A New Checksum Formula for Error Detecting Decimal Codes

Mert Bora ALPER
bora@boramalper.org

13 February 2017

Abstract

Decimal codes are used everywhere in modern societies to identify things such as credit cards, books, and even human beings through passport numbers. All these integers are subject to various transmission errors, due to machine, and mostly human errors. Hence various error detection algorithms have been developed to mitigate the issue, even if not completely solve it through error correction.

This essay addresses the problem of detecting transmission errors in decimal codes using a single decimal checksum digit. The aim is to develop a new algorithm that can detect most of the transmission errors while being flexible at the same time so that it can be adapted to different situations. The scope of the transmission errors are limited to the 6 most common transmission errors, which constitutes 91.4% of all transmission errors, according to the Kirtland's investigation. For the scope of this paper, the new error detection algorithm is expected to deal with situations where only one transmission error occurs at a time.

In the development of The New Algorithm, a mathematical model of the problem domain is created using arrays, and afterwards, the *conditions* which the modelling arrays must satisfy for detecting errors are identified under the concept of The Ideal Algorithm. Afterwards, each of the modelling *conditions* is assigned a score depending on how "detrimental" their violation will be (which is based on the frequency of the error the *condition* is interested in), which in turn allowed computer assisted brute-force method to be used to find the most suitable arrays for The New Algorithm.

The New Algorithm can detect 91.05% of all transmission errors and yet, thanks to the mathematical modelling, it can be adapted to different situations in future and different local contexts.

Word Count
3709

Civanıma, sevgilerimle.[1]

[1] To my youngster, with love.

Contents

1 Introduction

Since the very early days of civilization, humans have used numbers to count the objects around themselves but only recently the necessity to identify each of those objects lead to the usage of numbers as identifiers. Today, identification numbers are ubiquitous. They are used to identify geographical areas, banknotes, books, cars, weapons, bank accounts, and parts of complex machinery such as airplanes. Even humans are identified through the usage of passport numbers and/or social security codes. These numbers might be misread, misheard, misspoken, or miswritten; in other words, mistransmitted. Errors and their causes are numerous (pardon the pun) and the need for a (better) error detection mechanism for numbers is increasing each and every day. This essay is devoted to the development of a new algorithm that can accommodate today's needs and yet has the flexibility to endure time.

2 Common Transmission Errors

A study of common transmission errors is essential for any error detection algorithm that aims to have a practical use, therefore it is crucial for the success of The New Algorithm that the common transmission errors that are explained here are taken into consideration.

Table 1: Most common transmission errors[2]

Error Type	Example	Percentage in All Transmission Errors
Single-Digit	7 instead of 4	79.1%
Transposition	23 instead of 32	10.2%
Jump-Transposition	063 instead of 360	0.8%
Twin	77 instead of 44	0.5%
Phonetic	14 instead of 40	0.5%
Jump-Twin	343 instead of 242	0.3%
		Total: 91.4%

Single-Digit Errors Single-digit errors occur when a single digit in a code is mistransmitted. They are often caused by transcription errors, such as pressing the adjacent key on a PIN pad. Since single-digit errors constitute 79.1% of the all transmission errors, it is a must for any error detection algorithm to catch all of the single-digit errors that might occur.

Transposition Errors Transposition errors occur when two different digits swap positions (transpose). They are often caused by transcription errors while typing with two or more fingers. When one of the digits at stake is equal to 1, it is called *phonetic error* since the pronunciation of numbers $1D$ and $D1$ (for $D \geqslant 3$) is very similar to each other in English (and also in German). As transposition errors constitute significant amount (10.2%) of the all transmission errors, and hence it is crucial for any error detection algorithm to catch the majority, if not

all, of the transposition errors that might occur.

Jump-Transposition Errors Jump-transposition errors occur when two different digits separated by an adjacent digit swap positions. This is often caused by a transcription error while typing with three or more fingers.

Twin Errors Twin errors occur when a twice-repeating digit is mistransmitted. This is often caused by a transcription error while typing from muscle memory, which is usually the case for many human beings.

Phonetic Errors Phonetic errors occur when two adjacent digits in the form of $1D$ or $D1$ is mistransmitted as $D1$ or $1D$ (for $D \geqslant 3$), respectively. Phonetic errors are a special case of *transposition errors* since they are caused by the phonetic similarity in the pronunciation of those numbers in English (and in German) language.

Jump-Twin Errors Jump-twin errors occur when a twice-repeating digit that is separated by another digit in-between is mistransmitted (the digit in-between stays unaffected). This is often caused by a transcription error while typing from muscle memory and using three or more fingers at the same time.

3 Investigating The Ideal Algorithm

Although there might be no ideal error detection algorithm that detects all of the most common transmission errors, the conception of such an algorithm can guide the development of The New Algorithm. The sole aim of this investigation is to present the development of The New Algorithm to the reader, starting from the mathematical modelling of the problem domain.

The idea of an ideal algorithm is far too abstract to be investigated at this point, hence let us assume that we are using a checksum formula, that is, calculating a single check-digit by aggregating (which might be an arithmetic summation) all the digits after applying certain, well-defined operations on them. Afterwards, modular arithmetic will be used to reduce the aggregate to a single checksum digit.

For the scope of this investigation, The Ideal Algorithm is expected to detect all of the most common transmission errors, only if *only one* error occurs at the same time. Also, for the sake of simplicity and again the scope of this investigation, it is assumed that the check-digit is not subject to *any* of the errors, including but not limited to most common transmission errors which are described in the section 2 (*i.e.* the checksum never gets corrupt).

Throughout the investigation, let c be the check-digit and d_x be the x^{th} digit (or *symbol*, both terms are used synonymously) of the decimal code C with l digits (for $x \in [1, l]$).

All single-digit errors can be detected using the simplest modular arithmetic summation:

$$\begin{aligned} c &\equiv d_1 + d_2 + d_3 + \cdots + d_l \pmod{10} \\ &\equiv \left(\sum_{x=1}^{l} d_x \right) \bmod 10 \end{aligned}$$

In the case of a single-digit error, the difference between the correct and the erroneous decimal digit is necessarily within the range $[1, 9]$. Since the modulus of the congruence is greater than the greatest value within the difference range (*i.e.* since $10 > 9$), any *single* single-digit error in any place is proved to change the result of the congruence to a different value. This guarantee holds true as long as the modulus is greater than the greatest possible value a place can hold, hence the modulus has to be equal to 10 in order to be able to detect all single-digit errors in *decimal* codes.

Since addition operation in modular arithmetic is commutative, to be able to detect transposition and jump-transposition errors, the value of a digit must vary depending on its position during aggregation. The idea, in its most abstract form, is to substitute each digit with a value from the substitution array, that is associated with the digit's position, where the index is the digit itself. Different substitution arrays must be defined for different positions, so that the value of the digit may vary depending on its position in the decimal code.

Creating a different substitution array for each and every position in decimal codes of a given length would be a tedious task, and it is also very much unnecessary. Instead, looking at the most common errors, we can realize that jump-transposition and jump-twin errors involve three adjacent positions, while other errors involve either two or one positions. Therefore, a triple of substitution arrays should suffice in detecting all of the most common errors. This also allows us to use same substitution arrays for all decimal codes, regardless of their lengths.

An ideal triple of substitution arrays (A_{3n}, A_{3n+1} and A_{3n+2} for $n \in [0, 1, 2, \ldots, \lfloor l/3 \rfloor]$) must satisfy the following conditions for *all* possible values of transposing digits d_x and d_{x+1} (where $d_x \neq d_{x+1}$) if it detects *all* transposition errors:

$$A_{3n}[d_x] + A_{3n+1}[d_{x+1}] \not\equiv A_{3n}[d_{x+1}] + A_{3n+1}[d_x] \pmod{10} \tag{1}$$

$$A_{3n+1}[d_x] + A_{3n+2}[d_{x+1}] \not\equiv A_{3n+1}[d_{x+1}] + A_{3n+2}[d_x] \pmod{10} \tag{2}$$

$$A_{3n+2}[d_x] + A_{3n}[d_{x+1}] \not\equiv A_{3n+2}[d_{x+1}] + A_{3n}[d_x] \pmod{10} \tag{3}$$

Since the set of conditions for detecting *all* jump-transposition errors are same as the conditions for detecting *all* transposition errors, as shown in the figures below, nothing else needs to be added to detect *all* jump-transposition errors.

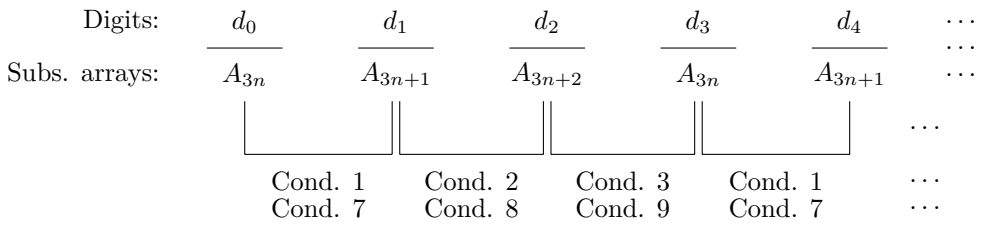

Figure 1: Conditions for Detecting Transposition (Conditions 1, 2, 3) and Twin (Conditions 7, 8, 9) Errors

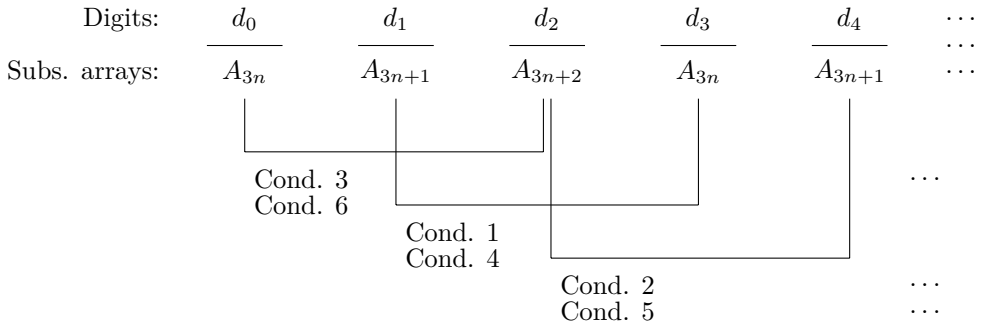

Figure 2: Conditions for Detecting Jump-Transposition (Conditions 1, 2, 3) and Jump-Twin (Conditions 4, 5, 6) Errors

Here are the conditions for detecting *all* twin errors for every possible x, e pair where x is the correct and e is the erroneous digit that is not equal to x:

$$A_{3n}[x] + A_{3n+1}[x] \not\equiv A_{3n}[e] + A_{3n+1}[e] \pmod{10} \tag{4}$$

$$A_{3n+1}[x] + A_{3n+2}[x] \not\equiv A_{3n+1}[e] + A_{3n+2}[e] \pmod{10} \tag{5}$$

$$A_{3n+2}[x] + A_{3n}[x] \not\equiv A_{3n+2}[e] + A_{3n}[e] \pmod{10} \tag{6}$$

Since the set of conditions for detecting *all* jump-twin errors is same as the conditions for detecting *all* twin errors, as shown in the figures nos. 1 and 2 above, nothing else needs to be added to detect *all* jump-twin errors.

The conditions for detecting *all* phonetic errors for every digit $x \geqslant 3$ (because 11 repeats the same digit twice, and twelve [*de.* zwölf] or twenty [*de.* zwanzig] cannot be phonetically confused such as the thirteen-thirty [*de.* dreizehn-dreizig] and so on):

$$A_{3n}[1] + A_{3n+1}[x] \not\equiv A_{3n}[x] + A_{3n+1}[0] \pmod{10} \tag{7}$$

$$A_{3n+1}[1] + A_{3n+2}[x] \not\equiv A_{3n+1}[x] + A_{3n+2}[0] \pmod{10} \tag{8}$$

$$A_{3n+2}[1] + A_{3n}[x] \not\equiv A_{3n+2}[x] + A_{3n}[0] \pmod{10} \tag{9}$$

The 9 conditions above, grouped in sets by kinds of error, can be used altogether as a set of

conditions to detect *all* of the *all* kinds of most common transmission errors described in the section 2.

4 The New Algorithm

In an ideal world, all transmission errors can be detected using ideal substitution arrays, though in reality, we may need to compromise as there might be no triple of substitution arrays that satisfy all of the 9 conditions explained in the previous section. The solution is, then, to prioritize each subset of conditions by the frequency percentage of the error it detects, and assign a penalty score that is directly proportional to its priority, in case if an instance for which the condition does not hold true is found. For example, 102 ($10.2\% * 100$) points must be "awarded" to the triple of substitution array in test for each d_x, d_{x+1} pair that violates a condition for detecting single-digit errors, multiplied by the amount of violated conditions. After evaluating all possible triples of substitution arrays, the array(s) with the least penalty score would be the best triple(s) of substitution arrays (*i.e.* the one(s) that can detect more errors than any other(s)).

Although using brute-force to scan the search space might be the first thing that comes to mind, and indeed it is the only possible way to attack the problem, it will be seen that considering the vastness of the search space, any brute-force method is doomed to fail using today's technology given the time constraints; there are 10! different substitution arrays, and ${}^{10!}C_3 \approx 8 \times 10^{18}$ triples of substitution arrays which a brute-force approach has to consider. Assuming that we could evaluate a billion triples per second, it would still take us 252.5 years to finish scanning the problem space! Nonetheless, source code of a Python program for purely brute-force approach has been included in the appendix A.1 as a starting point for the sake of completeness; whilst the need for a more clever approach that can find a sub-optimal solution in a reasonable amount of time is clearly evident, hence the next section is dedicated to the development of such approach.

4.1 Heuristic Methods in Seeking The Sub-Optimal Triple

By starting with a set of assumptions and propagating constraints, we can prune the search space significantly while seeking a sub-optimal solution. To begin with, instead of iterating through the search space in lexicographical order, we can prioritize triples that are expected to have a lower penalty score than others. Starting from the first triple in lexicographical order and continuing is indeed extremely inefficient; for every two different positions where the values at the same positions in any two substitution arrays are equal to each other, one of the conditions for detecting transposition errors will be violated (since the sum of the different values at different positions in different arrays would be equal to each other).

Remark. *Let x and y be two different positions (i.e. indices) where the values in substitution arrays A_0 and A_1 overlap, then the condition for detecting transposition errors that is explained in section 3 is violated.*

$$A_0[x] = A_1[x]$$
$$A_0[y] = A_1[y]$$

Proof by contradiction.

$$A_0[x] + A_1[y] \not\equiv A_0[y] + A_1[x] \pmod{10}$$

substitute $A_0[y]$ with $A_1[y]$, and $A_1[x]$ with $A_0[x]$

$$\implies A_0[x] + A_1[y] \not\equiv A_1[x] + A_0[y] \pmod{10}$$

Therefore it is proved that for every two substitution arrays and two different positions (*i.e.* indices) where the values in the arrays overlap, a condition for detecting transposition errors will be violated (this is also called a *defect*). □

For instance, let us look at the first triple in lexicographical order:

$$A_{3n} = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$$
$$A_{3n+1} = [0, 1, 2, 3, 4, 5, 6, 7, 9, 8]$$
$$A_{3n+2} = [0, 1, 2, 3, 4, 5, 6, 8, 7, 9]$$

As the first 7 values of the first two arrays overlap, it is hence known that there are at least ${}^7\mathrm{C}_2 = 21$ transposition errors and the total (penalty) score of this triple is at least $21 \times 102 = 2142$ even without considering the other two pairs of arrays.

While this new array generation method can speed up the brute-forcing process significantly, it is not of much use without knowing when to stop; next thing to do is to find how can we understand whether we are satisfied with a result or not. Looking at most common transmission errors in Table 1, it can be seen that transposition and jump-transposition errors are more frequent than less frequent transmission errors by an order of magnitude; the greatness of this difference in frequencies makes less frequent transmission errors more tolerable compared to transposition errors. This means, we can skip checking triples with $n + 1$ transposition error detection defects, without even checking for other conditions, if we are aware of another triple with n transposition error detection

defects and with a total sum of penalty scores for other kinds of errors being less than or equal to 1 transposition error penalty score (102 points).

To sum up and incorporate what has been said so far, let us describe how the search space will be iterated in such a way that ineffective triples can be pruned at the very beginning:

1. For every possible A_{3n} array which can be any permutation of 10 digits,
2. Every possible A_{3n+1} array will be generated in a such a way that during the generation process, it is ensured that the resultant array doesn't have any values that overlap with the reference array A_{3n}.
3. If there are any unordered (d_x, d_{x+1}) pairs that violates the condition for detecting transposition errors more than once in total (A_{3n} being the first array in the condition), create a new A_{3n+1}.
    - It is assumed that it is possible to find a pair of substitution arrays with 1 transposition error detection defect and with the total sum of penalty scores for the other kinds of errors being less than or equal to 102, so that other pairs with more than 1 transposition error detection defects can be skipped.
4. Repeat the step 2 and 3 for generating A_{3n+2} array, testing A_{3n+2} together with A_{3n}, and A_{3n+1} afterwards. As in the previous steps, A_{3n+2} cannot have more than two transposition error detection defects *in total* compared to the previous arrays.
5. Now a triple of substitution arrays is generated, with the *total* number of 3 transposition error detection defects.
6. If there is at least one triple with *total* sum of penalty scores for minor (jump-transposition, twin, phonetic, and jump-twin) error detection defects is less than or equal to 102, it is *guaranteed* that no triple with more than 3 transposition error detection defects can have a better score than the aforementioned triple.
    - This is the assumption that we have made at step 3. Given enough time, the process can prove its assumption through empirical evidence if the assumption holds true.
7. Keep searching to discover whether there is another triple with a better score, for an *indefinite* period of time.

There are several assumptions taken into consideration in the process described above:

- Since substitution arrays are generated sequentially one after another instead of in triples altogether, it might be that there are triples with less than $1 + 1 + 1 = 3$ transposition error

detection defects, although I do not have any proof or counter-proof to claim so or otherwise. During my prior investigation, despite all the different brute-force approaches I have tried, I did not encounter any triples of substitution arrays which have less than 3 transposition error detection defects, despite all the different brute-force methods I have tried.

- In the heuristic process, no values are allowed to overlap with another value at the same position in another array, although as described above, one transposition error detection defect requires two overlaps, and thus pairs of substitution arrays with 1 overlap, and 1 or 0 transmission error detection defect is entirely possible. Although, for the sake of brevity and more importantly for the speed of the computer program, the above-described process adheres to *no overlaps* rule very strictly and allows no overlaps at all.
- The search was terminated after 12 hours due to limited time and resources, assuming that the time given would be enough to find a good enough sub-optimal solution.

If one modifies the aforementioned process to eliminate the assumptions explained above, the process is guaranteed to find the best solution given enough time provided that 6^{th} step of the process holds true (again, which can be evidenced by the process itself if such a case exist; and it does exist).

4.2 The New Algorithm

Let A_{3n}, A_{3n+1}, and A_{3n+2} be three substitution arrays:

$$\begin{aligned} A_{3n} &= [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] \\ A_{3n+1} &= [1, 7, 9, 6, 3, 0, 8, 5, 2, 4] \\ A_{3n+2} &= [9, 6, 0, 2, 7, 5, 3, 8, 4, 1] \end{aligned}$$

Using the sub-optimal arrays above that are found after computer assisted brute-force following the aforementioned methodology, The New Algorithm is as follows:

$$\begin{aligned} c &= \sum_{x=1}^{l} \Big(A_{3n+((x-1) \bmod 3)}[d_x] \Big) \bmod 10 \\ &= \Big(A_{3n}[d_1] + A_{3n+1}[d_2] + A_{3n+2}[d_3] + A_{3n}[d_4] + \ldots + A_{3n+(l \bmod 3)}[d_l] \Big) \bmod 10 \end{aligned}$$

4.3 Analysis of The New Algorithm

Analysing the triple of substitution arrays used in The New Algorithm for the error detection defects, the following table is created:

Table 2: Error Detection Defects in the Triple of Substitution Arrays used in The New Algorithm

Error Type	Amount of Defects	Defects (x, y)		
		$A_{3n} \leftrightarrow A_{3n+1}$	$A_{3n} \leftrightarrow A_{3n+2}$	$A_{3n+1} \leftrightarrow A_{3n+2}$
Singe-Digit	0	$\varnothing$	$\varnothing$	$\varnothing$
Transposition	3	$(5, 9)$	$(0, 3)$	$(5, 6)$
Jump-Transposition	3	$(5, 9)$	$(0, 3)$	$(5, 6)$
Twin	8	$(0, 2)$	$(0, 6)$, $(2, 8)$	$(0, 4)$, $(1, 7)$
			$(3, 7)$, $(5, 9)$	$(5, 9)$
Phonetic (x)	2	$\varnothing$	(8)	(8)
Jump-Twin	8	$(0, 2)$	$(0, 6)$, $(2, 8)$	$(0, 4)$, $(1, 7)$
			$(3, 7)$, $(5, 9)$	$(5, 9)$

Since conditions for detecting transposition, jump-transposition, twin, and jump-twin errors are defined for a pair of violating digits, there are 45 different unordered (x, y) pairs where $x \neq y$ which might violate one of the conditions for detecting errors; instances of such pairs for each error type that they violate are given in the table above, grouped by the pairs of substitution arrays where the violation occurs. Conditions for detecting phonetic errors, on the other hand, are defined for a single digit $x \in \{3, 4, 5, 6, 7, 8, 9\}$; hence there are only 7 different x values which might violate one of the conditions for detecting phonetic errors.

As an example, let us calculate the percentage of undetected twin errors in all twin errors: for instance, if the double zeroes (or twos) are mistransmitted as double twos (or zeroes) -represented by the pair $(0, 2)$- *and* if this occurs at the positions 3_n and 3_{n+1}, then a twin error would slip undetected, as can be seen in the table. Translating into formal language, this is equal to the probability of getting the pair $(0, 2)$ among 45 unordered (x, y) pairs $(1/\binom{45}{1} = 1/45)$ *and* the probability of this happening at the pair of substitution arrays A_{3n} and A_{3n+1} $(1/\binom{3}{2} = 1/3)$, thus altogether $1/45 * 1/3 = 1/135$. By repeating the process for the remaining 7, the percentage of all undetected twin errors in all twin errors can be calculated as follows:

$$
\begin{aligned}
&= \frac{1}{45} * \frac{1}{3} + \frac{4}{45} * \frac{1}{3} + \frac{3}{45} * \frac{1}{3} \\
&= \frac{8}{135} \\
&= 5.93\%
\end{aligned}
$$

The percentages for the rest of the error types are calculated using the same method.

Table 3: Percentages of Undetected Errors

Error Type	Num. of Defects	Percentage of (1) Errors in All (2) Errors		
		(1) Undetected	(1) Detected	
		(2) *Error Type*	(2) *Error Type*	(2) Transmission
Single-Digit	0	0%	100%	79.10%
Transposition	3	2.22%	97.78%	9.97%
Jump-Transposition	3	2.22%	97.78%	0.78%
Twin	8	5.93%	94.07%	0.47%
Phonetic	2	9.52%	90.48%	0.45%
Jump-Twin	8	5.93%	94.07%	0.28%
				Total: 91.05%

It is hence shown that The New Algorithm will detect around 91.05% of all transmission errors if check-digit is not subject to any errors (an assumption made in section 3).

5 Benchmarking

Through benchmarking The New Algorithm with well-established standards, we can test its claimed efficiency in real-life situations. To simulate such situations, a computer program that generates 12 digits long (check-digit not included) decimal codes, measures how many random errors of a kind a given algorithm can detect. For each type of error and per algorithm, one million integers are randomly generated and randomly corrupted to create an erroneous integer of a wanted kind. Then program checks whether the check digits of the correct and erroneous integers are same or not. At the end of each test process, the total number and the ratio of the undetected errors is presented, together with the total performance of the error detection algorithm at the end of all error detection tests. Since many error detection algorithms are devised by the premise that the check digit will never be subject to corruption, although in reality it is, the program can simulate both cases; to be used both as a statistical analysis tool and as a stress-test in real-life situations.

Table 4: Ratio of undetected errors to all (check-digit incorruptible)

Algorithm	Error Type						
	S.-Digit	Trans.	J.-Trans.	Twin	Phonetic	J.-Twin	Weighted Total[1]
The New	0.00%	2.24%	2.14%	5.41%	8.57%	6.14%	0.33%
Damm	0.00%	0.00%	10.52%	8.69%	0.00%	11.39%	0.16%
Verhoeff	0.00%	0.00%	5.81%	4.49%	20.17%	5.82%	0.19%
Luhn	0.00%	2.13%	100%	6.79%	8.00%	11.11%	1.12%

Table 5: Ratio of undetected errors to all (check-digit corruptible)

Algorithm	Error Type						
	S.-Digit	Trans.	J.-Trans.	Twin	Phonetic	J.-Twin	Weighted Total[1]
The New	7.68%	2.05%	1.93%	5.18%	8.28%	5.86%	6.38%
Damm	7.67%	0.00%	9.55%	8.29%	0.00%	10.79%	6.22%
Verhoeff	7.72%	0.00%	5.25%	4.34%	19.64%	5.49%	6.28%
Luhn	7.71%	1.94%	90.89%	6.50%	7.73%	10.57%	7.13%

Damm algorithm Damm algorithm is developed by H. Michael Damm in 2004 in his doctorate thesis named *Total anti-symmetrische Quasigruppen* (en. *Totally anti-symmetric quasigroups*). It depends on the non-commutativity of totally anti-symmetric quasigroups to detect all transposition errors.[1] There are totally anti-symmetric quasigroups which can be used to detect all phonetic errors as well.

Verhoeff algorithm Verhoeff algorithm is developed by Jacobus Verhoeff in 1969. Being able to detect all single-digit errors *and* all transposition errors, it was the first of its kind. Verhoeff also made his groundbreaking investigation by classifying errors he observed in Dutch postal system, creating the table that is used throughout this essay as well. Verhoeff algorithm takes advantage of the non-commutative property of dihedral group of order 10, to detect all phonetic as well as single-digit errors.[5]

Luhn algorithm Luhn algorithm, also known as Luhn formula, is described by Hans Peter Luhn in 1954 in his patent application (US2950048 A) named *Computer for verifying numbers*[3]. It is designed primarily to detect single-digit and transposition errors. Luhn algorithm is now in the public domain and enjoys a widespread use, for instance, to detect errors in credit-card numbers of major issuers such as Visa, MasterCard, and many others.

[1]The percentage of undetected 6 most common transmission errors that are mentioned in section 2. Smaller is better.

6 Conclusion

Using the statistics about the percentages of most common transmission errors, a new algorithm has been developed from scratch. As can be seen in the results of the benchmark as well, the new algorithm has performed very closely to the established industry standards, and in some cases performs much better than them. The greatest flexibility of the new algorithm lies in usage of substitution arrays where different kinds of errors that might be encountered in the future can be modelled too (or if the percentages of the *most common transmission errors* are realized to be much different than thought, a new triple of substitution arrays can be brute-forced).

Finding the optimum way (*i.e.* the fastest) to brute-force triple of substitution arrays will lead to the discovery of *The Best Version of the New Algorithm*, although this being in the domain of computer science, is beyond the scope of this essay and left as an exercise for the reader.

References

[1] H. Michael Damm. Totally anti-symmetric quasigroups for all orders n≠2,6. *Discrete Mathematics*, 307(6):715 – 729, 2007.

[2] Joseph Kirtland. *Identification Numbers and Check Digit Schemes*. The Mathematical Association of America, first edition, jan 2001.

[3] Hans Peter Luhn. Computer for verifying numbers, aug 1960. US Patent 2,950,048.

[4] David Salomon. *Coding for Data and Computer Communications*. Springer, 2005.

[5] Jacobus Verhoeff. Error detecting decimal codes. (mathematical centre tracts, 29). *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik*, 51(3):240–241, 1971.

A Programs

Source codes of all of the programs that are mentioned in the essay are presented in this section for the sake of completeness of the essay and the hope that they might be helpful. All of the programs are written solely by the author of this essay and only for this essay. All of the programs are tested using Python® 3.5.2 on Ubuntu® 16.04 LTS (although any operating system where Python® interpreter can run should suffice); none of the programs has additional dependencies.

A.1 Source Code of the Program for Purely Brute-Force Approach

```
from itertools import combinations, permutations
from math import inf


def main():
    best_triple = None
    best_score = inf  # Set best score to Infinity so that we won't miss any
                      # triple (since Infinity is always greater than any
                      # integer, by definition).


    for A3n, A3n1, A3n2 in combinations(permutations(range(10)), 3):
        score = 0  # Reset score to 0 for every new triple.
        for x, y in combinations(range(10), 2):
            # Transposition and Jump-Transposition Errors
            if (A3n[x] + A3n1[y]) % 10  == (A3n[y] + A3n1[x]) % 10:   # Condition 1
                score += (102 + 8)
            if (A3n1[x] + A3n2[y]) % 10 == (A3n1[y] + A3n2[x]) % 10:  # Condition 2
                score += (102 + 8)
            if (A3n2[x] + A3n[y]) % 10  == (A3n2[y] + A3n[x]) % 10:   # Condition 3
                score += (102 + 8)

            # Twin and Jump-Twin Errors
            if (A3n[x] + A3n1[x]) % 10  == (A3n[y] + A3n1[y]) % 10:   # Condition 4
                score += (5 + 3)
            if (A3n1[x] + A3n2[x]) % 10 == (A3n1[y] + A3n2[y]) % 10:  # Condition 5
                score += (5 + 3)
            if (A3n2[x] + A3n[x]) % 10  == (A3n2[y] + A3n[y]) % 10:   # Condition 6
                score += (5 + 3)

        # Phonetic Errors
        for x in range(3, 10):
            if (A3n[1] + A3n1[x]) % 10  == (A3n[x] + A3n1[1]) % 10:   # Condition 7
                score += 5
            if (A3n1[1] + A3n2[x]) % 10 == (A3n1[x] + A3n2[1]) % 10:  # Condition 8
                score += 5
            if (A3n2[1] + A3n[x]) % 10  == (A3n2[x] + A3n[1]) % 10:   # Condition 9
                score += 5

        if score < best_score:
            best_score = score
            best_triple = (A3n, A3n1, A3n2)

            print("New Best Score:", score)
            print("\tA3n   ", A3n)
            print("\tA3n1  ", A3n1)
            print("\tA3n2  ", A3n2)

    print("The Best Score:", best_score)
    print("\tA3n   ", A3n)
    print("\tA3n1  ", A3n1)
    print("\tA3n2  ", A3n2)

if __name__ == '__main__':
    try:
        main()
        print("Completed!")
    except KeyboardInterrupt:
        print("\nStopped!")
```

A.2 Source Code of Optimized Brute Force

```
from itertools import combinations, permutations
from math import inf


def main():
    best_score = inf  # Set best score to Infinity so that we won't miss any
                      # triple (since Infinity is always greater than any
                      # integer, by definition).

    print("Computing...", end="\r")

    for A3n in permutations(range(10)):
        for A3n1 in array_generator(relative_to=A3n):
            # Skip current $A_{3n+1}$ substitution array if it has more than one
            # transposition error detecting defect with reference to $A_{3n}$.
            if count_transposition_defects(A3n, A3n1) > 1:
                continue

            for A3n2 in array_generator(relative_to=A3n1):
                # Skip current $A_{3n+2}$ substitution array if it has more than
                # one transposition error detecting defect per reference
                # (with references to $A_{3n}$ and $A_{3n+1}$).
                if count_transposition_defects(A3n, A3n2) > 1 \
                   or count_transposition_defects(A3n1, A3n2) > 1:
                    continue

                # Skip current $A_{3n+2}$ substitution array if the total sum of
                # penalty scores for minor (Jump–Transposition, Twin, Phonetic,
                # and Jump–Twin) error detecting defects is greater than 102.
                minor_defects_score = calculate_minor_defects_score(A3n,
                                                                    A3n1,
                                                                    A3n2)
                if minor_defects_score > 102:
                    continue

                print("A_{{3n}}      {}".format(A3n))
                print("A_{{3n+1}}    {}".format(A3n1))
                print("A_{{3n+2}}    {}".format(A3n2))

                score = 3 * 102 + minor_defects_score
                if score < best_score:
                    print("New Best Score: {} ({} minor)\n"
                          .format(score, minor_defects_score))
                    best_score = score
                elif score == best_score:
                    print("Another Best Score: {} ({} minor)\n"
                          .format(score, minor_defects_score))
                else:
                    print("Score: {} ({} minor)  [best: {}]\n"
                          .format(score, minor_defects_score, best_score))

                print("Computing...", end="\n", flush=True)


# Interface of array generation procedure.
#
# Generates a new array which none of its values at any position overlaps with
# another value at the same position in the `relative\_to` array.
def array_generator(relative_to):
    # Matrices are generated recursively; start recursive process.
    iterator = _array_generator_r(relative_to, ())
    while True:
        try:
            yield next(iterator)
        except StopIteration:
            return


def _array_generator_r(relative_to, current_array):
    # Terminating condition for recursion.
    if len(current_array) == 10:
```

```
72          yield current_array  # Yield the result...
73          return  # and terminate recursion.
74
75      # Set of decimal digits; curly brackets are used to create sets.
76      digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
77
78      possibilities = digits \
79                      - (set(current_array) | {relative_to[len(current_array)]})
80
81      # A possible next digit is any digit that:
82      #   1) is not equal to the value at the same position in
83      #      `relative\_to` array,
84      #   2) did not occur before in `current\_array`.
85      for possible_next_digit in \
86          digits - (set(current_array) | {relative_to[len(current_array)]}):
87      #   |          +------- 2 -------+   +---------------- 1 ----------------+|
88      #   |          +------------------------ Set Union -----------------------+
89      #   +-------------------------- Set Difference --------------------------+
90
91          iterator = _array_generator_r(relative_to,
92                                        current_array + (possible_next_digit,))
93          while True:
94              try:
95                  yield next(iterator)
96              except StopIteration:
97                  break
98
99
100 def count_transposition_defects(A0, A1):
101     count = 0
102
103     for x, y in combinations(range(10), 2):
104         # Condition for Detecting Transposition Errors
105         # (As Explained in Section 3).
106         if (A0[x] + A1[y]) % 10 == (A0[y] + A1[x]) % 10:
107             count += 1
108
109     return count
110
111
112 def calculate_minor_defects_score(A3n, A3n1, A3n2):
113     penalty_score = 0
114
115     for x, y in combinations(range(10), 2):
116         # Set of Conditions for Detecting Jump-Transposition Errors:
117         if (A3n[x] + A3n2[y]) % 10  == (A3n[y] + A3n2[x]) % 10:   # Condition 3
118             penalty_score += 8
119         if (A3n1[x] + A3n[y]) % 10  == (A3n1[y] + A3n[x]) % 10:   # Condition 1
120             penalty_score += 8
121         if (A3n2[x] + A3n1[y]) % 10 == (A3n2[y] + A3n1[x]) % 10:  # Condition 2
122             penalty_score += 8
123
124         # Set of Conditions for Detecting Twin and Jump-Twin Errors:
125         if (A3n[x] + A3n1[x]) % 10  == (A3n[y] + A3n1[y]) % 10:   # Condition 4
126             penalty_score += (5 + 3)
127         if (A3n1[x] + A3n2[x]) % 10 == (A3n1[y] + A3n2[y]) % 10:  # Condition 5
128             penalty_score += (5 + 3)
129         if (A3n2[x] + A3n[x]) % 10  == (A3n2[y] + A3n[y]) % 10:   # Condition 6
130             penalty_score += (5 + 3)
131
132     for x in range(3, 10):
133         # Set of Conditions for Detecting Phonetic Errors:
134         if (A3n[1] + A3n1[x]) % 10  == (A3n[x] + A3n1[1]) % 10:   # Condition 7
135             penalty_score += 5
136         if (A3n1[1] + A3n2[x]) % 10 == (A3n1[x] + A3n2[1]) % 10:  # Condition 8
137             penalty_score += 5
138         if (A3n2[1] + A3n[x]) % 10  == (A3n2[x] + A3n[1]) % 10:   # Condition 9
139             penalty_score += 5
140
141     return penalty_score
142
143
144 if __name__ == '__main__':
```

```
145     try:
146         main()
147         print("Completed!")
148     except KeyboardInterrupt:
149         print("\nStopped!")
```

A.3 Source Code for Analyzing a Triple

```
from collections import defaultdict
from itertools import combinations


def main():
    # Triple to be analyzed:
    A3n  = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
    A3n1 = (1, 7, 9, 6, 3, 0, 8, 5, 2, 4)
    A3n2 = (9, 6, 0, 2, 7, 5, 3, 8, 4, 1)

    defects = defaultdict(list)

    for x, y in combinations(range(10), 2):
        # Set of Conditions for Detecting Transposition Errors:
        if (A3n[x] + A3n1[y]) % 10  == (A3n[y] + A3n1[x]) % 10:   # Condition 1
            defects["Transposition"].append(("A3n<->A3n1", x, y))
        if (A3n1[x] + A3n2[y]) % 10 == (A3n1[y] + A3n2[x]) % 10:  # Condition 2
            defects["Transposition"].append(("A3n1<->A3n2", x, y))
        if (A3n2[x] + A3n[y]) % 10  == (A3n2[y] + A3n[x]) % 10:   # Condition 3
            defects["Transposition"].append(("A3n2<->A3n", x, y))

        # Set of Conditions for Detecting Jump-Transposition Errors:
        if (A3n[x] + A3n2[y]) % 10  == (A3n[y] + A3n2[x]) % 10:   # Condition 3
            defects["Jump-Transposition"].append(("A3n<->A3n2", x, y))
        if (A3n1[x] + A3n[y]) % 10  == (A3n1[y] + A3n[x]) % 10:   # Condition 1
            defects["Jump-Transposition"].append(("A3n1<->A3n", x, y))
        if (A3n2[x] + A3n1[y]) % 10 == (A3n2[y] + A3n1[x]) % 10:  # Condition 2
            defects["Jump-Transposition"].append(("A3n2<->A3n1", x, y))

        # Set of Conditions for Detecting Twin Errors:
        if (A3n[x] + A3n1[x]) % 10  == (A3n[y] + A3n1[y]) % 10:   # Condition 4
            defects["Twin"].append(("A3n<->A3n1", x, y))
        if (A3n1[x] + A3n2[x]) % 10 == (A3n1[y] + A3n2[y]) % 10:  # Condition 5
            defects["Twin"].append(("A3n1<->A3n2", x, y))
        if (A3n2[x] + A3n[x]) % 10  == (A3n2[y] + A3n[y]) % 10:   # Condition 6
            defects["Twin"].append(("A3n2<->A3n", x, y))

        # Set of Conditions for Detecting Jump-Twin Errors:
        if (A3n[x] + A3n2[x]) % 10  == (A3n[y] + A3n2[y]) % 10:   # Condition 6
            defects["Jump-Twin"].append(("A3n<->A3n2", x, y))
        if (A3n1[x] + A3n[x]) % 10  == (A3n1[y] + A3n[y]) % 10:   # Condition 4
            defects["Jump-Twin"].append(("A3n1<->A3n", x, y))
        if (A3n2[x] + A3n1[x]) % 10 == (A3n2[y] + A3n1[y]) % 10:  # Condition 5
            defects["Jump-Twin"].append(("A3n2<->A3n1", x, y))

    for x in range(3, 10):
        # Set of Conditions for Detecting Phonetic Errors:
        if (A3n[1] + A3n1[x]) % 10  == (A3n[x] + A3n1[0]) % 10:   # Condition 7
            defects["Phonetic"].append(("A3n<->A3n1", 1, x))
        if (A3n1[1] + A3n2[x]) % 10 == (A3n1[x] + A3n2[0]) % 10:  # Condition 8
            defects["Phonetic"].append(("A3n1<->A3n2", 1, x))
        if (A3n2[1] + A3n[x]) % 10  == (A3n2[x] + A3n[0]) % 10:   # Condition 9
            defects["Phonetic"].append(("A3n2<->A3n", 1, x))

    penalties = {
        "Transposition": 102,
        "Jump-Transposition": 8,
        "Twin": 5,
        "Phonetic": 5,
        "Jump-Twin": 3
    }

    total_score = sum(len(defects[defect_type])
                      * penalties[defect_type] for defect_type in defects)
    print("Results: {} points".format(total_score))

    for defect_type in defects:
        score = len(defects[defect_type]) * penalties[defect_type]

        print("\t{}: {} defects, {} points \n\t\t"
              .format(defect_type, len(defects[defect_type]), score), end="")
```

```
72          for defect in defects[defect_type]:
73              print(defect, end="  ")
74          print("\n")
75
76 if __name__ == '__main__':
77      main()
```

A.4 Source Code of Benchmarking Script

```
from math import log10
from random import SystemRandom
import re

random = SystemRandom()


# CONFIGURATION
# =============

# Length (\(l\)) of the decimal codes without check digit included.
length = 12

# Iterations per type of error.
iterations = 10 ** 6

# Set it to True if check digit too is subject to errors
check_digit_included = True


def main():
    algorithms = [the_new_algorithm, damm, luhn, verhoeff]
    tests = [test_single_digit, test_transposition, test_jump_transposition,
             test_twin, test_phonetic, test_jump_twin]

    print("Ratio of uncaught ... to all ({:,d} "
          "iterations per error type per algorithm):".format(iterations))

    for algorithm in algorithms:
        print("\tAlgorithm: {}".format(algorithm.__doc__))
        results = []
        for test in tests:
            result = test(algorithm)
            print("\t\t{:25}: {:6.2f}%  ({} undetected)"
                  .format(test.__doc__, result / iterations * 100, result))
            results.append(result / iterations * 100)

        weighted_total = results[0] * 79.1  \
                         + results[1] * 10.2  \
                         + results[2] * 0.8  \
                         + results[3] * 0.5  \
                         + results[4] * 0.5  \
                         + results[5] * 0.3
        print("\t\t{:25}: {:6.2f}%  (less is better)\n".format("Weighted Total",
            weighted_total / 100))

def the_new_algorithm(num):
    """The New Algorithm"""

    A3n  = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
    A3n1 = (1, 7, 9, 6, 3, 0, 8, 5, 2, 4)
    A3n2 = (9, 6, 0, 2, 7, 5, 3, 8, 4, 1)

    sum_ = 0

    for x, d in enumerate(num, 0):
        dx = int(d)
        if x % 3 == 0:
            sum_ += A3n[dx]
        elif x % 3 == 1:
            sum_ += A3n1[dx]
        else:
            sum_ += A3n2[dx]

        x += 1

    return str(sum_ % 10)


def damm(num):
    """Damm"""
```

```
72
73      t = ((0, 3, 1, 7, 5, 9, 8, 6, 4, 2),
74           (7, 0, 9, 2, 1, 5, 4, 8, 6, 3),
75           (4, 2, 0, 6, 8, 7, 1, 3, 5, 9),
76           (1, 7, 5, 0, 9, 8, 3, 4, 2, 6),
77           (6, 1, 2, 3, 0, 4, 5, 9, 7, 8),
78           (3, 6, 7, 4, 2, 0, 9, 5, 8, 1),
79           (5, 8, 6, 9, 7, 2, 0, 1, 3, 4),
80           (8, 9, 4, 5, 3, 6, 2, 0, 1, 7),
81           (9, 4, 3, 8, 6, 1, 7, 2, 0, 5),
82           (2, 5, 8, 1, 4, 3, 6, 7, 9, 0))
83
84      interim = 0
85
86      for d in num:
87          d = int(d)
88          interim = t[interim][d]
89
90      return str(interim)
91
92
93  def luhn(num):
94      """Luhn"""
95
96      o2 = ""
97
98      for i, d in enumerate(num):
99          if i % 2 == 0:
100             o2 += d
101         else :
102             o2 += str(int(d) * 2)
103
104     sum_ = 0
105
106     for d in o2:
107         sum_ += int(d)
108
109     return str((sum_ * 9) % 10)
110
111
112 def verhoeff(num):
113     """Verhoeff"""
114
115     d = ((0, 1, 2, 3, 4, 5, 6, 7, 8, 9),
116          (1, 2, 3, 4, 0, 6, 7, 8, 9, 5),
117          (2, 3, 4, 0, 1, 7, 8, 9, 5, 6),
118          (3, 4, 0, 1, 2, 8, 9, 5, 6, 7),
119          (4, 0, 1, 2, 3, 9, 5, 6, 7, 8),
120          (5, 9, 8, 7, 6, 0, 4, 3, 2, 1),
121          (6, 5, 9, 8, 7, 1, 0, 4, 3, 2),
122          (7, 6, 5, 9, 8, 2, 1, 0, 4, 3),
123          (8, 7, 6, 5, 9, 3, 2, 1, 0, 4),
124          (9, 8, 7, 6, 5, 4, 3, 2, 1, 0))
125
126     p = ((0, 1, 2, 3, 4, 5, 6, 7, 8, 9),
127          (1, 5, 7, 6, 2, 8, 3, 0, 9, 4),
128          (5, 8, 0, 3, 7, 9, 6, 1, 4, 2),
129          (8, 9, 1, 6, 0, 4, 3, 5, 2, 7),
130          (9, 4, 5, 3, 1, 2, 6, 8, 7, 0),
131          (4, 2, 8, 6, 5, 7, 3, 9, 0, 1),
132          (2, 7, 9, 3, 8, 0, 6, 4, 1, 5),
133          (7, 0, 4, 6, 9, 1, 3, 2, 5, 8))
134
135     inv = (0, 4, 3, 2, 1, 5, 6, 7, 8, 9)
136
137     c = 0
138     for i, n in enumerate(reversed(num + "0")):
139         ni = int(n)
140         c = d[c][p[i % 8][ni]]
141
142     return str(inv[c])
143
144
```

```
145 def test_single_digit(algorithm):
146     """Single-Digit Errors"""
147 
148     uncaught = 0
149 
150     for i in range(iterations):
151         number = random_number(length)
152         number += algorithm(number)
153 
154         while True:
155             error_index = random.randint(0,
156                 len(number) - (1 if check_digit_included else 2))
157             erroneous_number = number[:error_index] + str(random.randint(0, 9))\
158                                + number[error_index + 1:]
159 
160             if number != erroneous_number:
161                 break
162 
163         if algorithm(number[:-1]) == algorithm(erroneous_number[:-1]):
164             uncaught += 1
165 
166     return uncaught
167 
168 
169 def test_transposition(algorithm):
170     """Transposition Errors"""
171 
172     uncaught = 0
173 
174     for i in range(iterations):
175         number = random_number(length)
176         number += algorithm(number)
177 
178         while True:
179             error_index = random.randint(0,
180                 len(number) - (2 if check_digit_included else 3))
181             erroneous_number = number[:error_index] + number[error_index + 1] \
182                                + number[error_index] + number[error_index + 2:]
183 
184             if number != erroneous_number:
185                 break
186 
187         if algorithm(number[:-1]) == algorithm(erroneous_number[:-1]):
188             uncaught += 1
189 
190     return uncaught
191 
192 
193 def test_jump_transposition(algorithm):
194     """Jump-Transposition Errors"""
195 
196     uncaught = 0
197 
198     for i in range(iterations):
199         number = random_number(length)
200         number += algorithm(number)
201 
202         while True:
203             error_index = random.randint(0,
204                 len(number) - (3 if check_digit_included else 4))
205             erroneous_number = number[:error_index] + number[error_index + 2] \
206                                + number[error_index + 1] + number[error_index] \
207                                + number[error_index + 3:]
208 
209             if number != erroneous_number:
210                 break
211 
212         if algorithm(number[:-1]) == algorithm(erroneous_number[:-1]):
213             uncaught += 1
214 
215     return uncaught
216 
217 
```

```
218 def test_twin(algorithm):
219     """Twin Errors"""
220
221     uncaught = 0
222
223     for i in range(iterations):
224         while True:
225             number = random_number(length)
226             number += algorithm(number)
227
228             if check_digit_included:
229                 result = re.search(r"(\d)\1", number)
230             else:
231                 result = re.search(r"(\d)\1", number[:-1])
232
233             if result:
234                 index = result.span()[0]
235                 break
236
237         erroneous_digit = str(random.choice(tuple(set(range(10))
238                               - {int(number[index])})))
239         erroneous_number = number[:index] + 2 * erroneous_digit \
240                            + number[index + 2:]
241
242         if algorithm(number[:-1]) == algorithm(erroneous_number[:-1]):
243             uncaught += 1
244
245     return uncaught
246
247
248 def test_phonetic(algorithm):
249     """Phonetic Errors"""
250
251     uncaught = 0
252
253     for i in range(iterations):
254         while True:
255             number = random_number(length)
256             number += algorithm(number)
257
258             if check_digit_included:
259                 result = re.search(r"(1[3-9])|([3-9]0)", number)
260             else:
261                 result = re.search(r"(1[3-9])|([3-9]0)", number[:-1])
262
263
264             if result:
265                 index = result.span()[0]
266                 match = result.group()
267                 break
268
269         if number[index] == "1":
270             erroneous_number = number[:index] + number[index + 1] + "0" \
271                                + number[index + 2:]
272         elif number[index] >= "3":
273             erroneous_number = number[:index] + "1" + number[index] \
274                                + number[index + 2:]
275
276         if algorithm(number[:-1]) == algorithm(erroneous_number[:-1]):
277             uncaught += 1
278
279     return uncaught
280
281
282 def test_jump_twin(algorithm):
283     """Jump-Twin Errors"""
284
285     uncaught = 0
286
287     for i in range(iterations):
288         while True:
289             number = random_number(length)
290             number += algorithm(number)
```

```
291
292             if check_digit_included:
293                 result = re.search(r"(\d)\d\1", number)
294             else:
295                 result = re.search(r"(\d)\d\1", number[:-1])
296
297             if result:
298                 index = result.span()[0]
299                 break
300
301         erroneous_digit = str(random.choice(tuple(set(range(10))
302                               - {int(number[index])})))
303         erroneous_number = number[:index]  \
304                            + erroneous_digit  \
305                            + number[index + 1]  \
306                            + erroneous_digit  \
307                            +  number[index + 3:]  \
308
309         if algorithm(number[:-1]) == algorithm(erroneous_number[:-1]):
310             uncaught += 1
311
312     return uncaught
313
314
315 def random_number(length):
316     return str(random.randint(10 ** (length - 1), 10 ** length - 1))
317
318
319 if __name__ == '__main__':
320     main()
```