PRUC : P-Regions with User-Defined Constraint

Yongyi Liu
University of California, Riverside
Riverside, California
yliu786@ucr.edu

Ahmed R. Mahmood
Purdue University
West Lafayette, Indiana
amahmoo@cs.purdue.edu

Amr Magdy
University of California, Riverside
Riverside, California
amr@cs.ucr.edu

Sergio Rey
University of California, Riverside
Riverside, California
sergio.rey@ucr.edu

ABSTRACT

This paper introduces a generalized spatial regionalization problem, namely, PRUC (*P*-Regions with User-defined Constraint) that partitions spatial areas into homogeneous regions. PRUC accounts for user-defined constraints imposed over aggregate region properties. We show that PRUC is an NP-Hard problem. To solve PRUC, we introduce GSLO (Global Search with Local Optimization), a parallel stochastic regionalization algorithm. GSLO is composed of two phases: (1) *Global Search* that initially partitions areas into regions that satisfy a user-defined constraint, and (2) *Local Optimization* that further improves the quality of the partitioning with respect to intra-region similarity. We conduct an extensive experimental study using real datasets to evaluate the performance of GSLO. Experimental results show that GSLO is up to 100× faster than the state-of-the-art algorithms. GSLO provides partitioning that is up to 6× better with respect to intra-region similarity. Furthermore, GSLO is able to handle 4× larger datasets than the state-of-the-art algorithms.

PVLDB Reference Format:
Yongyi Liu, Ahmed R. Mahmood, Amr Magdy, and Sergio Rey. PRUC : P-Regions with User-Defined Constraint. PVLDB, 15(3): 491 - 503, 2022. doi:10.14778/3494124.3494133

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Yongyi-Liu/PRUC.

1 INTRODUCTION

Spatial regionalization is an important problem that aims at partitioning spatial areas into regions based on specific criteria. Spatial areas assigned to a region need to be spatially contiguous. Spatial regionalization has been adopted in numerous applications and domains, such as economics, e.g., imbalance in economic development [44], urban planning [24, 56], e.g., resource allocation in urban construction [24], environmental science [54, 55], e.g., understanding of environmental patterns in different geographical locations [55].

Spatial regionalization has multiple variations that are studied in the literature [3, 4, 6, 8, 9, 34, 35]. The p-regions problem [15, 17] is a popular spatial regionalization problem that partitions areas into p regions while maximizing the intra-region similarity with respect to a numerical attribute. For example, in urban planning, each area could be a municipality, and a region is a group of spatial contiguous municipalities. Maximization of the similarity of household income among municipalities within regions is an example of the numerical attribute. One important requirement of spatial regionalization is to account for user-defined constraints over regions. A typical use case in urban planning is to partition areas into a predefined number of regions where the total population of every region exceeds a specific threshold. Existing variations of the p-regions problem do not support user-defined constraints, which limit its applicability to various domains and a plethora of use cases.

In this paper, we formalize a generalized spatial regionalization problem, namely, PRUC (*P*-Regions with User-defined Constraint). PRUC aims to partition a set of areas into a predefined number of regions p while maximizing the similarity over a specific attribute, e.g., the household income. In PRUC, each region needs to satisfy a user-defined constraint on some aggregate attribute, e.g., the total population of each region needs to exceed a specific threshold. Table 1 shows 12 spatial areas to be partitioned into regions. In this example, it is required to partition the areas into three regions while maximizing the similarity of household income between the areas within each region. The user-defined constraint is having population above 500 in each region. Figure 1 shows an optimal PRUC partitioning that satisfies the aforementioned constraints.

PRUC is a generalization of the p-regions problem. The reason is that PRUC has the same optimization goal as the p-regions problem, but it enforces an additional user-defined threshold constraint on each region. The new input user-defined constraint in PRUC has introduced several challenges on building initial regions. First, existing techniques have a high probability, up to 80%, of producing regions that do not satisfy the input constraint. Second, producing valid solutions requires significant shuffling of spatial areas among initial regions so that invalid regions become valid but not the opposite. This adds a restrictive requirement on the spatial connectivity as regions that are vulnerable to spatial disconnection with shuffling retain the high probability of producing invalid solutions. Third, the additional overhead of producing valid solutions inflates the scalability problem and makes it harder to handle large datasets.

Proceedings of the VLDB Endowment, Vol. 15, No. 3 ISSN 2150-8097.
doi:10.14778/3494124.3494133

Table 1: Area attributes

Area	population	income
a1	210	1200
a2	120	1300
a3	180	1400
a4	150	1000
a5	120	2500
a6	180	2400
a7	150	2700
a8	160	3000
a9	150	4000
a10	100	4100
a11	180	4300
a12	180	4500

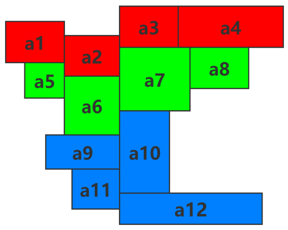


Figure 1: Partitioned areas

To address these challenges, we propose an efficient parallel algorithm called GSLO (Global Search with Local Optimization) to solve PRUC at scale. GSLO is stochastic and its results may vary on different runs. GSLO is composed of two phases: (1) Global Search and (2) Local Optimization. The Global Search phase proposes novel techniques to find a partitioning that satisfies the user-defined constraint with high probability of success. The regions grown in GSLO are robust against spatial disconnection and have a high probability of surviving the shuffling phase. The shuffling phase consists of two complementary steps that boost the probability of success. The Local Optimization phase employs parallel stages that incrementally improve the quality of the partitioning with respect to the similarity properties within each region.

There are two main approaches to build partitions, top-down edge-cut and bottom-up seeding. Top-down edge cut approaches [3, 4] are time-consuming and less flexible. They are time-consuming because they evaluate the effect of each edge cut on the entire graph and less flexible because once the edges are cut, there is no following reassignment of areas to regions to further optimize the objective of the regionalization. GSLO is a bottom-up seeding-based algorithm that can efficiently grow regions around seed areas locally, which breaks down the big problem into several smaller ones. The novel contributions of this paper are summarized as follows:

- We introduce a generalized spatial regionalization problem, namely, P-regions with User-defined Constraint (PRUC).
- We show that PRUC is an NP-Hard problem.
- We develop GSLO, an efficient parallel algorithm that solves PRUC with several novel techniques as follows.
  - A general seeding strategy that does not require any domain knowledge.
  - A region growth algorithm that maximizes the flexibility of the assignment of areas to regions.
  - Two novel complementary inter-region shuffling strategies that increase the ability of finding a feasible solution.
  - Heuristic search strategies to speed up optimizing the final solution and provide region-level parallelization.
- We conduct an extensive experimental evaluation using real datasets.

Our extensive experimental study shows that GSLO runs up to 100× faster, achieves up to 6× better solution quality, and scales up to 4× larger datasets than the state-of-the-art algorithms. The rest of this paper is organized as follows: Section 2 presents the related work on spatial regionalization. Section 3 defines the problem, and Section 4 proves it is an NP-hard problem. Section 5 details our proposed algorithm GSLO. Section 6 analyzes the complexity of GSLO. Section 7 presents an extensive experimental evaluation and Section 8 concludes the paper.

2 RELATED WORK

Spatial regionalization [3, 4, 8, 9, 14, 16, 27, 34] refers to the problem of grouping spatial areas into multiple regions that are spatially contiguous. There are several variations of the spatial regionalization problem. The p-regions problem [15] finds p regions that maximize the similarity between areas within a region. The p-compact regions problem [35] finds p regions that maximize the spatial compactness. The max-p regions problem [14] computes a maximal-sized partitioning that maximizes the similarity between areas within a region. PRUC is a generalization of the p-regions problem that enforces user-defined constraints.

Traditional clustering algorithms, e.g., k-means [37], are not directly applicable in spatial regionalization problems, as they are mainly designed for points rather than polygons and they do not enforce spatial contiguity that is required in spatial regionalization. Some techniques have tried to adapt them for regionalization in different contexts [18, 40, 51]. However, the lack of enforcing spatial contiguity constraints early in the algorithm makes it harder to scale for large datasets. To this end, the following approaches have been proposed to address spatial regionalization: *(1) linear programming*, *(2) graph partitioning*, and *(3) seeding*.

Duque et al. [15] transforms a spatial regionalization problem into a **mixed integer programming (MIP)** problem that can be solved using software package such as CPLEX [12]. However, this approach works only for tiny datasets. Hence, this approach is not suited to address PRUC over large inputs.

Graph-partitioning is used in SKATER [3] and SKATERCON [4] as the state-of-the-art techniques to address the p-regions problem. In this approach, graph nodes represent spatial areas and edges connect spatially contiguous areas. In SKATER, a minimum spanning tree (MST) is computed from the graph. The MST is then split into p subtrees, each corresponding to one of the p regions. However, the greedy approach adopted in MST generation results in suboptimal regions with low quality. Also, SKATER is computationally expensive and cannot handle large inputs. SKATERCON [4] enhances SKATER by generating multiple random spanning trees (RST). SKATER is then applied to all RSTs to generate multiple regionalization results. The different results are combined into a single solution using a consensus-based method [26]. SKATERCON is slower than SKATER, and cannot handle large inputs. Neither SKATER nor SKATERCON can be directly applied to solve PRUC as they do not support user-defined constraints.

Seeding is an important category of spatial regionalization algorithms. In seeding, multiple spatial areas are chosen as seeds for spatial regions. Then, regions grow around seeds by incrementally adding neighboring areas. REGAL [9] and SPATIAL [8] are two seeding algorithms that have been tailored to solve the school redistricting problem, so they cannot be used to solve PRUC. The reason is that they do not support general user-defined constraints. MERGE [35] is a seeding framework for solving the p-compact regions problem. However, MERGE cannot be used to solve PRUC as it is tailored to the p-compact regions problem and does not support user-defined constraints as well.

Recently, parallelization has been adopted to speedup spatial regionalization algorithms. Laura et al. [33] introduced a parallel algorithm to solve the p-compact-regions problem. Also, Sindhu et al. [48] used a parallel algorithm to address the max-p regions problem. Similarly, GSLO is a parallel algorithm to make the best use of multi-core environments.

3 PROBLEM DEFINITION

In this section, we give a formal definition of PRUC. Table 2 summarizes the notations used throughout this paper. Spatial regionalization is the problem of partitioning spatial areas into non-overlapping regions while satisfying specific constrains. An ***area***, say a, is a spatial polygon that is represented by a set of geographical coordinates, i.e., longitude and latitude. Two areas are neighbors if they share a common border. The list of neighbor areas of an area, say a_i, is represented as $a_i.NBR_A$.

A ***region***, say r, is a set of spatially contiguous areas $\{a_i, a_j, ...\}$. Each r has a unique identifier $r.id$. Figure 1, shows spatial areas that are partitioned into regions, where areas having the same color constitute a region. Each spatial area is associated with numerical attributes, e.g., population and household income as shown in Table 1. We denote the attribute used to quantify the similarity among areas as the ***similarity attribute***, i.e., $a_i.sim$. For example, in Figure 1, the average household income of an area is the *similarity attribute*. This attribute is used to group areas into regions having similar household income. We call the attribute used in region constraints the ***extensive attribute***. The *extensive attribute* of an area, say a_i, is represented as $a_i.ext$. In Figure 1, $a_i.ext$ is the population. For example, $a_1.ext = 210$. The region to which an area, say a, belongs to is termed $a.r$. The aggregate *extensive attribute* of region r is represented as $r.ext$, that is defined as the sum of the *extensive attribute* over all the areas in the region. In Figure 1, the *extensive attribute* of the green region refers to the total population of this region, which is computed as $r_{green}.ext = a_5.ext + a_6.ext + a_7.ext + a_8.ext = 610$.

The set of neighbor areas of a region r, i.e., $r.NBR_A$, is defined as the set of areas that do not belong to r and are neighbor to at least one area in r. In Figure 1, $r_{red}.NBR_A = \{a_5, a_6, a_7, a_8\}$. Formally,

$$r.NBR_A = \{a | \exists a_i (a_i.r = r.id \land a.r \neq a_i.r \land a_i \in a.NBR_A)\}$$

The set of neighbor regions of an area, i.e., $a.NBR_R$, is defined as the set of regions that have at least one area a_i within the region that is a neighbor area of a. In Figure 1, the neighbor regions of a_6 = $\{r_{red}, r_{blue}\}$. Formally,

$$a.NBR_R = \{r | \exists a_i (a_i.r = r.id \land a.r \neq r.id \land a_i \in a.NBR_A)\}$$

The set of margin areas of a region, say r, is defined as the set of areas that have at least one neighbor area that belongs to a neighbor region or is unassigned. In Figure 1, all of the areas in r_{red} are margin areas because they have at least one neighbor area that belongs to a neighbor region. Formally,

$$r.margin = \{a | a.r = r.id \land \exists a_i (a_i \in a.NBR_A \land a_i.r \neq a.r)\}$$

An area is considered an ***articulation*** area if removing this area disconnects its region, i.e., areas of the region are not contiguous. The set of *articulation* areas in r is represented as $r.art$. In Figure 1, $r_{red}.art = \{a_2, a_3\}$, since removing any of them breaks the region's contiguity.

A ***user-defined constraint*** is a numerical constraint that all regions must satisfy. In Figure 1, the user-defined constraint is that the aggregate population of each region must be at least 500. A region is ***incomplete*** if it does not satisfy the user-defined constraint and ***complete*** if it does.

A ***partition***, say P, of a set of areas is the set of regions $\{r_1, r_2, ..., r_p\}$ that includes all areas. Each area belongs to only one region. P is ***feasible*** if all its regions are *complete*, i.e., satisfy the user-defined constraint.

Heterogeneity is inversely proportional to the similarity among areas. The heterogeneity of a_i and a_j reflects the degree of dissimilarity between a_i and a_j and it is defined as the absolute difference between the *similarity attribute* of the two areas:

$$h(a_i, a_j) = |a_i.sim - a_j.sim|$$

For example, in Figure 1, the *similarity attribute* is the average household income. $h(a_1, a_2) = |a_1.sim - a_2.sim| = 100$. The heterogeneity of a region, say r, is defined as the heterogeneity sum of all pairs of areas in r:

$$h(r) = \sum_{\forall i<j, a_i.r = a_j.r = r.id} h(a_i, a_j)$$

In Figure 1, $h(r_{red}) = h(a_1, a_2) + h(a_1, a_3) + h(a_1, a_4) + h(a_2, a_3) + h(a_2, a_4) + h(a_3, a_4) = 1300$. The heterogeneity of a partition $h(P)$ is defined as the sum of the heterogeneity of all the regions:

$$h(P) = \sum_{\forall r \in P} h(r)$$

In Figure 1, $h(P) = h(r_{red}) + h(r_{green}) + h(r_{blue}) = 5000$.
A good partition has low heterogeneity and high intra-region similarity.

PRUC Problem. P-Regions with User-Defined Constraint (PRUC) problem is formally defined as follows: Given: (1) A set of n areas: $A = \{a_1, a_2, ..., a_n\}$. (2) An integer p. (3) A threshold T. PRUC finds a partition of regions $P = \{r_1, r_2, ..., r_p\}$ of size p, where each region r_i is a non-empty set of spatially continuous areas, $|r_i| \geq 1$, so that: (i) $r_i \cap r_j = \Phi, \forall r_i, r_j \in P \land \ i \neq j$, i.e., all regions are disjoint. (ii) $\bigcup_{i=1}^{p} r_i = A$. (iii) $r_i.ext > T$. (iv) The heterogeneity of P, $h(P)$, is minimum.

4 NP-HARDNESS OF PRUC

In this section, we provide a proof for the NP-hardness of PRUC problem using a reduction from the Node-attributed Spatial Graph Partitioning (NSGP) problem [6]. NSGP is an NP-Hard problem that aims to partition a node-attributed spatial graph into k subgraphs. The number of nodes in each subgraph must exceed a specific threshold. The nodes of this graph represent spatial locations and each node has an associated set of attributes. The objective of the NSGP is to minimize the heterogeneity of the generated subgraphs

Table 2: Summary of Notations

Notation	Description
n	The number of areas
p	The number of regions
A	A set of areas
a	A spatial area
P	A partition
e	An enclave area
s	A seed
r	A region
$a.r$	The region that a is assigned to
$a.ext$	The extensive attribute of area a
$a.sim$	The similarity attribute of area a
$r.margin$	The margin areas of r
$r.art$	The articulation areas of r
NBR_A	Neighboring areas
NBR_R	Neighboring regions
$h()$	Heterogeneity
$c(r)$	The number of areas in r
$hi(a, r)$	Heterogeneity increase of a to r
$conn(a, r)$	connectivity of a to r
$MBDRY(r_1, r_2)$	movable boundary between r_1 and r_2

and the number of edges with endpoints belonging to different subgraphs.

Let X be an instance of NSGP problem and $X = (A, E, N, k, s, g)$ where A is the set of spatial areas, E is the set of neighborhood relations, N is the set of node attributes, k is the number of subgraphs, s is the minimum number of nodes in a subgraph, and g is the optimization goal of NSGP. Let Y be an instance of PRUC problem and $Y = (A, p, T, f)$ where A is the set of spatial areas, p is the predefined number of regions, and T is the user-defined threshold. f is the optimization goal of PRUC. We make f to be the same as g (Note that changing the optimization goal in PRUC would not affect the way it works). We set the *extensive attribute* of each area in A to 1, and set p equal to k in the NSGP problem. Thus X is a special case of Y, and we construct Y from X in polynomial time, hence the proof.

5 PROPOSED SOLUTION

We introduce Global Search with Local Optimization (GSLO), a two-phase algorithm to efficiently address PRUC. The Global Search phase aims to find a *feasible* partition. Local optimization aims to further improve the heterogeneity over the partition without violating the user-defined constraint.

5.1 Global Search

The Global Search phase aims to find a *feasible* partition with high probability of success. This phase is divided into the following steps: Seed Identification, Region Growth, Enclaves Assignment, Inter-region Update, and Indirect Flow Push. Each step is optimized to increase the probability of successfully finding a *feasible* partition. The rest of this section details each step.

5.1.1 ***Seed Identification***. A seed, say s, is a set of p areas, i.e, $s = [a_1, ...a_p]$. Regions incrementally grow by attaching unassigned neighbor areas to the seed areas. The seeding-based regionalization literature [8, 9, 14, 33–35, 53] either selects seed areas randomly [14, 33–35, 53] or selects the seed areas manually according to problem-specific guidelines [8, 9]. In this section, we propose a general seeding method that does not require any domain knowledge. Having seed areas close to each other can restrict the growth of regions and increase the probability of failure. Hence, the objective is to select a seed whose areas are scattered. The distance between a_i and a_j refers to the euclidean distance between the centroids of a_i and a_j and is represented as $dist(a_i, a_j)$. The quality $q(s)$ of a seed s is defined as the minimum euclidean distance between the centroids of all pairs of areas in s.

$$q(s) = \min_{a_i \in s \wedge a_j \in s \wedge i \neq j} dist(a_i, a_j)$$

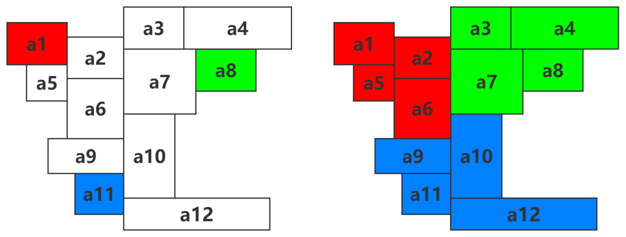
The objective is to maximize $q(s)$. First, p areas are selected randomly as the seed. The pair of seed areas having the least pairwise distance, say (a_i, a_j), is identified. Then, an area that does not belong to the seed is chosen at random to replace one of the areas (a_i, a_j). The replacement takes place only when there is improvement in $q(s)$. The last step is repeated m times, where m is a user-defined parameter.

The above Seed Identification algorithm assumes no islands present in the dataset, i.e., there is only one connected component. To support datasets with islands, first, we run a graph traversal algorithm on the spatial neighborhood graph to detect different connected components. We then compute the total *extensive attribute* on each connected component. If the total *extensive attribute* of any island is less than the user-defined constraint value, then the current user-defined constraint cannot be solved. Otherwise, the connected components are sorted in ascending order according to their total *extensive attribute*, i.e., $\forall 0 < i < j \leq C, cc_i.ext \leq cc_j.ext$, where C is the number of connected components, cc_i is the i^{th} component, and $cc_i.ext$ is cc_i's *extensive attribute*. Starting from cc_1, for each component cc_i, we put a number of seed areas proportional to the ratio of $cc_i.ext$ divided by the total *extensive attribute* of the whole input, where at least one seed area is placed in each component cc_i. In each connected component, we perform a number of iterations that are proportional to its number of seed areas to scatter seed areas in space as described above.

The Seed Identification phase aims to find spatially scattered seed areas. There are several metrics that can be used to quantify seed quality, i.e., scatteredness of the seed, e.g., sum of pairwise distance or minimum pairwise distance between seed areas. We choose minimum pairwise distance between seed areas as it guarantees that no pair of seed areas are close to each other. Other metrics may result in nearby seed areas that restrict region growth. Figure 4 shows that as the seed quality monotonically increases, the heterogeneity of the partition improves and converges to an optimal value.

Complexity analysis. Seed identification performs m iterations to improve the seed quality. The number of seed areas is p, each iteration takes $O(p^2)$ time, which gives a total time of $O(mp^2)$.

Remark 5.1. *Time complexity of Seed Identification is $O(mp^2)$.*



(a) Before Region Growth **(b) After Region Growth**

Figure 2: Region Growth

5.1.2 ***Region Growth.*** After the Seed Identification step, the seed areas become the initial p regions that will subsequently grow. A growing step of a region, say r, adds one of the unassigned areas from neighbor areas, i.e., $r.NBR_A$ to r. A region r stops growing when: (1) r satisfies the user-defined constraint, i.e., becomes a *complete* region, or (2) all the neighbor areas of r are assigned to other regions. If the user-defined constraint is not met for a region, the region is marked *incomplete*. The region with the least *extensive attribute* is chosen for each growing step in order to achieve a balanced distribution on the *extensive attribute* over each region.

Having many *articulation* areas in regions restricts the movement of areas across regions. This hinders the ability to find a *feasible* partition or the refinement of the partition in Local Optimization. So, a main objective of this step is minimizing the number of *articulation* areas in the partition. We define the *robustness* of a region, say r, to be the number of areas in its margin, i.e., $r.margin$ divided by the number of *articulation* areas in $r.margin$. The greater the *robustness* of a region, the less likely r becomes disconnected while attempting to move an area to the neighbor region.

Region Growth algorithm grows regions while attempting to increase their robustness. A basic approach would be to iterate over all the unassigned areas of $r.NBR_A$ and choose the area that gives the greatest increase in the *robustness* of r. However, identifying the *articulation* areas for every growing step is rather expensive. To this end, we adopt an approximate approach to find areas to be added to regions that improves the *robustness* of regions. We define the connectivity between a region r and an area a as the number of neighbor areas of a that belong to r.

$$conn(a, r) = |\{a_i | a_i.r = r.id \land a_i \in a.NBR_A\}|$$

Region Growth algorithm grows a region, say r, by choosing the neighbor area, say a, that has the greatest connectivity, i.e., $conn(a, r)$. We call this area a_{rbest}. Ties are broken arbitrarily.

Region Growth phase aims at building robust regions that provide reassignment flexibility rather than focusing only on heterogeneity. Region robustness is achieved by reducing the number of articulation areas. So, the region sustains its spatial connectivity even after moving areas to another region. This allows flexibility in area reassignments across regions in subsequent phases of GSLO, and leads to improved effectiveness and heterogeneity.

Figure 2 illustrates the Region Growth. Figure 2(a) illustrates the initial seed from the Seed Identification step. Figure 2(b) shows the regions after Region Growth. There are no *articulation* areas in any region. Hence, all regions have great *robustness*.

Complexity analysis. The Region Growth phase incrementally grows the regions from the seed areas. Assume $c(r)$ denotes the number of areas in region r, and r^i denotes the region that is selected to grow in the i^{th} iteration. In each iteration, retrieving the region with the minimum $r.ext$ takes $O(p)$. For a growing region r, we need to evaluate all its unassigned neighbor areas. Spatial neighborhood relations of areas are represented with a planar graph where nodes are areas and an edge exists between any pair of neighbor areas. Since the average degree of the vertices in a planar graph is strictly less than six [50], this implies: (i) the size of $r.NBR_A$ is $O(6c(r)) = O(c(r))$, and (ii) computing $conn(a, r)$, for $a \in r.NBR_A$, is $O(1)$. Meanwhile, computing the heterogeneity increase of a to r requires time $O(c(r))$ because we need to compute the heterogeneity between a and all areas in r. Consequently, in each iteration, growing a region r takes time $O(c(r)) + O(p)$. Region Growth phase performs in total $O(n)$ iterations, each adds an area to a region. Then, the total runtime of Region Growth phase is $(\sum_{i=1}^{n} O(c(r^i)) + O(p))$, where $1 < c(r^i) < n$, which is $O(n^2) + O(np) = O(n^2)$.

Remark 5.2. *Time complexity of Region Growth phase is* $O(n^2)$.

5.1.3 ***Enclaves Assignment.*** After the Region Growth step, some areas may remain unassigned. The reason is that Region Growth of a region terminates when it satisfies the user-defined constraint. This can prevent some areas from being assigned to regions. We name the remaining unassigned areas enclaves. In Enclaves Assignment, all enclaves are identified and processed one by one. The intuition of Enclaves Assignment is rather straightforward. An enclave area is assigned to a region that minimizes heterogeneity increase to keep the overall heterogeneity score at its minimum level before the Local Optimization phase. If an enclave, say a, is surrounded by only enclaves, it can not be assigned to any neighbor region at this moment. The assignment of a is delayed until some or all surrounding enclaves have been assigned to regions. If a is surrounded by one or more *complete* regions, we assign this enclave to the region with the minimum heterogeneity increase [14].

Complexity analysis. In Enlaves Assignment, there are v enclaves, $v < n$. Retrieving the next enclave to process takes $O(v)$. Processing each enclave is $O(n)$, in the worst case, to compute heterogeneity increase to all neighboring regions. This gives time complexity $O(v + n)$ for processing a single enclave, which is $O(v * (v + n)) = O(v^2 + vn)$ for v enclaves. As $v < n$, i.e., $v = O(n)$, the phase complexity is bounded by $O(n^2)$ in its worst case.

Remark 5.3. *Time complexity of Enclaves Assignment is* $O(n^2)$.

5.1.4 ***Inter-region Update.*** After the Enclaves Assignment step, all areas are assigned to regions. However, *incomplete* regions, i.e., regions that fail to satisfy the user-defined constraint, might still exist. This step attempts to render all regions *complete* by moving some areas from *complete* regions to neighbor *incomplete* regions. First, *incomplete* regions are identified and added to a queue. Then, for every *incomplete* region r_i, all its *complete* neighbor regions are identified. The Inter-region Update algorithm attempts to make r_i *complete* by moving an area a from one of r_i'*s complete* neighbor region to r_i. The region that donates an area is called r_{donor} and the region that receives that area is called $r_{receiver}$. A move is defined as a triple (a , r_{donor} , $r_{receiver}$).

For a given r_{donor} and a given $r_{receiver}$, Area a from r_{donor} is movable if it satisfies all the following properties:

- a is not an *articulation* area for r_{donor}, i.e. $a \notin r_{donor}.art$.
- a is a neighbor area of $r_{receiver}$, i.e., $a \in r_{receiver}.NBR_A$.

The *articulation* areas of a region are identified using Tarjan algorithm [45] to speedup excluding invalid moves that cause spatial disconnection. The above conditions do not prevent moves that switch *complete* regions to *incomplete* regions, which is an incorrect switch. However, this gives more flexibility and higher scalability to this step to fill *incomplete* regions. In case this incorrect switch happens for some regions, they are switched back to *complete* regions in the following step that indirectly flow extra areas from *complete* regions to all other *incomplete* regions.

Algorithm 1 describes the Inter-region Update step. In each iteration, we dequeue an *incomplete* region and consider it as $r_{receiver}$. If $r_{receiver}$ does not have a *complete* neighbor region, then we add the $r_{receiver}$ back to the queue to be processed later. Otherwise, the $r_{receiver}$'s *complete* neighbor regions are sorted based on the *extensive attribute* in descending order. Suppose the sorted *complete* neighbor regions are $\{r_a, r_b, r_c, ...\}$, we attempt to consider neighbor region with largest *extensive attribute* r_a as the r_{donor} and try to find the movable area from r_{donor} to $r_{receiver}$ that has the largest *extensive* attribute. If the list of movable areas is empty, then we turn to the region with the second-largest *extensive attribute* r_b, and so on. If no movable area is found among all the *complete* neighbor regions, then we put the current $r_{receiver}$ back to the queue to be processed later and start the next iteration. If $r_{receiver}$ is still *incomplete* after the move, then we add $r_{receiver}$ back to the queue. If r_{donor} becomes *incomplete* after the move, then we add r_{donor} to the queue as well. This procedure is repeated until a *feasible* partition is found or the maximum of n iterations allowed has been exhausted without finding a *feasible* partition.

Complexity analysis. In each iteration, for an *incomplete* region r, retrieving neighbor regions is $O(c(r)) = O(n)$ and sorting them is $O(plogp)$. Applying Tarjan algorithm takes $O(c(r)+e(r))$, where $e(r)$ denotes the number of edges within a spatial neighborhood graph G for region r's areas. As region r is also a planar graph, it has $e(r) \leq 3c(r) - 6$ [50], so $e(r) = O(c(r))$, and applying Tarjan algorithm is $O(c(r)) = O(n)$. Moving an area a from a donor region r' to a receiver region r takes $O(c(r) + c(r')) = O(n)$ time to locate a and compute heterogeneity changes of r and r'. The same applies if multiple donor regions are explored. Then, the runtime of a single iteration is $O(plogp + n)$. For n iterations, the overall complexity is $O(n^2 + nplogp)$.

Remark 5.4. *Time complexity of Inter-region Update is* $O(n^2 + nplogp)$.

Algorithm 1: Inter-region Update

```
Input: R : regions
       maxiter : the maximum number of attempts
Output: a feasible partition or FAILURE

incomplete-rs = new queue()
for i = 0 to p do
    if R[i] is incomplete then
        incomplete-rs.enqueue(R[i])
for i = 0 to maxiter do
    if incomplete-rs is empty then
        return R
    $r_{receiver}$ = incomplete-rs.dequeue()
    if $r_{receiver}$ does not have complete neighbor region then
        incomplete-rs.enqueue($r_{receiver}$)
    else
        donors = all $r_{receiver}$'s neighbor complete regions
        while donors is not empty do
            $r_{donor}$ = r with max r.ext from donors
            a = movable area from $r_{donor}$ that has the largest
              extensive attribute
            if a is not null then
                move a from $r_{donor}$ to $r_{receiver}$
                if $r_{donor}$ is incomplete then
                    incomplete-rs.enqueue($r_{donor}$)
                break
            else
                donors.remove($r_{donor}$)
    if $r_{receiver}$ is incomplete then
        incomplete-rs.enqueue($r_{receiver}$)
return FAILURE
```

5.1.5 ***Indirect Flow Push***

If there are remaining *incomplete* regions after Inter-region Update, then Indirect Flow Push is adopted to attempt transforming these regions into *complete*. In Inter-region Update, after an *incomplete* region r_i is converted to a *complete* region, it might serve as a donor region for some other *incomplete* neighbor region since it has now become *complete*. However, moving an *area* from r_i to its neighbor *incomplete* regions would likely to make r_i *incomplete* again because the *extensive attribute* of r_i is just above the threshold. In this case r_i is converted back to *incomplete* again. Those *incomplete* regions could frequently change status between *complete* and *incomplete*. This makes it hard for all the *incomplete* regions to become *complete* regions. We call this phenomena the *chained-flipping problem*.

We propose Indirect Flow Push to solve the *chained-flipping problem*. This phase is entered only if Inter-region Update does not find a *feasible* partition. The chained-flipping problem is caused by starting from *incomplete* regions and borrowing areas from neighbor *complete* regions. In Indirect Flow Push, instead of starting from *incomplete* regions and borrowing areas from the neighbor *complete* regions, we start with the *complete* regions with the largest *extensive attribute* and push its margin areas to neighbor regions that need them.

The partition of regions is considered as a flow network where regions are considered as nodes and *extensive attribute* is considered as flow. We push the flow through the network to ensure that there is a balanced distribution of *extensive attribute* over the regions. Each region in the flow network is assigned a state from the following:

- *Unprocessed* (UP): This is the initial state of a region. This region has at least two neighbor regions that it could donate areas to or receive areas from.

- *Exhausted-incomplete* (EI): This is an *incomplete* region having only one neighbor region that it can donate areas to or receive areas from.
- *Exhausted-complete* (EC): This is a *complete* region having only one neighbor region that it can donate areas to or receive areas from.
- *Processed* (P): This is the final state of a region. This region cannot donate or receive other areas.

Algorithm 2: Indirect Flow Push

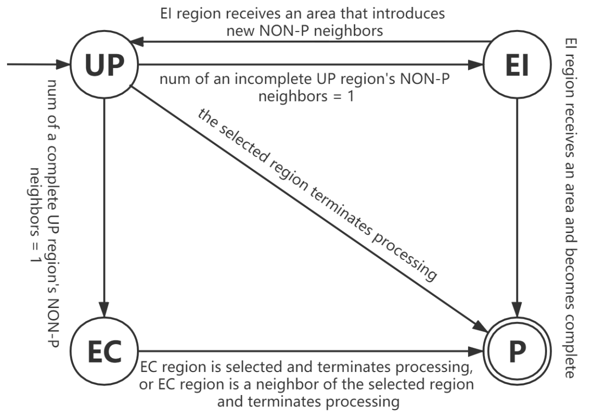
```
Input: R : set of regions
Output: a feasible partition or FAILURE

while exists incomplete region do
    if exists UP region then
        $r_{select}$ = UP region with max r.ext
    else
        $r_{select}$ = EC region with max r.ext
        if $r_{select}$ is null then
            return FAILURE
    $RS_{EC}$ = EC regions among $r_{select}.NBR_R$
    while $RS_{EC}$ is not empty do
        $r_g = r$ with max $r.ext$ from $RS_{EC}$
        $a_{best}$ = best area from $MBDRY(r_g, r_{select})$
        if $a_{best}$ is null then
            $r_g.status = P$
        else
            move $a_{best}$ from $r_g$ to $r_{select}$
    $RS_{EI}$ = EI regions among $r_{select}.NBR_R$
    while $RS_{EI}$ is not empty do
        $r_s = r$ with min $r.ext$ from $RS_{EI}$
        $a_{best}$ = best area from $MBDRY(r_{select}, r_s)$
        if $a_{best}$ is null then
            return FAILURE
        else
            move $a_{best}$ from $r_{select}$ to $r_s$
            if $r_s$ is complete then
                $r_s.status = P$
    $RS_{UP}$ = UP regions among $r_{select}.NBR_R$
    while $RS_{UP}$ is not empty do
        $r_s = r$ with min $r.ext$ from $RS_{UP}$
        $a_{best}$ = best area from $MBDRY(r_{select}, r_s)$
        if $a_{best}$ is null then
            $RS_{UP}$.remove($r_s$)
        else
            move $a_{best}$ from $r_{select}$ to $r_s$
    $r_{select}.status = P$
return R
```

Initially, all the regions are labeled as *UP*. Notice that regions with only one neighbor region are labeled as *EC* or *EI* according to their satisfaction of the user-defined constraint. The Indirect Flow Push step keeps track of all *incomplete* regions. At any stage, if the partition no longer contains *incomplete* region, this step terminates.

We define the movable boundary between r_1 and r_2 *MBDRY(r_1,r_2)* to be the set of areas A where each area a in A needs to satisfy the following properties:

- a belongs to r_1 , i.e., $a.r = r_1.id$, and neighbor to r_2, i.e., $a \in r_2.NBR_A$.



EI region receives an area that introduces new NON-P neighbors
UP
EI
num of an incomplete UP region's NON-P neighbors = 1
num of a complete UP region's NON-P neighbors = 1
the selected region terminates processing
EI region receives an area and becomes complete
EC
P
EC region is selected and terminates processing, or EC region is a neighbor of the selected region and terminates processing

Figure 3: The states of regions in Indirect Flow Push

- Removing a from r_1 would not make r_1 *incomplete*, i.e., $r_1.ext - a.ext > threshold$.
- a is not an *articulation* area for r_1, i.e., $a \notin r_1.art$.

In this phase, for a given $MBDRY(r_1, r_2)$, the best area in $MBDRY(r_1, r_2)$ to move from r_1 to r_2 is defined as the area a_{best} that maximizes $conn(a_{best}, r_2) - conn(a_{best}, r_1)$. Ties are broken arbitrarily. The area chosen to be moved from r_1 to r_2 has the most connections to areas in r_2 compared to r_1. Notice that this move may not result in the best heterogeneity improvement because the objective here is to ensure high robustness of regions. This allows areas to move without disconnecting regions.

Then, in each iteration, if there are *UP* regions, we select the *UP* region with the largest *extensive attribute* to be processed and name it as r_{select}. If there are no *UP* regions but *EC* regions, then we choose the *EC* region with the largest *extensive attribute.* If there are no *UP* or *EC* regions while having *incomplete* regions, then GSLO fails to identify a *feasible* partition of the input areas.

Algorithm 2 describes Indirect Flow Push and proceeds as follows: In each iteration, if r_{select} has *EC* neighbor regions, we select the *EC* neighbor region with the largest *extensive attribute*, say r_g and we compute $MBDRY(r_g, r_{select})$. If $MBDRY(r_g, r_{select})$ is empty, then r_g is transformed to *P*. The reason is that r_g is *complete* and cannot afford to donate any other area. Otherwise, we move a_{best} from r_g to r_{select}. If r_{select} has *EI* neighbor regions, we take the *EI* neighbor region with the least *extensive attribute*, say r_s and we compute $MBDRY(r_{select}, r_s)$. If $MBDRY(r_{select}, r_s)$ is empty, and r_s is still *incomplete*, then the last chance of making r_s *complete* has been exhausted. In this case, Indirect Flow Push step fails. Otherwise, we move a_{best} from r_{select} to r_s. If r_s becomes *complete* after this move, then r_s is transformed into *P*. Notice that when the *EI* neighbor region receives an area from r_{select}, one or more new *NON-P* neighbor regions of this *EI* region might be introduced. If this is the case, this *EI* region converts to *UP* as it now has two or more *NON-P* neighbor regions. If r_{select} does not have any *EC* or *EI* neighbor regions, then we choose the *UP* neighbor region with the least *extensive attribute*, say r_s', where $MBDRY(r_{select}, r_s')$ is not empty. If no movable area is found after all the neighbor *UP* regions are exhausted, r_{select} converts to *P*. If $MBDRY(r_{select}, r'_s)$ is not empty, we move a_{best} from r_{select} to r'_s.

For a given r_{select}, we give priority to *EC* and *EI* neighbor regions. This is the only opportunity for these regions to exchange an area with r_{select}. For a neighbor *EC* region, we move the *EC* region's margin areas to r_{select} until any further move would disconnect the *EC* region or make the *EC* region *incomplete*. This is because we want the total *extensive attribute* of this region to be just above the threshold. After the processing of r_{select}, all the redundant *extensive attribute* in this *EC* neighbor region would become stagnant, as this *EC* neighbor region will convert to *P*. For an *EI* neighbor region, r_{select} is the last opportunity to make it *complete*. Once r_{select} finishes processing, the *EI* neighbor region will not have a chance to exchange an area with its neighbor regions. *EI* region is converted to *P* once it becomes *complete*. We can think of areas within regions as flow that is being pushed from regions that have high *extensive attribute* to regions that have low *extensive attribute*. The state diagram of the Indirect Flow Push step is shown in Figure 3.

Complexity analysis. It takes $O(p)$ to find the neighbor region $\bar{r}$ that has either the minimum or the maximum *extensive* attribute. Then, computing $MBDRY(r, \bar{r})$, using Tarjan algorithm, takes $O(c(r))$. Filtering out all the areas in *r* that make *r incomplete* when removed or not in $\bar{r}.NBR_A$ also takes $O(c(r))$. Last of all, evaluating $conn(a, r)$ and $conn(a, \bar{r})$ for the remaining areas that takes $O(c(r))$, since there are $(c(r))$ areas in the boundary and computing $conn(.)$ is $O(1)$ due to the average constant degree of a node in a spatial neighborhood graph, which is a planar graph. Also, computing the heterogeneity variation on r and $\bar{r}$ takes time $O(c(r) + c(\bar{r}))$. Therefore, each move takes $O(c(r) + c(\bar{r})) + O(p) = O(n)$. Each area could be moved at most $O(p)$ times because an area never has a chance to be moved back to the same region where it comes from and there are in total *n* areas. Consequently, the overall time complexity of Indirect Flow Push is $O(n)O(np) = O(n^2p)$.

Remark 5.5. *Time complexity of Indirect Flow Push is* $O(n^2p)$.

5.2 Local Optimization

If a *feasible* partition is found in the Global Search phase, Local Optimization is applied to further improve the heterogeneity over the partition. Most regionalization algorithms [8, 9, 14, 34, 35, 53] include an optimization phase that improves the objective function by changing the membership of the border areas of the regions using heuristic searching strategies. Some regionalization algorithms [8, 9, 14] perform an extremely expensive exhaustive search of all possible reassignments of border areas just to pick only one reassignment step. This makes it hard to use on large datasets.

The Local Search in [53] has superior performance for the following reasons: (1) Local Search identifies movable areas instead of all the possible moves, (2) Local Search does not recalculate a new set of movable areas until the previous list has been exhausted. This makes Local Search efficient in improving the overall heterogeneity of the partition without performing extremely expensive computations that do not scale up for large data. An area *a* within region *r* is movable if: (i) *a* is on the margin of *r*, (ii) *a* is not an *articulation* area of *r*, (iii) *r* remains *complete* after the area *a* is removed. In each iteration, all the movable areas are put into a list. A random area is chosen to be moved to the neighbor region with the minimum heterogeneity. If the move decreases the heterogeneity over the current partition, then the move is accepted. Otherwise, the acceptance of the move is determined by the Boltzmann probability [31]. After a move is performed, all the areas belonging to the donor region and the receiver region are removed from the list. The heuristic does not identify the movable areas again unless the list of movable areas has been exhausted.

We further extend this heuristic to speed up the searching process and improve the optimization goal. First, for each selected movable unit, we reassign it to the neighboring region that results in the minimum heterogeneity increase instead of the neighboring region that has the minimum heterogeneity. Second, we parallelize Local Search by searching for movable areas of regions concurrently. Third, we adopt Tarjan algorithm [45] to find all the *articulation* areas that are not allowed to move. These improvements lead to 100x faster search in Local Optimization.

Complexity analysis. In Local Optimization, parallelly locating all the movable areas using Tarjan algorithm from each region takes $\sum_{i=1}^{p} O(\frac{c(r_i)}{T}) = O(\frac{n}{T})$ where T is the number of threads available in the parallel environment. After each move, areas from donor and receiver regions are removed from the list. So, on average, $\frac{p}{2}$ moves are performed. For each move, computing the heterogeneity change and selecting receiver region for reassigning area *a* in $a.NBR_R$ takes time $\sum_{i=1}^{|a.NBR_r|} c(r_i) = O(n)$. Consequently, each move attempt takes time $O(n) + O(\frac{n}{T})/\frac{p}{2} = O(n)$. So, the overall runtime of Local Optimization is $\alpha O(n)$, where α is the number of total move attempts and $O(n)$ is the cost for each move attempt.

Remark 5.6. *Time complexity of Local Optimization is* $O(\alpha n)$.

6 COMPLEXITY ANALYSIS

This section gives time and space complexity of GSLO. According to Remark 5.1, Remark 5.2, Remark 5.3, Remark 5.4, Remark 5.5, and Remark 5.6, the overall time complexity of GSLO is $O(n^2p + mp^2 + \alpha n)$, where α is the actual number of move attempts in the Local Optimization and m is the maximum number of iterations during Seed Identification. The value of α is mainly affected by the number of iterations in Local Optimization (*ILO*), i.e., the maximum number of non-improving moves allowed. Empirically, the value of α is 1-2 orders of magnitude of *ILO* parameter value (Table 3).

The space complexity of GSLO is $O(n)$. The input stores each area and its corresponding attributes, i.e., *extensive* attributes, *similarity* attributes, marginal coordinates and etc, which takes $O(n)$ storage. GSLO stores the neighbor areas of each input area. Since the spatial neighborhood is a planar graph, the number of neighbors is strictly less than six [50]. Consequently, storing the neighbors takes $6 * O(n) = O(n)$ space. On the region level, for each region *r*, we need to store $r.margin$ and $r.NBR_A$. Note that $\sum_{i=1}^{p} r_i.margin = O(n)$ and $\sum_{i=1}^{p} r_i.NBR_A = O(n)$. Consequently, the overall space complexity of GSLO is $O(n)$.

7 EXPERIMENTAL EVALUATION

In this section, we present extensive experimental evaluation to demonstrate the efficiency of GSLO. We use the following datasets: (1) TIGER shapefile dataset [10], and (2) Health, Income and Diversity dataset [19]. The TIGER dataset is a real dataset of the census tracts of individual states within the United States [10]. Each item in the TIGER dataset is a spatial polygon of census tract with multiple numerical attributes. The size of the dataset used in the experiments ranges from 2k to 40k spatial polygons, which is an order of magnitude larger than any dataset used in evaluating regionalization to the best of our knowledge. In our experiments, we consider the *ALAND*, which represents the current land area of a census tract, as the *extensive attribute*. Also, we consider *AWATER*, which represents the current water area of a census tract, as the *similarity attribute*.

The Health, Income and Diversity dataset [19] has 3k elements. Each item in this dataset is a county within the United States that is associated with multiple numerical attributes. We consider *cz_pop2000* as the *extensive attribute* and it represents the population of U.S. counties. We consider *ratio* as the *similarity attribute* and it represents each county's median income divided by the state's median income. For both datasets, all the island areas are removed. All experiments are based on Java 14 implementation using an Intel Xeon(R) server with CPU E5-2637 v4 (3.50 GHz) and 128GB RAM running Ubuntu 16.04. Table 3 summarizes the parameters used throughout the experimental evaluation. The bold values indicate the default setting for each parameter.

Our evaluation metrics are: (1) **heterogeneity**, (2) **runtime**, and (3) **effectiveness**, i.e., the probability of finding a *feasible* partition. The heterogeneity is calculated as the mean value from *feasible* partitions. If no *feasible* partition is generated among all the runs, then the heterogeneity is represented as inf. The number of runs in all experiments is 100 except for that the number of runs in the scalability test is 10 to avoid extremely long experimentation time.

We compare GSLO against four alternatives: (1) SKATER [3], (2) SKATERCON [4], (3) GS, and (4) Greedy. SKATER and SKATERCON are the state-of-the-art algorithms for the p-regions problem. GS is GSLO without Local Optimization. Greedy is a greedy baseline algorithm that proceeds as follows: (i) randomly select p seed areas, (ii) select the region with the least *extensive attribute* to grow by adding a neighboring area that results in the minimum heterogeneity increase, (iii) regions stop growing once the user-defined constraint has been satisfied or there are no neighboring areas, (iv) enclaves are assigned similar to GSLO.

Notice that SKATER and SKATERCON cannot directly solve PRUC because they do not consider user-defined constrains as described in Section 2. We modify SKATER and SKATERCON into SKATER* and SKATERCON*, respectively, to allow them to solve PRUC. **SKATER*** changes the tree-partitioning phase in SKATER as follows: The edge selection in SKATER* adopts the edge selection from SKATER. However, SKATER* enforces that the edge chosen to be split must be feasible, i.e., the subtrees produced by the split must exceed the threshold of the *extensive attribute*. SKATER* splits the tree in each iteration by choosing the edge cut that brings the greatest heterogeneity reduction among all feasible edge cuts. SKATERCON is modified to **SKATERCON*** by using SKATER* instead of SKATER and parallelizing the generation of spanning trees. The execution of SKATER* is also parallelized. Additionally, the subgraph with the largest *extensive attribute* is given the highest priority for partitioning in the last step of SKATERCON*. The runtime complexity for SKATER* is $O(n^3 p)$ and the runtime complexity for SKATERCON* is $O(\beta n^3 p)$ where n is the total number of spatial areas in the input, p is the predefined number of regions, and β is the number of random spanning trees in SKATERCON*.

7.1 GSLO Parameter Tuning

In this section, we experimentally identify the optimal values for the parameters of GSLO.

Number of Iterations in Seed Identification Figure 4 shows the heterogeneity and the runtime of GSLO under different number of iterations in Seed Identification (ISI) on the TIGER dataset. This figure shows that increasing the number of iterations results in improvement in the seed quality and the overall heterogeneity. Note that the seed quality is defined as the minimum area-area pair distance, which is discussed in Section 5.1.1 and only applicable for Seed Identification in GSLO. However, increasing the number of iterations increases the runtime of GSLO. We set the number of iterations to $1DS$ as it results in a balance between heterogeneity and overall runtime. Also, we compare our seeding strategy with random seeding and k-means++ [2] seeding, denoted as GSLO-random and GSLO-kmeans++, respectively. k-means++ seeding is reported as the lowest error seeding for k-means clustering [43]. Figure 4(c) shows that around $1DS$, GSLO seeding, random seeding, and k-means++ seeding achieve nearly the same runtime. Regarding heterogeneity, GSLO seeding slightly outperforms random seeding and k-means++ seeding. Figure 4(b) shows that the best heterogeneity GSLO obtained around $1DS$, i.e., the optimal setting as discussed above, achieves 11.1% better heterogeneity compared to random seeding and 4% better heterogeneity compared to k-means++ seeding, which demonstrates the superiority of our Seed Identification.

Iterations in Local Optimization Iterations in Local Optimization (ILO) refer to the maximum number of non-improving moves allowed in Local Optimization that is describe in 5.2. Figure 5 shows the heterogeneity and runtime under different ILO using the TIGER dataset. We see that the heterogeneity improves as the number of iterations in Local Optimization increases. However, the overall runtime also increases as ILO increases. We set the number of iterations in Local Optimization to $1DS$, i.e., the size of the dataset, as it achieves a good balance between runtime and heterogeneity in Local Optimization.

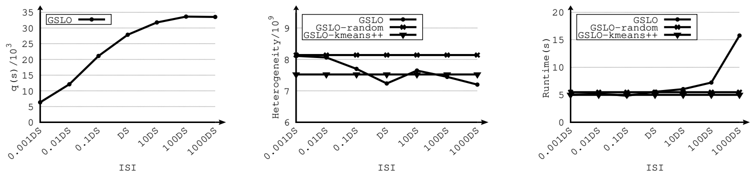
7.2 Performance Evaluation

This section analyzes the performance of GSLO under different parameter settings.

7.2.1 ***Time Breakdown Analysis.*** Figure 6 provides time breakdown analysis of GSLO under the TIGER dataset that shows the average runtime of each phase under different p. The figure shows that the runtime of Local Optimization dominates the runtime of GSLO and it decreases as p increases since larger p means less flexibility to reassign the border areas without violating the user-defined constraint. Seed Identification runtime increases slightly as p increases because more seed areas are involved. Region Growth runtime increases as p increases because there are more regions to grow. Enclaves Assignment runtime decreases as p increases because there are fewer enclaves to assign. Also, the runtime of Inter-region Update and Indirect Flow increases as p increases because a larger p results in a higher probability of generating *incomplete* regions and thus a higher probability of invoking both

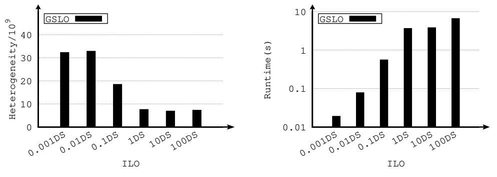
Table 3: Parameters and values

Parameter	Values
TIGER dataset size (DS)	**2k**, 3k(with island), 5k, 10k, 30k, 40k
HID dataset size (DS)	**3k**
p	5, **10**, 15, 20, 25, 30, 35, 40, 45, 50
Threshold (% of *extensive attribute*)	1%, **2%**, 3% , 4%, 5%, 6%, 7%, 8%, 9%, 10%
Num of iterations in Local Optimization (*ILO*)	0.001*DS*, 0.01*DS*, 0.1*DS*, **1*DS***, 10*DS*, 100*DS*
Num of iterations in Seed Identification (*ISI*)	0.001*DS*, 0.01*DS*, 0.1*DS*, **1*DS***, 10*DS*, 100*DS*, 1000*DS*



(a) Seed quality q(s) (b) Heterogeneity (c) Runtime

Figure 4: The effect of the number of iterations in Seed Identification under the TIGER dataset



(a) Heterogeneity (b) Runtime

Figure 5: The effect of the number of iterations in Local Optimization under the TIGER dataset

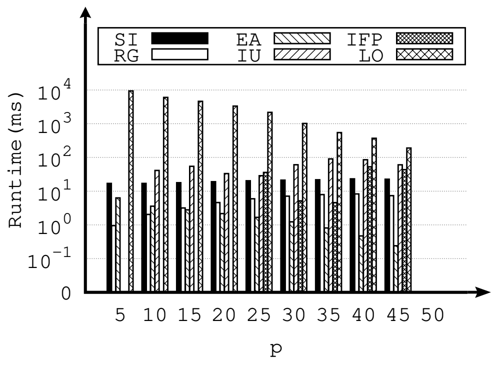
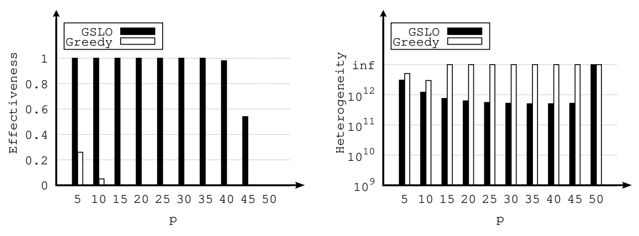


Figure 6: GSLO time breakdown (phase names abbreviated)



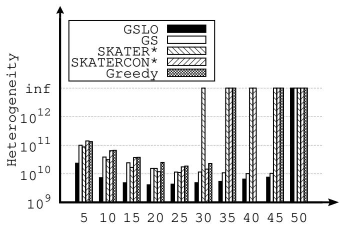
(a) Effectiveness (b) Heterogeneity

Figure 7: Support of islands

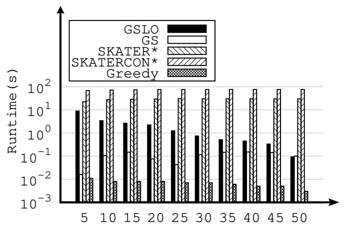
steps. Note that in this experiment the average runtime is computed from solved cases only. When $p = 50$, no feasible partition is found.

7.2.2 ***Exploring Island Dataset***. In this section, we experimentally explore the efficiency of GSLO over a dataset containing islands. We use a dataset that consists of two connected components of size 3k, and 0.2k, respectively. SKATER* and SKATERCON* do not support islands because the input for both must be a connected spatial neighborhood graph. Thus, we compare GSLO with Greedy. Figure 7 shows that GSLO consistently achieves better heterogeneity and effectiveness than Greedy in all cases. Notice that when $p > 10$, Greedy does not find a feasible partition at all, whereas GSLO finds the feasible partition with high probability in all solvable cases. GSLO's high effectiveness results from all phases of GSLO that take the *extensive* attribute into consideration.

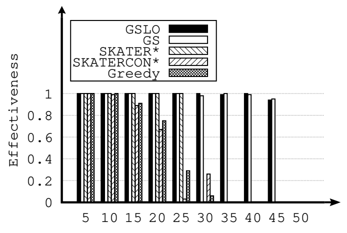
7.2.3 ***The effect of the number of regions p***. Figure 8a, Figure 8b, and Figure 8c show the performance of all alternatives under different p using the TIGER dataset. Note that the effectiveness of SKATER* is either 0 or 1 because SKATER* is deterministic. The result shows that GSLO consistently achieves the best heterogeneity and effectiveness. For the TIGER dataset, GSLO achieves up to 5.22× improvement in heterogeneity compared to GS, which demonstrates the efficiency of Local Optimization to further optimize the heterogeneity. GSLO achieves up to 9× improvement in heterogeneity compared to Greedy. Although the runtime in Greedy is the smallest among all, the worst effectiveness and heterogeneity make it impractical to use. Greedy has bad effectiveness because it does not balance the *extensive* attribute across different regions, and has bad heterogeneity because it makes local greedy decisions when growing regions, which leads to suboptimal solutions. GSLO achieves up to 4.3× improvement in heterogeneity compared to SKATER* and up to 8.8× improvement compared to SKATERCON*. Moreover, GSLO is up to 90.6× faster than SKATER* and up to 229.7× faster than SKATERCON*. Figure 8d, Figure 8e, and Figure 8f show that using the HID dataset, GSLO achieves up to 2.24× better heterogeneity compared to GS, and up to 31.6% improvement in heterogeneity compared to Greedy. GSLO achieves up to 21.5% improvement in heterogeneity compared to SKATER* and up to 52.3% improvement compared to SKATERCON*. With respect to



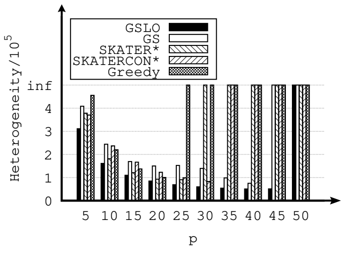
(a) Heterogeneity - TIGER



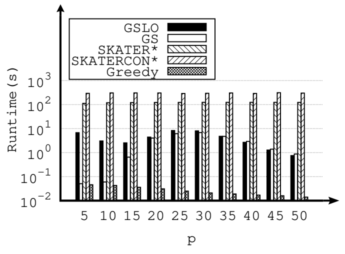
(b) Runtime - TIGER



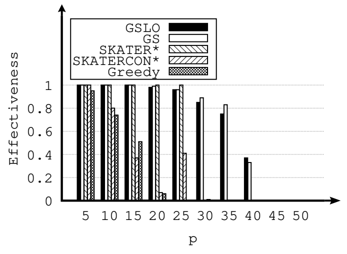
(c) Effectiveness - TIGER



(d) Heterogeneity - HID



(e) Runtime - HID



(f) Effectiveness - HID

Figure 8: The effect of p under the TIGER and HID datasets

runtime, GSLO is up to 97.1× faster than SKATER* and up to 244.7× faster than SKATERCON*. The percentage of heterogeneity reduction in the TIGER dataset of GSLO compared to the other baseline algorithms is much greater than in the HID dataset. This is because the *similarity attribute* in the TIGER dataset has a greater range and variance, thus different partitions constructed from the TIGER dataset have greater difference regarding heterogeneity compared to the HID dataset where the *similarity attribute* has closer values. Due to the fact that Greedy is inefficient and GS is part of GSLO, in the following experiments we will only compare GSLO with SKATER* and SKATERCON*.

Notice that the runtime of SKATER* and SKATERCON* increases as p increases. Furthermore, SKATERCON* has a higher runtime than SKATER* as SKATER* is a phase of SKATERCON*. However, GSLO requires less runtime as p increases. The reason is that, as p increases, the number of areas that can move between regions in Local Optimization is smaller. Hence, the number of possible moves is also smaller. This results in less runtime. Notice that GSLO is able to early detect that there is no *feasible* solution to the input problem up to 302.6× faster than SKATER* and SKATERCON*. The reason is that GSLO is able to make an early decision about the feasibility of the input problem in the Global Search phase.

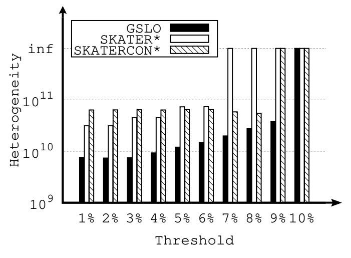
With respect to runtime, GSLO outperforms SKATER* and SKATERCON*, because GSLO is a seeding-based algorithm that incrementally grows regions around seed areas. However, SKATER* and SKATERCON* require finding successive expensive edge cuts on the input graph. From a theoretical perspective, the time complexity of SKATER* and SKATERCON* is cubic in n while the time complexity of GSLO is quadratic in n. This gives GSLO a consistent edge over SKATER* and SKATERCON*. GSLO achieves superior heterogeneity due to the Local Optimization step that reassigns areas to regions to improve the overall heterogeneity, whereas in SKATER* and SKATERCON*, once a partition is generated, no adjustment is made to further optimize the heterogeneity. GSLO has higher effectiveness due to Inter-region Update and Indirect Flow Push phases that produce *complete* regions. SKATER* and SKATERCON* do not have these abilities.

7.2.4 *The effect of varying the threshold.*

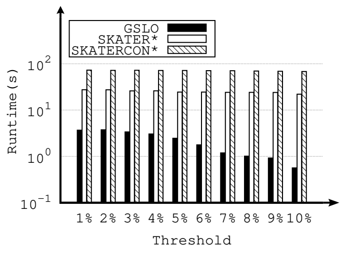
Figure 9 shows the heterogeneity, runtime, and effectiveness of GSLO, SKATER*, and SKATERCON* under different threshold values in the TIGER and HID datasets. Figure 9a, Figure 9b, and Figure 9c show that, under the TIGER dataset, GSLO achieves up to 6.1× better heterogeneity than SKATER* and up to 8.6× better heterogeneity than SKATERCON*. Additionally, GSLO is up to 26× faster than SKATER* and up to 74.5× faster than SKATERCON*. Figure 9d, Figure 9e, and Figure 9f show that, under the HID dataset, GSLO achieves up to 12.8% better heterogeneity than SKATER* and up to 48.1% better heterogeneity than SKATERCON*. GSLO is up to 37.5× faster than SKATER* and up to 95.9× faster than SKATERCON* under the HID dataset. GSLO achieves the best effectiveness in both datasets. The reason behind the good performance is similar to the one explained in Section 7.2.3.

7.2.5 *Using GSLO to Solve the p-regions problem.*

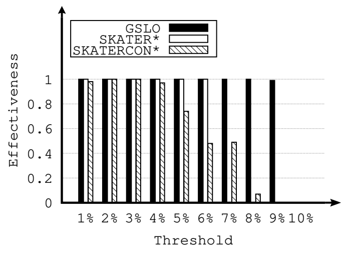
When the threshold value is set to 0, PRUC resembles the basic p-regions problem [15]. In this experiment, we compare GSLO to SKATER and SKATERCON when solving the p-regions problem. Figure 10 illustrates that GSLO achieves better results than both SKATER and SKATERCON for both heterogeneity and runtime. Under the TIGER dataset, GSLO achieves up to 4.1× better heterogeneity than SKATER and 8.7× better heterogeneity than SKATERCON. In addition, GSLO is up to 31.2× faster than SKATER and up to 73.2× faster than SKATERCON. Under the HID dataset, GSLO achieves up to 22% better heterogeneity than SKATER and up to 23.3% better heterogeneity than SKATERCON. Moreover, GSLO is up to 180.9× faster than SKATER and up to 425× faster than SKATERCON.



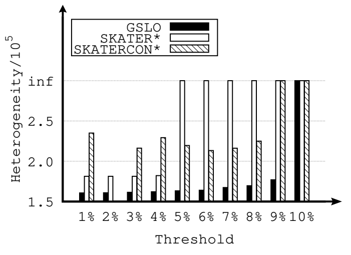
(a) Heterogeneity - TIGER



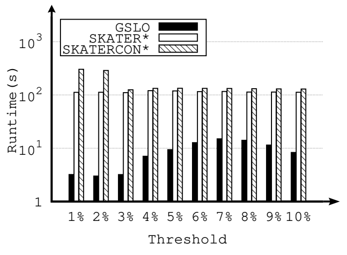
(b) Runtime - TIGER



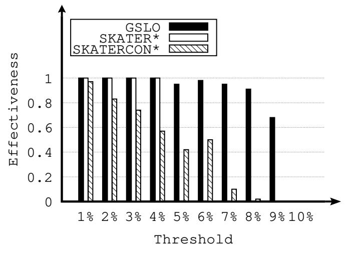
(c) Effectiveness - TIGER



(d) Heterogeneity - HID

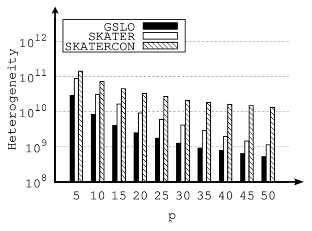


(e) Runtime - HID

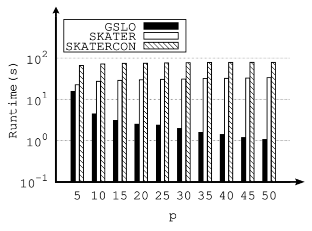


(f) Effectiveness - HID

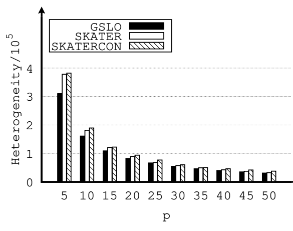
Figure 9: The effect of threshold under the TIGER and HID datasets



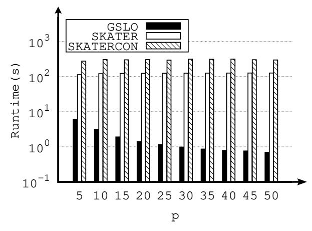
(a) Heterogeneity - TIGER



(b) Runtime - TIGER



(c) Heterogeneity - HID

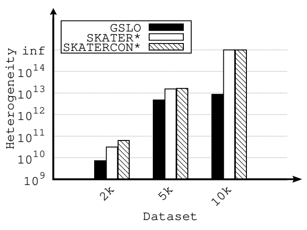


(d) Runtime - HID

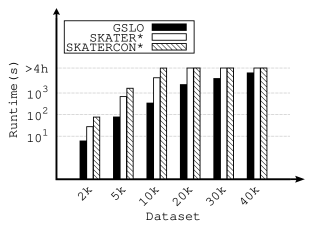
Figure 10: Solving p-regions problem under the TIGER and HID datasets

7.2.6 *The scalability of GSLO.*

Figure 11 demonstrates the scalability of GSLO compared to SKATER* and SKATERCON* on the TIGER dataset of different sizes. Within a predefined time limit, i.e., 4 hours, GSLO can handle up to 40k dataset while SKATER* and SKATERCON* can only handle up to 10k. Furthermore,GSLO achieves up to 5× better heterogeneity than SKATER* and SKATERCON*. This experiment shows that GSLO can handle up to 4× larger datasets than SKATER* and SKATERCON*.



(a) Heterogeneity



(b) Runtime(s)

Figure 11: Scalability test under the TIGER dataset

8 CONCLUSION

In this paper, we introduce PRUC, a generalized version of the p-regions problem that accounts for user-defined constraints. We develop an efficient parallel stochastic solution to PRUC which is divided into Global Search and Local Optimization. Experimental results show that GSLO is up to more than 100× faster and achieves up to 6× better heterogeneity than the state-of-the-art algorithms. In addition, GSLO solves the original p-regions problem with up to 4× better heterogeneity than existing algorithms. With respect to future work, we plan to use GSLO to solve other spatial regionalization problems, e.g., p-compact region problem [35], school redistricting problem [8, 9], Node-attributed Spatial Graph Partitioning [6], and MAX-P regions problem [14]. Also, we plan to investigate the support of incremental changes to the properties of input areas and multiple user-defined constraints.

ACKNOWLEDGMENTS

This work is partially supported by the National Science Foundation, USA, under grants IIS-1849971, SES-1831615, and CNS-2031418.

REFERENCES

[1] J. Aldstadt. Spatial Clustering. In *Handbook of applied spatial analysis*, pages 279–300. Springer, 2010.

[2] D. Arthur and S. Vassilvitskii. k-means++: The Advantages of Careful Seeding. Technical report, Stanford, 2006.

[3] R. M. Assunção, M. C. Neves, G. Câmara, and C. Da Costa Freitas. Efficient Regionalization Techniques for Socio-economic Geographical Units Using Minimum Spanning Trees. *International Journal of Geographical Information Science, IJGIS*, 20(7):797–811, 2006.

[4] O. Aydin, M. V. Janikas, R. Assunção, and T.-H. Lee. SKATER-CON: Unsupervised Regionalization via Stochastic Tree Partitioning Within a Consensus Framework Using Random Spanning Trees. In *Proceedings of the ACM SIGSPATIAL International Workshop on AI for Geographic Knowledge Discovery, ACM GeoAI*, pages 33–42, 2018.

[5] O. Aydin, M. V. Janikas, R. M. Assunção, and T. H. Lee. A Quantitative Comparison of Regionalization Methods. *International Journal of Geographical Information Science, IJGIS*, 35(11):2287–2315, 2021.

[6] D. Bereznyi, A. Qutbuddin, Y. Her, and K. Yang. Node-attributed Spatial Graph Partitioning. In *Proceedings of the ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, ACM GIS*, pages 58–67, 2020.

[7] L. Bertolini and W. Salet. Planning Concepts for Cities in Transition: Regionalization of Urbanity in the Amsterdam Structure Plan. *Planning Theory and Practice*, 4(2):131–146, 2003.

[8] S. Biswas, F. Chen, Z. Chen, C. T. Lu, and N. Ramakrishnan. Incorporating Domain Knowledge into Memetic Algorithms for Solving Spatial Optimization Problems. In *Proceedings of the ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, ACM GIS*, pages 25–35, 2020.

[9] S. Biswas, F. Chen, Z. Chen, A. Sistrunk, N. Self, C. T. Lu, and N. Ramakrishnan. REGAL: A Regionalization Framework for School Boundaries. In *Proceedings of the ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, ACM GIS*, pages 544–547, 2019.

[10] U. C. Bureau. TIGER/Line Shapefile, 2016, Series Information for the Current Census Tract State-based Shapefile, 2021. https://catalog.data.gov/dataset/tiger-line-shapefile-2016-series-information-for-the-current-census-tract-state-based-shapefile.

[11] P. S. Cowpertwait. A Regionalization Method Based on a Cluster Probability Model. *Water Resources Research*, 47(11), 2011.

[12] I. I. Cplex. V12. 1: User's Manual for CPLEX. *International Business Machines Corporation*, 46(53):157, 2009.

[13] F. Csillag, S. Kabos, and T. K. Remmel. A Spatial Clustering Perspective on Autocorrelation and Regionalization. *Environmental and ecological statistics*, 15(4):385–401, 2008.

[14] J. C. Duque, L. Anselin, and S. J. Rey. The Max-P-Regions Problem. *Journal of Regional Science, JRS*, 52(3):397–419, 2012.

[15] J. C. Duque, R. L. Church, and R. S. Middleton. The p-Regions Problem. *Geographical Analysis*, 43(1):104–126, 2011.

[16] J. C. Duque, R. Ramos, and J. Suriñach. Supervised Regionalization Methods: A Survey. *International Regional Science Review, IRSR*, 30(3):195–220, 2007.

[17] J. C. Duque, M. C. Vélez-Gallego, and L. C. Echeverri. On the Performance of the Subtour Elimination Constraints Approach for the p-Regions Problem: A Computational Study. *Geographical Analysis*, 50(1):32–52, 2018.

[18] M. M. Fischer. Regional Taxonomy: A Comparison of Some Hierarchic and Non-hierarchic Strategies. *Regional Science and Urban Economics*, 10(4):503–537, 1980.

[19] 2000 Health, Income and Diversity Shapefile, 2021. https://geodacenter.github.io.

[20] F. Glover. Heuristics for Integer Programming Using Surrogate Constraints. *Decision Science*, 8(1):156–166, 1977.

[21] D. Guo. Regionalization with Dynamically Constrained Agglomerative Clustering and Partitioning (REDCAP). *International Journal of Geographical Information Science, IJGIS*, 22(7):801–823, 2008.

[22] J. Harff and J. C. Davis. Regionalization in Geology by Multivariate Classification. *Mathematical Geology*, 22(5):573–588, 1990.

[23] X. He and B. Wei. A Hybrid Heuristic Algorithm for School District Division. *The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 42:1113–1120, 2020.

[24] J. Hurley. Regionalization and the Allocation of Healthcare Resources to Meet Population Health Needs. *HealthcarePapers*, 5:34–39, 2004.

[25] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data.* Prentice-Hall, Inc., 1988.

[26] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal of Scientific Computing, SISC*, 20(1):359–392, 1998.

[27] H. Kim, Y. Chun, and K. Kim. Delimitation of Functional Regions Using a p-Regions Problem Approach. *International Regional Science Review, IRSR*, 38(3):235–263, 2015.

[28] K. Kim, Y. Chun, and H. Kim. p-Functional Clusters Location Problem for Detecting Spatial Clusters with Covering Approach. *Geographical Analysis*, 49(1):101–121, 2017.

[29] K. Kim, Y. Chun, and H. Kim. A Robust Heuristic Approach for Regionalization Problems. In *GeoComputational Analysis and Modeling of Regional Systems*, pages 305–324. Springer, 2018.

[30] K. Kim, D. J. Dean, H. Kim, and Y. Chun. Spatial Optimization for Regionalization Problems with Spatial Interaction: a Heuristic Approach. *International Journal of Geographical Information Science, IJGIS*, 30(3):451–473, 2016.

[31] G. Kirkpatrick and M. Vechi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[32] P. M. Lankford. Regionalization: Theory and Alternative Algorithms. *Geographical Analysis*, 1(2):196–212, 1969.

[33] J. Laura, W. Li, S. J. Rey, and L. Anselin. Parallelization of a Regionalization Heuristic in Distributed Computing Platforms – a Case Study of Parallel-p-compact-regions Problem. *International Journal of Geographical Information Science, IJGIS*, 29(4):536–555, 2015.

[34] W. Li, R. L. Church, and M. F. Goodchild. An Extendable Heuristic Framework to Solve the p-compact-regions Problem for Urban Economic Modeling. *Computers, Environment and Urban Systems*, 43:1–13, 2014.

[35] W. Li, R. L. Church, and M. F. Goodchild. The p-compact-regions Problem. *Geographical Analysis*, 46(3):250–273, 2014.

[36] A. Ligmann-Zielinska. Spatial Optimization. *International Encyclopedia of Geography: People, the Earth, Environment and Technology: People, the Earth, Environment and Technology*, pages 1–6, 2016.

[37] J. MacQueen et al. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297, 1967.

[38] R. T. Marler and J. S. Arora. Survey of Multi-objective Optimization Methods for Engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004.

[39] L. Miranda, J. Viterbo Filho, and F. C. Bernardini. Regk-means: A Clustering Algorithm Using Spatial Contiguity Constraints For Regionalization Problems. In *Brazilian Conference on Intelligent Systems (BRACIS)*, pages 31–36, 2017.

[40] A. T. Murray and T. K. Shyy. Integrating Attribute and Space Characteristics in Choropleth Display and Spatial Data Mining. *International Journal of Geographical Information Science, IJGIS*, 14(7):649–667, 2000.

[41] J. Niesterowicz, T. Stepinski, and J. Jasiewicz. Unsupervised Regionalization of the United States into Landscape Pattern Types. *International Journal of Geographical Information Science, IJGIS*, 30(7):1450–1468, 2016.

[42] S. Openshaw. A Geographical Solution to Scale and Aggregation Problems in Region-Building, Partitioning and Spatial Modelling. *Transactions of the Institute of British Geographers*, pages 459–472, 1977.

[43] J. Ortiz-Bejar, E. S. Tellez, M. Graff, J. Ortiz-Bejar, J. C. Jacobo, and A. Zamora-Mendez. Performance Analysis of k-means Seeding Algorithms. In *2019 IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC)*, pages 1–6, 2019.

[44] M. M. Rahman. Regionalization of Urbanization and Spatial Development: Planning Regions in Bangladesh. *The Journal of Geo-Environment*, 4:31–46, 2004.

[45] T. Robert. Depth-first Search and Linear Graph Algorithms. *SIAM Journal on Computing, SICOMP*, 1(2):146–160, 1972.

[46] B. She, J. C. Duque, and X. Ye. The Network-max-P-regions Model. *International Journal of Geographical Information Science, IJGIS*, 31(5):962–981, 2017.

[47] A. Sheshasaayee and D. Sridevi. A Combined System for Regionalization in Spatial Data Mining Based on Fuzzy C-Means Algorithm with Gravitational Search Algorithm. In *Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications*, pages 517–524, 2017.

[48] V. Sindhu. Exploring Parallel Efficiency and Synergy for Max-P Region Problem Using Python. Master's thesis, Georgia State University, 2018.

[49] D. Tong and A. T. Murray. Spatial Optimization in Geography. *Annals of the Association of American Geographers*, 102(6):1290–1309, 2012.

[50] R. J. Trudeau. *Introduction to graph theory.* Courier Corporation, 2013.

[51] R. Webster and P. A. Burrough. Computer-Based Soil Mapping of Small Areas From Sample Data: Ii. Classification Smoothing. *European Journal of Soil Science, EJSS*, 23(2):222–234, 1972.

[52] R. Wei, S. Rey, and T. H. Grubesic. A Probabilistic Approach to Address Data Uncertainty in Regionalization. *Geographical Analysis*, 0:1–22, 2021.

[53] R. Wei, S. Rey, and E. Knaap. Efficient Regionalization for Spatially Explicit Neighborhood Delineation. *International Journal of Geographical Information Science, IJGIS*, 35(1):135–151, 2021.

[54] D. White, M. Richman, and B. Yarnal. Climate Regionalization and Rotation of Principal Components. *International Journal of Climatology*, 11(1):1–25, 1991.

[55] X. Yang, J. Magnusson, and C.-Y. Xu. Transferability of Regionalization Methods under Changing Climate. *Journal of Hydrology*, 568(March 2018):67–81, 2019.

[56] X. Ye, B. She, and S. Benya. Exploring Regionalization in the Network Urban Space. *Journal of Geovisualization and Spatial Analysis, JGSA*, 2(1):1–11, 2018.

[57] B. Zhang, M. Hsu, and U. Dayal. K-Harmonic Means - A Spatial Clustering Algorithm with Boosting. In *International Workshop on Temporal, Spatial, and Spatio-Temporal Data Mining*, pages 31–45, 2000.

[58] Y. Zhou, H. Cheng, and J. X. Yu. Graph Clustering Based on Structural/Attribute Similarities. *Proceedings of the VLDB Endowment, PVLDB*, 2(1):718–729, 2009.