Exploratory Datamorphic Testing of Classification Applications

Hong Zhu and Ian Bayley
Oxford Brookes University
Oxford OX33 1HX, UK
(hzhuibayley)@brookes.ac.uk

ABSTRACT
Testing has been widely recognised as difficult for AI applications. This paper proposes a set of testing strategies for testing machine learning applications in the framework of the datamorphism testing methodology. In these strategies, testing aims at exploring the data space of a classification or clustering application to discover the boundaries between classes that the machine learning application defines. This enables the tester to understand precisely the behaviour and function of the software under test. In the paper, three variants of exploratory strategies are presented, with the algorithms as implemented in the automated datamorphic testing tool Morphy. The correctness of these algorithms are formally proved. The paper also reports the results of some controlled experiments with Morphy that study the factors that effect the test effectiveness of the strategies.

CCS CONCEPTS
• Software and its engineering → Software notations and tools;
• Computing methodologies → Artificial intelligence; Machine learning.

KEYWORDS
Artificial intelligence, Software testing, Automated software testing, Test method, Testing tools, Datamorphic testing, Exploratory testing, Test strategies

ACM Reference Format:

1 INTRODUCTION
It is widely recognised that the generation of test data for AI applications is prohibitively expensive [11]. Checking the correctness of a test result is also notoriously difficult, if not completely impossible [7, 15]. Moreover, existing testing techniques for measuring test coverage and the automation of testing activities and processes are not directly applicable [20]. Testing AI applications is therefore a grave challenge for software engineering [2]. Developing novel approaches to test AI applications is highly desirable [4].

In [20, 21], we proposed a method called datamorphic testing for testing AI applications and reported a case study with face recognition applications. The method is further developed in [17, 18], in which the notion of test morphisms is introduced, an automated testing tools called Morphy is reported and a set of test strategies are formally defined and implemented. The case studies reported in [17] shows that test strategies can significantly improve the automation for testing AI applications. This paper presents another set of datamorphic test strategies specifically designed for testing the classification and clustering type of AI applications, which classify objects and entities according to their features and attributes. Classification and clustering are one of the largest categories of AI applications, and many AI techniques such as machine learning and data analytics generate applications of this category [1, 5, 8].

The proposed test strategies are based on the idea of exploratory testing in which testers interact with the application and use the information the application provides to change the course of testing in order to explore the application’s functionality [12]. It is different from testing for verification and validation, which aims to confirm the correctness of the software under test with respect to a given specification. In contrast, exploratory testing treats the software under test as an object unknown and regards software testing as a series of experiments with the software aimed at discovering its functions and features. Moreover, the traditional verification and validation testing methods regard test cases as independent from each other. In contrast, exploratory testing uses the result of the previous test cases to guide its choice of the next test case in order to maximise its effectiveness in the process of searching for useful information.

The paper is organised as follows. Section 2 briefly reviews the datamorphic testing method, the automated testing tool Morphy, and the basic concepts of classification applications. Section 3 defines a set of three exploration strategies and illustrate their use with an example. Section 4 reports the experiments with these strategies on their performances. Section 5 concludes the paper with a discussion of related work and future work.

2 PRELIMINARIES
In this section, we briefly review the datamorphic testing method and the testing tool Morphy, and clustering and classification techniques to set the context of the paper.

2.1 Overview of Datamorphic Testing Method
In the datamorphic software testing method [17], software artefacts involved in testing are classified into two types: entities and morphisms.
Test entities are objects and data that are used and/or generated in testing. These include test cases, test suites/sets, the programs under test, and test reports, etc.

Test morphisms are mappings between entities. They generate and transform test entities to achieve testing objectives. They can be implemented as test code and invoked to perform test activities and composed to form test processes. The following are the test morphisms recognised by the datamorphic test tool Morphy:

- **Test set creators** create sets of test cases. They are called seed test case makers in [16, 21]. A typical example is random test case generators like fuzzers [10].
- **Datamorphisms** are mappings from existing test cases to new test cases. They are called data mutation operators in [9].
- **Metamorphisms** are mappings from test cases to Boolean values that assert their correctness. They are test oracles. Formal specifications and metamorphic relations in metamorphic testing [3, 7] can also be used as metamorphisms.
- **Test case metrics** are mappings from test cases to real numbers. They measure test cases giving, for example, the similarity of a test case to the others in the test set.
- **Test case filters** are mappings from test cases to truth values. They can be used, for example, to decide whether a test case should be included in a test set.
- **Test set metrics** are mappings from test sets to real numbers. They measure the test set quality, such as its code coverage [19].
- **Test set filters** are mappings from test sets to sets of test cases for example, they may remove some test cases from a test set for regression testing.
- **Test executors** execute the program under test on test cases and receive the outputs from the program. They are mappings from a piece of program to a mapping from input data to output. That is, they are functors in category theory.
- **Test analysers** analyse test sets and generate test reports.

A test system $\mathcal{T} = (\mathcal{E}, \mathcal{M})$ in datamorphic testing consists of a set $\mathcal{E}$ of test entities and a set $\mathcal{M}$ of test morphisms. In Morphy [17], a test system is specified as a Java class that declares a set of attributes as test entities and a set of methods as test morphisms.

Given a test specification, Morphy provides testing facilities to automate testing at three different levels. At the lowest level, various test activities can be performed by invoking test morphisms via a click of buttons on Morphy’s GUI. At the medium level, Morphy implements various test strategies to perform complex testing activities through combinations and compositions of test morphisms. At the highest level, test processes are automated by recording, editing and replaying test scripts that consist of a sequence of invocations of test morphisms and strategies.

Test strategies are complex combinations of test morphisms designed to achieve test automation. Three sets of test strategies have been implemented in Morphy:

- **Mutant combination**: combining datamorphisms to generate mutant test cases; see [17].
- **Domain exploration**: searching for the borders between clusters/subdomains of the input space;
- **Test set optimisation**: optimising test sets by employing genetic algorithms.

This paper focuses on domain exploration strategies, which will be defined in Section 3. Those strategies that employ genetic algorithms to optimise test sets will be reported in another paper.

### 2.2 Classification Applications

Clustering as a data mining and machine learning problem is the partitioning of a given set of data points into groups containing similar data points. However, clustering does not only partition the data in the given data set, but also makes it possible to put new data into the right groups. The key concept of clustering is similarity between data points, which is defined formally in the form of a similarity or distance function on the data space. Two pieces of data that are similar to each other should be put into the same group, while the data that are dissimilar should be placed in different groups. Whereas clustering is unsupervised learning, classification is supervised learning. Given a number of examples of data points and their classifications, it learns how to assign data to groups [1, 5, 8].

In both clustering and classification, the result is a program $P$ that maps from the data space $D$ into a number of groups $G$. We say that $P$ is a classification application. We will write $P(x)$ to denote the output of $P$ on an input $x \in D$, and call $P(x)$ the classification of $x$ by $P$. We also assume that there is a function $\text{dist} : D \times D \rightarrow \mathbb{R}^+$ measuring the distances between any two points $x$ and $y$ in the space $D$ such that:

- $\forall x \in D, \text{dist}(x, x) = 0$;
- $\forall x, y \in D, \text{dist}(x, y) \geq 0$;
- $\forall x, y \in D, \text{dist}(x, y) = \text{dist}(y, x)$;
- $\forall x, y, z \in D, \text{dist}(x, y) + \text{dist}(y, z) \geq \text{dist}(x, z)$.

The distance function measures the similarity between data points in that the smaller the distance between two points the more similar they are.

For a classification program, it is crucial to classify data into correct classes. However, the borders between classes are often unknown if the classification program is obtained through machine learning and data mining. The goal of the exploration strategies proposed in this paper is to find a set of data pairs that represents the borders between classes. Thus, we introduce the notion of Pareto front of the classification as defined by the program $P$ under test.

**Definition 1.** *Pareto Front of Classification*

Let $P : D \rightarrow G$ be a classification program, $\text{dist} : D \times D \rightarrow \mathbb{R}$ be a distance metric defined on the input space $D$, and $\delta > 0$ be a given real number. A set $\langle a_i, b_i \rangle_{i=1, \ldots, n}$ of data pairs is a Pareto front of the classes of $D$ according to $P$ with respect to $\text{dist}$ and $\delta$, if for all $i = \ldots, n$, $P(a_i) \neq P(b_i)$ and $\text{dist}(a_i, b_i) \leq \delta$. □

A Pareto front can show accurately the borders between the classes, thus help testers to determine whether the classification is correct or not.

### 2.3 Exploratory Test Systems

To apply an exploratory test strategy to a classification program $P : D \rightarrow G$ with a distance function $\text{dist}$, we assume that the test system $\mathcal{T} = (\mathcal{E}, \mathcal{M})$ has the following properties.
This section presents the algorithms for three different exploratory strategies for testing clustering and classification applications. We also prove their correctness and illustrate their behaviour by using the running example given in the previous section.

1. In this example, data points \( x \) and \( y \) are classified in a two-dimensional continuous space. We use the midpoint method to calculate the geometric midpoint between \( x \) and \( y \). It is called the midpoint method.

Note that, for all \( x, y \in D \) and \( z = m(x, y) \in D \), we have:

\[
(P(x) \neq P(y)) \Rightarrow (P(x) \neq P(z)) \lor (P(y) \neq P(z)).
\]

Informally, if the program \( P \) under test classifies \( x \) and \( y \) into different classes, the midpoint between \( x \) and \( y \) must be either not in the same class as \( x \) or not in the same class as \( y \).

2.4 The Running Example

In Section 3, we will use the following simple classification program as a running example to illustrate the exploration strategies. It classifies the points in a two-dimensional continuous space \([0, 2\pi] \times [-1, 1]\) into three classes: red, black and blue as illustrated in Figure 1. In this example, data points \( x \) and \( y \) is a pair of Pareto Front between black and red classes, if \( x \) is red and \( y \) is black and they are very close to each other. Such pairs can show accurately the borders between the classes, thus help testers to determine whether the classification is correct or not.

![Figure 1: Data Space of the Running Example](image)

The Figure 2 gives the traversal and midpoint methods in the Morphy test specification. The midpoint method \( mid(x, y) \) calculates the geometric midpoint between \( x \) and \( y \).

It is easy to see that the running example forms an exploratory test system with the following distance function.

\[
Eucl((x_1, x_2), (y_1, y_2)) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}
\]

3 EXPLORATION STRATEGIES

This section presents the algorithms for three different exploratory strategies for testing clustering and classification applications. We also prove their correctness and illustrate their behaviour by using the running example given in the previous section.

1. The set \( \mathcal{M} \) of morphisms contains a test executor \( Exep(x) \) that executes the program \( P \) under test on a test case \( x \) and receives the output of \( P \); that is \( Exep(x) = P(x) \). In the sequel, we will write \( P(x) \) for \( Exep(x) \) for the sake of simplicity.

2. There is a set \( W \subseteq \mathcal{M} \) of unary datamorphisms defined on \( D \). Informally, for each \( w \in W \) and \( x \in D \), \( w(x), w^2(x), \ldots, w^n(x) \) generates a sequence of different data points in \( D \), where \( w^1(x) = w(x) \), \( w^{n+1}(x) = w(w^n(x)) \). These datamorphisms are called traversal methods.

3. There is also a binary datamorphism \( m \in \mathcal{M} \) such that for all \( x, y \in D \), \( dist(x, z) < dist(x, y) \) and \( dist(y, z) < dist(x, y) \), where \( z = m(x, y) \in D \). Informally, the datamorphism \( m \) calculates a point between \( x \) and \( y \) and it is called the midpoint method.

(1) The set \( \mathcal{M} \) of morphisms contains a test executor \( Exep(x) \) that executes the program \( P \) under test on a test case \( x \) and receives the output of \( P \); that is \( Exep(x) = P(x) \). In the sequel, we will write \( P(x) \) for \( Exep(x) \) for the sake of simplicity.

(2) There is a set \( W \subseteq \mathcal{M} \) of unary datamorphisms defined on \( D \). Informally, for each \( w \in W \) and \( x \in D \), \( w(x), w^2(x), \ldots, w^n(x) \) generates a sequence of different data points in \( D \), where \( w^1(x) = w(x) \), \( w^{n+1}(x) = w(w^n(x)) \). These datamorphisms are called traversal methods.

(3) There is also a binary datamorphism \( m \in \mathcal{M} \) such that for all \( x, y \in D \), \( dist(x, z) < dist(x, y) \) and \( dist(y, z) < dist(x, y) \), where \( z = m(x, y) \in D \). Informally, the datamorphism \( m \) calculates a point between \( x \) and \( y \) and it is called the midpoint method.

Note that, for all \( x, y \in D \) and \( z = m(x, y) \in D \), we have:

\[
(P(x) \neq P(y)) \Rightarrow (P(x) \neq P(z)) \lor (P(y) \neq P(z)).
\]

Informally, if the program \( P \) under test classifies \( x \) and \( y \) into different classes, the midpoint between \( x \) and \( y \) must be either not in the same class as \( x \) or not in the same class as \( y \).

2.4 The Running Example

In Section 3, we will use the following simple classification program as a running example to illustrate the exploration strategies. It classifies the points in a two-dimensional continuous space \([0, 2\pi] \times [-1, 1]\) into three classes: red, black and blue as illustrated in Figure 1. In this example, data points \( x \) and \( y \) is a pair of Pareto Front between black and red classes, if \( x \) is red and \( y \) is black and they are very close to each other. Such pairs can show accurately the borders between the classes, thus help testers to determine whether the classification is correct or not.

![Figure 1: Data Space of the Running Example](image)

The Figure 2 gives the traversal and midpoint methods in the Morphy test specification. The midpoint method \( mid(x, y) \) calculates the geometric midpoint between \( x \) and \( y \).

It is easy to see that the running example forms an exploratory test system with the following distance function.

\[
Eucl((x_1, x_2), (y_1, y_2)) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}
\]

3 EXPLORATION STRATEGIES

This section presents the algorithms for three different exploratory strategies for testing clustering and classification applications. We also prove their correctness and illustrate their behaviour by using the running example given in the previous section.

3.1 Random Target Strategy

Let’s start with a simple exploration strategy based on random selection of known test cases in order to find the Pareto front of the classification groups between these two test cases. We call this strategy random target strategy.

The strategy starts by selecting a pair of two test cases \( x \) and \( y \) at random. If the outputs of the program \( P \) under test on these test cases are different, i.e. \( P(x) \neq P(y) \), then a point \( z_1 \) between \( x \) and \( y \) are generated by using the binary datamorphism of the midpoint method \( mid(x, y) \), i.e. \( z_1 = mid(x, y) \). The program \( P \) is executed on this mutant test case \( z_1 \) to classify it. The classification of \( z_1 \) must be different from one of the original pair of test cases; say \( P(z_1) \neq P(x) \). Thus, we can repeat the above steps with \( x \) and \( z_1 \) as the pair of test cases, and a further mutant \( z_2 \) can be generated. This process is iterated for a number of times to ensure the distance between the final pair of points is small enough. See Algorithm 1.

Let \( n > 0 \) be any given natural number. We write \( RT(n) = (a, b) \) to denote the results of executing Algorithm 1 with \( n \) as the parameter steps and \( (a, b) \) as the output.

Assume that the exploratory test system has the following properties.

(1) There is a constant \( c > 1 \) such that

\[
\forall x, y \in D. \left( \frac{\max(dist(x, z), dist(z, y))}{dist(x, y)} \right) \leq 1/c,
\]

where \( z = mid(x, y) \).

(2) There is a constant \( d_m > 0 \) such that

\[
\forall x, y \in D. (dist(x, y) \leq d_m).
\]

Then, we have the following theorem about the correctness of the random target strategy algorithm.
Algorithm 1 (Random Target Strategy)

Input:
- testSet: Test Pool;
- steps: Integer;
- mid(x, y): Binary datamorphism;

Output:
- a, b: Test Case;

Begin
1: Select two different test cases x and y in testSet at random;
2: Execute program P on test cases x and y;
3: Check if a pair of Pareto front exits between x to y:
   if (x.output ≠ y.output) then return (null, null)
   end if
4: Refinement:
   for i ← 1 to steps do
     z = mid(x, y);
     if (x.output ≠ z.output) then y = z
     end if
   end for;
   a = x; b = y;
   return ⟨a, b⟩;
End

Theorem 1. If RT(n) = ⟨a, b⟩ ≠ ⟨null, null⟩, then (a, b) is a pair of Pareto front according to P with respect to dist and δ, if \(d_m/c^{n} < \delta\).

Proof. If RT(n) = ⟨a, b⟩ ≠ ⟨null, null⟩, then, the condition of the IF-statement in step (3) is false. Thus, the loop is executed. It is easy to see that the For-loop in Step 4 in the algorithm terminates.

We now proof that the following is a loop invariant of the loop by induction on the number i of iterations of the loop body:

\[
\text{dist}(x, y) \leq \frac{d_m}{c^i} \land P(x) \neq P(y).
\]

When entering the loop, by assumption (4), the distance between the data points stored in variable x and y satisfies the following inequality.

\[
\text{dist}(x, y) \leq d_m.
\]

Since the condition of the IF-statement is false, we have that

\[
P(x) = x.\text{output} \neq y.\text{output} = P(y).
\]

Therefore, the loop invariant is true for i = 0.

Assume that the loop invariant is true for i = n ≥ 0.

After the execution of the loop body one more time (i.e. i = n + 1), by applying the Hoare logic of the IF-statements in the loop body, the distance \(d'_n\) between the data points stored in variables x and y will become either dist(x, z) or dist(z, y), where z = mid(x, y). By assumption (3), in both cases we have that

\[
d'_n \leq \text{Max} (\text{dist}(x, z), \text{dist}(z, y)) \leq \text{dist}(x, y)/c \leq d_m/c^{n+1}.
\]

By the condition of the IF-statement in the loop body and the property (1), applying Hoare logic we have that, after the execution of the loop body, the data points stored in variables x and y have the property that P(x) ≠ P(y). Therefore, the condition is a loop invariant according to Hoare logic.

When the loop exits, i = steps = n. By Hoare logic, after executing the assignment statements \(a = x\) and \(b = y\), we have that

\[
dist(a, b) \leq \frac{d_m}{c^n} \land P(a) \neq P(b).
\]

Therefore, the theorem is true by Definition 1. □

The algorithm of random target strategy can be run multiple times to generate a number of pairs for the Pareto front.

Example 1. For example, applying the random target strategy to the running example, we can obtain a test set shown in Figure 3 when 1000 pairs of test cases are selected at random from a test set of 300 random test cases. A total of 641 pairs of Pareto front test cases were generated. The success rate in generating a pair for the Pareto front is 64.1%. The set of Pareto front pairs shows clearly the boundary between the subdomains classified by the software.

![Figure 3: Pareto Front Generated by Random Target](image_url)

In this example, the number of steps n is 20. Since the data space \(D = [0, 2\pi] \times [-1, 1]\), if the distance function \(\text{dist}(x, y)\) is \(\text{Eucl}(x, y)\), we have that \(d_m = 2\sqrt{\pi^2 + 1}\). By the definition of \(\text{mid}(x, y)\), we have that

\[
\frac{\text{Max}(\text{dist}(x, z), \text{dist}(y, z))}{\text{dist}(x, y)} = 1/2.
\]

So, c = 2. By Theorem 1, for the distance \(\delta\) between each pair in the Pareto front, we have that

\[
\delta \leq \frac{d_m}{c^n} = \frac{\sqrt{\pi^2 + 1}}{2^{19}}.
\]

Note, the distance between the test cases in each pair of Pareto front is so small that they are not visually distinguishable in Figure 3. □

3.2 Directed Walk Strategy

A variation of the random target strategy is to start with one test case (rather than a pair) and apply a unary datamorphism repeatedly until a test case of different classification is found. Then, the Pareto front between these two test cases is searched for in the same way as for the random target strategy. In this strategy, the unary datamorphism (i.e. a mutation operator) is the traversal method. The repeated application of the mutation operator makes a ‘walk’ in one direction until a test case in a different class is found or gives up the exploration if we have gone too far (i.e. too many iterations).

Note that, a walk in one direction may not be able to find a data point in a different class. In that case, the algorithm returns (null, null). Let \(m, n > 0\) be any given natural numbers. We write \(DW(m, n) = (a, b)\) to denote the results of executing Algorithm 2 with m as the walking distance and n as the number of steps and \((a, b)\) as the output. Assume that the exploratory test system satisfies assumption (3) and has the following property.
Algorithm 2 (Directed Walk)

Input:
- TestSet: test set;
- walkDistance: integer;
- steps: integer;
- d(x): Unary datamorphism;
- mid(x, y): Binary datamorphism;

Output:
- a, b: Test Case;

Begin
1: Select a test cases x in testSet at random;
2: Execute program P on test case x;
3: Walk in one direction as follows:
   Bool found = false;
   for i ← 1 to walkingDistance do
      y = d(x);
      Execute software on test case y;
      if (x.output = y.output) then
         found = true; break;
      else x = y;
   end if
4: Check if a Pareto front can be found:
   if (~found) then return (null, null);
   end if
5: Refinement
   for i ← 1 to steps do
      z = mid(x, y);
      if (x.output = z.output) then y = z;
      else x = z;
   end if
   a = x; b = y;
   return (a, b);
End

There is a constant $d_s > 0$ such that
\[
\forall x \in D. (\text{dist}(x, d(x)) \leq d_s).
\]
(5)

where $d_s$ is called the step size of the traversal method $d(x)$. Then, we have the following correctness theorem for the directed walk algorithm.

Theorem 2. If $DW(m, n) = (a, b) \neq (null, null)$, then, $(a, b)$ is a pair in the Pareto front according to $P$ with respect to $\text{dist}$ and $\delta$, if $d_s/c^n < \delta$, where $n$ is the number of steps.

Proof. If $DW(m, n) = (a, b) \neq (null, null)$, then the condition of the If-statement in step (4) is false. Thus, the For-loop of Step (5) is executed. It is easy to see that the For-loop in Step 5 Refinement in the algorithm terminates.

Similar to the proof of Theorem 1, by the definition of $d_s$ and assumption (5), the following is a loop invariant of the loop by induction on the number $i$ of iterations of the loop body.
\[
\text{dist}(x, y) \leq d_s/c^i \land P(x) \neq P(y).
\]

When the loop exits, $i = steps = n$. By Hoare logic, after executing the assignment statements $a = x$ and $b = y$, we have that
\[
\text{dist}(a, b) \leq d_s/c^n \land P(a) \neq P(b).
\]

Therefore, the theorem is true by Definition 1. $\square$

Example 2. For example, starting from 1000 random test cases using the directed walk strategy with the upward(x) datamorphism as the unary traversal method, a set of 161 pairs of Pareto front were generated, shown in Figure 4. The set of Pareto front pairs also shows clearly parts of the boundaries between classes. The success rate of finding a pair of Pareto front on one test case is 16.1%.

In this example, the number $n$ of steps is also 20. By the definition of upward(x) traversal method, we have that $d_s = 0.2$, if the distance function $\text{dist}(x, y)$ is Eucl(x, y). As in Example 1, by the definition of mid(x, y), we have that $c = 2$. By Theorem 2, for the distance $\delta$ between each pair of Pareto front, we have that
\[
\delta \leq d_s/c^n = 0.2 \times \frac{1}{2^{20}}.
\]

Again, the distance between the test cases in each pair of Pareto front is so small that they are not visually distinguishable, so they appear as one dot in Figure 4. $\square$

![Figure 4: Pareto Fronts Generated by Directed Walk](image)

3.3 Random Walk Strategy

If multiple traversal methods are available, a random walk can be performed by selecting the direction of the next step at random. This is similar to the random walk testing in hyperlink/web GUI test. The algorithm is given below.

We write $RW(m, n) = (a, b)$ to denote the results of executing Algorithm 3 with $m$ as the walking distance and $n$ as the steps and $(a, b)$ as the output. Assume that the exploratory test system satisfies assumption (3) and has the following property. There is a constant $d_{sm} > 0$ such that
\[
\forall x \in D, \forall d_i \in W. (\text{dist}(x, d_i(x)) \leq d_{sm}).
\]
(6)

where $d_{sm}$ is called the maximal step size of the traversal methods $d_i(x) \in W$. Then, we have the following correctness theorem for the algorithm of random walk strategy.

Theorem 3. If $RW(m, n) = (a, b) \neq (null, null)$, then, $(a, b)$ is a pair in the Pareto front according to $P$ with respect to $\text{dist}$ and $\delta$, if $d_{sm}/c^n < \delta$, where $n$ is the steps.

Proof. If $RW(m, n) = (a, b) \neq (null, null)$, then, the condition of the If-statement in step (4) is false. Thus, the For-loop of Step (5) is executed. It is easy to see that the For-loop in Step 5 Refinement in the algorithm terminates.
Algorithm 3 (Random Walk Strategy)

Input:
- testSet: Test Set;
- walkingDistance: Integer;
- steps: Integer;
- \(d_1(x), \ldots, d_k(x)\): Unary datamorphism; \(k > 1\)
- \(m(x, y)\): Binary datamorphism;

Output:
- \(a, b\): Test Case;

Begin
1: Select a test case \(x\) in testSet at random;
2: Execute program \(P\) on test case \(x\);
3: Walking at random to search for test case in a different class:
   \(\text{Bool } \text{found} = \text{false};\)
   \(\text{for } i \leftarrow 1 \text{ to } \text{walkingDistance} \text{ do} \)
   \(\quad \text{Get a random integer } r \text{ in the range } [1, k];\)
   \(\quad y = d_i(x);\)
   \(\quad \text{Execute program } P \text{ on test case } y;\)
   \(\quad \text{if } (x.\text{output} \neq y.\text{output}) \text{ then} \)
   \(\quad \quad \text{found} = \text{true}; \text{break; }\)
   \(\quad \text{else } x = y;\)
   \(\text{end if}\)
\(\text{end for}\)
4: Check if a Pareto front can be found:
\(\text{if } (\neg \text{found}) \text{ then return } (\text{null, null});\)
\(\text{end if}\)
5: Refinement:
\(\text{for } i \leftarrow 1 \text{ to } \text{steps} \text{ do} \)
\(\quad z = m(x, y);\)
\(\quad \text{if } (x.\text{output} \neq z.\text{output}) \text{ then} y = z;\)
\(\quad \text{else } x = z;\)
\(\text{end if}\)
\(\text{end for}\)
\(a = x; b = y;\)
\(\text{return } (a, b);\)

End

Similar to the proof of Theorem 1, by the definiton of \(d_{\text{sm}}\) and assumption (6), we can prove that the following is a loop invariant of the loop by induction on the number \(i\) of iterations of the loop body.

\[
dist(x, y) \leq \frac{d_{\text{sm}}}{c^i} \land P(x) \neq P(y).
\]

When the loop exits, \(i = \text{steps} = n\). After executing the assignment statements \(a = x\) and \(b = y\), the following is true by Hoare logic.

\[
dist(a, b) \leq \frac{d_{\text{sm}}}{c^n} \land P(a) \neq P(b).
\]

Therefore, the theorem is true by Definition 1.

Example 3. For example, by applying the random walk strategy on a test set containing 300 random test cases, 1000 random walks generated 805 pairs of Pareto front test cases shown in Figure 5, where the walking distance was 20 steps.

In this example, the number \(n\) of steps is also 20. By the definition of \(\text{upward}(x), \text{downward}(x), \text{leftward}(x)\) and \(\text{rightward}(x)\) traversal methods, we have that \(d_s = 0.2\), if the distance function \(\text{dist}(x, y)\) is \(\text{Eucl}(x, y)\). As in Example 1 and 2, by the definition of \(m(x, y)\), we have that \(c = 2\). By Theorem 3, the distance \(\delta\) between each pair of Pareto front satisfies the following inequality.

\[
\delta \leq \frac{d_s}{c^{20}} = 0.2 \times \frac{1}{2^{20}}. \quad \square
\]

4 EXPERIMENTS

Controlled experiments with the exploratory test strategies have been conducted using the automated datamorphic testing tool Morphy to study their test effectiveness. This section report the results of the experiments.

4.1 Design of the Experiments

4.1.1 Objectives of the Experiments. As discussed in the previous sections, exploration strategies are designed to test classification applications. They aim to find the borders between subdomains of the classifications. The goal of the experiments is to study the factors that have effect on the effectiveness of these test strategies in terms to their capability of finding the Pareto fronts between subdomains. The measurement of test effectiveness is the number of test executions per border points found by the test strategy.

It is worth noting that the experiments are not for comparison of the strategies, which each has its own suitable applications.

4.1.2 Subject applications. The experiments are carried out with ten classification applications shown in Figure 6. These applications are on the same input domain, i.e. two-dimensional real numbers in the range of \([0, 2\pi] \times [-1, 1]\).

4.2 Experiment process and the results

For each subject application, three exploration strategies are used with various parameters. Each test is repeated for 10 times using the testing tool Morphy and the average of the data is used to analyse the results.

4.2.1 Experiments with the directed walk strategy. The experiments used various numbers of random test cases from 200 to 1200 as shown in Table 1; here, the column \#Seed TCs is the number of seed test cases in the experiment. These seed test cases are generated at random from the uniform distribution. From each seed test case, one walk in one direction is made for up to 20 steps. The experiments used the upward datamorphism. The column Avg \#Runs in Table 1 gives the average number of test executions of the subject program under test. The column Avg \#mutant TC gives the average number of mutant test cases generated; these are test cases on the borders of the clusters.
The experimental data shows that the number of mutant test cases (i.e., the pairs of test cases in the Pareto front) generated by using the directed walk strategy increases linearly with the number of walks; see Figure 7. Similarly, the number of test executions is also linear with respect to the number of walks. In Figure 7, the X axis is the number of random seed test cases, which equals the number of walks, and the Y axis of (a) and (b) are the average numbers of mutant test cases and test executions, respectively. The average numbers of test executions on various subject programs are so close to each other that they are not visually separable in Figure 7(a).

The test effectiveness is measured in terms of the number of test executions per mutant test case generated. It is fairly invariant for each subject while the number of random seed test cases varies from 200 to 1200; see Figure 7(e), where the Y axis is the average test effectiveness. The experiment data also show that the test effectiveness varies significantly for different subject programs; see Figure 7(d), which gives the overall average effectiveness of testing various subject programs.

4.2.2 Experiments with the random walk strategy. There are two parameters in the random walk strategy: (1) the number of seed test cases, and (2) the number of walks starting from the seed test cases. Two sets of experiments were designed and conducted. The first is with a fixed number of seed test cases (200 test cases) but variable numbers of random walks (range from 200 to 1200). The second is with a fixed number of random walks (800 walks) but variable numbers of random seeds (range from 200 to 1200).

Table 2 gives the result data of the first set of experiments.

The results of the experiments show that the average number of test executions and the average number of mutant test cases generated is linear in the number of random walks; see Figure 8. The test effectiveness increases with the number of walks; see Figure 8. Although the overall average test effectiveness varies...
Figure 8: Results of The Random Walk Strategy

between subject programs, the differences on test effectiveness are much smaller than the directed walk strategy. As shown in Figure 8(c) and (d), the test effectiveness for testing subject programs Box 1, Triangle 1 and Circle 1 are poorer than those for the other subjects. The second set of experiments were with fixed number of walks but variable numbers of seed test cases. Table 3 shows the results of the experiment in which 800 walks were run with variable number of seeds.

Table 3: Experiments Data of Variable Number of Test Cases

<table>
<thead>
<tr>
<th>Subject</th>
<th>#Seeds</th>
<th>Avg #Runs</th>
<th>Avg #Execs</th>
<th>Avg #Mutants</th>
<th>Avg #Execs/Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Box 1</td>
<td>100</td>
<td>387.40</td>
<td>400.80</td>
<td>200.00</td>
<td>11170.00</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>389.60</td>
<td>392.40</td>
<td>200.00</td>
<td>400.00</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>395.80</td>
<td>399.20</td>
<td>200.00</td>
<td>648.60</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>397.20</td>
<td>401.00</td>
<td>200.00</td>
<td>883.60</td>
</tr>
<tr>
<td>Circle 1</td>
<td>100</td>
<td>1096.00</td>
<td>1200.00</td>
<td>500.00</td>
<td>11904.00</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>1103.10</td>
<td>1204.90</td>
<td>500.00</td>
<td>1910.00</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>1106.10</td>
<td>1209.00</td>
<td>500.00</td>
<td>2900.00</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>1108.20</td>
<td>1212.00</td>
<td>500.00</td>
<td>3900.00</td>
</tr>
<tr>
<td>Triangle 1</td>
<td>100</td>
<td>1093.80</td>
<td>1200.00</td>
<td>500.00</td>
<td>11380.00</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>1101.60</td>
<td>1204.90</td>
<td>500.00</td>
<td>1910.00</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>1104.60</td>
<td>1209.00</td>
<td>500.00</td>
<td>2900.00</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>1106.60</td>
<td>1212.00</td>
<td>500.00</td>
<td>3900.00</td>
</tr>
</tbody>
</table>

The data show that the average number of test executions and the average number of mutant test cases generated are linear in the number of walks for all subject programs as shown in Figure 10.

The test effectiveness increases with the number of walks since the average number of test executions needed to generate a mutant test cases decreases with the number of walks increases. The test effectiveness of the random target strategy is given in Figure 10(b). The data show that the test effectiveness for subjects Triangle 1, Box 1 and Circle 1 are significantly poorer than those for the other subjects. This is also shown clearly in Figure 10(d).

4.3 Discussion

From the experiments, we observed the following phenomena.

4.2.3 Experiments with the random target strategy. The random target strategy only has one parameter: the number of pairs of test cases selected at random. The experiments are conducted with this parameter ranging from 200 to 1200. The experiment data are given in Table 4 below.
Theoretically speaking, the test effectiveness for the directed walk strategy is determined by the probability that there is a border between two subdomains in the right direction from a test case and within the walking distance. For the random target strategy, the test effectiveness is determined by the probability that two random test cases fall in two different subdomains. For the random walk strategy, the test effectiveness is determined by the probability that there is a border nearby to a randomly selected test case. These properties have a number of implications.

First, given a classification application, one should select the most effective strategy to explore the Pareto fronts between subdomains based on the understanding of the application. The data obtained from our experiments are not sufficient to compare the strategies on their effectiveness. This is because the probability of finding a pair in the Pareto front heavily depends on the size and location of the subdomains of the classification application. There is no benchmark on such parameters in real applications as far as we know. Our subjects in the experiments may not be representative of the distribution of the parameters in real applications.

Second, it provides a good explanation of the observations made in the previous sections that the number of pairs generated for the Pareto front is a linear function of the number of seed test cases or number of walks since they are independent. Moreover, although the test effectiveness is mostly determined by the size, shape and location of the subdomains that the program classifies, for directly walk and random walk strategies, it is also affected by the number of steps walked and the number of iterations in the refinement. The number of steps walked influences the probability of finding two points in different subdomains and also the total number of test executions. The longer the walk, the more likely one is to find two points in different subdomains, but this requires more test executions. Thus, a balance between these two contradictory factors of test effectiveness must be made to achieve the best test effectiveness.

Finally, the number of iterations in the refinement loop controls the distance between the pair of test cases in the Pareto fronts generated. It has no impact on the probability of finding two data points in different subdomains, but does have an affect on test effectiveness. The shorter distance mutations requires more iterations, thus more test executions, and therefore, is less effective. For random walk and directed walk strategies, the number of iterations can be selected for correctness theorems proved in this paper. For the random target strategy, usually more iterations are required than the other two strategies.

4.3.2 Validity of the Experiments. As pointed out at the beginning of the section, the experiments are designed to determine which factors have an effect on the test effectiveness of the strategies. The subject programs used in the controlled experiments have subdomains that are of typical shapes in data mining and machine learning applications [1, 5, 8]. As discussed above, the conclusion that we draw from the experiments are not depending on specific features of subdomains such as the size and location. However, as discussed above, they do provide insight on the factors that affect test effectiveness. Therefore, we are confident that the conclusions drawn from the experiments are valid.
5 CONCLUSION

5.1 Related Work
Exploratory testing was originally proposed for improving GUI-based manual testing of web-based applications, which also often lacks a clear definition of software correctness [12]. The name is given to a common practice in industry that existed for many years without guidance until recently. The notion of exploratory strategy was first defined by Whittaker [12] as guidance on how to manually explore the software in the most effective way.

Research on testing AI applications has been active in recent years [2, 4, 6]. It is interesting to observe that datamorphisms are actually used to testing AI applications like driverless vehicles [11, 15]. Our case study with face recognition shows that test automation can be improved by reusing datamorphisms if they are explicitly defined and supported by a testing tool [17, 18, 21]. However, the testing of classification applications has not been studied intensively. An interesting work by Xie et al. [13] is the development of a set of metamorphic relations as test oracles for the clustering and classification applications [19]. In [14], a case study is reported to use these metamorphic relations to test a clustering function generated by the data mining tool Weka, in which datamorphisms are also used although they are not explicitly defined.

5.2 Main Contributions
The main contribution of this paper is the adaptation of the notion of exploratory strategy to the testing of AI applications. We demonstrated that such strategies can be formally defined in the datamorphic testing framework. They have also been implemented in the automated datamorphic testing tool Morphy [17].

In this paper, we studied the theoretical properties of three exploratory strategies for the discovery of the Pareto front of classification applications. We formally proved the correctness of the algorithms that implement the strategies.

We have also conducted controlled experiments with the exploration strategies. Experimental data demonstrated the factors that have impact on test effectiveness of these strategies. The observations obtained from experiments provide a guidance to the selection of the strategies for a given classification application and the choices of parameters to apply the strategies.

5.3 Future Work
The data spaces of the running example and the subjects of the experiments have fixed dimensions on continuous values. This is for the purpose of easily visualising the results. The strategies are independent to the continuity and dimensions of the data space, thus they are also applicable to other types of classification applications. The proofs of their correctness are also independent to these features, thus the correctness theorems also hold for such data spaces. It is interesting to conduct experiments using different types of data spaces, such as image, audio, video and text values. We are conducting case studies with real machine learning applications to evaluate the practical usability of the strategies.

There are also many possible variations of the strategies proposed and studied in this paper. In particular, the algorithms in this paper do not need a test morphisms that measure the distances between two test cases. If such a test morphism is available, the termination of the refinement loop can be determined by the distances between the pair of test cases.

The analysis of the phenomena observed in the experiments suggested that the test effectiveness depended on the probability of finding two test cases that are in different classification subdomains. A formal proof of this property will give a solid foundation for understanding these strategies and providing precise guidance to the selection of the parameters of the strategies. Thus, it is worth further research.

REFERENCES