**Figure 2.1: A minimum spanning tree of the WRB soil groups. The nodes represent soil groups, while the edges represent the taxonomic distance**

The goal of this criterion is to maintain the semantic similarity, between the input and the output as much as possible. Of course, we cannot maintain the semantics completely, because this would lead to an identical map which will perform poorly for the other criteria. By including this criterion however, the area that the small patches occupy in the input raster will tend to be filled with a class similar to the initial one.

The taxonomic similarity of the output is evaluated with a cell by cell comparison between the original and the generalized map. If the class of a pixel has remained the same a cost of 0 is assigned, representing completely maintenance of the semantics, otherwise a cost is assigned equal to the semantic distance between the two classes (Equation 2). The total cost associated with the semantic similarity of the generalized map, is the sum of the costs of all the pixels of the map.

$$Taxonomy\ Total\ Cost = \sum_{pixels} Taxonomic\ Distance\ (\ Initial\ Class - Final\ Class\ ) \quad (2)$$

### 2.1.3 Shape

The third criterion regards the general shape of the patch. It is usual in every generalization process to simplify the shape of the map entities, like the methodology chosen for comparison does, at the initial steps (Section 1.3.2). In addition, even though the area of the patch may be larger than the area threshold, still the patch may be too narrow to visualize. A vertical or horizontal row of pixels of the same class, for example, would have a relative large area, however due to the shape of the object it would not be distinguished by the user. According to Nakos, (2001), these sliver objects can be identified by a numerical expression determined by the ratio between their perimeter and the square root of their area, a number independent of the size and units(Equation 3). Nakos suggests that a patch should be considered as sliver, if the value of the shape index is bigger than 7.5.

$$Shape\ Index = \ L \ / \sqrt{A} \tag{3}$$

Where:

$\boldsymbol{L}$ is the perimeter of the object

$\boldsymbol{A}$ is the area of the object

After experimentation, it was decided that patches will be penalized for a value of the shape index above 7. More precisely, no cost is assigned to patches that have a shape index smaller or equal to 7 and a maximum cost of 1 is assigned to patches that have an index value bigger or equal to 15. For patches that have a shape index in between the two thresholds, the cost value increases linearly from 0 to 1 (Equation 4). The overall shape cost of an output equals the sum of the individual costs of all contained patches.

$$Shape\ Total\ Cost = \sum_{patches} (\ Shape\ Index - 7\ )\ /\ (\ 15 - 7\ ) \tag{4}$$

### 2.1.4 Classes Proportion

The fourth and final generalization criterion, regards the maintenance of the proportion that the classes occupy, in the initial input raster. The goal for each categorical class is to occupy the same area proportion in the generalized output, as in the original map. In contrast with the taxonomic similarity criterion, complete maintenance of the original situation does not necessarily lead to a map identical with the input. A generalized output could have a class proportions cost of 0 (all classes occupy the same area proportion as the initial) and still be different from the input.

Classes for which the amount of space that they occupy has changed, are penalized according to that difference value (Equation 5). The total cost of the class proportions criterion derives from the sum of all classes of the generalized raster. The minimum total cost value is 0, while the maximum is 1 – (area of smallest class / total area) for the entire map, since no new classes can be introduced and thus, the worst case scenario is the initially smallest class to expand to the entire map.

$$Classes\ Proportion\ Total\ Cost = \sum_{Classes}(abs(Input\ Area - Output\ Area))/\ (2 * Total\ Area) \tag{5}$$

## 2.2 Objective Function

The four generalization criteria are combined in a weighted objective function to express the overall cost of a generalized raster (Equation 6). A corresponding weight has been assigned to each individual criterion, expressing the criterion's impact on the output. Thus, by modifying these weights the procedure can be controlled, depending on the desired output. We can, for example, introduce a high weight ($a_2$) for the total semantic distance accumulation ($C_2$), if we do not mind that small objects still exist in the output, or we can define a low weight, if the elimination of such objects is our main concern and we have a small interest at maintaining the semantic similarity.

$$C = a_1\, C_1 + \ a_2\ \ C_2 + \ a_3\, C_3\ + \ a_4\, C_4 \tag{6}$$

Where:

***C*** is the overall cost of the generalized map.

***$a_1$, $a_2$, $a_3$, $a_4$*** are the weights associated with criteria 1, 2, 3 and 4, respectively.

***$C_1$, $C_2$, $C_3$, $C_4$*** are the partial costs associated with criteria 1, 2, 3 and 4, respectively.

Even though each criterion is evaluated for the entire map, they penalize different properties of the output. For instance, the area and shape criteria assign a cost value per patch, while the taxonomic criterion assigns a cost per pixel and the class proportions criterion assigns a cost per categorical class. In addition, the range of potential values of these costs varies significantly. Thus, the weights used in different criteria are not comparable; if the weight of the shape cost, for example, is double than the weight of the taxonomy cost, that does not mean that the former would have twice the impact to the output than the latter. The criteria weights for the SoilGrids1km application were determined experimentally; based on the specific inputs and the target generalization level (Chapter 6).

# 3. SIMULATED ANNEALING OPTIMIZATION

## 3.1 Optimization Algorithms

Optimization algorithms or methods are used in order to select the best out of all potential solutions of a given problem (Neun et al., 2008). The space of all these potential solutions is called the solution space. If the solution space is relatively small exhaustive search methods can be used to determine the best solution. Exhaustive search methods are the simplest of all optimization algorithms, as they try all the solutions of the solution space and select the best one (Kokash, 2005). However, if the solution space is big, more complex optimization methods are used. Even in the simplest case where we have a 10x10 raster map and only 4 classes to assign the pixels to, this would result to a solution space of $4^{100} \approx 1.6 \cdot 10^{60}$ unique maps. It is obvious that even in simple cases the space of all potential solutions is much too big for exhaustive search methods

### 3.1.1 Heuristic Optimizers

Heuristic optimization algorithms make use of mechanisms found in nature that guide the search towards promising regions of the solution space (Maringer, 2005). Instead of returning the global optimum solution they usually return a near-optimum one. These optimization methods try to find iteratively the best solution by maximizing or minimizing a cost function (Steiniger and Weibel, 2005), like the one described in Section 2.2. Heuristic methods like Simulated Annealing, Genetic Algorithms, Ant Colony Optimization, and Particle Swarm Optimization have been successfully applied in solving multi-extremes function problems, combinatorial optimization problems, and other complex large scale optimization problems (Yang and Yang, 2011). These algorithms are different from each other, but they share a common flowchart (Figure 3.1).

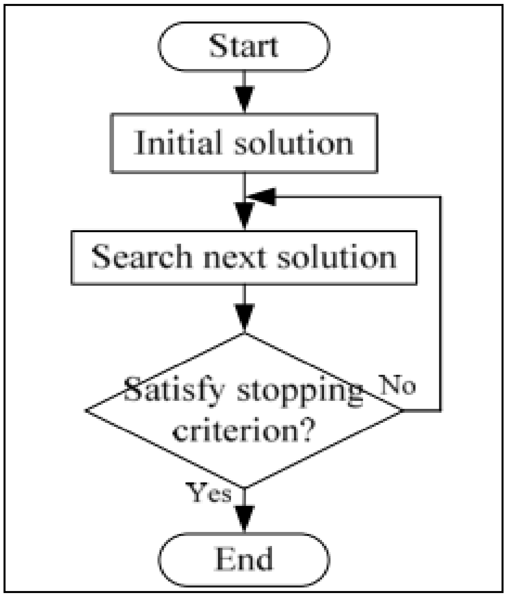


**Figure 3.1: Flow chart of iterative optimization algorithms** (Yang and Yang 2011)

The search procedures and control parameters that guide the search process vary according to the specific heuristic algorithm chosen (Pukkala and Kurttila, 2005). For instance, important parameters of the Particle Swam Optimization algorithm are the number of particles and the number of neighbors of each particle, while important Genetic Algorithm's parameters are population size and mutation percentage. These parameters are associated with the way each optimization algorithm searches for the best solution and they must be configured, in order the algorithm to begin the searching procedure. The more suitable values these parameters take the better the selected solution will be and less computational time will be needed.

An important issue of optimization algorithms is that they can be trapped in local optima. Local optima are zones in the solution space where the solution is the best locally (i.e. surrounded by worse solutions), but not globally optimal (Letourneau, 2007). Some heuristic algorithms have the ability to escape from local optima, while others do not.

## 3.2 Simulated Annealing

Simulated annealing (SA) is a heuristic optimization algorithm which mimics the phenomenon of annealing in solids. Annealing is a process in metallurgy where the metal is heated and then cooled slowly, in order to alter its properties by re-distributing the atoms in a configuration of lower energy. SA was introduced independently by Kirkpatrick et al. (1983) and Černý (1985) and since then it has been applied to a variety of optimization problems in different domains.

The wide use of the SA algorithm is a result of three reasons: 1) the ease of implementation that derives from the simplicity of the algorithm; 2) the performance, in respect to the quality of the solution in combination with the computational time; 3) the applicability of the algorithm to different problems (Michiels et al., 2007). Another advantage of the algorithm is that SA can escape from local optima, by occasionally accepting worse solutions.

### 3.2.1 Conceptual Framework

Using the analogy from the physical annealing process, the procedure starts with an initial temperature $t_0$ and an initial solution, which in most cases is derived randomly. At every iteration, a random candidate solution is generated and its energy (cost) is evaluated and compared with the energy of the current solution. If the candidate energy is less than the current, the solution is accepted and replaces the current one. If the energy of the candidate solution is more than the current's, the solution may still be accepted according to an acceptance probability function, which depends on the temperature value and the energy difference between the two solutions. The terms energy and cost are used interchangeably in this report.

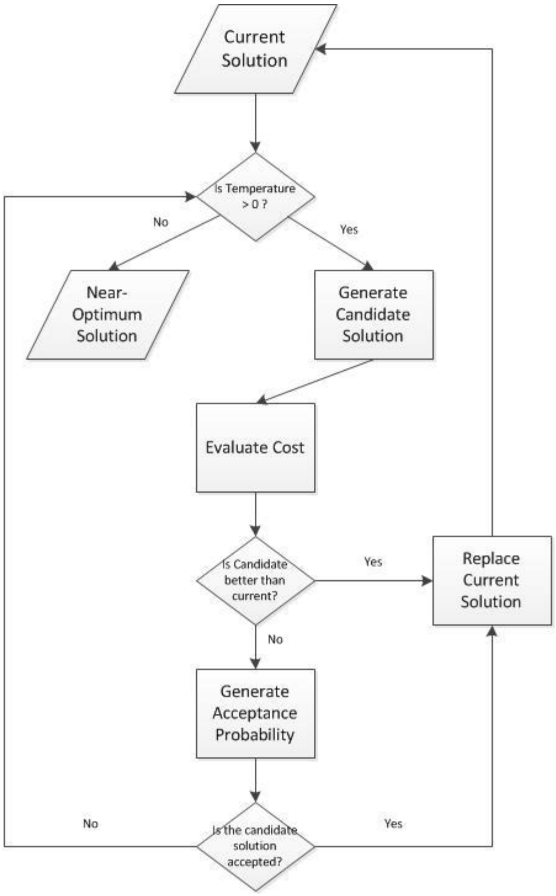

Current Solution
Is Temperature > 0 ?
No
Yes
Near-Optimum Solution
Generate Candidate Solution
Evaluate Cost
Is Candidate better than current?
Yes
Replace Current Solution
No
Generate Acceptance Probability
No
Is the candidate solution accepted?
Yes

Figure 3.2: Flowchart of Simulated Annealing

As the iteration proceeds, the temperature of the system decreases, according to the algorithm’s cooling schedule. The procedure is repeated until the temperature of the system reaches a lower threshold, where the system is considered cooled.  The SA flowchart is presented in Figure 3.2

At the beginning of the process, when the temperature is high, worse solutions are accepted with a high frequency. While the iterations proceed, worsening steps, although still possible, become less likely. When the temperature is very low , it is extremely rare for a worse solution to be accepted (Poole and Mackworth, 2010).

With the above mechanism SA is able to escape from local optima and find a global near-optimum solution. In fact, SA is guaranteed to find the optimal solution, if it is allowed an infinite number of iterations. In practice, however, good quality solutions can be obtained, but at the cost of increasing computational time (Michiels et al., 2007).

### 3.2.2 SA Parameters

In this section the parameters of the SA algorithm are presented. The parameters must be carefully chosen before the beginning of the iterative procedure as they have a strong impact on the quality of the results and the computation time.

- The **current state** (s) of the system. The configuration of the current solution. At the beginning a random initial solution can be used.
- The **candidate state** (s’) of the system. The configuration of the candidate solution. It is produced by the candidate generator procedure by slightly altering the current configuration and thus creating a solution in the neighborhood of the current.
- The **candidate generator procedure** (g). The stochastic procedure that produce the candidate solution from the current one.
- The **energy** (e) of the system at the current state.
- The **candidate energy** (e’). The energy of the candidate solution.

- The **temperature** (T) of the system. At the beginning of the iteration T = max and slowly decreases until a lower threshold close to 0.
- The **stopping criteria.** The iteration continues until a minimum temperature is reached or no more candidate solutions are accepted for many iterations, which means that result has become stable.
- The **acceptance probability function** P(e,e',T). The probability that a worse solution is accepted. It is proportional to the temperature, and reaches 0 for very small temperature values.
- The **cooling schedule**. The rate in which the temperature of the system decreases.

# 4. GENERALIZATION ALGORITHM

A generalization methodology for raster categorical maps has been developed, which applies the SA optimizer, presented in Section 3.2, to minimize the weighted objective function (Section 2.2). First, the conceptual framework of the developed algorithm is presented (Section 4.1). Next, in Section 4.2 the pseudo-code is demonstrated and in Section 4.3 issues about the implementation are analyzed. Finally, in Section 4.4, the Graphical User Interface of the developed algorithm is presented.

## 4.1 Conceptual Description

As described in Chapter 2, generalization is seen here as a minimization of the cost of the objective function, which expresses the objectives of the generalization. In order to adapt our generalization problem to SA terminology certain parameters are required. Considering the initial soil map as the initial state of the system, the candidate generation procedure will create a candidate solution in the neighborhood of the initial one, by slightly altering the initial raster map. Specifically, at each iteration one patch (clump of pixels of the same class), is randomly selected out of the current solution and all its pixels are reclassified, also randomly, to a class of a neighbor patch or its own initial class (in case its current class differs from the initial).

After the patch has been reclassified, the energy (e’) of the candidate solution is compared with the energy (e) of the current one. If e’<e, then the candidate solution is accepted and replaces the current. If e’>e, then the candidate solution is accepted with a probability equal to the value of the acceptance probability function (Section 4.3.3 below). As the iteration continues and the temperature is gradually decreased, the probability that a worse solution is accepted decreases, as well. Finally, as the temperature reaches 0 only better solutions are accepted. At the end of this iterative procedure, the last solution is the one with the minimum energy/cost found.

Patches are the raster equivalent of polygons in vector terminology. This means that this conceptual framework applies to vector data, as well. Some patches may consist of only one isolated pixel, while others consist of more connected pixels of the same class. By reclassifying a patch to the class of one of its neighbors, neighbor patches will be merged into one bigger patch (hereafter called *super-patch*), creating bigger entities and thus potentially decreasing the solution’s energy. The super-patches contain at least one patch and can grow and shrink (the number of patches they contain can increase or decrease), split into or unite two or more super-patches and get eliminated or created (Section 4.2.2 below).

Some choices were made in order to decrease the solution space and thus the required computation time. As said above, the potential classes that the patch can be reclassified to are limited to the ones of its neighbors. As the iterations proceed, however, and the class of the patches changes, so do their neighbors, giving in this way the possibility to a patch to reclassify to a class that was not initially in its immediate neighborhood. In addition, not all patches can be selected for reclassification, but only the small ones. If a patch is considered as small however, depends on the target scale of the generalization.

The area threshold that a patch can be selected for reclassification was set to 1.5 times the minimum mapable area of the target scale.

## 4.2 Pseudo-Code

The pseudo-code of the generalization algorithm is in Figure 4.1 First, pre-processing takes place in order to assess the energy of the initial solution and retrieve the necessary information. Next, the actual SA process begins. For as long as neither of the two stopping criteria are satisfied (Section 3.2.2), a new solution is generated, by changing the class of a randomly chosen patch. Then, its energy is evaluated and compared with the energy of the current solution. If the candidate solution is better (lower energy), an acceptance probability of 1 is assigned, which means that the solution is accepted. If, on the other hand, the candidate solution is worse than the current one, an acceptance probability value is generated (which depends on the current temperature of the system and the energy difference between the two solutions) which determines the acceptance and rejection rate of the solution.

```
Initialize temperature
                         Pre-Processing
Estimate the proportion of the classes in the initial raster map
Identify patches in raster map
Identify the class of each patch
Evaluate the area and perimeter of each patch
Identify the neighbors of each patch
Evaluate the common boundary between each patch and its neighbors.
Assess current energy
                         Simulated Annealing
While temperature > minTemperature and result_Is_Stable == False:
    Create candidate solution (candidate generator procedure)
    Evaluate candidate solution
    If candidate energy ≤  current energy:
        acceptance  probability = 1
    else:
        acceptance probability  =  generate acceptance probability
    if   acceptance probability  ≥ random(0,1):
        accept candidate solution
    else:
        reject candidate solution
    Decrease temperature
```

Figure 4.1: Pseudo-code of the developed generalization algorithm

## 4.3 Implementation of the Generalization Algorithm

For the implementation the python programming language was used, in combination with the ArcPy package from ESRI. Python is an open source scripting language that is becoming more and more popular, mainly due to its simplicity. Especially in the geo-sciences domain python is used widely. On the other hand, the ArcPy package is a commercial python library, which makes use of the spatial functions of the ArcGIS suite. This means that the developed algorithm can only run in a computer that has fully licensed ArcGIS software installed.

For a more technical description the developed methodology has been divided into four parts:

1. The pre-processing steps required for the actual SA process to begin.
2. The energy evaluation of the candidate solution.
3. The chosen acceptance probability function.
4. The chosen cooling schedule.

### 4.3.1 Pre-processing Steps

During the first steps of the algorithm and before the SA process takes place, pre-processing is required in order to gather the necessary information and store it to appropriate python data handlers. Five pieces of information about each patch are essential in order to be able to evaluate the initial and every candidate solution.

- The size of the patch
- The perimeter (to be able to evaluate the shape cost).
- The initial class
- The neighbor patches.
- The length of the shared boundary with each neighbor patch (shape cost).

In order to distinguish the separate patches of the initial raster categorical map, the ArcGIS Region Group tool is used. The result is a raster attribute table (RAT) with the unique patches of the initial map, which contain also information about the patches' size and class.

The only thing missing is information about the patch's neighbors and the length of the common boundary they share. For that the data were converted to a vector format, in order to apply the Polygon Neighbors tool. The Polygon Neighbors tool returns a table (dbf) with the neighbors of each patch and the length of their common boundary. This table in combination with the one created before, provide all necessary information the developed generalized algorithm needs to run.

### 4.3.2 Energy Evaluation

This section focuses on the procedure to evaluate the cost (energy) of the candidate solution. Since this is an iterative procedure that must be executed many times (for the SoilGrids1km application 100,000 runs were used), it would not be efficient to calculate the energy at each iteration again. For example, it would be highly inefficient to execute the Region Group tool at each iteration, in order to distinguish the patches of each solution and thus be able to evaluate the energy.

Due to the fact that a candidate map differs from the current only at the pixels of one small patch, the cost of the candidate solution is calculated by adding the cost difference between the current and candidate to the current cost. This means that we only need to focus on the local changes in the map. First we need to determine the super-patches that are affected by the reclassification of the selected patch, in order to determine the new areas and shapes, as well as the new value of the semantic disturbance and the class proportions. Second, the energy will be evaluated only for the affected patches at the current and the candidate state.

To determine the super-patches (and the contained patches) that are affected by the reclassification of the randomly selected patch, all special cases that the reclassification can lead to, must be considered. Two and only two of the below cases can apply at each time. One case for the super-patch that the selected patch belongs to at the current solution, and one case for the super-patch that the reclassified patch belongs to at the candidate solution.

The reclassification of a patch can lead to:

1. Aggregation with another patch (super-patch grow).
2. Dissolution from the previous super-patch (shrink).
3. Union of two or more super-patches.
4. Split of one super-patch into two or more.
5. Elimination of super-patch.
6. Creation of a new super-patch.

At the initial state when a patch is reclassified to the class of one of its neighbors, a super-patch is created containing the reclassified patch and its neighbor.  After a number of iterations more complex super-patches will occur (Figure 4.2). Figure 4.3 presents iteration n + 1 where the patch with id 5 has been aggregated to the patch with id 10.This patch, however, is part of a super-patch that contains the patches with ids 9, 10 and 11. This super-patch will grow containing also the patch with id 5. In this case only the above mentioned patches are affected by the reclassification. Thus, the total difference at the energy between solution n and solution n + 1 will be equal to the difference at the energy for those patches only, since all other patches (and super-patches) remain identical at the two solutions.

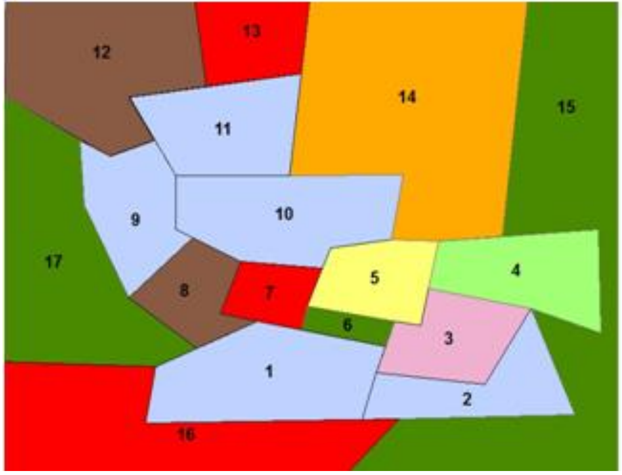


Figure 4.2: State of the system at iteration n

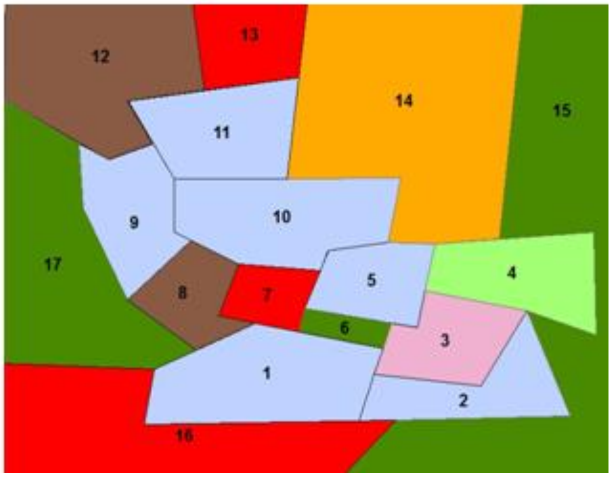


Figure 4.3: Iteration n + 1. Super-patch grow after patch 5 has been reclassified.

Figure 4.4 presents iteration n + 2 where the reclassification of patch 6 leads to the union of two super-patches, the first one containing patches [9, 10, 11, 5] and the second one containing patches [1, 2]. The new super-patch contains all the ids of the two old super-patches plus the reclassified patch. If, in a future iteration the class of patch 6 changes again, the super-patch that was just created will split back into two super-patches.

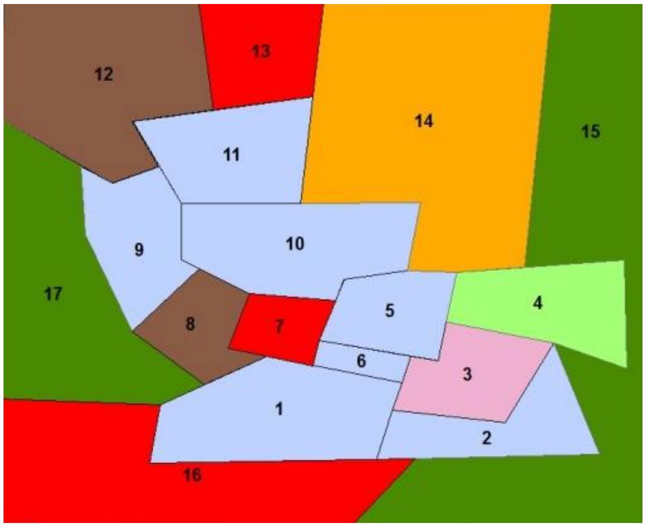


Figure 4.4: Iteration n + 2. Union of two super-patches After Patch 6 Reclassification

Additionally, a super-patch, that contains only one patch, will get eliminated when this patch is reclassified. Finally, a new super-patch can be created, when a patch changes its class to the initial value and at the same time there are no neighbors with that class to connect to.

Identifying all these cases is essential in order to be able to determine which super-patches (and thus which patches) are affected by the reclassification, and focus the energy evaluation at those areas specifically.

### 4.3.3 Acceptance Probability Function

The acceptance probability function generates the acceptance probability of a candidate solution, in case its energy is more than the current's (worse solution). In case the candidate cost is less than the current, the acceptance probability equals 1, which means that the candidate solution is accepted. Its value depends on the difference of the energies of the two solutions and the current temperature of the system (Equation 7). It is evaluated based on the Metropolis criterion (Aerts and Heuvelink, 2002; Nourani and Andresen, 1998).

$$P(accept\ worse\ solution) = \exp(\frac{e-e'}{T}) \tag{7}$$

Where:

$\boldsymbol{e}$ = Energy of the current solution

$\boldsymbol{e'}$= Energy of the candidate solution

$\boldsymbol{T}$= Temperature of the system

From the above equation derives that the smaller the difference between the two solutions, the higher the probability of acceptance, for a given temperature. Thus, candidate solutions with cost closer to the current one will be accepted more frequently than solutions with cost much worse than the current's. As the iterations continues, the temperature decreases and worse solutions are accepted less and less often, until only better solutions are accepted for small temperature values. Table 4.1 presents the acceptance probabilities values for various temperatures and candidate solutions. The K-worse labels in the table mean the units of energy difference. For example, for the first element the probability of acceptance for energy difference of 1 and temperature of 10, is 0.9. For the last element, the acceptance probability for energy difference of 3 and temperature of 0.25 is equal to 0.000006.

Table 4.1: Acceptance probabilities for various temperature and energy differences

| Temperature | Probability of acceptance | | |
|---|---|---|---|
| | 1-worse | 2-worse | 3-worse |
| 10 | 0.9 | 0.82 | 0.74 |
| 1 | 0.37 | 0.14 | 0.05 |
| 0.25 | 0.018 | 0.0003 | 0.000006 |

As regards the developed generalization algorithm, a pre-research analysis showed that for the application on the SoilGrids1km product and an area threshold of 5 pixels, an average energy difference of 0.8 is expected. Figure 4.5 presents a graph of the accepted probability plotted against the temperature, for worse candidate solutions that differ 0.8 energy units from the current solution. One can see that when the temperature takes values smaller than 0.5, the corresponding acceptance probability reaches 0.

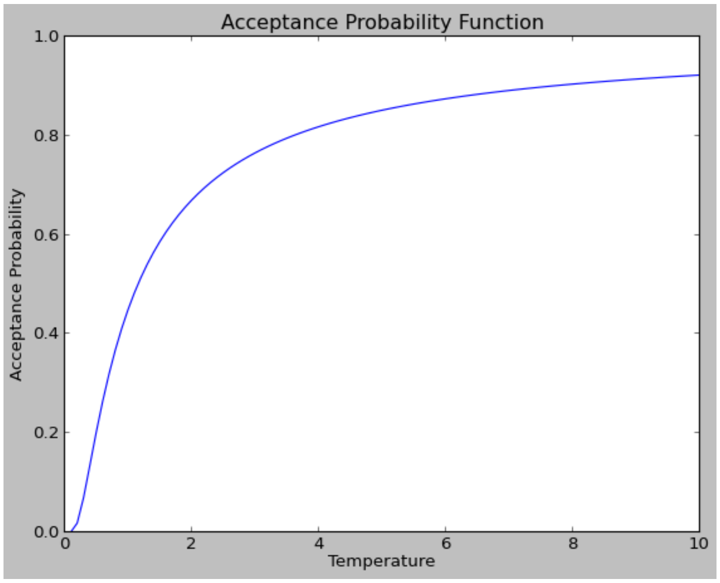


Figure 4.5 Acceptance probability against the temperature of the system, for energy difference between the solutions = 0.8

### 4.3.4 Cooling Schedule

The temperature of the system is the only parameter in SA whose value is not fixed, but decreases during the iterative procedure. The cooling or annealing schedule defines the value of the temperature at each iteration. It is considered the most important parameter of the algorithm; this is why several attempts have been made in the literature to derive or suggest effective cooling schedules (Nourani and Andresen, 1998). This has resulted to several theoretical and empirical cooling schedules, that can be categorized as monotonic schedules, adaptive schedules, geometric schedules and quadratic cooling schedules (Azizi and Zolfaghari, 2004).

In this research, an exponential cooling schedule is used, as shown in Equation 8. With a given initial and final temperature, the constant factor **a** defines the total number of iterations. Figure 4.6 presents the temperature of the system plotted against the iterations for initial temperature equals 10, minimum temperature of 0.1 and three different values of the constant factor, which corresponds to 100, 200 and 500 iterations.

$$t(i) = t0 * a^{i} \qquad (8)$$

Where:

*t(i)* = Temperature of the system at iteration i.

*t0* = Initial temperature of the system.

*a* = Constant factor

*i* = Number of iteration.

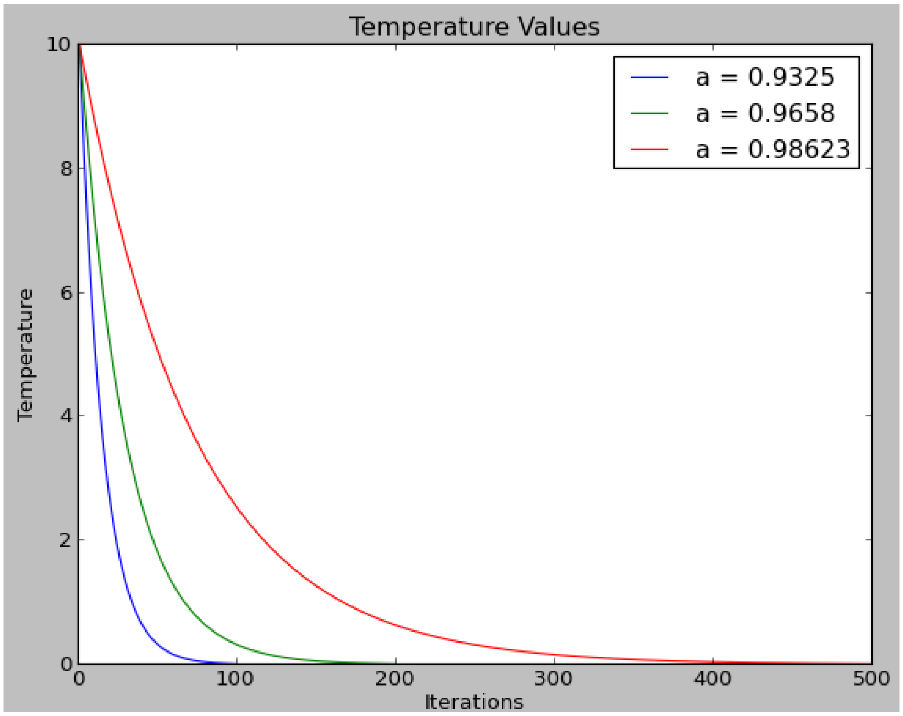


Figure 4.6: The temperature of the system as the time of the execution for 3 different values of the cooling schedule's constant factor. The initial temperature is 10 and the final 0.01.

For application to the SoilGrids1km product the parameters chosen are: initial temperature equals 100, minimum temperature equals 0.001 (first stopping criterion) and the constant factor equals 0.9998849, which correspond to 100,000 iterations. As regards the second stopping criterion, the algorithm will stop if the current energy has not improved for consecutive 5,000 iterations, which means that the result has become stable.

Figure 4.7 presents a graph of the probability of acceptance plotted against the number of iterations, for the above parameters of the cooling schedule and for an energy difference between the current and candidate solution of 0.8. Approximately, for the first half of the iterations worse candidate solutions are accepted with a decreasing probability. At the second half, the probability of accepting a worse solution is extremely small. In this way, for the first 50.000 iterations the algorithm is allowed to explore the solution space, while later on it only looks only for solutions that improve the current energy.

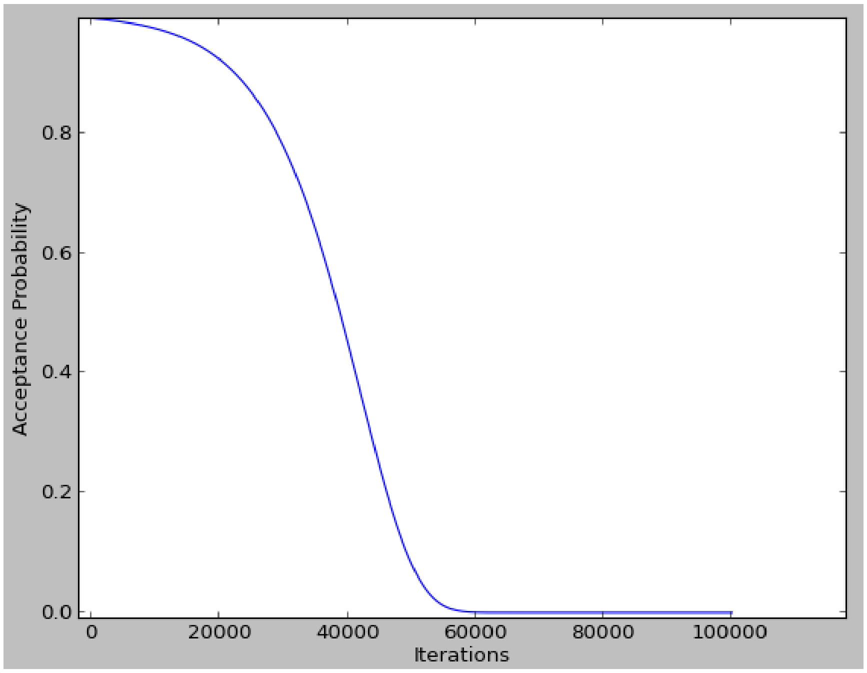


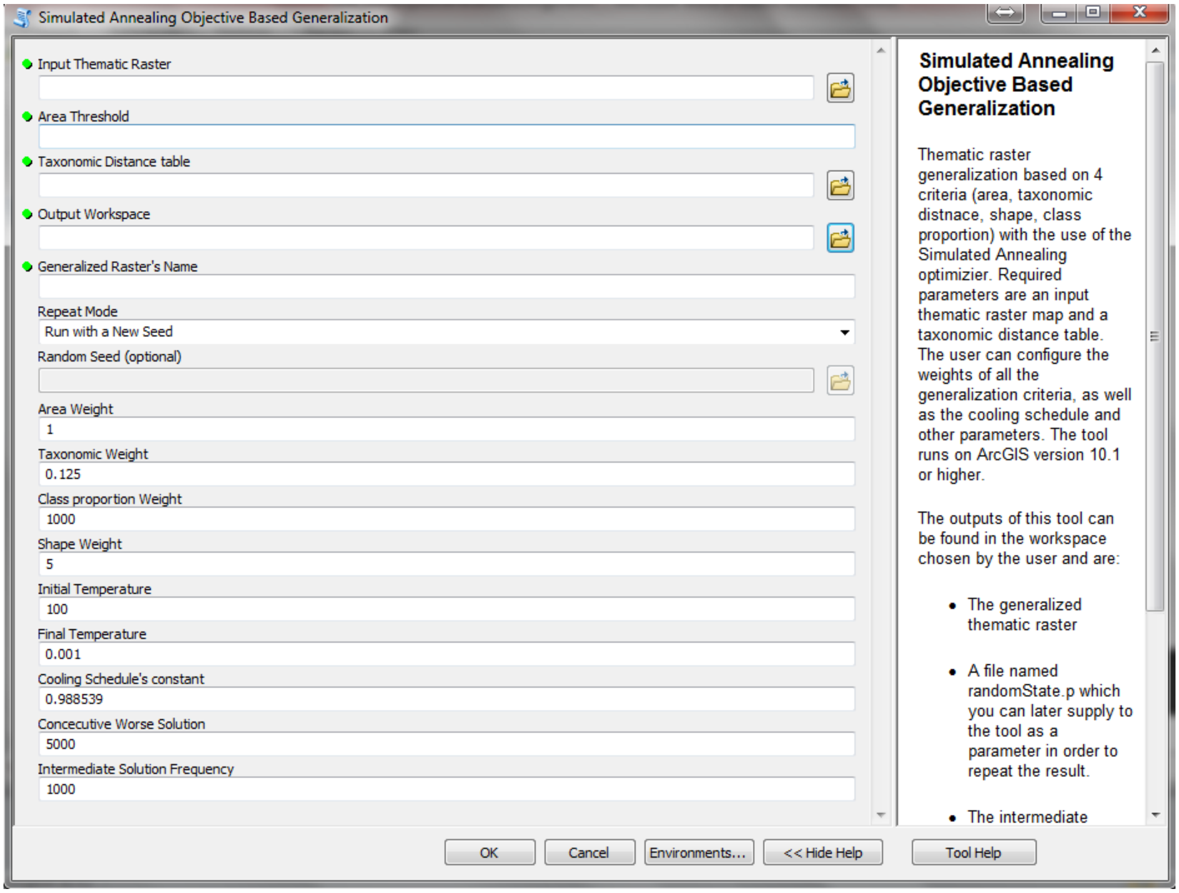
Figure 4.7: Acceptance probability against the iterations for the parameters chosen for the application on the SoilGrids1km

## 4.4 Graphical User Interface

An ArcGIS script tool was created from the developed algorithm that serves as a graphical user interface (GUI). The script tool runs in ArcGIS version 10.1 or higher, since it makes use of the "Polygon Neighbors" tool, which can only be found in those software versions. The tool accepts an initial raster categorical map and a table of the taxonomic distance of the classes and returns 4 different outputs, in the specified workplace.

- The generalized map in GeoTIFF format.
- An image of the plot of the energy of the system against the number of iterations.
- Intermediate solutions of the system with an interval chosen by the user.
- A file that stores the initial state of the random seed at the time of execution. This file can be provided to the tool in another run, in order to repeat the (random) results.

The user can configure all individual cost weights, as well as the parameters regarding the cooling schedule and the frequency of the intermediate solutions. Figure 4.8 presents the GUI of the developed methodology. At http://www.geo-informatie.nl/bruin004/Thanos/SAGeneralizationTool.zip can be found 1. the generalization tool, 2. the two SoilGrids1km datasets used and 3. the table with the taxonomic distances of the WRB classification system. The python code is also provided in the appendix.



Simulated Annealing Objective Based Generalization

Input Thematic Raster

Area Threshold

Taxonomic Distance table

Output Workspace

Generalized Raster's Name

Repeat Mode: Run with a New Seed

Random Seed (optional)

Area Weight: 1

Taxonomic Weight: 0.125

Class proportion Weight: 1000

Shape Weight: 5

Initial Temperature: 100

Final Temperature: 0.001

Cooling Schedule's constant: 0.988539

Concecutive Worse Solution: 5000

Intermediate Solution Frequency: 1000

**Simulated Annealing Objective Based Generalization**

Thematic raster generalization based on 4 criteria (area, taxonomic distnace, shape, class proportion) with the use of the Simulated Annealing optimizier. Required parameters are an input thematic raster map and a taxonomic distance table. The user can configure the weights of all the generalization criteria, as well as the cooling schedule and other parameters. The tool runs on ArcGIS version 10.1 or higher.

The outputs of this tool can be found in the workspace chosen by the user and are:

- The generalized thematic raster
- A file named randomState.p which you can later supply to the tool as a parameter in order to repeat the result.
- The intermediate

Figure 4.8 Graphical user interface of the developed generalization algorithm

# 5. SYNTHETIC DATASETS

The generalization algorithm was first applied on two 10x10 synthetic datasets, in order to gain a general idea of how the algorithm performs in practice and potentially to improve its conceptual design, before the actual application on the SoilGrids1km product. First an overview of the synthetic datasets is given (Section 5.1) and then the results of the application on those datasets are presented (Section 5.2).

For all the experiments presented, the generalization algorithm penalized super-patches smaller than five pixels or in other words, the area generalization threshold is five pixels. The objective function used to assess the energy (cost) of each solution is presented in Equation 9. The only difference among the experiments is the cooling schedule's parameters, which result in different number of iterations.

$$Cost = 1 \text{ x } Area_{cost} + 1 \text{ x } Shape_{cost} + 0.25 \text{ x } Taxonomic_{cost} + 5 \text{ x } Classes\ Proportion_{cost} \tag{9}$$

## 5.1 Datasets Overview

Figure 5.1 presents the two 10x10 synthetics datasets. The first dataset contains 15 individual patches of pixels (11 < area threshold), while the second one has 35 patches (28 < 5 pixels). In both the datasets 4 different classes exists. The fact that the second dataset contains more than double number of patches than the first one, results in a larger number of potential solutions, thus a bigger solution space. Therefore, more iterations may be required for a near-optimum solution to be found.

The taxonomic distance chosen for the synthetic classes is presented in Tables 5.1 and 5.2, for the first and the second datasets, respectively. Different taxonomic distances were chosen between classes in order to investigate the effect of the taxonomic distance on the output.

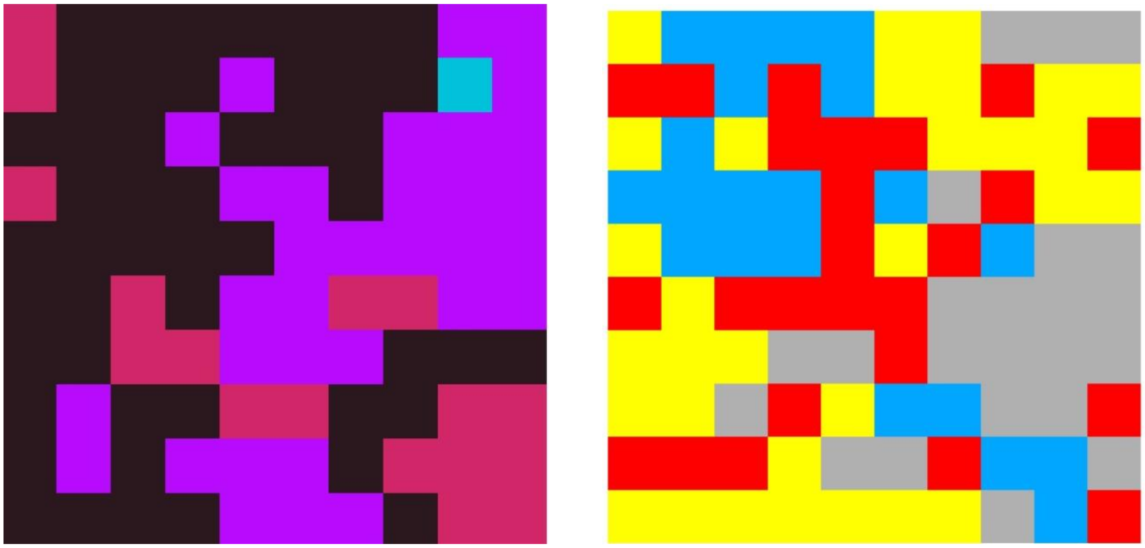
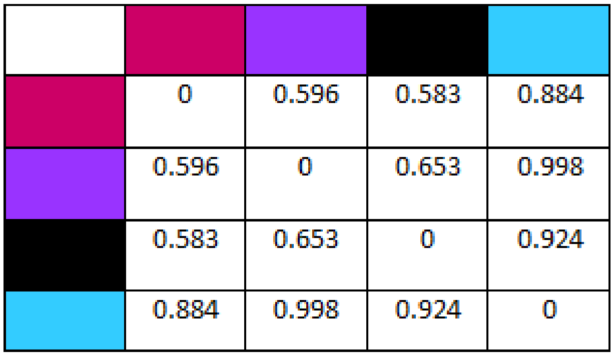


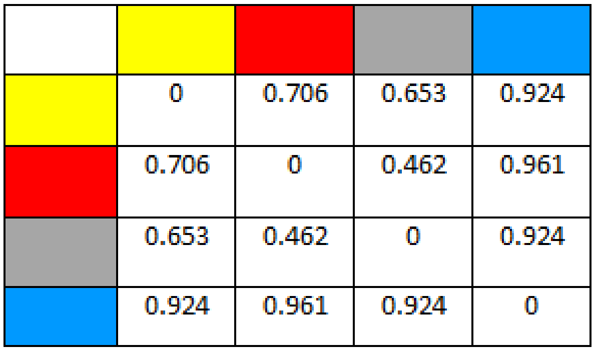
Figure 5.1a: First synthetic dataset (left), 5.1b: Second synthetic dataset (right). Colors refer to different classes.

Table 5.1: Taxonomic distance of the synthetic classes. First dataset



| | 🟥 | 🟪 | ⬛ | 🟦 |
|---|---|---|---|---|
| 🟥 | 0 | 0.596 | 0.583 | 0.884 |
| 🟪 | 0.596 | 0 | 0.653 | 0.998 |
| ⬛ | 0.583 | 0.653 | 0 | 0.924 |
| 🟦 | 0.884 | 0.998 | 0.924 | 0 |

Table 5.2: Taxonomic distance of the synthetic classes. Second dataset



| | 🟨 | 🟥 | ⬜ | 🟦 |
|---|---|---|---|---|
| 🟨 | 0 | 0.706 | 0.653 | 0.924 |
| 🟥 | 0.706 | 0 | 0.462 | 0.961 |
| ⬜ | 0.653 | 0.462 | 0 | 0.924 |
| 🟦 | 0.924 | 0.961 | 0.924 | 0 |

## 5.2 Application on the Synthetic Datasets

The developed generalization methodology was applied on the synthetic datasets three different times, for different number of iterations. That was done to investigate the performance of the algorithm for different number of iterations and inputs with different level of complexity. More precisely, the objective based algorithm was applied once for 500, once for 1000 and once for 2000 iterations, on each dataset. A second stopping criterion was not used. The final energy found in each run, as well as the required time are presented in Table 5.3.

In all three runs with the first dataset the same solution was retrieved. This suggests that we may have reached the global optimum (best possible solution). However, this is not the case for the second synthetic dataset. In this example the energy of the solutions still improved with the number of iterations, due to the bigger solution space.

The second datasets also required slightly more time than the first one. More precisely, the first dataset on average ran at 19 solutions per second, while for the second 18 potential solutions per second were explored. This small time difference is caused by the larger number of initial patches in the second dataset. An additional reason could be that, in the first dataset, during the last iterations the optimum has already been found and no more changes are accepted, which leads to reduced computational time.

Selections of intermediate solutions of the 500 iteration experiment for both synthetic datasets are presented in Figures 5.2 and 5.3. These figures can be viewed together with Figure 5.4, where a graph of the energy of the system against the number of iterations is plotted, for the same experiment.

Table 5.3: Solutions energies and time required for both the synthetic datasets for 500, 1000 and 2000 iterations.

| | **Iterations** | | |
|---|---|---|---|
| | 500 | 1000 | 2000 |
| *Synthetic Dataset 1* | | | |
| Best Solution's Cost | 2.94 | 2.94 | 2.94 |
| Time (Sec) | 27 | 52 | 102 |
| *Synthetic Dataset 2* | | | |
| Best Solution's Cost | 4.65 | 4.39 | 3.97 |
| Time (Sec) | 30 | 54 | 109 |

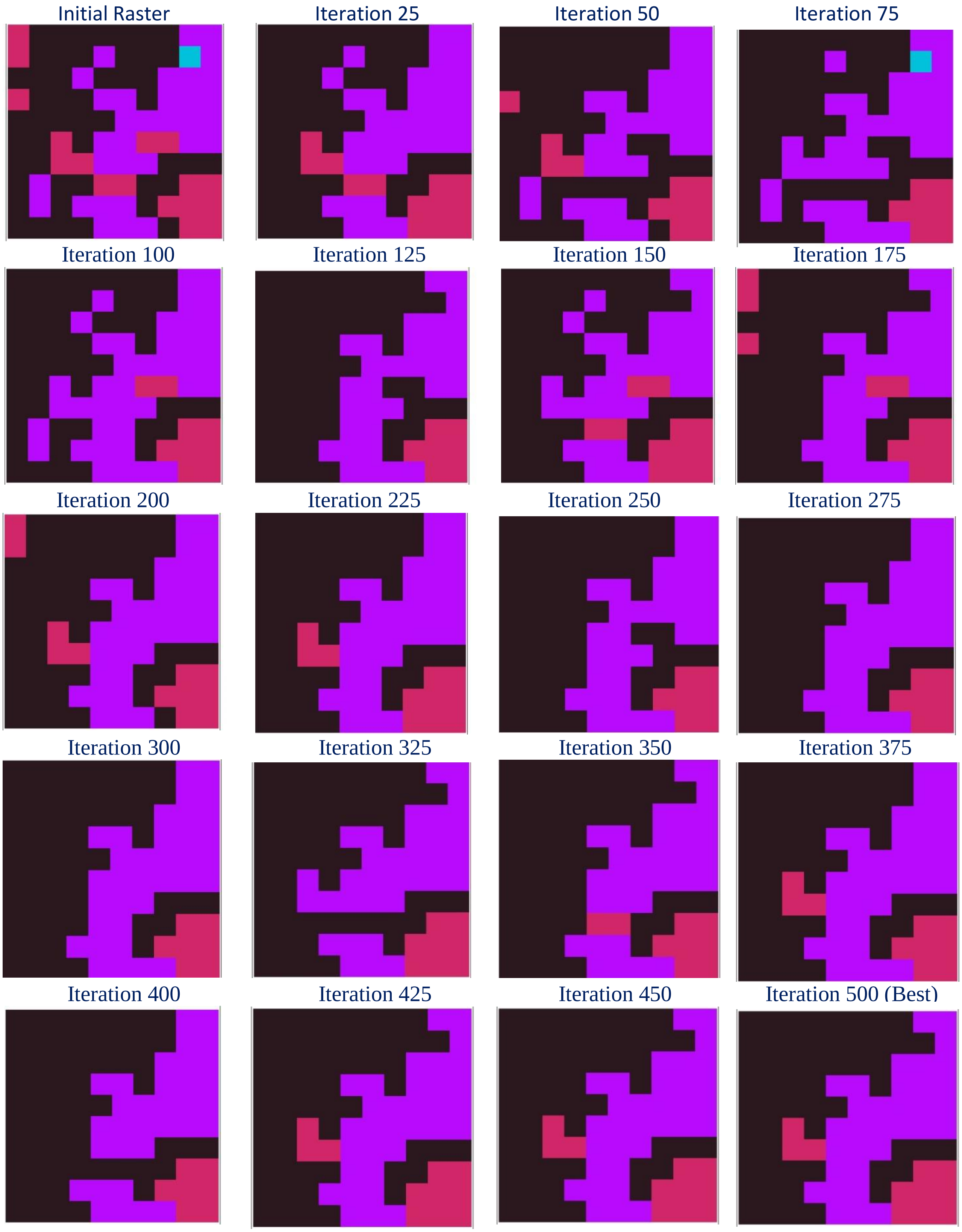


**Figure 5.2: Initial, final and a selection of intermediate solutions of the first synthetic dataset, for 500 iterations.**

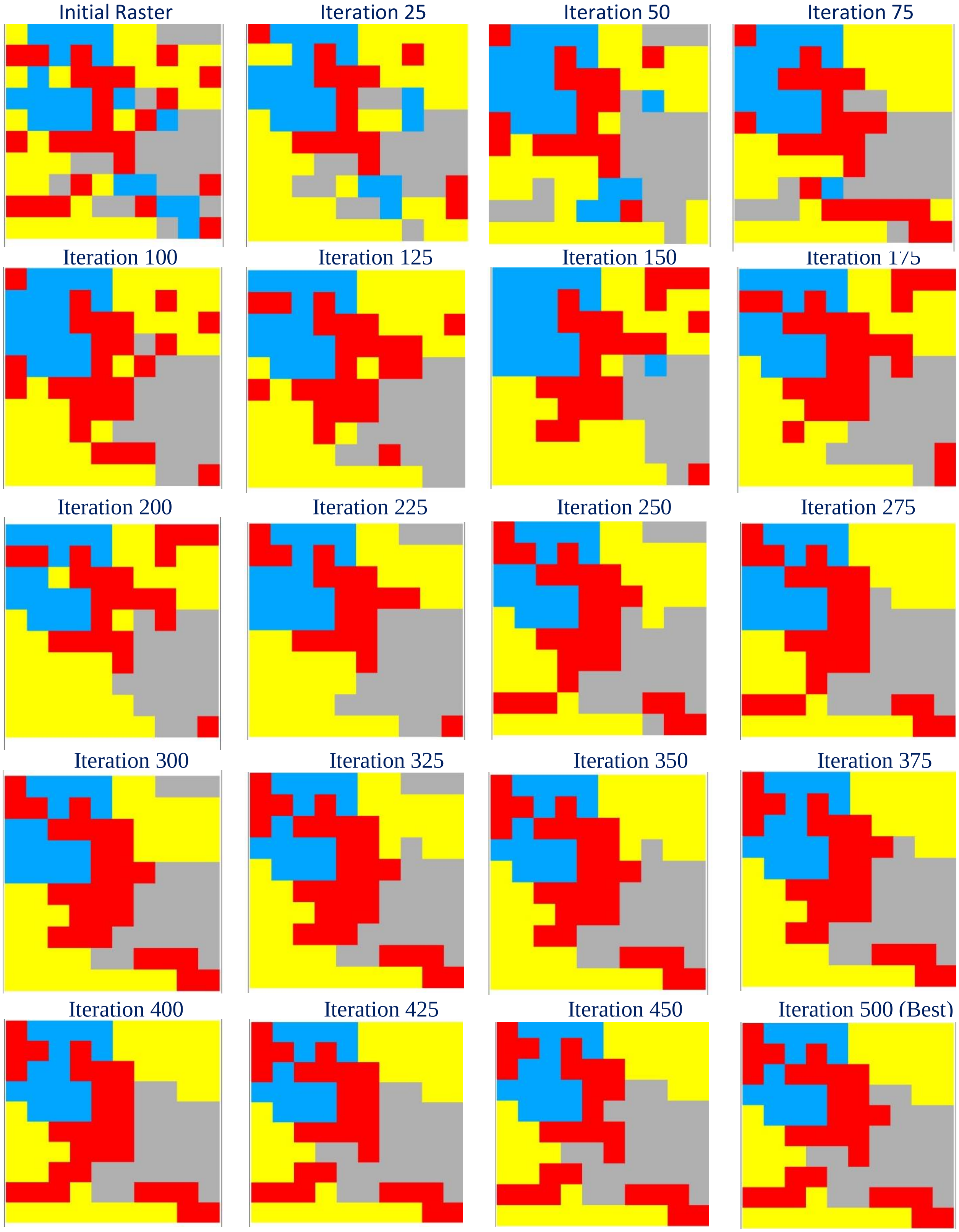


**Figure 5.3: Initial, final and a selection of intermediate solutions of the second synthetic datasets for 500 iterations.**

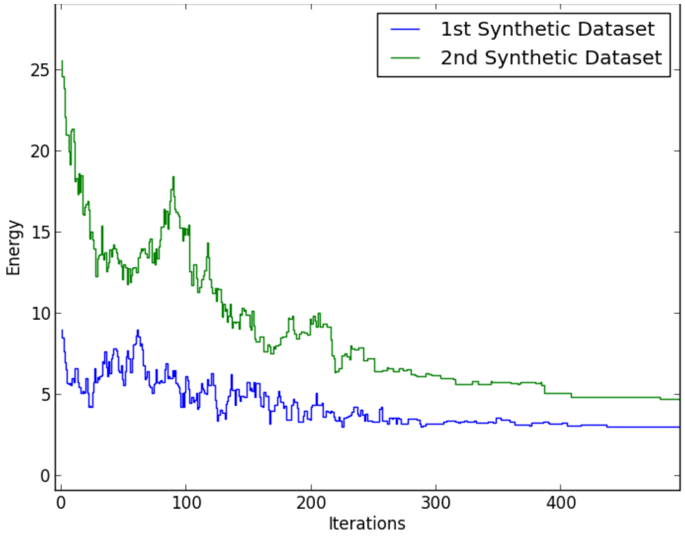


Figure 5.4: The energy of the system during 500 iterations, for the two synthetic datasets

Finally, in Figures 5.5 and 5.6 the results of the criterion-based generalization methodology on the synthetic datasets are visually compared with those of the generalization with the commercial generalization procedure. One can see that the two results differ significantly.

In both cases the developed algorithm outperforms the commercial one, in terms of value of the objective function. More specifically, for the first synthetic dataset, the solution of the developed algorithm scored 2.94, while the solution of the commercial methodology had a cost of 4.31. For the second and more complex synthetic example, the solutions costs were 3.97 and 5.12, respectively.

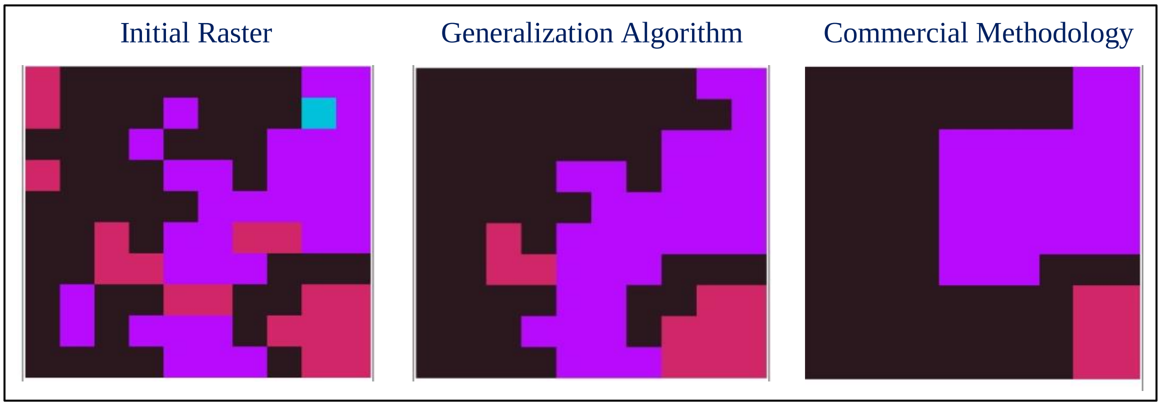


Figure 5.5: Best result found of the criterion-based algorithm in contrast with the result of the commercial methodology, for the first synthetic data.

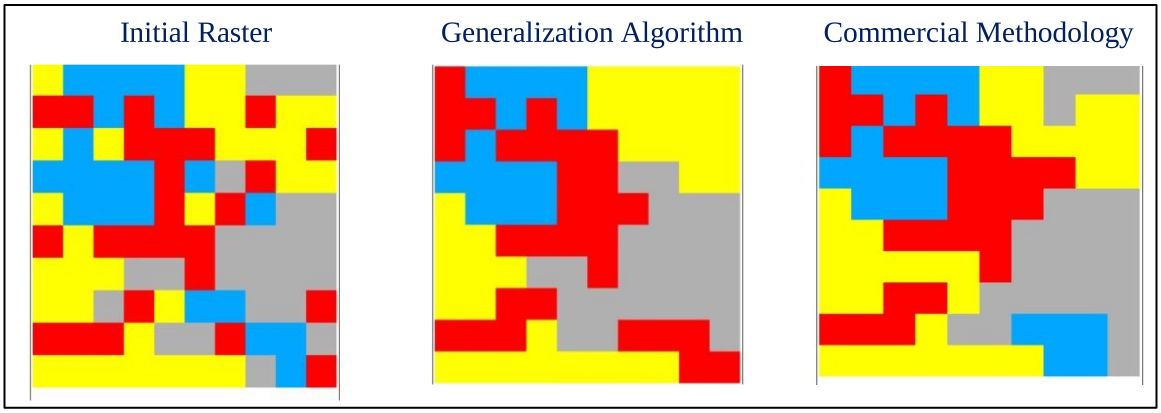


**Figure 5.6: Best result found of the criterion-based algorithm in contrast with the result of the commercial methodology, for the second synthetic data.**

# 6. WEIGHTS CONFIGURATION

## 6.1 Description of Weights Configuration Procedure

The selected generalization criteria penalize different properties of the map. The area and shape criteria apply to individual patches and their total cost equals the sum of the partial costs of all the patches of a raster. On the other hand, the class proportions criterion applies to each class and the semantics cost is evaluated at pixel level. Thus, defining the weights of each individual criterion is a challenging task. With the aim to get a reasonable set of weights, different runs were performed with different weighs configuration.

The experimentation was done only for the high complexity area and the first level of generalization (5 pixels). The maximum number of iterations was set to 50,000. At the end of the experimentation process, the weighted objective function used in the application on SoilGrids1km for the first generalization level, was derived. Next, the derived weights were adjusted to apply to the other test area and generalization levels.

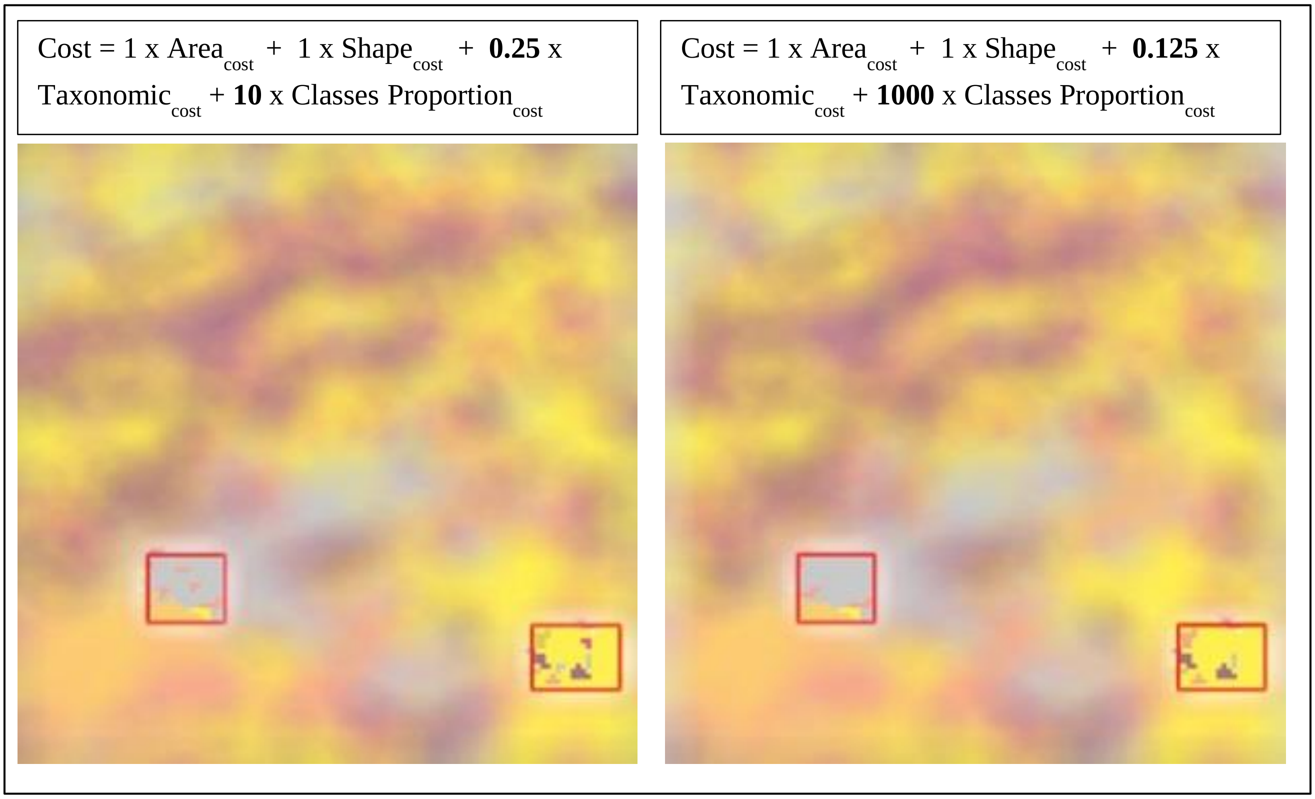
In this chapter the results with different criteria weights are presented blurred. Emphasis is given only in specific sub-areas where the sensitivity of the results to the change of specific weights is visible. The actual results are presented in Chapter 7.

## 6.2 Presentation of Different Experiments

In every step of the algorithm, a patch is randomly selected and reclassified to a neighbor class. Imagine a patch of 2 pixels size, in the first generalization level (5 pixels area threshold), in its initial state. To this patch corresponds 0.75 area (weightless) cost.  When this patch gets reclassified to the value of its neighbor, the area cost decreases (or becoming zero), while semantics cost is introduced. For taxonomic distance between the two classes equals 0.5 (average value), the semantics cost for the reclassified patch is 0.5 x 2 pixels = 1, which is bigger than the initial cost. Thus, with equal weights applied, the latter solution is considered by the algorithm worse than the initial, which is something that is not desired.

To deal with that issue, at the first experiment a weight value of 0.25 was introduced to the semantics criterion. Additionally, due to the fact that the class proportions cost has maximum value of 1, for the whole map, its total cost was multiplied by 10. One thing worth noting about the produced result is that it contains enough patches below the area threshold (Figure 7.1a). In addition, the class proportions cost, even though multiplied by a factor of ten, is responsible for only 0.02% of the total cost.

In order to decrease the number of patches below the threshold produced and make the class proportions criterion to play a bigger (but not big) role in the output, the taxonomic distance weight was decreased to 0.125 and the weight of the proportions of the classes was increased to 1000. Figure 7.1b presents the results for that experiment, in comparison with the previous figure, where one can focus in specific sub-areas of the two outputs. The latter weight configuration resulted in approximately 12% less small patches, than the previous one.

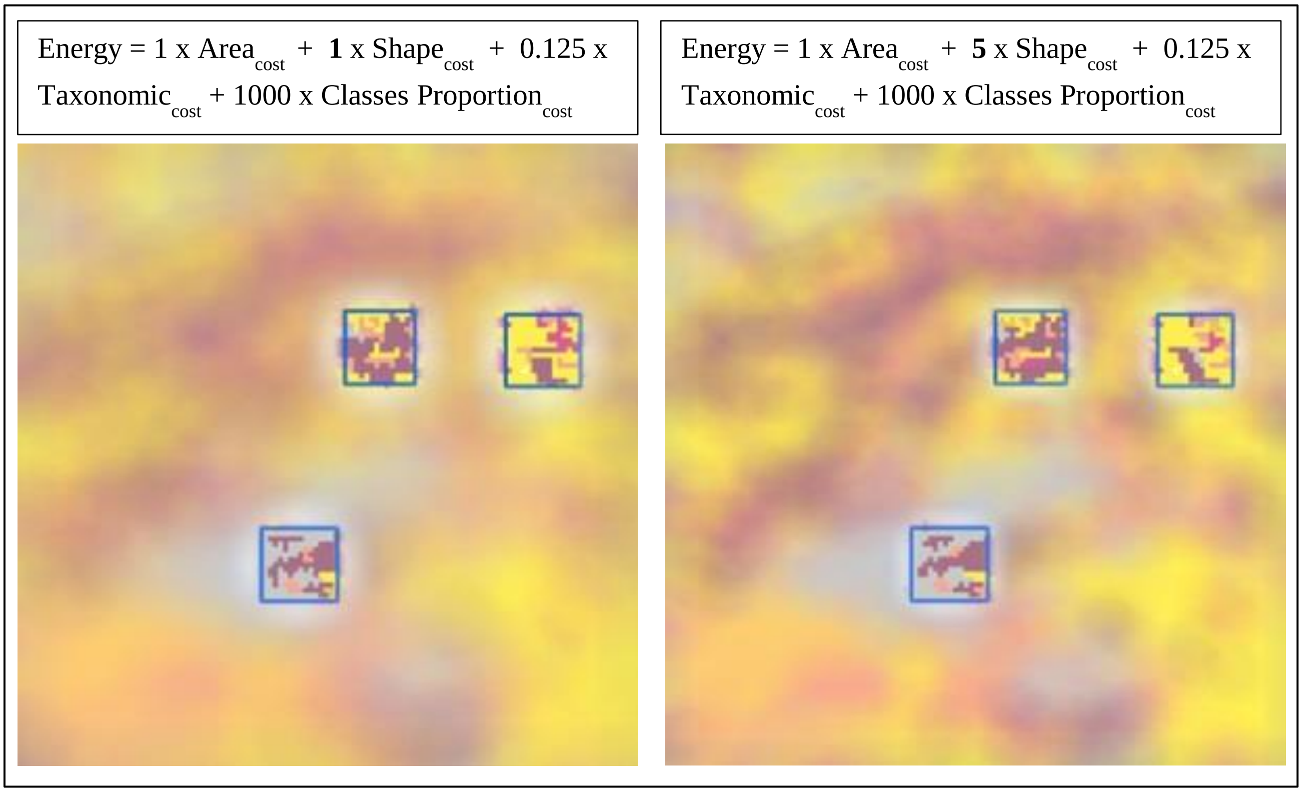


$\text{Cost} = 1 \text{ x } \text{Area}_{\text{cost}} + 1 \text{ x } \text{Shape}_{\text{cost}} + \mathbf{0.25} \text{ x } \text{Taxonomic}_{\text{cost}} + \mathbf{10} \text{ x } \text{Classes Proportion}_{\text{cost}}$

$\text{Cost} = 1 \text{ x } \text{Area}_{\text{cost}} + 1 \text{ x } \text{Shape}_{\text{cost}} + \mathbf{0.125} \text{ x } \text{Taxonomic}_{\text{cost}} + \mathbf{1000} \text{ x } \text{Classes Proportion}_{\text{cost}}$

**Figure 6.1a (Left) and 6.1b (right): Generalized results with different weighted objective functions.**

Despite shape criterion, in both the experiments a significant amount of long -sliver patches exist. Figure 6.2 presents the third experiment, where the weight of the shape criterion was multiplied by a factor of 5. The rest of the criteria remained identical with the previous configuration. The figure is presented side by side with the results of the previous experiment, to highlight differences in specific sub-areas. In general, even though long patches still exist in the result, they are significantly less than the previous experiments. Additionally, the shape weight re-configuration resulted in simplification of the shape of many patches. The weights of the last experiment are the ones used in the SoilGrids1Km application, for the high complexity area and first level of generalization.



Energy = 1 x $\text{Area}_{cost}$ + **1** x $\text{Shape}_{cost}$ + 0.125 x $\text{Taxonomic}_{cost}$ + 1000 x Classes $\text{Proportion}_{cost}$

Energy = 1 x $\text{Area}_{cost}$ + **5** x $\text{Shape}_{cost}$ + 0.125 x $\text{Taxonomic}_{cost}$ + 1000 x Classes $\text{Proportion}_{cost}$

Figure 6.2: 3rd Sensitivity of the results with changed the weight of the shape criterion

# 7. APPLICATION ON SOILGRIDS1KM

The developed generalization algorithm was applied to two different subsets of the SoilGrid1km product, as presented in Section 2.4. The two 120 x 120 pixel sections were chosen to demonstrate behavior of the generalization at different degree of map complexity.

The generalization took place at three different levels, with area threshold values of 5, 17 and 65 pixels for the minimum mapable unit. With a pixel size of 930m, according to Wambeke et al. (1986), these areas correspond to map scales of approximately 1 : 329,000, 1 : 606,000 and 1: 1,186,000, respectively.

Table 7.1 presents the weights of the partial costs for each generalization level and both test areas. At each level, the weights of the costs are identical, with the exemption of the class proportions weight, which was set to 100 for the area with low complexity, while it was set to 1000 for the area with higher complexity.

As regards the stopping criteria, in all runs the algorithm was scheduled to stop after a maximum of 100,000 iterations or 5,000 consecutively non-accepted solutions. Additional runs were executed at the first generalization level (5 pixels) for both the test areas, in order to see if the results of different runs are similar or not. Furthermore, experiments were performed also for 200,000 iterations, to investigate if and how much the cost decreases with additional iterations. The results of the additional runs are presented in Section 7.4.2.

In this chapter, first the generalization results are presented, for the two test areas at all three generalization levels (Section 7.1, Section 7.2) and an analysis of their costs is shown in Section 7.3. Next, the algorithms' performance is addressed (7.4) and finally, the results of the chosen commercial generalization methodology are presented and compared to those of the criterion-based algorithm (7.5).

**Table 7.1: Weights of the costs of the objective function for all 3 levels of generalization for both test areas**

| Generalization Level | **Weights of Partial Costs**<br>Low Complexity Area / High Complexity Area | | | |
|---|---|---|---|---|
| | Area | Shape | Taxonomic distance | Class Proportions |
| 1st | 1 | 5 | 0.125 | 100;1000 |
| 2nd | 2 | 5 | 0.03125 | 100;1000 |
| 3rd | 4 | 5 | 0.0078125 | 100;1000 |

## 7.1 Test Area 1 – Low Complexity

The low complexity test area (Figure 7.1) contains 1364 individual patches of pixels, of which 1284 are smaller than 5 pixels (1st generalization level). In addition the map contains 958 single pixel patches. This means that for all three area thresholds the majority of the patches has to be reclassified. There are 16 soil classes in the image, which corresponds to half of the classes of the WRB classification system.

Approximately 73.2% of the area belongs to the podzols soil class, 11.3% is occupied by fluvisols while the remaining area is mainly covered by ferralsols and solonetz. Due to their small area (< 0.05 %), kastanozems, solonchaks and vertisols are potential candidates for elimination from the generalized result.

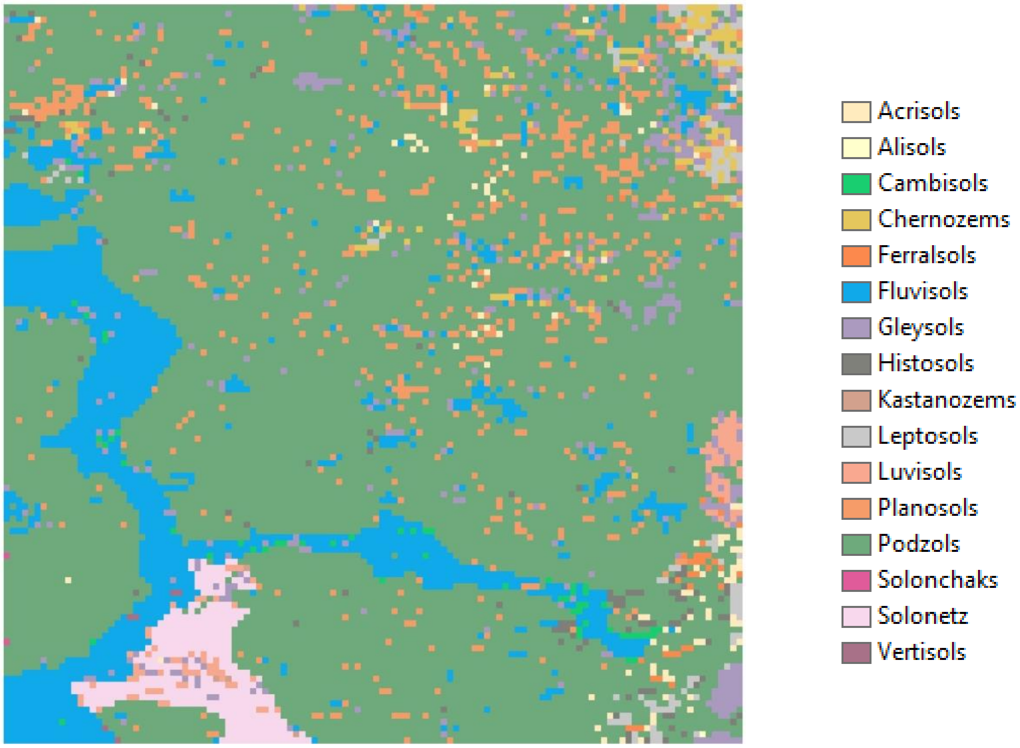


Figure 7.1: Test area 1 – low complexity area. Initial soil map

### 7.1.1 Generalization Level 1

At the first level of generalization, the initial raster soil map was generalized with an area threshold value of 5 pixels and with a weighted objective functions as shown in Table 7.1. Figure 7.2 visualizes the process in which, from the initial map and through intermediate solutions, the final generalized result derives. We can see from the figure how the result is getting gradually generalized, by consecutively decreasing the number of individual patches and thus also decreasing the solution's cost.

The best (and last) solution is presented separately, in Figure 7.3. The generalized map still contains 31 patches with area smaller than the area threshold. Only twelve of the 16 initial soil classes were retained in the generalized result. Alisols, kastanozems, solonchaks and vertisols were eliminated.

In Figure 7.4 the energy of the system at the time of the algorithm execution is plotted against the number of iterations. This Figure can be viewed along with Figure 7.2, to identify the energy corresponding to specific iterations. The final energy of the system is 138.59.

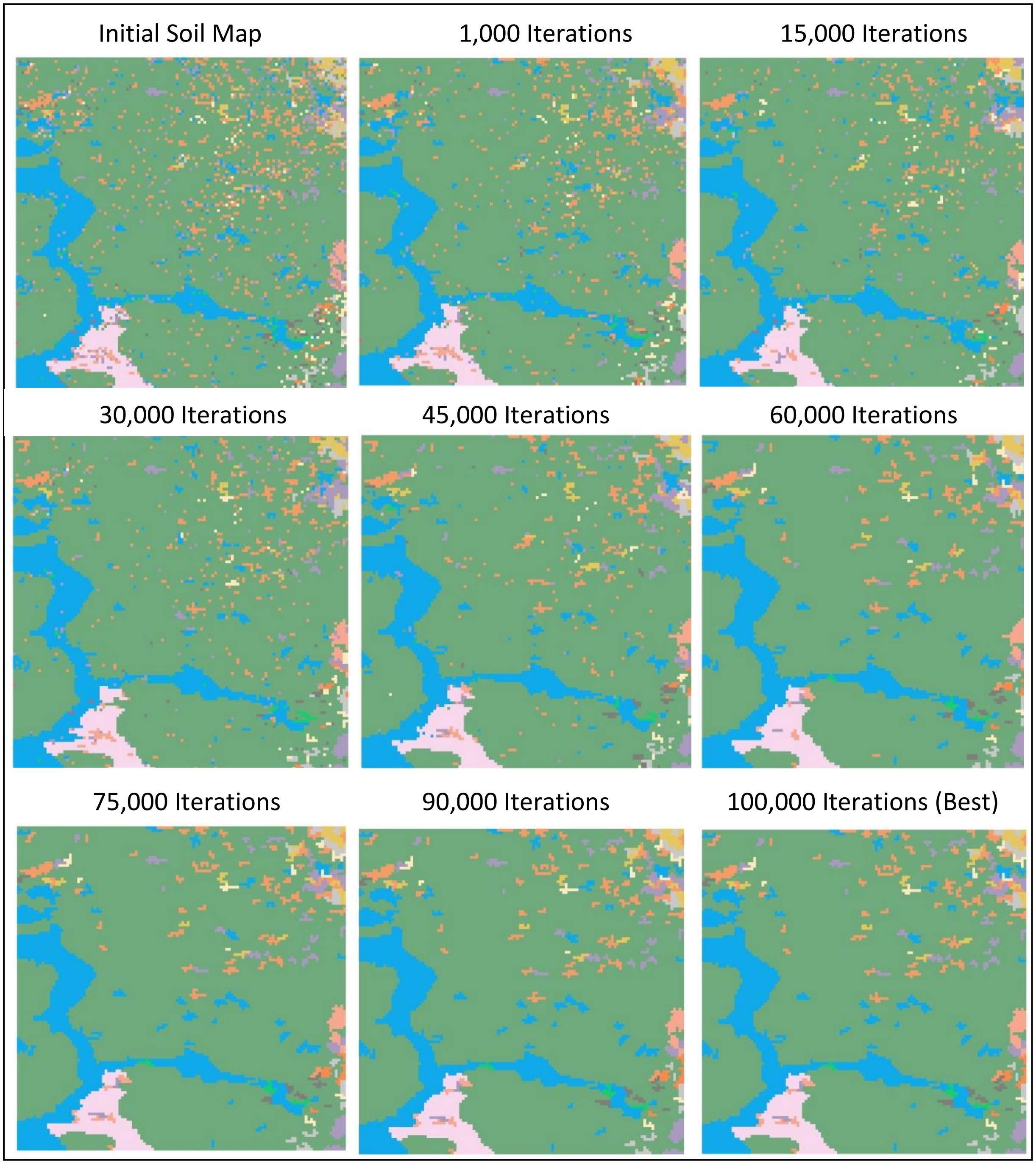


Figure 7.2: From the initial raster map to the generalized result; selected intermediate solutions of the system at various iterations

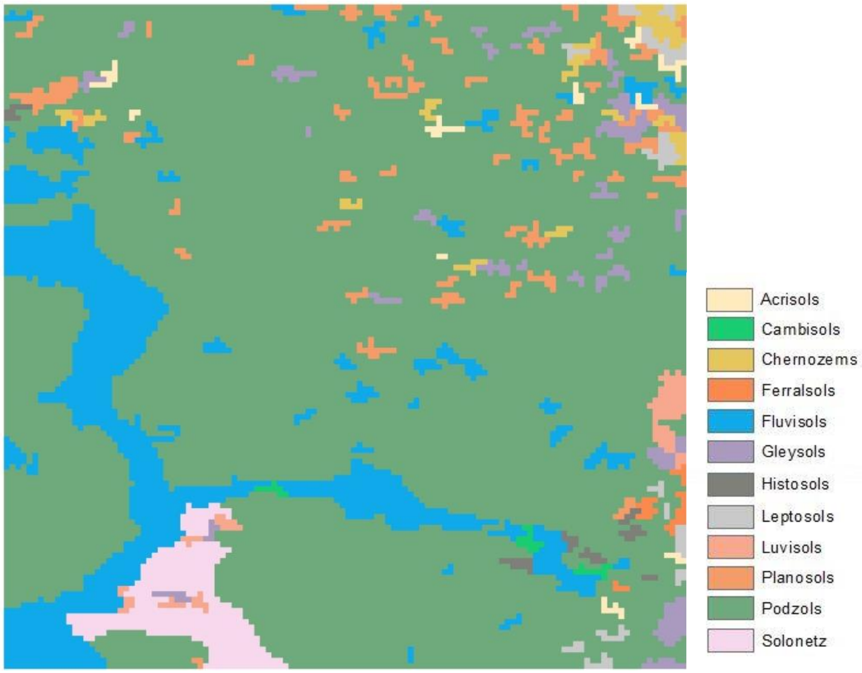


Figure 7.3: The generalized result with area threshold = 5 pixels

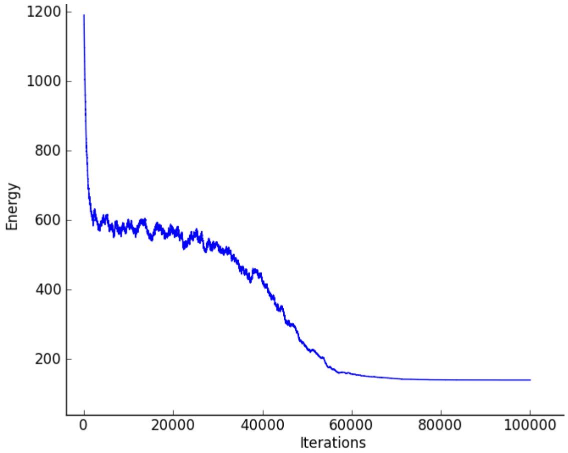


Figure 7.4: Graph of the energy (cost) of the system during the first generalization level

### 7.1.2 Generalization Level 2

At the second level, the initial soil raster map was generalized with an area threshold value of 17 pixels. Due to the use of different cost weights, the energy of the initial map is more than twice that of the first level of generalization. The final energy of the system is 85.33. The generalized output is presented in Figure 7.5.

While the initial map contains 1,364 patches, the generalized result contains only 70 and 30 of them are smaller than the area threshold. The generalized map contains eleven soil classes. The same four classes as the previous generalization level have been eliminated from the result, as well as the cambisols class, which in the initial map occupies an area of 56 pixels.

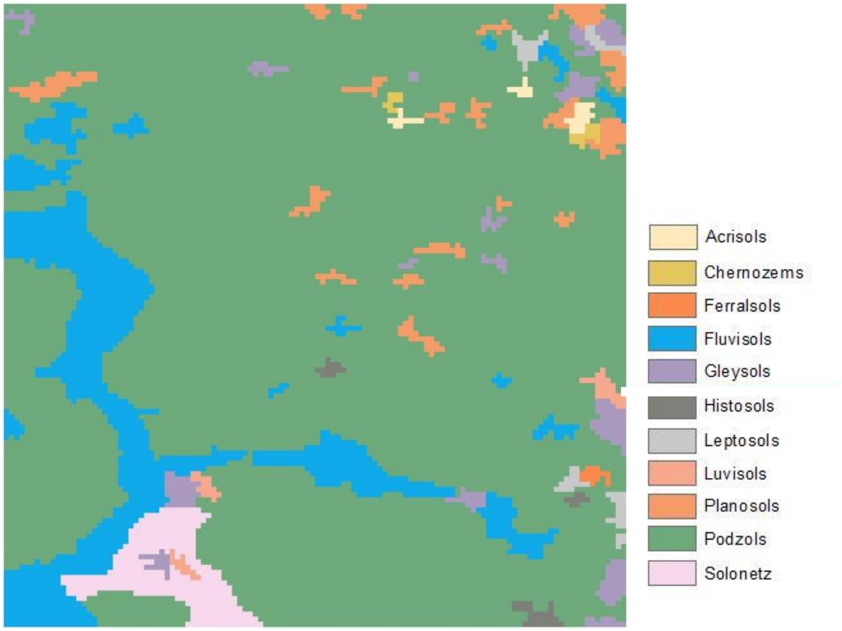


Figure 7.5: The generalized result with area threshold = 17 pixels

### 7.1.3 Generalization Level 3

The area threshold value for the third generalization level is 65 pixels. The output (Figure 7.6) contains 33 patches in total, 21 patches of which are below the area threshold. There are twelve soil classes. The classes that were eliminated are the same as those of the first generalization level. The cambisols class, which was not present in the result of the second generalization level, now exists at the output with an area of 73 pixels. The final energy is 79.73.

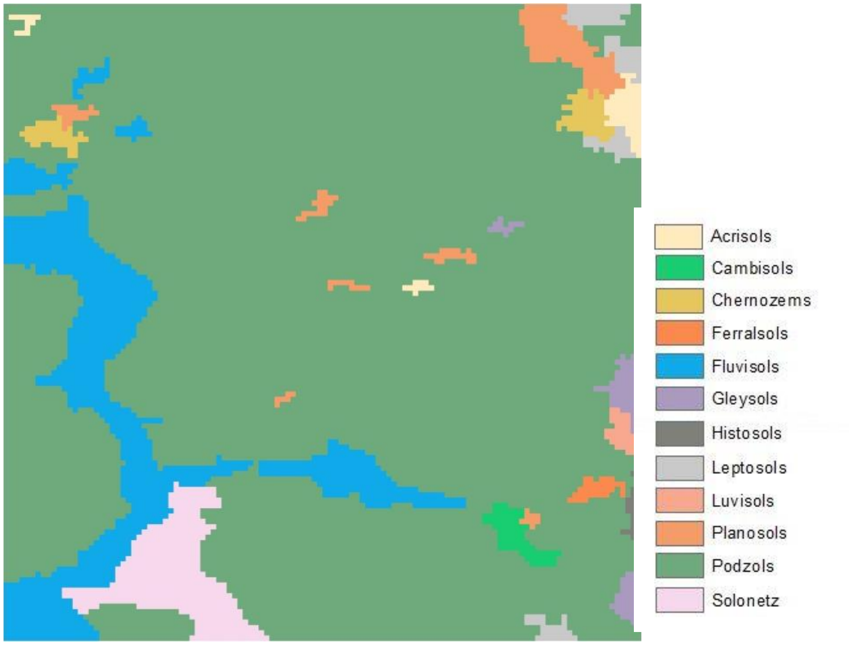


Figure 7.6: The generalized result with area threshold = 65 pixels

## 7.2 Test Area 2 – High Complexity

The second and more complex test area (Figure 7.7) contains 2729 individual patches of pixels, which 2375 of them are smaller than 5 pixels (1st generalization level). Additionally, 1675 single pixel patches exist. Again like the first test area, for all 3 levels of generalization the majority of the patches will be allowed to be reclassified. 8 different soil classes exist in the image with the bigger of them being the calcisols with 5,120 (out of 14,400) pixels and the smaller the fluvisols with only 3 pixels.

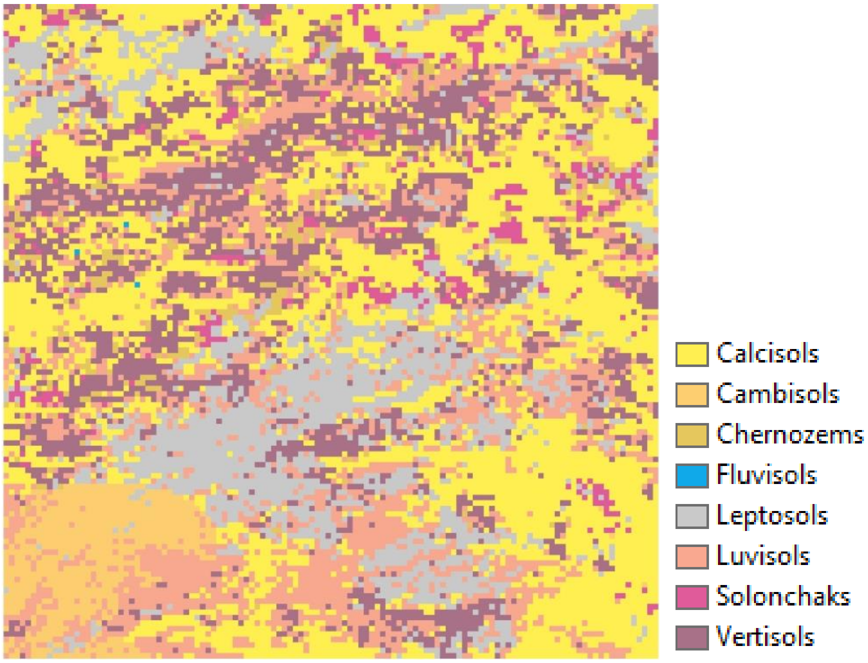


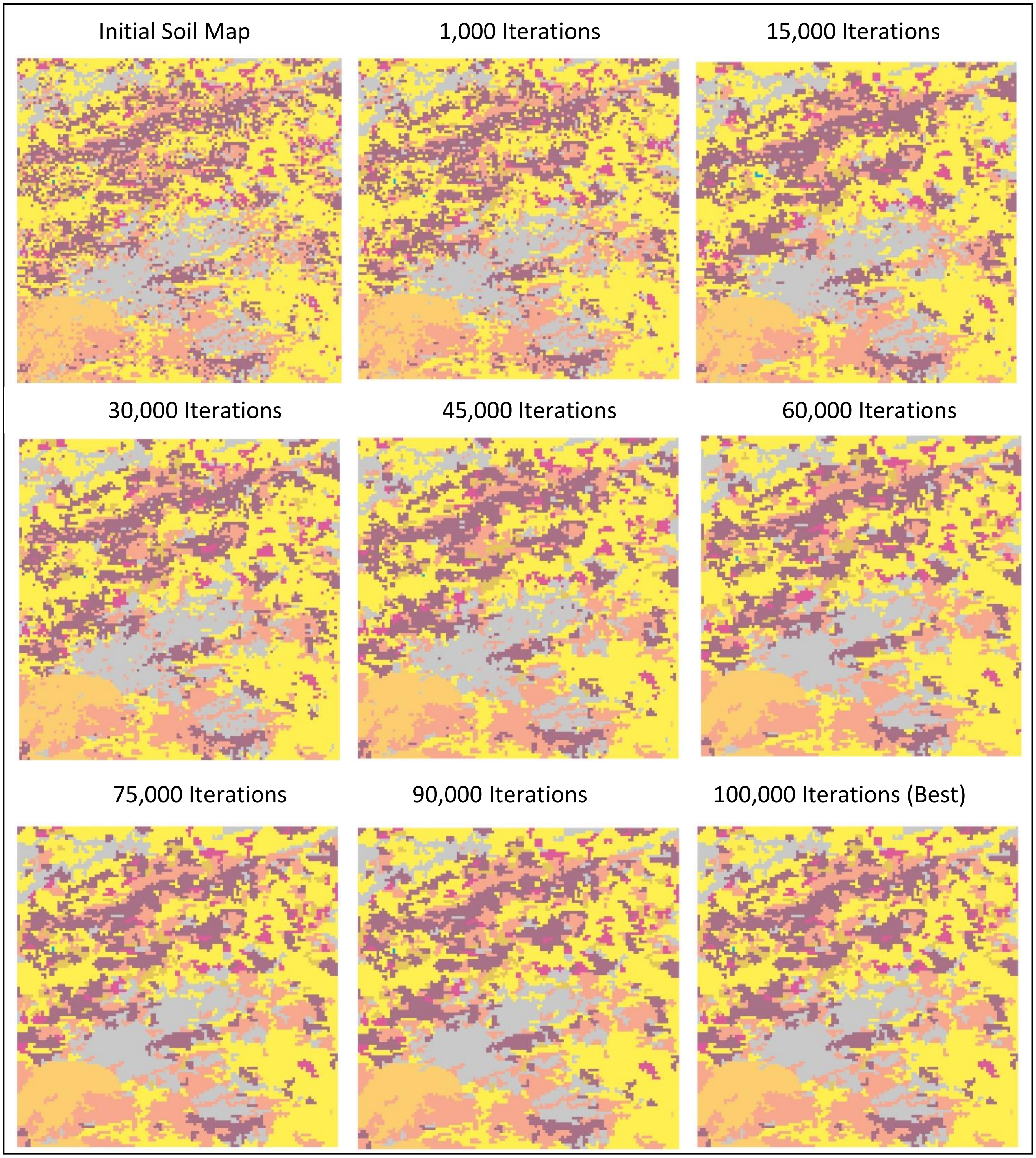
Figure 7.7: Test area 2. High complexity. Initial soil map

### 7.2.1 Generalization Level 1

Test area two was generalized with area threshold of 5 pixels and weighted objective function as presented in Table 7.1. The weights of the objective functions are identical to those used for test area 1, except for the weight of the class proportions which was set to 1000.

Figure 7.8 presents a selection of intermediate solutions; the final generalization result is presented in Figure 7.9. The output contains 465 patches (versus 2,729 in the initial map), of which 39 have area smaller than 5 pixels. The generalized map retained all initial soil classes.

Figure 7.10 presents a graph of the system's energy plotted against the number of iterations. The graph has the same pattern as the graph presented in Figure 7.4. This similarity is discussed in Chapter 8. The initial cost of test area two is around 2,300 and it is gradually lowered to 310.05, which is the energy of the generalized result.



Initial Soil Map

1,000 Iterations

15,000 Iterations

30,000 Iterations

45,000 Iterations

60,000 Iterations

75,000 Iterations

90,000 Iterations

100,000 Iterations (Best)

**Figure 7.8: From the initial raster map to the generalized result; selected intermediate solutions of the system at various iterations**

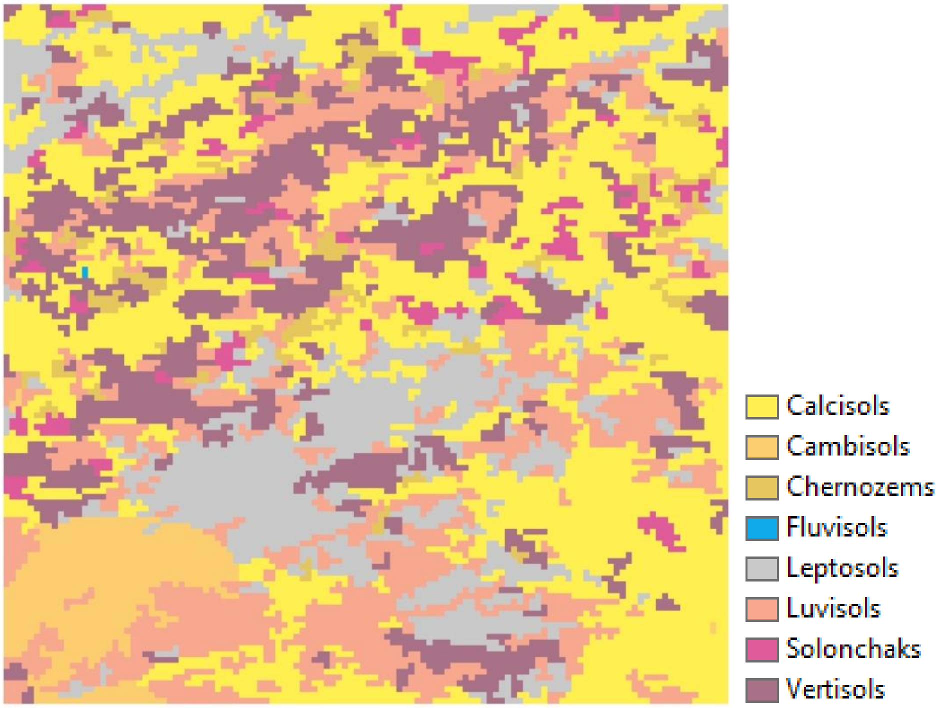


**Figure 7.9: The generalized result with area threshold = 5 pixels**

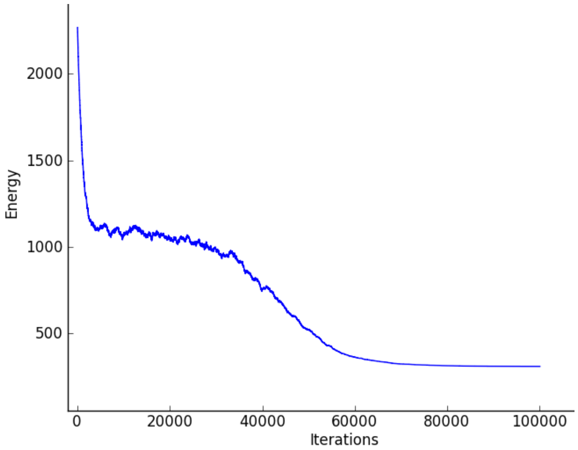


**Figure 7.10: Energy of the system plotted against the number of iterations, for test area 2, generalization level 1**

### 7.2.2 Generalization Level 2

At the second level of generalization the developed algorithm ran with an area threshold of 17 pixels. The generalized output is presented at Figure 7.11. The result contains 211 individual patches of pixels and 54 of them are smaller than the generalization threshold. The energy of the system after 100,000 iterations is 194.31. Seven out of the eight initial soil classes exist in the output, due to the fact that the fluvisols class has been eliminated.

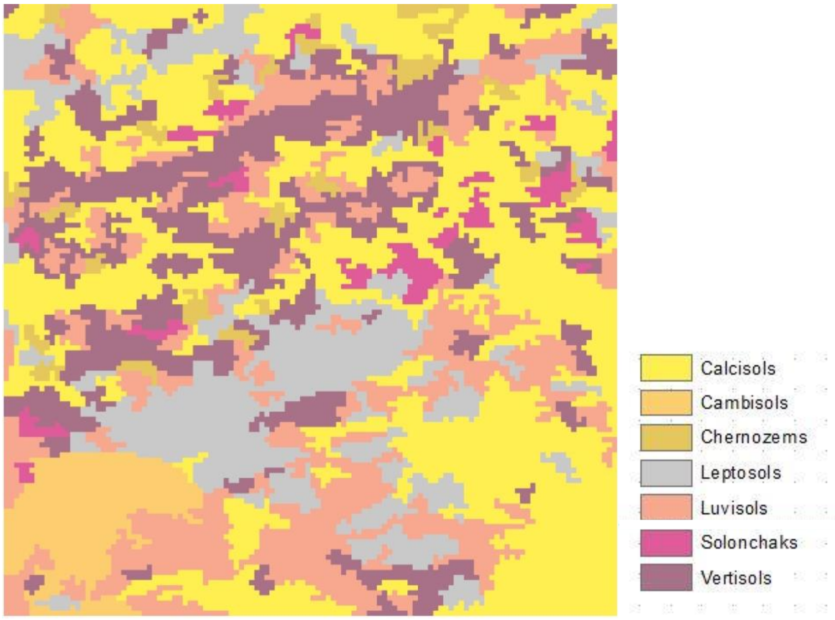


Figure 7.11: The generalized result with area threshold = 17 pixels

### 7.2.3 Generalization Level 3

With an area threshold of 65 pixels, the result of the third generalization level (Figure 7.12) contains 113 patches in total, of which 67 are smaller or equal than the area threshold; 52 patches are between 40 and 64 pixels while 15 are smaller than 40 pixels. Again, the fluvisols class is the only class from the initial soil map that is not present in the generalized result.

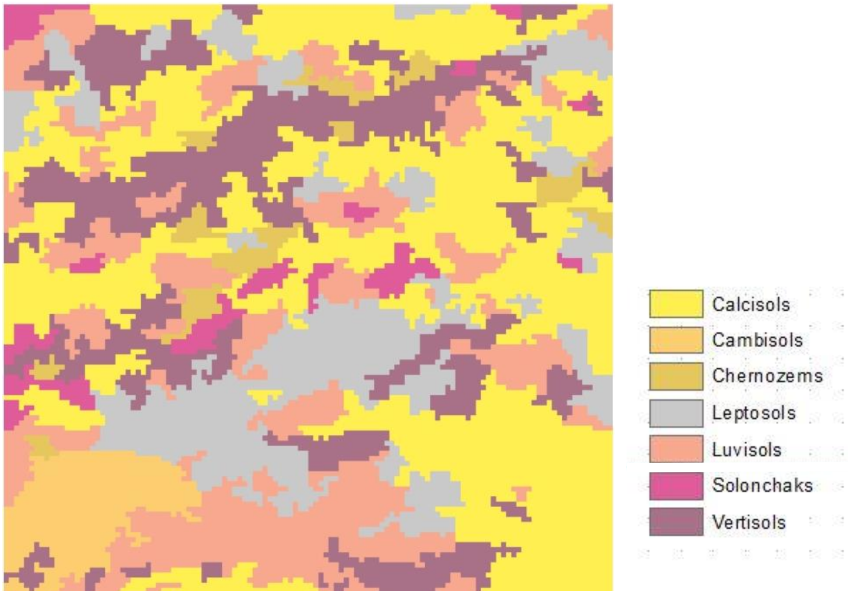


Figure 7.12: The generalized result with area threshold = 65 pixels

## 7.3 Cumulative Results

Tables 7.2 and 7.3 present the weighted partial costs of the generalized results for the two tests areas. For the first level of generalization the taxonomic cost constitutes the main part of the total cost. For the levels two and three however, due to the altered weights, the area cost plays a more significant role.

As it was expected, the number of patches decreases with the level of generalization. However, even after generalization, the maps still contain several patches with area smaller than the area threshold. In most of the cases at least one soil class is eliminated during generalization.

**Table 7.2: Analysis of the weighted partial costs of the results for test area one (number have been rounded to the first decimal).**

| *Test Area 1 - Low Complexity* | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | **Costs** | | | | |
| *Generalization Level* | *Nr of Patches* | *Patches Below Threshold* | *Soil Classes* | *Area* | *Shape* | *Taxonomic distance* | *Class proportions* | *Total* |
| 1st | 186 | 31 | 12 | 12.5 | 10.7 | 110.4 | 4.9 | **138.6** |
| 2nd | 70 | 30 | 11 | 25.1 | 11.3 | 41.3 | 7.6 | **85.3** |
| 3rd | 33 | 21 | 12 | 51.0 | 7.0 | 12.2 | 9.5 | **79.7** |

**Table 7.3: Analysis of the weighted partial costs of the results for test area two (number have been rounded to the first decimal).**

| *Test Area 2 - High Complexity* | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | **Costs** | | | | |
| *Generalization Level* | *Nr of Patches* | *Patches Below Threshold* | *Soil Classes* | *Area* | *Shape* | *Taxonomic distance* | *Class proportions* | *Total* |
| 1st | 465 | 39 | 8 | 16 | 72.9 | 219.3 | 1.7 | **310.0** |
| 2nd | 211 | 54 | 7 | 38.8 | 56.5 | 95.6 | 3.5 | **194.3** |
| 3rd | 113 | 67 | 7 | 155.5 | 55.3 | 36.4 | 0.2 | **247.5** |

## 7.4 Performance

### 7.4.1 Processing Time

The goal of every optimization algorithm is to find a good enough solution within reasonable time. The processing time depends directly on the size of the solution space, since the more potential solutions the more time is required to find a near-optimum one.

The developed methodology ran on a desktop computer (processor: Intel i7@3.5GHz, Ram: 16GB). It is concluded that the processing time depends on the size of the input raster and the generalization area threshold. The complexity of the inputs seems to play a minor role, as well. For instance, at the application on the 10x10 synthetic datasets (Section 5.2) for an area threshold of 5 pixels, the algorithm explored approximately 18.5 solutions per second. When the algorithm was applied to the two SoilGrids1km test areas, however, it required much more time. For both the test areas the average time needed for 100,000 iterations was: 9 hours for the first generalization level, 11 hours for the second and 12 hours for the third one. Therefore, the algorithm explored 3.09, 2.77 and 2.53 solutions per second, respectively.

That significant difference at processing time among the synthetic datasets and the real world test areas, is caused by the different size of the inputs (10x10 the first and 120x120 pixels the second). Usually, more pixels mean more initial patches, making the variables (lists) that the algorithm uses to store the data more "heavy", which leads to decreased performance. For example, the two synthetic examples contain 15 and 35 initial patches, while the two real-world test areas 1,364 and 2,729. An additional reason for the difference in speed, is that as the algorithm runs the super-patches are getting bigger (containing more patches), making in this way the cost evaluation more time consuming.

The latter is also the reason of the small difference in processing time between different generalization levels. The bigger the area threshold becomes, so is the eventual size of the super-patches. In principal, the super-patches at the third generalization level would contain more individual patches than the super-patches of the first one. More complex super-patches mean that when the energy of the super-patch is getting evaluated/updated, not only more individual patches must be taken into account but also all of their neighbors (for the evaluation of the shape cost). Therefore, more complex super-patches require in general more time for their cost to be evaluated. It was observed that during the very first iterations, when the super-patches are still small in size, the algorithm runs faster, than later when the super-patches have grown.

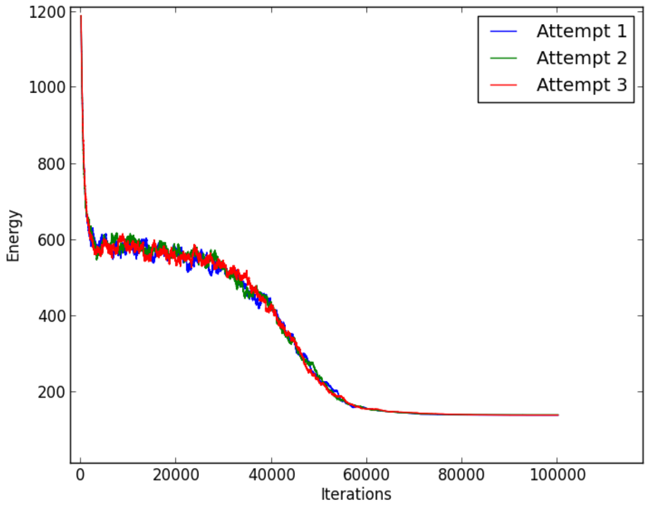
### 7.4.2 Stability of Results

In order to evaluate stability of results, two more runs were performed for the first generalization level (5 pixels) for both the test areas, with the same number of iterations and identical parameter values as the initial run of the application, but with different initializations of the pseudo random number generator. Furthermore, for the more complex test area, the algorithm was repeated three times, with different values for the cooling schedule parameters, which resulted to a maximum number of 200,000 iterations. The experiments with more iterations took place in order to investigate if and how much the solution cost is decreased, by increasing the number of iterations.

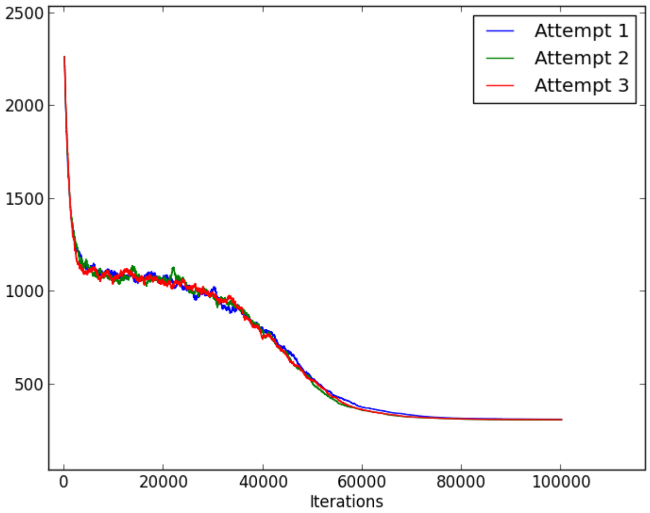
Table 7.4 presents the energy/cost values of the generalized maps, for the first generalization level, for all algorithm's executions, including the initial application. For the first, low complexity area no better solutions were retrieved during the two additional runs, as attempt 2 scored 139.5 and attempt 3 has a cost value of 138.7. For the second, high complexity test area one of the two additional runs outperformed the initial one, while the other found a solution similar to the initial. One can see that the results produced by the developed algorithm, for identical number of iterations, are stable. Figures 7.15 and 7.16 present a graph of the energy of the system plotted against the number of iterations, for all 3 attempts, for both the test areas. The similar pattern of the graphs is discussed on Chapter 8.

**Table 7.4: Cost of the results for all additional runs of the algorithm, at the first level of generalization (5 pixels).**

| *Runs* | *Area 1: 100,000 Iterations* | *Area 2: 100,000 Iterations* | *Area 2: 200,000 Iterations* |
|---|---|---|---|
| 1st | 138.6 | 310.0 | 294.8 |
| 2nd | 139.5 | 308.6 | 300.3 |
| 3rd | 138.7 | 310.1 | 291.6 |



**7.15 Energy of the system for three different attempts for test area 1**



**7.16 Energy of the system for three different attempts for test area 2**

The algorithm was also applied three times for a maximum number of 200,000 iterations, only for test area 2. Figure 7.17 presents the energies of the 3 attempts with different settings of the cooling schedule. We can see from the graph that during those attempts it was the first time that second stopping criterion (rejection of 5,000 consecutive candidate solutions) was triggered and attempt 1 and 3 stopped after 153,000 and 178,000 iterations, accordingly. In all 3 attempts the final cost was lower than any of the runs with 100,000 iterations. The lowest cost was 291.6 (third attempt), which is substantially lower than the 310, reported in section 7.2.1.

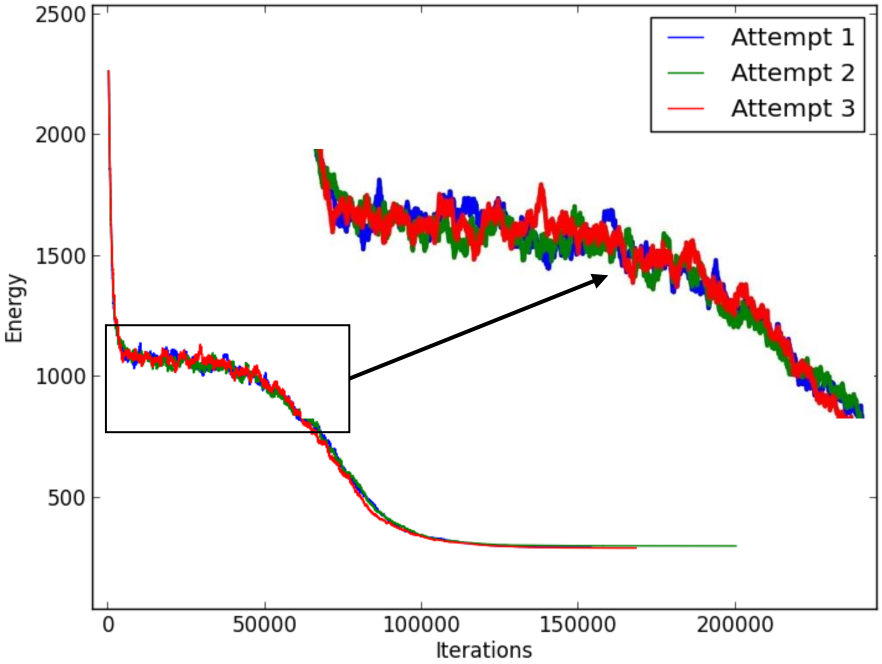
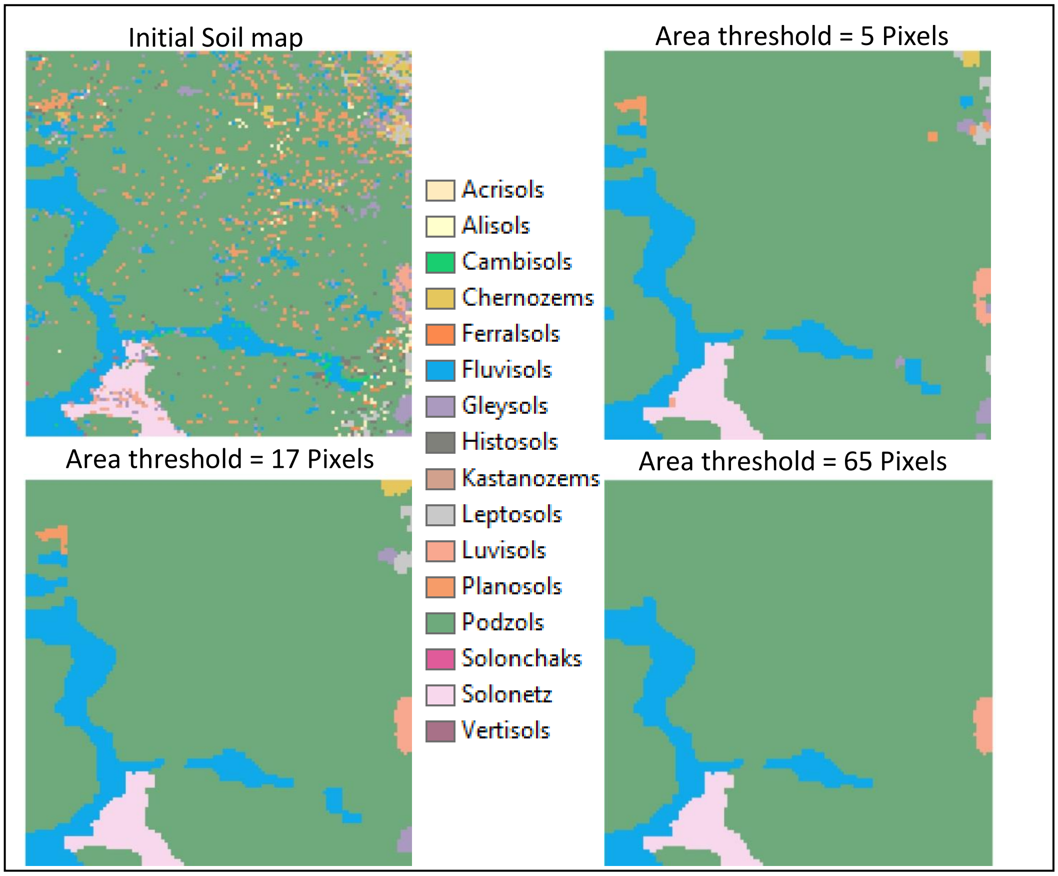


Figure 7.17 Energy of the system for three different attempts for high complexity test area and for 200,000 maximum iterations

## 7.5 Results of the Commercial Generalization Methodology

Figure 7.18 presents the results of the commercial generalized methodology, for all 3 levels of generalization, applied to the low complexity test area. Neither of the images contains patches of pixels less than the area threshold and thus, no area cost applies. The energies (cost) of the results are: 204.2 for the first level of generalization, 65.3 for the second one and 32.1 for the third level. Figure 7.19 presents the outputs of the commercial generalization methodology for the second, high complexity area. The energies of the results are: 450.1 for the first level of generalization, 253.3 for the second and 202.0 for the third one.

When the costs of the results are compared with the ones produced by the developed generalization methodology (Tables 7.2 and 7.3), one sees that the commercial methodology outperformed the criterion-based algorithm in half of the cases. That issue is discussed in Chapter 8.



**7.18: Results of the chosen generalization methodology, at low complexity test area, for all three generalizations levels.**

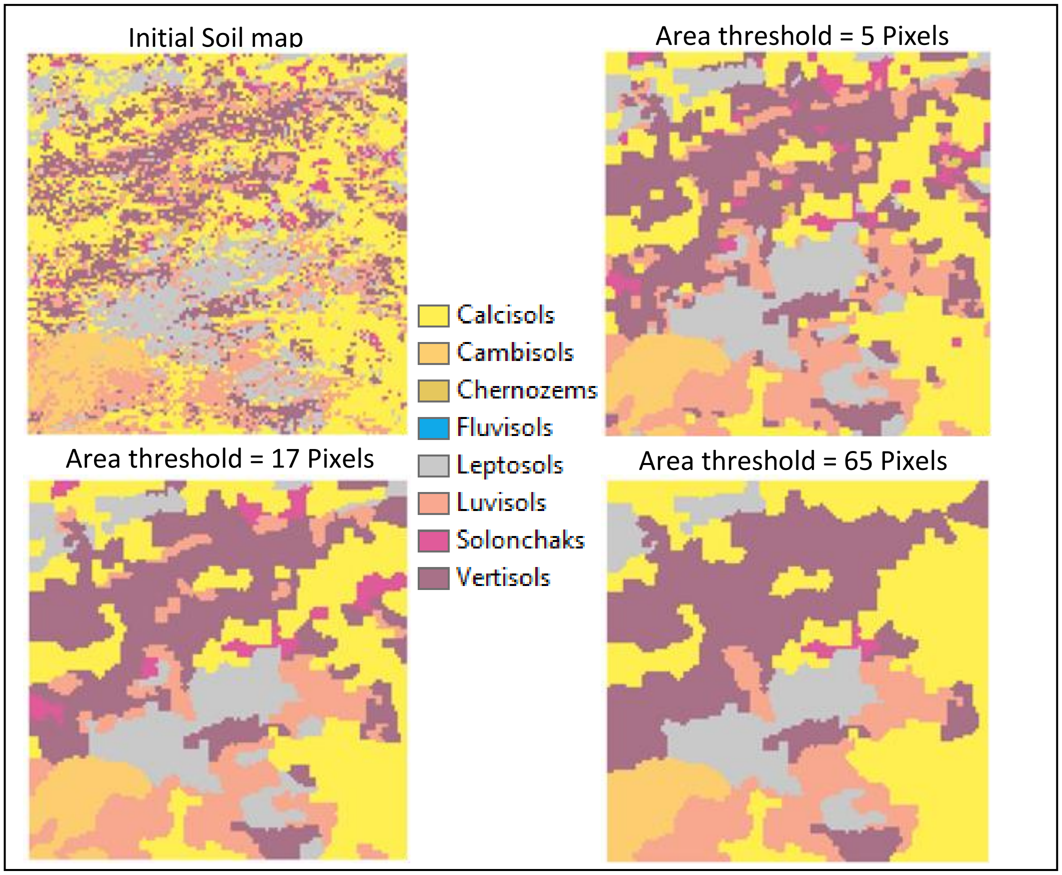


**Figure 7.19: Results of the chosen generalization methodology, at high complexity test area, for all three generalizations levels**

# 8. DISCUSSION

In this chapter a discussion over the results produced by the criterion-based generalization algorithm is conducted, specifically focusing on issues regarding local results, the pattern of the energy graph and the comparison with the commercial methodology.

## 8.1 Local results

There is interest in examining the results of the criterion-based generalization algorithm in specific sub-areas and comparing it visually with those of the commercial methodology. While in some areas the generalization achieved was visually satisfactory and had a close similarity to the input, in others the generalized result was a map that differed substantially from the input. Additionally, different levels of generalization of the same subsets may result in generalized maps that are completely different from each other. Only sub-areas from the initial results (presented in Chapters 7.1 and 7.2) are taken into account, and not from additional runs of the algorithm, that took place for investigation purposes. Figure 8.1 presents the sub-areas that the discussion focuses on; one sub-area for the first low complexity example and two sub-areas for the second high complexity one.

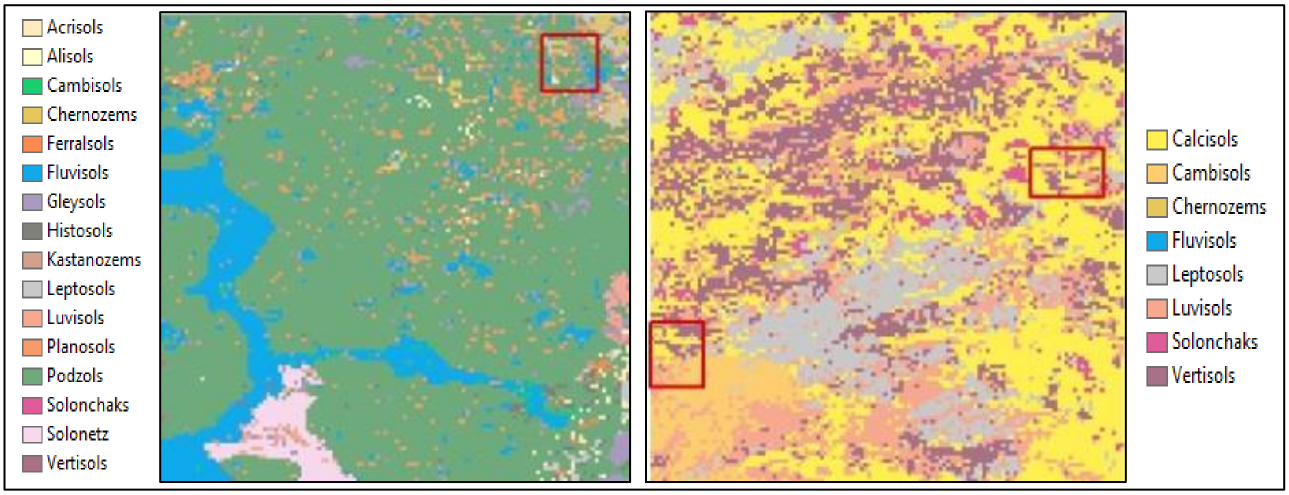


Figure 8.1: Sub-areas of the SoilGrids1km test areas

The first example in Figure 8.2 presents a case where the developed generalization algorithm produced a satisfactory result, at least in terms of similarity with the input. Especially, for the first generalization level (5 pixels) most of the small patches (mainly isolated pixels) have been reclassified in such a way that the semantics of the map have not been disturbed much, while patches smaller than 5 pixels no longer exist. On the contrary, the results produced by the commercial methodology tend to exaggerate big patches and completely eliminate the small ones, or in other words the commercial methodology tends to over-generalize. A characteristic of that over-generalization is the fact that the result of the 2nd generalization level (17 pixels) of the developed algorithm looks similar to the result of the commercial methodology, at the 1st level of generalization.

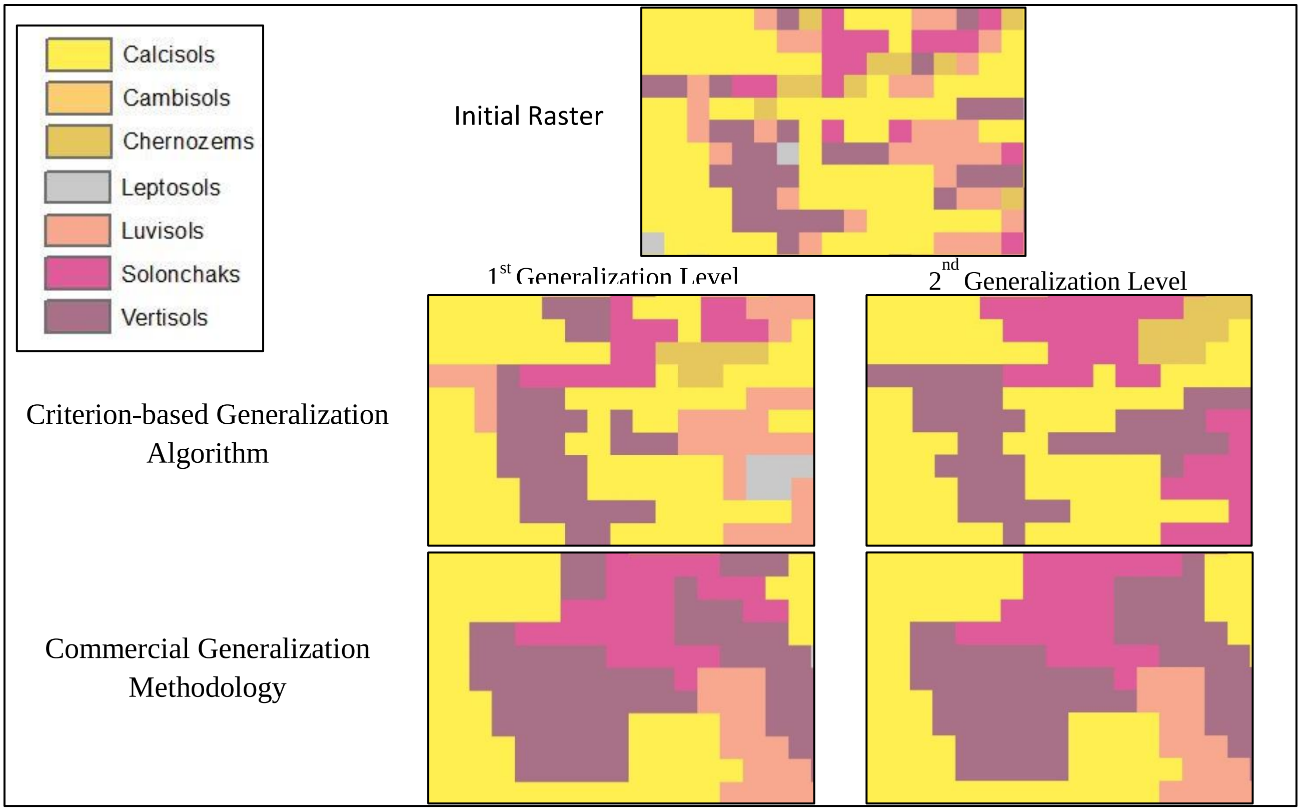


Figure 8.2: Local results of the developed algorithm and the commercial software, for 1st and 2nd generalization levels

Figure 8.3 presents a different case, where the output a generalization algorithm, are not that satisfactory. At the 1st level of generalization a light purple patch (solonchaks soil class) has been created at the upper left side of the image, which is not representative of the original situation. Furthermore, at the 2nd level of generalization, two patches of the solonchaks class appear in areas where one would not expect them. Especially for the patch at the lower part of the image, not only none of its pixels belonged initially in the solonchaks class, but also none of them had a neighbor of that class. This may seem odd, since the patches can only be reclassified to the value of one of its neighbors; however, as explained in Section 4.1, it is possible for a patch to eventually change into a class that was not in its initial neighborhood. It is worth noticing that for this specific sub-area the commercial methodology produced identical maps for both levels of generalization.

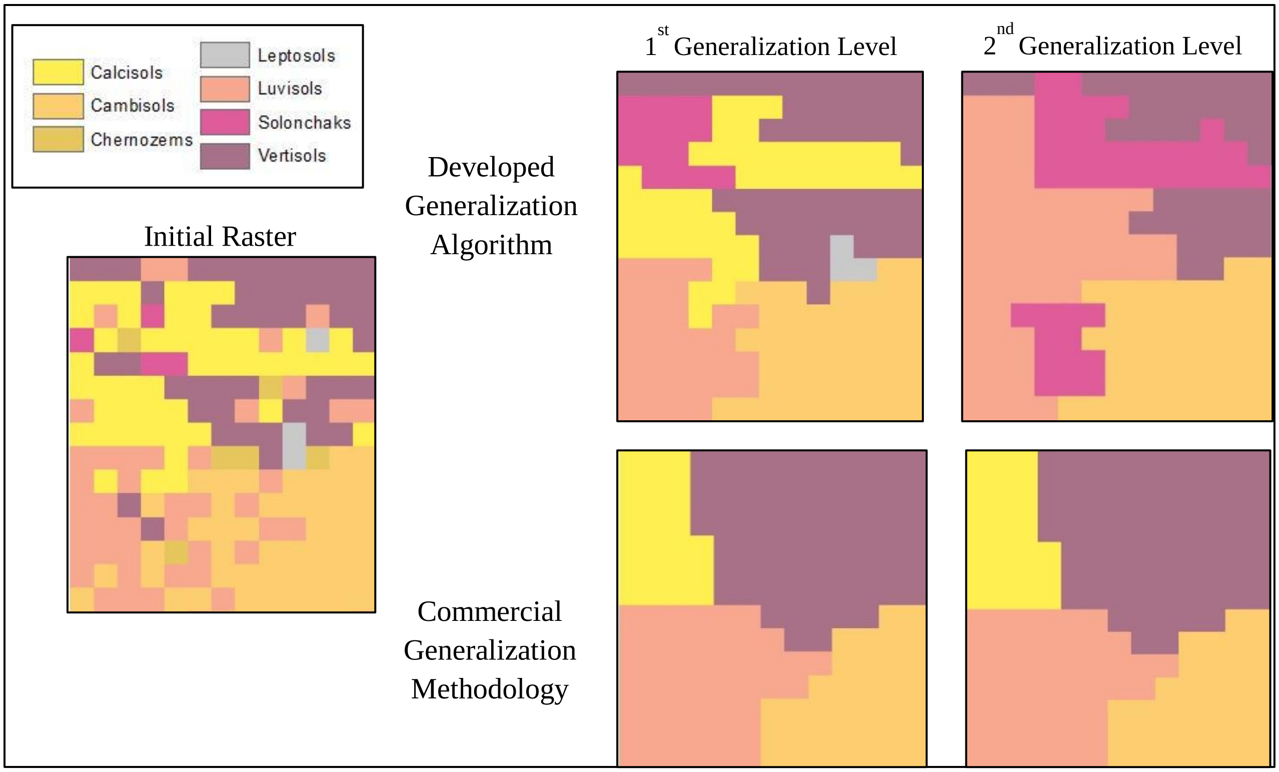


Figure 8.3: Local results of the developed algorithm and the commercial software, for 1st and 2nd generalization levels

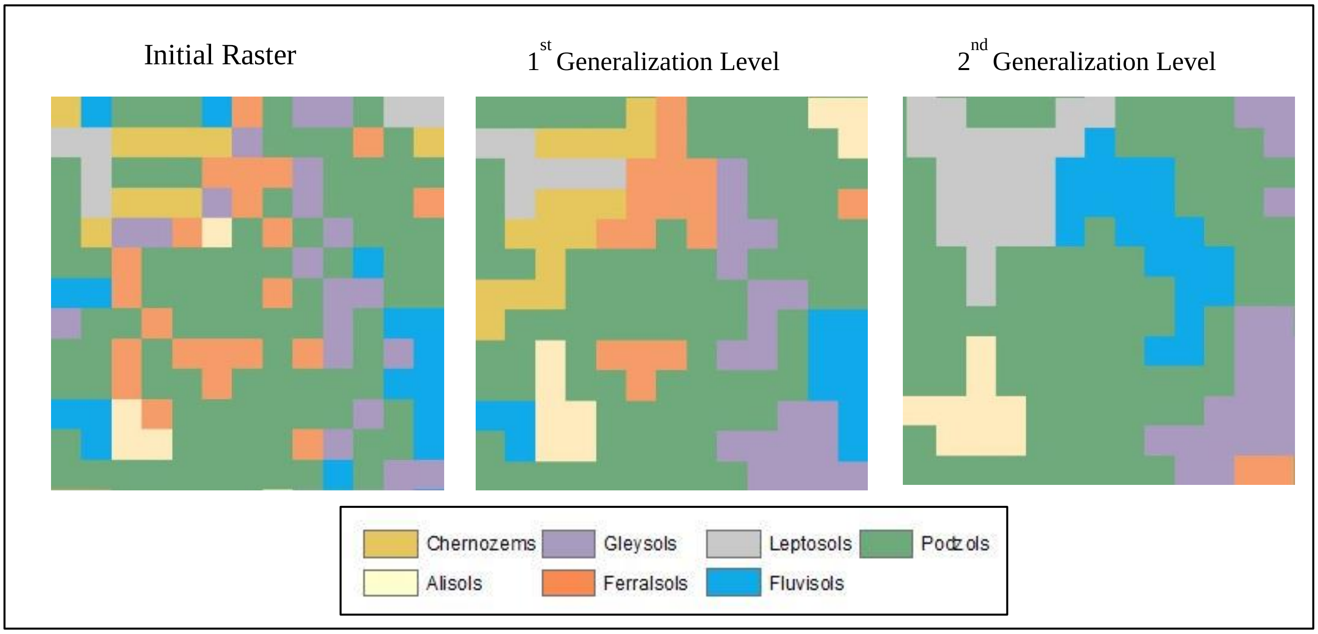


Figure 8.4: Local results of the developed algorithm for the first two levels of generalization

The last sub-area presented for discussion, is shown in Figure 8.4 and regards the first low complexity example. This example is similar with the one presented in Figure 8.3, as again results from different levels of generalization differ substantially. Specifically, in the 2$^{nd}$ generalization level a new patch appears (fluvisols class, blue), even though only one of its pixels belonged initially to that class.

The problem arises here is that when the potential user of the generalized maps wants to zoom in to a specific soil class; this soil class may not exist in the more detailed generalization level. In figure 8.4 for instance, if the user of the application zooms in to the fluvisols class at generalization level two, this class is not present in that specific sub-area in the first level of generalization. In principal, in generalizatied products new classes appear when the user zooms in, but classes from the previous zoom level do not disappear, since they express the dominant class of the area. In the developed algorithm however, such cases are possible.

Nevertheless, as regards local areas the criterion-based algorithm can produce generalized results that have a close similarity to the initial map. Especially for the runs at the first level of generalization, when compared with the ones of commercial methodology, the results maintain more details of the initial map.

## 8.2 Energy Graph

During the presentation of the results we came across several times with the graph of the energy of the system, plotted against the number of iterations (Figures 5.5, 7.4, 7.10, 7.15, 7.16 and 7.17). Regardless the input data and the amount of iterations the algorithm ran, the graph has the same basic shape. The graph in Figure 7.10 is used as the basis of this discussion and can be combined with Figure 7.8, which presents the intermediate solutions that corresponds to that energy values.

For the first thousand iterations the energy value drops significantly. After that and until approximately iteration 40,000 the energy slowly decreases. Between 40,000 and 60,000 the energy drops fast again where after that, it gets stable, with minor improvements until the end of the procedure.

The initial soil raster map of the high complexity test area contains 2,729 individual patches. The majority of them (1,675) contain only one pixel, while 700 additional patches exist with area between 2 and 4 pixels. Thus, at the beginning of the process when the first patches are randomly selected and reclassified, the chance that an isolated pixel is chosen is high. Additionally, the chance that this pixel is merged with another isolated pixel is high, as well. This case will lead to a significant decrease of the energy of the system. This is why during the first thousand iterations that significant drop in energy occurs. As it is shown in Figure 7.8, after 1,000 iterations, the amount of isolated pixels has decreased considerably.

After the first thousand iterations the shape of the energy graph can be explained taking into account the algorithm’s cooling schedule and the probability of accepting worse solutions. Until iteration 40,000, while the temperature is still high, the energy is not improved much, since the acceptance probability is high as well and the majority of worse solutions are accepted. Significant variations on the energy occur at this stage, as the algorithm explores the solution space. Between iteration 40,000 and 60,000, the decrease of the temperature and of the acceptance probability leads to rejection of most worse solutions and thus the energy value to drop. After iteration 60,000, the result does not seem to change considerably. This occurs due to the fact that at those iterations the temperature of the system is too low and thus only better solutions are accepted; however after 60,000 iterations the current solution is already good enough and better ones are hard to find, which leads to the similarity of the intermediate results at high iterations.

## 8.3 Comparison with Commercial Methodology

Surprisingly, in half out of the 6 examples (Section7.3), the commercial methodology outperformed the criterion-based algorithm. More precisely, the commercial generalization methodology produced a better result in both test areas in the third generalization level, while it was outperformed in both areas in the first one. That result was not expected, since one would expect an algorithm that optimizes specific criteria, to be able to find always a better solution than an algorithm that uses another objective function, when compared based on the criteria of the first one. Thus, investigation of the problem is required.

A potential answer could be that this is caused by choices made to increase the algorithm's efficiency. First, it was decided to reclassify entire patches instead of individual pixels, due to the fact that individual pixel reclassification would lead to an enormous solution space. Second, a patch was allowed only to be reclassified into the class of one its neighbor and not to any class of the map. That said, better solutions may exist out of the solution space of the developed algorithm and thus cannot be explored. Both choices were made in order to reduce the solution space, allowing the algorithm to explore it more efficiently. Such limitations do not apply for the commercial methodology.

Alternatively, the unexpected results may be caused by the failure of the criterion-based algorithm to reach a near-optimum solution, as a result of either the small number of iterations or of the selection of not appropriate parameters. Increasing the maximum number of iterations would probably lead to improved results (as shown in Section 7.4), with a cost on the computation time. It is not very likely, however, the improved results to be better than those of the commercial's methodology, since especially in the third generalization level the energy difference is significant.

As regards the parameters of SA, attention was paid to the cooling schedule (Section 4.3.4). Since the experimentation for the algorithms parameters configuration took place at the first level of generalization only, it was considered possible these parameter values not to apply for other levels. Specifically, the parameters of the cooling schedule were configured based on the average expected energy difference between the current and the candidate solution. That energy difference is taken into account in the acceptance probability function (Section 4.3.2). However, the average energy difference between two solutions varies considerably in different generalization levels, due to the fact that different weighted objective functions are used. The higher energy difference of the third generalization level results in a decreased value of the acceptance probability, making the algorithm to accept worse solutions with a lower rate.

A new experiment was performed for the high complexity area and the third level of generalization. In this new experiment the cooling schedule's parameters were re-configured, in order more iterations to be allowed in the phase were the algorithm still explores the solution space and accepts the majority of the solutions. A graph of the energy of the new attempt, compared with the initial run at this level, is presented in Figure 8.5. The parameters of the cooling schedule used are: maximum temperature = 100, minimum temperature = 0.05, constant factor = 0.999924, which results to 100,000 Iterations. The new experiment found a solution with a cost of 194.3, when the initial run had a cost of 247.5 and the commercial methodology scored 202. Thus, the choices of the SA parameters for the second and the third level of generalization seem to be a plausible explanation for the criterion-based algorithm to be outperformed by the commercial generalization.

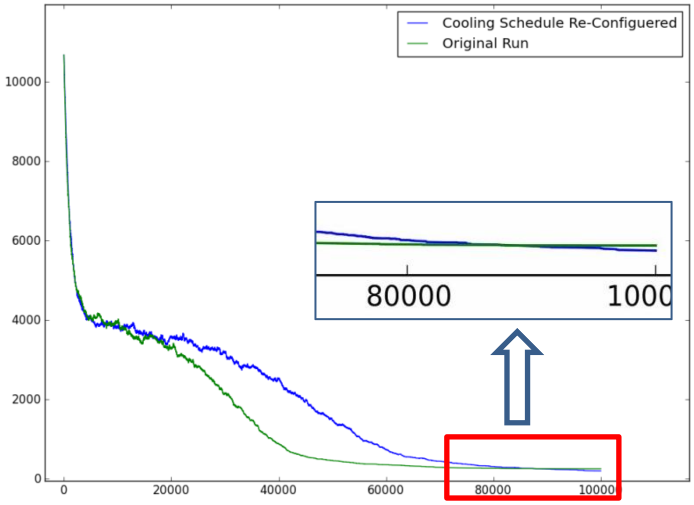


Figure 8.5: Energy of the system against the number of iterations. Attempt with the parameters of the cooling schedule re-configured and the initial run, for the third generalization level and the high complexity area.

# 9. CONCLUSIONS AND RECOMMENDATIONS

The general objective of this research was to develop, implement and evaluate an automated cartographic generalization procedure for categorical raster data. The application on the SoilGrids1km allow us to answer the research questions.

- Which criteria are sensible to use for cartographic generalization of categorical maps and which metrics can be used for evaluation of the result?

The selection of the appropriate criteria is the most important part in the generalization process, since they express the objective of the generalization. Criteria used in categorical maps can be divided into graphical (minimum object), topological (connectivity), structural (semantics) and gestalt (aesthetics). Four different criteria used in this research that regard the minimum area and the complexity of the shape of map entities, as well as the maintenance of the map semantics and the initial proportion of the classes.

Generalization criteria are also used to evaluate the result.  The weighted objective function penalizes properties of the generalized result that violate the selected criteria. The total cost of the result equals the weighted sum of all individual criteria. In that way different generalized results could be compared, based on the value of the objective function.

- How can Simulated Annealing be used for generating a near optimum result and which parameter settings are suitable?

The generalization problem was expressed as finding the map configuration that minimizes the total cost of the criteria. In order to retrieve a solution with a close to minimum cost the Simulated Annealing optimizer was used. The SA process was adapted to our specific generalization problem. In every step of the algorithm one patch was randomly selected and reclassified to the class of one of its neighbors, in order bigger map entities to be created.

As regards the SA parameters, the initial soil map was chosen as the initial solution (state) of the system. The initial temperature value was set to 10 and in every iteration was decreased based on an exponential cooling schedule. The procedure was scheduled to stop when the temperature value reached 0.001 or when 5,000 consecutive candidate solutions were rejected. Finally, the probability of accepting worse solutions was evaluated based on the Metropolis criterion, which takes into account the current temperature of the system, as well as the energy difference between the current and the candidate solutions.

- How does the procedure perform in practice when applied to the SoilGrids1km map in terms of value of the objective function (goodness of the generalization) and processing time?

The algorithm was applied on two square sub-sets of the SoilGrids1km product, for three different levels of generalization. The results were compared with the outputs of a conventional generalization methodology, which uses tools from the ArcGIS suite. The comparison was made based on the value of the objective function, as defined by the generalization criteria.

During the first generalization level (5 pixels) the developed algorithm produced better results than the comparison methodology. However, when additional runs were executed for twice the number iterations, lower-cost results were obtained; which mean that in the initial application, the global optimum had not been reached. For the third generalization level (65pixels), the commercial methodology outperformed the criterion-based algorithm.

As regards the algorithms processing time, the average time needed for 100,000 iterations for both the test areas was: 9 hours for the first generalization level, 11 hours for the second and 12 hours for the third one. This difference in processing time between different levels of generalization is caused by the different minimum mapable areas of each level. The degree of complexity of the input map plays a minor role on the required processing time.

*Recommendations*

This research deled with the cartographic generalization of raster categorical data and was applied to the SoilGrids1km product. After the selection and quantification of the generalization objectives, a generalization algorithm was developed, which makes use of the Simulated Annealing optimizer in order to find a solution that minimizes the sum of the criteria. It was shown that the SA optimizer can be successful on applying in problems of cartographic generalization, since it can produce solutions that satisfy the generalization objectives. Nevertheless, the algorithm requires a significant amount of computational time and extra attention must be paid on the configuration of the SA parameters and specifically of the cooling schedule.

As regards the usage of the developed algorithm in real world generalization problems, an obvious drawback is the required computation time. In addition, it was shown that as the input data increases in size, even more computation time is required. Thus, if an entire SoilGrids1km tile (1200x1200 pixels) is used as an input, the required processing time will increase dramatically.

One significant reason that causes the problem of the speed is the large number of isolated pixels existed in the application datasets. In both the test areas, the majority of the initial patches were isolated pixels. That said, if these isolated pixels were excluded from the procedure before SA applies (e.g. with a majority filter), less number of iterations would be required to reach a near-optimum result. Furthermore, since python is not considered as the fastest programming language, improved performance can be expected if the developed algorithm is implemented in another environment (c, java).

## 10. REFERENCES

A.Letourneau. 2007. “Calibration of Markovian Land Use Change Models”. M.S. thesis, Wageningen University and Research Centre.

Aerts, Jeroen C. J. H., and Gerard B. M. Heuvelink. 2002. “Using Simulated Annealing for Resource Allocation.” *International Journal of Geographical Information Science* 16 (6).

Azizi, Nader, and Saeed Zolfaghari. 2004. “Adaptive Temperature Control for Simulated Annealing: A Comparative Study.” *Computers & Operations Research* 31 (14): 2439–51.

Bader, Matthias, and Robert Weibel. 1997. “Detecting and Resolving Size and Proximity Conflicts in the Generalization of Polygonal Maps.” In Proceedings of the 18th International Cartographic Association Conference , 23–27. Stockholm, Sweden.

Carrao, H, and Roberto Henriques. 2001. “MapGen–Automated Generalisation for Thematic Cartography.” In *Proceedings of 16th ESRI EMEA User Conference,* 1–18.

Černý, V. 1985. “Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm.” *Journal of Optimization Theory and Applications* 45 (l): 41–51.

Cheng, Tao, and Zhilin Li. 2006. “Effect of Generalization on Area Features: A Comparative Study of Two Strategies.” *The Cartographic Journal* 43 (2): 157–70.

Edwardes, Alistair, and William Mackaness. 2000. “Modelling Knowledge for Automated Generalization of Categorical Maps-a Constraint Based Approach.” *GIS and GeoComputation (Innovations in GIS),* no. August: 161–73.

ESRI. 2014. “Generalization of Classified Raster Imagery.” http://resources.arcgis.com/en/help/main/10.2/index.html#//009z0000003p000000.

Föerster, Theodor, Jantien Stoter, and Barend Köbben. 2007. *Towards a Formal Classification of Generalization Operators*. *Proceedings of the 23th International Cartographic Conference*. Enschede, The Netherlands.

Gao, W, A Song, and J Gong. 2004. “Constraint-Based Generalization of Soil Map.” In *Proceedings of the XXth ISPRS Congress*, 2–6. Instanbul: ISPRS.

Kazemi, S, S Lim, and C Rizos. 2004. “A Review of Map and Spatial Database Generalization for Developing a Generalization Framework.” In *XXth Congress of the International Society for Photogrammetry and Remote Sensing*. Istanbul, Turkey.

Kirkpatrick, S., C. D. Gelatt, and M. P. Vecchi. 1983. “Optimization by Simulated Annealing” Science, 220 (4598): 671–80.

Kokash, Natallia. 2005. An Introduction to Heuristic Algorithms. Belarusian State University, Department of Informatics and Telecommunications

Li, Bo, G Wilkinson, and Souheil Khaddaj. 2001. “Cell-Based Model for GIS Generalization.” *Proc. of the 6th International Conference on GeoComputation, University of Queensland, Brisbane, Australia, 24-26 September.*

Li, Zhilin. 2007. “Digital Map Generalization at the Age of Enlightenment: A Review of the First Forty Years.” *The Cartographic Journal* 44 (1): 80–93.

Mackaness, William A., Anne Ruas, and L. Tiina Sarjakoski. 2007. *Generalisation of Geographic Information: Cartographic Modelling and Applications*. *Generalisation of Geographic Information*. Elsevier Ltd: 11-36.

Maringer, D. 2005. *Portfolio Management with Heuristic Optimization*. Springer: 38-72.

Michiels, W, E Aarts, and J Korst. 2007. *Theoretical Aspects of Local Search*. New York: Springer.

Minasny, Budiman, Alex. B. McBratney, and Alfred E. Hartemink. 2010. “Global Pedodiversity, Taxonomic Distance, and the World Reference Base.” *Geoderma* 155 (3-4). Elsevier

Molenaar, Martien. 1996. “Methods for the Generalization of Geo-Databases.” *Netherlands Geodetic Commission, Publications on Geodesy New Series 43*, no. 3 1.

Nakos, Byron. 2001. “On the Assessment of Manual Line Simplification Based on Sliver Polygon Shape Analysis.” In *4th ICA Workshop on Progress in Automated Map Generalisation*. Beijing.

Neun, Moritz, Dirk Burghardt, and Robert Weibel. 2008. “Automated Processing for Map Generalization Using Web Services.” *GeoInformatica* 13 (4): 425–52.

Nourani, Yaghout, and Bjarne Andresen. 1998. “A Comparison of Simulated Annealing Cooling Strategies.” *Journal of Physics A: Mathematical and Theoretical* 31: 8373–85.

Peter, Beat, and Robert Weibel. 1999. “Using Vector and Raster-Based Techniques in Categorical Map Generalization.” In *Third ICA Workshop on Progress in Automated Map Generalization*, 12–14. Ottawa.

Phillips, J. D. 2013. “Evaluating Taxonomic Adjacency as a Source of Soil Map Uncertainty.” *European Journal of Soil Science* 64 (4): 391–400..

Poole, DL, and AK Mackworth. 2010. *Artificial Intelligence: Foundations of Computational Agents*. Cambride: Cambridge University Press.

Pukkala, Timo, and Mikko Kurttila. 2005. “Examining the Performance of Six Heuristic Optimisation Techniques in Different Forest Planning Problems.” *Silva Fennica* 39 (August 2004).

Smirnoff, Alex, Gabriel Huot-Vézina, Serge J. Paradis, and Ruth Boivin. 2012. “Generalizing Geological Maps with the GeoScaler Software: The Case Study Approach.” *Computers & Geosciences* 40 (March): 66–86.

Smirnoff, Alex, Serge J. Paradis, and Ruth Boivin. 2008. “Generalizing Surficial Geological Maps for Scale Change: ArcGIS Tools vs. Cellular Automata Model.” *Computers & Geosciences* 34 (11): 1550–68.

Steiniger, Stefan, and Robert Weibel. 2005. “A Conceptual Framework for Automated Generalization and Its Application to Geologic and Soil Maps.” In *Proceedings of XXII Int. Cartographic Conference*, 11–16.

Stoter, J. 2005. “Generalisation: The Gap between Research and Practice.” In *In Proceedings of the 8th ICA Workshop on Generalisation and Multiple Representation,* 1–10. A Coruña, Spain.

Stoter, Jantien, Dirk Burghardt, Cécile Duchêne, Blanca Baella, Nico Bakker, Connie Blok, Maria Pla, Nicolas Regnauld, Guillaume Touya, and Stefan Schmid. 2009. “Methodology for Evaluating Automated Map Generalization in Commercial Software.” *Computers, Environment and Urban Systems* 33 (5). Elsevier Ltd: 311–24.

Wambeke, A van, and TR Forbes. 1986. *Guidelines for Using Soil Taxonomy in the Names of Soil Map Units*. New York, USA.

Ware, J. Mark, Christopher B. Jones, and Nathan Thomas. 2003. “Automated Map Generalization with Multiple Operators: A Simulated Annealing Approach.” *International Journal of Geographical Information Science* 17 (8): 743–69.

Yang, JianGang, and JinQiu Yang. 2011. “Intelligence Optimization Algorithms: A Survey.” *International Journal of Advancements in Computing Technology* 3 (4): 144–52.

Yaolin, L, Martin Molenaar, and MJ Kraak. 2002. “Semantic Similarity Evaluation Model in Categorical Database Generalization.” In *Symposium on Geospatial Theory, Processing and Applications*. Ottawa, Canada.

# APPENDIX

### Main Python Script

```
# import required libraries
import os
os.chdir(working directory)
import functions
import time

# start time
start = time.time()

####### Pre-Processing############

functions.CreatePatches("data\\initialRaster.tif")
functions.CreatePatchInfoTable("data\\patches.tif")
functions.CreateNeighborList()
functions.CreateSmallPatchesList(functions.areaThreshold * 1.5)
functions.CreateSuperPatchesList("data\\PatchInfo.dbf")

####### Simulated Annealing ############

functions.iteration = 1

# stopping criteria
while functions.temperature > 0.001\
      and functions.concecutiveWorseSolutions <= 5000:

    # produce and evaluate candidate solution

    functions.GenerateCandidateSolution()
    functions.EvaluateCandidateEnergy()
    functions.AcceptancePropability()
    functions.temperature = functions.t0 * (0.9998849 ** functions.iteration)

    # write intermidiate solutions to file

    if functions.iteration % 1000 == 0:
        functions.WriteSolutionToFile(functions.iteration)

    functions.iteration= functions.iteration + 1

# write final solution to file
functions.WriteSolutionToFile("LastSolution")
end = time.time()
durationSec = end - start

print "best solution = " + " " + str(functions.bestSolution)
print "duration = ", durationSec/3600,"hours "

# plot energy graph
functions.plotEnergies()
```

### Functions Script (functions.py)

```
from __future__ import division
import arcpy
from arcpy import env
from arcpy.sa import *
import os
import csv
import random
import math
import copy
import pickle
import matplotlib.pyplot as plt

areaThreshold = 5

# weights
semanticsWeight = 0.125
shapeWeight = 5
proportionWeight = 1000


# global variables
neighborList=[[]]
commonBoundaryList=[[]]
currentEnergy = None
candidateEnergy = 0
selectedPatch=None
patchPreviousClass = None
patchCurrentClass = None
smallPatches=[]
t0 = 100
temperature = t0
bestSolution = 10000000000
bestIteration = 0
iteration = 0
sprPatches=[[]]
sprPatchesCost =[]
candidateSprPatches = [[]]
candidateSprPatchesCost = []
initialClassesArea = {}
currentClassesArea = {}
totalPixels = 0
repeatMode = False
energiesList = []
concecutiveWorseSolutions = 0




# set the enviroment settings
env.workspace = os.getcwd()
env.overwriteOutput = True

# if we are in repeat mode set the initial state of the random function
if repeatMode:
    seed = pickle.load(open("randomState.p","rb"))
    random.setstate(seed)

# get the random's function state (seed) and save it to file
seed = random.getstate()
pickle.dump(seed,open("randomState.p","wb"))
seed = None


# creates the raster (geoTiff) with the individual patches
def CreatePatches(raster):
    global totalPixels

    #create rasters attribute table
    arcpy.BuildRasterAttributeTable_management(raster, "Overwrite")

    # check spatial analyst extension
    arcpy.CheckOutExtension("Spatial")

    # Distinguish patches (create patches)
    patches = RegionGroup(raster,"FOUR","WITHIN","ADD_LINK",0)

    # make id field
    arcpy.AddField_management(patches, "ID", "SHORT", 5,
                              "", "", "", "NON_NULLABLE", "REQUIRED")
    arcpy.CalculateField_management(patches, "ID","!VALUE! - 1","PYTHON_9.3")

    # find the total pixels of the image
    desc = arcpy.Describe(patches)
    totalPixels = desc.height * desc.width



    # save and release the variable
    patches.save("data\\patches.tif")
    patches = None



def CreatePatchInfoTable(patches):
    # create patchInfo table
    dataFolder = str(env.workspace + "\\data")
    patchTable =
arcpy.TableToTable_conversion(patches,dataFolder,"PatchInfo.dbf")

    # Add area, initial and current class fields
    arcpy.AddField_management(patchTable, "AREA_PIX", "SHORT", 10,
                              "", "", "", "NON_NULLABLE", "REQUIRED")
    arcpy.AddField_management(patchTable, "INIT_CLASS", "SHORT", 5,
                              "", "", "", "NON_NULLABLE", "REQUIRED")
    arcpy.AddField_management(patchTable, "CURR_CLASS", "SHORT", 5,
                              "", "", "", "NON_NULLABLE", "REQUIRED")
    # calculate fields value
    arcpy.CalculateField_management(patchTable,
"AREA_PIX","!COUNT!","PYTHON_9.3")
    arcpy.CalculateField_management(patchTable,
"INIT_CLASS","!LINK!","PYTHON_9.3")
```

```
    arcpy.CalculateField_management(patchTable,
"CURR_CLASS","!INIT_CLASS!","PYTHON_9.3")

    # Drop old fields
    arcpy.DeleteField_management(patchTable,"COUNT")
    arcpy.DeleteField_management(patchTable,"LINK")
    arcpy.DeleteField_management(patchTable,"VALUE")

    ############ CREATE NEIGHBORHOUD AND PERIMETER INFO ###########

    # first baptize the cs as a metric one, so we can calculate the
(unitless) shape index
    # cerate spatial reference  object
    projection = arcpy.SpatialReference("WGS 1984 World Mercator")

    # copy the raster
    arcpy.CopyRaster_management(patches, "data\\projectedRaster.tif")

    # define the projection  (any metric system will work)
    arcpy.DefineProjection_management("data\\projectedRaster.tif",
projection)



    # vectorize the projected raster

arcpy.RasterToPolygon_conversion("data\\projectedRaster.tif","data\\patches.s
hp",\
                                     "NO_SIMPLIFY", "ID")

    # add area and perimeter fields in shapefile's table
    arcpy.AddField_management("data\\patches.shp","PERIMETER","Double")
    arcpy.AddField_management("data\\patches.shp","AREA","Double")

    expression1 = "{0}".format("!SHAPE.length@METERS!")
    expression2 = "{0}".format("!SHAPE.area@SQUAREMETERS!")

    arcpy.CalculateField_management("data\\patches.shp", "PERIMETER",
expression1, "PYTHON",)
    arcpy.CalculateField_management("data\\patches.shp", "AREA", expression2,
"PYTHON",)

    # join perimeter and area fields to PatchInfo.dbf
    arcpy.JoinField_management("data\\PatchInfo.dbf", "ID",
"data\\patches.shp",\
                              "GRIDCODE", "PERIMETER")
    arcpy.JoinField_management("data\\PatchInfo.dbf", "ID",
"data\\patches.shp",\
                              "GRIDCODE", "AREA")

    # create table (dbf) of polygon neighbors (required for the
CreateSuperPatchesList function)
    arcpy.PolygonNeighbors_analysis ("data\\patches.shp",
"data\\neighbors.dbf",\
                                     "GRIDCODE",
"NO_AREA_OVERLAP","BOTH_SIDES",\
                                     "","METERS")
```

```
def CreateSuperPatchesList(patchTable):
    # create super object list of list
    global sprPatches
    global sprPatchesCost
    global currentEnergy
    global initialClassesArea
    global currentClassesArea

    # Create cursor object to read dbf table
    fields = ["ID","AREA_PIX","CURR_CLASS"]

    with arcpy.da.SearchCursor( patchTable,fields) as cursor:
        # iterate through the PatchInfo Table to fill sprPatches list
        # and the initialClassesArea dictionary
        sprPatches = []
        for row in cursor:
            # fill sprPatches list
            patchID=row[0]
            patchID=[patchID]
            sprPatches.append(patchID)
            # fill initialClassesArea dictionary
            area = row[1]
            className = row[2]
            if className in initialClassesArea:
                initialClassesArea[className] = initialClassesArea[className]
+ area
            else:
                initialClassesArea[className] = area


    # create currentClassesArea dictionary
    currentClassesArea = initialClassesArea.copy()

    # fill super patches cost list with values
    # and evaluate the energy of the initial raster
    totalCost = 0
    for i in range(len(sprPatches)):
        cost = UpdateSprPatchEnergy(sprPatches,i)
        sprPatchesCost.append(cost)
        # Total cost of initial solution
        totalCost= totalCost + cost

    currentEnergy = totalCost




def CreateNeighborList():
    global neighborList
    global commonBoundaryList

    # get the total nr of the patches
    nrOfPatches =
int(arcpy.GetCount_management("data\\PatchInfo.dbf").getOutput(0))
```

```
    # create neighbor and common boundaries lists from neighbors.dbf
    listOfNeighbors=[]
    commonnBoundary = []

    for i in range(nrOfPatches):
        listOfNeighbors.append([])
        commonnBoundary.append([])
        # prepare sql where clause and query the dbf
        whereClause = "src_GRIDCO = %i" %i + " " + 'AND LENGTH <> 0'
        fields = ["nbr_GRIDCO","LENGTH"]

        with arcpy.da.SearchCursor( "data\\neighbors.dbf",fields,whereClause)
as cursor:
            # get the neighborID and its perimeter length and add them to the
lists
            for row in cursor:
                 neighbor = int(row[0])
                 length = row[1]
                 listOfNeighbors[i].append(neighbor)
                 commonnBoundary[i].append(length)

    # convert to tuple (immutable)
    neighborList = tuple(listOfNeighbors)
    commonBoundaryList = tuple(commonnBoundary)




# creates a list of patches with area < than the threshold.
# From this list the patches will be selected randomly
# The threshold is not nesecerilly equal to the areaThreshold


def CreateSmallPatchesList(threshold = areaThreshold):
    global smallPatches

    # prepare sql where clause and query the dbf
    whereClause ="AREA_PIX <= %i" %threshold
    fields = "ID"
    with arcpy.da.SearchCursor("data\\PatchInfo.dbf",fields,whereClause) as
cursor:
        # insert patchId to smallpatches list
        for row in cursor:
            patchID = row[0]
            smallPatches.append(patchID)



# returns the cost of the input super-patch
def UpdateSprPatchEnergy(sprPatchesList,index):
    global areaThreshold
    global proportionWeight
    global shapeWeight


    # sum the partial costs up
    cost=AreaCost(sprPatchesList,index)
    cost=cost + SemanticsCost(sprPatchesList,index)/semanticsWeight
    cost= cost + (ShapeCost(sprPatchesList,index) * shapeWeight)

    return cost



# randomly seleect a patch from the small patches list
# and also randomly change its class
def GenerateCandidateSolution():
    global smallPatches
    global selectedPatch
    global patchPreviousClass
    global patchCurrentClass
    global neighborList
    global currentEnergy
    global bestIteration
    global iteration

    # add the current energy for plot puproses
    energiesList.append((iteration,currentEnergy))


    # the first while loop is for avoiding cases where the patch cannot be
    # reclassified  to any other class and thus the neighbors list becomes
empty
    reSelect = True
    while reSelect == True:
        reSelect = False
        # randomly select patch from small patces list
        selectedPatch = random.choice(smallPatches)
        # get the class of the selected class
        whereClause ="ID = %i" %selectedPatch

        with
arcpy.da.SearchCursor("data\\PatchInfo.dbf","CURR_CLASS",whereClause)as
cursor:
            for row in cursor:
                currentClass = row[0]

        # select a neighbor class (or the initial)
        neighbors = neighborList[selectedPatch][:]
        neighbors.append(selectedPatch)#(include also initial class)
        # get neighbor's class
        classFound = False
        while classFound == False:
            if len(neighbors) > 0:
                randomNeighbor= random.choice(neighbors)
                whereClause ="ID = %i" %randomNeighbor
                fields=["INIT_CLASS","CURR_CLASS"]
                with
arcpy.da.SearchCursor("data\\PatchInfo.dbf",fields,whereClause)as cursor:
                    # if select itself give the initial value
                    if randomNeighbor == selectedPatch:
                        for row in cursor:
                            selectedClass = row[0]
                    # get the class of the selected neighbor
                    else:
                        for row in cursor:
                            selectedClass = row[1]

                # check if current and candidate classe differ. If yes remove
patch from list
                if selectedClass == currentClass:
                    neighbors.remove(randomNeighbor)
                else:
                    classFound = True
            # if list has become empty, select another patch (outer loop)
            elif len(neighbors) == 0:
                reSelect = True
                classFound = True


    # change the class of the selected patch
    whereClause ="ID = %i" %selectedPatch
    with
arcpy.da.UpdateCursor("data\\PatchInfo.dbf","CURR_CLASS",whereClause)as
cursor:
        for row in cursor:
            # give value to the global var patchPreviousClass
            # and change class value to PatchInfo table
            patchPreviousClass = row[0]
            row[0] = selectedClass
            cursor.updateRow(row)


    # give value to global var patchCurrentClass
    patchCurrentClass = selectedClass

    # print results
    print "to class" + " " + str(randomNeighbor)


# evaluates the energy of the candidate solution and accepts or rejects the
solution
def EvaluateCandidateEnergy():
    # copy super Object and super obbjec's cost lists
    global neighborList
    global selectedPatch
    global patchPreviousClass
    global patchCurrentClass
    global currentEnergy
    global bestSolution
    global bestIteration
    global iteration
    global sprPatches
    global sprPatchesCost
    global candidateSprPatches
    global candidateSprPatchesCost
    global candidateEnergy
    global currentClassesArea
    global proportionWeight
    ############# current solution evaluation################

    # find the super-patch that the patch used to belong to.
    dissolvedSprPatch = [findSprPatch(selectedPatch,sprPatches)] # as a list


    # find the super-patch(es) that the patch  has been aggregated to
    # the plural is for the case the patch has unite 2 or more super-patches
    # there is also the case the patch has been reclassified to its initial
class
    # and thus the aggregatedSprPatch will be an empty list
    aggregatedSprPatch = CreateAggegatedSprPatchList()

    # calculate current dissolved and aggregated super-patches costs
    dissPatchCost = sprPatchesCost[dissolvedSprPatch[0]]
    aggPatchCost=0
    for item in aggregatedSprPatch:
        aggPatchCost = aggPatchCost + sprPatchesCost[item]

    # current proportion cost of the two affected classes(with weights)
    currPrevClassCost = ProportionCost(patchPreviousClass)
    currCurrClassCost = ProportionCost(patchCurrentClass)
    currentProportionCost = (currPrevClassCost + currCurrClassCost) *
proportionWeight #bweight

    # current cost of the selected super-objects and the affected classes
    currentPartialCost=dissPatchCost + aggPatchCost + currentProportionCost


    ############# candidate solution evaluation################

    # copy super patches and super patches's cost lists without aliasing
    candidateSprPatches = []
    for patches in sprPatches:
        candidateSprPatches.append(patches[:])

    candidateSprPatchesCost = sprPatchesCost[:]

    # update currentClassesArea dictionary
    whereClause ="ID = %i" %selectedPatch
    with
arcpy.da.SearchCursor("data\\PatchInfo.dbf","AREA_PIX",whereClause)as cursor:
        # first get the patches area (in pixels)
        for row in cursor:
            patchArea = row[0]

    # then, update currentClassesArea
    currentClassesArea[patchPreviousClass] =
currentClassesArea[patchPreviousClass]\
                                             - patchArea
    currentClassesArea[patchCurrentClass] =
currentClassesArea[patchCurrentClass]\
                                             + patchArea


    # if aggregated super patch is empty (patch has been raclassified
    # to its initial class) create it
    if len(aggregatedSprPatch) == 0 :
        candidateSprPatches.append([])
        candidateSprPatchesCost.append(0.0)
        # the last element of the candidateSprPatches is the new
aggregatedSprPatch
        aggregatedSprPatch.append(len(candidateSprPatches) - 1)

    # move selected patch from the dissolved to the aggregated super-patch
    candidateSprPatches[dissolvedSprPatch[0]].remove(selectedPatch)
    candidateSprPatches[aggregatedSprPatch[0]].append(selectedPatch)

    # join aggregated super object elements (if > 1).If len = 1 no need to do
anything
    if len(aggregatedSprPatch)>1:
        # make a copy so can remove items while looping
        CopyCandidateSprPatches = []
        for patches in candidateSprPatches:
            CopyCandidateSprPatches.append(patches[:])

        for i in range(1,len(aggregatedSprPatch)):
            sprPatchIndex = aggregatedSprPatch[i]
            # move all patches from this super-patch
            for patch in (CopyCandidateSprPatches[sprPatchIndex]):
                candidateSprPatches[aggregatedSprPatch[0]].append(patch)
                candidateSprPatches[sprPatchIndex].remove(patch)
        CopyCandidateSprPatches = None
        # remove additional super patches  from aggregatedSprPatches
        del aggregatedSprPatch[1:]

    # split (if necesery) the dissolvedSprPatch into individual super-patches
    # returns dissolvedSprPatch as a kist of uper-patches
    SplitDissolvedSprPatch(dissolvedSprPatch)

    # update candidate super-patches costs
    for item in dissolvedSprPatch:
        candidateSprPatchesCost[item] =
UpdateSprPatchEnergy(candidateSprPatches,item)
    candidateSprPatchesCost[aggregatedSprPatch[0]] =
UpdateSprPatchEnergy(candidateSprPatches,aggregatedSprPatch[0])

    # evaluate costs of new super-patches
    candAggSprPatchCost = candidateSprPatchesCost[aggregatedSprPatch[0]]
    candDisSprPatchCost = 0
    for sprPatchIndex in dissolvedSprPatch:
        candDisSprPatchCost = candDisSprPatchCost +
candidateSprPatchesCost[sprPatchIndex]

    # current proportion cost of the two affected classes
    candPrevClassCost = ProportionCost(patchPreviousClass)
    candCurrClassCost = ProportionCost(patchCurrentClass)
    candidateProportionCost = (candPrevClassCost + candCurrClassCost) *
proportionWeight # weight

    # candidate cost of the affected super objects + and classes
    candidatePartialCost = candAggSprPatchCost + candDisSprPatchCost +
candidateProportionCost
```

```
    ##### total  cost for the candidate solution #####
    candidateEnergy =  (candidatePartialCost - currentPartialCost) +
currentEnergy




# compares the candidate with the current energy and assigns an acceptance
propability
def AcceptancePropability():
    global currentEnergy
    global candidateEnergy
    global temperature
    global sprPatches
    global sprPatchesCost
    global candidateSprPatches
    global candidateSprPatches
    global energiesList
    global iteration
    global bestSolution
    global selectedPatch
    global patchPreviousClass
    global patchCurrentClass
    global currentClassesArea
    global concecutiveWorseSolutions


    # acceptance probability function
    if candidateEnergy < currentEnergy:
        probability = 1.0
    else:
        probability = math.exp( (currentEnergy - candidateEnergy) /
temperature)

    # accept or not the candidate solution

    if random.random() < probability:

        concecutiveWorseSolutions = 0
        solution = "solution accepted"
        #accept the solution.  update and save lists
        CopyCandidateSprPatches = []
        for patches in candidateSprPatches:
            CopyCandidateSprPatches.append(patches[:])

        # remove empty elements from the lists
        for lst in CopyCandidateSprPatches:
            if len(lst) == 0:
                index= candidateSprPatches.index(lst)
                candidateSprPatches.remove(lst)
                del candidateSprPatchesCost[index]

        # replace current energy
        currentEnergy = candidateEnergy
        energiesList.append((iteration,currentEnergy))
        if currentEnergy < bestSolution:
            bestSolution = currentEnergy

        # replace global variables with the new super-patches lists
        sprPatches = candidateSprPatches
        sprPatchesCost = candidateSprPatchesCost

    # if solution is not accepted change the class value and the
    # currentClassesArea back
    else:
        concecutiveWorseSolutions = concecutiveWorseSolutions + 1
        solution = "solution not accepted"
        whereClause ="ID = %i" %selectedPatch
        fields = ["AREA_PIX", "CURR_CLASS"]
        with
arcpy.da.UpdateCursor("data\\PatchInfo.dbf",fields,whereClause)as cursor:
            for row in cursor:
                patchArea = row[0]
                row[1] = patchPreviousClass
                cursor.updateRow(row)

        # give previous values to currentClassesArea
        currentClassesArea[patchPreviousClass] =
currentClassesArea[patchPreviousClass]\
                                             + patchArea
        currentClassesArea[patchCurrentClass] =
currentClassesArea[patchCurrentClass]\
                                             - patchArea



    # print feedback to the user
    print "Solution at temperature" + " " + str(temperature)+ ".  " + \
          "Object with id" + " " + str(selectedPatch) + " " + "has been
reclassified." \
          + "\n" + " Solution's Energy is" + " " +str(candidateEnergy) + ". "
\
          + solution + " " + "with probability=" + str(probability)





# create a list of the  paches that have been aggregated
def CreateAggegatedSprPatchList():
    global selectedPatch
    global sprPatches

    # get selected's patch new class
    whereClause ="ID = %i" %selectedPatch
    with
arcpy.da.SearchCursor("data\\PatchInfo.dbf","CURR_CLASS",whereClause) as
cursor:
        for row in cursor:
            selectedPatchClass = row[0]
    # finf the neighbors of the selected patch
    neighbors = neighborList[selectedPatch][:]
    aggregatedSprPatch =[]

    # find the super object(s) that the selected patch has been aggregated to
    for patch in neighbors:
        whereClause ="ID = %i" %patch
        with
arcpy.da.SearchCursor("data\\PatchInfo.dbf","CURR_CLASS",whereClause) as
cursor:
            for row in cursor:
                neighborClass = row[0]

        # check classes
        if neighborClass == selectedPatchClass:
            # find the super patch that the neighbor belongs to
            sprPatchID = findSprPatch(patch,sprPatches)
            # check if super-patch is already in list
            if sprPatchID not in aggregatedSprPatch:
                 aggregatedSprPatch.append(sprPatchID)

    return aggregatedSprPatch


# split the dissolvedSprPatch into more super patches (if required)
def SplitDissolvedSprPatch(dissolvedSprPatch):
    global candidateSprPatches
    global candidateSprPatchesCost

    newSprPatches = [[]]
    patchesList = candidateSprPatches[dissolvedSprPatch[0]][:] #copy without
aliasing
    if len(patchesList) > 1:
        # move first item so the search can begin
        newSprPatches[0] = [patchesList[0]]
        del patchesList[0]

        # fill the newSprPatches with values
        for lst in newSprPatches:
            for patch in lst:
                neighbors=neighborList[patch][:]
                foundNeighbor = False
                CopyPatchesList = patchesList[:] # to be able to remove items
while iterating
                # loop through patche's neighbors to rearrenge patches
                for potentialNeighbor in CopyPatchesList:
                    if potentialNeighbor in neighbors:
                        lst.append(potentialNeighbor)
                        patchesList.remove(potentialNeighbor)
                        foundNeighbor = True
            # if more patches remaining that do not neighbor with the
previous patches
            if foundNeighbor == False and len(patchesList) > 0:
                # move first item so the procedure can start over
                newList = [patchesList[0]]
                del patchesList[0]
                newSprPatches.append(newList)

        # re-arrange patces ids on sprPatches list
        if len(newSprPatches) > 1:
            for i in range (1, len(newSprPatches)):
                for patch in newSprPatches[i]:
                    candidateSprPatches[dissolvedSprPatch[0]].remove(patch)
                candidateSprPatches.append(newSprPatches[i])
                candidateSprPatchesCost.append(0.0)
                dissolvedSprPatch.append(len(candidateSprPatches) - 1)

    # update cost list
    # if len = 0 means that the initial super-patch has been eliminited
    if len(candidateSprPatches[dissolvedSprPatch[0]]) > 0:
        for index in dissolvedSprPatch:
            candidateSprPatchesCost[index] =
UpdateSprPatchEnergy(candidateSprPatches,index)
    else:
           candidateSprPatchesCost[dissolvedSprPatch[0]] = 0.0 # will be
deleted later



# returns the index of the super-patch  that the input patch belongs.
def findSprPatch(patch,sprPatchesList):

    # loop that finds the super-patch and then exits

    for sprPatch in sprPatchesList:
        if patch in sprPatch:
            sprPatchIndex = sprPatchesList.index(sprPatch)
            break


    return sprPatchIndex



def LoadSemanticDistanceList():
    outputList=[[]] * 33
    csvObject = csv.reader(open("taxonomicDistance.csv", "rb"),
delimiter=",")

    i=1
    for row in csvObject:
        outputList[i]=["0.000"]
        outputList[i]=outputList[i] + row
        i = i + 1

    # make list of floats
    i=0
    for lst in outputList:
        j=0
        for items in lst:
            outputList[i][j]= float(outputList[i][j])
            j=j+1
        i= i + 1

    return outputList


# returns the area cost of the input super-patch
def AreaCost(sprPatchesList,index):
    global areaThreshold
    area=0
    cost=0

    # copy super-patch without aliasing
    patchList= sprPatchesList[index][:]
    if len(patchList) > 0:
        # prepare sql where clause
        for patch in patchList:
            if patchList.index(patch) == 0:
                wherePart = "OID = " + str(patch)
                whereClause = wherePart
            else:
                wherePart = "OR OID = " + str(patch)
                whereClause = whereClause + " " + wherePart


        with
arcpy.da.SearchCursor("data\\PatchInfo.dbf","AREA_PIX",whereClause) as
cursor:
            for row in cursor:
                 area= area + row[0]
        # assign area cost
        if area <= areaThreshold:
            cost = ((areaThreshold)-area)/(areaThreshold - 1)
    return cost


# returns the semantics cost of the input super-patch
def SemanticsCost(sprPatchesList,index):
    global selectedPatch
    global patchPreviousClass
    global areaThreshold
    cost=0
     # copy super-patch without aliasing
    patchList= sprPatchesList[index][:]
    if len(patchList) > 0:

        # prepare sql where clause and query the dbf
        for patch in patchList:
            if patchList.index(patch) == 0:
                wherePart = "OID = " + str(patch)
                whereClause = wherePart
            else:
                wherePart = "OR OID = " + str(patch)
                whereClause = whereClause + " " + wherePart

        fields = ["INIT_CLASS","CURR_CLASS","AREA_PIX"]
        with arcpy.da.SearchCursor("data\\PatchInfo.dbf",fields,whereClause)
as cursor:
```

```
            # get the patches' initial and current classes and area size in
pixels
            # and calculate the semantic distance
            for row in cursor:
                initialClass = row[0]
                currentClass = row[1]
                area =row[2]
                semanticDistance =
CalculateSemanticDistance(initialClass,currentClass)
                # assign the semantic cost
                patchCost = semanticDistance * area
                cost = cost + patchCost


    return cost

# returns the shape cost of the input super-patch
def ShapeCost(sprPatchesList,index):

    global neighborList
    global commonBoundaryList
    shapeLowerThreshold = 7
    shapeUpperThreshold = 15
    totalPerimeter = 0.0
    area = 0.0
    patchList = sprPatchesList[index][:]
    cost = 0

    if len(patchList) > 0:
        # prepare sql where clause and query the dbf
        for patch in patchList:
            if patchList.index(patch) == 0:
                wherePart = "OID = " + str(patch)
                whereClause = wherePart
            else:
                wherePart = "OR OID = " + str(patch)
                whereClause = whereClause + " " + wherePart

        fields = ["PERIMETER","AREA"]
        with arcpy.da.SearchCursor("data\\PatchInfo.dbf",fields,whereClause)
as cursor:
            # get perimeter and area to calculate shapee index
            for row in cursor:
                patchPerimeter = row[0]
                areaMeters = row[1]
                totalPerimeter = totalPerimeter + patchPerimeter
                area = area + areaMeters
        # loop through the patches neighbors to remove common
        # boundary length from the total perimeter
        for patch in patchList:
            for neighbor in neighborList[patch]:
                if neighbor in patchList:
                    neighborIndex = neighborList[patch].index(neighbor)
                    boundary = commonBoundaryList[patch][neighborIndex]
                    totalPerimeter = totalPerimeter - boundary
        shapeIndex = totalPerimeter /math.sqrt(area)
        if shapeIndex > shapeLowerThreshold:
            if shapeIndex > shapeUpperThreshold:
                cost = 1
            else: cost = (shapeIndex - shapeLowerThreshold)/
(shapeUpperThreshold - shapeLowerThreshold)

    return cost

# evaluates the proportion cost for a given class
def ProportionCost(aClass):
    global initialClassesArea
    global currentClassesArea
    global totalPixels

    initialArea = initialClassesArea[aClass]
    currentArea = currentClassesArea[aClass]

    cost = (abs(initialArea - currentArea)/ totalPixels) / 2
    return cost





def CalculateSemanticDistance(class1, class2):
    semanticList = LoadSemanticDistanceList()
    semanticDistance=semanticList[class1][class2]
    return semanticDistance



def WriteSolutionToFile(name):
    # first check if it is the best solution or an intermediate solution
    if isinstance(name, int): # if it is integer
        rasterName = "data\\results\\iteration" + str(name)+ ".tif"
    else:
        rasterName = "data\\results\\" + name + ".tif"

    # Write raster to file
    arcpy.JoinField_management("data\\patches.tif", "OID",
"data\\PatchInfo.dbf", "OID", "CURR_CLASS")
    solution = Lookup("data\\patches.tif", "CURR_CLASS")
    solution.save(rasterName)
    # delete joined field
    arcpy.DeleteField_management("data\\patches.tif", "CURR_CLASS")

# plots a graph with the energies
def plotEnergies():
    global energiesList

    # save energies to file for late proccessing
    pickle.dump(energiesList,open("energies","wb"))

    #plot energy
```

```
iteration,energy = zip(*energiesList)
plt.plot(iteration,energy)
title = "best solution = " + " " + str(bestSolution)
plt.title(title)
plt.xlabel("Iterations")
plt.ylabel("Energy")
plt.show()
```