A Spatially-Explicit Evolutionary Algorithm for the Spatial Partitioning Problem

Yan Y. Liu*, Wendy K. Tam Cho

Abstract

Spatial optimization is an important and prolific field of interdisciplinary research. Spatial optimization methods seek optimal allocation or arrangement of spatial units under constraints such as distance, adjacency, contiguity, and pattern. Evolutionary Algorithms (EAs) are well-known optimization heuristics. However, classic EAs, based on binary problem encoding and bit-operation-based offspring operators, are spatially unaware and do not capture topological and geometric relationships. Unsurprisingly when spatial characteristics are not explicitly considered in the design of EA operators, an EA becomes ineffective because satisfying spatial constraints is computationally expensive. These issues are exacerbated in large spatial partitioning optimization problems. We design and develop novel spatially explicit EA recombination operators, inspired by the path relinking and ejection chain heuristic strategies, that implement crossover and mutation using intelligently guided strategies in a spatially constrained decision space. We establish a generalized spatial EA approach that slots well into the foundational theory of evolutionary algorithms for spatial optimization. We demonstrate improved solution quality and computational performance with spatial partitioning applications.

Keywords:
Combinatorial Optimization, Evolutionary Algorithm, Spatial Optimization, Heuristics, Parallel Computing

*Yan Y. Liu is Senior Research Programmer in the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign. (2019). 1205 West Clark Street, Urbana, IL 61801.

Email addresses: yanliu@illinois.edu (Yan Y. Liu), wendycho@illinois.edu (Wendy K. Tam Cho)

1Wendy K. Tam Cho is Professor in the Departments of Political Science, Statistics, Mathematics, Asian American Studies, and the College of Law, Senior Research Scientist at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign. 420 David Kinley Hall, 1407 W. Gregory St., Urbana, IL 61801
1 INTRODUCTION

1. Introduction

Spatial optimization problems seek the optimal allocation or arrangement of spatial units according to some objective or measure of goodness [60]. The spatial properties embedded in the objectives and constraints may include, but are not limited to, distance, adjacency, contiguity, containment, intersection, shape, partition, and pattern [52]. Spatial optimization has been studied in a wide variety of contexts, dating back to the 1800s with explorations of efficient land use activity [53]. More recent research includes studies of location models [19, 41, 50, 66], coverage problems [11, 37, 38], zoning and spatial aggregation [62, 54], and spatial characteristics in land cover classification [28, 10, 15]. Across an array of different areas, research in spatial optimization is lively and generates significant technical and substantive interest.

Many problems in spatial optimization are computationally challenging and computationally intractable (i.e., NP-Hard [12, 27]). Examples include the regionalization problem that aims to group spatial units into a small set of regions that satisfy optimization objectives (e.g., $p$-region problem [9] and spatial clustering-based regionalization [17, 54]), the spatial zoning problem that seeks to partition spatial units into a set of contiguous zones [31], the maximal covering location problem where the coverage of a set of facilities is maximized [11], the spatial allocation problem that seeks to allocate specific activities to particular spatial units [21], and the $p$-hub location problem that locates $p$ transportation hubs and allocates demand to specific hubs to minimize total transportation costs [42]. As the size and number of dimensions of spatial data have increased, the associated spatial optimization problem becomes increasingly intricate, characterized by massive decision spaces that eclipse the capabilities of exact algorithms to identify optimal solutions. Given these trends, continued success in deploying future spatial optimization applications will require innovation in spatial optimization methods.

One path for solving difficult optimization problems is via the heuristic method of Evolutionary Algorithms (EAs) that are inspired by natural selection [22]. EAs mimic an evolutionary process that encodes a problem as a chromosome and generates an initial population of individual solutions with random chromosomes. Through randomized EA operators (e.g., selection, crossover, mutation, and replacement), the population evolves based on a “survival of the fittest” rule [58, 16]. This route has been successful for many large-scale optimization problems. However, difficult spatial optimization problems pose interesting and non-trivial challenges, necessitating novel strategies for incorporating spatial elements into EA operators [23].
Xiao [60] categorized problems in spatial optimization as either assignment or partition problems, with or without spatial constraints. Among the four categories, we investigate partition problems with spatial constraints (hereafter spatial partitioning), which enforce strong spatial constraints such as contiguity. Here, changing the partition of a spatial unit necessarily involves reassigning neighboring units in order to satisfying the contiguity constraint. A particular search complication that arises from strong spatial constraints in these partitioning problems is that considerations of spatially defined locality and adjacency have a direct and explicit effect on solution feasibility. Non-spatial set partitioning problems involve the unconstrained combinatorial construction of sets. In contrast, in spatial partitioning, every spatial constraint redefines solution feasibility because of the system of spatial dependency among the decision variables. One consequence is that the decision space becomes “patchy,” with feasible and infeasible solution areas juxtaposed. In addition, as the problem size grows, the disparity between the number of feasible and infeasible partitions grows, with infeasible partitions quickly eclipsing the feasible ones.

Although randomized components powerfully enable EA performance through classic recombination operators, such operations that generate new solutions by exchanging bits on linearly-encoded chromosome are ineffective in traversing patchy spatially-constrained decision spaces because they do not capture or preserve the variable interdependencies introduced by the spatial constraints [30]. This issue becomes even more poignant for large applications that have increasingly greater proportions of infeasible solutions. In addition, other conventional EA strategies such as designing a penalty function for spatial constraint handling are also unlikely to be effective for the same basic reason that they expend enormous computational effort wading into infeasible regions rather than avoiding them all together.

While it is possible, it would be highly unusual that a spatial partitioning problem could be solved by conventional EAs using only linear recombination operators [23]. A possible route is to use the classic mutation and crossover operators to generate new solutions, and then to subsequently repair the broken spatial relationships. While this strategy may work on small problem instances, it becomes computationally prohibitive as the problem size increases and thus turns an EA process into essentially a random search for feasible solutions rather than a heuristic that improves solution quality. We argue that EA recombination operators, including crossover and mutation, must not only be spatially aware, but also spatially explicit in exploring the spatially constrained decision space. Such intelligently guided constructive methods are more promising than repair methods,
particularly for large problem sizes where a large number of infeasible solution regions are scattered throughout an enormous decision space.

We propose a novel spatial evolutionary algorithm that implements spatially explicit crossover and mutation operations via adaptation of the path relinking and ejection chain heuristics. Our proposed EA preserves solution feasibility and thus reduces wasted computation exploring infeasible solution regions. Consequently, our EA is able to improve solution quality on large problem instances and maintain desirable computational performance. Since we incorporate spatial heuristics into the EA operators, the algorithm falls into the class of memetic algorithms [36] or hybrid EAs [2]. We develop our approach based on a general definition of the spatial partitioning problem and then solve a specific spatial zoning application to demonstrate the algorithmic and computational performance of our approach and its sequential and parallel implementations.

2. Literature Review

The EA approach to spatial optimization has been an active area of research in geography for decades. Although EAs are recognized as a general heuristic for problem-solving and have been successfully employed in various spatial optimization problems [5, 1, 35, 47, 46], how to align EA encoding and operations with an application’s inherent data idiosyncrasies [20] (including spatial relationships) has remained a challenge for achieving desirable performance. For example, Hosage and Goodchild [23] found that the computational efficiency of EAs lags other algorithms for solving the $p$-median problem, commenting that “one would expect somewhat different performance if the coding order preserved spatial relationships than if it were essentially random.”

Adapting heuristics to function efficiently with spatial data has been a consistent effort in geographic analysis research. Openshaw [43] first introduced a contiguity-preserving method that generates new solutions by morphing only on zone boundaries. He used this algorithm to study the scale and aggregation problem in geospatial partitioning applications. Based on this method, Openshaw and Rao [44] developed tabu and simulated annealing heuristic methods for census zoning. Tong, Murray and Xiao [51] implemented a spatial genetic algorithm for the maximum coverage problem with a specialized crossover procedure that avoids spatial clusters and promotes spatial dispersion. Xiao, Bennett and Armstrong [61] designed a spatial multi-objective EA for the site search problem by modifying both the mutation and crossover operators to take
spatial considerations into account. Location-based operators were devised to move a subset of units to a randomly identified location while morph-based operators identified moveable units between two spatially correlated (e.g., contiguous) sites and “morphed” or exchanged their site assignment. Xiao [60], when categorizing problems in spatial optimization, conceptualized baseline algorithms for EA operators using a graph theoretic approach. Wei and Murray [56] designed a multi-objective EA for optimizing spatial uncertainty by treating uncertainty as an objective and constraint. Wu, Murray and Xiao [59] designed a multi-objective EA for optimizing spatial contiguity in a reserve network design by incorporating relative contiguity. Wang [55] developed a spatial EA for large ground water modeling where spatial characteristics are encoded in an adaptive quad-tree data structure to allow the branches of the tree to be grafted from one parent solution to another in the crossover operation while splitting, merging, and alternation are applied on the tree in the mutation operator.

Many of these spatial optimization heuristics contend with a spatial contiguity constraint [60, 25]. They have employed one of three main strategies. First, solutions can be generated freely without spatial considerations, and then non-contiguous solutions are later discarded [34, 3]. While this strategy is realizable, it is impractical for large applications where non-contiguous solutions far outnumber contiguous ones. Second, one can generate solutions freely, but then repair the broken spatial relationships. Again, while this strategy may work on small problem instances, it becomes computationally prohibitive as the problem size increases since repair operators are computationally costly. Lastly, contiguity can be enforced by creating only contiguous aggregations. This approach is the most promising and has been explored by a number of scholars. Cova and Church [7] maintain contiguity via a series of math inequalities using 1D and 2D hamming distances to represent the principle that no spatial unit can be chosen before a unit closer to a reference unit is chosen. Cleverly, the hamming distance is defined through a directional spatial relation between a cell and its neighbors toward the reference site. This representation permits a mixed integer programming (MIP) system to make implicit spatial neighborhood moves via branching that is based on distance. Using a similar representation, Williams [57] addressed the exact contiguity constraint for a vector-based problem formulation. Shirabe [48] then optimized the contiguity representation for integer programming in a way that reduces the search cost for linear solvers. Shirabe [49] and Murray, Grubesic and Wei [40] conceptualized unit connectivity
as a network and defined contiguity to be a classic network flow problem, in which a connected
graph allows the selection of paths from source to sink.

While a mathematical programming solution to contiguity-constrained spatial partitioning
needs to address how to best represent and incorporate the contiguity constraint in a solver sys-
tem, heuristic solutions need to consider contiguity explicitly in neighborhood search functions. In
this quest, one could introduce a penalty function as a general solution, though as Runarsson and
Yao [45] point out, a penalty function is both hard to define and to make efficient. Instead, more
intelligent heuristic search operators are often considered [65]. Existing efforts devise specialized
routines to satisfy contiguity. For example, in their spatial scanner that searches for clusters in a
map, Izakian and Pedrycz [24] designed a particle swarm optimization algorithm that checked the
contiguity of each cluster. King et al. [26] designed a specialized geo-graph data structure. While
their formulation improved a single contiguity check, overall improvement was limited because
of the significant computational performance penalty imposed by the need to perform this check
at every EA iteration. Indeed, Liu, Cho and Wang [31] improved efficiency by two orders of mag-
nitude by forcing contiguity in initial solutions and only requiring contiguity checking in solution
updates.

To be sure, our goal is not simply to incorporate spatially aware operators, but to do so in a
computationally efficient manner that will scale with problem size. In this pursuit, scalability is
most promisingly fostered by avoiding spatially infeasible solutions with operators that are care-
fully designed to preserve spatial constraints. Here, we must be cognizant of the inherent trade-
offs between operator sophistication and algorithm convergence. For instance, both Dahlin and
Sallnäs [8] and Lockwood and Moore [32] incorporated a penalty parameter to guide the solution
search toward feasible regions. However, Murray and Church [39] pointed out that this design
resulted in shifting computational effort toward identifying feasible solutions at the expense of
improving the search by identifying higher quality solutions.

3. Methodology

Our main observation is that a failure to recognize and incorporate spatial relationships and
constraints in EA crossover and mutation significantly degrades the performance of EA operators.
In this section, we start with a MIP formulation of a generalized form of the spatial partitioning
problem constrained by contiguity. We highlight how conventional EA crossover and mutation
operators create computational issues when they do not incorporate spatial information. We then propose an avenue to overcome these limitations by devising new EA operators that incorporate spatially explicit operations.

### The Specification of the Spatial Partitioning Problem

#### Problem Statement

Given \( n \) spatial units, partition the \( n \) units into \( k \) disjoint zones that satisfy a specified set of spatial and non-spatial constraints, in which the contiguity constraint requires all \( k \) zones and all of the units in each of the \( k \) zones to be connected.

#### A Mathematical Formulation

\( I: \) set of spatial units; 
\( A: \) set of adjacent unit pairs; 
\( K: \) set of zones; 
\( n_k: \) number of units in zone \( k; \)

\[ x = \{ x_{ik} \}: \quad x_{ik} = \begin{cases} 1 & \text{if unit } i \text{ is assigned to zone } k \\ 0 & \text{otherwise} \end{cases} \]

\( y_{ijk}: \) flow from unit \( i \) to unit \( j \) for zone \( k \)

\[ w_{ik} = \begin{cases} 1 & \text{if unit } i \text{ is the hub of zone } k \\ 0 & \text{otherwise} \end{cases} \]

**Objective** \[ \min f(x) \]

**Constraints**

\[ \sum_{j \mid (i,j) \in A} y_{ijk} - \sum_{j \mid (j,i) \in A} y_{jik} = n_k w_{ik} - x_{ik} \quad \forall k \in K, \forall i \in I \] (1)

\[ \sum_{j \mid (j,i) \in A} y_{jik} \leq (n_k - 1)x_{ik} \quad \forall k \in K, \forall i \in I \] (2)

\[ \sum_{k \in K} x_{ik} = 1 \quad \forall i \in I \] (3)

\[ \sum_{i \in I} w_{ik} = 1 \quad \forall k \in K \] (4)

\[ ax \leq b \] (5)

\[ x_{ik}, w_{ik} \in \{0, 1\} \quad \forall k \in K, \forall i \in I \] (6)

\[ y_{ijk} \geq 0 \quad \forall k \in K, \forall (i,j) \in A \] (7)

The above formulation is a variation of Shirabe [49], which defines the contiguity for each partition as a network flow from all of the spatial units in each zone to their zone hub that receives
the flow. The objective function is a weighted sum of spatial and non-spatial objectives. For instance, if the partitioning problem considers weight balance among zones, the objective function can minimize the weight difference. Constraint (1) requires that the difference of flow into and out of a unit \(i\) must be \((n_k - 1)\). This means that if unit \(i\) is the hub of zone \(k\), the flow traverses each unit in the zone exactly once. Otherwise, this constraint has no effect. Constraint (2) ensures that no unit is visited twice. Constraint (3) guarantees that each unit is a member of one and only one zone. Constraint (4) guarantees that each zone has only one hub. These four constraints together ensure that all of the units are partitioned into exactly \(k\) contiguous zones. Constraint (5) denotes all other non-spatial constraints. While unit assignment is discrete (Constraint (6)), the flow is formulated as a continuous variable (Constraint (7)). We can also see that this problem is computationally intractable since Constraints (1) and (2) generate a number of inequalities that increases exponentially with the number of units.

It is worth clarifying that the spatial partitioning problem is similar to, but distinct from, both the graph partitioning problem as well as the \(k\)-means clustering problem [12]. While the graph partitioning problem also seeks to balance aggregated weights among the \(k\) partitions, it also requires the edge weights across the partitions to be minimized, but does not enforce contiguity. Spatial partitioning is also similar to the \(k\)-means clustering problem [33], but the decision space search is based on the adjacency graph, instead of on units and distance.

The above network flow abstraction provides a mathematical interface for an MIP solver. For heuristic solutions, however, such a flow definition is not necessary because the contiguity constraint can be handled with a penalty function or enforced within the neighborhood search routine. An EA solver typically encodes a partition solution into a linear array that is indexed by spatial units, \(\{x_i | x_i \in I\}\), where the zone index is the value of each element. A fitness function is defined to capture the objective. If a penalty function is defined to handle infeasible solutions that violate any of the constraints, the fitness function would be different from the objective function. Alternatively, the fitness and objective functions can be identical and one can define a separate unfitness function. New solutions can then be generated using the crossover and mutation operators in each EA iteration. The evaluation of new solutions checks all of the constraints and returns a score for each solution. A replacement strategy uses this score (and the value of the unfitness function, if defined) to select a subset of the population and possibly replace them with the new solutions. We now investigate how to preserve contiguity in the crossover and mutation operations.
3 METHODOLOGY

Figure 1: **Search space of mutation for spatial optimization problems (in a rook neighborhood).** Subfigure (a) is a solution for 9 spatial variables on a 2D raster layout. Subfigure (b) shows the number of mutable possibilities for each cell. Subfigure (c) illustrates search paths needed to repair a mutation operation.

3.1. **Satisfying Spatial Constraints**

We begin with a simple illustration to demonstrate how a contiguity requirement alters the search of the decision space for the classic 1-bit EA mutation operator. For ease of illustration, our example employs a raster representation of the spatial variables. Our algorithm, however, applies to both vector- and raster-based problems since we adopt an adjacency graph structure to represent neighborhood, which generalizes the adjacency of spatial objects of both regular (e.g., raster) and irregular shapes. In this example, the adjacency is rook-based, which means that if two units are connected by only a point (queen connectivity), then they are not considered connected.

In Figure 1, the leftmost figure shows nine spatial units partitioned into three contiguous zones. Using a classic 1-bit mutation, the new solutions that result from mutating a single cell are shown in the middle figure. The number of possibilities is $2 \times 9 = 18$, because each cell can be re-assigned to either of the other two zones. However, note that mutating cell (1,1) breaks contiguity and necessitates a repair operator to restore contiguity. The rightmost figure in Figure 1 demonstrates a repair scenario. First, cell (1, 2) switches from yellow to blue. The repair operator then connects it to the existing blue zone, {3,3}, by identifying a route between them and re-assigning the cells on that route to the blue zone while maintaining the two other zones. The arrows show possible repair paths. Note that the route (1, 1) → (2, 1) → (3, 1) → (3, 2) is not possible because it eliminates the yellow zone. Similarly, the route (2, 2) → (2, 3) is not possible because it splits the green zone. As we can see, repairing non-contiguous solutions is computationally costly, even for a simple mutation operator.
3.2. **Limitations of EA Crossover Operators**

Classic EA crossover operators that linearly combine parent solutions have two main drawbacks. First, even in an unconstrained decision space, the search space derived from linear recombination is smaller than the enumerable possibilities [13]. Hence, these operators limit and may miss feasible search regions. Second, since these recombination operators are spatially unaware, they may generate solutions that violate the spatial constraints. These solutions would either be discarded, leading to poor efficiency, or would be repaired, which is non-trivial and computationally expensive.

Figure 2 illustrates these issues with two commonly employed crossover operators. The first, the classic linear crossover using a single cutpoint, is shown in the upper diagram of Figure 2. Here, the two parent solutions are shown on the left as 2D and 1D encoded (chromosome) views. After a cutpoint is chosen, the crossover generates two new solutions, shown on the right, by swapping the second part of one chromosome and attaching it to the first part of the other chromosome. However, since neither of the two new solutions is contiguous, they are both infeasible.

The bottom diagram in Figure 2 illustrates a spatially-aware crossover operator that overlaps two solutions and randomly reassigns the resulting subdivisions into contiguous zones [60]. A subdivision is identified with a unique \(<\text{zone1}, \text{zone2}>\) label which denotes a cell’s zone assign-
3 METHODOLOGY

ment from the two parent solutions. The goal of this type of crossover is to preserve the favorable attributes from both parent solutions. This is intuitively appealing, though not always successful as we can see from the subdivisions shown in the diagram. In the extreme case, the overlap creates 9 subdivisions, which results in no progress since the new problem is identical in complexity to the original problem. Moreover, it is not clear how a random graph cut can be specified to preserve and propagate favorable subdivisions. Despite these drawbacks, however, a significant benefit of the overlap crossover remains—it can create a search space of size at least $3^3 - 2 = 25$ (by taking the largest three subdivisions as starting zones and parceling out the remaining three cells, but subtracting the two existing solutions), which is a significant improvement over the classic EA crossover.

3.3. Spatial Recombination Approach

These examples highlight that an important component has been missing from the extant EA spatial recombination operators. In particular, they do not consider spatial characteristics during the new solution generation phase. Spatial recombination operators must transform the spatial characteristics into quantifiable measures that direct the decision space search. This is akin to incorporating domain knowledge into heuristics. The main difference is that spatial characteristics require an integrated framework for incorporating multiple spatial elements while maintaining the efficiency of each EA iteration. In this direction, Xiao [60] made an initial effort by integrating graph theoretic components to categorize spatial optimization problems. This effort needs to be further extended, and more importantly, must be systematic.

We propose a Spatially Explicit Evolutionary Computation (SEEC) approach that adheres to the basic structure of EA recombination procedures, which seeks to pass desirable attributes from parent solutions to child solutions, as well as enables large-scale spatial optimization. Similar to Xiao [60], we represent spatial relationships using graph data structures, on which we design a chain of local moves that collectively comprise a large disturbance in the search neighborhood. Chained moves are desirable since a chain, which is tied on either end to parent solutions, allows one to design globally large, but locally incremental moves, within the search space. The length of the chain may be adjusted to permit a controlled but sufficiently large move to make an impact. Auspiciously, chained moves can be seen as a generalization of the mutation and crossover operators in EAs. A mutation operator can be designed as a series of chained moves where the chain, at
both ends, is anchored to a single parent solution. A crossover operator can be designed as a set of chained moves that comprise a “walk” from one parent solution to a different parent solution. The movement along this chain generates a series of intermediate or child solutions.

Coupling randomization with a chaining mechanism provides a constructive method for designing spatial recombination strategies that incorporate spatial neighborhoods and chaining on graphs. This observation inspires the design of our spatial recombination operators as spatial adaptations of two well-established heuristic methods, path relinking [14] and ejection chain [63], both of which embody the chaining concept to efficiently and effectively solve large combinatorial (non-spatial) optimization problems.

3.4. Adapting Path Relinking

A path relinking process begins by identifying two solutions, an initial (or source) solution, \( S \), and a reference (or target) solution, \( T \). A path that links these two solutions transforms the source solution to the target solution. The hope is that somewhere along the path (in the neighboring space) lie new and better solutions that are a mix of elements found in \( S \) and \( T \). These moves can be designed adaptively to improve the performance of the heuristic. The path links multiple random and purposeful moves, where each move is an incremental change from the previous state. Path length is a function of solution distance from \( S \) to \( T \), which can be defined for non-spatial problems as the number of variables where the two parent solutions have different values. At each step, a neighborhood function randomly identifies a local move. With each step forward, the distance is reduced along the relinking path and reaches zero when the path relinking is complete.

It is worth noting that EAs and path relinking are closely related. Glover [13] relates Genetic Algorithms and scatter search, a specific form of path relinking in linear programming. He highlights that the link between these two approaches is rooted in principles underlying mathematical relaxations. Scatter search, and its more general form, path relinking, provides an ordered way to strategically explore the neighborhood space for new solutions. The neighborhood space is defined by the population, not the affinity of a single solution, and thus needs one or multiple parent solutions from the current population as input. In this way, scatter search exhibits commonalities with EAs since it is iterative, selects moves within a neighborhood, and performs recombination through the exploration of the neighborhood space.
3 METHODOLOGY

These ideas underlie the design of our spatial recombination methodology. We must, however, adapt existing path relinking and ejection chain methods to a spatial context. For example, solution distance, which is often defined as the number of alleles with different values between two solutions, is not applicable in spatial configurations because spatial partitions differ in their partition shape. Zones with the same shape but different zone indices are identical but would be considered different in a chromosome encoding. For this reason, while a path relinking process can surely transform one chromosome to another by flipping the bits that differ, it may not be able to walk from one spatial configuration to another because spatial constraints such as contiguity, containment, or shape, may shorten, lengthen, or even block the path. The design of spatial recombination operators is, thus, significantly more complex than classic crossover and mutation, or conventional path relinking and basic ejection chain methods.

3.5. Spatial Crossover through Path Relinking

Figure 3 shows a simple partitioning problem with a contiguity constraint and illustrates the steps in our path relinking-based crossover (PRCRX) algorithm. The example problem partitions 4 contiguous zones on a 4x4 grid, where each cell is a problem variable. Subfigure (a) shows two parent solutions. The solution on the left is the source solution, $S$, while the solution on the right is the target solution, $T$. The colors represent different zones. We first “overlap” these two
solutions (shown in subfigure (b)), and assign a zone label, \( s \cdot t \), to each cell, where \( s \) is the cell’s zone assignment (here, representing the cell color) in \( S \), and \( t \) is the zone assignment in \( T \). This creates a set of connected components. Units in one connected component share a common zone label. Call each connected component a group, \( G \). Adjacent groups have different zone labels, but non-adjacent groups may have the same zone label.

Suppose this is a \( k \)-partition problem. The next step is then to pick \( k \) seed groups, where each group has a unique zone label, \( t \). For example, we could simply pick those groups with zone label \( z \cdot z \), for \( z = 1, \ldots, k \), if they exist. The solution distance is defined to be the difference between the total number of cells and the number of cells in the \( k \) seed groups. We can see in Figure 3(c) that, at most, 7 moves are needed to “walk” from the source solution to the target solution. The 7 mutable units (i.e., the cells with a label and outlined with a black border) form \( k \) candidate sets, each indexed by the label \( t \). A candidate set, \( C_z \), is the complement of the corresponding seed group \( G_z \), i.e., \( G_z \cup C_z = Z_z \), where \( Z_z \) is zone \( z \). That is, \( C_z \) comprises all of the remaining units of zone \( z \) in \( T \). In Figure 3(c), we can see that the 4 candidate sets include 1 yellow, 3 red, 1 blue, and 2 green cells in the target solution. The solution distance can be defined using \( G \) and \( Z \) as follows.

**Definition.** The solution distance, \( d \), between the source solution \( S \) and the target solution \( T \) is defined as \( d = \sum_{z=1}^{k} |C_z| = \sum_{z=1}^{k} |Z_z - G_z| \).

We may begin the relinking process between the two solutions by first choosing a particular source zone, i.e., a set of units that share the same zone label, \( s \). In each move, or step of the path, we convert a cell from zone \( s \) to zone \( t \), reducing the solution distance, \( d \), by 1. Subfigures 3(d–f) provide an example of a path relinking process that morphs the source solution (subfigure (c)) to the target solution (subfigure (g)). To move from subfigure (c) to subfigure (d), we convert all of the cells labeled “Y” to the yellow zone. To move to the solution shown in subfigure (e), we convert all of the cells labeled “R” to the red zone. To obtain subfigure (f), we convert the cells labeled “B” to the blue zone. Finally, we convert the cells labeled “G” to the green zone, which completes the path. Here, our path of length 4 travels a distance of 7 and generates 4 intermediate solutions.

The path can be constructed in multiple ways. One could, alternatively, move each candidate cell individually, generating 7 intermediate solutions (6 new). Or, as long as each move does not violate the spatial constraints, we could build a path with varying step sizes. The path length is flexible, but constrained by the number of available mutable candidate cells. It is possible that a
particular order of candidate set/unit visits may not be viable, but this does not pose an issue since the purpose of the walk is to generate intermediate solutions, not necessarily to end at solution \( T \).

In the construction of any path, an important consideration is to properly deliberate seed groups before the relinking process begins to avoid the unnecessary computation that arises from generating infeasible intermediate solutions. For example, recall that when we created the \( k \) seed groups, we chose those groups with identical \( s \) and \( t \) zone labels. This is one of a set of possible choices. We could have chosen any \( s \cdot t \) label where the \( k \) seed groups have \( k \) unique \( s \) and \( k \) unique \( t \) labels. However, as Figure 4 illustrates, there is not always a unique zone labeling in the overlap solution. If a unique zone labeling for both \( S \) and \( T \) does not exist, the two source zones will merge as we expand the groups, \( G_t \), because there are at least two seed groups that share a source zone. Moreover, in the expansion of the two seed groups to target zones, the fitness of the intermediate solutions is likely to be worse. Accordingly, in our implementation, we seek to identify unique zone labelings and to minimize groups with the same source zone labels. Another complication that may arise is that a violation of the contiguity constraint may prevent the completion of a path. Figure 5 illustrates that if a group, \( G_t \), or units in \( C_t \) are poorly selected, it may be impossible to expand \( G_t \).

It is difficult to know a priori whether a seed grouping or a particular selection will lead to one of these problematic scenarios. This phenomenon affects the search space size, but the effect on fitness improvement is unknown. Furthermore, sophisticated algorithms for preventing such scenarios may be too costly to compute. Our current implementation detects if a potential move violates the contiguity of any source zone. If so, we choose a different move. If none of the
3 METHODOLOGY

Figure 5: **Seed group selection constrained by source zone contiguity.** The selection of the yellow seed groups makes it impossible to maintain the contiguity of source zone 1. The expansion of group $G_2(2 \bullet 2)$ to group $G_2(1 \bullet 2)$ disconnects source zone 1. An appropriate zone labeling would select $G_2(1 \bullet 2)$ as the seed group.

Candidate moves is possible, we simply terminate the relinking process. Other implementations are possible and can be flexibly designed.

3.6. Spatial Mutation through Ejection Chain

ECMUT (ejection chain-based mutation) builds on our previous spatial mutation operator [31] by generalizing the chained mutation steps into the ejection chain heuristic framework [64]. An ejection chain is a series of moves, each of which “ejects” an assignment of a problem variable to the next variable. This type of ejection forms a chain that can either be cyclic or acyclic. An ejection chain is a generalization of local neighborhood functions such as shift (chain length = 1) and swap (chain length = 2). Non-spatial or spatial considerations may be utilized to determine the chain length and the units to eject in each move. To transform a general ejection chain to one that is spatially explicit, we need to specify a spatial neighborhood function, which could be, for example, defined by a traversal that proceeds along the adjacent vertices.

Notice that ejection chain-based mutation has an advantage over the location-based morph spatial design where one moves a few units to a new location [60]. The drawback of the latter is that, while the new area is able to contain the shape of the units, variable interdependencies are not preserved. Ejection chain-based mutation, on the other hand, preserves spatial dependencies since the exchange of unit assignment occurs among neighboring units.

Figure 6 provides an illustration of the ECMUT operator process. Here, our ejection chain is cyclic, with the cycle (yellow → blue → red → green →), which returns to yellow after green. Subfigure (a) shows a parent solution with 4 colored zones. At each step of the chain, a random set of contiguous cells from one zone is moved or “ejected” to its neighboring zone. Since our zone order is yellow → blue → red → green →, we first randomly move selected yellow border
cells (subfigure (b)), then blue border cells (subfigure (c)), then red border cells (subfigure (d)), and finally green border cells (subfigure (e)), to complete a cycle. Each of the four intermediate solutions shown in subfigures (b)–(e) is a valid solution that is considered at the EA replacement phase.

4. The PRCRX and ECMUT Algorithms

In graph terminology, the spatial partitioning problem of partitioning $n$ spatial units into $k$ disjoint but contiguous zones is equivalent to finding $k$ disjoint connected components on the adjacency graph that cover all $n$ vertices. In our SEEC implementation, both rook and queen adjacency are supported. Rook adjacency leads to a clearly planar graph while queen adjacency may not lead to a planar graph. We employ a chromosome encoding for solutions and groups with operations that rely on the adjacency graph and its associated graph operations.

4.1. Spatial Crossover (PRCRX) Algorithm

Our spatial crossover method has the following characteristics.

- The overlap (with its associated unique zone labeling) creates a set of connected components that mix the spatial configuration of two parent solutions. Overlapped groups serve as the basic unit for crossover recombination.

- The seeding of $k$ groups defines the solution distance, which is the maximum path length. Which $k$ groups should be selected as seeds is a question whose answer is closely related to the maintenance of spatial constraints and the search paths that lead to improved fitness.

- Group expansion transforms the source solution to the target solution. The expansion is based on the selection of adjacent units in candidate sets.

- The relinking path is a transformation of the source zone into the intermediate solutions. The mutable units chosen for expansion are determined by the fitness improvement of these intermediate solutions. To avoid costly repair operations, this expansion preserves contiguity.
The possibility of generating different and new solutions lies in the flexibility of choosing both different candidate sets as well as different units in these sets at different steps of the path.

Randomization is invoked at many stages, including group seeding, ordering of the candidate set, and the selection of the adjacent candidate unit.

Algorithm 1 A general overlap operation.

1: function \textsc{Overlap}(U, S, T)
2: for \( u \in U \) do
3: \( i \leftarrow S[u] \)
4: \( j \leftarrow T[u] \) \quad \triangleright \text{zone assignment in the two solutions}
5: \( X_{i,j} \leftarrow X_{i,j} \cup \{u\} \)
6: return \( \{X_{i,j} | X_{i,j} \neq \emptyset\} \) \quad \triangleright \text{return the overlap set}

The path relinking process has three primary steps: overlapping, group seeding, and path building. Algorithm 1 outlines a general overlap algorithm that applies to all combinatorial chromosome encoding optimization problems. It returns the overlap, \( X \), as a collection of group sets with a time complexity of \( O(n) \). However, since a group in \( X \) is indexed using a zone label, it does not differentiate two groups with the same zone label, which is possible when two zones from \( S \) and \( T \) intersect and result in multiple connected components.

Algorithm 2 modifies the basic overlap operator to be spatially explicit. This algorithm is based on the well-known connected component labeling algorithm [4], but has been adapted to our graph representation of the problem. The algorithm applies breadth-first search (BFS) and returns the overlap, \( G \), and its chromosome encoding, \( X \), where \( X[j] \) is the group assignment of unit \( j \). The time complexity is \( O(m + n) \), where \( m \) is the number of edges on the adjacency graph. For planar graphs, \( m \) is a constant factor of \( n \), according to Euler’s Formula. Other efficient implementations may use the Union-Find algorithm [4] within \( O(n\alpha(n)) \), where \( \alpha \) is a very slow-growing inverse of the rapidly increasing Ackermann function.

The group seeding step appropriately selects \( k \) seed groups from \( \{G[i], i = 1, \ldots, l\} \), where \( l \) is the number of groups derived from \( X \). The solution distance between partition \( S \) and \( T \) is \( l - k \). The overlap algorithm creates at least \( k \) and at most \( n \) groups. If the \( k \) seed groups are small in size such that they do not comprise many units, the solution distance will be large. In this case, the path building process is computationally costly. If the \( k \) selected seed groups are large in size, the number of available moves is limited and consequently less likely to generate desirable new solutions along the path. However, this issue is ameliorated in large scale problems. In
Algorithm 2 The spatial overlap operation.

```plaintext
1: function SPATIAL OVERLAP(U, S, T)
2:    X = [0]
3:    i = 1
4:    Q = ∅
5:    for u ∈ U do
6:        if u has been assigned a group then
7:            continue
8:        X[u] = i
9:        Q.enqueue(u)
10:       while Q ≠ ∅ do
11:           u = dequeue(Q)
12:           Nu = {v | v is a neighbor of u on the adjacency graph}
13:           for v ∈ Nu do
14:               if v has been assigned a group then
15:                   continue
16:               if S[u] = S[v] and T[u] = T[v] then
17:                   X[v] = X[u]
18:                   Q.enqueue(v)
19:           i = i + 1
20:       l = i;
21:    build G = {G[i], i = 1, . . . , l} from X
22: return X and G
```

our empirical work, we find that selecting $k$ large seed groups with unique zone labeling, while as effective as random group selection, is more efficient because the solution distance is shorter. Finally, the seeding step returns seed groups, $G_t$, and their corresponding candidate sets, $C_t$, for $t = 1, . . . , k$.

Once seed groups have been determined, we begin the path building process. Algorithm 3 presents a general path relinking algorithm based on the overlap of two solutions. It applies to non-spatial problems through the expansion of the $G_t$ groups and the associated update to the source solution. New solutions are generated by changing a randomly selected unit in $C_t$ to the assigned value of $G_t$ variables. Without modification, this algorithm applies to set partitioning problems. With modifications to how $C_t$ is constructed, the algorithm can be applied to assignment problems without the partition requirement. In this way, the coupling of overlapping and path relinking (or ejection chain) provides a new EA crossover choice.

Notice that our algorithm expands the $G_t$ groups unit by unit, not group by group. Since the unit is the finest level of granularity, adapting to expand an arbitrary number of mutable units in $C_t$ for a single movement is straightforward. There is a trade off. Expanding a group (i.e., a set of units) is more efficient because the number of solution evaluation steps (at line 16) is reduced. However, unit level expansion permits greater exploration of the search space.
Algorithm 3 A general path relinking crossover algorithm.

1: function FLIP(S′, Gt, Ct)
2: randomly select a unit \( u \in C_t \)
3: \( G_t = G_t \cup \{u\} \)
4: \( S'[u] = \) the source zone of \( G_t \)
5: \( C_t = C_t - \{u\} \)
6: return \( S' \)

8: function PRCRX(S, T)
9: \( X = overlap(S, T) \)
10: \( \{G_t, \{C_t\} = seedKuniq(X, k) \)
11: \( d = n - \sum_{t=1}^{k}(|G_t|) \)
12: \( S_{best} = S' = S \)
13: for \( i = 1, \ldots, d - 1 \) do
14: randomly select a zone \( z \) in \( T \)
15: \( flip(S', G_t, C_t) \)
16: evaluate \( S' \)
17: if \( S' \) is better than \( S_{best} \) then
18: \( S_{best} = S' \)
19: return \( S_{best} \)

Algorithm 4 adapts the general path relinking algorithm to a spatial path relinking algorithm. In Algorithm 3, while the FLIP( ) function may produce disconnected components, in Algorithm 4, this issue is bypassed by building paths based on the spatial relationships represented on the adjacency graph and graph search methods. It conducts seed group expansion on neighboring candidate units and ensures contiguity by maintaining a second adjacency graph for the zone boundary units.

As illustrated in Figure 5, at the time of seed group expansion, we must ensure that the addition of a mutable unit does not disconnect the source zone. One way to check zone contiguity is to count, starting from a randomly picked unit, the number of connected units in the zone. If the count before the mutable unit removal is not equal to one more than the count after the removal, the zone is broken. This is an easy, though computationally expensive check that requires \( O(d \times \frac{n}{k} \times \mu) \) time for \( k \) zones, where \( \mu \) is the average degree of unit connectivity. Although \( \mu \) is not large on a planar graph, this function must be called for each mutable unit. A more efficient way to ensure contiguity is to check whether the boundary units (using queen neighborhood) are connected only before and after unit removal. This check takes \( O(d \times \phi(n, k) \times \mu') \), where the function \( \phi \), which depends on the shape of the zones, estimates the number of boundary units in a zone and \( \mu' \) is the average degree of connectivity among boundary units, which is much smaller than \( \mu \). Compact shapes have small \( \phi \) values.
Algorithm 4 PRCRX: a spatial crossover algorithm based on path relinking.

1: function PRCRX(S, T)                                   ▷ adjacency graph for all the n units. $N(u)$ returns $u$’s neighbors
2:     $N \triangleq$ adjacency graph for all the $n$ units. $N(u)$ returns $u$’s neighbors
3:     $X = spatial\_overlap(S, T)$ ▷ build overlap
4:     $\{G_1, C_1\} = seedKuniq(X, k)$ ▷ build seed groups and candidate sets
5:     $d = n - \sum_{t=1}^{k} \vert G_t \vert \triangleq solution\_distance$  
6:     $S_{best} = S'$
7:     for zone $z \in S'$ do ▷ establish boundary unit adjacency graph for contiguity check
8:         build adjacency graph $BG_z$ for boundary units in $z$
9:     for zone $z \in T$ do ▷ build initial adjacent unit set to $G_z$ as candidate units
10:        $AC_z = \emptyset$  
11:        for $u \in G_z$ do ▷ a candidate unit neighbors $G_z$ but in $C_z$
12:           $AC_z = AC_z \cup \{v \mid v \in N(u) \land v \in C_z\}$
13:           $zsgt_z = \text{the zone of the initial } G_z \text{ in } S$ ▷ the zone where mutable unit will move
14:        $path = []$  
15:        $pathlen = 0$
16:     Do
17:         zone seq = a random sequence of size $k$ to visit each zone in $T$
18:     for $z \in zone\_seq$ do ▷ the mutable unit
19:         $mu = 0$ ▷ stores the best candidate unit in $C_z$
20:         $cu_{best} = 0$
21:     for $cu \in AC_z$ do ▷ the zone of $cu$ in $S$, not in $S'$
22:         $zs = S[cu]$ ▷ same zone, no effect; but update $G_z, C_z, AC_z$
23:         if $zs = zsgt_z$ then
24:             $mu = cu$
25:             break
26:     else ▷ peek to see if it is mutable and with fitness improvement
27:         check $BG$ for contiguity  
28:         if contiguity is maintained then ▷ whether adding $cu$ to $zsgt_z$ disconnects a zone in $S'$
29:             evaluate fitness of $S'$ if $cu$ is moved
30:         if the $cu$ move leads to a better solution then
31:             $cu_{best} = cu$
32:     if $mu = 0$ then ▷ not ineffective move
33:         if $cu_{best} > 0$ then ▷ found an effective mutable unit
34:             $mu = cu_{best}$
35:     $AC_z = AC_z \setminus \{mu\}$; expand $AC_z$ to include $mu$’s neighbors
36:     $G_z = G_z \cup \{mu\}$ ▷ update $G_z$
37:     $C_z = C_z \setminus \{mu\}$ ▷ update $C_z$
38:     if $mu$ is effective then ▷ update boundary unit adjacency graph of $S'$
39:         update $BG$
40:         $S'[mu] = zsgt_z$  
41:         $path[pathlen++] = mu$ ▷ update the intermediate solution  
42:         if $S'$ is better than $S_{best}$ then ▷ record the move on the path
43:             $S_{best} = S'$
44:     while there was successful expansion on any $G_t$
45:     return $path$ and $S_{best}$
Algorithm 5 A greedy algorithm for optimizing a relinked path.

function $PATH_{\text{OPTIMIZE}}(S', T, G_t, path_1, path_1\text{len})$

$S_{\text{best}} = S'' = S$

for zone $z \in S''$ do

▷ establish boundary unit adjacency graph for contiguity check

build adjacency graph $BG_z$ for boundary units in $z$

for $z \in T$ do

$muindex_z = \text{index of the first moved unit in } G_z$

$path_2 = []$

$path_2\text{len} = 0$

while $path_2\text{len} < path_1\text{len}$ do

$mu = 0$

$zsmu = 0$

for zone $z \in T$ do

if $muindex_z = \text{len}(G_z)$ then

continue

end if

$cu = G_z[muindex_z]$

$zsgt = S''[cu]$

check $BG$: whether adding $cu$ to $zsgt$ disconnects a zone in $S''$

if contiguity is maintained then

evaluate fitness of $S''$ if $cu$ is moved

if the $cu$ move leads to a better solution then

$mu = cu$

$zsmu = zsgt$

end if

end if

if $mu = 0$ then

break

end if

update $BG$

$S''[mu] = zsmu$

$path_2[path_2\text{len}] = mu$

if $S''$ is better than $S_{\text{best}}$ then

$S_{\text{best}} = S''$

$path_2\text{len}++$

$muindex_z++$

end if

end while

return $path_2$ and $S_{\text{best}}$
PRCRX randomizes the order in which the $T$ zones are visited as well as the order for adding adjacent units in candidate sets. This randomization diversifies the decision space search. The resulting path, called path1, can be further optimized at little cost. Algorithm 5 implements an optimization strategy by scanning path1 in a greedy fashion. It begins with $k$ seed groups, formed at the beginning of the relinking process in PRCRX, and checks the $k G_t$ groups to identify the best mutable unit to add to a new path. This process requires $O(path \_ length)$ time and creates another path, called path2, along which another best solution is found. This process allows us to identify the best choice from the two paths as the output of the path relinking process.

4.2. Spatial Mutation (ECMUT) Algorithm

Our mutation algorithm is designed within the same path relinking framework. The primary difference is that, in the mutation operator, the initial solution and the target solution are identical. That is, the operator searches a neighborhood along a path that begins and ends at the same solution. The chaining process is similar. In ECMUT, we alternate the zones to visit, and each visit identifies a number of mutable units, at least one of which is on the zone boundary. These units are then moved to a neighboring zone, identified from the zone adjacency graph. Algorithm 6 shows how a general ejection chain mutation algorithm can be adapted from a path relinking framework. Algorithm 7 modifies this general algorithm to incorporate spatial considerations. It extends a previous implementation in Liu, Cho and Wang [31] by allowing the chain length to be flexible in the multi-mutation strategy. Depending on particular spatial configurations, ECMUT may or may not be able to produce cyclic chains, which is not required by the optimization process.

\begin{algorithm}
\begin{algorithmic}
\Function{ECMUT0}{$S, ecLength, cyclic$}
\State $seq$ = a random sequence of $1, \ldots, n$
\State $z_{inject} = S[seq[1]]$
\For{$i : 2 \ldots \max(n, ecLength)$}
\State $z_{eject} = S[seq[i]]$
\State $S[seq[i]] = z_{in}$
\State $z_{inject} = z_{eject}$
\EndFor
\If{cyclic is True}
\State $S[seq[1]] = z_{inject}$
\EndIf
\EndFunction
\end{algorithmic}
\caption{A general ejection chain-based mutation algorithm.}
\end{algorithm}
Algorithm 7 ECMUT: a spatial mutation algorithm based on ejection chain.

1: function ECMUT(U, S, ecLength, blocksize)  
   \(\triangleright\) randomize the order of zone visit. ecLength: chain length
2:   seq = a random sequence of 1, \ldots, ecLength  \(\triangleright\) solution set that holds improved solutions during the chaining
3:   Solutions = ∅  \(\triangleright\) iterative ejection chain building
4:   fitness0 = fitness(S)
5:   q = 1  \(\triangleright\) MU sequence index
6:   for i = 1, \ldots, ecLength do  \(\triangleright\) initialize units in each zone
7:     zeject = (seq[i] \% k) + 1  \(\triangleright\) zone index of the eject zone; k is the number of zones
8:     Zeject = Z[zeject]  \(\triangleright\) unit set of the eject zone
9:     zinject = randomly select a receiving zone for zeject
10:    Zinject = Z[zinject]  \(\triangleright\) unit set of the inject zone
11:    Ubinject = \{ u | u is a boundary unit of zone zinject \}  \(\triangleright\) select set of contiguous units, \(\Delta_q\) where \(\Delta_q \subseteq Z_{eject}, |\Delta_q| \leq \text{blocksize}\) and \(\Delta_q\) is adjacent to Zinject
12:    if \(\Delta_q = \emptyset\) then continue  \(\triangleright\) move these units to the inject zone
13:       for u ∈ \(\Delta_q\) do
14:          \(\triangleright\) move these units to the inject zone
15:             S[u] = zinject
16:            if fitness(S) is better than fitness0 then
17:               Solutions = Solutions ∪ \{ S \}
18:       return Solutions
19: 20:
21: function SELECT\_MUTABLE(P, B, maxCount, C)
22: \(\triangleright\) P: a pool of contiguous units
23: \(\triangleright\) B: neighboring units to P, each sharing a common border to at least one unit in P
24: \(\triangleright\) maxCount: max number of movable units to select, as the stopping rule in the search
25: \(\triangleright\) C: an intermediate solution in which each zone’s contiguity holds after MU selection
26:   Ub = \{ u | u ∈ P and u is adjacent to at least one unit in B \}  \(\triangleright\) border units in P to B
27:   u0 = a unit randomly selected in Ub  \(\triangleright\) initial movable unit set
28:   M = u0  \(\triangleright\) randomized recursive traversal of the adjacency graph of P to select movable units
29:   randUnitSearchRecur(P, M, maxCount, C)
30: return M
31: 32:
33: function RAND\_UNIT\_SEARCH\_RECUR(P, M, maxCount, C)
34: \(\triangleright\) M: reference to the modifiable set of movable units
35: if maxCount = 0 then  \(\triangleright\) recursion exit
36:   return
37: \(\triangleright\) find neighboring units not yet included in the movable unit set
38:   if N = \{ neighbors of units of M in P \} then  \(\triangleright\) recursion exit
39:      m = a random number in [1, min(|N|, maxCount)]
40:     M’ = up to m randomly selected units from N, that keep C contiguous
41:   M = M ∪ M’
42:   randUnitSearchRecur(P, M, maxCount – m)
5. Empirical Evaluation

We evaluate PRCRX and ECMUT with an application to a general spatial zoning problem that has many applications, including, for example, the drawing of school district boundaries, land use planning, and urban planning [6]. This zoning problem follows the general definition of the spatial partitioning problem presented in section 3. In this application, the objective function has two components. The first is a measure of the difference in weights between the \( k \) zones. The second is a measure of competitiveness, which can be thought of generally as the occupancy rate of two competing agents. The fitness of each solution is a weighted measure of the competitiveness and weight balance of the zones. The fitness improves as the weights are increasingly balanced across the set of zones and the occupancy rate of the agents is close to 50% in each zone [18]. While this is how we define fitness in our particular application, the fitness is a modular component and can be flexibly specified as appropriate for any particular application. Contiguity is the only spatial constraint considered in the evaluation.

We evaluate our spatial EA performance in five ways. First, we compare the solution quality obtained and the computational performance with other existing heuristics. Second, we examine the performance of just the ECMUT operator. Third, we compare the performance gain when PRCRX is used in addition to ECMUT. Fourth, we study the performance characteristics of PRCRX. Finally, we conduct tests in a parallel computing environment to examine the algorithm’s scalability and convergence properties.

5.1. Implementation and Case Study

Liu, Cho and Wang [31] developed PEAR, a high-performance computing tool for the spatial partitioning of geographic units and demonstrates its utility for political redistricting. PEAR implements a mutation operator and a basic overlay+expansion crossover operator. They found that the mutation operator was effective in searching the decision space but the performance of the crossover operator was limited. In our SEEC implementation, the mutation operator is similar, though we replace it with a modified version, ECMUT, and we develop a new spatial crossover operator, PRCRX. These new operators fit within the ejection chain and path relinking heuristic frameworks. We also now include a random restart feature to avoid EA population homogeneity when a local population converges to similar solutions in a small search region where new progress become difficult to obtain.
SEEC is implemented in C++, and compiled using GCC 4.9.2. The parallel implementation uses the MPICH 3.1.4 library. To enable asynchronous migration [29], we utilize MPI non-blocking functions (i.e., `MPI_Isend()`, `MPI_Iprobe()`), and regular `MPI_Recv()`. The SPRNG 2.0 C library provides a unique random number sequence for each MPI process. Performance tests are conducted on the ROGER supercomputer at the National Center for Supercomputing Applications (NCSA), Urbana, Illinois, USA. Each node on ROGER is configured with the Intel Xeon E5-2660 processor (2.6GHz, 20 cores/node), and 256GB memory. The cluster is connected by a high-speed network with 40Gb/s switches in the core and 10Gb/s uplinks to each node. The experimental environment is the same as that used in Liu, Cho and Wang [31]. Scalability development for SEEC is conducted on the Blue Waters supercomputer.

Our EA search process starts with a set of feasible solutions and considers only contiguous candidates in each iteration. Non-contiguous infeasible solutions are excluded to enhance computational efficiency. Contiguous but infeasible solutions that violate other constraints are permitted. Instead of employing a penalty function, we define an unfitness function to score infeasibility. This function is part of the replacement strategy for updating the EA population. This means that our fitness function definition is the same as the objective function (whereas for a penalty function, the fitness function is usually different from the objective function).
Table 1: **PEA Parameter Settings.**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fitness</td>
<td>$0.2 \times \text{equipopulation} + 0.8 \times \text{competitiveness}$</td>
</tr>
<tr>
<td>PRCRX output</td>
<td>the better solution from PRCRX() and PATH_OPTIMIZE()</td>
</tr>
<tr>
<td>Population size</td>
<td>200</td>
</tr>
<tr>
<td>Selection</td>
<td>binary selection</td>
</tr>
<tr>
<td>Initial population</td>
<td>80% by region border; 20% by administration border</td>
</tr>
<tr>
<td>Elitism</td>
<td>on</td>
</tr>
<tr>
<td>Homogeneity check interval</td>
<td>20,000 iterations</td>
</tr>
<tr>
<td>Homogeneity threshold</td>
<td>95% population’s solution distance $&lt; 10% \times n$</td>
</tr>
<tr>
<td>Export/import interval (PEA)</td>
<td>100/50</td>
</tr>
<tr>
<td>Migration rate (PEA)</td>
<td>2</td>
</tr>
<tr>
<td>Sending parallelism (PEA)</td>
<td>4</td>
</tr>
</tbody>
</table>

We demonstrate the performance of our algorithm with data from the year 2011 in the state of North Carolina. The study area includes 2,690 spatial units, shown in Figure 7. The rook and queen neighborhood matrices are obtained using the open source GIS libraries, PySAL (http://pysal.org) and GDAL (http://gdal.org). For each of the experiments reported, SEEC is configured with the parameters specified in Table 1.

### 5.2. Comparison with Other Heuristics

We first compare the sequential EA (with the PRCRX and ECMUT operators) to alternative optimization heuristics that have been designed by others for the spatial partitioning problem. These other heuristics include simulated annealing, greedy algorithm, tabu search, GRASP, and GRASP (contiguous), which has contiguity support. Each experiment is conducted on a single dedicated computing node on ROGER. The results from the alternative heuristics are reproduced from Liu, Cho and Wang [31].

We can see in Table 2 that among the alternative algorithms, the GRASP (contiguous) algorithm produced the best result. Its best fitness, 0.0411, was identified near the 5 hour mark of a run with about 20-hours of total computing time. Compare this performance with the ECMUT-only run. The “ECMUT snapshot” line shows when ECMUT outperformed the best result from the GRASP (contiguous) run. As we can see, ECMUT handily outperformed these other algorithms, identifying a solution with a fitness value of 0.0410 in just 728.2 seconds. Moreover, ECMUT

---

2These data are publicly available from the U.S. Census Bureau (https://www.census.gov/programs-surveys/geography/geographies/reference-maps.2011.html) and North Carolina General Assembly web site (https://www.ncleg.gov).
continued to improve over the course of its one-hour run. Its best fitness value was 0.0362, as shown on the line labeled “ECMUT best.”

The performance of SEEC improved even more significantly when the PRCRX operator was included along with ECMUT. The results when both EA operators were employed are shown in the lines labeled “PRCRX+ECMUT.” The snapshot line shows that PRCRX+ECMUT surpassed the best solution identified by GRASP (contiguous) in just 148.96 seconds. That is, to identify a solution of similar solution quality, GRASP (contiguous) required 5.3 hours while ECMUT took only 12.1 minutes and PRCRX+ECMUT surpassed at this solution quality in less than 2.5 minutes. Moreover, while the GRASP (contiguous) algorithm identified its best solution around 5 hours, it searched for an additional 15 hours without improving upon that solution. In contrast, both ECMUT and PRCRX+ECMUT exhibited continual improvement. The PRCRX+ECMUT algorithm continued to identify better solutions for the rest of the hour that it ran, at which time it identified a solution with a fitness value of 0.0098, shown on the line labeled “PRCRX+ECMUT best,” with notably impressive result on both of the fitness components.

The iterations per second for PRCRX+ECMUT are somewhat elevated due to PRCRX’s rapid convergence, which quickly leads to a homogeneous population. This then affects the subsequent computations because the distance between solutions becomes small. Since most of the solutions in the population are feasible, there was no need to perform feasibility improvement. As a result, iterations accumulated without PRCRX and the feasibility improvement from ECMUT. The iteration speed measured at the time of surpassing GRASP (contiguous) is a more accurate gauge at 5.19 iterations per second, indicating that PRCRX is 17 times slower than ECMUT, compared to ECMUT’s 86.08 iterations per second. This is not unusual since the length of the path for a crossover operation is often much longer than the length of an ejection chain for a mutation operation.

We further examine the impact of PRCRX on the evolutionary process by sampling new solutions generated in both the ECMUT and the PRCRX+ECMUT runs. The change in fitness for the feasible elite solutions is shown in Figure 8. At the start of the EA, PRCRX is somewhat disruptive. PRCRX sufficiently diversified the search so that the first feasible solution occurred later than it did in the ECMUT run. Shortly thereafter, however, with more feasible solutions in the population, the path relinking process became highly effective. Several apparent fitness improvements
Table 2: Performance comparison.

<table>
<thead>
<tr>
<th></th>
<th>Solution quality</th>
<th>Cost</th>
<th>Cost per improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best fitness</td>
<td>Time (in seconds)</td>
<td>Improvements</td>
</tr>
<tr>
<td></td>
<td>Competitiveness</td>
<td>Iterations</td>
<td>per second</td>
</tr>
<tr>
<td></td>
<td>Equal population</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulated Annealing</td>
<td>0.1237</td>
<td>1,489.12</td>
<td>1,472</td>
</tr>
<tr>
<td>Greedy</td>
<td>0.0980</td>
<td>3,817.91</td>
<td>186,619</td>
</tr>
<tr>
<td>Tabu</td>
<td>0.0984</td>
<td>1,968.94</td>
<td>92,659</td>
</tr>
<tr>
<td>GRASP (default)</td>
<td>0.0980</td>
<td>3,910.87</td>
<td>186,619</td>
</tr>
<tr>
<td>GRASP (contiguous)</td>
<td>0.0411</td>
<td>19,140.91</td>
<td>320,386</td>
</tr>
<tr>
<td>ECMUT snapshot</td>
<td>0.0410</td>
<td>728.21</td>
<td>62,687</td>
</tr>
<tr>
<td>ECMUT best</td>
<td>0.0362</td>
<td>2,662.03</td>
<td>250,712</td>
</tr>
<tr>
<td>PRCRX+ECMUT snapshot</td>
<td>0.0383</td>
<td>148.96</td>
<td>774</td>
</tr>
<tr>
<td>PRCRX+ECMUT best</td>
<td>0.0098</td>
<td>3,558.72</td>
<td>351,949</td>
</tr>
</tbody>
</table>
result from random restarts, which keep the top 10% solutions in the current population and inject the other 90% with random solutions.

5.2.1. PRCRX vs. ECMUT

To examine the difference in solution quality between PRCRX and ECMUT, we reconfigured the EA so that PRCRX and ECMUT take the same parent solutions from the population. Figure 9 shows the solution fitness (feasible and infeasible) from the first two thousand iterations. Here, PRCRX outperformed ECMUT 91.65% of the time. In a broader sampling of 20,000 iterations, PRCRX produced a better solution quality 99.16% of the time. Two particular patterns are evident. First, fitness improves slowly with ECMUT, which is to be expected given that it embodies only small movements. Second, PRCRX produced rapid population convergence (shown in the right part of the orange curve) and was not able to progress much further until the first random restart. Hence the performance of ECMUT is comparable with PRCRX at the beginning of the search, but this pattern flips with fitness improvement. Interestingly, around the 400th iteration, there is an obvious and significant fitness improvement phase, which indicates a pivot point that is often described as the phase transition point in an optimization process.
Figure 9: Performance comparison: PRCRX vs. ECMUT in the first 2000 iterations. For minimization problems, lower fitness values indicate better solution quality.

Figure 10: Performance comparison: PRCRX and the basic overlay+expansion. Both take the same input. Fitness difference between the input and output solution is plotted. Negative difference values indicate fitness improvement.
5.2.2. PRCRX vs. Basic Overlap-based Recombination

Overlapping is an effective way to combine two spatial solutions from which one generates a set of connected components of finer granularity for recombination. Based on the resulting overlap, different methods can be developed to generate new solutions. In this experiment, we compare our path relinking-based crossover with a simple overlap-based recombination method, which simply expands the \(k\) seed groups into \(k\) zones. The results are shown in Figure 10 where the fitness change from the same parent solutions is plotted. We can see from the figure that PRCRX is able to effectively improve the fitness of the population, with the main source of fitness improvement originating from the path relinking process, while the basic overlay+expansion crossover procedure is not effective.

5.3. Path analysis

Figure 11 plots the solution distance, path length, and the location offset of the best solution found on the path for the first 2,000 PRCRX calls in a run. Path 1, from \texttt{PRCRX()}, and the optimized path 2, from \texttt{PATH\_OPTIMIZE()}, display similar patterns. First, notice the sudden drop of solution distance and path length around iteration 1,200. This may indicate a pivot point at which the population becomes more homogeneous. It may also be the turning point from a diversified search to an intensified search. Second, the gap between solution distance and path length exists throughout the test runs, which indicates that spatial constraints may be inhibiting the walk from the source solution, \(S\), to the target solution, \(T\). Third, in PRCRX, since the better of the two parent solutions is chosen as the target solution, \(T\), one might intuit that the best solution should be
Table 3: **Weak scalability**, which measures how long it takes (in seconds) to reach multiple fitness thresholds using different number of processors in parallel runs. A measure is left empty if a threshold was not reached within an hour.

<table>
<thead>
<tr>
<th>$2 \times np$</th>
<th>1000</th>
<th>750</th>
<th>500</th>
<th>250</th>
<th>200</th>
<th>150</th>
<th>100</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>77.05</td>
<td>93.38</td>
<td>158.19</td>
<td>158.19</td>
<td>278.02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>13.26</td>
<td>78.78</td>
<td>119.90</td>
<td>204.28</td>
<td>251.64</td>
<td>563.06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>76.46</td>
<td>76.46</td>
<td>125.61</td>
<td>188.10</td>
<td>253.12</td>
<td>995.91</td>
<td>2043.65</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>25.99</td>
<td>67.45</td>
<td>154.85</td>
<td>242.60</td>
<td>377.14</td>
<td>921.40</td>
<td>1769.73</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>0.66</td>
<td>88.95</td>
<td>124.24</td>
<td>165.59</td>
<td>200.18</td>
<td>368.39</td>
<td>846.78</td>
<td>998.98</td>
</tr>
</tbody>
</table>

*np* = number of processors

found closer to $T$. However, the location of the best solution (green points) along the path did not seem to follow a pattern, suggesting that the distribution of better solutions is more dispersed.

We also examined whether the best solution tends to emanate from path 1 or 2 before the first random restart. On average, 68.83% of the best solutions originate from path 2, which seems intuitive because path 2 is built upon path 1. However, 31.17% of the best solutions still arise from path 1, indicating that the exploration of path 1 in `PATH_OPTIMIZE()` may be affected by spatial constraints which inhibit further walk on path 1.

### 5.4. PRCRX in a Parallel Computing Environment

We now turn to examining the performance of our algorithm in a parallel computing environment where each processor evolves an independent EA but solution exchanges are introduced via asynchronous migration among EA processes running in parallel [29]. The parallel computing enables an enormous global EA population. This experiment investigates the computational performance of SEEC in such environment. We ran five different experiments with SEEC on ROGER, where each time, we varied the number of processors utilized (using 10, 20, 40, 80, and 160 processor cores). These results are shown in Table 3. Each computing node utilized 10 MPI processes for a total of one hour. We conducted weak scaling tests to measure the capability of our parallel implementation to perform more effective numerical work when more computing power is enlisted. As we increase the number of processors, the size of the global population increases as well. We measured the time taken to reach multiple fitness thresholds by using different numbers of processors. Overall, as we employ more processors, SEEC was able to reach tighter fitness thresholds. Within the same tight thresholds (150, 100, 80), utilizing additional processors was associated with reduced computing time. As is normal for randomized algorithms, there are some variations in
Figure 12: **Parallel EA convergence.**

this pattern. For example, the run with 20 processors exhibited particularly impressive performance.

Figure 12 illustrates fitness change over time. Compared to the sequential runs, the parallel runs are more effective, but also converged more quickly to a homogeneous population. While we observe this effect in the runs using 10 and 20 processors, our random restart implementation ameliorates this effect in the runs with more processors where steady improvement was observed.

### 6. Conclusion

An effective EA for spatial optimization must leverage information from the underlying spatial configuration to guide the solution search while adhering to the fundamental principles underlying crossover and mutation operators. Our approach preserves the features of EA operators while extending their reach with a novel spatially explicit evolutionary computation approach. The classic crossover and mutation operators are replaced with PRCRX, a spatially cognizant path relinking operator, and ECMUT, an ejection chain heuristic operator, which embed guided spatial moves within the underlying non-linear decision space. SEEC is a novel EA approach with general spatial crossover and mutation operators systematically designed to preserve contiguity in decision space traversal for effective large-scale spatial partitioning.
PRCRX and ECMUT satisfy spatial constraints and generate new feasible solutions by chaining random spatial moves. The performance gain over the basic overlap-expansion technique is substantial. While overlapping combines two solutions, it is unclear how the intersection of the shapes at different locations preserves the desirable components of the parent solutions. Path re-linking, on the other hand, leverages overlapping to obtain seeding groups, and bridges parent solutions through incrementally constructed paths. The recombination occurs along the path, allowing a controlled search for new solutions. Our spatial data structures and routines result in convincing algorithmic and computational performance for large-scale spatial optimization.

Our work can be extended in a number of ways. Efficiency can be enhanced whenever early convergence can be avoided. Here, however, one must find a way to overcome the computational expense involved with detecting population homogeneity. Even the simple binary distance measure on $p$ chromosomes requires the building of a $p \times p$ matrix and takes $O(p^2 \times n)$ to compute, where $n$ is the number of variables. Since solution distance and path length appear to be related to population homogeneity, one might explore the effect of separating the target solution set from the population and explicitly controlling solution distance among the target solutions. In the parallel EA implementation, a distributed intensification and diversification protocol could be developed to enhance search performance by improving coordination in the evolution of the global population.

Our spatial recombination approach flexibly allows further investigation and development of effective recombination strategies. The abstraction of spatial neighborhood functions and the re-linking and chaining mechanisms provide the basic building blocks for exploring different search orders at different granularities. Additional spatial elements can be modularly fused into the heuristic search framework.

Acknowledgements

This material is based in part upon work supported by the National Science Foundation (NSF) under grant SES–1725418. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not reflect the views of NSF. Computational experiments in this research used the Blue Waters sustained-petascale computing resources, which is supported by the NSF (Grants OCI–0725070 and ACI–1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercom-
puting Applications. The performance study used the ROGER supercomputer at the University of Illinois, which is supported by NSF (Grant 1429699). We are grateful for helpful discussions and suggestions from Ximing Cai, Sara McLafferty, Kiumars Soltani, Shaowen Wang, and Dan-dong Yin.
REFERENCES

References


REFERENCES


