applied sciences

MDPI

Article

Cycle Mutation: Evolving Permutations via Cycle Induction

Vincent A. Cicirello

Computer Science, Stockton University, 101 Vera King Farris Dr, Galloway, NJ 08205, USA; cicirelv@stockton.edu

Abstract: Evolutionary algorithms solve problems by simulating the evolution of a population of candidate solutions. We focus on evolving permutations for ordering problems such as the traveling salesperson problem (TSP), as well as assignment problems such as the quadratic assignment problem (QAP) and largest common subgraph (LCS). We propose cycle mutation, a new mutation operator whose inspiration is the well-known cycle crossover operator, and the concept of a permutation cycle. We use fitness landscape analysis to explore the problem characteristics for which cycle mutation works best. As a prerequisite, we develop new permutation distance measures: cycle distance, k-cycle distance, and cycle edit distance. The fitness landscape analysis predicts that cycle mutation is better suited for assignment and mapping problems than it is for ordering problems. We experimentally validate these findings showing cycle mutation's strengths on problems such as QAP and LCS, and its limitations on problems such as the TSP, while also showing that it is less prone to local optima than commonly used alternatives. We integrate cycle mutation into the open source Chips-n-Salsa library, and the new distance metrics into the open source JavaPermutationTools library.

Keywords: combinatorial optimization; evolutionary algorithms; fitness distance correlation; fitness landscape analysis; genetic algorithms; mutation; permutation cycles; permutation distance

PACS: 02.70.-c; 07.05.Mh; 89.20.Ff

MSC: 05A05; 05C60; 68T05; 68T20; 68W20; 68W50; 90C27; 90C59

check for updates

Citation: Cicirello, V.A. Cycle Mutation: Evolving Permutations via Cycle Induction. *Appl. Sci.* **2022**, *12*, 5506. https://doi.org/10.3390/app12115506

Academic Editor: Giancarlo Mauri

Received: 6 May 2022
Accepted: 27 May 2022
Published: 29 May 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

In an evolutionary algorithm (EA), a problem is solved through the simulated evolution of a population that evolves over many generations. There are many types of EA that mostly differ in the types of problems they solve and in how solutions are represented. For example, a genetic algorithm (GA) [1], the original EA, usually represents a candidate solution to an optimization problem with a vector of bits. Evolution strategies (ESs) [2] focus specifically on real-valued function optimization, utilizing a vector of floating-point values to represent each candidate solution. Genetic programming [3] is an approach to automatic inductive programming, and evolves a population of programs, each of which is typically represented with a tree structure.

This paper focuses on EAs that encode solutions with permutations, often referred to as a permutation-based GA [4–7], while others prefer the more general term EA [8,9] to avoid confusion with a binary-encoded GA. Solutions to some problems are more naturally represented with a permutation than with another representation. The classic example is the traveling salesperson problem (TSP), where a solution is a tour of the cities, and thus can be represented in a straightforward way as a permutation of indexes into a list of cities.

One challenge with designing a permutation EA is deciding which genetic operators to use. This is less of an issue with the classic bit-vector GA or a real-valued ES because it is possible to mutate bits independently of the rest of a bit-vector in a GA, and real-valued alleles in an ES can likewise be mutated independently from the vector as a whole, such as with Gaussian mutation [10] or Cauchy mutation [11]. Within a permutation-based EA, mutation cannot change individual elements independent of the rest of the

permutation. For example, in the permutation $[2, 4, 0, 3, 1]$ of the first five integers, we cannot mutate one value in isolation or it leads to an invalid permutation. Crossover cannot naively exchange parts of parents. For example, if one parent is as above, and the other is $[4, 2, 1, 0, 3]$, exchanging the second and third elements between the parents leads to two invalid permutations, $[2, 2, 1, 3, 1]$ and $[4, 4, 0, 0, 3]$. Thus, for permutations, mutation and crossover must consider the overall structure to ensure a valid encoding.

As a consequence, many mutation and crossover operators exist for permutations. The suitability of each depends upon the characteristics of the permutation that most significantly impacts solution fitness for the problem at hand, such as absolute element positions, relative element positions, or element precedences [12,13]. Only relative positions matter to fitness of a TSP solution. For example, if cities i and j are adjacent in the permutation, then the solution includes the edge (i, j) regardless of where the pair of cities appears. The absolute element positions are most important for other problems, such as the largest common subgraph (LCS). The LCS is an NP-hard [14] optimization problem involving finding a one-to-one mapping between the vertex sets of a pair of graphs to maximize the number of edges of the common subgraph implied by the mapping. As a permutation problem, one holds the vertexes of one graph in a fixed order, and a permutation of the vertexes of the other graph represents a mapping. The absolute index of a vertex in the permutation therefore corresponds to the vertex it is mapped to in the other graph.

Mutation for permutations is most often one of swap mutation, insertion mutation, reversal mutation, or scramble mutation [15]. There are many permutation crossover operators that focus on maintaining different characteristics of the parents, including order crossover [16], non-wrapping order crossover [17], uniform order-based crossover [18], partially matched crossover [19], uniform partially matched crossover [20], precedence preservative crossover [21], edge assembly crossover [22,23], and cycle crossover (CX) [24].

The central aim of this paper is development of a mutation operator for permutations that (a) is characterized by small random perturbations on average, (b) has a large neighborhood size, and (c) is tunable. These properties enable focused search with improved handling of local optima. No such mutation operators currently exist for permutations. For bit-vectors, the standard bit-flip mutation satisfies all of these properties. For example, the bit-flip mutation rate M is usually set to a low value for a small average number of bits flipped, but can be tuned for problems where greater mutation is beneficial, and as long as M is non-zero, the neighborhood includes all other bit-vectors. Similarly, Gaussian mutation in an ES satisfies all of these properties, with a tunable parameter, the standard deviation of the Gaussian. All existing permutation mutation operators have a fixed neighborhood size, which in most cases is small, that depends only on permutation length.

To achieve this aim, we present a new mutation operator called cycle mutation, which relies on the concept of a permutation cycle and is inspired by CX. Rather than operating on two parents as CX does, cycle mutation instead mutates a single member of the population. Thus, it is also applicable to non-population metaheuristics such as simulated annealing (SA) [25–27]. We develop two variations of cycle mutation, Cycle$(kmax)$ and Cycle(α), offering two ways of addressing locally optimal solutions.

To formally demonstrate that cycle mutation achieves the target properties of large neighborhood but small average changes, we conduct a fitness landscape analysis of cycle mutation, and other permutation mutation operators for three NP-hard optimization problems [14], the TSP, the LCS, and the quadratic assignment problem (QAP). The fitness landscape analysis predicts that cycle mutation likely performs well for assignment and mapping problems, such as LCS and QAP, where absolute positions directly affect fitness, but that it may be less well-suited to relative ordering problems such as the TSP. We use fitness distance correlation (FDC) [28], which requires a measure of distance between solutions that corresponds to the operator under analysis. Appropriate measures of distance exist for the operators to which we compare. However, no existing permutation distance

functions are suitable for the new cycle mutation. Therefore, we introduce three new permutation distance measures: cycle distance, k-cycle distance, and cycle edit distance.

To validate the new cycle mutation, we experiment with the LCS, QAP, and TSP, comparing cycle mutation with commonly used permutation mutation operators within a $(1+1)$-EA as well as within SA. The results support the predictions of the fitness landscape analysis, demonstrating the efficacy of cycle mutation for assignment problems such as LCS and QAP, while also showing that cycle mutation is inferior to alternatives for the TSP where relative element positions are more important than absolute locations. The experiments also show that the Cycle(α) variation is especially effective at escaping local optima.

A secondary aim of this research is to enable reproducibility [29], as well as to advance the state of practice. Therefore, we integrate our Java implementation of cycle mutation into the open source Chips-n-Salsa library [30], and cycle distance, k-cycle distance, and cycle edit distance into the open source JavaPermutationTools library [31]. Chips-n-Salsa is a library for stochastic and adaptive local search as well as EAs. JavaPermutationTools is a library for computation on permutations and sequences, with a focus on measures of distance. We also disseminate the source code for the fitness landscape analysis and experiments, as well as the raw and processed experiment data, on GitHub (https://github.com/cicirello/cycle-mutation-experiments, accessed on 26 May 2022).

We begin by introducing necessary background in Section 2. We proceed with our methods in Section 3, including deriving the new cycle mutation and distance metrics, as well as performing the fitness landscape analysis. Results are presented in Section 4. We wrap up with a discussion and conclusions in Section 5 including discussing insights into the situations where the new cycle mutation is likely to excel.

2. Background

This section provides background on common mutation operators for permutations (Section 2.1); permutation cycles (Section 2.2), which is the theoretical basis for the new cycle mutation; the CX operator (Section 2.3) on which the new cycle mutation is based; and NP-hard combinatorial optimization problems (Section 2.4) used later in this article.

2.1. Permutation Mutation Operators

We later compare cycle mutation to the most common permutation mutation operators, including the following. Swap picks two different elements uniformly at random and exchanges their locations within the permutation. All other elements remain in their current positions. Insertion picks an element uniformly at random, removes it from the permutation, and reinserts it at a different position chosen uniformly at random. This has the effect of shifting all elements between the removal and insertion points. Reversal (also known as inversion) reverses the order of a subsequence of the permutation, where the end points are chosen uniformly at random. Scramble (also known as shuffle) randomizes the order of a subsequence of the permutation, where the end points of the subsequence are chosen uniformly at random. Scramble is the most disruptive of these operators.

Prior fitness landscape analyses (e.g., [32]) show that swap strongly dominates when absolute positions are most important to fitness; reversal is best for relative positions with undirected edges, followed by insertion and swap, but reversal performs poorly with directed edges; and insertion is best when element precedences most greatly affect fitness.

2.2. Permutation Cycles

Cycle mutation relies upon the concept of a permutation cycle [33]. Align two permutations such that corresponding positions are vertically adjacent, such as

$$\begin{aligned} p_1 &= [0,1,2,3,4,5,6,7,8,9] \\ p_2 &= [2,3,0,5,6,7,8,9,4,1]. \end{aligned} \tag{1}$$

Consider a directed graph with one vertex for each element. In this example, the hypothetical graph has 10 vertexes. Corresponding positions define directed edges. For example, p_1 and p_2 have 0 and 2 at the beginning, respectively, which implies an edge from vertex 0 to vertex 2. Thus, the directed edges of the graph induced by p_1 and p_2 are

$$\{(0,2),(1,3),(2,0),(3,5),(4,6),(5,7),(6,8),(7,9),(8,4),(9,1)\}. \tag{2}$$

A permutation cycle is a cycle in this graph. Thus, in this example, there are three permutation cycles, consisting of the following sets of vertexes:

$$\{0,2\},\{1,3,5,7,9\},\{4,6,8\}. \tag{3}$$

2.3. Cycle Crossover (CX)

The CX [24] operator creates two children from two parents as follows. It first selects an index into one parent uniformly at random. It computes the permutation cycle for the pair of parents that includes the chosen element. Child c_1 receives the positions of the elements that are in the cycle from p_2 and the positions of the other elements from p_1. Likewise, child c_2 receives the positions of the elements that are in the cycle from p_1 and the positions of the others from p_2. The runtime to apply CX is $\Theta(n)$, where n is the permutation length.

Consider an example where the parents are the permutations of Equation (1), which consist of three permutation cycles (see Equation (3)). The result of CX depends upon the random starting element. If element 0 or 2 begins the cycle, then the children are:

$$\begin{aligned} c_1 &= [2,1,0,3,4,5,6,7,8,9] \\ c_2 &= [0,3,2,5,6,7,8,9,4,1]. \end{aligned} \tag{4}$$

If one of the elements $\{1,3,5,7,9\}$ begins the cycle, then the children are:

$$\begin{aligned} c_1 &= [0,3,2,5,4,7,6,9,8,1] \\ c_2 &= [2,1,0,3,6,5,8,7,4,9]. \end{aligned} \tag{5}$$

Otherwise, if one of the elements $\{4,6,8\}$ begins the cycle, then the children are:

$$\begin{aligned} c_1 &= [0,1,2,3,6,5,8,7,4,9] \\ c_2 &= [2,3,0,5,4,7,6,9,8,1]. \end{aligned} \tag{6}$$

Since the starting element is chosen uniformly at random, larger cycles are chosen with greater probability. In this example, the probability of generating the first pair of children above is 0.2, while the probability of generating the second pair of children is 0.5, and the probability of generating the last pair of children is 0.3.

One characteristic of CX is that every element of each child has its absolute position within the permutation from one or the other of the two parents. In this way, it is particularly well suited to permutation problems where absolute position has greatest effect on fitness, since the children are inheriting absolute element position from the parents.

2.4. Test Problems

We now provide background on the TSP, QAP, and LCS, which are NP-hard combinatorial optimization problems used in the fitness landscape analysis and experiments.

2.4.1. TSP

In the TSP, a salesperson must complete a tour of n cities to minimize total cost, usually distance traveled. The cities are vertexes of a completely connected graph. A tour is a simple cycle that includes all n vertexes. The NP-complete decision variant of the TSP asks whether a tour exists with cost at most C; and the NP-hard optimization problem

seeks the minimum cost tour [14]. The TSP is widely studied and is perhaps the most common combinatorial optimization problem in experimental studies. It has been used in machine learning [34,35], ant colony optimization [36,37], GA [19,24,38,39], other forms of EA [22,23,40], and other metaheuristics [41–45]. There are variations of the problem, such as the asymmetric TSP (ATSP), where the cost of using an edge differs depending upon the direction of travel along the edge [46,47]. Some variations include problem domain characteristics, such as in delivery route planning [43] and wireless sensor networks [48].

2.4.2. QAP

The formal definition of the QAP is as follows. We are given an n by n cost matrix C, and an m by m distance matrix D, such that $m \geq n$. The NP-complete decision version of the problem [14] asks the question, for a given bound B: Does there exist a one-to-one function $f : \{0, 1, \ldots, n-1\} \to \{0, 1, \ldots, m-1\}$, such that:

$$\sum_{i=0}^{n-1} \sum_{\substack{j=0 \\ j \neq i}}^{n-1} C_{i,j} D_{f(i),f(j)} \leq B? \tag{7}$$

The NP-hard optimization version of the QAP is to find the one-to-one function $f : \{0, 1, \ldots, n-1\} \to \{0, 1, \ldots, m-1\}$ that minimizes

$$\sum_{i=0}^{n-1} \sum_{\substack{j=0 \\ j \neq i}}^{n-1} C_{i,j} D_{f(i),f(j)}. \tag{8}$$

Most authors consider the case when $n = m$. This problem is naturally represented with permutations. When $n = m$, a solution (i.e., the one-to-one function f) is simply represented as a permutation of the integers $\{0, 1, \ldots, n-1\}$. In the more general case, a solution is the first n integers in a permutation of the integers $\{0, 1, \ldots, m-1\}$. The QAP is an especially challenging NP-hard optimization problem, where you most often find experimental studies utilizing what seems to be rather small instances (e.g., $n < 50$). A wide variety of EA, metaheuristics, and heuristic approaches have been proposed for the QAP [49–55].

2.4.3. LCS

The LCS problem [14] is closely related to the subgraph isomorphism problem. In the LCS problem, we are given graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, with vertex sets V_1 and V_2, and edge sets E_1 and E_2. In the optimization variant of the problem, we must find the graph $G_3 = (V_3, E_3)$, such that G_3 is isomorphic to a subgraph of G_1 and G_3 is isomorphic to a subgraph of G_2, and such that the cardinality of edge set $|E_3|$ is maximized. This problem is NP-hard. The NP-complete decision variant of the problem asks whether there exists such a graph G_3 such that $|E_3| \geq K$ for some threshold K.

If $|V_1| = |V_2|$, then a solution is represented by a permutation of $\{0, 1, \ldots, |V_1| - 1\}$, and more generally the solution is represented by the first $\min(|V_1|, |V_2|)$ integers of a permutation of $\{0, 1, \ldots, \max(|V_1|, |V_2|) - 1\}$. Without loss of generality, assume that $|V_1| \leq |V_2|$. For permutation p of integers $\{0, 1, \ldots, |V_2| - 1\}$, the number of edges $|E_3|$ in the common subgraph implied by such a permutation is then computed as

$$|E_3| = \sum_{(u,v) \in E_1} \begin{cases} 1 & \text{if } (p(u), p(v)) \in E_2 \\ 0 & \text{if } (p(u), p(v)) \notin E_2, \end{cases} \tag{9}$$

where $p(u)$ means the element in position u of permutation p. This is most efficiently implemented if the smaller graph, G_1, is represented with adjacency lists or by a simple list of ordered pairs as the edge set; and if the larger graph G_2 is represented with an adjacency matrix to enable constant time checks for existence of edges.

Prior approaches to the LCS problem include GA [20], hill-climbing [56], and heuristic approaches [57–59]. There are applications of the LCS problem in computer-aided engineering [56], software engineering [59], protein molecule comparisons [58], integrated circuit design [57,60], natural language processing [61], and cybersecurity [62], among others.

3. Methods

To achieve the objective of a tunable mutation operator with a large neighborhood but small average changes, we derive two forms of the new cycle mutation in Section 3.1.

To gain an understanding of the topology of fitness landscapes associated with cycle mutation, we use a variety of fitness landscape analysis tools. This requires measures of permutation distance corresponding to the mutation operators. Ideally, these should be edit distances where the mutation operator is the edit operation. Edit distance is defined as the minimum cost of the edit operations necessary to transform one structure into the other, and originated within the context of string distance [63,64]. There are many permutation distance measures available in the literature [12,13,31,65–68], including several edit distances. However, none of these are suitable for characterizing the distance between permutations within the context of cycle mutation. Therefore, in Section 3.2, we derive new measures of permutation distance: cycle distance, cycle edit distance, and k-cycle distance.

We then proceed with the fitness landscape analysis in Section 3.3. We derive the diameter of the fitness landscapes for both forms of cycle mutation, as well as for other common permutation mutation operators. The diameter of a fitness landscape is the distance between the two furthest points, where distance is the minimum number of applications of the operators to transform one point to the other. Thus, diameter directly relates to neighborhood size, where larger neighborhood corresponds to smaller diameter, providing a means to quantify our objective of a mutation with large neighborhood. The fitness landscape analysis then utilizes FDC, which is the Pearson correlation coefficient between the fitness of solutions, and the distance to the nearest optimal solution [28]. The FDC analysis uses the TSP, LCS, and QAP problems, so that we can explore the behavior of the cycle mutation operator on a variety of permutation problems. To directly address the objective of mutation with small average changes, the fitness landscape analysis also uses search landscape calculus [32], which examines the average local rate of fitness change.

3.1. Cycle Mutation

We present two variations of cycle mutation. Both forms mutate permutation p by inducing a permutation cycle of length k. The primary difference between the two versions is in how k is chosen. We provide notation and operations shared by the two variations in Section 3.1.1, followed by the presentations of the two versions, Cycle($kmax$) in Section 3.1.2 and Cycle(α) in Section 3.1.3, and finally a comparison of the asymptotic runtime of the new cycle mutation with commonly used mutation operators in Section 3.1.4.

3.1.1. Shared Notation and Operations

Let p be a permutation of length n, such that $p(i)$ is the element at index i. Assume that indexes into p are 0-based (i.e., valid indexes are $\{0, 1, \ldots, n-1\}$), as are array indexes. Algorithm 1 shows pseudocode for the core operation of both variations of cycle mutation. Namely, it induces a cycle in the given permutation p from an array of indexes into p.

Algorithm 1 CreateCycle(p, indexes)

```
temp ← p(indexes[0])
for i = 1 to n − 1 do
    p(indexes[i − 1]) ← p(indexes[i])
p(indexes[n − 1]) ← temp
```

As an example of its behavior, consider the following permutation:

$$p = [2, 6, 0, 5, 3, 8, 7, 9, 4, 1]. \tag{10}$$

Now consider the following array of indexes into p:

$$\text{indexes} = [3, 7, 1, 4]. \tag{11}$$

Executing CreateCycle$(p, \text{indexes})$ will produce the permutation p' such that $p'(3) = p(7)$, $p'(7) = p(1)$, $p'(1) = p(4)$, and $p'(4) = p(3)$, resulting in the following:

$$p' = [2, 3, 0, 9, 5, 8, 7, 6, 4, 1]. \tag{12}$$

The runtime of CreateCycle is linear in the induced cycle length.

One of the steps of cycle mutation requires sampling k random indexes into permutation p without replacement. Algorithm 2 shows our sampling approach, which uses one of three algorithms depending on the value of k relative to the permutation length n.

Algorithm 2 Sample(n, k)

```
1: if k ≥ n/2 then
2:     return ReservoirSample(n, k)
3: else if k ≥ √n then
4:     return PoolSample(n, k)
5: else
6:     return InsertionSample(n, k)
```

When $k \geq \frac{n}{2}$, we use Vitter's reservoir sampling algorithm [69] (line 2 of Algorithm 2), which has a runtime of $O(n)$ and utilizes $O(n - k)$ random numbers. When $\sqrt{n} \leq k < \frac{n}{2}$, we use Ernvall and Nevalainen's sampling algorithm [70], which we refer to as PoolSample in line 4 of Algorithm 2, and which also has a runtime of $O(n)$, but requires $O(k)$ random numbers. Asymptotic runtime of both of these options is $O(n)$, but since random number generation is a costly operation with very significant impact on the runtime of an EA [71], our approach chooses the sampling algorithm that requires fewer random numbers.

When $k < \sqrt{n}$, we use what we believe is a brand new sampling algorithm: insertion sampling (Algorithm 2, line 6). Insertion sampling's runtime is $O(k^2)$, and requires $O(k)$ random numbers. Since runtime increases quadratically in k, insertion sampling is only a good choice when k is very small relative to n. Pseudocode for insertion sampling is in Algorithm 3. To ensure a duplicate-free result, it maintains a sorted list of the integers selected thus far, and inserts into that list in a way similar to insertion sort. The Rand(a, b) in Algorithm 3 is a uniform random variable over the interval $[a, b]$, inclusive.

Algorithm 3 InsertionSample(n, k)

```
result ← a new array of length k
for i = 0 to k − 1 do
    v ← Rand(0, n − i − 1)
    j ← k − i
    while j < k and v ≥ result[j] do
        v ← v + 1
        result[j − 1] ← result[j]
        j ← j + 1
    result[j − 1] ← v
return result
```

The composite sampling algorithm of Algorithm 2 runs in $O(\min(n, k^2))$ time and requires $O(\min(k, n - k))$ random numbers. We integrated this composite sampling algo-

rithm, the new insertion sampling algorithm, as well as our implementations of reservoir sampling and pool sampling into an open source Java library $\rho\mu$ (https://github.com/cicirello/rho-mu, accessed on 26 May 2022), independent of the application to EA of this article.

3.1.2. Cycle($kmax$)

In the first version of cycle mutation (Algorithm 4), called Cycle($kmax$), the induced cycle length k is uniformly random from the interval $[2, kmax]$, such that $kmax \geq 2$.

Algorithm 4 CycleMutation($p, kmax$)

1: $k \leftarrow$ Rand$(2, kmax)$
2: indexes $\leftarrow$ Sample(n, k)
3: Shuffle(indexes)
4: CreateCycle(p, indexes)

It first selects the cycle length k in $O(1)$ time. Next, k indexes are sampled uniformly at random without replacement (line 2) with a call to Sample(n, k) of Algorithm 2. Since $k \leq kmax$, the worst-case runtime of that step is $O(\min(n, kmax^2))$. This array of indexes is randomized (line 3), with a worst-case cost of $O(kmax)$. Finally, a cycle is induced from the randomized list of k indexes with a call to the CreateCycle of Algorithm 1 in line 4, which also costs $O(kmax)$ time in the worst case. Thus, the worst-case runtime is $O(\min(n, kmax^2))$, since the most costly step is sampling the indexes. The average case is also $O(\min(n, kmax^2))$ since the average cycle length $\bar{k} = \frac{kmax+2}{2}$ is proportional to $kmax$.

Cycle($kmax$) limits the induced permutation cycle to a predetermined maximum length, with all cycle lengths up to $kmax$ equally likely. This enables tuning the size of the local neighborhood, with lower $kmax$ leading to a smaller neighborhood and higher $kmax$ creating a larger neighborhood. Thus, increasing $kmax$ may lead to fewer local optima in the fitness landscape, but would also lead to a more disruptive and less focused search.

3.1.3. Cycle(α)

The second version of cycle mutation, called Cycle(α), maximizes the size of the local neighborhood while retaining the local focus of a small cycle length. Cycle(α) allows any possible cycle length $k \in [2, n]$, but chooses a smaller cycle length with higher probability than a greater cycle length. Specifically, the probability of choosing cycle length k is proportional to α^{k-2}. Thus, the probability $P(k)$ of choosing cycle length k is

$$P(k) = \frac{\alpha^{k-2}}{\sum_{i=2}^{n} \alpha^{i-2}} = \frac{\alpha^{k-2}(1-\alpha)}{1-\alpha^{n-1}}. \tag{13}$$

The α is a parameter of the operator such that $0 < \alpha < 1$. The nearer α is to 0, the more probabilistic weight is placed upon lower values of k relative to higher values.

From this, we can derive a mathematical transformation from uniform random number $U \in [0.0, 1.0)$ to corresponding cycle length k. Choose k according to the following:

$$k = \begin{cases} 2 & \text{if } 0 \leq U < P(2) \\ j & \text{if } \sum_{i=2}^{j-1} P(i) \leq U < \sum_{i=2}^{j} P(i). \end{cases} \tag{14}$$

This is equivalent to

$$k = \underset{j \in [2,n]}{\operatorname{argmin}} \left\{ \sum_{i=2}^{j} P(i) \right\} \text{ subject to } \left\{ \sum_{i=2}^{j} P(i) > U \right\}. \tag{15}$$

Substituting Equation (13) into the constraint and simplifying arrives at

$$k = \underset{j \in [2,n]}{\operatorname{argmin}} \left\{ \sum_{i=2}^{j} P(i) \right\} \text{ subject to } \left\{ \frac{1-\alpha^{j-1}}{1-\alpha^{n-1}} > U \right\}. \tag{16}$$

Solve the constraint for j to derive

$$k = \underset{j \in [2,n]}{\operatorname{argmin}} \left\{ \sum_{i=2}^{j} P(i) \right\} \text{ subject to } \left\{ j > \frac{\log(1-(1-\alpha^{n-1})U)}{\log(\alpha)} + 1 \right\}. \tag{17}$$

Since the summation inside the argmin increases as j increases, this is equivalent to

$$k = \min_{j \in [2,n]} \{j\} \text{ subject to } \left\{ j > \frac{\log(1-(1-\alpha^{n-1})U)}{\log(\alpha)} + 1 \right\}. \tag{18}$$

Finally, we compute k from U via

$$k = 2 + \left\lceil \frac{\log(1-(1-\alpha^{n-1})U)}{\log(\alpha)} \right\rceil. \tag{19}$$

Since α is a parameter that does not change during the run, and since the permutation length n is likewise fixed based on the problem instance we are solving, the $(1-\alpha^{n-1})$ and the $\log(\alpha)$ are constants that can be computed a single time at the start of the run.

Algorithm 5 shows pseudocode of Cycle(α). The worst-case runtime is $O(n)$, which occurs when the random k equals n, leading lines 3–4 to cost $O(n)$. The worst case is a rare occurrence. Since mutation is applied a very large number of times during an EA, it is more meaningful to examine the average runtime of mutation. To determine the average runtime, we must first compute the expected cycle length $E[k]$ as follows:

$$\begin{aligned} E[k] &= \sum_{k=2}^{n} kP(k) \\ &= \frac{(1-\alpha)}{1-\alpha^{n-1}} \sum_{k=2}^{n} k\alpha^{k-2} \\ &= \frac{2-\alpha+n\alpha^{n}-n\alpha^{n-1}-\alpha^{n-1}}{(1-\alpha)(1-\alpha^{n-1})} \\ &\leq \lim_{n\to\infty} \frac{2-\alpha+n\alpha^{n}-n\alpha^{n-1}-\alpha^{n-1}}{(1-\alpha)(1-\alpha^{n-1})} \\ &= \frac{2-\alpha}{1-\alpha}. \end{aligned} \tag{20}$$

The expected cycle lengths for Cycle(0.25), Cycle(0.5), and Cycle(0.75) are $E[k] \leq 2\frac{1}{3}$, $E[k] \leq 3$, and $E[k] \leq 5$, respectively. The average runtime of Cycle(α) is therefore $O(\min(n, (\frac{2-\alpha}{1-\alpha})^2))$ due to the call to Sample(n,k) in line 2 of Algorithm 5.

Algorithm 5 CycleMutation(p, α)

1: $k \leftarrow 2 + \left\lceil \frac{\log(1-(1-\alpha^{n-1})U)}{\log(\alpha)} \right\rceil$
2: indexes $\leftarrow$ Sample(n,k)
3: Shuffle(indexes)
4: CreateCycle$(p,$ indexes$)$

3.1.4. Asymptotic Runtime Summary

Table 1 summarizes the asymptotic runtime of cycle mutation and common mutation operators. Swap mutation is a constant time operation, while the worst -case runtime of insertion, reversal, scramble, and Cycle(α) is linear in the permutation length n. The worst case for Cycle($kmax$) is between these extremes, depending upon $kmax$.

Since mutation is computed a very large number of times during an EA, average runtime is more meaningful. The average runtime of insertion, reversal, and scramble is $O(n)$. The average runtime of Cycle(α) depends upon α. However, other than values of α very near 1.0, the average runtime is essentially a constant. Although the runtime of swap is constant, and that of cycle mutation is very nearly constant depending on α or $kmax$, they are not strictly superior to the linear time operators for all problems. Some problem characteristics may lead to superior performance with fewer applications of one of the linear time mutation operators than if the constant time swap was instead used.

Table 1. Asymptotic runtime of cycle mutation and common permutation mutation operators.

Mutation Operator	Worst Case	Average Case
Cycle($kmax$)	$O(\min(n, kmax^2))$	$O(\min(n, kmax^2))$
Cycle(α)	$O(n)$	$O(\min(n, (\frac{2-\alpha}{1-\alpha})^2))$
Swap	$O(1)$	$O(1)$
Insertion	$O(n)$	$O(n)$
Reversal	$O(n)$	$O(n)$
Scramble	$O(n)$	$O(n)$

3.2. New Measures of Permutation Distance

We now present three new measures of permutation distance: cycle distance (Section 3.2.1), cycle edit distance (Section 3.2.2), and k-cycle distance (Section 3.2.3).

3.2.1. Cycle Distance

As a first step toward a distance measure appropriate for use in analyzing fitness landscapes associated with the Cycle(α) form of cycle mutation, we present cycle distance. Cycle(α) mutates a permutation by inducing a cycle whose length is only limited by the permutation length itself. Therefore, cycle distance is the count of the number of non-singleton cycles. We can define the cycle distance between permutations p_1 and p_2 as

$$\delta(p_1, p_2) = \text{CycleCount}(p_1, p_2) - \text{FixedPointCount}(p_1, p_2), \tag{21}$$

where CycleCount is the number of permutation cycles, and FixedPointCount is the number of singleton cycles or fixed points, which is a cycle of length one (i.e., a point where both permutations contain the same element). We compute cycle distance in $O(n)$ time.

Cycle distance is a semi-metric, since it satisfies all of the metric properties except for the triangle inequality. First, it obviously satisfies non-negativity since $\delta(p_1, p_2)$ clearly cannot be negative since there cannot be a negative number of non-singleton permutation cycles. Second, it is obvious that $\delta(p_1, p_2) = \delta(p_2, p_1)$, and is thus symmetric. Third, it satisfies the identity of indiscernibles as follows. When $p_1 = p_2 = p$, we have

$$\delta(p, p) = \text{CycleCount}(p, p) - \text{FixedPointCount}(p, p) = n - n = 0, \tag{22}$$

since the cycle count is n (i.e., n singleton cycles), and thus the number of fixed points is also n. In the other direction, we have

$$\delta(p_1, p_2) = 0 \implies \text{CycleCount}(p_1, p_2) = \text{FixedPointCount}(p_1, p_2) \implies p_1 = p_2, \tag{23}$$

since the only way that every cycle is a fixed point is if p_1 and p_2 are identical.

To demonstrate the violation of the triangle inequality, consider the permutations:

$$\begin{aligned} p_1 &= [0,1,2,3,4,5,6,7,8,9] \\ p_2 &= [1,0,3,2,5,4,7,6,9,8] \\ p_3 &= [0,3,2,5,4,7,6,9,8,1]. \end{aligned} \tag{24}$$

Observe $\delta(p_1, p_2) = 5$ since every consecutive pair of elements in p_1 is swapped in p_2, creating five cycles of length two. All even-numbered elements are fixed points for p_1 and p_3, and all odd-numbered elements form a single cycle. That is, keep the even elements fixed and cycle the odd elements to the left within p_1 to obtain p_3. Thus, $\delta(p_1, p_3) = 1$. Finally, inspect p_3 and p_2 to note that if we cycle all of the elements of p_3 one position to the right, we obtain p_2. Therefore, $\delta(p_3, p_2) = 1$. Thus, since $\delta(p_1, p_2) > \delta(p_1, p_3) + \delta(p_3, p_2)$, cycle distance does not satisfy the triangle inequality, and is only a semi-metric.

The diameter of the space of permutations S_n of length n given a measure of distance δ is the maximal distance between points in that space. Define the diameter $D(n, \delta)$ as

$$D(n, \delta) = \max_{p_1, p_2 \in S_n} \{\delta(p_1, p_2)\}. \tag{25}$$

Since each non-singleton cycle contributes one to cycle distance, independent of cycle length, the maximum case is when the number of non-singleton cycles is maximized. The smallest non-singleton cycle is length two. The maximum cycle distance therefore occurs when there are $[n/2]$ cycles of length two, leading to a diameter of

$$D(n, \delta) = \left[\frac{n}{2}\right]. \tag{26}$$

3.2.2. Cycle Edit Distance

Although cycle distance relates to the Cycle(α) operator, it is not actually an edit distance with Cycle(α) as the edit operation. However, we can utilize it to define such an edit distance. To define cycle edit distance as the minimum number of induced permutation cycles necessary to transform p_1 into p_2, we can formally define cycle edit distance as

$$\delta_e(p_1, p_2) = \begin{cases} 0 & \text{if } p_1 = p_2 \\ 1 & \text{if } \delta(p_1, p_2) = 1 \\ 2 & \text{if } \delta(p_1, p_2) > 1, \end{cases} \tag{27}$$

where $\delta(p_1, p_2)$ is cycle distance (Equation (21)). If there is exactly one non-singleton cycle, we can trivially transform it to all fixed points by cycling the elements of that cycle. If there are two or more non-singleton cycles, there exists a cycle of the union of the elements of those cycles that will merge all of them into a single larger cycle, possibly producing some fixed points. See Equation (24) for an example. Thus, one cycle edit operation merges all non-singleton cycles, and a second cycle edit transforms it into all fixed points. We can compute cycle edit distance in $O(n)$ time since we can compute cycle distance in $O(n)$ time.

Cycle edit distance satisfies all of the metric properties as follows. From Equation (27), it is trivial to confirm non-negativity, symmetry, and the identity of indiscernibles. Without loss of generality, assume that p_1, p_2, and p_3 are all different. Thus, $\delta_e(p_1, p_3) + \delta_e(p_3, p_2) \geq 1 + 1 \geq 2$, implying $\delta_e(p_1, p_2) \leq \delta_e(p_1, p_3) + \delta_e(p_3, p_2)$ since $\delta_e(p_1, p_2) \leq 2$ by definition. Thus, cycle edit distance satisfies the triangle inequality, and it is therefore a metric.

Multiple non-singleton cycles can only exist if permutation length $n \geq 4$, and permutations of length one must be identical. Thus, the diameter of cycle edit distance is

$$D(n, \delta_e) = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 & \text{if } 1 < n \leq 3 \\ 2 & \text{if } 3 < n. \end{cases} \tag{28}$$

3.2.3. K-Cycle Distance

Cycle distance and cycle edit distance assume that arbitrary length cycles can be induced in a single operation, which is needed for Cycle(α) since it does not restrict the induced cycle length. However, this is not suitable when characterizing fitness landscapes for the Cycle($kmax$) mutation operator, which does limit the cycle length to $kmax$.

We now define k-cycle distance, which limits operations to inducing cycles of lengths $\{2, 3, \ldots, k\}$. The k-cycle distance is not an edit distance. Instead, it is a weighted sum over the cycles of the pair of permutations, where the weight of each cycle is the minimum number of induced cycles of length at most k necessary to transform the cycle to all fixed points. That is, k-cycle distance does not consider operations that span multiple cycles.

A cycle of length $c \leq k$ can be transformed to c fixed points by inducing a cycle of length c. If the cycle length $c > k$, a sequence of cycles can derive c fixed points. For k-cycle distance, iteratively induce cycles of length k until the remaining cycle length is at most k, completing the transformation with one final cycle operation. Each intermediate cycle creates $k - 1$ fixed points. Consider the following example beginning with permutations:

$$\begin{aligned} p_1 &= [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] \\ p_2 &= [2, 3, 1, 5, 6, 7, 8, 9, 4, 0]. \end{aligned} \tag{29}$$

There are two non-singleton permutation cycles in this example with element sets:

$$\{1, 3, 5, 7, 9, 0, 2\}, \{4, 6, 8\}. \tag{30}$$

If we are computing 3-cycle distance, then we consider cycle mutations of lengths 2 and 3. The cycle with elements $\{4, 6, 8\}$ can thus be transformed into all fixed points with a single cycle mutation since its length is less than or equal to 3 to obtain

$$\begin{aligned} p_1 &= [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] \\ p_2 &= [2, 3, 1, 5, 4, 7, 6, 9, 8, 0]. \end{aligned} \tag{31}$$

The longer cycle $\{1, 3, 5, 7, 9, 0, 2\}$ requires a sequence of three operations. The first two produce $k - 1 = 2$ fixed points each with one final cycle mutation for the remaining three elements. First, cycle elements 3, 5, and 7, resulting in fixed points for elements 3 and 5:

$$\begin{aligned} p_1 &= [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] \\ p_2 &= [2, 7, 1, 3, 4, 5, 6, 9, 8, 0]. \end{aligned} \tag{32}$$

A cycle of $\{1, 7, 9, 0, 2\}$ remains. Cycle elements 7, 9, and 0, creating fixed points 7 and 9:

$$\begin{aligned} p_1 &= [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] \\ p_2 &= [2, 0, 1, 3, 4, 5, 6, 7, 8, 9]. \end{aligned} \tag{33}$$

A cycle of $\{1, 0, 2\}$ remains, which transforms into fixed points with one final cycle mutation:

$$\begin{aligned} p_1 &= [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] \\ p_2 &= [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. \end{aligned} \tag{34}$$

With this example in mind, we now derive the k-cycle distance, assuming $k \geq 2$. First, given a cycle of length c, the fewest cycle edits of length at most k necessary to transform it to c fixed points is $[(c-1)/(k-1)]$. To compute k-cycle distance, sum this over all cycles, CycleSet(p_1, p_2), of the permutations p_1 and p_2. Therefore, define k-cycle distance by

$$\delta_k(p_1, p_2) = \sum_{\substack{\text{cycle} \in \text{CycleSet}(p_1, p_2) \\ c = |\text{cycle}|}} \left[\frac{c-1}{k-1} \right]. \tag{35}$$

The k-cycle distance can be computed in $O(n)$ time. It is a metric when $k \leq 4$, but it is only a semi-metric for $k \geq 5$. Independent of k, it trivially satisfies non-negativity since the expression within the sum of Equation (35) is never negative; and since CycleSet(p_1, p_2) computes the same set of cycles as CycleSet(p_2, p_1), it is also trivially symmetric. It satisfies the identity of indiscernibles as follows. If $p_1 = p_2 = p$, then there are n singleton cycles, and since the expression within the sum of Equation (35) is 0 when $c = 1$, we have $\delta_k(p, p) = 0$. In the other direction, if $\delta_k(p_1, p_2) = 0$, then all elements in the summation must be 0, and that only occurs with fixed points. Thus, $\delta_k(p_1, p_2) = 0 \implies p_1 = p_2$.

The triangle inequality is violated if $k \geq 5$. Consider this case with $k = 6$:

$$\begin{aligned} p_1 &= [0, 1, 2, 3, 4, 5] \\ p_2 &= [1, 0, 3, 2, 5, 4] \\ p_3 &= [0, 3, 2, 5, 4, 1]. \end{aligned} \tag{36}$$

The 6-cycle distance allows cycle operations of length up to six. Note that $\delta_6(p_1, p_2) = 3$ since there are three cycles of length two; $\delta_6(p_1, p_3) = 1$ since the even-numbered elements are fixed points, and the odd-numbered elements form a single cycle of length three; and $\delta_6(p_3, p_2) = 1$, with a single cycle of length six (i.e., cycle the elements of p_3 one position to the right to obtain p_2). Thus, $\delta_6(p_1, p_3) + \delta_6(p_3, p_2) = 1 + 1 = 2 < \delta_6(p_1, p_2)$. Therefore, 6-cycle distance is only a semi-metric, as is k-cycle distance for any other $k \geq 6$.

We can produce a similar example for the case of $k = 5$ as follows:

$$\begin{aligned} p_1 &= [0, 1, 2, 3, 4, 5] \\ p_2 &= [1, 0, 3, 2, 5, 4] \\ p_3 &= [0, 3, 2, 5, 1, 4]. \end{aligned} \tag{37}$$

As in the previous example, $\delta_5(p_1, p_2) = 3$. Note that $\delta_5(p_1, p_3) = 1$ since $\{0, 2\}$ are fixed points, and there is a single cycle of the elements $\{1, 3, 5, 4\}$; and $\delta_5(p_3, p_2) = 1$, with a fixed point for element 4, and a single cycle of the remaining five elements. In this example, $\delta_5(p_1, p_3) + \delta_5(p_3, p_2) < \delta_5(p_1, p_2)$. Thus, 5-cycle distance is also only a semi-metric.

In general, as many as k non-singleton cycles can be merged into a single larger cycle by cycling k elements, where the cycle edit includes at least one element of each of the merged cycles. This is also true when $k < 5$, but for $k < 5$ the resulting merged cycle is larger than k, requiring multiple cycle edits to transform into all fixed points.

When $k \leq 4$, the k-cycle distance satisfies the triangle inequality, and is a metric; but when $k \geq 5$, it is only a semi-metric. Indeed, when $k \leq 4$, the k-cycle distance is equivalent to an edit distance with cycles of length up to k as the edit operations. For example, 2-cycle distance is equivalent to an existing distance metric known as interchange distance, which is an edit distance with swap as its edit operation, which is a metric.

The diameter of the space of permutations for k-cycle distance depends upon the permutation length n in relation to k. When k is high, distance is maximized by maximizing the number of permutation cycles, which occurs when there are $[n/2]$ cycles of length 2. When k is low, distance is maximized when there is a single cycle of length n, which requires $[(n-1)/(k-1)]$ cycle operations to transform to n fixed points. Thus, the diameter is

$$D(n, \delta_k) = \max\left\{ \left[\frac{n}{2}\right], \left[\frac{n-1}{k-1}\right] \right\}. \tag{38}$$

3.3. Fitness Landscape Analysis

The fitness landscape analysis includes calculation of landscape diameters in Section 3.3.1, FDC analysis in Section 3.3.2, and an analysis of the search landscape calculus in Section 3.3.3. We synthesize the fitness landscape analysis findings in Section 3.3.4.

When an edit distance is required, such as to compute FDC and the diameter, we use cycle edit distance for Cycle(α), and k-cycle distance for Cycle($kmax$). Swap's edit distance is interchange distance, the minimum number of swaps to transform p_1 into p_2:

$$\delta_i(p_1, p_2) = n - \text{CycleCount}(p_1, p_2), \tag{39}$$

where CycleCount(p_1, p_2) is the number of permutation cycles. The edit distance for insertion mutation is known as reinsertion distance, the minimum number of insertion mutations needed to transform p_1 into p_2. It is efficiently computed as [32]:

$$\delta_r(p_1, p_2) = n - |\text{LongestCommonSubsequence}(p_1, p_2)|, \tag{40}$$

where LongestCommonSubsequence(p_1, p_2) is the longest common subsequence. It is not feasible to utilize an edit distance to analyze reversal mutation landscapes, because computing reversal edit distance is NP-hard [72]. Instead we utilize cyclic edge distance [68], which interprets a permutation as a cyclic sequence of edges (e.g., 1 following 2 is treated as an edge between 1 and 2) and counts the edges in p_1 that are not in p_2. For scramble mutation, a trivial edit distance from a permutation to itself is 0, and to any other permutation is 1, since any permutation is reachable by some shuffling of any other.

3.3.1. Fitness Landscape Diameter

Table 2 compares the diameters of fitness landscapes for the various mutation operators. Except where noted, we use the metrics identified above. The diameter of both swap and insertion landscapes is $n - 1$. The maximum distance for swap occurs for a single n element cycle. The maximum case for insertion occurs when one permutation is the reverse of the other. For reversal mutation, we use the exact diameter for a reversal edit distance rather than relying on the surrogate distance measure identified earlier. The diameter of a reversal landscape is $n - 1$, proven by Bafna and Pevzner [73], the proof of which is well beyond the scope of this article. The diameter of a scramble landscape is simply 1 since every permutation is reachable from any other via a single scramble operation.

Table 2. Fitness landscape diameter for cycle mutation and common permutation mutation operators.

Mutation Operator	Diameter
Cycle($kmax$), $kmax \geq 5$	$\approx 2n/kmax$
Cycle($kmax$), $kmax \leq 4$	$\max\{[n/2], [(n-1)/(kmax-1)]\}$
Cycle(α)	2
Swap	$n - 1$
Insertion	$n - 1$
Reversal	$n - 1$
Scramble	1

The diameter of a Cycle(α) landscape is 2, which is the diameter of the space of permutations for cycle edit distance (Section 3.2.2). When $kmax \leq 4$, the diameter of Cycle($kmax$) is $\max\{[n/2], [(n-1)/(kmax-1)]\}$, which is the maximum k-cycle distance (Section 3.2.3).

We previously saw that k-cycle distance does not satisfy the triangle inequality if $kmax \geq 5$. Although computing a Cycle($kmax$) edit distance for $kmax \geq 5$ is too costly for our purpose, it is straightforward to compute its diameter. Recall the examples illustrating that k-cycle distance violates the triangle inequality, and specifically that two cycle edits of length k can transform a set of cycles with a total of k elements into k fixed points. Thus, the maximum case of $[n/2]$ cycles of length 2 can be transformed to n fixed points with approximately $2n/kmax$ applications of the Cycle($kmax$) operator when $kmax \geq 5$.

3.3.2. Fitness Distance Correlation

We now compute FDC for small instances of the TSP, LCS, and QAP. To compute FDC for a problem instance, you need all optimal solutions to that instance. This necessitates utilizing an instance small enough to feasibly and reliably determine all optimal solutions. The FDC calculations in this section use a permutation length $n = 10$ for this reason, which means a 10-city TSP, an LCS with graphs of 10 vertexes, and a QAP with 10 by 10 cost and distance matrices. The EA experiments later in the paper use larger problem instances to experimentally compare the performance of the different mutation operators.

We use a TSP instance with 10 cities arranged equidistantly around a circle of radius 10, and Euclidean distance for the edge costs. In this way, the optimal solutions are known a priori. There are 20 optimal permutations, all representing the same TSP tour following the cities around the circle, including ten possible starting cities and two travel directions.

We generate a QAP instance such that a random permutation p of length 10 is the known optimal solution. Let $A = [(1, 90), (2, 89), \ldots, (90, 1)]$. Shuffle this array of ordered pairs. The 90 non-diagonal elements of the 10 by 10 cost matrix C are populated row by row using the first element of the tuples in A. Let (u, v) be one of these tuples. If $C[i][j]$ is set to u, then $D[p(i)][p(j)]$ in the distance matrix D is set to v.

We generate an LCS instance consisting of a pair of isomorphic graphs. In this way, we know that the LCS is either of those graphs or any of the other automorphisms of the graph. We use a strongly regular graph in the analysis. Strongly regular graphs are especially challenging for algorithms for the LCS and other related problems. Specifically, we use the Petersen graph [74], shown in Figure 1, which has 120 automorphisms, and thus 120 optimal vertex mappings. Such instances are hard because many vertexes of a strongly regular graph look the same locally to a solver, which can be simultaneously attracted toward different distinct solutions in a space plagued by complex local optima.

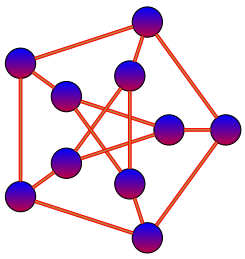


Figure 1. The Petersen graph, a strongly regular graph.

Although guidance on interpreting FDC varies, an $r \leq -0.15$ is commonly considered an easier fitness landscape since it implies that fitness increases as distance to an optimal solution decreases, while $r \geq 0.15$ is likely a deceptive landscape since fitness increases as you move away from optimal solutions, and $-0.15 < r < 0.15$ is considered difficult since there is very little correlation between fitness and distance to an optimal solution [28].

For each combination of problem and mutation operator, we compute FDC exactly, calculating Pearson correlation over all $n!$ = 3,628,800 permutations of length $n = 10$. Table 3 summarizes the results. Without loss of generality, for TSP and QAP we compute FDC from the cost function (to minimize) rather than a fitness function (to maximize), flipping the sign of the FDC with positive FDC, implying more straightforward problem solving. Since LCS is a maximization problem, the sign of the FDC is interpreted normally.

Table 3. FDC for combinations of mutation operator and problem.

Mutation Operator	TSP	QAP	LCS
Cycle(α)	−0.0569	0.0213	−0.0278
Cycle(5)	0.1801	0.1339	−0.5342
Cycle(4)	0.1667	0.1737	−0.3984
Cycle(3)	0.2482	0.2210	−0.6180
Swap	0.3318	0.2245	−0.6355
Insertion	0.5277	0.0305	−0.3547
Reversal	0.8459	0.0189	−0.0350
Scramble	0.0117	0.0048	−0.0340

For the TSP, there is very strong FDC for reversal mutation, and lesser but still strong FDC for insertion mutation. The FDC analysis predicts that these will perform better than the others. Swap has the next-highest FDC. Although not as high, all cases of Cycle($kmax$) exhibit $r > 0.15$. Given the correlation strength of reversal and insertion landscapes, we expect them to dominate the others, but swap and Cycle($kmax$) may also perform well.

The strength of the correlations are not nearly as high for the QAP, and only three of the mutation operators have an $r \geq 0.15$, including swap mutation and Cycle($kmax$) with a $kmax$ of either 3 or 4. This may be a problem where cycle mutation is better suited.

Because LCS is a maximization problem, FDC should be interpreted in the ordinary sense with FDC computed from fitness where higher fitness implies better solutions. Thus, negative FDC implies easier problem solving. In this case, there is very strong FDC for swap mutation, Cycle(3) mutation, and Cycle(5) mutation. Both Cycle(4) and insertion mutation also have $r \leq -0.15$, so should also progressively lead toward the solution.

The FDC for scramble mutation is very near 0 for all three problems. Scramble is thus unlikely to perform well. The FDC for Cycle(α) is also very near 0 for all three problems. However, we will see that FDC misses an important behavioral property of this operator.

3.3.3. Search Landscape Calculus

Let $\eta(p)$ be the neighbors of p (i.e., solutions reachable with one mutation) and $f(p)$ is fitness. Search landscape calculus [32] defines the average local rate of fitness change:

$$\Delta[f](p) = \frac{1}{|\eta(p)|} \sum_{p' \in \eta(p)} |f(p) - f(p')|. \tag{41}$$

Search landscape calculus then defines $\Delta[f]$ as the average of $\Delta[f](p)$ over all p. It is infeasible to directly compute $\Delta[f]$ for the true fitness function f. Therefore, the search landscape calculus focuses on topological properties that influence fitness, such as absolute positions, relative positions, and element precedences for permutation problems, replacing f by a distance function δ relevant to the context of a problem feature. Thus, define $\Delta[\delta](p)$:

$$\Delta[\delta](p) = \frac{1}{|\eta(p)|} \sum_{p' \in \eta(p)} \delta(p, p'), \tag{42}$$

and from this derive $\Delta[\delta]$ as the average of $\Delta[\delta](p)$ over all points p. The distance δ must correspond to the topological property under analysis, and its maximum must be proportional to the permutation length n (i.e., $\max\{\delta(p_1, p_2)\} \in \Theta(n)$).

We focus on two topological features, absolute element positions and edges, and thus require corresponding distance functions. We use exact match distance [67] (denoted as δ_{em}), which is the count of the number of positions containing different elements, as a measure of how different two permutations are within the context of absolute positioning; and we use cyclic edge distance [68] (denoted as δ_{ce}) as a measure of how different two permutations are within the context of relative positions. Both meet the requirement that the maximum distance is proportional to n. It is exactly n in both cases.

Table 4 summarizes $\Delta[\delta_{em}]$ and $\Delta[\delta_{ce}]$ for both forms of cycle mutation, and the other mutation operators considered. The rates of change of the topological properties of swap, insertion, reversal, and scramble were determined elsewhere [32]. The $\Delta[\delta_{em}]$ is the average number of elements whose absolute positions are changed by a single application of the operator. For cycle mutation, this is the average length of the induced cycle, determined earlier in the article—Cycle(α) in Section 3.1.3 and Cycle($kmax$) in Section 3.1.2. To compute $\Delta[\delta_{ce}]$ we need the average number of edges changed by the operator. For cycle mutation, this depends upon the cycle length, and it also depends upon whether any cycle elements are adjacent. As the permutation length n increases, the probability of adjacent cycle elements decreases. For sufficiently large n, the probability of adjacent cycle elements approaches 0, in which case the number of edges replaced by cycle mutation is, on average, twice the length of the cycle. Thus, $\Delta[\delta_{ce}] = 2\Delta[\delta_{em}]$ for both forms of cycle mutation.

Table 4. Average rates of change of fitness landscape topological properties.

Mutation Operator	$\Delta[\delta_{em}]$	$\Delta[\delta_{ce}]$
Cycle(α)	$(2-\alpha)/(1-\alpha)$	$(4-2\alpha)/(1-\alpha)$
Cycle($kmax$)	$(kmax+2)/2$	$kmax+2$
Swap	2	4
Insertion	$(n+4)/3$	3
Reversal	$[(n+1)/3, (n+4)/3]$	2
Scramble	$(n+1)/3$	$(n+1)/3$

In the search landscape calculus, when $\lim_{n\to\infty} \Delta[\delta] = \infty$, the fitness landscape exhibits very large local changes in fitness, often due to many deep local optima that are difficult to escape. Thus, we expect poor performance from insertion, reversal, and scramble on absolute-positioning problems since $\Delta[\delta_{em}]$ grows with n, as is the case for scramble when relative positions are more important. When $\lim_{n\to\infty} \Delta[\delta] = C$, for a non-zero finite constant C, the search landscape calculus suggests that the landscape is smooth locally. Due to constant $\Delta[\delta_{em}]$, swap and cycle mutation should perform better than the others for problems such as LCS and QAP where absolute positions more greatly impact fitness. All operators except for scramble have constant $\Delta[\delta_{ce}]$, and are thus potentially relevant to problems such as the TSP where edges influence fitness more than absolute element positions.

For cycle mutation, it should be noted that its topological characteristics depend upon the specific parameter settings. For example, higher values of $kmax$ likely lead to the same sort of disruption inherent in scramble mutation, as would values of α very near 1.0.

Previously, we saw that FDC suggested that Cycle(α) would likely perform poorly for all three problems due to FDC very near 0; whereas the search landscape calculus suggests that it may be relevant for all three problems. We now reconcile the discrepancy between these two fitness landscape analysis tools. The number of permutations within one step of a given permutation with respect to Cycle(α) is enormous, leading to extremely low variation in distance and then subsequently to near 0 FDC. However, a large proportion of the Cycle(α) neighbors are very low probability events. Thus, most of the time it behaves more similar to Cycle($kmax$) with a very low $kmax$, but with the ability to make larger jumps.

3.3.4. Summary of Fitness Landscape Analysis Findings

Due to strong FDC, we expect reversal mutation to dominate the others for the TSP, finding lower-cost solutions with the same or fewer evaluations. Insertion should likely be the next best since it also has strong FDC, and a low constant rate of fitness change for edge-focused problems. Swap and cycle mutation, if configured to emphasize smaller cycles, may also be relevant for such problems. Due to strong FDC for QAP and LCS, and low constant rates of change of exact match distance, we anticipate swap and both forms of cycle mutation will find lower cost solutions for absolute-positioning focused problems.

4. Results

We experimentally compare the two forms of cycle mutation with commonly encountered permutation mutation operators. Our experiments are implemented in Java. Our test machine is a Windows 10 PC, with an AMD A10-5700 3.4 GHz CPU, and 8 GB memory. The code is compiled with OpenJDK 17.0.2 for a Java 11 target, and runs on an OpenJDK 64-bit Server VM Temurin-17.0.2+8. The code is open source, licensed via the GPL 3.0, and available on GitHub at https://github.com/cicirello/cycle-mutation-experiments (accessed on 26 May 2022), and includes the code to analyze the experiment data and to generate the figures.

In the experiments, we apply each of the mutation operators within both a $(1+1)$-EA as well as in an SA. In a $(\mu+\lambda)$-EA [15], the population size is μ, λ offspring are created in each generation, and the best μ individuals from the combination of parents and offspring survive into the next generation. The $(1+1)$-EA is commonly used in experimental studies. We employ it here since it removes the impact of choice of selection operator and crossover operator, and also eliminates many parameters such as population size, mutation rate, etc. Additionally, it supports a more direct comparison with the non-population approach of SA. For the SA, we use the parameter-free Self-Tuning Lam Annealing [75] that adaptively adjusts the temperature parameter based on problem-solving feedback.

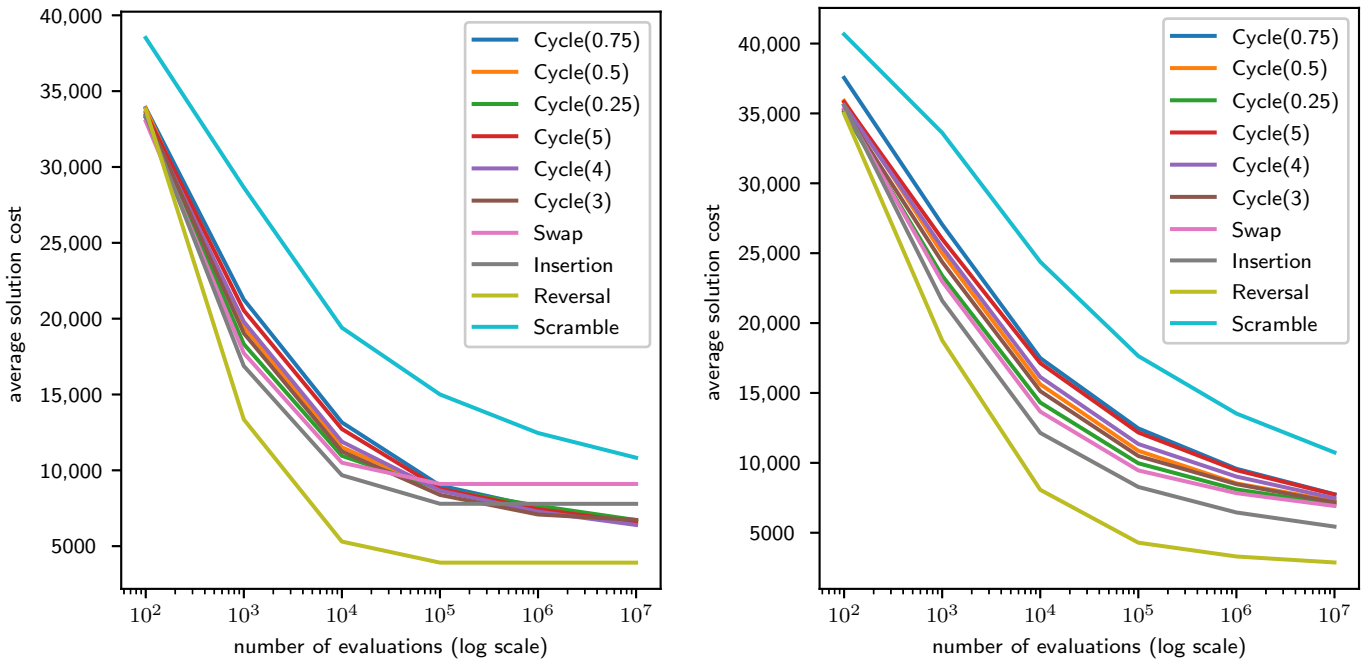
We consider three cases of Cycle($kmax$) mutation, including Cycle(3), Cycle(4), and Cycle(5); and three cases of Cycle(α) mutation, including Cycle(0.25), Cycle(0.5), and Cycle(0.75). All results are 50 run averages. We use run lengths $\{10^2, 10^3, 10^4, 10^5, 10^6, 10^7\}$ in number of evaluations. We test significance with the Wilcoxon rank sum test. The results on the TSP, QAP, and LCS are presented in Section 4.1, Section 4.2 and Section 4.3, respectively.

4.1. TSP Results

Each of the 50 TSP instances consists of 100 cities. An instance is defined by a random distance matrix, such that the distance between cities is a uniformly random integer from the interval $[1, 1000]$. The results are shown in Figure 2 with number of evaluations on the horizontal axis at log scale, and average solution cost over 50 runs on the vertical axis.

Consistent with the extremely strong FDC that we earlier observed for reversal mutation for the TSP, we find that reversal is clearly dominant on the 100-city TSP instances within both the EA (Figure 2a) and SA (Figure 2b), finding lower-cost solutions with fewer evaluations. All comparisons to reversal mutation are extremely statistically significant (e.g., Wilcoxon rank sum test p-values very near 0) except for the shortest 100 evaluation runs. For SA, the second-best mutation operator is insertion (results also statistically significant). We earlier saw that insertion had the second strongest FDC for the TSP. Swap and the various cases of cycle mutation all find solutions of approximately equivalent cost within the SA, especially for very long run lengths.

The EA comparison (Figure 2a) is a bit more interesting. Although for mid-length runs, insertion is second-best at statistically significant levels, the various cases of cycle mutation surpass insertion mutation for the longest runs. Insertion mutation exhibits a premature convergence effect, converging to a significantly suboptimal solution 10^5 evaluations into the EA runs, as does swap. However, cycle mutation continues to find lower-cost solutions, overtaking insertion mutation. The convergence effect in the EA is due to the smaller neighborhoods of swap and insertion, leading to greater impact of local optima. Cycle mutation has a larger neighborhood so better avoids this, continuing to show progress. In fact, even though we saw near-zero FDC for Cycle(α), all cases of Cycle(α) continue to make progress, although converging at a slower rate than reversal.



(**a**) (**b**)

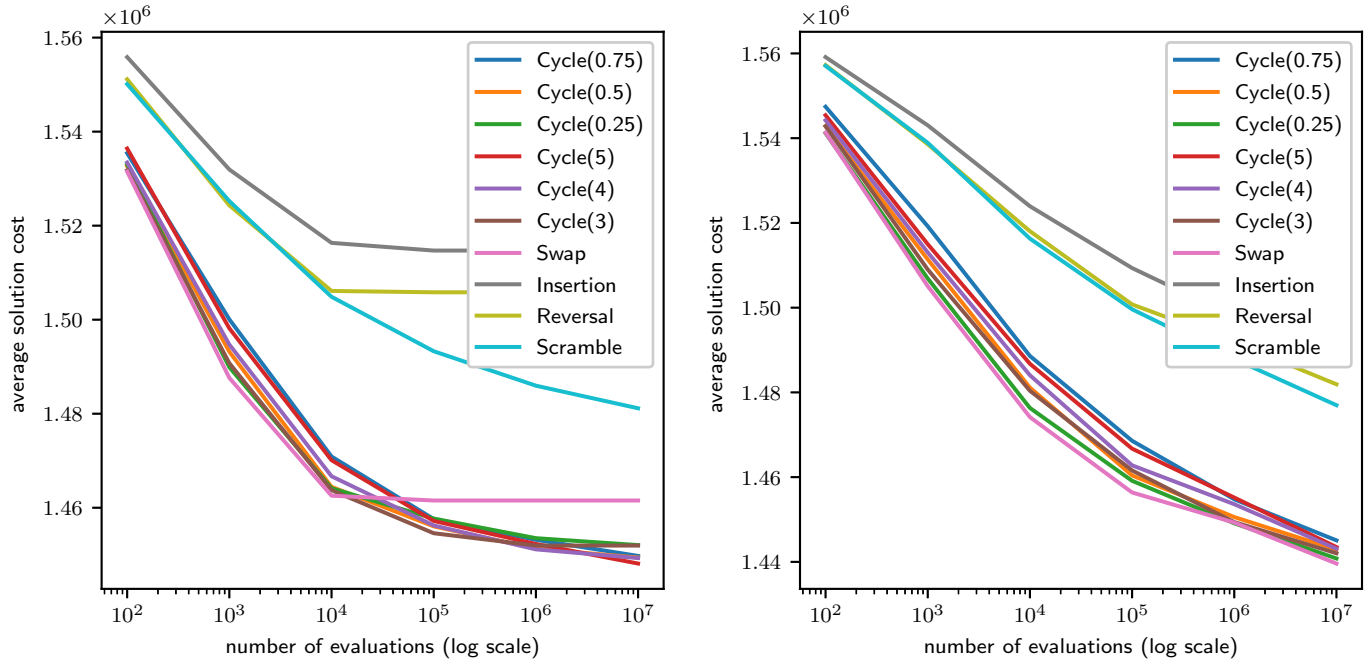
Figure 2. Results on the TSP for (**a**) $(1+1)$-EA, and (**b**) SA.

4.2. QAP Results

Each of the 50 QAP instances consists of a 50 by 50 cost matrix and a 50 by 50 distance matrix, with integer costs and distances generated uniformly at random from $[1, 50]$.

The results are visualized in Figure 3. We earlier saw that FDC suggested that swap, Cycle(3), and Cycle(4) would likely perform better than the others, while the search landscape calculus suggested that swap and all forms of cycle mutation are appropriate for the QAP. Consistent with the fitness landscape analysis, scramble, reversal, and insertion all perform poorly for QAP within both the EA and SA. For the longest SA runs (Figure 3b), there is little difference in solution cost among swap and the various cases of cycle mutation. For runs of 10^4 to 10^5 SA evaluations, swap and Cycle(0.25) find solutions with lower values of the cost function than the others at statistically significant levels.

The EA results are especially interesting (Figure 3a). Swap suffers from a premature convergence effect at 10^4 evaluations, while all cycle mutation cases continue to make substantial progress, minimizing the cost function. Furthermore, Cycle(0.25) and Cycle(3)'s rates of convergence are equivalent to swap up to that point. The superior convergence effect is attributed to cycle mutation's larger neighborhood, especially in the case of Cycle(α). We do not see the same behavior with SA (Figure 3b) because SA has a built-in way of handling local optima, allowing swap to continue to improve solution quality despite swap's much smaller local neighborhood.



(**a**) (**b**)

Figure 3. Results on the QAP for (**a**) $(1+1)$-EA, and (**b**) SA.

4.3. LCS Results

Consider two sets of experiments with the LCS problem, one with random graphs and the other with strongly regular graphs. In the random graph case, we have 50 instances, each consisting of a pair of randomly generated isomorphic graphs. We use pairs of isomorphic graphs so that each problem instance has a known optimal solution. That is, the largest common subgraph is simply the graph itself. Each graph has 50 vertexes and edge density 0.5, which means that the probability of each possible edge is 0.5. Thus, each graph has 50 vertexes, and the expected number of edges is $0.5 \times 50 \times 49/2 = 612.5$. The second graph for each instance is formed by relabeling the vertexes randomly.

We use generalized Petersen graph $G(25,2)$ for the strongly regular case. Generalized Petersen graph [76] $G(n,k)$ has $2n$ vertexes $V = \{u_0, u_1, \ldots, u_{n-1}, v_0, v_1, \ldots, v_{n-1}\}$, and $3n$ edges $E = \{(u_i, v_i) \mid 0 \leq i < n\} \bigcup \{(u_i, u_{i+1 \mod n}) \mid 0 \leq i < n\} \bigcup \{(v_i, v_{i+k \mod n}) \mid 0 \leq i < n\}$. The original Petersen graph (Figure 1) is $G(5,2)$. Figure 4 shows a generalized Petersen graph $G(25,2)$ that has 50 vertexes and 75 edges. We again average over 50 runs, but in this case, each instance consists of graph $G(25,2)$ and a second graph isomorphic to it that is formed by randomly relabeling the vertexes.

LCS is a maximization problem; however, for consistency we transform it to a minimization problem. Since each instance is a pair of isomorphic graphs, the LCS has $|E|$ edges, where E is the edge set. Thus, redefine LCS to minimize the cost of permutation p: $C(p) = |E| - |E'|$, where E' is the edge set of the subgraph implied by vertex mapping p.

The results are in Figure 5 (random graphs) and Figure 6 (strongly regular graphs). For SA and random graphs (Figure 5b), scramble, insertion, and reversal all perform very poorly compared with the others. Cycle(5) and Cycle(0.75) are inferior to the other cycle mutation cases for the longest runs at statistically significant levels. Within an EA for random graphs (Figure 5a), the behavior is similar to that of the QAP. Swap exhibits a premature convergence effect, while cycle mutation finds increasingly lower cost solutions. Cycle(3) also prematurely converges at 10^6 evaluations, while cycle mutation configured with a larger neighborhood continues to improve solution cost beyond that point. Cycle(0.25) is best at statistically significant levels for the 10^7 evaluation runs.

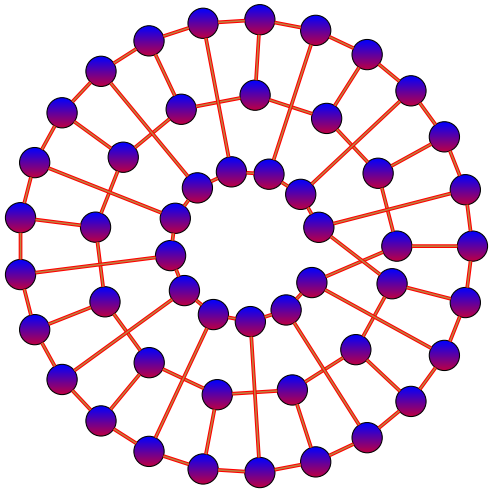
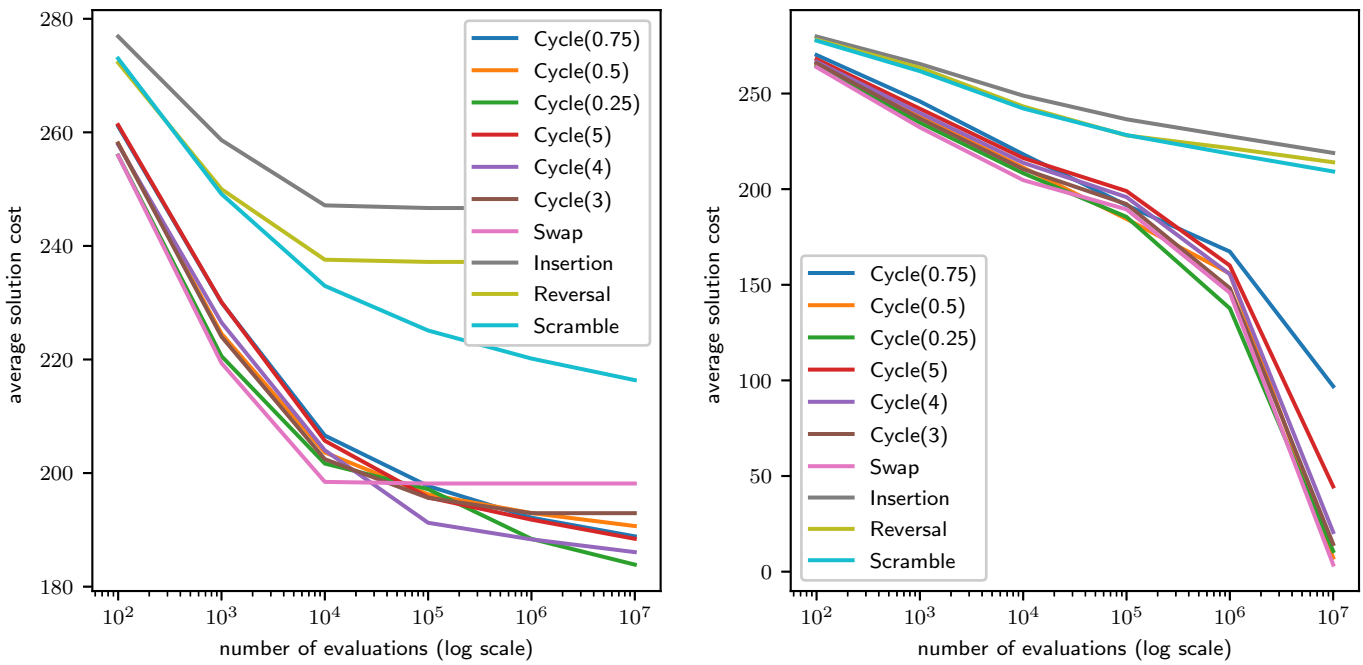


Figure 4. The generalized Petersen graph, $G(25, 2)$, a strongly regular graph.

These results are consistent with the fitness landscape analysis. First, FDC suggested that swap, Cycle(3), and Cycle(5) were best suited to the problem, followed by Cycle(4) and insertion. The FDC analysis was likely misled with respect to insertion mutation due to the small size of the instance used in the FDC analysis, which is likely why we find insertion performing poorly on the larger instance. This is consistent with the search landscape calculus analysis which suggests that swap and cycle mutation are both well suited to the problem, and that the other operators, including insertion, are not. The Cycle(0.25) is most effective within longer runs of the EA due to the combination of small average change (average cycle length of 2.33) and very large neighborhood size. Within SA, cycle mutation is neither better nor worse than swap, converging at the same rate.



(**a**) (**b**)

Figure 5. Results on the LCS problem with random graphs for (**a**) $(1+1)$-EA, and (**b**) SA.

Observe that the strongly regular graph results (Figure 6) are similar to those of random graphs, but are more pronounced. For example, in an EA (Figure 6a), swap and Cycle(3) prematurely converge, while Cycle(4) and Cycle(5) exhibit superior convergence effect,

outperforming all others at statistically significant levels from 10^4 evaluations onward, but differences between them are not significant.

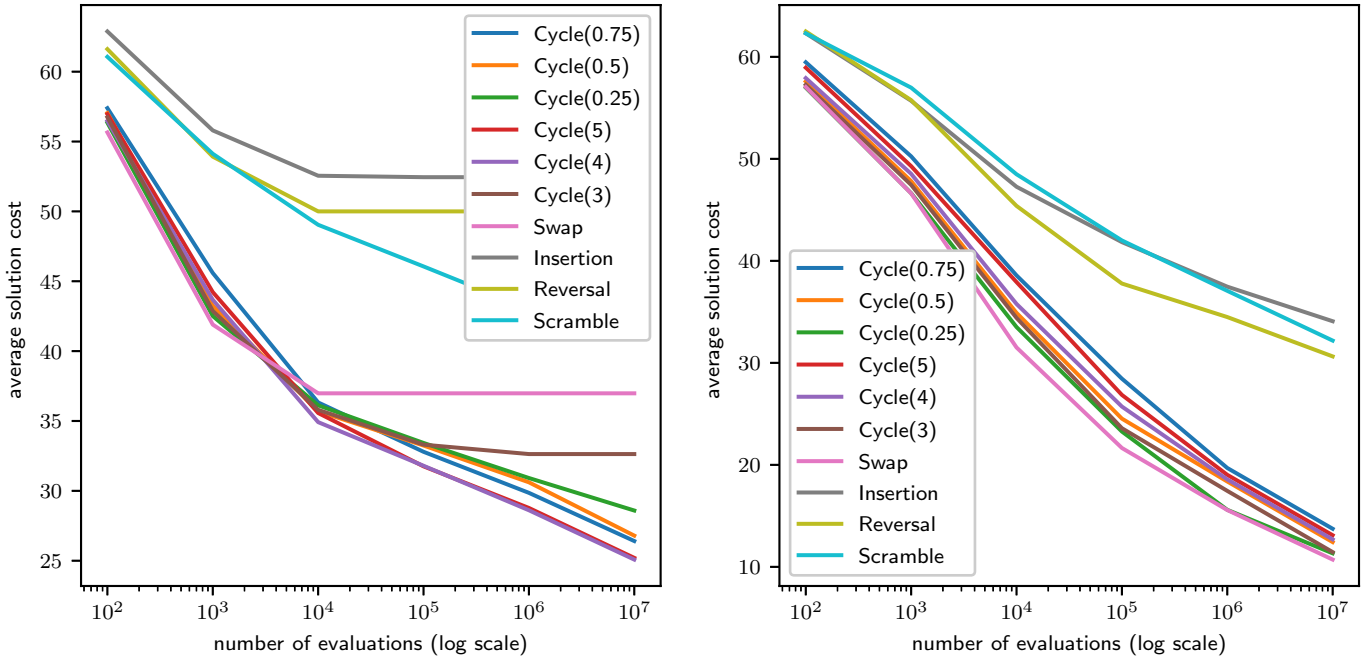


Figure 6. Results on the LCS problem with strongly regular graphs for (**a**) $(1+1)$-EA, and (**b**) SA.

5. Discussion and Conclusions

In this paper, we proposed a new mutation operator for use by EA and other related algorithms, such as SA, when evolving permutations. This new operator is called cycle mutation, and includes two variations: Cycle(α) and Cycle($kmax$). Cycle mutation induces a random permutation cycle. The difference between the two operators is in how the cycle length is chosen. The Cycle($kmax$) operator has a small maximum cycle length, while Cycle(α) does not impose any maximum cycle length. Thus, Cycle(α) has a significantly larger neighborhood size, while retaining a low average cycle length.

The runtime of cycle mutation in the worst case is linear, similar to insertion, reversal, and scramble mutation; but unlike those operators, cycle mutation's average runtime is constant. Therefore, the computational time to use cycle mutation within an EA is little more than that of the constant time swap. Helping to achieve this efficient runtime, cycle mutation relies on a new algorithm for sampling k elements from an n element set, called insertion sampling, which is faster than existing alternatives for low k.

The fitness landscape analysis showed that cycle mutation is well suited to permutation problems such as the QAP and LCS, where absolute positions more greatly impact fitness than relative ordering. The fitness landscape analysis also showed that cycle mutation may be relevant to relative ordering problems such as the TSP, provided cycle length is kept low. While undertaking the fitness landscape analysis, we developed three new measures of permutation distance: cycle distance, k-cycle distance, and cycle edit distance.

Validating the fitness landscape analysis, cycle mutation experimentally outperformed the others for QAP and LCS within an EA, especially for long runs, finding lower-cost solutions with fewer evaluations. Furthermore, while swap suffers from a premature convergence effect due to small neighborhood, cycle mutation continues to make optimization progress even for the longest runs; and the Cycle(α) form of cycle mutation is especially good at managing local optima, due to its very large neighborhood size enabling it to make large jumps when necessary. Thus, Cycle(α) exhibits superior convergence effect.

Cycle mutation does have limitations. Cycle mutation does not show any advantage within SA, although its rate of convergence is similar to that of swap in SA so it may be worth considering for SA nonetheless. Additionally, within an EA, Cycle($kmax$) may exhibit a premature convergence effect for some problems if $kmax$ is set too low, such as what we saw with Cycle(3) for the LCS. However, Cycle(α) overcomes this limitation.

Our Java implementations of cycle mutation are integrated into an open source library, Chips-n-Salsa [30], and our Java implementations of cycle distance, k-cycle distance, and cycle edit distance are integrated into the open source JavaPermutationTools [31] library. By disseminating the implementations in open source libraries, we hope to contribute not only to the research literature, but also to the state of practice. Additionally, all of the code to reproduce the experiments, and to analyze the results, is available in a GitHub repository, https://github.com/cicirello/cycle-mutation-experiments (accessed on 26 May 2022).

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All experiment data (raw and post-processed) are available on GitHub: https://github.com/cicirello/cycle-mutation-experiments (accessed on 26 May 2022), which also includes all source code of our experiments, as well as instructions for compiling and running the experiments.

Conflicts of Interest: The author declares no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CX	Cycle crossover
EA	Evolutionary algorithm
ES	Evolution strategies
FDC	Fitness distance correlation
GA	Genetic algorithm
LCS	Largest common subgraph
QAP	Quadratic assignment problem
SA	Simulated annealing
TSP	Traveling salesperson problem

References

1. Mitchell, M. *An Introduction to Genetic Algorithms*; MIT Press: Cambridge, MA, USA, 1998.
2. Beyer, H. *The Theory of Evolution Strategies*; Natural Computing Series; Springer: Berlin/Heidelberg, Germany, 2013.
3. Langdon, W.; Poli, R. *Foundations of Genetic Programming*; Springer: Berlin/Heidelberg, Germany, 2013.
4. Cuéllar, M.; Gómez-Torrecillas, J.; Lobillo, F.; Navarro, G. Genetic algorithms with permutation-based representation for computing the distance of linear codes. *Swarm Evol. Comput.* **2021**, *60*, 100797. [CrossRef]
5. Koohestani, B. A crossover operator for improving the efficiency of permutation-based genetic algorithms. *Expert Syst. Appl.* **2020**, *151*, 113381. [CrossRef]
6. Shabash, B.; Wiese, K.C. PEvoSAT: A Novel Permutation Based Genetic Algorithm for Solving the Boolean Satisfiability Problem. In Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, Amsterdam, The Netherlands, 6–10 July 2013; Association for Computing Machinery: New York, NY, USA, 2013; pp. 861–868. [CrossRef]
7. Kalita, Z.; Datta, D.; Palubeckis, G. Bi-objective corridor allocation problem using a permutation-based genetic algorithm hybridized with a local search technique. *Soft Comput.* **2019**, *23*, 961–986. [CrossRef]
8. Shakya, S.; Lee, B.S.; Di Cairano-Gilfedder, C.; Owusu, G. Spares parts optimization for legacy telecom networks using a permutation-based evolutionary algorithm. In Proceedings of the IEEE Congress on Evolutionary Computation (CEC), Donostia, Spain, 5–8 June 2017; pp. 1742–1748. [CrossRef]
9. Mironovich, V.; Buzdalov, M.; Vyatkin, V. Evaluation of Permutation-Based Mutation Operators on the Problem of Automatic Connection Matching in Closed-Loop Control System. In *Recent Advances in Soft Computing and Cybernetics*; Matoušek, R., Kůdela, J., Eds.; Springer International Publishing: Berlin/Heidelberg, Germany, 2021; pp. 41–51. [CrossRef]

10. Hinterding, R. Gaussian mutation and self-adaption for numeric genetic algorithms. In Proceedings of the IEEE International Conference on Evolutionary Computation, Perth, WA, Australia, 29 November–1 December 1995; Volume 1, pp. 384–389. [CrossRef]
11. Szu, H.; Hartley, R. Nonconvex optimization by fast simulated annealing. *Proc. IEEE* **1987**, *75*, 1538–1540. [CrossRef]
12. Campos, V.; Laguna, M.; Marti, R. Context-Independent Scatter and Tabu Search for Permutation Problems. *INFORMS J. Comput.* **2005**, *17*, 111–122. [CrossRef]
13. Cicirello, V.A. Classification of Permutation Distance Metrics for Fitness Landscape Analysis. In Proceedings of the 11th International Conference on Bio-Inspired Information and Communication Technologies, Pittsburgh, PA, USA, 13–14 March 2019; Springer: New York, NY, USA, 2019; pp. 81–97._7. [CrossRef]
14. Garey, M.R.; Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*; W. H. Freeman & Co.: New York, NY, USA, 1979.
15. Eiben, A.E.; Smith, J.E. *Introduction to Evolutionary Computing*, 2nd ed.; Springer: Berlin/Heidelberg, Germany, 2015.
16. Davis, L. Applying Adaptive Algorithms to Epistatic Domains. In Proceedings of the International Joint Conference on Artificial Intelligence, San Francisco, CA, USA, 18–23 August 1985; pp. 162–164.
17. Cicirello, V.A. Non-Wrapping Order Crossover: An Order Preserving Crossover Operator that Respects Absolute Position. In Proceedings of the Genetic and Evolutionary Computation Conference, Seattle, WA, USA, 8–12 July 2006; ACM Press: New York, NY, USA, 2006; Volume 2, pp. 1125–1131. [CrossRef]
18. Syswerda, G. Schedule Optimization using Genetic Algorithms. In *Handbook of Genetic Algorithms*; Davis, L., Ed.; Van Nostrand Reinhold: New York, NY, USA, 1991.
19. Goldberg, D.E.; Lingle, R. Alleles, Loci, and the Traveling Salesman Problem. In Proceedings of the 1st International Conference on Genetic Algorithms, Sheffield, UK, 12–14 September 1995; Lawrence Erlbaum Associates, Inc.: Mahwah, NJ, USA, 1985; pp. 154–159.
20. Cicirello, V.A.; Smith, S.F. Modeling GA Performance for Control Parameter Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*; Morgan Kaufmann Publishers: San Francisco, CA, USA, 2000; pp. 235–242.
21. Bierwirth, C.; Mattfeld, D.; Kopfer, H. On permutation representations for scheduling problems. In *Proceedings of the International Conference on Parallel Problem Solving from Nature*; Springer: Berlin/Heidelberg, Germany, 1996; pp. 310–318.
22. Nagata, Y.; Kobayashi, S. Edge Assembly Crossover: A High-Power Genetic Algorithm for the Travelling Salesman Problem. In Proceedings of the International Conference on Genetic Algorithms, East Lansing, MI, USA, 19–23 July 1997; pp. 450–457.
23. Watson, J.P.; Ross, C.; Eisele, V.; Denton, J.; Bins, J.; Guerra, C.; Whitley, L.D.; Howe, A.E. The Traveling Salesrep Problem, Edge Assembly Crossover, and 2-opt. In Proceedings of the International Conference on Parallel Problem Solving from Nature, Amsterdam, The Netherlands, 27–30 September 1998; Springer: Berlin/Heidelberg, Germany, 1998; pp. 823–834.
24. Oliver, I.M.; Smith, D.J.; Holland, J.R.C. A study of permutation crossover operators on the traveling salesman problem. In Proceedings of the 2nd International Conference on Genetic Algorithms, Cambridge, MA, USA, 1 July 1987; Lawrence Erlbaum Associates, Inc.: Mahwah, NJ, USA, 1987; pp. 224–230.
25. Delahaye, D.; Chaimatanan, S.; Mongeau, M. Simulated Annealing: From Basics to Applications. In *Handbook of Metaheuristics*; Gendreau, M., Potvin, J.Y., Eds.; Springer: Berlin/Heidelberg, Germany, 2019; pp. 1–35._1. [CrossRef]
26. Laarhoven, P.J.M.; Aarts, E.H.L. *Simulated Annealing: Theory and Applications*; Kluwer Academic Publishers: Norwell, MA, USA, 1987.
27. Kirkpatrick, S.; Gelatt, C.D.; Vecchi, M.P. Optimization by Simulated Annealing. *Science* **1983**, *220*, 671–680. [CrossRef]
28. Jones, T.; Forrest, S. Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms. In Proceedings of the 6th International Conference on Genetic Algorithms, Pittsburgh, PA, USA, 15–19 July 1995; Morgan Kaufmann: San Francisco, CA, USA, 1995; pp. 184–192.
29. National Academies of Sciences, Engineering, and Medicine. *Reproducibility and Replicability in Science*; The National Academies Press: Washington, DC, USA, 2019. [CrossRef]
30. Cicirello, V.A. Chips-n-Salsa: A Java Library of Customizable, Hybridizable, Iterative, Parallel, Stochastic, and Self-Adaptive Local Search Algorithms. *J. Open Source Softw.* **2020**, *5*, 2448. [CrossRef]
31. Cicirello, V.A. JavaPermutationTools: A Java Library of Permutation Distance Metrics. *J. Open Source Softw.* **2018**, *3*, 950. [CrossRef]
32. Cicirello, V.A. The Permutation in a Haystack Problem and the Calculus of Search Landscapes. *IEEE Trans. Evol. Comput.* **2016**, *20*, 434–446. [CrossRef]
33. Knuth, D.E. *The Art of Computer Programming, Volume 1, Fundamental Algorithms*, 3rd ed.; Addison Wesley: Boston, MA, USA, 1997.
34. Junior Mele, U.; Maria Gambardella, L.; Montemanni, R. Machine Learning Approaches for the Traveling Salesman Problem: A Survey. In Proceedings of the 8th International Conference on Industrial Engineering and Applications, Barcelona, Spain, 8–11 January 2021; ACM: New York, NY, USA, 2021; pp. 182–186. [CrossRef]
35. Mele, U.J.; Chou, X.; Gambardella, L.M.; Montemanni, R. Reinforcement Learning and Additional Rewards for the Traveling Salesman Problem. In Proceedings of the 8th International Conference on Industrial Engineering and Applications, Barcelona, Spain, 8–11 January 2021; ACM: New York, NY, USA, 2021; pp. 198–204. [CrossRef]

36. Wang, R.L.; Gao, S. A Co-Evolutionary Hybrid ACO for Solving Traveling Salesman Problem. In Proceedings of the 5th International Conference on Computer Science and Application Engineering, Virtual Conference, Sanya, China, 19–21 October 2021; ACM: New York, NY, USA, 2021; pp. 1–4.
37. Dinh, Q.T.; Do, D.D.; Hà, M.H. Ants Can Solve the Parallel Drone Scheduling Traveling Salesman Problem. In Proceedings of the Genetic and Evolutionary Computation Conference, Lille, France, 10–14 July 2021; ACM: New York, NY, USA, 2021; pp. 14–21. [CrossRef]
38. Varadarajan, S.; Whitley, D. A Parallel Ensemble Genetic Algorithm for the Traveling Salesman Problem. In Proceedings of the Genetic and Evolutionary Computation Conference, Lille, France, 10–14 July 2021; ACM: New York, NY, USA, 2021; pp. 636–643. [CrossRef]
39. Nagata, Y. High-Order Entropy-Based Population Diversity Measures in the Traveling Salesman Problem. *Evol. Comput.* **2020**, *28*, 595–619. [CrossRef]
40. Ibada, A.J.; Tuu-Szabo, B.; Koczy, L.T. A New Efficient Tour Construction Heuristic for the Traveling Salesman Problem. In Proceedings of the 5th International Conference on Intelligent Systems, Metaheuristics & Swarm Intelligence, Victoria, Seychelles, 10–11 April 2021; ACM: New York, NY, USA, 2021; pp. 71–76. [CrossRef]
41. Dell'Amico, M.; Montemanni, R.; Novellani, S. A Random Restart Local Search Matheuristic for the Flying Sidekick Traveling Salesman Problem. In Proceedings of the 8th International Conference on Industrial Engineering and Applications, Barcelona, Spain, 8–11 January 2021; ACM: New York, NY, USA, 2021; pp. 205–209. [CrossRef]
42. Gao, Y.; Shen, Y.; Yang, Z.; Chen, D.; Yuan, M. Immune Optimization Algorithm for Traveling Salesman Problem Based on Clustering Analysis and Self-Circulation. In Proceedings of the 3rd International Conference on Advanced Information Science and System, Sanya, China, 26–28 November 2021; ACM: New York, NY, USA, 2021. [CrossRef]
43. Tong, B.; Wang, J.; Wang, X.; Zhou, F.; Mao, X.; Zheng, W. Optimal Route Planning for Truck–Drone Delivery Using Variable Neighborhood Tabu Search Algorithm. *Appl. Sci.* **2022**, *12*, 529. [CrossRef]
44. Qamar, M.S.; Tu, S.; Ali, F.; Armghan, A.; Munir, M.F.; Alenezi, F.; Muhammad, F.; Ali, A.; Alnaim, N. Improvement of Traveling Salesman Problem Solution Using Hybrid Algorithm Based on Best-Worst Ant System and Particle Swarm Optimization. *Appl. Sci.* **2021**, *11*, 4780. [CrossRef]
45. Rico-Garcia, H.; Sanchez-Romero, J.L.; Jimeno-Morenilla, A.; Migallon-Gomis, H. A Parallel Meta-Heuristic Approach to Reduce Vehicle Travel Time in Smart Cities. *Appl. Sci.* **2021**, *11*, 818. [CrossRef]
46. An, H.C.; Kleinberg, R.; Shmoys, D.B. Approximation Algorithms for the Bottleneck Asymmetric Traveling Salesman Problem. *ACM Trans. Algorithms* **2021**, *17*, 1–12. [CrossRef]
47. Svensson, O.; Tarnawski, J.; Végh, L.A. A Constant-Factor Approximation Algorithm for the Asymmetric Traveling Salesman Problem. *J. ACM* **2020**, *67*, 1–53. [CrossRef]
48. Tsilomitrou, O.; Tzes, A. Mobile Data-Mule Optimal Path Planning for Wireless Sensor Networks. *Appl. Sci.* **2022**, *12*, 247. [CrossRef]
49. He, L.; Liu, Z.Y.; Liu, M.; Yang, X.; Zhang, F.Y. Quadratic Assignment Problem via a Convex and Concave Relaxations Procedure. In Proceedings of the 3rd International Conference on Robotics, Control and Automation, Chengdu, China, 11–13 August 2018; ACM: New York, NY, USA, 2018; pp. 147–153. [CrossRef]
50. Beham, A.; Affenzeller, M.; Wagner, S. Instance-Based Algorithm Selection on Quadratic Assignment Problem Landscapes. In Proceedings of the Genetic and Evolutionary Computation Conference Companion, Berlin, Germany, 15–19 July 2017; ACM: New York, NY, USA, 2017; pp. 1471–1478. [CrossRef]
51. Benavides, X.; Ceberio, J.; Hernando, L. On the Symmetry of the Quadratic Assignment Problem through Elementary Landscape Decomposition. In Proceedings of the Genetic and Evolutionary Computation Conference Companion, Lille, France, 10–14 July 2021; ACM: New York, NY, USA, 2021; pp. 1414–1422. [CrossRef]
52. Baioletti, M.; Milani, A.; Santucci, V.; Tomassini, M. Search Moves in the Local Optima Networks of Permutation Spaces: The QAP Case. In Proceedings of the Genetic and Evolutionary Computation Conference Companion, Prague, Czech Republic, 13–17 July 2019; ACM: New York, NY, USA, 2019; pp. 1535–1542. [CrossRef]
53. Novaes, G.A.S.; Moreira, L.C.; Chau, W.J. Exploring Tabu Search Based Algorithms for Mapping and Placement in NoC-Based Reconfigurable Systems. In Proceedings of the 32nd Symposium on Integrated Circuits and Systems Design, Sao Paulo, Brazil, 26–30 August 2019; ACM: New York, NY, USA, 2019. [CrossRef]
54. Thomson, S.L.; Ochoa, G.; Daolio, F.; Veerapen, N. The Effect of Landscape Funnels in QAPLIB Instances. In Proceedings of the Genetic and Evolutionary Computation Conference Companion, Berlin, Germany, 15–19 July 2017; ACM: New York, NY, USA, 2017; pp. 1495–1500. [CrossRef]
55. Irurozki, E.; Ceberio, J.; Santamaria, J.; Santana, R.; Mendiburu, A. Algorithm 989: Perm_mateda: A Matlab Toolbox of Estimation of Distribution Algorithms for Permutation-Based Combinatorial Optimization Problems. *ACM Trans. Math. Softw.* **2018**, *44*, 47. [CrossRef]
56. Cicirello, V.A.; Regli, W.C. An Approach to a Feature-based Comparison of Solid Models of Machined Parts. *Artif. Intell. Eng. Des. Anal. Manuf.* **2002**, *16*, 385–399. [CrossRef]
57. Chen, J.; Zaman, M.; Makris, Y.; Blanton, R.D.S.; Mitra, S.; Schafer, B.C. DECOY: Deflection-Driven HLS-Based Computation Partitioning for Obfuscating Intellectual Property. In Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference, Virtual Conference, 20–24 July 2020; IEEE Press: Piscataway, NJ, USA, 2020; pp. 1–6.

58. Stoichev, S.; Petrova, D. An Application of an Algorithm for Common Subgraph Detection for Comparison of Protein Molecules. In Proceedings of the International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing, Ruse, Bulgaria, 18–19 June 2009; ACM: New York, NY, USA, 2009; pp. 1–6. [CrossRef]
59. Zeller, A. Isolating Cause-Effect Chains from Computer Programs. In Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, Charleston, SC, USA, 18–22 November 2002; ACM: New York, NY, USA, 2002; pp. 1–10. [CrossRef]
60. Wong, J.L.; Kourshanfar, F.; Potkonjak, M. Flexible ASIC: Shared Masking for Multiple Media Processors. In Proceedings of the 42nd Annual Design Automation Conference, Anaheim, CA, USA, 3–17 June 2005; ACM: New York, NY, USA, 2005; pp. 909–914. [CrossRef]
61. Jordan, P.W.; Makatchev, M.; Pappuswamy, U. Understanding Complex Natural Language Explanations in Tutorial Applications. In Proceedings of the Third Workshop on Scalable Natural Language Understanding, Stroudsburg, PA, USA, 8 June 2006; Association for Computational Linguistics: Stroudsburg, PE, USA, 2006; pp. 17–24.
62. Puodzius, C.; Zendra, O.; Heuser, A.; Noureddine, L. Accurate and Robust Malware Analysis through Similarity of External Calls Dependency Graphs. In Proceedings of the 16th International Conference on Availability, Reliability and Security, Vienna, Austria, 17–20 August 2021; ACM: New York, NY, USA, 2021; pp. 1–12. [CrossRef]
63. Wagner, R.A.; Fischer, M.J. The String-to-String Correction Problem. *J. ACM* **1974**, *21*, 168–173. [CrossRef]
64. Levenshtein, V.I. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Sov. Phys. Dokl.* **1966**, *10*, 707–710.
65. Sörensen, K. Distance measures based on the edit distance for permutation-type representations. *J. Heuristics* **2007**, *13*, 35–47. [CrossRef]
66. Schiavinotto, T.; Stützle, T. A review of metrics on permutations for search landscape analysis. *Comput. Oper. Res.* **2007**, *34*, 3143–3153. [CrossRef]
67. Ronald, S. More Distance Functions for Order-based Encodings. In Proceedings of the IEEE Congress on Evolutionary Computation, Anchorage, AK, USA, 4–9 May 1998; pp. 558–563.
68. Ronald, S. Distance Functions for Order-based Encodings. In Proceedings of the IEEE Congress on Evolutionary Computation, Indianapolis, IN, USA, 13–16 April 1997; pp. 49–54.
69. Vitter, J.S. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* **1985**, *11*, 37–57. [CrossRef]
70. Ernvall, J.; Nevalainen, O. An Algorithm for Unbiased Random Sampling. *Comput. J.* **1982**, *25*, 45–47. [CrossRef]
71. Cicirello, V.A. Impact of Random Number Generation on Parallel Genetic Algorithms. In Proceedings of the Thirty-First International Florida Artificial Intelligence Research Society Conference, Melbourne, FL, USA, 21–23 May 2018; AAAI Press: Palo Alto, CA, USA, 2018; pp. 2–7.
72. Caprara, A. Sorting by Reversals is Difficult. In Proceedings of the International Conference on Computational Molecular Biology, Santa Fe, NM, USA, 20–23 January 1997; pp. 75–83.
73. Bafna, V.; Pevzner, P.A. Genome Rearrangements and Sorting by Reversals. *SIAM J. Comput.* **1996**, *25*, 272–289. [CrossRef]
74. Harary, F. *Graph Theory*; Addison-Wesley: Boston, MA, USA, 1967.
75. Cicirello, V.A. Self-Tuning Lam Annealing: Learning Hyperparameters While Problem Solving. *Appl. Sci.* **2021**, *11*, 9828. [CrossRef]
76. Watkins, M.E. A theorem on tait colorings with an application to the generalized Petersen graphs. *J. Comb. Theory* **1969**, *6*, 152–164. [CrossRef]