UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Vjatšeslav Antoškin

Analysis of a Metaheuristic for Redistricting

Bachelor's Thesis (9 ECTS)

Supervisor: Benson Muite, DPhil

Tartu 2018

Analysis of a Metaheuristic for Redistricting

Abstract:
Redistricting is the process of redrawing district boundaries. It can be formulated as a combinatorial optimization problem in order to achieve a particular political objective. However, the solution space is staggering in this case, and there are no known methods to tackle the problem in a consistent manner. Metaheuristics are one of the methods that have been proposed in order to address the problem.

In this thesis, test suites are created for analysis of metaheuristics for redistricting purposes. Two versions of simulated annealing, a well-known metaheuristic, are implemented: sequential and parallel. They are analyzed using the created test suites. It is shown that simulated annealing can probably be used for redistricting of a region with a small number of districts such as the state of Iowa, which has 4 congressional districts. However, the implemented versions of simulated annealing are ineffective for regions with a larger number of districts such as the state of New York, which has 27 congressional districts.

Keywords:
Redistricting, gerrymandering, optimization, metaheuristics, simulated annealing

CERCS:
P170 Computer science, numerical analysis, systems, control
S170 Political and administrative sciences

Metaheuristika analüüs ümberjaotamise puhul

Lühikokkuvõte:
Ümberjaotamine on protsess, mille käigus paigutatakse ümber ringkonna piirid. Seda võib formuleerida kombinatoorse optimeerimise probleemina, et saavutada soovitud poliitiline eesmärk. Lahendusruum on selles olukorras hiigelsuur ja puuduvad teadaolevad meetodid järjepidevaks probleemi lahendamiseks. Metaheuristika on üks meetodeid, mis on välja pakutud probleemi lahendamiseks.

Käesolevas töös luuakse teste, et analüüsida metaheuristikaid ümberjoonistamise eesmärgil. Kaks implementeeritud simuleeritud lõõmutamise versiooni, mis on tuntud metaheuristika, on järgmised: järjestikune ja paralleelne. Eelnevalt väljatoodud versioone analüüsitakse töös loodud testide põhjal. On näidatud, et simuleeritud lõõmutamist võib kasutada piirkonna ringkondade piiride ümberjoonistamiseks vähese arvuga ringkondade puhul, näiteks Iowa osariik, millel on 4 kongressiringkonda. Aga loodud simuleeritud lõõmutamise versioonid ei ole efektiivsed piirkondades,

kus on suur arv ringkondi nagu New Yorki osariik, kus on 27 kongressiringkonda.

Võtmesõnad:
Ümberjaotamine, gerrymandering, optimisatsioon, metaheuristikad, simuleeritud lõõmutamine

CERCS:
P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)
S170 Poliitikateadused, administreerimine sotsiaalteadused

Contents

Acknowledgments

I would like to thank my supervisor Benson Muite for providing guidance, ideas and feedback during the thesis process. I am grateful to Andri Poolakese for helping me to translate the title and the abstract of the thesis into Estonian.

Also, I would like to thank the Information Technology Foundation for Education for the financial support that I have received during my studies through the IT Academy scholarship program.

Introduction

Redistricting is the process of redrawing district boundaries. Depending on the context it can mean school redistricting, state redistricting, city redistricting or any other redistricting that involves changing district boundaries of a certain region.

Redistricting is particularly relevant for electoral maps because there are significant political implications that come with redrawing those. Meaning that redrawing of electoral maps can be done to favor a certain political candidate so that he could have higher chances to win. The practice is called gerrymandering.

The problem is particularly relevant for the United States. In 2010 Republicans managed to win 680 state legislature seats and, therefore, were able to completely control the redistricting process of 190 congressional districts [Tim10]. They redraw districts in such a way so that they get as many seats in the Congress as possible [New17]. It meant that in the 2012 midterm elections Republicans gained additional 33 seats in the United States House of Representatives [Sel13].

The redrawn maps will stay the same until 2020 when the next Census takes place. It means that whoever elected in 2020 they will control the redistricting process after the Census and can redraw districts in such a way so that they could politically benefit from it. Having understood the power of redistricting, Democrats intend to invest significant resources into 2020 elections campaigns in order to prevent Republicans to control the redrawing process of so many congressional districts in 2021 [Pos18][Pol17].

In order to reduce the political bias in redistricting, there have been suggestions to automate the process using a computer[AM10]. However, it is not easy to do because the problem is challenging from the technical point of view. The main reason for that is because the number of possible ways to redraw a state that has K districts and N precincts is the Stirling number of the second kind, denoted by [Mat18]

$$S(n,k) \approx \frac{k^n}{k!}$$

Florida has 27 congressional districts, and 484,481 census blocks according to the 2010 Census [Bur]. The number of possible ways to redistrict the state is approximately

$$\frac{27^{484481}}{27!} \approx 10^{69000},$$

which is a staggering number. Even if districts are to be drawn using only census tracts, which Florida has 4,245 [Bur], the number of possible ways to do redistricting still remains huge.

Much of the research in automated redistricting has focused on using metaheuristics which allow for selective exploration of a large solution space. In recent years one of the most interesting attempts to advance research in this area are BARD [AM11a] and PEAR [LCW16]. BARD is a software package that implements the most common metaheuristics for redistricting. PEAR is a new evolutionary optimization algorithm. Its aim is to generate many high-quality redistricting maps. These generated maps can be used to detect instances of bias in redistricted plans proposed by politicians. However, the automated redistricting research has not really focused on analyzing the quality of metaheuristics in terms of its generated maps in a consistent matter. Therefore, the aim of the thesis is to analyze how good redistricting maps a metaheuristic is capable of generating.

In this thesis, two test suites were designed specifically for analyzing metaheuristics. The test suites contain test cases that have known optimal solutions. Two versions of simulated annealing, a well-known metaheuristic, were implemented: sequential and parallel. The algorithms are tested on those test suites, and the obtained results are analyzed.

The thesis is organized as follows. Chapter 1 gives background information about redistricting in the US and metaheuristics that have been used in the research related to redistricting. It also introduces some concepts from mathematical optimization and geospatial redistricting. Chapter 2 gives an overview of recent and relevant work done in the area of automated redistricting. Chapter 3 introduces the methodology that is used in analyzing a metaheuristic. Chapter 4 presents the results obtained in the thesis. Chapter 5 concludes and discusses the obtained results.

1 Background

The main focus of the background section is on redistricting in the United States for understanding the BARD [AM11a] and PEAR [LCW16] papers. Even though the focus is on the United States, political redistricting is not specific only to it. Germany is subdivided into electoral districts before a parliamentary election [GW17]. In England, local government electoral boundaries are redrawn to account for population changes that happen over time [Ral+04].

The chapter gives a brief overview of redistricting in the United States, mathematical optimization, metaheuristics and geospatial concepts.

1.1 Redistricting in the United States

The section is based on the book "A Citizen's Guide to Redistricting" by Justin Levitt and Erika L. Wood [LW10].

The United States is a *federal state* — a union of partially self-governing states united by the central government. The U.S. states have a high degree of autonomy in terms of how they handle their internal affairs. One of the state institutions that is directly involved in state internal affairs is a *state legislature* which is a legislative branch of a state. Likewise, U.S. states are indirectly involved in the process of shaping national policies through their members in *Congress* which is a legislative branch of the central government that consists of two chambers: the *Senate* and the *House of Representatives*.

The members of Congress who serve in the Senate are called *senators*. Each state is allocated two senator seats independent of its population size (100 senators in total). Senators serve six-year terms, but approximately one-third of the Senate seats are up for election every two years such that both seats from the same state are not contested in the same election. Senators are elected in direct *statewide* elections, meaning that voters throughout a whole state select from the same candidates for a contested Senate seat. As such, there is no need to divide a state into electoral districts for a Senate election.

The members of Congress who serve in the House of Representatives are called *representatives*, *congressmen* or *congresswomen*. The House is made up of 435 representatives, and they are elected every two years by *congressional districts* which are geographic areas of a state that are established specifically for the purpose of elections to the House of Representatives. Every congressional district elects only one representative through the *winner-take-all* system — a system in which

a candidate who gets the most votes wins. However, unlike state boundaries, congressional districts boundaries change over time due to *reapportionment* and *redistricting*.

Reapportionment is the process of deciding how many seats in the House should be allocated to each state, meaning that after reapportionment states might gain or lose seats. As U.S. states tend to experience population changes over time, reapportionment is required in order to ensure that states are represented in the House in proportion to their populations. It is conducted after the federal Census, which is required by the United States Constitution to be taken every 10 years.

Redistricting is the process of redrawing electoral district boundaries. The term refers to redrawing congressional districts or *congressional redistricting* as well as to redrawing state legislative districts or *legislative redistricting*. Redistricting is *usually* conducted after reapportionment, but *not always*. The reason for that is because each state has its own set of rules regarding how redistricting should be handled: who should conduct it, who should be involved in the process, when it should be conducted.

In most states, state legislatures are responsible for both congressional and legislative redistricting. Therefore, a political party that has control over a state legislature can redraw district boundaries in order to maximize their chances of winning the most seats in an election. This practice is called *gerrymandering*.

1.2 Mathematical Optimization

Optimization is a broad term that can be understood differently depending on the context. However, the common underlying idea can be phrased in the following way: selecting the best element from the set of all possible elements which is called the *optimal solution*.

Problems in mathematical optimization are called *mathematical optimization problems* or *optimization problems* and can be formulated as follows [NW06]:

$$\text{minimize (maximize)} \quad f(\mathbf{x}) \tag{1}$$

$$\text{subject to} \quad c_i(\mathbf{x}) \leq b_i, \quad \forall i \in \{1, ..., m\}. \tag{2}$$

where the vector of *variables* $\mathbf{x} = (x_1, ..., x_n)$ is called the *optimization variable*, $f : \mathbb{R}^n \to \mathbb{R}$ is the *objective function* of the vector $\mathbf{x}$ that we want to minimize (maximize), the *constraint functions* $c_i : \mathbb{R}^n \to \mathbb{R}$ and constants b_i form inequality *constraints* that the unknown vector $\mathbf{x}$ must satisfy.

There are different types of optimization problems. For example, one might be *continuous*: variables are allowed to take any value over a continuous range, and the other one might be *discrete*: variables are drawn from a finite set of values [BV04].

Combinatorial optimization problems belong to the class of discrete optimization problems. Examples of combinatorial optimization problems are the Travelling Salesman Problem, timetabling and scheduling problems. For some combinatorial optimization problems, there are no reasonable algorithms that guarantee to find the optimal solution. Therefore, metaheuristics are one of the methods that are used to tackle this class of problems. Metaheuristics that are usually for solving combinatorial optimization problems are ant colony optimization, evolutionary algorithms, simulated annealing.[BR03]

1.3 Metaheuristics

A *metaheuristic* is an algorithmic framework that that is used to build heuristic optimization algorithms [GS15]. Metaheuristics are abstract in nature and do not provide specific steps in order to find any solution not necessarily optimal (it all depends on the problem domain). Instead, they give a framework that will guide the search for the solution, but the user has to provide steps as to how a solution should be found and what constitutes a good solution. Many metaheuristics are stochastic in nature and tend to employ randomness to aid the search for the optimal solution.

Unlike exact methods that guarantee with a proof that the optimal solution will be found in a finite amount of time, metaheuristics do not make such guarantees. However, when there are no known exact methods to solve the problem or exact methods take a large amount of time to find the optimal solution, then metaheuristics might be a good choice to try.

Local search, constructive and population-based metaheuristics are the three major classes of metaheuristics algorithms. *Local search* metaheuristics explore the solution space by iteratively making changes, usually small, to the current solution in order to move to a neighbor solution. Local search metaheuristics tend to get trapped in the local optimum. Therefore they usually employ a strategy in order to escape the local optimum and continue the search for the global optimum. This strategy is usually the main characteristic of a local search algorithm. Members of this class are Simulated annealing and tabu search. *Constructive metaheuristics* do not improve existing solutions but rather constructing a new solution from scratch by adding one element at a time. An example of constructive metaheuristics

is GRASP (Greedy Randomized Adaptive Search Procedure). *Population-based* metaheuristics generate a set of solutions called population and iteratively select and combine solutions from this set in order to find new solutions. Evolutionary algorithms belong to this class of metaheuristics. [GS15]

The sections below introduce in more detail some of the metaheuristics that have been used in the research related to redistricting.

1.3.1 Evolutionary Algorithms

This section is based on the first two chapters of the book *Introduction to Evolutionary Computing*, by Agoston E. Eiben and Jim E. Smith [ES03].

Evolutionary computation is a field of computer science that studies algorithms based on the process of natural evolution. One of the main subfields of evolutionary computation is *evolutionary algorithms* or *EAs*. The main idea behind these algorithms is to find the globally optimal solution to a problem by repeatedly improving the fitness of a population using natural selection. The pseudocode of the evolutionary algorithm is shown in Algorithm 1.

Algorithm 1 Pseudocode of the evolutionary algorithm

```
1: Randomly generate an initial population of individuals
2: Evaluate the fitness of each individual in the initial population
3: while not terminated do
4:     Select individuals for reproduction
5:     Pair individuals and produce offspring through crossover
6:     Mutate the offspring
7:     Evaluate new individuals
8:     Select individuals for the next generation from the new and old individuals
```

The *creation of an initial population (generation)* is the first step in the evolutionary algorithm. It usually involves utilizing randomization so that the composition of the initial population is different each time the EA is run. The size of the initial generation can be set to any positive number, and it usually does not change in future generations.

A *fitness* or *evaluation* function is a function that evaluates an *individual*, also called a *candidate solution*, and assigns it a *fitness score*. The fitness function is used to guide the EA towards the globally optimal solution, and is designed specifically for each problem because it needs problem-specific information in order

to work effectively.

The selection of *parents* (*individuals for reproduction*) is the process of selecting individuals that will create offspring during the variation process. The selection is typically a probabilistic process that gives fitter individuals a higher chance to become parents rather than those that are less fit. The reason for giving less fit individuals a positive chance is to preserve variety in population and to avoid getting stuck in a local optimum.

The *variation operators* are used to create new individuals from old ones. There are two types of variation operators in the EA. The first one is a *crossover* operator which is a binary operator that takes two parents as inputs and produces a new individual. The information to produce a child is taken from parents in a random way. Crossover operator with higher arity are possible but not commonly used. The second one is a *mutation* operator which is a unary operator that takes an individual as an input and returns a slightly modified version of it by making random changes to the individual.

A *replacement strategy* is a mechanism that replaces a subset of the current population with a subset of new individuals that are created during the variation process. A replacement strategy is usually deterministic and *fitness-biased*, meaning that offspring and old individuals (those that represent the current generation) are ranked and the top segment is chosen for the next generation. There are also other replacement strategies such as the *age-biased* strategy that takes individuals for the next generation only from offspring.

1.3.2 Simulated Annealing

Simulated annealing or *SA* is a metaheuristic that was first introduced by S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi in their seminal paper *Optimization by simulated annealing* [KGV83]. The algorithm was adapted from the Metropolis algorithm [Met+53] in order to solve optimization problems [KGV83]. The name for the metaheuristic was taken from a metallurgical technique called *annealing* [Luk13] which is used for heating and cooling down materials in order to make them more workable.

Algorithm 2 Pseudocode of simulated annealing

```
1: procedure SA(maxIterations, maxTemp)
2:     current = createInitialSolution()
3:     curEnergy = calculateEnergy(current)
4:     best = current
5:     bestEnergy = curEnergy
6:     for i ∈ {1, ..., maxIterations} do
7:         temp = calculateTemp(i, maxTemp)
8:         neighbor = findNeighbor(current)
9:         neighborEnergy = calculateEnergy(neighbor)
10:        if neighborEnergy ≤ curEnergy then
11:            current = neighbor
12:            curEnergy = neighborEnergy
13:            if neighborEnergy ≤ bestEnergy then
14:                best = neighbor
15:                bestEnergy = neighborEnergy
16:        else if rand(0, 1) < calcAcceptanceProbability() then
17:            current = neighbor
18:            curEnergy = neighborEnergy
19:    return best
```

Algorithm 2 demonstrates the pseudocode of simulated annealing. Simulated annealing as any other metaheuristic does not provide steps as to how a solution should be generated, meaning that functions *createInitialSolution*, *calculcateEnergy* (which is basically an objective function) and $findNeighbor$ have to be implemented by a developer.

The *annealing (cooling) schedule* and the *acceptance probability* are the components that are intrinsic to simulated annealing. The annealing schedule represented by *calculateTemp* in Algorithm 2. It determines the temperature of the neighboring solution. Based on the temperature the acceptance probability is calculated.

The acceptance probability determines if a neighboring solution that is worse the current one based on the energy (objective) function can replace the current solution. In the provided pseudocode the acceptance probability represented by *calcAcceptanceProbability*.

1.3.3 Tabu Search

Tabu search is a metaheuristic that was introduced by Fred Glover in 1986 [Glo86]. It is a modified version of hill-climbing that retains the history of recent moves, also called *short-term memory*, and avoids those moves unless they meet particular *aspiration criterion* which is the criterion that determines if a candidate is available for selection. The pseudocode of tabu search that uses short-term memory is presented in Algorithm 3.

Algorithm 3 Pseudocode of tabu search with short-term memory

```
1:  procedure TABUSEARCH
2:      bestSln = createInitialSolution()
3:      while not stopCondition() do
4:          candidateList = createCandidateList()
5:          bestCandidate = removeFirstCandidate(candidateList)
6:          for candidate ∈ candidateList do
7:              if not isTabu(candidate) or isAspirationSatisfied(candidate) then
8:                  bestCandidate = chooseBestCand(bestCandidate, candidate)
9:          bestSln = chooseBestSln(bestCandidate, bestSln)
10:         updateTabuRestrictions()
11:         updateAspirationCriteria()
12:     return bestSln
```

1.3.4 GRASP

This subsection is based on the chapter "Greedy Randomized Adaptive Search Procedures" authored by Mauricio G.C. Resende and Celso C. Ribeiro in *Handbook of Metaheuristics* [RR03].

GRASP or *Greedy Randomized Adaptive Search Procedures* is a metaheuristic for combinatorial problems that was first proposed by Mauricio G.C. Resende and Thomas A. Feo in 1989 [FR89]. The meta-algorithm is characterized by an iterative process that consists of two main phases: construction and local search.

Algorithm 4 Pseudocode of GRASP

```
1: procedure GRASP(maxIterations)
2:     bestSolution = createRandomSolution()
3:     for i ∈ {1, ..., maxIterations} do
4:         solution = createGreedyRandomizedConstruction()
5:         solution = localSearch(solution)
6:         bestSolution = getBestSolution(solution, bestSolution)
7:     return bestSolution
```

The goal of the construction phase is to generate the starting solution of an iteration in GRASP. The phase is the reason why the algorithm is *random*, *adaptive* and *greedy*. The greed of the algorithm lies in the creation of a *restricted candidate list* or a *RCL*. The list is formed by elements "whose incorporation into the current partial solution results in the smallest incremental costs" (direct quote; might require paraphrasing). The randomness of the algorithm comes from the fact that an element that should be added to the partial solution is selected at random. The adaptiveness of the algorithm resides in the update of the RCL through recalculation of incremental costs of elements.

Algorithm 5 Pseudocode of GRASP greedy randomized construction

```
1: procedure CREATEGREEDYRANDOMIZEDCONSTRUCTION
2:     solution = ∅
3:     Get candidate elements and evaluate their incremental costs
4:     while solution is not a complete solution do
5:         Build the restricted candidate list (RCL)
6:         Select an element s from the RCL at random
7:         solution = solution ∪ {s}
8:         Reevaluate the incremental costs
9:     return solution
```

The aim of the local search phase is to improve the starting solution generated during the construction phase. Local search strives to achieve this aim by finding a neighbor of the current solution who has a lower score (minimization) according to the objective function. Once the current solution is changed by a neighbor, a new iteration starts and the local search function will try to improve the new solution through its neighbors. The process continues until a better (locally optimal) solution could not be found in an iteration.

Algorithm 6 Pseudocode of GRASP basic local search

```
1: procedure LOCALSEARCH(solution)
2:     while solution is not locally optimal do
3:         Find s' ∈ N(solution) such that f(s') < f(solution)
4:         solution = s'
5:     return solution
```

There are two strategies for searching the solution neighborhood (line 3 in Algorithm 6). The first one is *first-improving*. It replaces the current solution with the first neighbor that is better than the current solution. The second one is called *best-improving* which searches through all neighbors of the current solution in order to find the best neighbor that will replace the current solution.

1.4 Geospatial Redistricting Concepts

Geospatial analysis or *spatial statistics* is the process of understanding and interpreting geospatial data. There are several concepts from the geospatial analysis that are used in redistricting. The first concept is the *hole-free requirement* which prohibits one district from being encircled by another region [Kin+12]. If an encircled district is located on the boundary of some geographic area, as illustrated, for example, in Figure 1, then, in this case, the hole-free requirement is intact.

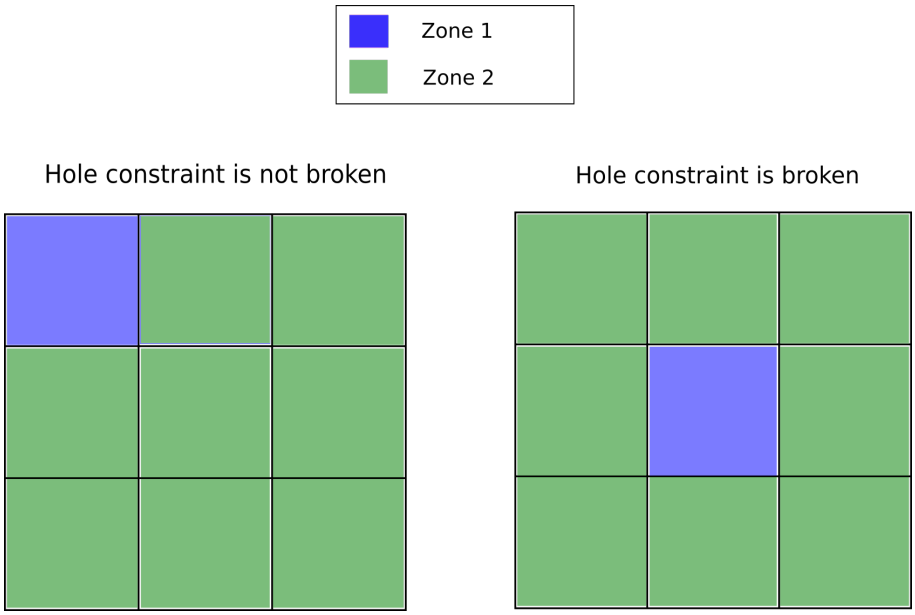

Figure 1. Examples of the broken and the unbroken hole-free requirement

A district is said to be *contiguous* if all parts of the district are connected at some point with the rest of the district [Sta]. A contiguous district is formed by connecting each element of the zone with its neighbors. There are different ways as to how to determine if an element is a neighbor of another element. The two most popular ones are: *rook* and *queen* neighborhoods. They are both illustrated in Figure 2.

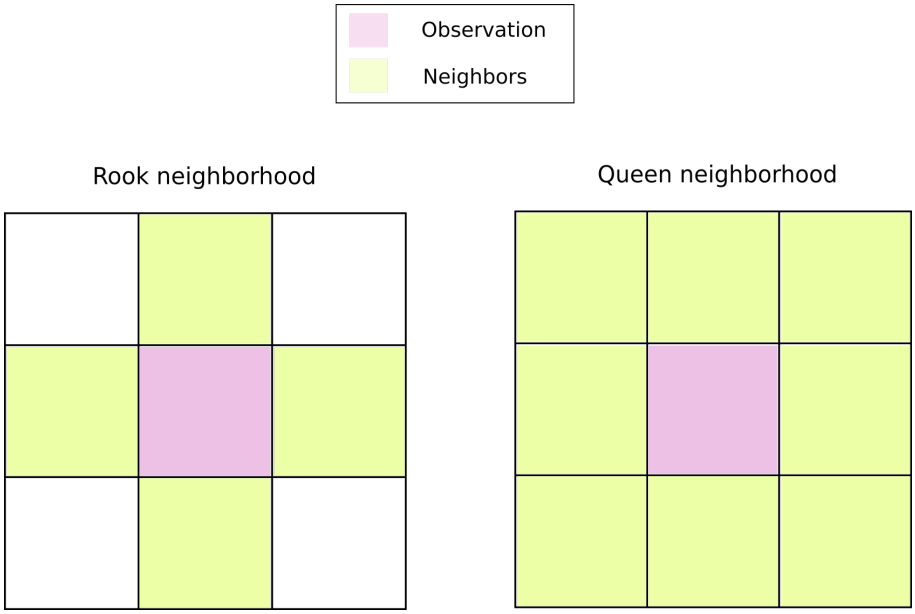

Figure 2. Rook and queen neighborhoods

Compactness is also one of the geospatial concepts that often appears in the redistricting literature. It is the rule for addressing the geometric shape of a district [LW10]. National Conference of State Legislatures of the United States defines compactness as "Having the minimum distance between all the parts of a constituency (a circle, square or a hexagon is the most compact district)" [Sta]. However, there is no legal definition that would allow to determine if a district is compact or not. As a result, measuring compactness is open to interpretation. One of the ways to measure district compactness is to use the *Polsby-Popper* score which is defined as

$$C = \frac{4\pi A}{p^2},$$

where A is the area of the district, p is the perimeter of the district [CJ14].

2 Related work

There has been a substantial research to address the redistricting problem [LCW16]. Yan Liu, one of the authors of the PEAR algorithm [LCW16], researched in his doctoral dissertation as to how large political redistricting problems could be tackled [Liu17]. Myung Jin Kim explored the application of metaheuristics for redistricting in her doctoral dissertation [Kim11]. The results that are presented in the dissertation correspond to some of the results obtained in the thesis and will be discussed in Chapter 5.

Due to the limited scope of the thesis the chapter focuses on the PEAR [LCW16] and BARD [AM11a] papers. The PEAR paper is discussed because this thesis was largely motivated by that paper. The BARD paper is included in this chapter because it is the most well-known open-source redistricting software package, and PEAR is tested against BARD [AM11b].

2.1 BARD

BARD, an abbreviation for *Better Automated Redistricting*, is an open source software package for general redistricting and redistricting analysis. The initial version of BARD was released in 2007. It is primarily written in R with some parts written in C for better performance. BARD is not the only available software package for redistricting, but most other packages with similar features are proprietary and expensive or not publicly available. By open-sourcing BARD, authors aim to increase public involvement in the redistricting process and allow anyone to do redistricting.

BARD consists of four components. The first component reads redistricting plans in the shapefile format. The second component evaluates read redistricting plans. The third component can generate initial redistricting plans if none are provided in the first component. This component is also responsible for refining plans. It does that using one of the five metaheuristics: GRASP (Greedy Randomized Adaptive Search), simulated annealing, tabu search, greedy search and genetic algorithms.

The main shortcoming of BARD is that it does not give any guarantees as to whether the metaheuristics that are used in the refinement process perform well and are capable of finding optimal or near-optimal solutions for redistricting.

2.2 PEAR

PEAR or *Parallel Evolutionary Algorithm for Redistricting* is an evolutionary algorithm whose main goal is to help to detect instances of gerrymandering in redistricting maps proposed by lawmakers. The algorithm does that by producing a large number of redistricting plans that satisfy the most common redistricting criteria such as contiguity, compactness and hole-free constraints. Produced plans can then be used in the statistical analysis that, in turn, can help to tell if a map proposed is an instance of gerrymandering.

PEAR approaches the redistricting problem as an optimization problem. First, it imposes constraints on what on what can be qualified as a feasible (possible) solution (map). The first constraint is that a unit, the underlying element of a solution that can be thought of as a census block or a county, must belong exactly to one district. The second constraint is that the population deviation across all K districts cannot exceed some value M or

$$\left|P_{d_i} - P_{d_j}\right| \leq M, \text{ for } i, j = \{1, 2, ..., K\}, \tag{3}$$

where d_i and d_j are any districts. The other constraints are contiguity, the hole-free constraint and compactness.

PEAR also defines a weighted multi-objective function that it seeks to minimize. The function consists of three other objectives: compactness, which is an objective and a constraint at the same time, competitiveness, the population deviation. The first objective is compactness objective which is defined as $1 - C$, where

$$C = \frac{4\pi A}{p^2} \tag{4}$$

and A is the district area and p is the district perimeter.

The second objective that PEAR seeks to minimize is the population deviation. It is formulated as

$$g = \begin{cases} p & \text{if } p \leq 1 \\ 1 & \text{if } p \geq 1 \end{cases}, \quad \text{where} \quad p = \frac{\max\limits_k(P_k) - \min\limits_k(P_k)}{\bar{P}} \tag{5}$$

p measures the population deviation across districts, and the lower its value, the less there is the population deviation in a solution. $\bar{P}$ is the average population across all districts. $\max_k(P_k)$ and $\min_k(P_k)$ are the maximum population and the minimum populations across among all districts respectively. The value of p can potentially exceed 1. In this case, the g function sets the p value to 1 for

normalization purposes so that the p range is between 0 and 1.

The third objective is competitiveness, and it is defined as

$$f = T_p(1 + \alpha T_e)\beta, \quad \text{where}$$

$$T_p = \frac{1}{K}(\sum_{k=1}^{K} \left| \frac{R_k}{D_k + R_k} - \frac{1}{2} \right|), \quad T_e = \left| \frac{B_R}{K} - \frac{1}{2} \right| \tag{6}$$

$$\text{for } 0 \leq T_p \leq 0.5, \ \ 0 \leq T_e \leq 0.5, \ \text{ and } \ \alpha, \beta \geq 0$$

The aim of the f function is to ensure that the support for the Democratic and the Republican party is more or less equal across all districts. K is the total number of districts. The competitiveness function consists of two components. The first component is T_p. It requires information the Democratic party registration, D_k, and the Republican party registration, R_k, in district k, where $k \in [1 \dots K]$. T_p measures to what extent the Republican proportion of the two-party registration differs from 0.5 across all districts. Due to its design $T_p \in [0, 0.5]$. T_e is a weighting factor for T_p. T_e measures to what extent the distribution of district seats between the two parties is unequal. 0 means that Democrats and Republicans have an equal number of seats, while 0.5 means that one party has all the seats. T_e uses information about the number of districts where the Republican party registration is large than the Democratic party registration, which is represented as B_R. $\alpha = 1$ and $\beta = \frac{4}{3}$ so that $f \in [0, 1]$.

PEAR is an evolutionary algorithm and its basic structure follows Algorithm 1. The main distinguishable features of PEAR are its variation operators: crossover and mutation. The operators are the algorithms that explore the solution space in order to find good solutions.

Algorithm 7 Pseudocode of the mutation operator

```
1: procedure MUTATE(solution)
2:     Generate a random sequence s of size K using the Fisher-Yates algorithm
3:     for dist ∈ s in solution do
4:         if dist is a source district of previous shifts then
5:             continue
6:         if population in dist is below a threshold then
7:             continue
8:         Randomly select a dstDist from neighboring districts of dist
9:         shift(dist, dstDist)
10:
11: // Select a subset of units from the source district (srcDist) to the destination
    district (dstDist)
12: procedure SHIFT(srcDist, dstDist)
13:     Randomly select up to two adjacent units in srcDist bordering dstDist
14:     subDist = Selected units
15:     while |subDist| < maxMutUnits do
16:         Find neighbor units U of subDist
17:         Generate a random number q ∈ 1, 2, ..., |U|
18:         Randomly choose a subset U' ⊂ U, where |U'| = q
19:         subDist = subDist ∪ U'
20:     dstDist = dstDist ∪ subDist
21:     srcDist = srcDist - subDist
```

The pseudocode of the mutation operator is shown in Algorithm 7. The main goal of the mutation operator is to explore the neighborhood of a potential solution in order to find out if the neighborhood contains better solutions than the current one. The algorithm does that by modifying the passed solution. The modifications involve looping over all districts and randomly changing the affiliation of units to districts.

Algorithm 8 Pseudocode of the crossover operator

```
1: procedure CROSSOVER(solution1, solution2)
2:     for unit in units do
3:         Calculate split label as z1z2 where z1 is the district index of unit in
   solution1, and z2 is the district index of unit in solution2
4:     Form splits by grouping units with the same split label into the same split
5:     newSplits = ∅
6:     for split ∈ splits do
7:         while split ≠ ∅ do
8:             Find unit u in split, construct a spanning tree in split, rooted at u
9:             Form new split split′ to include all of the units on the spanning tree
10:            newSplits = newSplits ∪ split′
11:            Remove all the units in split′ from split
12:    Construct a split graph from newSplits that defines relationships such as
   adjacency between splits in newSplits
13:    Check for holes and repair
14:    Generate a split-level solution using the constructed split graph
15:    Convert the split-level solution to unit-level solution newSolution
16:    return newSolution
```

The pseudocode of the crossover operator is demonstrated in Algorithm 8. The main aim of the *crossover* operator is to make "jumps" in the solution space or, in other words, generate solutions that are substantially different from the current ones. This is needed in order to prevent the search from getting stuck in local optima. The crossover operator makes those "jumps" by merging two passed solutions into one. The merge process is basically done the following way. First, all units in a region are labeled based on their district affiliations in *solution*1 and *solution*2. For example, if a unit belongs to *district* 12 in *solution*1 and to *district* 4 in *solution*2, then its label will be 124. Then units are grouped into *split* by their labels and placed into *splits*. Then each *split* in *splits* is checked in order to make sure that *split* is contiguous and not scattered over the region and placed into *newSplits*. If *split* not contiguous, then new splits are formed from it. Then after additional operations a new solution is formed using *newSplits*.

PEAR is implemented in ANSI C. In [LCW16] comparisons of metaheuristics of PEAR and BARD are made. To be precise, the PEAR algorithm is compared against BARD versions of simulated annealing, greedy search, tabu search and GRASP (default). In [LCW16] a modified version of GRASP (modified) was created because the BARD version of the algorithm violates the contiguity constraint. In [LCW16] a modified version of the objective function is used for evaluating the

solution quality because the standard BARD implementation calculates compactness too slowly.

In [LCW16] it is reported that the sequential PEAR algorithm outperforms BARD metaheuristics that are also sequential. The size of the region on which both algorithms were tested is 2690 units and 13 districts [Liu18]. After 45,358 iterations PEAR managed to reach a fitness value of 0.0403, and after 329,572 iterations it reached a fitness score of 0.0145. It took about 3 hours for the PEAR algorithm to complete 329,572 iterations. The BARD metaheuristics performed the following way: simulated annealing reached a score of 0.1237 after 1472 iterations, greedy search reached 0.0980 after 186,619 iterations, tabu search reached 0.0984 after 92,659 iterations and GRASP (default) reached 0.0980 after 186,619 iterations. The modified version of GRASP is the only metaheuristic that came close to the PEAR performance. It reached a fitness score of 0.0411 after 320,386 iterations, which took about 5 hours to complete.

3 Methods

3.1 Test Suites

Metaheuristics do not guarantee to find the optimal solution. To determine if they are effective to tackle a certain problem, they need to be tested. Therefore, specific test cases need to be designed on which metaheuristics could be tested. The test cases should contain known optimal solutions for two reasons. The first one is to check if a metaheuristic finds the known optimal solution. The second reason is to compare the optimal solution with the best solution that a metaheuristic managed to find.

The thesis mainly focuses on redistricting in the United States. Therefore, test cases were designed to be simplified versions of states of the United States. Each test cases consists of *units*. Units are the smallest geographic entities of a test case. Each unit has a population of size 100. A population consists only of Democratic party and Republican party supporters. The level of support that both parties have in each unit is known, but it varies among units. Appendix A and Appendix B show how the political support is distributed in some test cases of Test Suite 1 and Test Suite 2

Test cases were created such that there is at least one optimal solution which has a fitness score of 0 according to the objective function, which is discussed in next section. To achieve this, test cases were designed such that optimal solutions represent grids that contain districts of a square shape. The districts in optimal solutions are stacked on top of each other. The only exception to the described rule is Region 3x3 in Test Suite 2 which was designed manually. Some optimal solutions for Test Suite 1 and Test Suite 2 are demonstrated in Appendix C and Appendix D respectively.

Within the scope of the thesis, two test suites were created in order to test simulated annealing. Test cases in these test suites use rook adjacency, meaning that a unit can have at most 4 neighbors.

The first test suite, *Test Suite 1*, consists of 7 test cases: Region 4x4, Region 6x6, Region 8x8, Region 10x10, Region 12x12, Region 14x14, Region 16x16. The distinguishing feature of the test suite is that the known optimal solutions of the test cases assume that each test case will be divided into 4 districts. This test suite will help to determine how simulated annealing works when the number of districts is small.

The second test suite, *Test Suite 2*, consists of 5 test cases: Region 3x3, Region

6x6, Region 18x18, Region 24x24 and Region 30x30. The distinguishing feature of the test suite is that the optimal solutions of the test cases have a different number of districts, and the number of districts increases as the number of units increases. To be specific, the known optimal solutions assume that Region 3x3 will be divided into 3 districts, Region 6x6 into 4 districts, Region 18x18 into 9 districts, Region 24x24 into 16 districts and Region 30x30 into 25 districts.

3.2 Objective function and Constraints

The objective function of simulated annealing, h, is a weighted multi-objective function adapted from the PEAR algorithm [LCW16]. It is defined as $h = \frac{1}{2}f + \frac{1}{2}g$, where $0 \leq f, g \leq 1$. It uses two objectives: competitiveness, f from Equation (6), and the population deviation, g from Equation (5). For simplicity, the compactness objective was not used.

In order for a solution to be considered feasible, it must satisfy three constraints: contiguity, the hole-free constraint, and a unit must belong to only one district. These constraints are also used in [LCW16]. For simplicity compactness and the population deviation were not used as constraints as it was done in [LCW16].

Simulated annealing is implemented in such a way that it guarantees that a unit belongs to only one district. The other constraints, however, are not guaranteed and need to be enforced. Two algorithms were designed in order to help with this.

Algorithm 9 Pseudocode of the contiguity checking algorithm

```
1:  function checkIfContiguous(district)
2:      districtUnits = getUnits(district)
3:      root = getFirstElement(districtUnits)
4:      queue = ∅
5:      result = ∅
6:      result.add(root)
7:      while queue not empty do
8:          localRoot = queue.pop(0)
9:          neighbors = district.getDistrictNeighborsOf(localRoot)
10:         for unit in neighbors do
11:             if unit not in result then
12:                 result.add(unit)
13:                 queue.append(unit)
14:     return result.size > 0 and result.size == districtUnits.size
```

The contiguity checking algorithm, illustrated in Algorithm 9, helps to enforce the contiguity constraint. The algorithm is an application of the depth-first search algorithm. It takes the first unit that belongs to the passed district. Then using the taken unit, it traverses the district recording all units that it visited. If after the completion of the traversal, there are units that have not been visited, it means that the district is not contiguous and is scattered over the region. In this case, the algorithm returns False. If the district is contiguous, it returns True.

Algorithm 10 Pseudocode of the hole checking algorithm

```
1: function checkIfHasHoles(region)
2:     districts = getDistricts(region)
3:     for dist in districts do
4:         neighboringDistricts = getNeighboringDistrictsOf(dist)
5:         if neighboringDistricts.size > 1 then
6:             continue
7:         unitsOnRegionBorder = getDistrictUnitsOnRegionBorder(dist)
8:         if unitsOnRegionBorder.size == 0 then
9:             return True
10:    return False
```

The hole checking algorithm, demonstrated in Algorithm 10, helps to identify solutions that violate the hole-free constraint. The approach as to how to check for holes in a region is borrowed from [LCW16]. The algorithm checks if a region contains holes by examining each district in the region and controlling if a district satisfies two conditions. First, it checks if a district has only one neighboring district. Second, it checks if a district has at least one unit that is located on the region border. If both conditions are met, then the district in question is a hole, and the region violates the hole-free constraint. In this case, the algorithm returns True. If there are no districts that satisfy both conditions, then the region does not contain any holes. In this case, the algorithm returns false.

3.3 Sequential Simulated Annealing

The metaheuristic that was chosen for analysis in redistricting tasks is simulated annealing or *SA*. It is implemented in Python. The basis of the algorithm follows the pseudocode of simulated annealing described in Algorithm 2.

To control how the temperature decreases in simulated annealing, the *geometric cooling schedule* was selected. It is one of the most common annealing (cooling) schedules and its analytic form looks the following way [Yan11]:

$$T(i) = T_0 \times c^i, \quad \text{where} \quad 0 < c < 1, T_0 > 0 \text{ and } i \in \{1, \ldots, n\} \tag{7}$$

In the function T_0 represents the *initial temperature* or the *maximum temperature*, and c is a *cooling factor* which determines how fast temperature decreases.

The implementation of simulated annealing use $T_0 = 10000$ and $c = 0.9967$. The values were determined experimentally so that the temperature would not get too close to 0 until SA makes at least 5000 iterations.

Algorithm 11 Pseudocode of the algorithm that generates initial solutions

```
1:  function generateInitSolution(k)
2:      seeds = ∅
3:      unitsOnBorder = getUnitsOnRegionBorder()
4:      while seeds.size < k do:
5:          unit = getRandomElement(unitsOnBorder)
6:          if unit not in seeds then
7:              seeds.add(unit)
8:      region = createDistrictForEachUnit(seeds)
9:      pool = seeds
10:     while pool is not empty do:
11:         unit = pool.pop()
12:         districtOfUnit = findDistrictByUnit(unit)
13:         neighbors = getNeighborsOf(unit)
14:         for neighbor in neighbors do
15:             neighborDistrict = findDistrictByUnit(neighbor)
16:             if neighborDistrict == None and neighbor not in pool then:
17:                 pool.add(neighbor)
18:                 districtOfUnit.add(neighbor)
19:     return region
```

Algorithm 11 shows the algorithm that generates initial solutions. The algorithm is adapted from the initial solutions generating algorithm of the PEAR algorithm [LCW16]. The contiguity and the hole-free requirement are incorporated into the algorithm. It means that solutions generated by this algorithm are guaranteed to be contiguous and not to contain holes. The algorithm works the following way. First, it finds all the units that are located on the region border. After that, it randomly selects k units and creates first k out of them. Then remaining units are added to the created k districts. The adding process is done by finding all

neighbors of a unit and adding them to the same district. If a neighbor already belongs to another district, then it is not added.

Another algorithm that generates initial solutions was also implemented for testing purposes. Unlike Algorithm 11 it selected the first k units from all units of a region so that initial solutions would contain districts that are not located on the region border. However, after some testing of the strategy, it does not seem to perform much better than Algorithm 11.

Algorithm 12 Pseudocode of the function that generates neighbor solutions

```
1: function createNeighboringSolution(region)
2:     units = getAllUnits()
3:     for unit ∈ units do
4:         curDistrict = region.findDistrictByUnit(unit)
5:         if curDistrict.units() ≤ 1 then
6:             continue
7:         neighbors = getNeighborsOf(curDistrict)
8:         borderingDistricts = getDistrictsBorderingUnit(unit)
9:         if borderingDistricts == 0 then
10:            continue
11:        rdDistrict = selectRandomly(borderingDistricts)
12:        rdDistrict.remove(unit)
13:        curDistrict.add(unit)
14:        if not checkIfContiguous(curDistrict) and checkIfHasHoles(region)
   then
15:            curDistrict.remove(unit)
16:            rdDistrict.add(unit)
17:    return region
```

Algorithm 12 demonstrates is the algorithm that is responsible for generating candidate solutions. The inspiration for the approach that the algorithm uses to create neighbors (maps) comes from the Metropolis algorithm applied to the 2-dimensional Ising model [Kot08]. *createNeighboringSolution* creates a new region using the passed region. It does this by changing the affiliation of units to districts. To be precise, units that are located on the border of their districts and border units of other districts are "moved" to other districts. The algorithm utilizes the contiguity checking algorithm and the hole checking algorithm because a move can potentially break the contiguity and the hole-free constraint. If a move violates one of these constraints, then the move is reverted, and the unit is placed back into the district to which it belonged to previously.

3.4 Parallel Simulated Annealing

The parallel version of simulated annealing was implemented using Python and mpi4py, which is a package that provides Python bindings for *MPI* or *Message Passing Interface* standard. The implementation of the standard that was used to run the parallel simulated annealing is *OpenMPI*.

Algorithm 13 Pseudocode of parallel simulated annealing

```
1: function parallelSA
2:     Initialise maxIterations, numberOfDistricts, maxTemp
3:     rank = getRank()   ▷ rank is the process id. Its value ranges from 0 to n
4:     size = getSize()   ▷ size tells how many processes were launched
5:     results = None
6:     if rank == 0 then
7:         results = ∅
8:     bestRegion, bestRegionFitnessScore = runSequentialSA(rank, maxTemp
   maxIterations, numberOfDistricts)
9:     results = gatherResults((rank, bestRegion, bestRegionFitnessScore),
   root=0)
10:    if rank == 0 then
11:        return results
```

The parallel version of simulated annealing was created by parallelizing the sequential version. The parallelization is achieved the following way. First, n parallel processes of simulated annealing are created. The pseudocode of a parallel process is shown in Algorithm 13. Each process launches the sequential version of simulated annealing. After the sequential version finishes running, it returns *bestRegion* and *bestRegionFitnessScore*. Then all results are gathered by process 0 into variable *results*. Once all results are gathered, process 0 returns them.

4 Results

The parallel and sequential versions of simulated annealing were run on the Rocket cluster of the University of Tartu. The cluster has 135 nodes and each node has 20 cores of 2 x Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz [Hig]. The system through which the algorithms were run is called *SLURM* or *Simple Linux Utility for Resource Management* which is a cluster management and job scheduling system for Linux clusters [Sch13].

4.1 Sequential Simulated Annealing

The sequential simulated annealing algorithm was run on Test Suite 1 with the following parameters: 5000 iterations, maximum temperature 10000. The algorithm utilized one core while running.

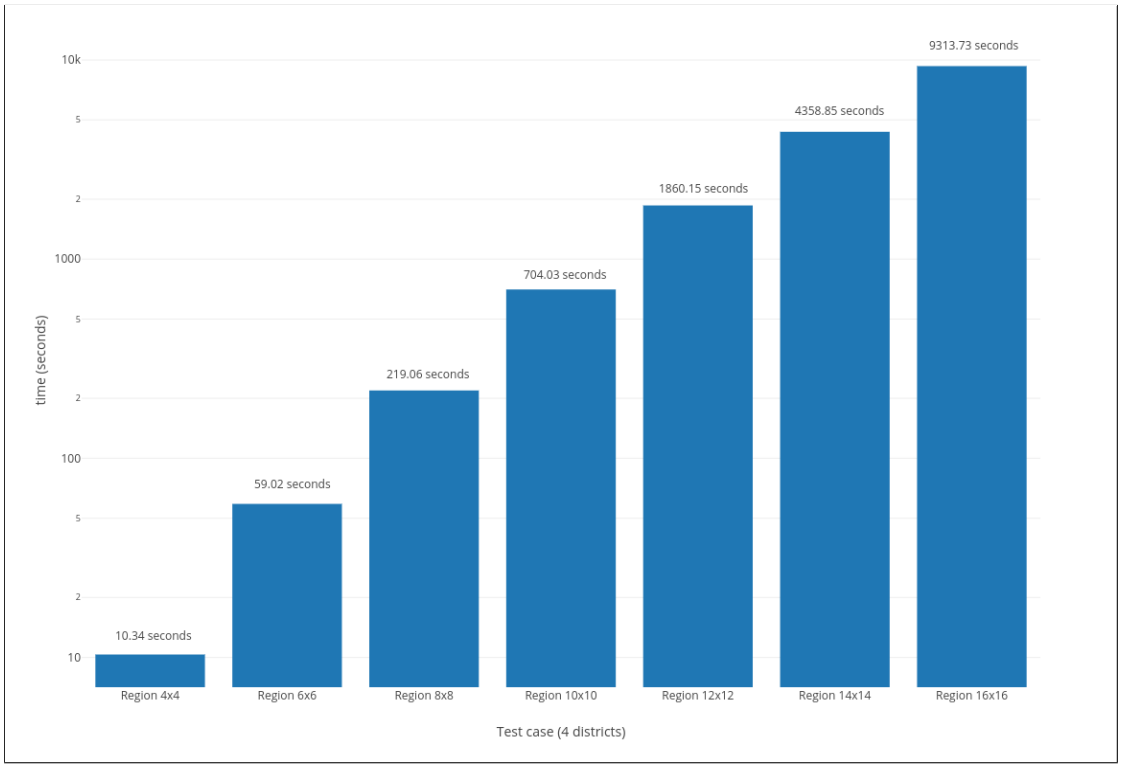

Figure 3. The running time of sequential SA for Test Suite 1 (y-axis has a logarithmic scale)

Figure 3 shows, expectedly, that test case 4x4 took the shortest time to run and Region 16x16 the longest time. The sequential simulated annealing algorithm

completed 5000 iterations for Region 4x4 in 10 seconds and for Region 16x16 in 9313.73 seconds or about 155 minutes.

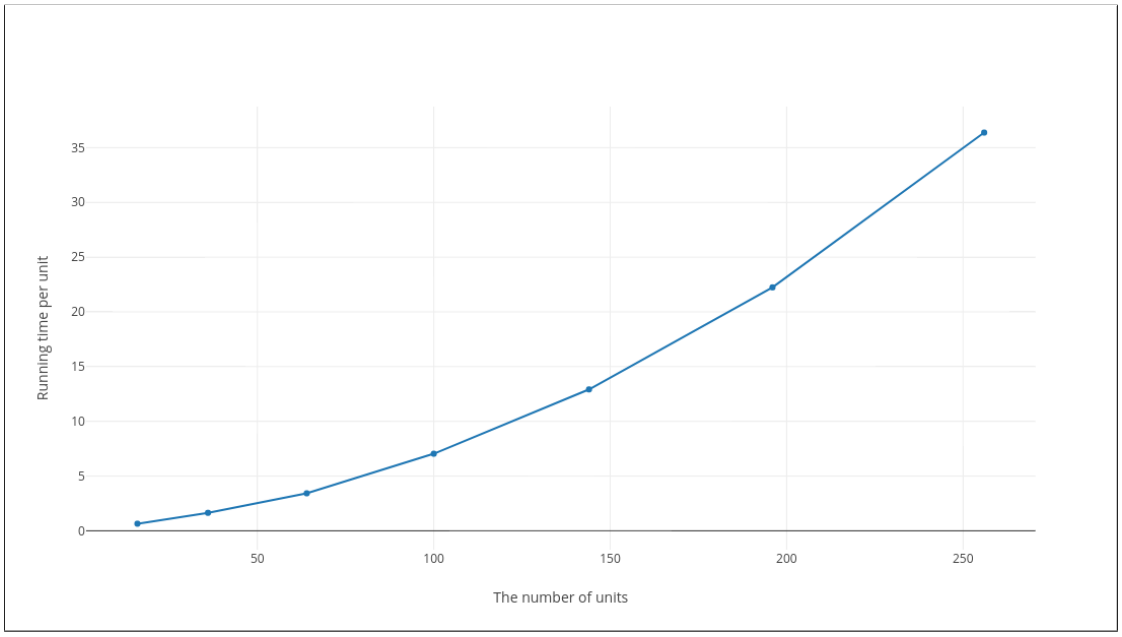

Figure 4. Running time per unit against units, Sequential SA, Test Suites 1

Figure 4 demonstrates that the running time per unit of the sequential version of simulated annealing increases, as the number of units in a test case of Test Suite 1 gets larger. It is clear that the growth rate is not linear, but superlinear.

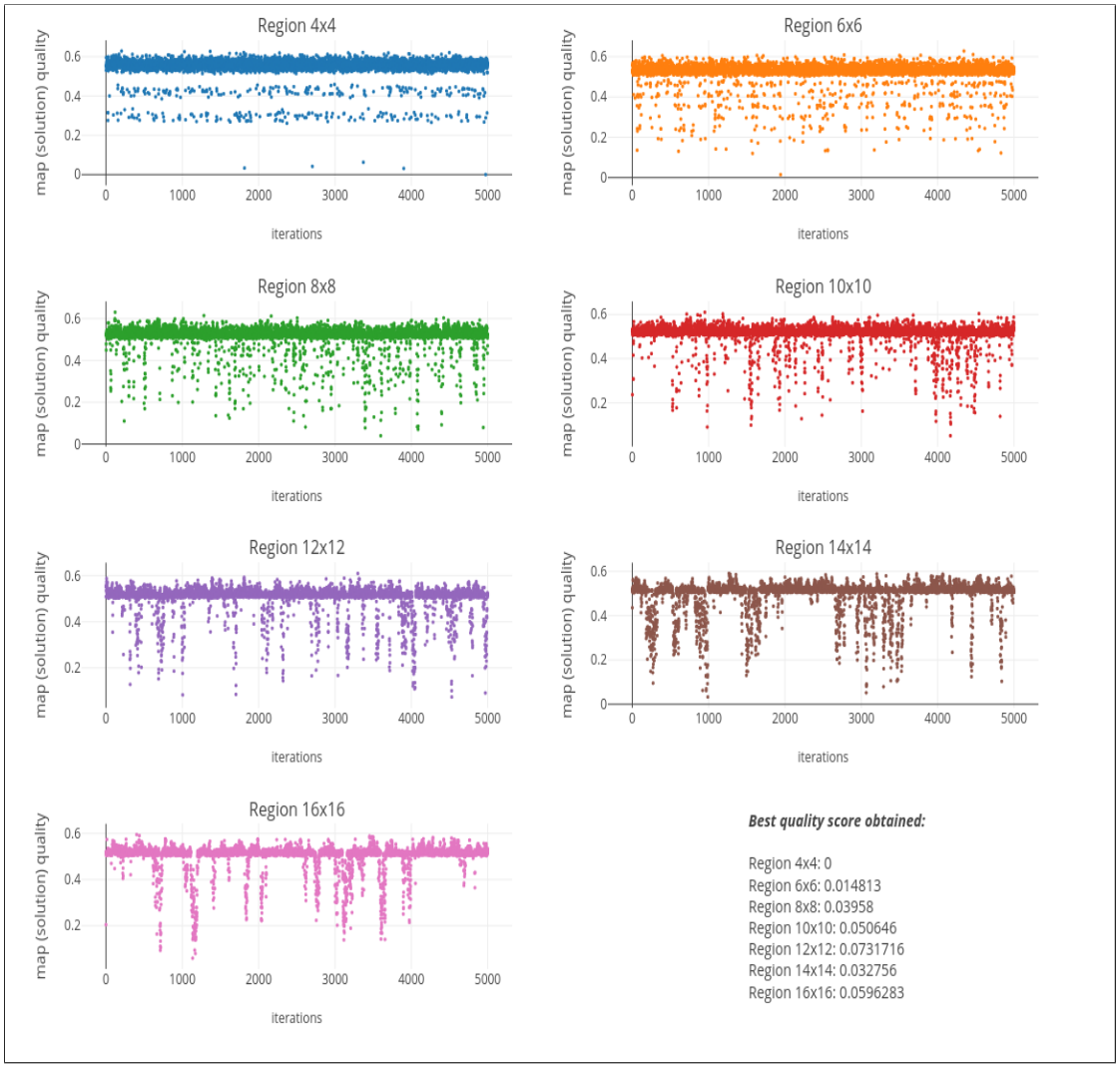

Figure 5. The quality of a generated neighbor per iteration (Sequential SA, Test Suite 1)

Figure 5 shows that the algorithm managed to find near-optimal solutions for all test-cases. The algorithm managed to find the optimal solution for Region 4x4. However, the Region 4x4 subplot also shows that the algorithm produced solutions which values converge to three subsequences. It is probably because the test case is relatively small containing only 16 units, and, as a result, there is not a lot of variation.

Other subplots indicate that show that the algorithm produced solutions some of which converge to a fitness score of 0.5. Also, these subplots show patterns

of steep drops leading to substantial improvements in generated neighbors and resulting in reaching low points where near-optimal solutions were found. Those patterns clearly show that the algorithm tries its to improve iteratively using the previous solution. Appendix E shows the best solution that the sequential version of simulated annealing managed to generate for Region 16x16.

The sequential simulated annealing algorithm was run on Test Suite 2 with the same parameters as on Test Suite 1: with the same temperature and the same number of iterations.

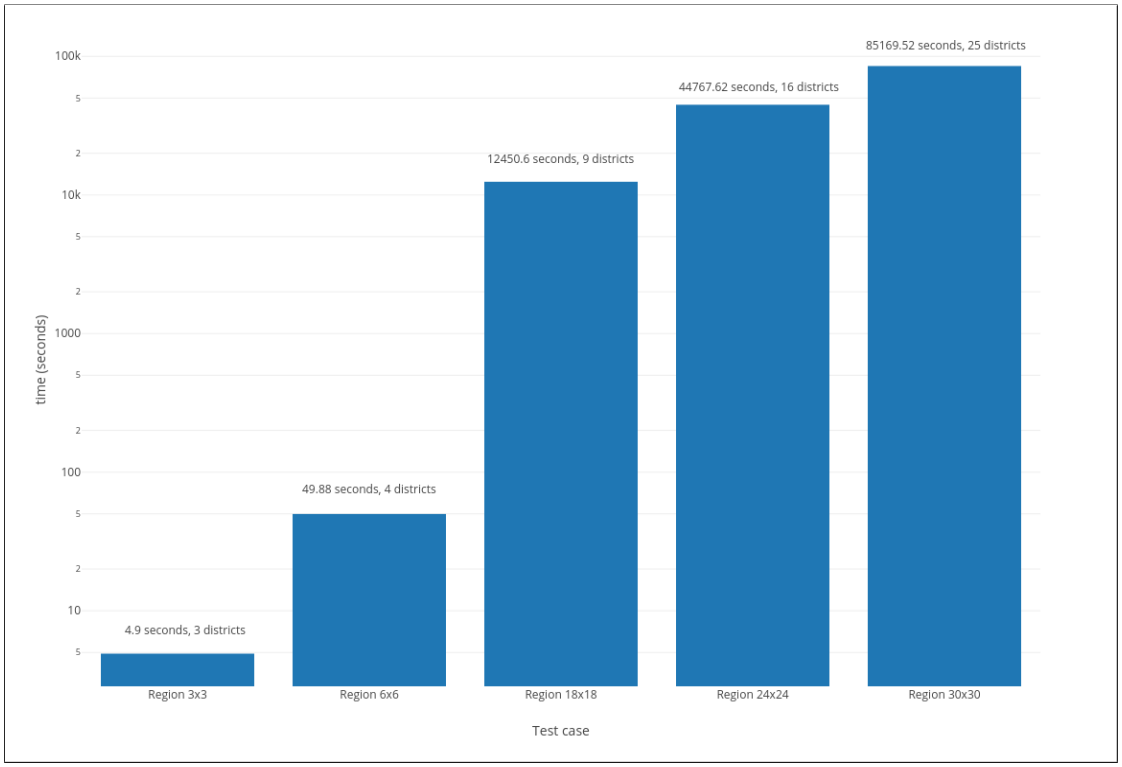

Figure 6. The running time of sequential SA for Test Suite 2 (y-axis has a logarithmic scale)

Figure 6 demonstrates anticipated results: the algorithm took the least amount of time, 5 seconds, to complete 5000 iterations for Region 3x3 and the largest amount of time, 85169 seconds (which is about 23 hours and 39 minutes) for Region 30x30.

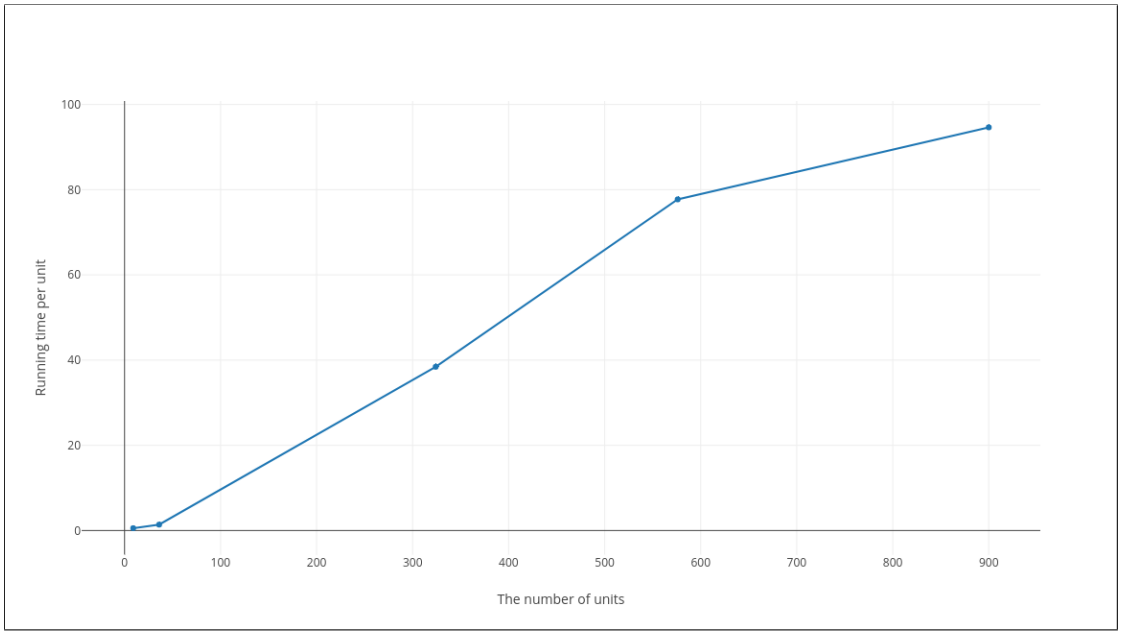

Figure 7. Running time per unit against units, Sequential SA, Test Suites 2

Figure 7 shows the running time per unit of the sequential version of simulated annealing on Test Suite 2. The complexity is not very clear (but appears to be linear or superlinear) from the plot probably because all test cases in the test suite have a different number of districts and are not as standardized as test cases in Test Suite 1.

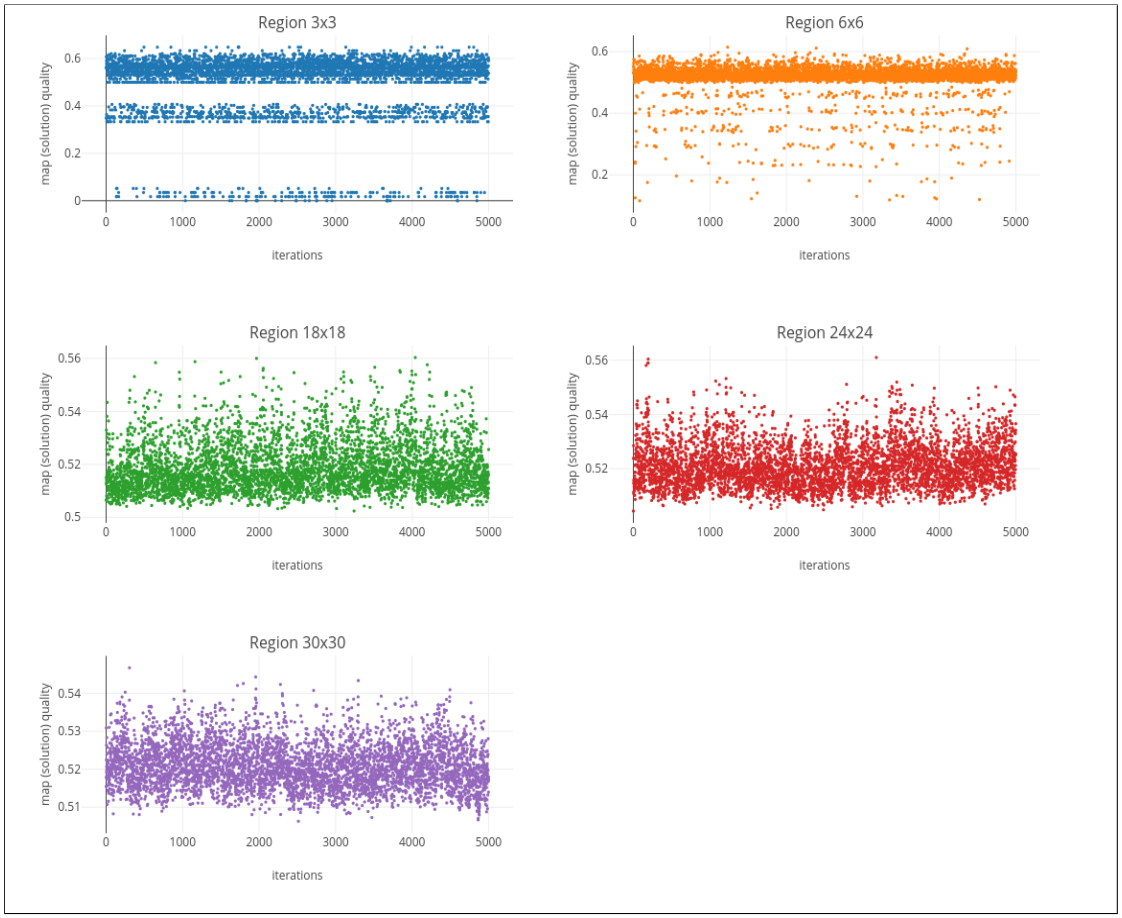

Figure 8. The quality of a generated neighbor per iteration (Sequential SA, Test Suite 2)

Figure 8 shows trends that are different from Figure 5. The only test cases where the algorithm could achieve optimal or near-optimal solutions are Region 3x3 and Region 6x6. These two test cases are similar to test cases in Test Suite 1 because they also have a small number of districts.

Subplots for the other test cases (Region 18x18, Region 24x24 and Region 30x30) show that the algorithm was not able to approach anything that could resemble a near-optimal solution at all. For these test cases algorithm managed to produce solutions within the 0.50–0.56 range. Appendix F shows the best solution that the sequential version of simulated annealing managed to generate for Region 30x30.

4.2 Parallel Simulated Annealing

The parallel version of simulated annealing was also run on Test Suite 1 and Test Suite 2 with the exception that it was not run on Region 3x3 and Region 6x6 in Test Suite 2. It was run with the following parameters: max temperature 10000, 60 instances (population size) and 1000 iterations for each instance. The algorithm utilized one core per instance, which means that running it on one test case utilized 60 cores in total.

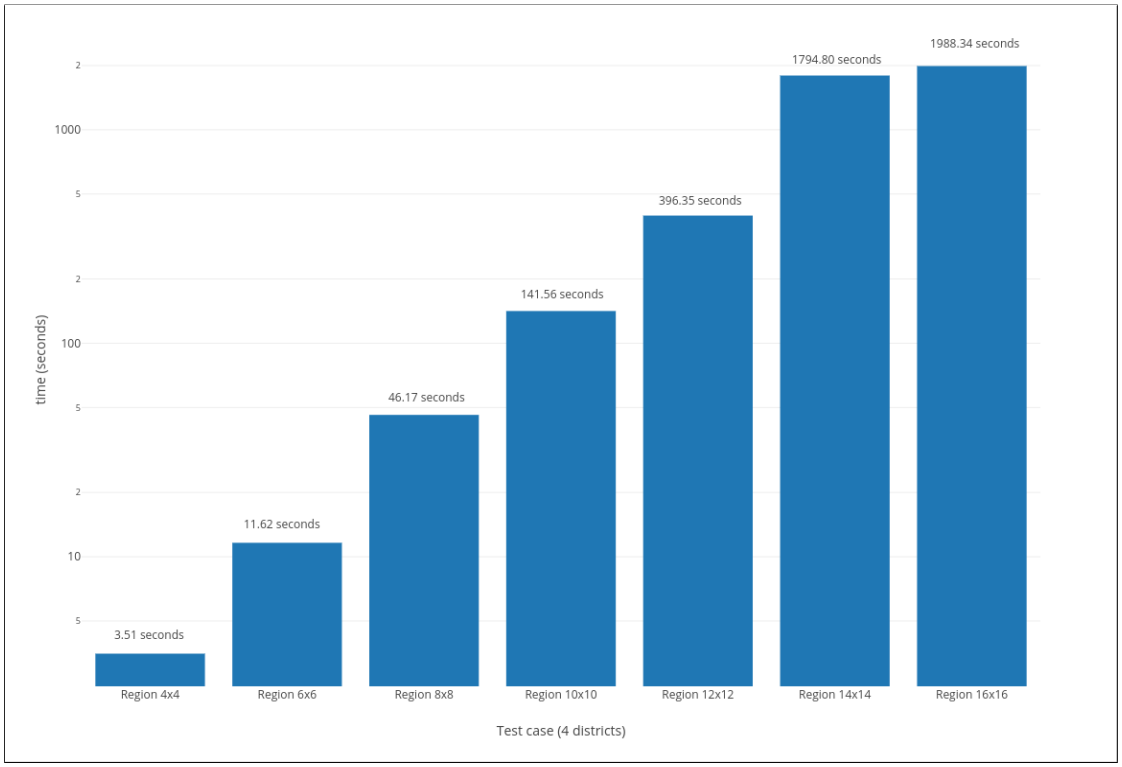

Figure 9. The running time of parallel SA for Test Suite 1 (y-axis has a logarithmic scale)

Figure 9, the chart with the running time for Test Suite 1, shows, expectedly, that Region 4x4 took the shortest time to run and Region 16x16 the longest time. The parallel Simulated Annealing algorithm completed 1000 iterations for Region 4x4 in 3.51 seconds. It took 1988.34 seconds or about 33 minutes to finish working Region 16x16. 1988.34 is about 5 times less than the amount of time it took for the sequential version to complete 5000 iterations, which, in some sense, expected because the parallel version utilizes the sequential version underneath. The same trend applies to all other test cases, except Region 14x14. It took the algorithm

1794.80 seconds to complete 1000 iterations for each instance. Almost the same time as Region 16x16. The reason was that is because the algorithm experienced difficulties with generating neighbors for 2 instances. The other 58 instances managed to complete their work in 973.54 seconds.

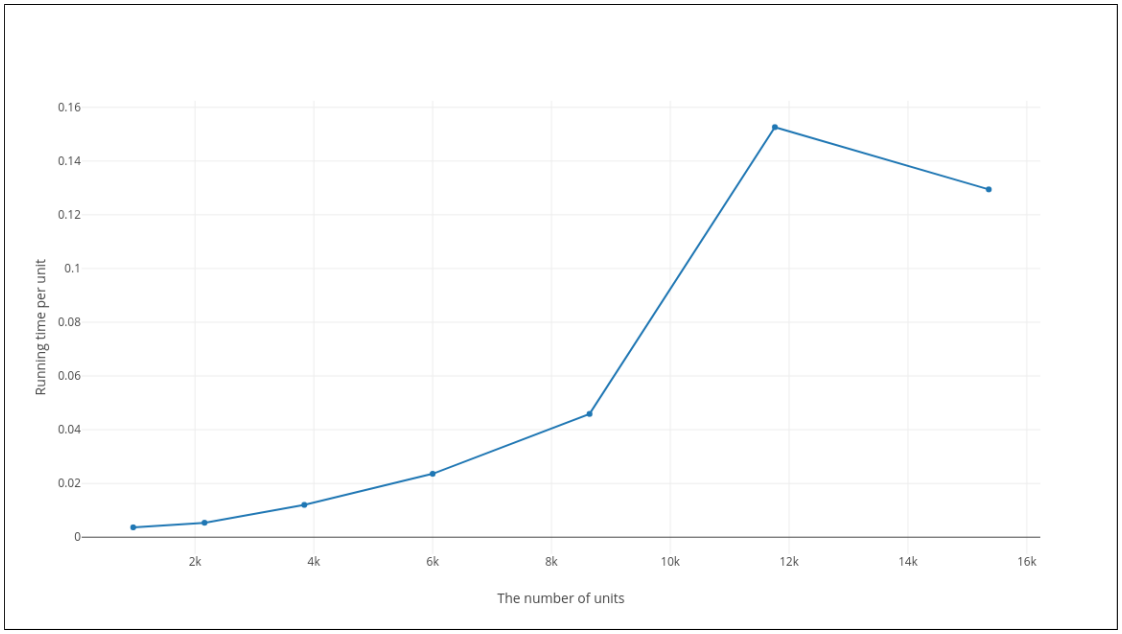

Figure 10. Running time per unit against units, Parallel SA, Test Suites 1.

Figure 10 shows that the running time per unit for the parallel version of simulated annealing on Test Suite 1 is superlinear. If outliers in Region 14x14 are excluded, then this plot resembles the plot in Figure 4. The number of units is calculated as

$$numberOfInstances \times numberOfUnitsInTesCase$$

The running time per unit is calculated as

$$\frac{totalRunningTimeOfAllInstances}{numberOfInstances \times numberOfUnitsInTesCase}$$

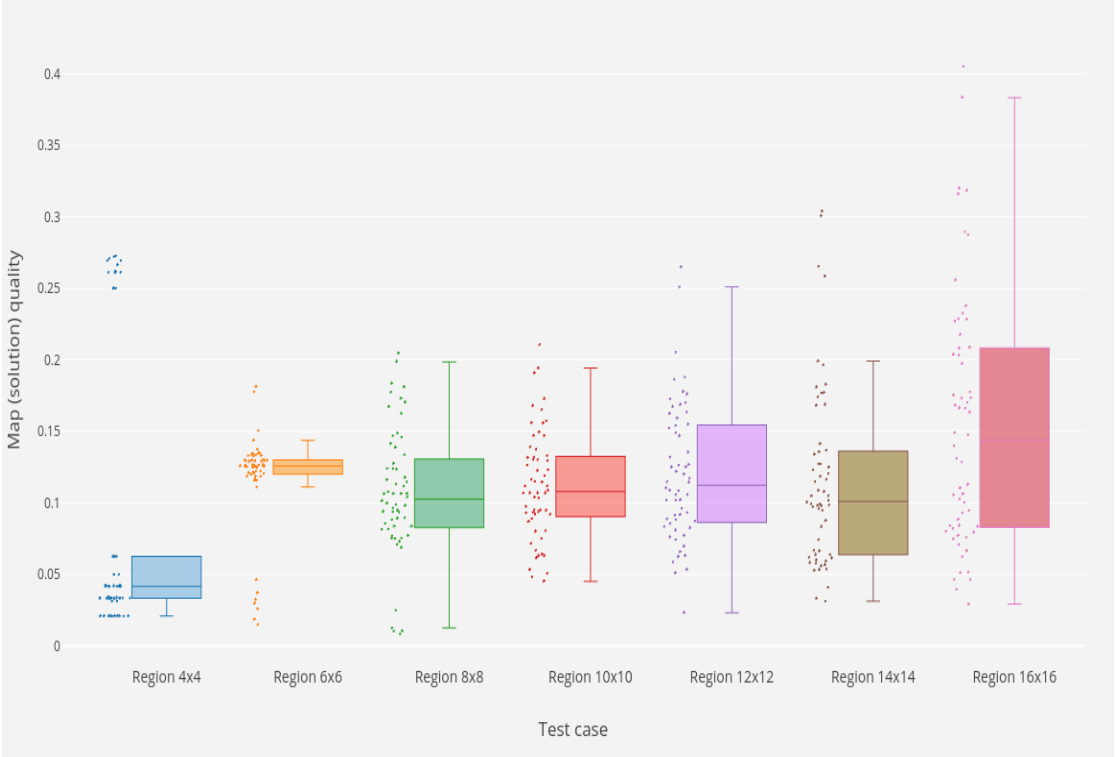

Figure 11. The quality of a generated neighbor per iteration (Parallel SA, Test Suite 1)

Figure 11 shows that the parallel version of SA, similar to sequential SA, managed to find near-optimal solutions. For all test cases the parallel version managed to find near-optimal solutions that are within the 0.00–0.05 range.

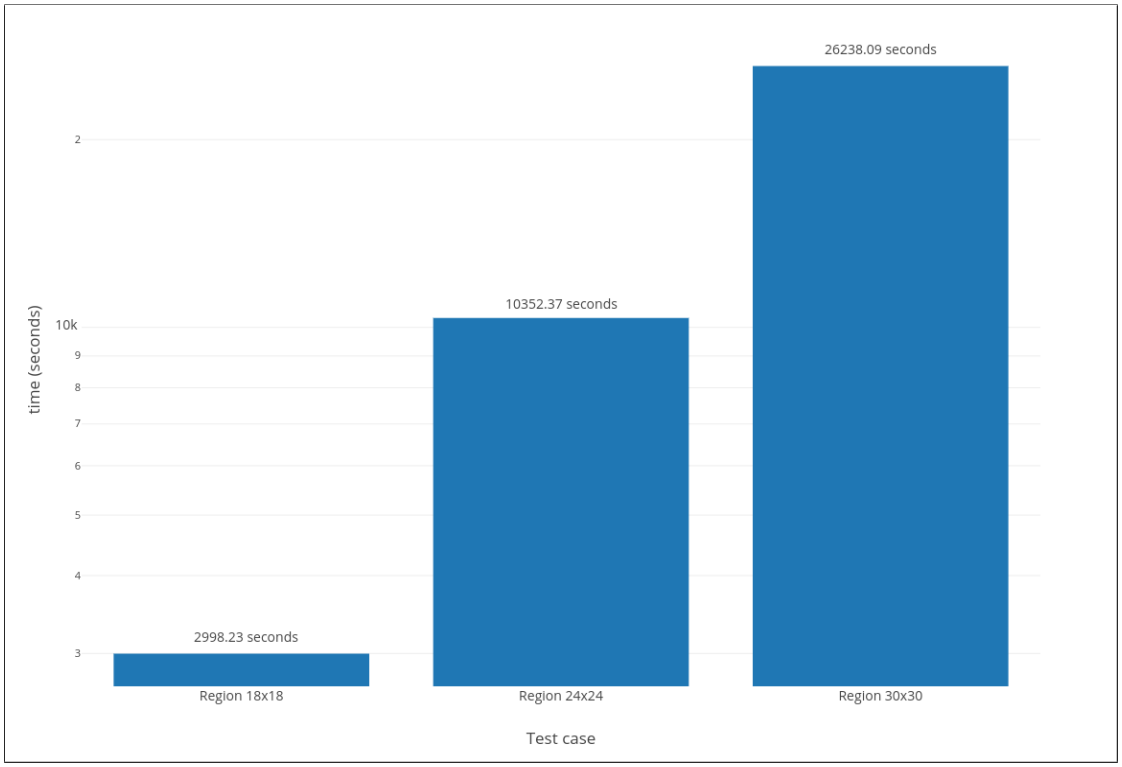

Figure 12. The running time of parallel SA for Test Suite 2 (y-axis has a logarithmic scale)

Figure 12 shows running times of the parallel version of SA for three test cases on Test Suite 2: Region 18x18, Region 24x24, Region 30x30. As anticipated, the running time for Region 18x18 is the lowest and for Region 30x30 it is the highest. It took the algorithm 5 times less time to complete working on Region 18x18 and Region 24x24 than the sequential SA algorithm. It took 5 times more time to finish 1000 iterations for all 60 instances for Region 30x30 compared to the sequential version.

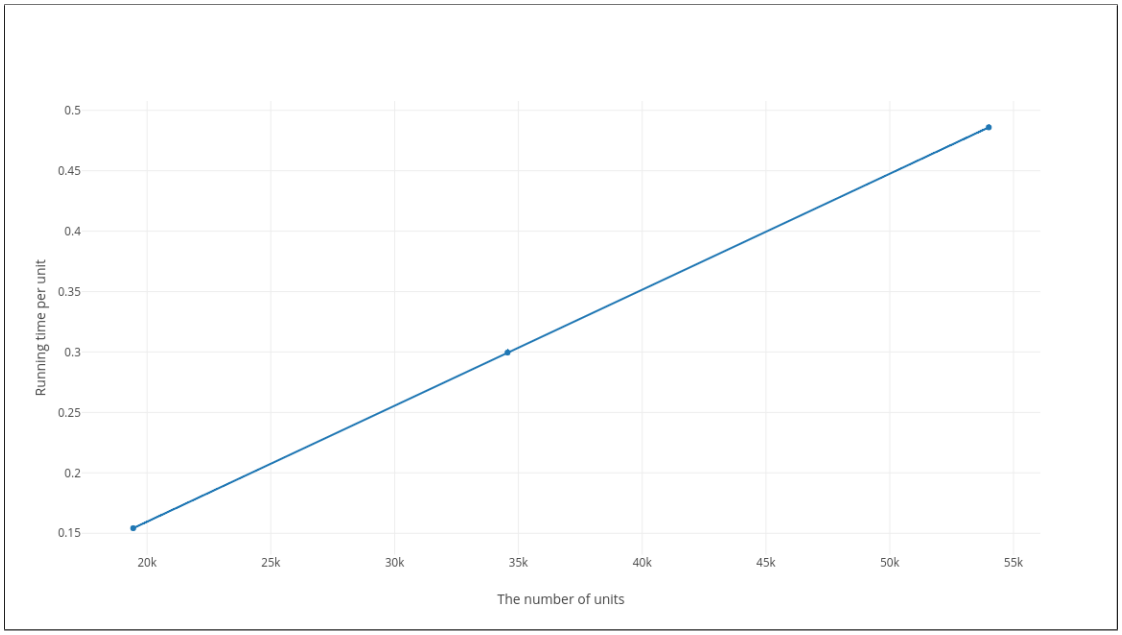

Figure 13. Running time per unit against units, Parallel SA, Test Suites 2

Figure 13 shows the running time per unit of the parallel version of simulated annealing on Test Suite 2. The growth rate seems to be linear, however, it might appear so because the parallel version was run only 3 test cases.

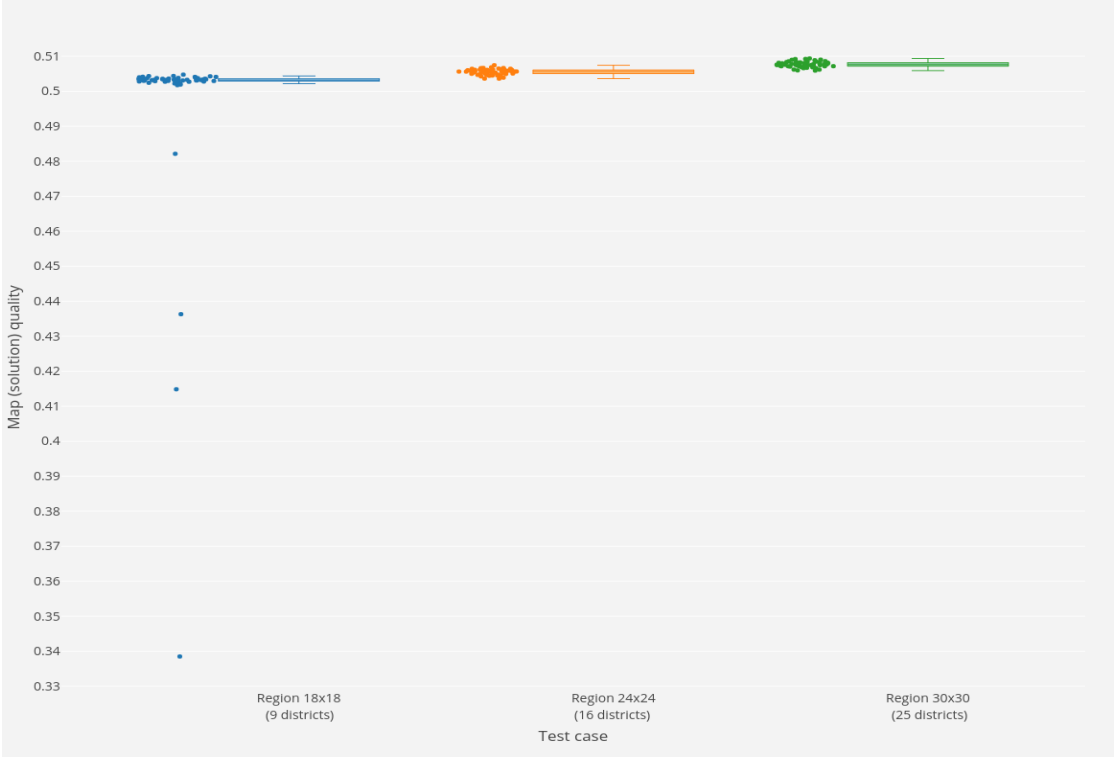

Figure 14. The quality of a generated neighbor per iteration (Parallel SA, Test Suite 2)

Figure 14 shows the parallel version did not manage to produce substantially better solutions for test cases Region 18x18, Region 24x24 and Region 30x30 than the sequential version. For Region 18x18 the algorithm managed to achieve a score of 0.34. For all other cases solutions are within the 0.50–0.51 range.

Appendix G shows the initials solutions generated by Algorithm 11 for Region 18x18. The solutions shown in the charts exhibit one clear pattern which is a characteristic of the algorithm. The pattern is that there are no districts that are not on the region border in any of the generated initial solutions. It is one of the reasons why simulated annealing fails to generate high-quality solutions for large test cases with a large number of districts. The reason for that is because it might be difficult for Algorithm 12 to generate neighbor solutions that would be similar to the optimal solutions shown in Appendix D.

5 Discussion

5.1 Implications of the results

The results show that the simulated annealing algorithms performed well against Test Suite 1. It means that the algorithms could be used for redistricting of states such as the state of Iowa, which has 4 congressional districts, at the county level. The conclusion was already verified by Myung Jin Kim in her doctoral dissertation [Kim11]. Her version of simulated annealing was tested by redistricting the state of Iowa at the county level, and it managed to achieve near-optimal results.

However, Figure 8 and Figure 14 demonstrate that the current versions of simulated annealing cannot be used for redistricting regions that have a large number of districts. The result is interesting because the parallel version of simulated annealing explored a larger fraction of the solution space. The result shows the importance of algorithms that generate neighboring solutions and initial solutions. In addition, it explains why the PEAR algorithm [LCW16] has such a sophisticated crossover operator because without such an algorithm the search would not get stuck in local optima that cannot be escaped.

Figure 14 shows that the parallel version of simulated annealing managed to find better solutions than the sequential version of simulated annealing for Region 18x18. Those solutions are not optimal or near-optimal, but they are significantly better than those found by the sequential version. It shows the potential of working on improving multiple solutions at the same time for a lesser period of time than working on improving only one solution for a longer period of time.

5.2 Future work

Simulated annealing is implemented in Python. Therefore, the performance is not perfect, and running time results cannot be considered as objective benchmarks. If simulated annealing is implemented in another language such as C or Fortran, significant decreases in running time can be expected. That would allow to efficiently test simulated annealing on larger test cases because right now it would take a prohibitively large amount of time.

Test suites can also be improved. First, larger cases such as Region 300x300 could be added to Test Suite 1. The reason for that is because the current test cases of Test Suite 1 do not tell if simulated annealing is a good algorithm if the number of districts remains small, but the number of units is at the census block level. Also, test cases are designed such that optimal solutions exhibit a certain pattern.

To improve objectivity, test cases could be designed such that optimal solutions exhibit more randomness. In addition, all units in all test cases have the same population size, which is far from reality. To increase objectivity, units should have different population sizes.

In addition, 8 Figure 14 shows that the current algorithm that generates neighbor solutions fails to generate good solutions if the number of units and districts are at least 576 and 16 respectively. Therefore, it would be worth investigating how the algorithm could be improved or what other algorithms could be used instead of this one.

Summary

In this thesis, the goal was to implement one metaheuristic and analyze it with redistricting in mind. Two test suites were created to conduct the analysis. The implemented metaheuristic was simulated annealing of which two versions were created: parallel and sequential. It was discovered that simulated annealing can probably be used be used for redistricting of a region with a small number of districts such as the state of Iowa, which has 4 congressional districts. At the same time, the implemented versions of simulated annealing are ineffective for redistricting of regions with a larger number of districts such as the state of New York, which has 27 congressional districts.

The author of this thesis was responsible for designing and creating test suites, implementing two versions of simulated annealing, running the algorithms, visualizing and analyzing the data. The research questions as well as some ideas regarding the design of the test suites and the implementations of simulated annealing were suggested by the supervisor.

References

[AM10] Micah Altman and Michael P. McDonald. "The Promise and Perils of Computers in Redistricting". In: *Duke J Const Law Pub Poly* 5 (2010), 69–159. URL: http://www.law.duke.edu/journals/djclpp/?action=downloadarticle%5C&id=167.

[AM11a] Micah Altman and Michael P. McDonald. "BARD: Better Automated Redistricting". In: *Journal of Statistical Software* 42.4 (2011), pp. 1–28. URL: http://www.jstatsoft.org/v42/i04.

[AM11b] Micah Altman and Michael P. McDonald. *GitHub - cran/BARD: Better Automated ReDistricting*. https://github.com/cran/BARD. (Accessed on 05/13/2018). Apr. 2011.

[BR03] Christian Blum and Andrea Roli. "Metaheuristics in combinatorial optimization: Overview and conceptual comparison". In: *ACM computing surveys (CSUR)* 35.3 (2003), pp. 268–308.

[Bur] U.S. Census Bureau. *2010 Census Tallies of Census Tracts, Block Groups & Blocks - Geography - U.S. Census Bureau*. https://www.census.gov/geo/maps-data/data/tallies/tractblock.html. (Accessed on 04/29/2018).

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[CJ14] Karl-Dieter Crisman and Michael A. Jones. *The Mathematics of Decisions, Elections, and Games:* Contemporary Mathematics. American Mathematical Society, 2014. ISBN: 9780821898666. URL: https://books.google.ee/books?id=xdZzBAAAQBAJ.

[ES03] Agoston E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2003. ISBN: 3540401849.

[FR89] Thomas A. Feo and Mauricio G. C. Resende. "A Probabilistic Heuristic for a Computationally Difficult Set Covering Problem". In: *Oper. Res. Lett.* 8.2 (Apr. 1989), pp. 67–71. ISSN: 0167-6377. DOI: 10.1016/0167-6377(89)90002-3. URL: http://dx.doi.org/10.1016/0167-6377(89)90002-3.

[Glo86] Fred Glover. "Future Paths for Integer Programming and Links to Artificial Intelligence". In: *Comput. Oper. Res.* 13.5 (May 1986), pp. 533–549. ISSN: 0305-0548. DOI: 10.1016/0305-0548(86)90048-1. URL: http://dx.doi.org/10.1016/0305-0548(86)90048-1.

[GS15] Fred Glover and Kenneth Sörensen. “Metaheuristics”. In: *Scholarpedia* 10.4 (2015). revision #149834, p. 6532. DOI: 10.4249/scholarpedia.6532.

[GW17] Sebastian Goderbauer and Jeff Winandy. *Political Districting Problem: Literature Review and Discussion with regard to Federal Elections in Germany*. Tech. rep. 2017–042. Operations Research, RWTH Aachen University, Nov. 2017. URL: http://www.or.rwth-aachen.de/research/publications/LitSurvey_PoliticalDistricting__Goderbauer_Winandy_20171123.pdf.

[Hig] University of Tartu High Performance Computing Center. *Rocket Cluster - High Performance Computing Center, University of Tartu*. https://hpc.ut.ee/en_US/web/guest/rocket-cluster. (Accessed on 05/03/2018).

[KGV83] Scott Kirkpatrick, Daniel C. Gelatt, and Mario P. Vecchi. “Optimization by Simulated Annealing”. In: *Science*. New Series 220.4598 (1983), pp. 671–680. ISSN: 00368075. DOI: 10.1126/science.220.4598.671. URL: http://www.jstor.org/stable/1690046.

[Kim11] Myung Jin Kim. “Optimization Approaches to Political Redistricting Problems”. PhD thesis. The Ohio State University, 2011.

[Kin+12] Douglas M. King et al. “Geo-Graphs: An Efficient Model for Enforcing Contiguity and Hole Constraints in Planar Graph Partitioning”. In: *Operations Research* 60.5 (2012), pp. 1213–1228. DOI: 10.1287/opre.1120.1083. eprint: https://doi.org/10.1287/opre.1120.1083. URL: https://doi.org/10.1287/opre.1120.1083.

[Kot08] Jacques Kotze. “Introduction to Monte Carlo methods for an Ising Model of a Ferromagnet”. In: *ArXiv e-prints* (Mar. 2008). arXiv: 0803.0217 [cond-mat.stat-mech].

[LCW16] Yan Y. Liu, Wendy K.Tam Cho, and Shaowen Wang. “PEAR: a massively parallel evolutionary computation approach for political redistricting optimization and analysis”. English (US). In: *Swarm and Evolutionary Computation* 30 (Oct. 2016), pp. 78–92. ISSN: 2210-6502. DOI: 10.1016/j.swevo.2016.04.004.

[Liu17] Yan Liu. “High-performance evolutionary computation for scalable spatial optimization”. PhD thesis. University of Illinois at Urbana-Champaign, 2017.

[Liu18] Yan Y. Liu. *personal communication*. May 2018.

[Luk13] Sean Luke. *Essentials of Metaheuristics*. second. Lulu, 2013. URL: http://cs.gmu.edu/$%5Csim$sean/book/metaheuristics/.

[LW10] Justin Levitt and Erika Wood. *A Citizen's Guide to Redistricting*. Brennan Center for Justice at New York University School of Law, 2010. URL: https://books.google.ee/books?id=uPihnQEACAAJ.

[Mat18] NIST Digital Library of Mathematical Functions. *DLMF: 26.8 Set Partitions: Stirling Numbers*. https://dlmf.nist.gov/26.8#E42. (Accessed on 05/12/2018). Mar. 2018.

[Met+53] Nicholas Metropolis et al. "Equation of State Calculations by Fast Computing Machines". In: *The Journal of Chemical Physics* 21.6 (1953), pp. 1087–1092. DOI: 10.1063/1.1699114. URL: http://link.aip.org/link/?JCP/21/1087/1.

[New17] NBC News. *Partisan Gerrymandering Has Benefited the GOP, Analysis Shows*. https://www.nbcnews.com/news/us-news/partisan-gerrymandering-has-benefited-gop-analysis-shows-n776436. (Accessed on 04/29/2018). June 2017.

[NW06] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[Pol17] Politico. *Democrats' early money haul stuns GOP - POLITICO*. https://www.politico.com/story/2017/10/23/2018-fundraising-democrats-house-races-244044. (Accessed on 04/29/2018). Oct. 2017.

[Pos18] Washington Post. *Republicans are poised to dominate redistricting again. Can Eric Holder stop them? - The Washington Post*. https://www.washingtonpost.com/news/wonk/wp/2018/02/14/republicans-are-poised-to-dominate-redistricting-again-can-eric-holder-stop-them/?utm_term=.49a1f41448e4. (Accessed on 04/29/2018). Feb. 2018.

[Ral+04] Colin Rallings et al. "Redistricting Local Governments in England: Rules, Procedures, and Electoral Outcomes". In: *State Politics & Policy Quarterly* 4.4 (2004), pp. 470–490.

[RR03] Mauricio G. C. Resende and Celso C. Ribeiro. "Greedy Randomized Adaptive Search Procedures". In: *Handbook of Metaheuristics*. Ed. by Fred Glover and Gary A. Kochenberger. Boston, MA: Springer US, 2003, pp. 219–249. ISBN: 978-0-306-48056-0. DOI: 10.1007/0-306-48056-5_8. URL: https://doi.org/10.1007/0-306-48056-5_8.

[Sch13] SchedMD. *Slurm Workload Manager*. https://slurm.schedmd.com/overview.html. (Accessed on 05/03/2018). Mar. 2013.

[Sel13] W. Gardner Selby. *Republicans won more House seats than more popular Democrats, though not entirely because of how districts were drawn | PolitiFact Texas*. http://www.politifact.com/texas/statements/2013/nov/26/lloyd-doggett/democrats-outpolled-republicans-who-landed-33-seat/. (Accessed on 05/11/2018). Nov. 2013.

[Sta] National Conference of State Legislators. *Redistricting Criteria*. http://www.ncsl.org/research/redistricting/redistricting-criteria.aspx. (Accessed on 04/21/2018).

[Tim10] Los Angeles Times. *State legislative gains give Republicans unprecedented clout to remake districts - latimes*. http://articles.latimes.com/2010/nov/03/news/la-pn-state-legislatures-20101104. (Accessed on 04/29/2018). 2010.

[Yan11] Xin-She Yang. "Metaheuristic Optimization". In: *Scholarpedia* 6.8 (2011). revision #91488, p. 11472. DOI: 10.4249/scholarpedia.11472.

Appendices

Appendix A The political landscape, Test Suite 1

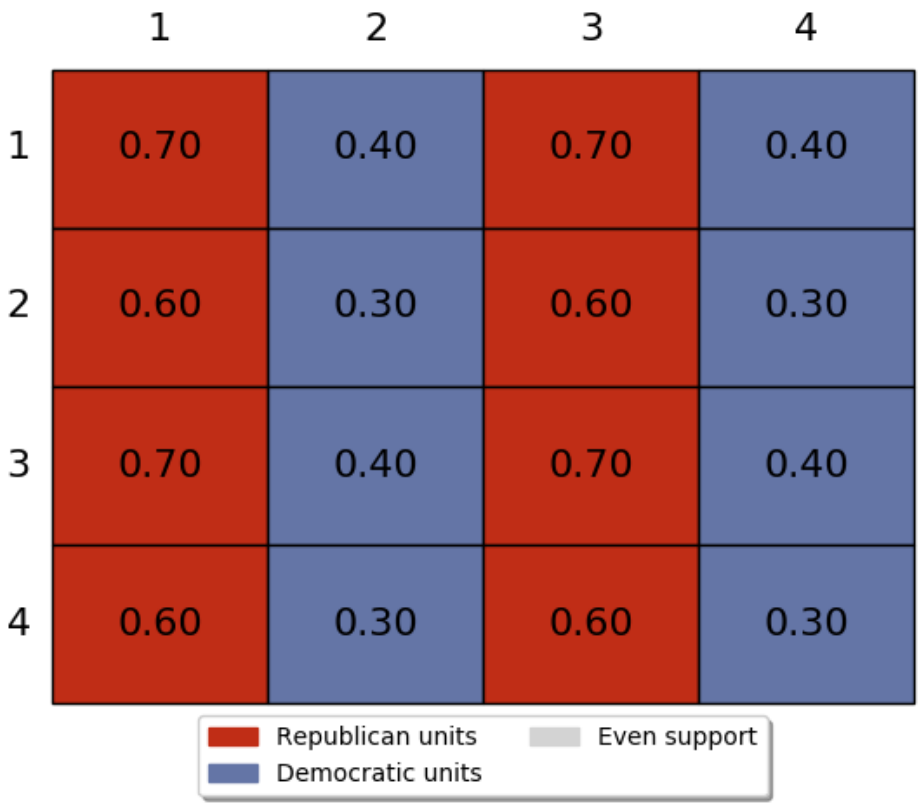

Figure 15. The political landscape, Test Suite 1, Region 4x4

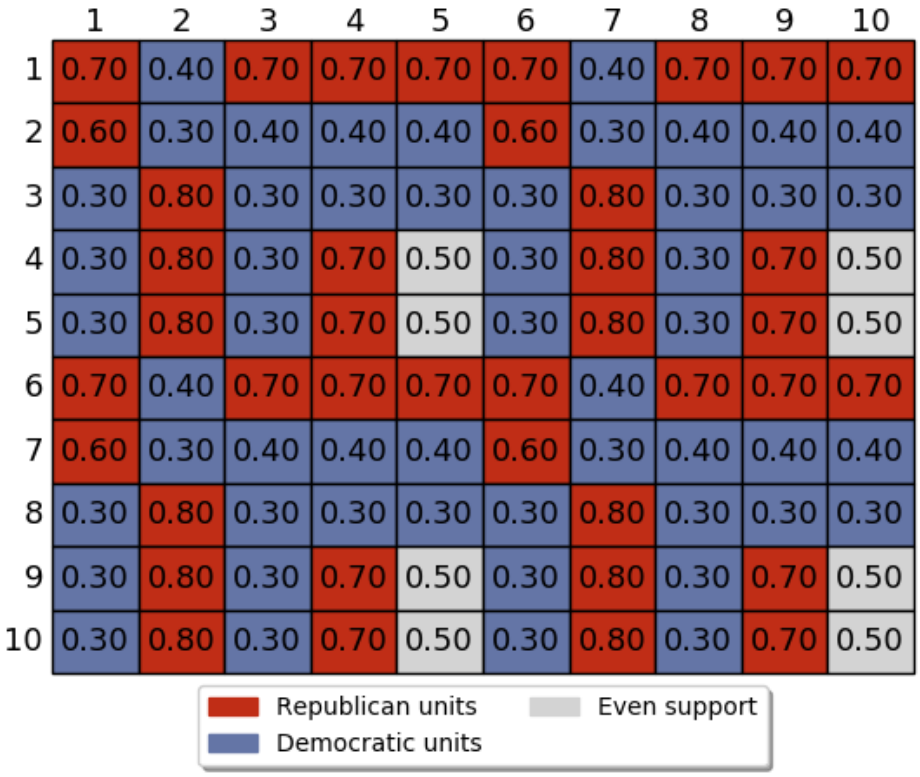

Figure 16. The political landscape, Test Suite 1, Region 10x10

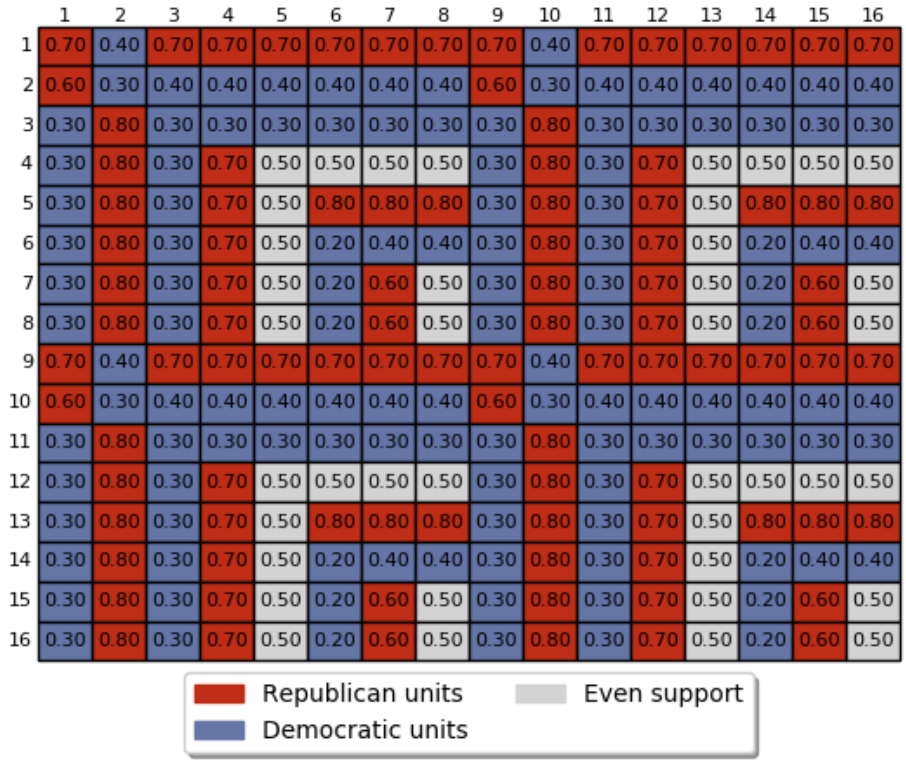

Figure 17. The political landscape, Test Suite 1, Region 16x16

Appendix B The political landscape, Test Suite 2

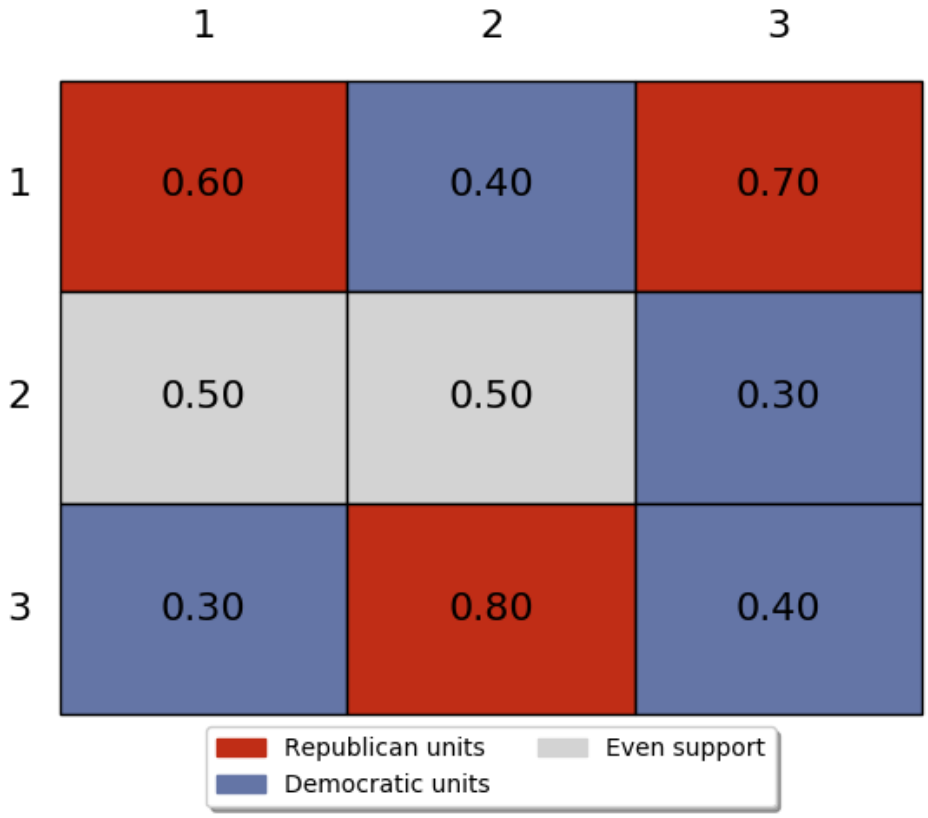

Figure 18. The political landscape, Test Suite 2, Region 3x3

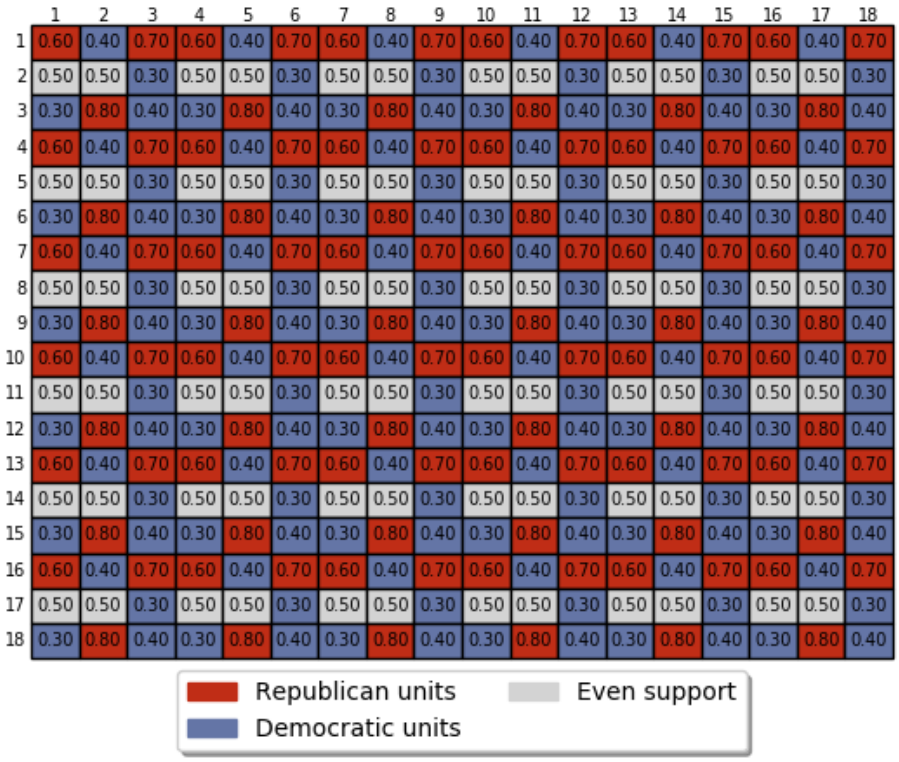

Figure 19. The political landscape, Test Suite 2, Region 18x18

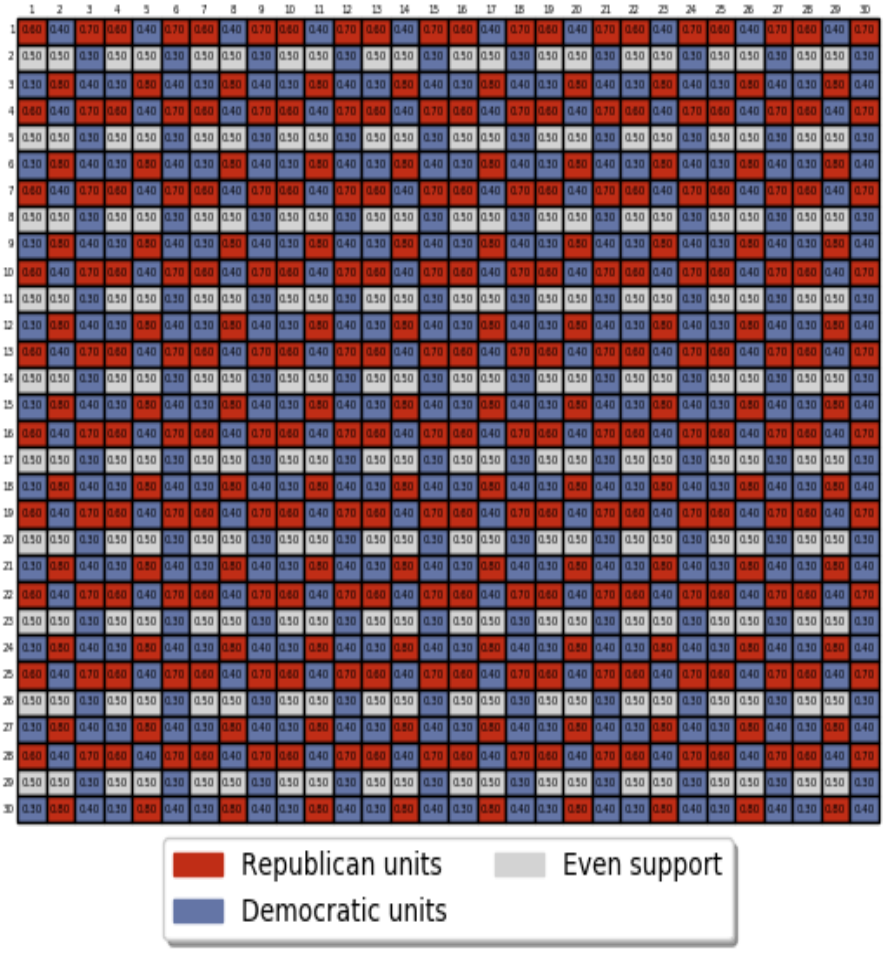

Figure 20. The political landscape, Test Suite 2, Region 30x30

Appendix C Optimal solutions, Test Suite 1

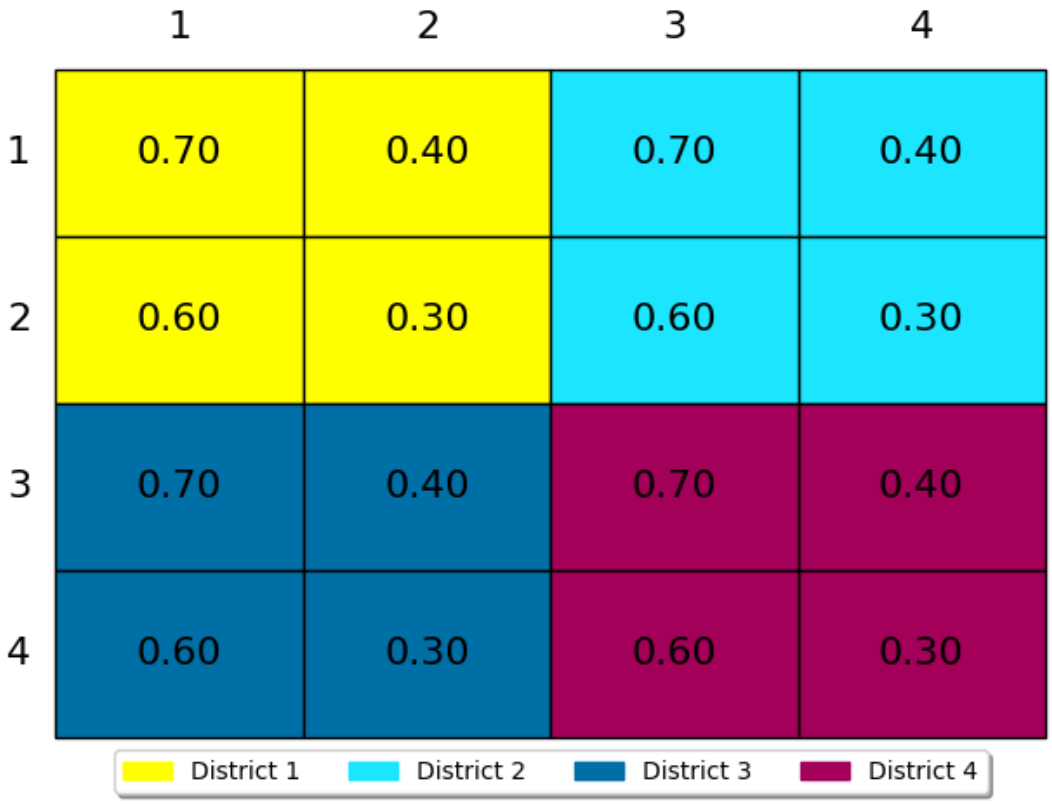

Figure 21. The optimal solution, Test Suite 1, Region 4x4

Figure 22. The optimal solution, Test Suite 1, Region 10x10

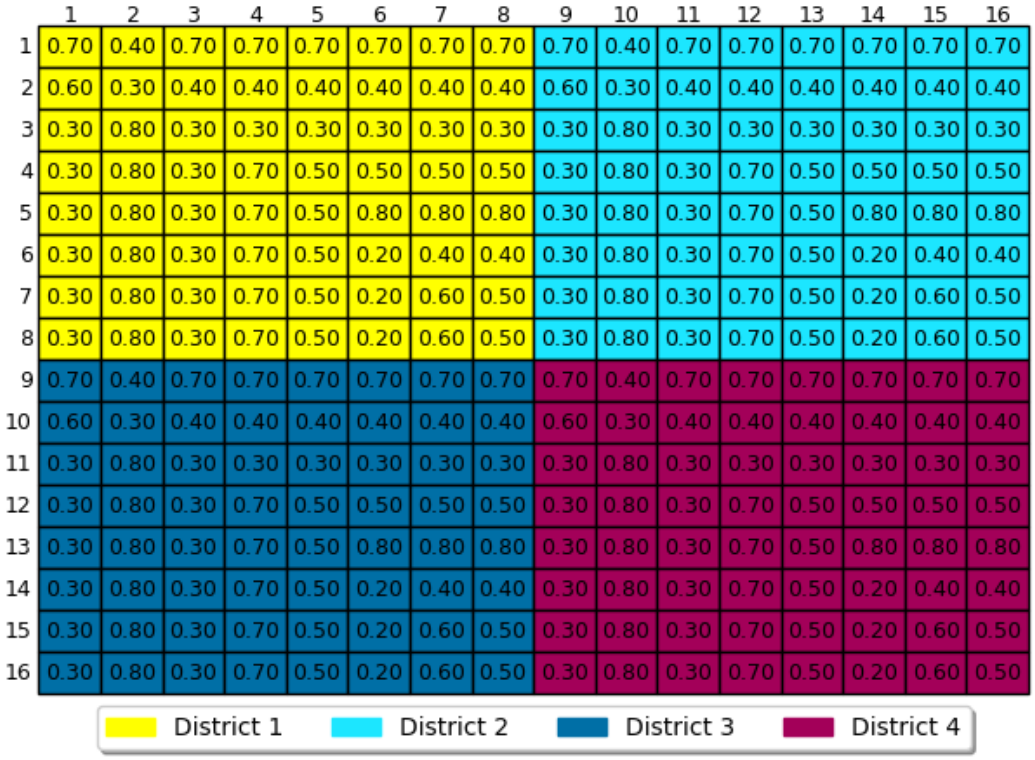

Figure 23. The optimal solution, Test Suite 1, Region 16x16

Appendix D Optimal solutions, Test Suite 2

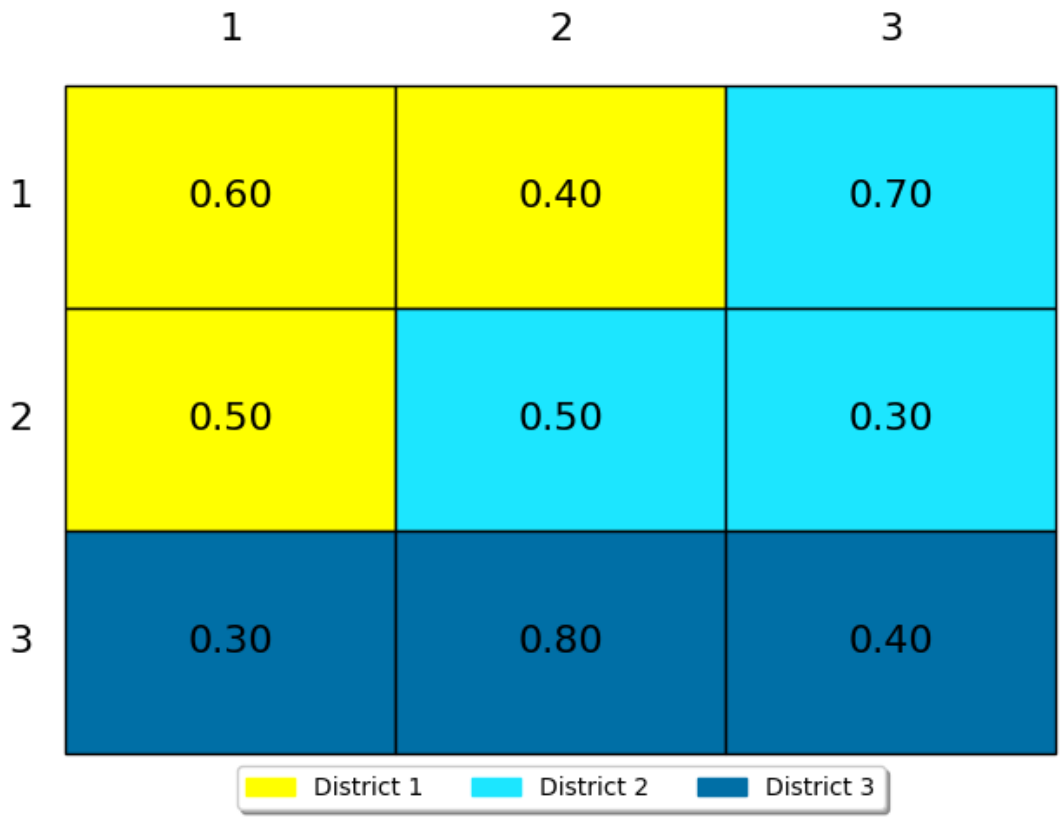

Figure 24. The optimal solution, Test Suite 2, Region 3x3

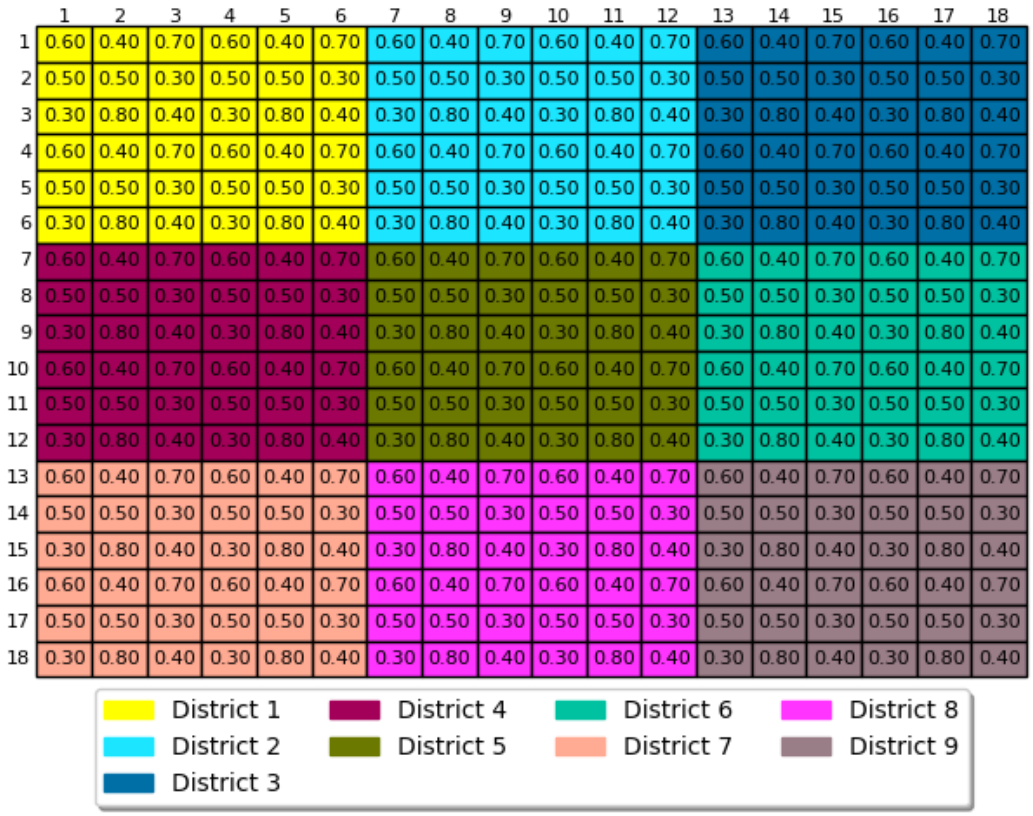

Figure 25. The optimal solution, Test Suite 2, Region 18x18

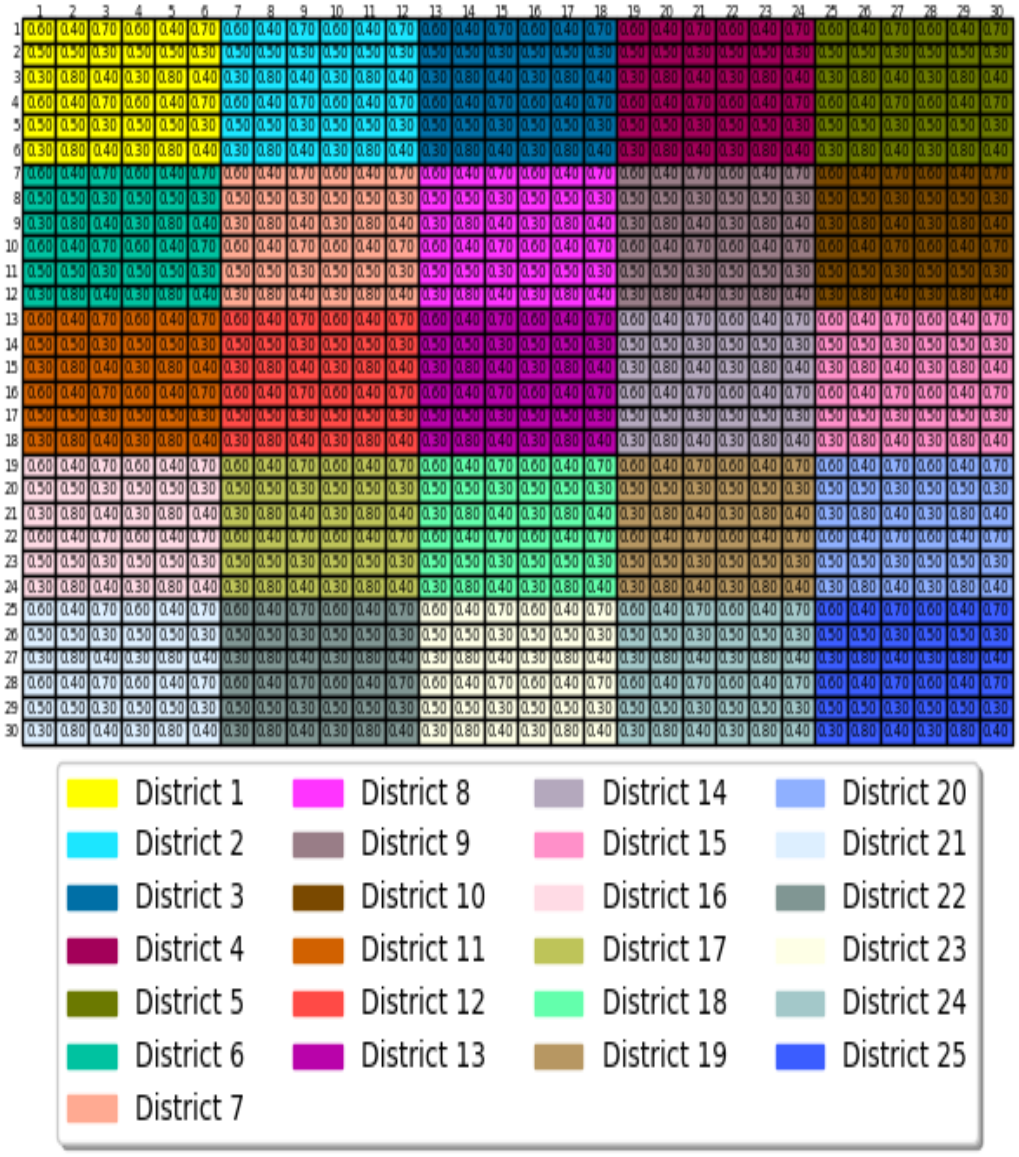

Figure 26. The optimal solution, Test Suite 2, Region 30x30

Appendix E The best found solution, Sequential SA, Test Suite 1, Region 16x16

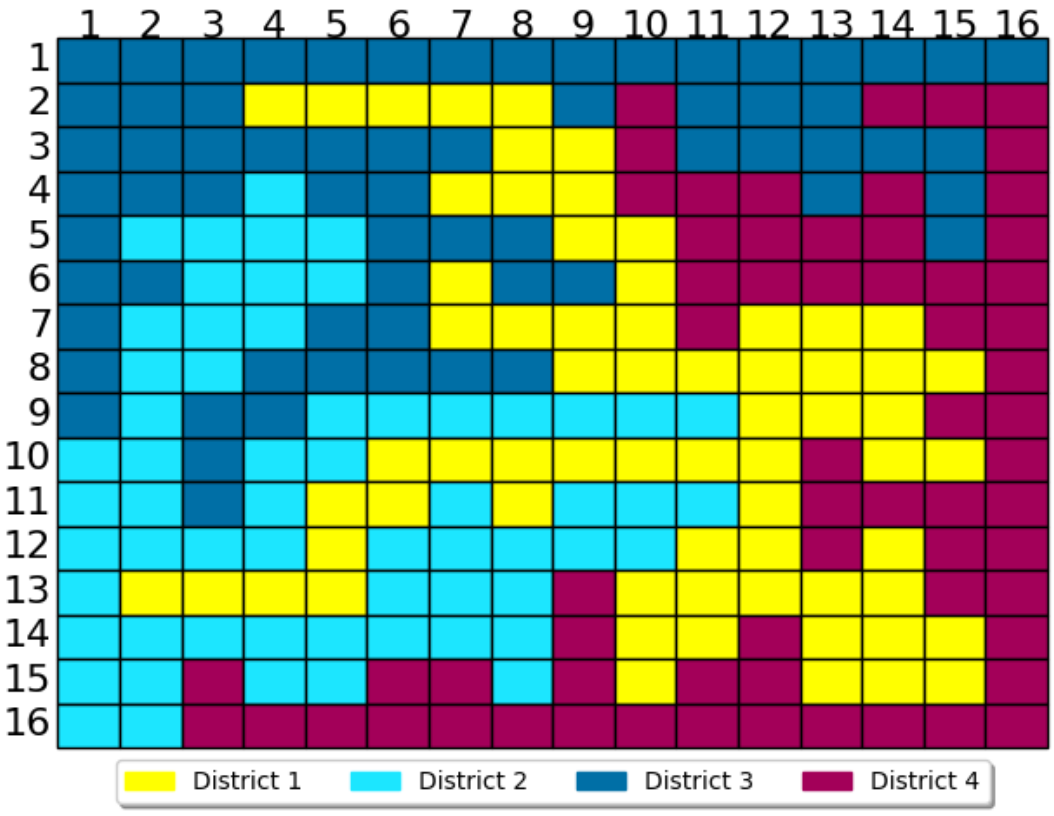

Figure 27. The best found solution, Sequential SA, Test Suite 1, Region 16x16

Appendix F The best found solution, Sequential SA, Test Suite 2, Region 30x30

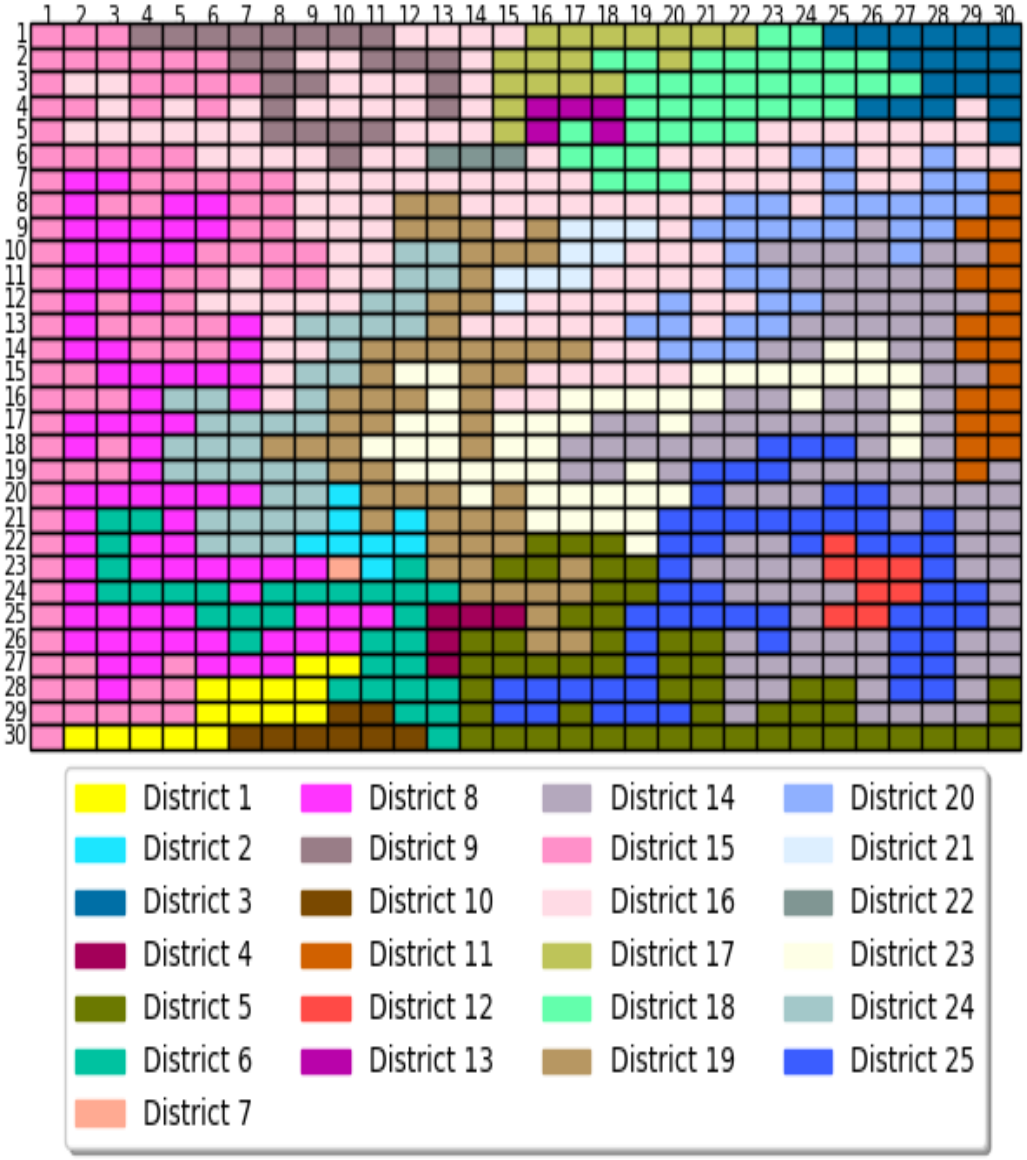

Figure 28. The best found solution, Sequential SA, Test Suite 2, Region 30x30

Appendix G Initial solutions, Parallel SA, Test Suite 2, Region 18x18, Parts 1-5

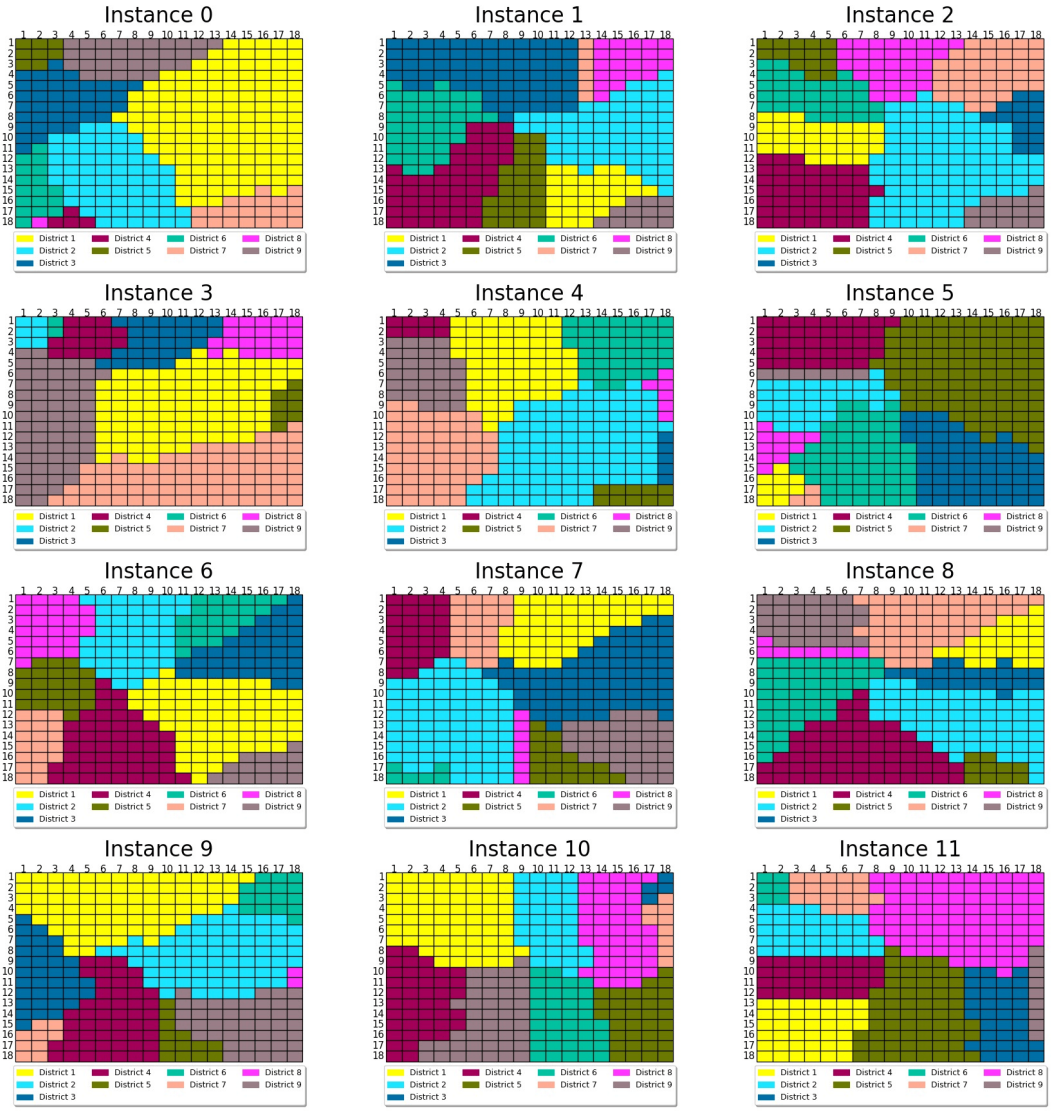

Figure 29. Parallel SA, Initial solutions, Test Suite 2, Region 18x18, Part 1

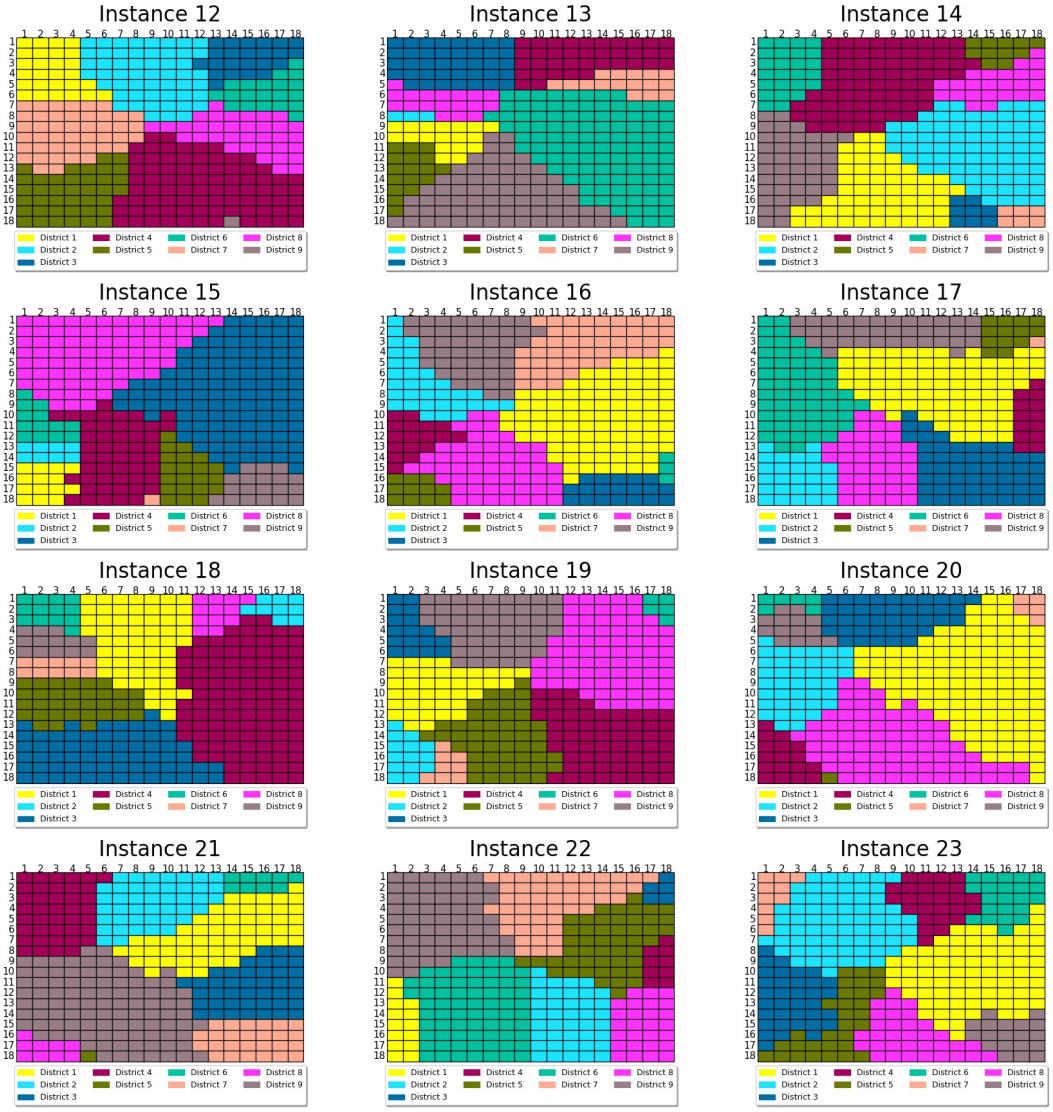

Figure 30. Parallel SA, Initial solutions, Test Suite 2, Region 18x18, Part 2

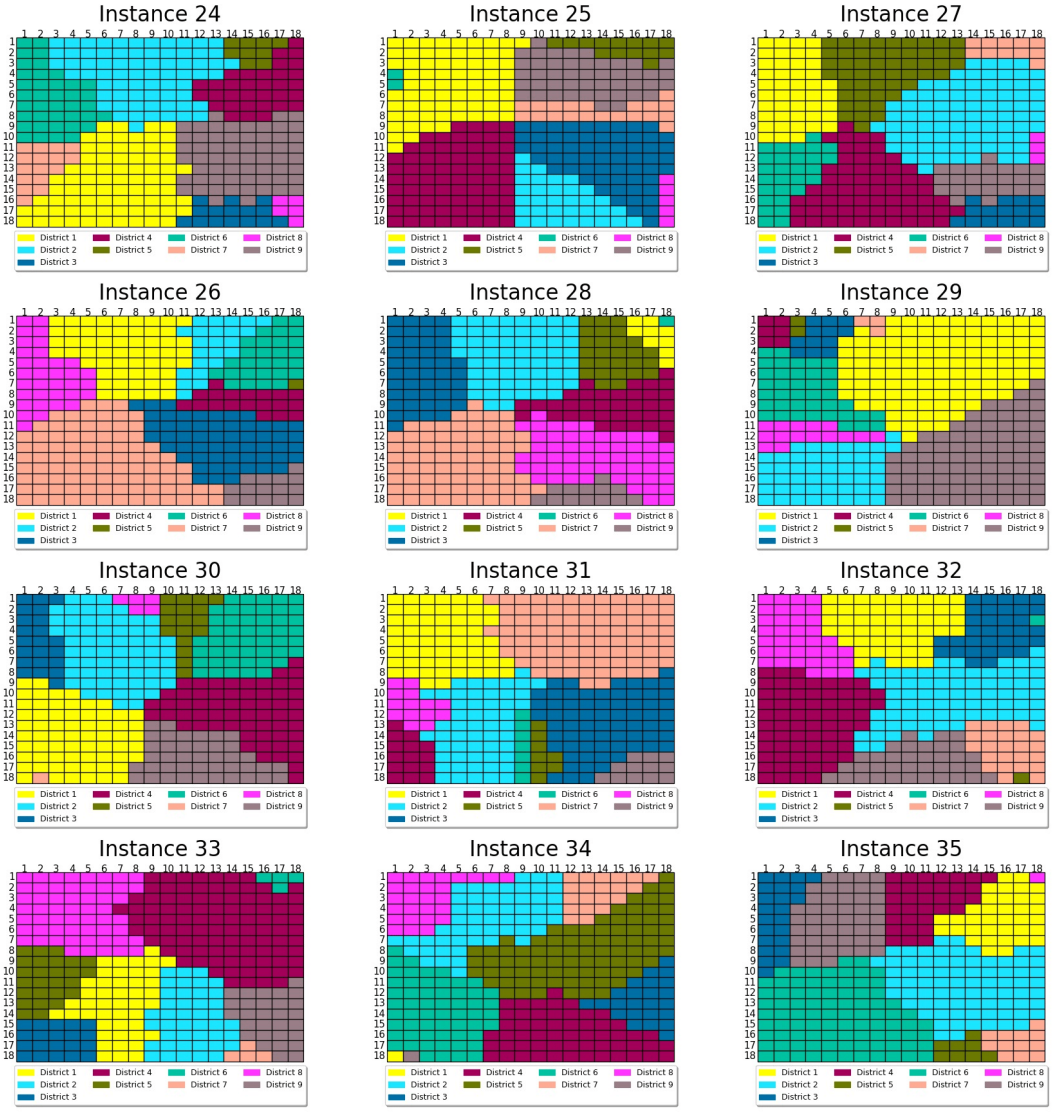

Figure 31. Parallel SA, Initial solutions, Test Suite 2, Region 18x18, Part 3

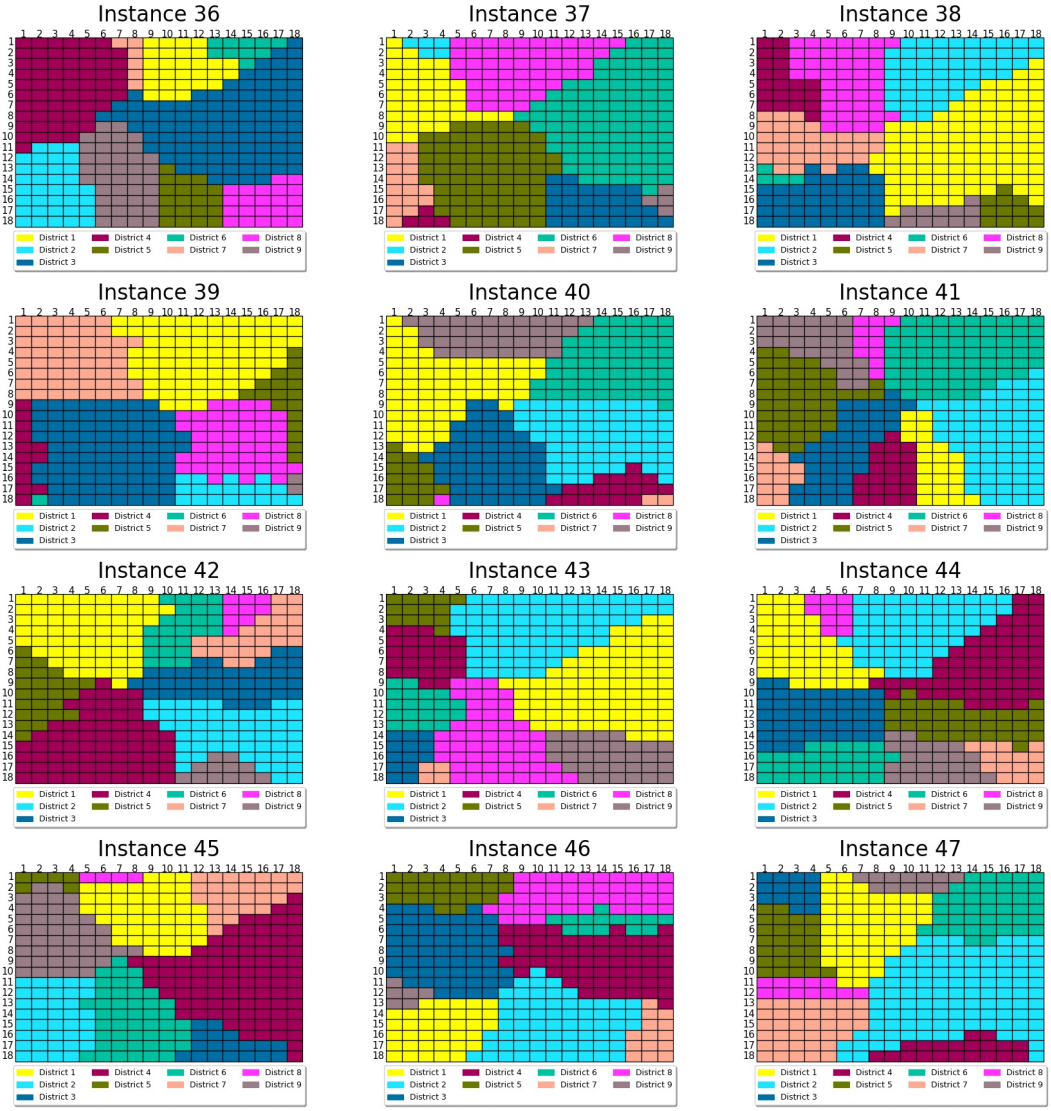

Figure 32. Parallel SA, Initial solutions, Test Suite 2, Region 18x18, Part 4

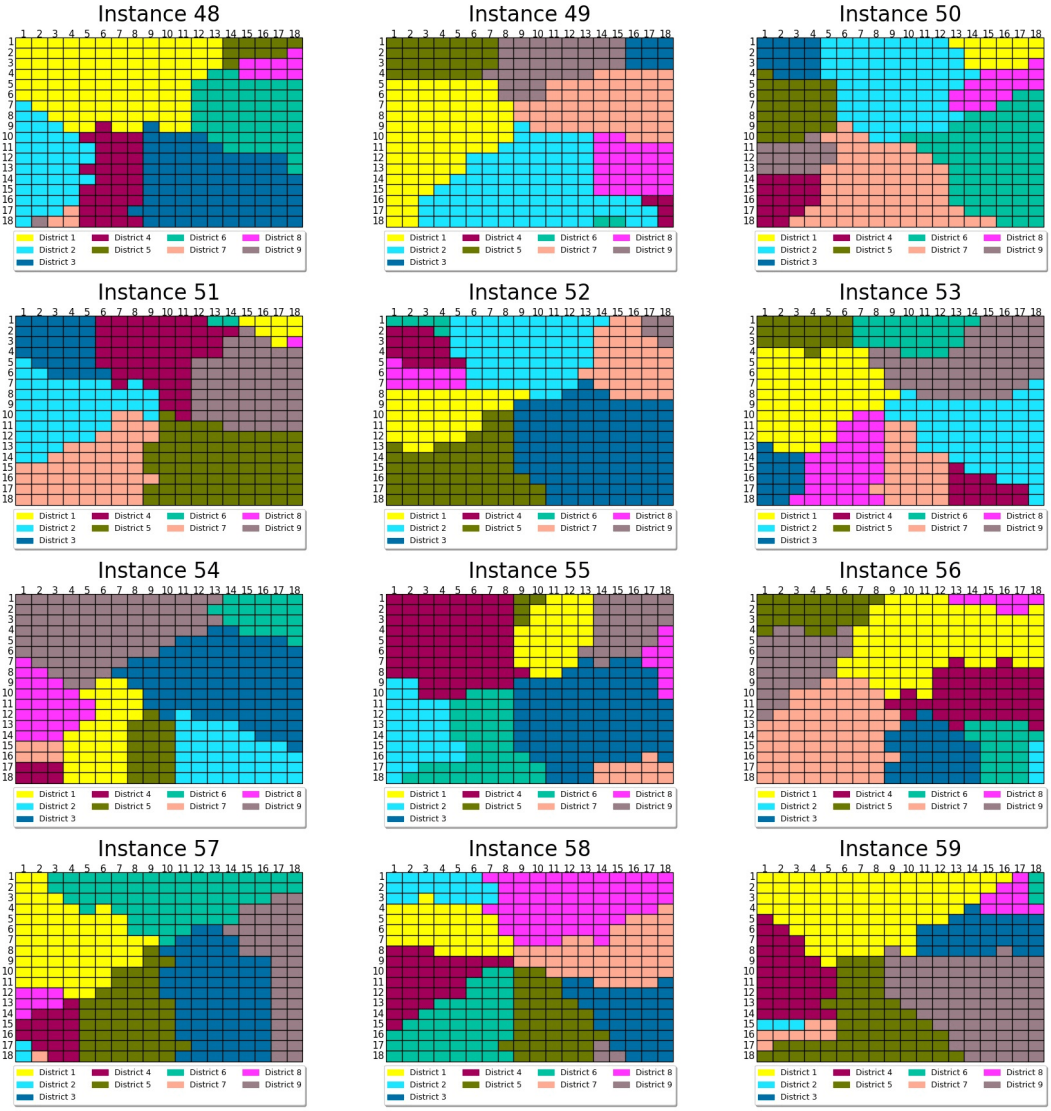

Figure 33. Parallel SA, Initial solutions, Test Suite 2, Region 18x18, Part 5

Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, Vjatšeslav Antoškin,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright of my thesis,

   Analysis of a Metaheuristic for Redistricting

   supervised by Benson Muite

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 14.05.2018