mathematics

MDPI

Article

A Population-Based Local Search Algorithm for the Identifying Code Problem

Alejandro Lara-Caballero * and Diego González-Moreno

Department of Applied Mathematics and Systems, Universidad Autónoma Metropolitana, Cuajimalpa, Mexico City 05348, Mexico; dgonzalez@cua.uam.mx
* Correspondence: alarac@cua.uam.mx

Abstract: The identifying code problem for a given graph involves finding a minimum subset of vertices such that each vertex of the graph is uniquely specified by its nonempty neighborhood within the identifying code. The combinatorial optimization problem has a wide variety of applications in location and detection schemes. Finding an identifying code of minimum possible size is a difficult task. In fact, it has been proven to be computationally intractable (NP-complete). Therefore, the use of heuristics to provide good approximations in a reasonable amount of time is justified. In this work, we present a new population-based local search algorithm for finding identifying codes of minimum cost. Computational experiments show that the proposed approach was found to be more effective than other state-of-the-art algorithms at generating high-quality solutions in different types of graphs with varying numbers of vertices.

Keywords: identifying code; combinatorial optimization; population-based search; local search; configuration checking

MSC: 68W50; 05C90

Citation: Lara-Caballero, A.; González-Moreno, D. A Population-Based Local Search Algorithm for the Identifying Code Problem. *Mathematics* **2023**, *11*, 4361. https://doi.org/10.3390/math11204361

Academic Editor: Alma Y. Alanis

Received: 18 August 2023
Revised: 30 September 2023
Accepted: 6 October 2023
Published: 20 October 2023

1. Introduction

An identifying code is a dominating subset of vertices in a graph such that the closed neighborhood of any vertex has a distinct intersection with the identifying code. The fundamental idea is to uniquely identify an element in a system by its neighbors. The goal of the identifying code problem is to find a code of minimum cardinality for any given graph. Identifying codes can be applied to a wide variety of applications such as fault detection and location detection [1,2]. The definition of identifying codes was motivated by the need to solve fault diagnosis problems in multiprocessor systems [1]. The main objective of fault diagnosis is to test a system and locate faulty nodes. The aim is to choose the smallest subset of processors, known as a code, to perform fault diagnostics at the lowest possible cost. A processor in the code is called a codeword. A codeword tests itself and all the processors it is connected to. If a codeword detects a fault, it sends an alarm signal. Once some alarms are activated, the exact location of the fault can be determined.

The term identifying codes was first coined by Karpovsky et al. in [1], where they described the conceptual foundations and provided results and lower bounds for graphs with specific topologies, such as binary cubes, non-binary cubes, and trees. Since this prominent paper, the theoretical and practical aspects of identifying codes have attracted the attention of many scholars during the last few decades.

From a theoretical perspective, researchers have studied the identifying code for many classes of graphs. For instance, Charon et al. [3] studied the identifying codes of different types of infinite graphs, such as square and triangular lattices. They also showed that the problem complexity in several types of graphs is NP-hard. Some classes that have been extensively studied include trees [4], paths [5], cycles, [5–7], hypercubes [8,9], and

grids [10]. For additional references on the theoretical aspect of identifying codes and other related problems, we refer the reader to D. Jean's web page [11].

From a practical viewpoint, the identifying code problem has numerous applications in real-world domains, such as fault diagnosis of multiprocessor systems [1], compact routing in networks [12], emergency sensor networks in facilities [2], and the analysis of secondary RNA structures [13]. Due to its wide applicability, some researchers have focused on designing algorithms and techniques to solve the problem computationally.

Laifenfeld et al. studied covering problems using identifying codes in [14]. Approximation algorithms for computing identifying codes for certain types of graphs are given in [15,16]. In the context of emergency sensor networks, there have been several attempts to apply identifying codes for target identification [17]. In particular, Ray et al. [2] proposed a greedy algorithm called ID-CODE to construct irreducible identifying codes. It should be emphasized that their algorithm converges to a local minimum, which is not necessarily close to the global minimum. In [18], Xiao et al. formulated the problem using an integer program and developed a genetic algorithm to solve it. Recently, Horan et al. [19] compared three approaches for finding minimum identifying codes on Brujin graphs using quantum annealing.

The identifying code problem is proven to be NP-complete, even when restricted to the case of bipartite graphs [20]. This means that there is no known polynomial-time algorithm to solve the problem, so the use of heuristics is justified to obtain quality solutions in a reasonable time frame. To the best of our knowledge, the use of population-based approaches on the identifying code problem from a graph theory perspective is limited in the literature. This paper presents our work in this direction.

This study proposes a new population-based local search method (PB-LS) for solving the identifying code problem. While existing techniques offer viable solutions, they often struggle to balance exploration and exploitation, especially for large problem instances. Our main contribution is a local search strategy based on configuration checking that addresses these limitations by improving the quality of solutions. Our goal is to contribute to ongoing efforts to find higher-quality solutions for the identifying code problem.

The experimental results demonstrate that the proposed algorithm outperforms other state-of-the-art methods in several test instances, suggesting that combining population-based heuristics with a local search is a promising approach for solving the identifying code problem.

The rest of the manuscript is organized as follows. Section 2 provides some necessary notation and definitions. Section 3 introduces the proposed algorithm and details its components. Section 4 presents the computational experiments and discussion. In Section 5, conclusions and paths for future investigation are drawn.

2. Preliminaries

A graph G consists of a pair of sets $(V(G), E(G))$, where $V(G)$ is the vertex set of G and $E(G)$ is the edge set of G. Specifically, $E(G)$ is a set of unordered pairs of vertices of $V(G)$. If $u, v \in V(G)$, an edge between u and v is denoted by uv. As we mentioned previously, the physical location of the components in many mathematical and real-life problems can be modeled using graphs. For example, given a multiprocessor system, we associate it with a graph G in the following way: the vertices of G are the processors and the edges represent the links between them. If there is just one fault at the time, an identifying code can be used to determine the exact location of the fault once a certain number of alarms are activated.

Let G be a graph and let v be a vertex of G. The *neighborhood* $N(v)$ of v is the set of all vertices adjacent (directly connected) to v and the *closed neighborhood* $N[v]$ of v is the set $N(v) \cup \{v\}$. That is, the closed neighborhood includes both the neighbor vertices of v and v itself. Some definitions are described as follows.

Definition 1. *Given a graph G, a subset C of $V(G)$ is called a separating code if for each pair of distinct vertices u, v of G, we have $N[u] \cap C \neq N[v] \cap C$. (An example is shown in Figure 1).*

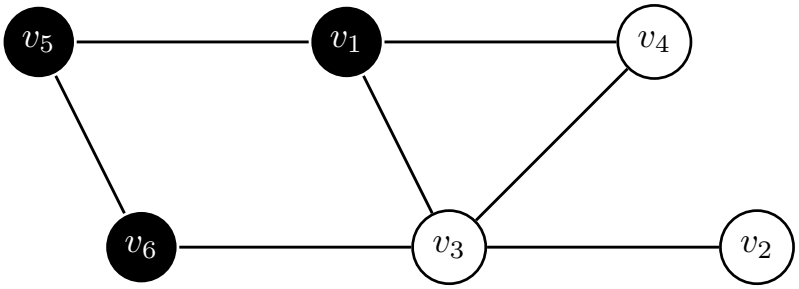


Figure 1. An example of a separating code on the graph. The black vertices are the members of the separating code.

Definition 2. *Given a graph G, a subset C of $V(G)$ is called dominating if for every vertex $v \in V(G)$, we have $C \cap N[v] \neq \emptyset$. (An example is shown in Figure 2).*

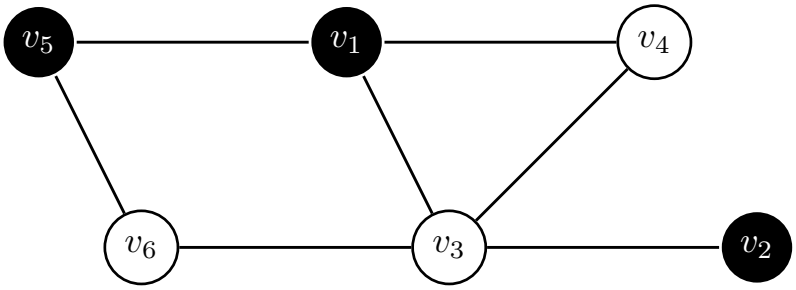


Figure 2. An example of a dominating set on the graph. The black vertices are the members of the dominating set.

Definition 3. *Given a graph G, a subset C of $V(G)$ is called an identifying code if C is both a separating code and a dominating code of G. (An example is shown in Figure 3).*

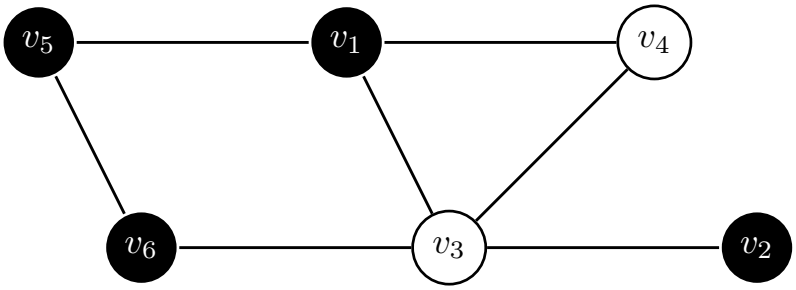


Figure 3. An example of an identifying code on the graph. The black vertices are the codewords.

To illustrate the meaning of the identifying code, we use the graph in Figure 3, which represents a simplified multiprocessor system. To diagnose faulty processors, specific software routines are executed on a select group of processors. Suppose that these special processors are located at the vertices $C = \{v_1, v_2, v_5, v_6\}$. It can be verified that the intersection between this group and the close neighborhood of each processor is unique: $N[v_1] \cap C = \{v_1, v_5\}, N[v_2] \cap C = \{v_2\}, N[v_3] \cap C = \{v_1, v_2, v_6\}$, $N[v_4] \cap C = \{v_1\}$, $N[v_5] \cap C = \{v_1, v_5, v_6\}$, $N[v_6] \cap C = \{v_5, v_6\}$. Therefore, the vertex set C allows for the unique identification of a faulty processor and is an identifying code.

The aforementioned relationships can be expressed conveniently in matrix notation. If G is a graph with $V(G) = \{u_1, u_2, \ldots, u_n\}$, the *adjacency matrix of* G is the $n \times n$ $(0,1)$-matrix $A(G) = [a_{ij}]$, where

$$[a_{ij}] = \begin{cases} 1 & \text{if } u_i u_j \in E(G), \\ 0 & \text{if } u_i u_j \notin E(G). \end{cases} \tag{1}$$

The closed adjacency matrix is the matrix $A + I$, where I is the $n \times n$-identity matrix. Let S be a subset of $V(G)$. We can associate a vector $(s_1, s_2, \ldots, s_n)$ with S such that $s_i = 1$, if $s_i \in S$ and $s_i = 0$, if $s_i \notin S$. The closed neighborhood of a vertex u_i of G is the ith row of the closed adjacency matrix, that is

$$N[u_i] = (a_{i1}, a_{i2}, \ldots, a_{in}). \tag{2}$$

Let G be a graph of order n and vertex set $V(G) = \{u_1, u_2, \ldots, u_n\}$. Let C be a subset of vertices of G with $|C| = k$. Let $M = [m_{ij}]$ be a $k \times n$ matrix with entries from $\{0, 1\}$ such that

$$[m_{ij}] = \begin{cases} 1 & \text{if } u_i \in N[u_j] \cap C, \\ 0 & \text{if } u_i \notin N[u_j] \cap C. \end{cases} \tag{3}$$

In other words, M is a matrix obtained from the closed adjacency matrix of G by deleting the rows corresponding to the vertices not in C. Observe that C is an identifying code of G if M satisfies the following conditions:

(i) all the columns of M contain at least a 1,
(ii) all the columns of M are distinct.

Condition (i) states that $N[u] \cap C \neq \emptyset$ for all $v \in V(G)$, and condition (ii) states that $N[u] \cap C \neq N[v] \cap C$, for every $u, v \in V(G)$. To clarify the expression of an identifying code using the matrix point of view, consider as an example the graph G depicted in Figure 3. The vertex set $C = \{v_1, v_2, v_5, v_6\}$ is an identifying code of G. We present the adjacency closed matrix A and the matrix M obtained by considering the row corresponding to vertices of C. Note that M satisfies conditions (i) and (ii).

$$A = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}, M = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}. \tag{4}$$

Therefore, the problem of finding a minimum identifying code is equivalent to that of finding a $(0,1)$-matrix M of n columns and the minimum number of rows satisfying the previous conditions; the rows of this matrix are an identifying code of the graph.

An Integer Formulation for the Identifying Code Problem

The minimum identifying code problem in graphs can be modeled by an integer program formulation [21]. Let G be a graph of order n and let $A = [a_{ij}]$ be the closed adjacency matrix of G, as defined in Section 2. Our vertex subset C is defined as a vector $(c_1, c_2, \ldots, c_n)^T$ with $c_i = 1$, if $c_i \in C$ and $c_i = 0$, if $c_i \notin C$. Then, C is an identifying code of G if and only if the following conditions hold:

$$\sum_{k=1}^{n} |a_{ik} - a_{jk}| \cdot c_k \geq 1, \tag{5}$$

$$A \cdot C \geq 1^T. \tag{6}$$

Inequality (5) implies that in order for C to be a separating code, the condition must be fulfilled for all pairs of vertices v_i and v_j, where $i \neq j$ and $i, j \in \{1, 2, \ldots, n\}$. Inequality (6) is required for the dominating property to be satisfied. Inequalities in (5) and (6) can

be grouped in a matrix $U = [u_{ij}]_{mxn}$, where $m = (n^2 - n)/2$. We can then define the augmented matrix,

$$I_G = \left[\frac{U}{A}\right]. \tag{7}$$

Observe that I_G consolidates both the separating and dominating constraints of the problem; we call I_G the identifying matrix of G. Thus, the integer programming model is given by the following:

$$\begin{array}{ll} \min & \sum_{i=1}^{n} c_i \\ \text{s.t.} & I_G \cdot C \geq 1^T \\ & c_i \in \{0,1\}, i \in \{1,2,\ldots,n\} \end{array} \tag{8}$$

The formulation described in (8) has all binary variables, and the goal is to find the minimum-size code. Although the identifying code problem differs from set covering in terms of the type of constraints, some of the ideas from set covering can be adapted to solve it. The integer programming model has $n^2 + n/2$ rows constraints, which can be numerous even for small graphs. For example, in a graph with 100 vertices, the number of constraints is approximately 5000. This makes it challenging to solve large-scale instances of the minimum identifying code problem. For this reason, the development of efficient algorithms to solve the problem is an important research topic, and new algorithms are needed.

3. Proposed Population-Based Local Search

In this section, we focus on the proposed approach for solving the identifying code problem. Population-based algorithms are general-purpose optimization methods that have been successfully applied to a wide range of combinatorial optimization problems [22]. Generally, population-based heuristics are strong at exploring the search space but weak at exploiting the solutions they find. Therefore, in this work, we employ a local search mechanism to augment the intensification ability. The flowchart of the proposed algorithm is shown in Figure 4.

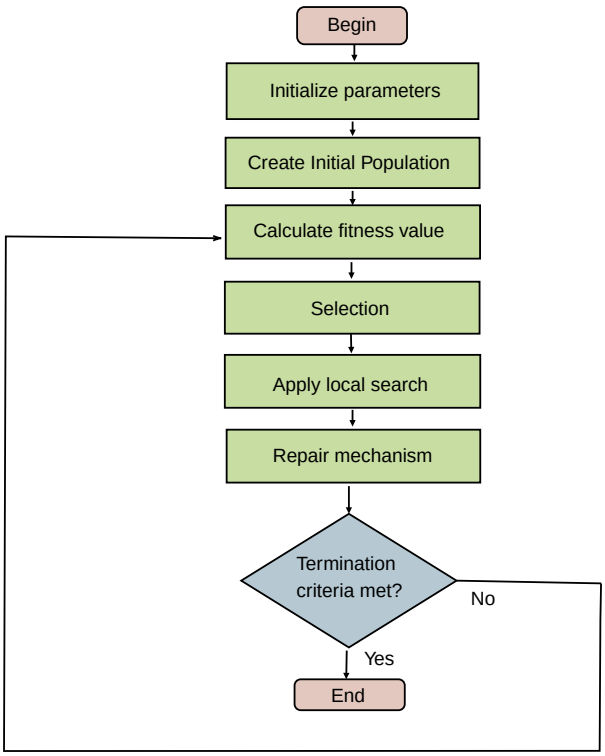


Figure 4. Flowchart of the proposed algorithm.

In the following subsections, we describe the solution representation, fitness function, initial population creation, selection, and repair mechanism of the proposed algorithm. The repair mechanism was inspired on the work by Xu et al. [18], but we have made some modifications to improve its performance. We also describe the local search procedure in detail and the update population strategy.

3.1. Solution Representation and Fitness Function

A solution is represented as a binary array of length equal to the number of vertices in the instance that needs to be solved. Specifically, a value of 1 in the lth element of the array indicates that vertex l is in the solution; a value of 0 indicates that it is not. An example is shown in Figure 5. Here, we can see that vertices 1, 3 and 4 belong to the solution, while vertices 2 and 5 do not.

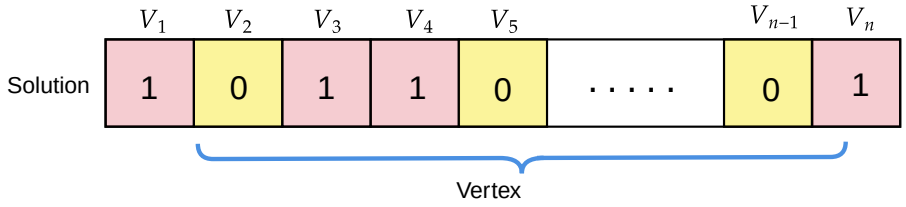


Figure 5. Solution encoding scheme.

The fitness function used to evaluate the population is identical to our optimization objective. That is, the fitness of a vector solution $C = (c_1, c_2, \ldots, c_n)$ is calculated by the following equation:

$$fitness = \sum_{i=1}^{n} c_i \tag{9}$$

In other words, the smaller the size of the code, the better the fitness of a solution. One challenge of using binary representation is that operators can produce infeasible solutions (i.e., solutions that violate one or more constraints). Several mechanisms exist for handling infeasible solutions such as penalty functions and repair mechanisms [23]. Penalty functions penalize the fitnesses of infeasible solutions without distorting the fitness landscape. Repair mechanisms, on the other hand, seek to design specific operators to transform infeasible solutions into feasible ones. We chose the repair mechanism approach due to its good results in the literature [18] and because it can be difficult to determine an effective penalty function [24,25]. More details about the repair mechanism can be found in Section 3.5.

3.2. Population Initialization

The algorithm begins by creating a set of feasible solutions to form the initial population. To ensure that the initial population is diverse and spread out across the solution space, a random initialization method is used. The pseudocode for creating an initial solution is detailed in Algorithm 1.

At first, every vertex in the solution is established as a codeword (Line 1). It is known that a set that includes all vertices of a graph is an identifying code [2]. Once all vertices are set, to generate a uniformly distributed population over the search space, each vertex has a 0.5 probability of being removed (Lines 2–4). If removing the vertex results in an identifying code, we proceed to the next vertex. Otherwise, the removed vertex is reinserted into the solution, and we proceed to consider another vertex (Lines 5–9).

After this step, the feasible solution may still contain non-essential or redundant vertices. A vertex is considered redundant if it can be deleted from the solution and still be an identifying code. Consequently, some unnecessary vertices are eliminated randomly (Lines 10–17). It is worth noting that not all redundant vertices are removed. This initial

redundancy in the solutions can provide more possibilities in the search process. This procedure is repeated N times until the initial population is fully populated.

Algorithm 1 Initial_Solution()

```
Generate C := V
for all vertex v in C do
    if rand() < 0.5 then
        C := C \ {v};
        if C is not an identifying code after removal then
            CS := CS ∪ {v};
        end if
    end if
end for
t := |C|
while t > 0 do
    v := Randomly select a vertex from the first vertices of C;
    if v is not essential then
        C := C \ {v};
        t := t − 1;
    end if
end while
return C;
```

3.3. Selection

The selection process determines which solution is chosen for being improved by the local search mechanism. The main idea is that the better a solution, the higher its likelihood of being selected. There are several selection mechanisms, but tournament selection is probably the most popular strategy due to its simplicity and efficiency [26].

This method involves randomly selecting n individuals from a population P; then, the selected individuals compete against each other. If $n = 2$, the process is called a binary tournament selection. In our algorithm, a binary tournament selection has yielded effective outcomes.

3.4. Developed Local Search Algorithm

In large and complex search spaces, population-based algorithms have been seen as search techniques that can find high-performance regions. However, they are not ideal for fine-tuning solutions [27]. This paper presents a local search method based on the concept of configuration checking to intelligently add and remove vertices from the solution. Local search methods show great performance on seeking good solutions by exploring the neighborhood structure. In the following subsections, we discuss the score function, the configuration checking mechanism, and the main local search procedure.

3.4.1. Scoring Function

To calculate the score value, denoted as $sc(v)$, we consider the frequency of all the rows in I_G, referring to the idea in [28]. Let $R(I_G) = \{r_1, r_2, \ldots, r_m\}$ be the set that contains all the row vectors of I_G and let $q(r)$ be an attribute for each r_i, $i \in \{1, 2, \ldots, m\}$ denotating the frequency of the ith row. For a row r of the constraints, if $I(r, v) = 1$, we say that "row r is covered by vertex v".

At the beginning, each row r is assigned a frequency of 1. After each iteration, the frequency of any row r that is not covered by the vertices in the candidate solution, CS, is increased by 1. This encourages the algorithm to find vertices that fulfill the dominating

and separating code constraints for those uncovered rows in the next iterations. Based on this idea, we propose the following scoring function:

$$sc(v) = \begin{cases} -\sum_{r \in C_1} q(r), & \text{if } v \in CS, \\ \sum_{r \in C_2} q(r), & \text{if } v \notin CS. \end{cases} \tag{10}$$

where C_1 is the set of rows which will become uncovered if vertex v is removed from CS, while C_2 is a set of rows whose current status is uncovered and will be covered by adding v to CS. Regardless of whether we are removing or inserting vertices, we prefer one with the highest $sc(v)$ value.

3.4.2. Configuration Checking Strategy

Configuration checking (CC) is a diversification strategy that was introduced in [29] to reduce the cycling problem in local search. The intuition behind CC is that by reducing cycles on local structures of the candidate solution, we may also reduce cycles on the whole candidate solution. CC has been shown to be effective in reducing cycling in local search for combinatorial optimization problems such as combinatorial interaction testing [30], minimum weighted vertex cover [31–33], constraint satisfaction problems such as satisfiability [34–36], maximum satisfiability [37], and a minimum weighted clique problem [38]. However, to the best of our knowledge, the configuration checking mechanism has not been adapted to the identifying code problem.

The CC strategy depends on the configuration concept. For each vertex, v, its configuration denotes the states of all the vertex in its neighborhood, $N(v)$. If the configuration of v has not changed since the last time it was removed from the candidate solution, CS, then it is not allowed to be added back to CS. To implement the CC strategy, a boolean array named $conf$ is generally employed, if $conf[v] = 1$, then v is authorized to be added to CS; otherwise, $conf[v] = 0$ and v is forbidden to be added to CS.

In this work, we used the improved CC strategy, *ES-CC*, proposed by Wang et al. [28] to update the $conf$ array after each iteration. As opposed to the traditional CC strategy, *ES-CC* considers the element or row states in addition to the neighboring states. A set R_{change} that includes all the rows of the I_G that change its state to respect to v (i.e., rows that were covered by v and are now uncovered after v is removed) must be defined. Also, the set P_v which contains all the rows of the I_G covered by v should be specified.

For the identifying code problem, the strategy can be adapted as follows. In the beginning, the value of $conf[v]$ is initialized as 1 for each vertex $v \in (G)$. Afterwards, if a vertex v has changed its state (i.e., being added or removed from CS), then the value of $conf[v]$ is assigned to 0. For each vertex $u \in N(v)$, if $R_{change} \cap P_v \neq \emptyset$, then $conf[u]$ is assigned to 1.

Considering the score function and the CC strategy, the following rules to remove and add vertices can be defined.

- Removing rule: Choose a vertex v with the highest $sc(v)$ value and update the configuration to reflect this change.
- Adding rule: Randomly select an uncovered row r and choose a vertex v that covers it. Vertex v should be allowed to be added to the solution with $conf[v] = 1$ and the highest $sc(v)$ value. Ties are broken randomly. Update the configuration to reflect this change.

3.4.3. Local Search Main Procedure

The pseudocode of the local search is shown in Algorithm 2. The procedure receives as input a candidate solution, CS, and the maximum number of local search iterations, max_iter. At the beginning, some variables are initialized: $iter$, $tabu$, $q(r)$ for all rows, while $conf$ and $sc(v)$ values are set for all vertices (Line 1). Also, the best solution found so far CS^* is established as CS (Line 2).

Algorithm 2 Local Search Procedure (CS,max_iter)

1: Initialize $iter$, $tabu$, $q(r)$ for all rows, $conf(v)$ and $sc(v)$ for all vertices;
2: $CS^* := CS$;
3: **while** $iter < max\_iter$ **do**
4: **if** all rows in I_G are covered **then**
5: **if** $CS < CS^*$ **then**
6: $CS^* := CS$;
7: $iter = 0$;
8: **end if**
9: $v :=$ a vertex selected based on **Removal Rule**;
10: $CS := CS \setminus \{v\}$
11: **end if**
12: $v :=$ a vertex selected based on **Removal Rule** and $v \notin tabu$;
13: $tabu := \emptyset$
14: $CS := CS \setminus \{v\}$;
15: $v :=$ a vertex selected based on **Adding Rule**;
16: $CS := CS \cup \{v\}$;
17: $tabu := tabu \cup \{v\}$;
18: **end while**
19: **return** CS^*;

While the maximum number of iterations is not reached, the solution is improved (Lines 13–18). If the algorithm arrives at a CS in which all vertices form an identifying code, then we update the best known solution CS^* and reduce the size of CS (Lines 5–10). We define that $CS < CS^*$ if for the associated vectors $CS = (c_1, c_2, \ldots, c_n)$ and $CS^* = (c_1^*, c_2^*, \ldots, c_n^*)$, it follows that $\sum_{i=1}^{n} c_i < \sum_{i=1}^{n} c_i^*$. To remove a vertex, the algorithm decides based on the highest $sc(v)$ value and prevents the vertex that has been added in the previous step from being removed with a tabu list (Lines 12–14). A new vertex is chosen by means of Adding Rule (Lines 15–16). Finally, the incumbent solution CS^* is returned (Line 19).

3.5. Repair Mechanism

Because infeasible solutions can occur during the search process, the following repair mechanism is applied. This procedure consists of two main steps. First, we verify that all dominating and separating constraints are fulfilled. If some constraints are missing, this means that some rows of the I_G matrix are uncovered. A row of matrix I_G is said to be uncovered if there is no vertex that is included in the solution that satisfies the row constraint. We proceed to compute the set of these uncovered rows. Then, we consider these rows one at a time in their natural order. With probability of 0.5, we select a vertex that covers the row in consideration and covers the greatest number of remaining uncovered rows. Otherwise, we randomly select a vertex from all the vertices that cover the row.

Our approach differs from previous ones for identifying codes, such as the one proposed in [18]. Previous approaches used a greedy policy to select the vertex that covers the row in consideration. Our approach combines a greedy policy that considers the minimum cost-to-benefit ratio with a restricted criterion of randomness. In our experience, this flexibility promotes higher-quality solutions.

After the first phase, there may be some redundant vertices in the solution. In order to obtain an irreducible code, as stated in [2], we define a procedure to delete them. A vertex is considered redundant if it can be deleted from the solution and still be an identifying code. Therefore, we iterate over the solution sequentially and seek for possible removals by flipping set bits and verifying if the solution is still feasible.

3.6. Update Population Strategy

An adequate population update strategy is essential to maintain diversity and explore promising areas of the search space. After applying the proposed local search and repair

mechanism, a retention policy should be used. This policy outlines whether a solution is good enough to join the population, and if so, which existing solution should be replaced.

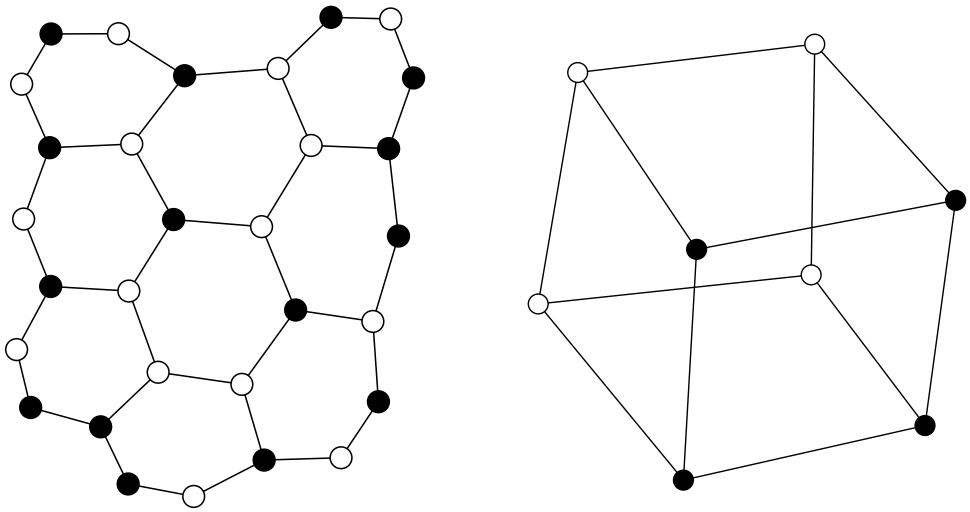
In nature, the fittest members in a population are more likely to survive. This is known as the elite retention strategy [26]. Inspired by this strategy, we retain the top elements in the population following a steady-state model. Let $|Pop|$ be the population size. During the search process, the population constantly maintains the top $|Pop|$ optimal solutions. After each iteration, if the fitness of the new solution found by local search is higher than the fitness of the worst solution, the new solution replaces the inferior one. To increase diversity, if the fitness of the new solution is equal to the fitness of the worst solution, the new solution replaces the worst solution at random. One advantage of this model is that the newly generated solution is immediately available for selection. In our experiments, we found that the strategy was able to improve convergence.

4. Computational Experiments

This section presents the results of the experiments conducted to evaluate the proposed algorithm on a variety of graphs. We compare our algorithm with other state-of-the-art heuristics and an integer programming approach.

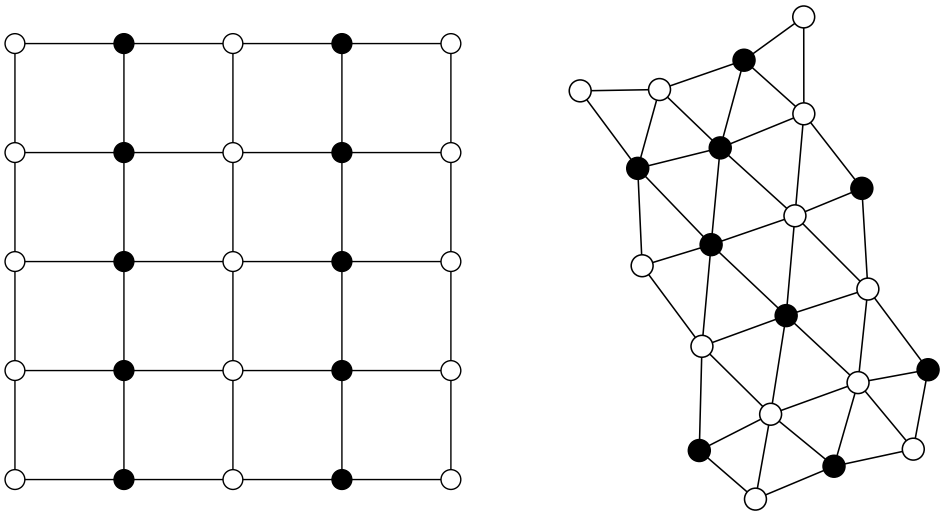
4.1. Test Instances

The computational experiments were carried out using different types of lattice graphs, namely square, triangular, and hexagonal grids. In previous studies, grids have been used to test algorithms for solving the identifying code problem. These studies have shown that grids can have complex landscapes for metaheuristics [18]. Also, the properties of lattices are well described in the literature, and they depict some essential components of real-world networks, such as roads [39]. We also explored the use of hypercube graphs due to their suitability as a network topology for communications [40]. To illustrate the different types of graphs employed in this paper, Figure 6 depicts some small graphs and their minimum identifying codes.



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

Figure 6. *Cont.*



(**c**) (**d**)

Figure 6. Identifying codes for different types of graphs: (**a**) 3×3 hexagonal grid. (**b**) 3 hypercube. (**c**) 5×5 square grid. (**d**) 5×5 triangular grid. The black vertices are the codewords.

4.2. Data Preprocessing

The computational complexity of solving the identifying code problem depends primarily on the number of rows and columns in the I_G. To reduce the dimension of the test instances, we implemented some of the reduction procedures outlined by Beasley [41]. Specifically, there are two basic steps: eliminating redundant rows (constraints) and eliminating columns (vertices) that are subsets of other columns.

Table 1 presents the details of the reduced instances for each type of graph. Input data for these well-known graph families can be generated using open-source frameworks such as Octave (https://octave.org) and SageMath (https://www.sagemath.org). In our work, we used SageMath to generate the adjacency matrices of the graphs of interest, which we then exported to text files that could be read by our C++ program. The type of graph, its dimensions, and the name of the instance are specified in columns 1, 2, and 3, respectively. The information related to the vertices is reported in columns 4–7, while the information regarding constraints is reported in columns 8–10. It is worth noting that increasing the size of the grids and hypercubes significantly increases the search space for each instance.

Table 1. Test instances information.

Graph	Dimension	Name	Vertices			Constraints		
			Total	Removed	Core	Total	Removed	Core
Hexagonal	5×5	H1	70	0	70	2485	2202	283
	10×10	H2	240	0	240	28,929	27,787	1133
	15×15	H3	510	0	510	130,305	127,772	2533
	20×20	H4	880	0	880	387,640	383,157	4483
Hypercube	5	Q1	32	0	32	528	2080	272
	6	Q2	64	0	64	2080	1344	736
	7	Q3	128	0	128	8256	6336	1920
	8	Q4	256	0	256	32,896	28,032	4864
Square	5×5	S1	25	0	25	325	206	119
	10×10	S2	100	0	100	5050	4456	594
	15×15	S3	225	0	225	25,425	24,006	1419
	20×20	S4	400	0	400	80,200	77,606	2594
	25×25	S5	625	0	625	195,625	191,506	4119

Table 1. *Cont.*

Graph	Dimension	Name	Vertices			Constraints		
			Total	Removed	Core	Total	Removed	Core
Triangular	5×5	T1	21	0	21	231	173	58
	10×10	T2	66	0	66	2211	1952	259
	15×15	T3	136	0	136	9316	8665	651
	20×20	T4	231	0	231	26,796	25,570	1226
	25×25	T5	351	0	351	61,776	59,800	1976

As can be observed, this method does not allow us to remove any vertex from the solution a priori. In our experiments, all algorithms only consider the essential constraints, or rows, in the core. It is worth noting that working on this reduced scheme is equivalent to working on the original one. In our experience, this preprocessing step does not significantly impact the quality of the solutions, but it reduces the overall running time of the algorithm.

4.3. Experimental Settings

The proposed approach is compared to several state-of-the-art metaheuristics: the genetic algorithm for the identifying code problem by Xu et al. [18] (GA), artificial bee colony (ABC), firefly algorithm (FFA), and cat swarm optimization (CSO). These algorithms have shown great performance in a wide variety of discrete problems. In the remainder of this paper, the following acronyms will be used to identify each algorithm: GA, ABC, FFA, CSO, and PB-LS (for the population-based local search algorithm described in Section 3). We also used the commercial software Lingo 19.0 to solve the integer program and compare the results to those obtained with the metaheuristics.

The techniques were coded in the C++ language and deployed on a desktop computer with a Pentium i7 (3.4 GHz) processor, 32 GB of RAM, and Microsoft Windows 10 operating system. The algorithms were compiled using MinGW 8.0.

Regarding the operating parameter settings, there are two main parameters that need to be tuned: the population size N and max_iter, which controls the maximum number of iterations in the local search stage. The parameters were tuned using a brute force approach, as described in [42]. These parameter values provide good results, although they may not be optimal for all instances. To ensure a fair comparison with the standard algorithms, all parameter values are consistent, and the same parameters that appear in different algorithms have the same values. The remaining parameters for GA, ABC, FFA, and CSO were obtained from [18,43–45], respectively. The parameters for the proposed algorithm are shown in Table 2.

Table 2. Parameters of the proposed algorithm.

Parameter	Description	Value
N	Population size	100
max_iter	Local search iterations	150

To ensure an impartial comparison of the internals of each metaheuristic, the initialization procedure from Section 3.2 was used for all algorithms. To handle constraints, the repair mechanism as described in Section 3.5 was applied uniformly across all techniques.

4.4. Experiments

Following previous works on identifying codes [18] and other related graph and set problems, we used a time limit as the stopping criterion for all algorithms. The cutoff time was set to 900 s (15 min) for all instances and all algorithms. To address the stochastic nature of heuristic techniques, 30 independent runs were performed for each algorithm. Additionally, the same seeds were used for all algorithms to ensure that all techniques started with the same initial solutions. Because the Lingo solver consistently produces the same result across different executions, we only recorded the minimum value found.

Tables 3–6 show the detailed results obtained by different instances of our proposed PB-LS. The algorithms are compared in terms of their minimum solution values (min), average solution values (avg) over 30 executions, and relative percentage deviation (RPD), which are given by the following:

$$RPD = \frac{min - best}{best} \times 100, \tag{11}$$

where *best* is the top solution obtained from all runs and algorithms for each instance. The best results per table row and instance are indicated in bold font.

Table 3 shows the results from the hexagonal lattice graphs. Our algorithm outperformed all others in three out of four instances, both in terms of the minimum and average solutions found. Only in instance H1 did GA, ABC, FFA, CSO, and Lingo find the best solution. ABC was the second-best competitor in terms of average solution. Instances H3 and H4 are among the most complex due to their dimensions, and PB-LS was able to find the best solutions for these instances.

Table 4 summarizes the results obtained from the hypercube test instances. Our algorithm achieved the best results on two out of four instances. It is worth noting that instances Q1 and Q2 have the fewest number of vertices and constraints among all the test instances.

Table 5 shows the results obtained from the grid graphs. Our algorithm had a good performance, achieving four out of five of the best solutions. GA, ABC, FFA, CSO and Lingo had similar results in terms of the minimum solution found.

Table 6 shows the results obtained from the triangular lattice test instances. Here, PB-LS was able to find three out of five of the best solutions. It is worth noting that both GA and FFA struggled with this type of graph, as reflected by their high RPD values.

Previous experiments and discussions have shown that the PB-LS algorithm for the identifying code problem outperforms competing algorithms, especially on large graphs.

Table 3. Computational experiments on hexagonal lattice test instances.

Instance		**H1**	**H2**	**H3**	**H4**
	Best	32	110	236	416
Our approach					
PB-LS	Min	**32**	**110**	**236**	**416**
	Avg	**32**	**110.4**	**237.5**	**418.6**
	RPD	**0**	**0**	**0**	**0**
Other approaches					
GA	Min	**32**	115	247	433
	Avg	32.7	115.8	248.1	433.7
	RPD	**0**	4.5	4.6	4
ABC	Min	**32**	112	243	425
	Avg	**32**	112.5	243.8	426.6
	RPD	**0**	1.8	2.9	2.16
FFA	Min	**32**	116	249	434
	Avg	32.7	116.2	249.6	434
	RPD	**0**	5.4	5.5	4.32
CSO	Min	**32**	111	248	433
	Avg	32.2	113.8	248.9	435.5
	RPD	**0**	0.9	5.1	4.01
Lingo	Min	**32**	114	251	432
	Avg	-	-	-	-
	RPD	**0**	3.6	6.4	3.8

Table 4. Computational experiments on hypercube test instances.

Instance		**Q1**	**Q2**	**Q3**	**Q4**
	Best	10	19	34	63
Our approach					
PB-LS	Min	**10**	**19**	**34**	**63**
	Avg	**10**	**19**	**34**	**63.5**
	RPD	**0**	**0**	**0**	**0**
Other approaches					
GA	Min	**10**	**19**	35	67
	Avg	**10**	**19**	36.1	67.8
	RPD	**0**	**0**	2.9	6.3
ABC	Min	**10**	**19**	35	66
	Avg	**10**	**19**	35	66.4
	RPD	**0**	**0**	2.9	4.7
FFA	Min	**10**	**19**	35	66
	Avg	**10**	**19**	35	66.5
	RPD	**0**	**0**	2.9	4.7
CSO	Min	**10**	**19**	35	67
	Avg	**10**	**19**	35.8	68.3
	RPD	**0**	**0**	2.9	6.3
Lingo	Min	**10**	**19**	35	65
	Avg	-	-	-	-
	RPD	**0**	**0**	2.9	3.2

Table 5. Computational experiments on square lattice test instances.

Instance		**S1**	**S2**	**S3**	**S4**	**S5**
	Best	10	39	87	158	246
Our approach						
PB-LS	Min	**10**	**39**	**87**	**158**	**246**
	Avg	**10**	**39.6**	**88.4**	**158.5**	**247.8**
	RPD	**0**	**0**	**0**	**0**	**0**
Other approaches						
GA	Min	**10**	40	90	164	260
	Avg	**10**	40.7	92.5	166	260.7
	RPD	**0**	2.6	3.4	3.8	5.7
ABC	Min	**10**	40	89	160	255
	Avg	**10**	40	89.8	161.7	256.3
	RPD	**0**	2.5	2.2	1.2	3.6
FFA	Min	**10**	40	92	166	258
	Avg	**10**	40.9	92.7	166.5	259.8
	RPD	**0**	2.5	5.7	5.1	4.8
CSO	Min	**10**	40	90	164	259
	Avg	**10**	40.6	90.7	164.4	260.2
	RPD	**0**	2.5	3.4	3.7	5.2
Lingo	Min	**10**	40	90	160	250
	Avg	-	-	-	-	-
	RPD	**0**	2.5	3.4	1.2	1.6

Table 6. Computational experiments on triangular lattice test instances.

Instance		T1	T2	T3	T4	T5
	Best	9	23	46	76	116
Our approach						
PB-LS	Min	**9**	**23**	**46**	**76**	**116**
	Avg	**9**	**23**	**46.7**	**78.1**	**116.3**
	RPD	**0**	**0**	**0**	**0**	**0**
Other approaches						
GA	Min	**9**	**23**	48	82	124
	Avg	**9**	23.8	48.9	82.4	126.3
	RPD	**0**	**0**	4.3	6.5	6.9
ABC	Min	**9**	**23**	47	78	121
	Avg	**9**	**23**	47.2	78.8	121.6
	RPD	**0**	**0**	2.2	2.6	4.3
FFA	Min	**9**	**23**	49	84	127
	Avg	**9**	23.7	49	84	127
	RPD	**0**	**0**	6.5	10.5	9.4
CSO	Min	**9**	**23**	47	78	120
	Avg	**9**	24.2	48.2	80.7	123.2
	RPD	**0**	**0**	2.2	2.6	3.4
Lingo	Min	**9**	**23**	47	80	120
	Avg	-	-	-	-	-
	RPD	**0**	**0**	2.2	5.3	3.4

5. Conclusions

Identifying codes are a concept borrowed from graph theory that can be used to solve problems such as fault detection and location detection. The computational complexity of the problem is NP-complete, making it challenging to find high-quality solutions using traditional techniques. In this scenario, heuristics are a viable alternative that can find good solutions in a reasonable amount of time.

In this paper, we propose a new population-based local search algorithm (PB-LS) to find an identifying code with minimum cost. Our PB-LS incorporates a local search mechanism with the concept of configuration checking to avoid cycles and achieve high-quality solutions.

The effectiveness of the proposed technique was evaluated using four different graph families with different vertex sizes. We compared the algorithm to other state-of-the-art metaheuristics, including GA, ABC, FFA, and CSO, in terms of minimum and average code size. We also used the commercial software Lingo to solve the integer program. The experimental results showed that the proposed algorithm was superior to other algorithms on most instances. However, the PB-LS is considered a strong alternative for finding minimum identifying codes.

There are several potential directions for future research. In particular, we believe that exploring advanced constraint handling methods such as adaptive penalty functions can improve solution quality. We are also interested in combining other population-based techniques and local search methods to create more effective algorithms for solving the identifying code problem. Additionally, we plan to explore the use of metaheuristics in conjunction with other approaches, such as mathematical programming, constraint programming, or machine learning. Finally, we plan to adapt the proposed strategy to solve other interesting problems. We believe that some of the ideas devised in this work can be generalized and applied to solve other known NP-hard problems in graph theory.

Author Contributions: Conceptualization, D.G.-M.; methodology, A.L.-C. and D.G.-M.; software, A.L.-C.; validation, A.L.-C. and D.G.-M.; formal analysis, A.L.-C. and D.G.-M.; investigation, A.L.-C. and D.G.-M.; data curation, A.L.-C.; writing—original draft preparation, A.L.-C. and D.G.-M.; writing—review and editing, A.L.-C. and D.G.-M.; visualization, A.L.-C. and D.G.-M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by CONAHCYT under project CB-2017-2018 No. 45208.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Karpovsky, M.; Chakrabarty, K.; Levitin, L. On a new class of codes for identifying vertices in graphs. *IEEE Trans. Inform. Theory* **1998**, *44*, 599–611.
2. Ray, S.; Ungrangsi, R.; De Pellegrini, F.; Trachtenberg, A.; Starobinski, D. Robust location detection in emergency sensor networks. *IEEE J. Sel. Areas Commun.* **2004**, *22*, 1016–1025.
3. Charon, I.; Hudry, O.; Lobstein, A. Identifying codes with small radius in some infinite regular graphs. *Electron. J. Comb.* **2002**, *9*, R11. [CrossRef] [PubMed]
4. Bertrand, N.; Charon, I.; Hudry, O.; Lobstein, A. 1-identifying codes on trees. *Australas. J. Comb.* **2005**, *31*, 21–36.
5. Chen, C.; Lu, C.; Miao, Z. Identifying codes and locating–dominating sets on paths and cycles. *Discret. Appl. Math.* **2011**, *159*, 1540–1547.
6. Bertrand, N.; Charon, I.; Hudry, O.; Lobstein, A. Identifying and locating-dominating codes on chains and cycles. *Eur. J. Comb.* **2004**, *25*, 969–987.
7. Gravier, S.; Moncel, J.; Semri, A. Identifying codes of cycles. *Eur. J. Comb.* **2006**, *27*, 767–776.
8. Moncel, J. Monotonicity of the minimum cardinality of an identifying code in the hypercube. *Discret. Appl. Math.* **2006**, *154*, 898–899.
9. Karpovsky, M.; Chakrabarty, K.; Levitin, L.; Avreky, D. On the covering of vertices for fault diagnosis in hypercubes. *Inform. Process. Lett.* **1999**, *69*, 99–103
10. Ben-Haim, Y.; Litsyn, S. Exact minimum density of codes identifying vertices in the square grid. *SIAM J. Discret. Math.* **2005**, *19*, 69–82.
11. Cohen, G.; Maffray, F.; Manoussakis, Y.; Slater, P. Watching Systems, Identifying, Locating-Dominating and Discriminatiing Codes in Graphs. Available online: https://dragazo.github.io/bibdom/main.pdf (accessed on 9 August 2023).
12. Laifenfeld, A.; Trachtenberg, R.C.; Starobinski, D. Joint monitoring and routing in wireless sensor networks using robust identifying codes. *Mob. Netw. Appl.* **2009**, *14*, 415–432.
13. Haynes, T.; Knisley, D.; Seier, E.; Zou, D. A quantitative analysis of secondary RNA structure using domination based parameters on trees. *BMC Bioinform.* **2006**, *7*, 108.
14. Laifenfeld, M.; Trachtenberg, A. Identifying codes and covering problems. *IEEE Trans. Inf. Theory* **2008**, *54*, 3929–3950.
15. Suomela, J. Approximability of identifying codes and locating–dominating codes. *Inf. Process. Lett.* **2007**, *103*, 28–33.
16. Xiao, Y.; Hadjicostis, C.; Thulasiraman, K. The d-identifying codes problem for vertex identification in graphs: Probabilistic analysis and an approximation algorithm. In Proceedings of the International Computing and Combinatorics Conference, Taipei, Taiwan, 15–18 August 2006.
17. Qi, H.; Iyengar, S.S.; Chakrabarty, K. Distributed sensor networks—A review of recent research. *J. Frankl. Inst.* **2001**, *338*, 655–668.
18. Xu, Y.C.; Xiao, R.B. Solving the Identifying Code Problem by a Genetic Algorithm. *IEEE Trans. Syst. Man Cybern.-Part A Syst. Hum.* **2007**, *37*, 41–46.
19. Horan, V.; Adachi, S.; Bak, S. A comparison of approaches for finding minimum identifying codes on graphs. *Quantum Inf. Process.* **2016**, *15*, 1827–1848.
20. Charon, I.; Hudry, O. Minimizing the size of an identifying or locating-dominating code in a graph is NP-hard. *Theor. Comput. Sci.* **2003**, *290*, 2109–2120.
21. Xu, Y.C.; Xiao, R.B. Identifying code for directed graph. In Proceedings of the Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007), Qingdao, China, 30 July–1 August 2007.
22. Weinand, J.M.; Sörensen, K.; San Segundo, P.; Kleinebrahm, M.; McKenna, R. Research trends in combinatorial optimization. *Int. Trans. Oper. Res.* **2022**, *29*, 667–705.
23. Rahimi, I.; Gandomi, A.H.; Chen, F.; Mezura-Montes, E. A Review on Constraint Handling Techniques for Population-based Algorithms: From single-objective to multi-objective optimization. *Arch. Comput. Methods Eng.* **2023**, *30*, 2181–2209.
24. Back, T.; Schutz, M.; Khuri, S. A comparative study of a penalty function, a repair heuristic, and stochastic operators with the set-covering problem. In Proceedings of the European Conference on Artificial Evolution, Brest, France, 4–6 September 1995; pp. 320–332.
25. Bilal, N.; Galinier, P.; Guibault, F. A new formulation of the set covering problem for metaheuristic approaches. *Int. Sch. Res. Not.* **2013**, *2013*, 203032.
26. Goldberg, D.E.; Deb, K. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithm*; Whitley, D., Ed.; Elsevier: Amsterdam, The Netherlands, 1991; pp. 69–93.

27. Blum, C.; Roli, A. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.* **2003**, *35*, 268–308.
28. Wang, Y.; Pan, S.; Al-Shihabi, S.; Zhou, J.; Yang, N.; Yin, M. An improved configuration checking-based algorithm for the unicost set covering problem. *Eur. J. Oper. Res.* **2021**, *294*, 476–491.
29. Cai, S.; Su, K.; Sattar, A. Local search with edge weighting and configuration checking heuristics for minimum vertex cover. *Artif. Intell.* **2011**, *175*, 1672–1696.
30. Fu, Y.; Lei, Z.; Cai, S.; Lin, J.; Wang, H. WCA: A weighting local search for constrained combinatorial test optimization. *Inf. Softw. Technol.* **2020**, *122*, 1106288.
31. Li, R.; Hu, S.; Zhang, H.; Yin, M. An efficient local search framework for the minimum weighted vertex cover problem. *Inf. Sci.* **2016**, *372*, 428–445.
32. Li, R.; Hu, S.; Cai, S.; Gao, J.; Wang, Y.; Yin, M. NuMWVC: A novel local search for minimum weighted vertex cover problem. *J. Oper. Res. Soc.* **2019**, *71*, 1498–1509.
33. Zhou, Y.; Li, J.; Liu, Y.; Lv, S.; Lai, Y.; Wang, J. Improved Memetic Algorithm for Solving the Minimum Weight Vertex Independent Dominating Set. *Mathematics* **2020**, *8*, 1155.
34. Cai, S.W.; Su, K.L. Local search for Boolean Satisfiability with configuration checking and subscore. *Artif. Intell.* **2013**, *204*, 75–98.
35. Cai, S.; Su, K. Configuration checking with aspiration in local search for sat. In Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, Toronto, ON, Canada, 22–26 July 2012.
36. Luo, C.; Cai, S.; Su, K.; Wu, W. Clause states based configuration checking in local search for satisfiability. *IEEE Trans. Cybern.* **2015**, *45*, 1014–1027.
37. Luo, C.; Cai, S.; Su, K.; Wu, W. Ccls: An efficient local search algorithm for weighted maximum satisfiability. *IEEE Trans. Comput.* **2015**, *64*, 1830–1843.
38. Wang, Y.Y.; Cai, S.W.; Yin, M.H. Two efficient local search algorithms for maximum weight clique problem. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016.
39. Church, R.L.; Wang, S. Solving the p-median problem on regular and lattice networks. *Comput. Oper. Res.* **2020**, *123*, 105057.
40. Laing, A.K.; Krumme, D.W. Optimal permutation routing for low-dimensional hypercubes. *Netw. Int. J.* **2010**, *55*, 149–167.
41. Beasley, J.E. An algorithm for set covering problem. *Eur. J. Oper. Res.* **1987**, *31*, 85–93. [CrossRef]
42. Birattari, M. *Tuning Metaheuristics: A Machine Learning Perspective*; Springer: Berlin/Heidelberg, Germany, 2009.
43. Crawford, B.; Soto, R.; Cuesta, R.; Paredes, F.T. Application of the artificial bee colony algorithm for solving the set covering problem. *Sci. World J.* **2014**, *2014*, 189164.
44. Crawford, B.; Soto, R.; Suárez, M.O.; Paredes, F.; Johnson, F. Binary firefly algorithm for the set covering problem. In Proceedings of the 2014 9th Iberian Conference on Information Systems and Technologies (CISTI), Barcelona, Spain, 18–21 June 2014.
45. Crawford, B.; Soto, R., Berríos; N., Johnson; F.; Paredes, F. Binary cat swarm optimization for the set covering problem. In Proceedings of the 2015 10th Iberian Conference on Information Systems and Technologies (CISTI), Aveiro, Portugal, 17–20 June 2015; pp. 1–4.