Testing of Heuristic Methods: A Case Study of Greedy Algorithm

A.C. Barus*, T.Y. Chen, D. Grant, F.-C. Kuo, and M.F. Lau

Faculty of Information and Communication Technologies,
Swinburne University of Technology
John St., Hawthorn 3122 Australia
{abarus,tchen,dgrant,dkuo,elau}@ict.swin.edu.au
http://www.swin.edu.au/ict/

Abstract. Algorithms which seek global optima are computationally expensive. Alternatively, heuristic methods have been proposed to find approximate solutions. Because heuristic algorithms do not always deliver exact solutions it is difficult to verify the computed solutions. Such a problem is known as the *oracle problem*. In this paper, we propose to apply Metamorphic Testing (MT) in such situations because MT is designed to alleviate the oracle problem and can be automated. We demonstrate the failure detection capability of MT on testing a heuristic method, called the Greedy Algorithm (GA), applied to solve the set covering problem (SCP). The experimental results show that MT is an effective method to test GA.

Keywords: heuristic method, greedy algorithm, metamorphic testing.

1 Introduction

Normally algorithms which deliver global optima are computationally expensive. Alternatively, heuristic methods have been proposed to provide approximate optima. These heuristic algorithms may be based on educated guesses, intuitive judgement, or simply common sense to seek answers which are hopefully close to the global optima. Such algorithms may deliver solutions that are global optima, close to global optima, local optima or close to local optima. In other words, there are uncertainties in the solutions delivered by these methods. Examples of heuristic methods include algorithms proposed by Johnson for combinatorial problems [1], by Bodorik *et al.* for distributed query processing [2], and by Cheng *et al.* for real-time data aggregation in wireless sensor networks [3].

When such algorithms are implemented as software, it is important to ensure the correctness of the implementation. The availability of a *test oracle* — a mechanism to verify the output of software — is necessary to determine whether the software passes the test undertaken. Heuristic methods may not give exact solutions for the computed problems. Therefore, it is difficult to verify outputs

* Corresponding author.

Z. Huzar et al. (Eds.): CEE-SET 2008, LNCS 4980, pp. 246–260, 2011.

of the corresponding software, which is known as the *test oracle problem* (or, simply *the oracle problem*) in Software Testing. *Metamorphic testing* (MT) was developed to deal with the oracle problem[4]. MT can be used to validate computed outputs automatically without the presence of a test oracle. It uses some properties of the computed problem, which are known as metamorphic relations (MRs), to help validate the correctness of the computed outputs.

In this study we propose to apply MT to software implementing the *greedy algorithm* (GA), which is a simple and straightforward heuristic method, to solve the *set covering problem* (SCP) [1]. Particularly, we conduct a case study aimed at demonstrating the failure detection capability of MT.

This paper is organized as follows: Section 2 presents the definition of MT, Section 3 explains GA on SCP, and Section 4 describes the metamorphic relations (MRs) identified in this study. Details of experimental work, results and discussions are presented in Section 5, and Section 6 concludes the paper.

2 The Metamorphic Testing (MT)

A test oracle is a mechanism that can be used by testers to verify the correctness of computed outputs of a program [5]. We encounter the test oracle problem when (i) there is no such oracle or (ii) the application of such an oracle becomes too expensive. To alleviate this problem, Chen *et al.* [4] have developed the metamorphic testing (MT) approach which has been successfully applied in various application domains ([7], [8], [9], [10], [11], [12]).

MT is a property-based testing method. We use the sine function to illustrate the idea of MT. Suppose P is a program that computes the sine function. We assume that we do not have an oracle for this problem - i.e., we do not know exactly what is the value of the sine of an arbitrary input. Let 0.49 (radians) be a test input of P. After executing P with 0.49, the corresponding output is P(0.49). Due to the lack of an oracle, P(0.49) may be correct but we have no way to verify it. The key idea of MT is to use a relationship called a *test case relation* to generate some *follow-up test cases*, whose behaviour is predictable from the original test case. If the predicted behaviour is not exhibited, then this is indicative of an error in P. In the case of the sine function, we might choose $2\pi+0.49$ and $4\pi+0.49$ as follow-up test cases, as the sine function ought to yield the same value for these as for 0.49. The test case of 0.49 is referred to as the *source test case* in order to distinguish it from the follow-up test cases. As noted, for the sine function, we expect that $sin(0.49) = sin(2\pi+0.49) = sin(4\pi+0.49)$. This relationship is referred to as the *test result relation.* After executing P with $2\pi+0.49$ and $4\pi+0.49$, we can then check whether the following equalities hold: $P(0.49) = P(2\pi + 0.49) = P(4\pi + 0.49)$. If either one of the equalities does not hold, we know that P contains error. As exemplified by the sine function, a test case relation may involve the output of the source test case. The success of MT relies on the existence of a metamorphic relation (MR) which comprises of the two interrelated relations: the test case relation and the test result relation. Once an MR is defined, the generation of the follow-up test cases from the source test case and the verification of the test result relationship can be automated.

The following are formal definitions of MR and the procedure of MT [13]:

Definition MR. Suppose a function f has inputs, $\mathrm{I}_1 = \{x_1, x_2, ..., x_i\}$ where $i \geq 1$ and let $\mathrm{O}_1 = \{f(x_1), f(x_2), ..., f(x_i)\}$ be the corresponding outputs. Let $\mathrm{S} = \{f(x_{s_1}), f(x_{s_2}), ..., f(x_{s_k})\}$ denote a subset of O_1 where S may be empty. Let $I_2 = \{x_{i+1}, x_{i+2}, ..., x_j\}$ be other inputs to f where $j \geq i+1$ and $O_2 = \{f(x_{i+1}), f(x_{i+2}), ..., f(x_j)\}$ be the corresponding outputs. Suppose there exists a relation R_1 among I_1, S and I_2, and another relation R_2 among I_1, I_2, O_1 and O_2 such that R_2 must be satisfied whenever R_1 is satisfied. Then, a metamorphic relation MR can be defined as:

$\mathbf{MR} = \{ (x_1, x_2, ..., x_j, f(x_1), f(x_2), ..., f(x_j)) | \mathrm{R}_1(x_1, x_2, ..., x_i, f(x_{s_1}), f(x_{s_2}), ..., f(x_{s_k}), x_{i+1}, x_{i+2}, ..., x_j) \rightarrow \mathrm{R}_2(x_1, x_2, ..., x_j, f(x_1), f(x_2), ..., f(x_j))\}$
Elements of I_1 and I_2 are referred to as source test cases and follow-up test cases, respectively. Relations R_1 and R_2 are referred to as the test case relation and the test result relation, respectively.

Procedure MT. Suppose the function f is implemented by a program P. The procedure of MT using the MR described in the above definition consists of the following steps:

1. Run P using a series of test cases I_1 as source test cases and get the corresponding outputs O_1.
2. Use R_1, I_1, and O_1 to generate follow-up test cases I_2.
3. Run P using I_2 as inputs to get the corresponding outputs O_2.
4. Check the relation R_2: if R_2 does not hold then a failure is revealed.

As program failures may be sensitive to different MRs, it is recommended to identify more than one MR when applying MT. For our sine function example, other possible MR is as follow. For any inputs x_1 and x_2 where $\pi/2 < x_1 < x_2 < 3\pi/2$, $\sin(x_1)$ must be greater than $\sin(x_2)$. Formally speaking, $\mathrm{MR}_{\sin_2}$: $\pi/2 < x_1 < x_2 < 3\pi/2 \rightarrow \sin(x_1) > \sin(x_2)$.

3 Greedy Algorithm on Set Covering Problem

The set covering problem (SCP) is one of the NP-complete problems [14] that has been well studied in computer science and complexity theory. Given a set of objects O and a set of requirements R that can be collectively satisfied by objects in O, SCP is to find the smallest subset of O that satisfies all requirements in R. For ease of discussion, in this paper, we use a key to represent an object in O and a lock to represent a requirement in R. SCP can be rephrased as a key-lock problem (KLP). Given a set K of keys that can collectively open a set L of locks, find a set of keys in K of smallest size that can open all locks in L.

In this study, we focus on the greedy algorithm (GA) as one of many heuristic solutions to solve SCP. We refer to the expression of GA in [15] that can be translated to pseudocode presented in the Appendix. GA consists of a series of search steps. In each step, it looks for a local optimum which is a key that opens the largest number of locks that cannot be opened by previously selected keys. In general, the set of keys selected by GA may not be a global optimum.

Formally, suppose there are a set of keys, $\mathrm{K} = \{k_1, k_2, ..., k_x\}$ and a set of locks, $\mathrm{L} = \{l_1, l_2, ..., l_y\}$ where $x,\ y > 0$. For every pair $(k_m, l_n) \in (\mathrm{K} \times \mathrm{L})$, we define $r(m, n)$ as a relationship between key k_m and lock l_n such that $r(m, n) = 1$ if k_m opens lock l_n and $r(m, n) = 0$, otherwise. Initially, the relationship $r(m, n)$ is stored in the $(m, n)^{th}$ element of matrix M, $\forall m,\ 1 \leq m \leq x$, $\forall n,\ 1 \leq n \leq y$. However, M consists of $(x{+}1)$ rows and $(y{+}1)$ columns. The additional column contains all identifiers for the keys in K and the additional row contains all identifiers for the locks in L. Each $M[m][n]$ corresponds to $r(m, n)$, the relationship between key k_m and lock l_n where the key identifier k_m is stored in $M[m][y+1]$, $\forall m,\ 1 \leq m \leq x$ and the lock identifier l_n is stored in $M[x+1][n]$, $\forall n,\ 1 \leq n \leq y$. Intuitively speaking, $M[m][\,]$ (the m^{th} row), $\forall m,\ 1 \leq m \leq x$ corresponds to key k_m and $M[\,][n]$ (the n^{th} column), $\forall n,\ 1 \leq n \leq y$ corresponds to lock l_n. Note that after GA in the Appendix selects the first key, its corresponding rows and columns (representing locks opened by the key) will be removed. Hence, we need the extra row and column to identify the remaining keys and locks in M. Matrix M can be presented as follows:

$$M = \begin{pmatrix} r(1,1) & r(1,2) & \ldots & r(1,y) & k_1 \\ r(2,1) & r(2,2) & \ldots & r(2,y) & k_2 \\ . & . & \ldots . & & . \\ . & . & \ldots . & & . \\ r(x,1) & r(x,2) & \ldots & r(x,y) & k_x \\ l_1 & l_2 & \ldots & l_y & \end{pmatrix}$$

Given a set of keys K, a set of locks L, and their relationship stored in a matrix M, GA considers the number of locks in L that can be opened by each key in K (in other words, the total number of "1"s appearing in each row of M) in order to make a series of decisions regarding the local optimum. As all key identifiers of K and all lock identifiers of L have been embedded in M, GA will merely analyse M to guide its search process. Basically, GA's search consists of iterations of the following steps.

1. For each row of M, count the number of locks opened by the key corresponding to the row. Note: hereafter, the number of locks opened by a key k is referred as numOpenL(k).
2. Select a key (say k_x) in M such that k_x can open the most locks in M. In other words, numOpenL(k_x) is the largest among all numOpenL(k) for all keys k in M. In case of a tie, select the key with the smallest row index.
3. Append the selected key k_x to O, an array storing output elements of GA.
4. Remove columns in M corresponding to all locks that can be opened by k_x.
5. Remove the row in M corresponding to k_x.
6. Repeat steps 1 to 5 until M has one column left.

At the end of the search, M only contains a column storing the identifiers of unselected keys. However, if all keys are selected into O, this column is empty. The output of GA is O which contains all keys selected in the search process.

A test oracle for GA can be obtained manually only when the size of M is small. Even when M is moderate in size (say, with 30 or more rows and columns),

there is an *oracle problem* for testing the implementation of GA. Therefore, in this study we propose to use Metamorphic Testing (MT) to verify the implementation of GA.

As mentioned, GA uses and manipulates M to determine the local optima based on the largest numOpenL(k_m), for each k_m in M. Details of the algorithm are presented in the Appendix.

We illustrate GA in an instance of KLP, namely *KL-example*.

KL-example. Suppose there is a set of keys, $\{k_1, k_2, ..., k_5\}$, a set of locks, $\{l_1, l_2, ..., l_9\}$, and the associated input matrix M_{KL} to GA is as follows:

$$M_{KL} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & k_1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & k_2 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & k_3 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & k_4 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & k_5 \\ l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 & l_9 & \end{pmatrix}$$

Since k_4 opens the most number of locks in M_{KL} (that is four locks: l_1, l_3, l_7 and l_8), GA selects k_4 as a local optimum and appends k_4 to O so that now O $= [k_4]$. The row corresponding to k_4 and the columns corresponding to l_1, l_3, l_7 and l_8 are deleted from the matrix. As a result, the matrix is updated as follows:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & k_1 \\ 0 & 1 & 0 & 1 & 0 & k_2 \\ 1 & 0 & 0 & 0 & 0 & k_3 \\ 0 & 0 & 1 & 0 & 1 & k_5 \\ l_2 & l_4 & l_5 & l_6 & l_9 & \end{pmatrix}$$

In the second round of search, both k_2 and k_5 open the most number of remaining locks. However, because the row index of k_2 is smaller than the row index of k_5, GA selects k_2 so that now O $= [k_4, k_2]$. The row corresponding to k_2 and columns corresponding to locks opened by k_2 (that is, l_4 and l_6) are deleted from the matrix. As a result, the matrix is updated as follows:

$$\begin{pmatrix} 0 & 0 & 0 & k_1 \\ 1 & 0 & 0 & k_3 \\ 0 & 1 & 1 & k_5 \\ l_2 & l_5 & l_9 & \end{pmatrix}$$

Using the same procedure, in the third round, k_5 is picked as the local optimum so that now O $= [k_4, k_2, k_5]$. Then the matrix is updated as follows:

$$\begin{pmatrix} 0 & k_1 \\ 1 & k_3 \\ l_2 & \end{pmatrix}$$

Finally, GA selects k_3 so that now O = $[k_4, k_2, k_5, k_3]$ and deletes all columns and rows associated with k_3. Then, the matrix is updated as follows:

$$\begin{pmatrix} k_1 \\ \end{pmatrix}$$

At this point, the matrix contains only one column. Hence, GA stops the search process and returns its output O = $[k_4, k_2, k_5, k_3]$. However, in this example, we can see that the global minimum solution is $[k_2, k_3, k_5]$. So how can we know whether the implementation of GA is correct? (We emphasise that we refer to our implementation of GA. We do not know what the output of GA should be. We do have the output of our program. How do we check it is correct?)

4 Metamorphic Relation (MR)

Identification of MRs is an essential step in conducting MT. In this study, we propose nine MRs, MR1 to MR9, in applying MT on GA to solve SCP. We use M and M' to denote a source test case and the follow-up test case respectively in describing the MRs. To illustrate those MRs further, we reuse the KL-example with M_{KL} as the source test case, as discussed in the previous section. The follow-up test case generated using a particular MR, say MR-i, is denoted as M'(MR-i). In the following discussion, the highlighted cells of M'(MR-i) indicate cells modified after applying MR-i. We also use O and O$'$ to denote the output of the source and follow-up test cases, respectively and $numO$ to denote the size (the number of elements) of O. Note that unlike the initial M_{KL} specified in Section 3, the follow-up test cases M'(MR-i) in this section may have the m^{th} row corresponding to a key other than k_m in K and the n^{th} column to a lock other than l_n in L. We now discuss the nine MRs.

1. MR1 (**Interchanging columns related to the key-lock relationship**). If we generate the follow-up test case M' by interchanging two columns of the source test case M which are related to the key-lock relationship, then O$'$ = O. (This corresponds to re-labelling two locks, and has no effect on the keys chosen by GA) For example, if we apply MR1 by interchanging columns related to l_2 and l_4 in M_{KL},

$$M'(\text{MR1}) = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & k_1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & k_2 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & k_3 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & k_4 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & k_5 \\ l_1 & l_4 & l_3 & l_2 & l_5 & l_6 & l_7 & l_8 & l_9 & \end{pmatrix}$$

   then we have O$'$ = O = $[k_4, k_2, k_5, k_3]$
2. MR2 (**Adding a useless key row**). A key is said to be *useless* if it cannot open any locks. If M' is obtained from M by adding a row corresponding to a useless key, then O$'$ = O. For example, if we apply MR2 to generate M'(MR2) by adding a row corresponding to the useless key k_6 in M_{KL},

$$M'(\text{MR2}) = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & k_1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & k_2 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & k_3 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & k_4 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & k_5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & k_6 \\ l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 & l_9 & \end{pmatrix}$$

then we have $\text{O}' = \text{O} = [k_4, k_2, k_5, k_3]$

3. MR3 (**Adding an insecure lock column**). A lock is *insecure* if it can be opened by any key. If M' is obtained from M by adding a column corresponding to an insecure lock, then $\text{O}' = \text{O}$. For example, if we apply MR3 to generate a follow-up test case $M'(\text{MR3})$ by adding a column corresponding to the insecure lock l_{10} in M_{KL},

$$M'(\text{MR3}) = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & k_1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & k_2 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & k_3 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & k_4 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & k_5 \\ l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 & l_9 & l_{10} & \end{pmatrix}$$

then we have $\text{O}' = \text{O} = [k_4, k_2, k_5, k_3]$

4. MR4 (**Rearranging rows corresponding to the selected keys on top while preserving their order**). M' is obtained from M by rearranging 1) the rows of the selected keys to be the top rows of M' while preserving their selected order and 2) those rows related to unselected keys all placed below the rows of selected keys, in any order. Then, we have $\text{O}' = \text{O}$. We want to preserve the ordering so that the checking of the test result relation ($\text{O}' = \text{O}$) can be implemented in linear time when O is stored as a 1-dimensional array. For example, if we apply MR4 to generate a follow-up test case $M'(\text{MR4})$ from M_{KL},

$$M'(\text{MR4}) = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & k_4 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & k_2 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & k_5 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & k_3 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & k_1 \\ l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 & l_9 & \end{pmatrix}$$

then we have $\text{O}' = \text{O} = [k_4, k_2, k_5, k_3]$

5. MR5 (**Adding a combined key of two consecutively selected keys**). If two keys are combined together as one key, the resulting key can open all locks that can be opened by any individual key. The resulting key is referred to as a *combined key* of the two individual keys. In applying MR5, M' is obtained from M by appending a row corresponding to a key, say k_x, which combines two selected keys. To simplify the discussion, we restrict our discussion to the second and third selected keys, namely k_{o_2} and k_{o_3},

respectively. In fact, this can be generalized to any two consecutive selected keys. To generate the follow-up test case, there are two interrelated steps:

(a) (Adding a combined key) Append a row corresponding to k_x — the combined key of k_{o_2} and k_{o_3}. In other words, the relationship between k_x and each lock l_n in M can be defined as follow: $r(x, n) = 1$ if either $r(o_2, n) = 1$ or $r(o_3, n) = 1$.

(b) (Preserving the ordering) Since the combined key k_x may open more locks than the first selected key k_{o_1}, we need to add N extra locks that can only be opened by k_{o_1} so that k_{o_1} is selected before k_x in the new solution. By doing so, we can preserve the ordering of the selected keys so that the test result relation of this MR is simple and easily verified. If numOpenL(k_x) > numOpenL(k_{o1}), then the number of extra locks needed for k_{o_1} is numOpenL(k_x) - numOpenL(k_{o1}). Otherwise, no extra locks are needed. Note: a lock that can be opened by one and only one key k_x is referred to as an *exclusive lock* for k_x.

Then, we have k_x in the second selected key in O′ and O′ - k_x = (O - k_{o_2}) - k_{o_3}. Note that the minus (-) operator here (and also in the rest of the paper) denotes that the right operand, in this case a key, is deleted from the left operand, in this case an array of keys. For example, if we apply MR5 to generate a follow-up test case M'(MR5) by appending k_6 — a combined key of k_2 and k_5 — in M_{KL} and then appending two exclusive locks to k_4 (l_{10} and l_{11}) because numOpenL(k_4) - numOpenL(k_6) = 6 − 4 = 2),

$$M'(\text{MR5}) = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & k_1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & k_2 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & k_3 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & k_4 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & k_5 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & k_6 \\ l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 & l_9 & l_{10} & l_{11} & \end{pmatrix}$$

then we have k_6 = O′[2] and O′ - k_6 = (O - k_2) - k_5. In this example, O′ = $[k_4, k_6, k_3]$.

6. MR6 (**Excluding a selected key other than the first selected key while preserving the order of the remaining selected keys**). By excluding a selected key other than the first selected key, say k_x, so that it will not be selected in the next output, we need to reset the selected keys preceding k_x in the original output, so that they can open the locks that can be opened by the excluded key. In order to preserve the ordering of the remaining selected keys for ease of checking the two solutions, we need to take some precautions. First, we cannot simply reset the selected key preceding k_x in O to be able to open all locks opened by k_x. This is because the total number of locks opened by the key will increase and hence, it is possible that that key would then be selected earlier. Second, we cannot simply reset the first selected key k_{o_1} to open all these locks as well. This is because it

is possible that the selection order of the remaining keys might be upset, as shown in the following example.

$$M = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & k_1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & k_2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & k_3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & k_4 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & k_5 \\ l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 & l_9 & l_{10} & l_{11} & l_{12} & l_{13} & l_{14} & \end{pmatrix}$$

In this case, we have $\mathrm{O} = [k_1, k_2, k_3, k_4, k_5]$. Suppose we would like to exclude k_5 in the next output by resetting the first selected key k_1 as follows:

$$M' = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & k_1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & k_2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & k_3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & k_4 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & k_5 \\ l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 & l_9 & l_{10} & l_{11} & l_{12} & l_{13} & l_{14} & \end{pmatrix}$$

Then $\mathrm{O}' = [k_1, k_3, k_2, k_4]$ and, hence, the ordering of these previously selected keys is different from that in O. Therefore, in applying MR6, M' is obtained from M by resetting the first selected key k_{o_1} so that it can only open those locks that can be opened by k_x (say k_x is $\mathrm{O}[i]$, $2 \leq i \leq numO$) but not by any key k_y where $k_y = \mathrm{O}[j]$, $2 \leq j < i$. In other words, when $r(x, n) = 1$ and $r(y, n) = 0$ then $r(o_1, n)$ is updated to "1", for all l_n in M. Then, we have $\mathrm{O}' = \mathrm{O}$ - k_x. For example, suppose we want to apply MR6 to generate M'(MR6) by excluding k_5 of M_{KL} in O'. The first selected key in O, k_4 is reset so that it can open l_5 and l_9, which are locks that can be opened by k_5 but not by the other selected key preceding k_5, which is k_2. The follow-up test case is given by

$$M'(\mathrm{MR6}) = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & k_1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & k_2 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & k_3 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & k_4 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & k_5 \\ l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 & l_9 & \end{pmatrix}$$

and we then have $\mathrm{O}' = \mathrm{O}$ - k_5. In this example, $\mathrm{O}' = [k_4, k_2, k_3]$.

7. MR7 (**Deleting a selected key while preserving the order of the remaining selected keys**)**.** This MR is different from MR6 because the row corresponding to a selected key say, k_x, is deleted from M. All columns related to locks opened by k_x are also deleted from M. However, in order to preserve the order of the remaining selected keys, we have to add extra columns corresponding to exclusive locks to each of the selected keys preceding k_x in O. In more detail, M' is obtained from M in the following manner:

(a) The row corresponding to k_x is deleted from M'.
(b) All locks that can be opened by k_x are deleted from M'.
(c) For each k_y, a selected key preceding k_x in O, N exclusive locks to k_y are appended to M' where N is the number of locks that can be opened by both k_x and k_y. Note: a lock that can be opened by both k_x and k_y is referred to as *share lock* of k_x and k_y. The checking of the share locks is according to the order of the selected keys preceding k_x in O. Once a lock has been considered as a share lock of any pair of keys, it cannot be used as a share lock of other pairs.

Then, we have O′ = O - k_x. For example, if we apply MR7 to generate a follow-up test case M'(MR7) by deleting k_3 in M_{KL}, the row corresponding to k_3 and all columns corresponding to locks opened by k_3 are deleted. For the keys preceding k_3 in O (that is k_4, k_2, and k_5), we find that k_3 and k_4 have two share locks; k_3 and k_2 have no share locks; and k_3 and k_5 also have no share locks. Accordingly, we need to add two locks, l_{10} and l_{11}, that are exclusive to k_4. Hence, the follow-up test case is:

$$M'(\text{MR7}) = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & k_1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & k_2 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & k_3 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & k_4 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & k_5 \\ l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 & l_9 & l_{10} & l_{11} & \end{pmatrix}$$

Note that the darker highlighted cells will be deleted. Then, we have O′ = O - k_3. In this example, O′ = $[k_4, k_2, k_5]$.

8. MR8 (**Adding an exclusive lock to an unselected key**). If M' is obtained from M by adding an extra column that corresponds to an exclusive lock to a particular unselected key, then we expect that the unselected key will be in the new solution. Since the unselected key may open other existing locks, some previously selected key may be excluded from the solution. However, we cannot pre-determine which key will be excluded. For example, if we apply MR8 to generate a follow-up test case M'(MR8) by adding an exclusive lock l_{10} to k_1 — a key which is unselected in O, then

$$M'(\text{MR8}) = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & k_1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & k_2 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & k_3 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & k_4 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & k_5 \\ l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 & l_9 & l_{10} & \end{pmatrix}$$

and we have k_1 in O′. In this example, O′ = $[k_1, k_2, k_3, k_5]$. Please note that the key k_4 (originally in O) is not in O′ .

9. MR9 (**Adding an exclusive lock to an unselected key while preserving the order of the previously selected keys**). This MR is different from MR8 because the order of the previously selected keys is preserved. In

more details, the follow-up test case M' of this MR is obtained as follows: Add an extra column that corresponds to an exclusive lock to a particular unselected key and reset the unselected key so that it can open and only open that exclusive lock. Then, we are guaranteed that the order of the previously selected keys in both solutions are the same. In other words, we have O′ - k_x = O. For example, if we apply MR9 to generate a follow-up test case M'(MR9) by adding an exclusive lock l_{10} to k_1 and resetting k_1 such that it can open and only open l_{10}, then

$$M'(\text{MR9}) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & k_1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & k_2 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & k_3 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & k_4 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & k_5 \\ l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 & l_9 & l_{10} & \end{pmatrix}$$

Then, we have O′ - k_1 = O. In this example, O′ = [k_4, k_2, k_1, k_3, k_5]

5 Results and Observation

We applied MT to test a program implementing the algorithm in the Appendix. It was written in Java and the testing was conducted in a Linux environment. We applied the fault seeding technique — which is widely used in software testing experiments ([9], [16])— to create five faulty versions of the program. We inserted one bug into each faulty version. The bug insertion process used *mutant operators* introduced by Agrawal *et al.* [17]. The mutant operators were chosen randomly and independently. Faults resulted from those mutant operators are detailed in Table 1.

Source test cases were generated in matrix format with random sizes of rows and columns in the range between 1 and 100 (there were no empty matrix inputs). The content of matrixes (K, L and their relationship) were also generated randomly, subject to the constraint that every row had at least one "1" and every column had at least one "1" (in other words, every key can open at least one lock and every lock can be opened by at least one key). We generated a test suite of one hundred test cases. These test cases were used as source test cases for testing faulty programs V1 to V5 using the nine MRs introduced in Section 4. The results are presented in Table 2.

From Table 2, we observe that most of the MRs contributed to reveal failure in at least one faulty version, except MR1, which did not reveal any failures. This is reasonable because we cannot guarantee that every MR can reveal any failure. Such a case simply exhibits the general limitation of software testing, that is, there is no guarantee that a certain fault will be exposed.

We consider an entry in Table 2 as the execution of a faulty version using one hundred pairs of source test cases and follow-up test cases corresponding to a specific MR. Therefore, we have 9 x 5 = 45 entries in our experiments of which 24 entries are non-zero. If all pairs of source test cases and follow-up

Table 1. Faults in faulty programs V1 to V5 based on pseudocode in the Appendix

Faulty Program	Line#	Original Statement	Faulty Statement
$V1$	8	$i = 1$	$i = 2$
$V2$	11	$M[i][j] = 1$	$M[i][j] \neq 1$
$V3$	14	$noOLocks > maxOLocks$	$noOLocks \geq maxOLocks$
$V4$	15	$maxOLocks := noOLocks$	$maxOLocks := i$
$V5$	23	$i = 1$	$i = 2$

Table 2. Percentage of tests revealing failures in the faulty versions by MRs

	MR1	MR2	MR3	MR4	MR5	MR6	MR7	MR8	MR9
V1	0	6	0	93	0	0	0	18	11
V2	0	100	0	2	100	100	69	0	0
V3	0	0	0	100	95	0	0	0	0
V4	0	30	43	100	80	4	25	0	8
V5	0	10	31	56	67	31	31	0	0

test cases are considered (a total of 4500 pairs), there are 1210 pairs (26.9 %) which reveal failures. Given that there is no way to verify the correctness of the computed outputs, and that MT could be fully automated once the MRs have been defined, a failure detection effectiveness of 26.9 % is in fact very encouraging. It should also be noted that to apply MT in generating test cases, the software test engineers only require minimal programming skills and the relevant problem domain knowledge.

Different failures in our experiments can be detected by different MRs because of their different characteristics. For example, the faulty version V2 selects the key that opens the least number of locks instead of the maximum. Hence, MR2 can reveal this failure because it adds a useless key (a key that cannot open any lock) in the follow-up test case - a key that will be chosen in the faulty program but not in a correct program. On the contrary, MR3 cannot reveal this failure because it adds an insecure lock (a lock that can be opened by every key) in the follow-up test case, and such a key will never be chosen in V2.

As our experiments show that some MRs, say MR4, can reveal more failures than the others, the selection of good MRs is crucial. Obviously, all MRs could potentially be used in MT. However, due to resource limitations — in industrial examples, running test cases even on powerful computers may take many hours — it is essential to identify which MRs should be given higher priority. Chen *et al.* have conducted some case studies on the selection of useful MRs in MT[9]. One of their general conclusions is that the bigger the differences between program executions on the source and follow-up test cases, the better are the MRs.

6 Conclusion

Heuristic methods do not deliver exact answers. Accordingly, software implementing these methods are subject to the oracle problem. We propose to apply metamorphic testing (MT) — an automated property-based testing method — to test such software. In this study, we investigated the application of MT on the greedy algorithm (GA) applied to the set covering problem. We identified nine MRs to apply MT on GA and conducted testing on five faulty versions of a program implementing GA. Based on the experimental results, we found that MT reveals at least one failure in each of the faulty versions under investigation. It demonstrates the fault detection capability of MT in automatically testing heuristic programs such as GA.

The experimental results report that some MRs can contribute in revealing more failures than others. This is due to the different characteristics of the MRs. Hence, it is crucial to select good MRs particularly when the resources are limited for conducting testing.

Our study is limited by the following threats. (1) Threat to internal validity (e.g. experimental setup). We have carefully examined the processes to make sure that no such threat exists. (2) Threat to construct validity (e.g. evaluation method). The failure detection effectiveness of MT is only studied based on the percentage of pairs of source and follow-up test cases which revealed failures. It is worthwhile to examine the effectiveness of MT in other measurements, as part of the future work. (3) Threat to external validity (e.g. number and size of subjects in the case study). We only applied MT to one simple program of GA in the case study. In the future work, it is interesting to investigate the effectiveness of MT using more and bigger sized programs.

In the future, we also propose to study a new approach for defining MRs. In the new approach, prior to defining MRs, we will attempt to use knowledge of possible faults in the software under test, (whereas our current approach is *ad-hoc*, defining the MRs independently from the possible faults or in this study, independently from the mutant generation). The faults can be identified at either high level (specification-based) or lower level (source code-based). Afterwards, we will attempt to define MRs for targeting such faults. We expect that this approach may contribute a more effective set of MRs than our current approach.

Acknowledgment. We would like to acknowledge the support given to this project by an Australian Research Council Discovery Grant (ARC DP0771733).

References

1. Johnson, D.S.: Application of algorithms for combinatorial problems. Journal of Computer and System Science 9(3), 256–278 (1974)
2. Bodorik, P., Riordon, J.S.: Heuristic algorithms for distributed query processing. In: Proceedings of the First International Symposium on Databases in Parallel and Distributed Systems, DPDS'88. IEEE Computer Society Press, Los Alamitos (2000)

3. Cheng, H., Liu, Q., Jia, X.: Heuristic algorithms for real-time data aggregation in wireless sensor networks. In: Proceedings of the 2006 International Conference on Wireless Communication and Mobile Computing, pp. 1123–1128 (2006)
4. Chen, T.Y., Cheung, S.C., Yiu, S.M.: Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong (1998)
5. Weyuker, E.J.: On testing non-testable programs. The Computer Journal 25(4), 465–470 (1982)
6. Chan, W.K., Cheung, S.C., Leung, K.R.P.H.: Towards a metamorphic testing methodology for service-oriented software applications. In: Proceedings of the 5th International Conference on Quality Software(QSIC 2005), pp. 470–476. IEEE Computer Society Press, Los Alamitos (2005)
7. Chan, W.K., Cheung, S.C., Leung, K.R.P.H.: A metamorphic testing approach for online testing of service-oriented software applications. A Special Issue on Service Engineering of International Journal of Web Services Research 4(2), 60–80 (2007)
8. Chen, T.Y., Feng, J., Tse, T.H.: Metamorphic testing of programs on partial differential equations: a case study. In: Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC), pp. 327–333. IEEE Computer Society Press, Los Alamitos (2002)
9. Chen, T.Y., Huang, D., Tse, T.H., Zhou, Z.Q.: Case studies on the selection of useful relations in metamorphic testing. In: Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC), Polytechnic University of Madrid, pp. 569–583. Polytechnic University of Madrid (2004)
10. Chen, T.Y., Tse, T.H., Zhou, Z.Q.: Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 191–195. ACM Press, New York (2002)
11. Chen, T.Y., Tse, T.H., Zhou, Z.Q.: Fault-based testing without the need of oracles. Information and Software Technology 45(2), 1–9 (2003)
12. Gotlieb, A.: Exploiting symmetries to test programs. In: Proceedings of the 14th International Symposium on Software Reliability Engineering, ISSRE (2003)
13. Chan, W.K., Chen, T.Y., Lu, H., Tse, T.H., Yau, S.S.: Integration testing of context-sensitive middleware-based applications: a metamorphic approach. International Journal of Software Engineering and Knowledge Engineering 16(5), 677–703 (2006)
14. Garey, M.R., Johnson, D.S.: Computers and Interactibility: A Guide to the Theory of NP-Completeness. W.H. Freeman, New York (1979)
15. Cormen, T.H., Leisevsen, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge (1990)
16. Do, H., Rothermel, G., Kinneer, A.: Prioritizing JUnit Test Cases: An Empirical Assessment and Cost-Benefits Analysis. An International Journal Empirical Software Engineering 11(1), 33–70 (2006)
17. Agrawal, H., DeMillo, R.A., Hathaway, R., Hsu, W., Hsu, W., Krauser, E.W., Martin, R.J., Mathur, A.P., Spafford, E.H.: Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, USA (March 1989)

Appendix: Greedy Algorithm on the Key-Lock Problem

```
1  INPUT M, where M is a matrix with (x + 1) rows and (y + 1) columns.
2  O := [], an empty array to store the selected keys as GA's output
3  numO := 0, a variable to count the number of selected keys
4  WHILE (x > 1), DO BEGIN
5     maxOLocks := 0
6     bestRowIndex := 0
7     bestKeyID := 0
8     FOR i = 1 to y, DO BEGIN
9        nOLocks := 0
10       FOR j = 1 to y, DO BEGIN
11          IF M[i][j] = 1, THEN BEGIN
12             nOLocks := nOLocks + 1
13       END
14       IF nOLocks > maxOLocks, THEN BEGIN
15          maxOLocks := nOLocks
16          bestRowIndex := i
17          bestKeyID := M[bestRowIndex][y + 1]
18       END
19    END
20    arrOpenedLocks := []
21    FOR j = 1 to y, DO BEGIN
22       IF M[bestRowIndex][j] = 1 THEN
23          FOR i := 1 to x +1, DO BEGIN
24             append M[i][j] to arrOpenedLocks
25       END
26    END
27    Remove arrOpenedLocks from matrix M
28    Remove M[bestRowIndex][] from matrix M
29    x := x − 1
30    y := y − maxOLocks
31    O[numO] := bestKeyID
32    numO = numO + 1
33 END
34 OUTPUT O
```