Fundamenta Informaticae XXI (2001) 1001–1021

DOI 10.3233/FI-2012-0000

IOS Press

The Incremental Maintenance of Transition Tour

Gábor Árpád Németh

High Speed Networks Laboratory,

Department of Telecommunications and Media Informatics,

Budapest University of Technology and Economics,

Magyar tudósok körútja 2, Budapest, H-1117, HUNGARY

gabor.nemeth@tmit.bme.hu

Zoltán Pap

High Speed Networks Laboratory,

Department of Telecommunications and Media Informatics,

Budapest University of Technology and Economics,

Magyar tudósok körútja 2, Budapest, H-1117, HUNGARY

pap@tmit.bme.hu

Abstract. While evolutionary development methodologies have become increasingly prevalent, incremental testing methods are lagging behind. Most traditional test generation algorithms – including the Transition Tour method – rebuild test sequences from scratch even if minimal changes to the system have been made. In the current paper we propose two incremental algorithms to update a Transition Tour test sequence after changes in a deterministic finite state machine model. Our solution uses existing information – the Eulerian graph of a previous version of the system and an Euler tour in it – to update the test cases of the system in response to modification. The first algorithm keeps an Eulerian graph up to date, while the second algorithm maintains an Euler tour in the augmented graph. Analytical and practical analyses show that our algorithms are very efficient in the case of changing specifications. We also demonstrate our methods through an example.

Keywords: finite state machine; test generation algorithms; incremental algorithms; incremental test generation.

1. Introduction

Although test generation methods based on finite state machine (FSM) have been studied extensively in the last decades [18, 4, 6] very little effort has been invested in dynamic scenarios involving changing system specifications. This is especially surprising considering that most recent system development models propose incremental approaches involving step-by-step refinement of the system under development [27, 13, 7]. From the testing perspective, evolutionary approaches open new possibilities to improve test generation methods. By identifying the effect of changes in each iteration step during the development, one may want to reuse as much as possible of the test set of the previous version of the system, and focus test generation only on specific parts of the system. This may result in significantly faster test generation in most iterative development scenarios.

In 2007 an efficient, bounded incremental algorithm was proposed [23] to maintain the test set of the HIS-method [24, 34] if an FSM specification is modified by a change. However, in large systems, the test suite of the HIS is typically too extensive, thus, its practical usage is limited. In a typical software testing framework, the much shorter test sequence of the Transition Tour [21] is also adequate.

In this paper we develop two novel bounded incremental algorithms to automatically modify the Transition Tour test sequence in response to changes to the system specification. It is shown that the time complexity of the proposed algorithm depends on the size of the modified part of the specification.

The rest of the paper is organized as follows. A short overview of our assumptions and notations is given in Section 2. The Chinese Postman Problem and the traditional approach to generate Transition Tours are also briefly discussed. Section 3 gives a quick overview of incremental test generation and incremental testing. In Section 4 we introduce the incremental algorithms for maintaining the Transition Tour across changes, demonstrate them through an example and provide analytic and practical analyses of their complexity. Finally, we conclude the paper with Section 5.

2. Preliminaries

2.1. Graphs

Let $G = (\mathcal{V}, \mathcal{E})$ be a labeled directed graph (possibly with loops and parallel edges), where $\mathcal{V} = \{s_1, \ldots, s_n\}$ denotes the set of nodes and $\mathcal{E} = \{e_1, \ldots, e_m\}$ denotes the set of directed edges. Let the number of nodes and edges be denoted by $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$, respectively.

A *directed path* in a directed graph G is a sequence of nodes such that from each node there is a directed edge in G to the next node in the sequence. A finite path always has a first node, called *start node*, and a last node, called *end node*. Let the directed path from start node s_y to end node s_x be denoted by $s_y \rightarrow \ldots \rightarrow s_x$.

A directed graph is *strongly connected*, iff a directed path exists from each node to every other node. The *strongly connected components* (SCCs) of graph G are the maximal strongly connected subgraphs of graph G. If an SCC consists of a single node, we call it a *singleton* SCC, otherwise it is a non-singleton SCC. A *directed acyclic graph* (DAG) is a directed graph that does not have directed cycles. A directed graph is DAG iff all of its SCCs are singletons.

Let the number of directed edges originating from node s_j be denoted by $d^+(s_j)$, and the number of directed edges that lead to node s_j by $d^-(s_j)$. We say that node s_j is *balanced* iff $d^-(s_j) = d^+(s_j)$, otherwise unbalanced. We say that a directed graph is *Eulerian*, if it is strongly connected and balanced for every node.

2.2. Finite State Machines

Finite State Machines (FSMs) have been widely used for decades to model systems in various areas, such as sequential circuits [11], communication protocols [14], some types of programs [1] (in lexical analysis, pattern matching, etc.) and object-oriented software testing [2]. Several specification languages, such as SDL [15] and ESTELLE [31], are extensions of the FSM formalism.

An FSM M is a quadruple $M = (I, O, S, T)$ where I, O, and S are the finite and nonempty sets of input symbols, output symbols and states, respectively. $T \subseteq S \times I \times O \times S$ is the finite set of transitions between states. Each transition $t \in T$ is a quadruple $t = (s_j, i, o, s_k)$, where $s_j \in S$ is the start state, $i \in I$ is an input symbol, $o \in O$ is an output symbol and $s_k \in S$ is the next state. FSM M is *deterministic* if for all (s_j, i) state–input pairs there exists at most one transition in T.

An FSM can be represented by a state transition graph, a directed edge-labeled graph whose nodes correspond to the states of the machine and whose edges to the state transitions. Each edge is labeled with a pair of input and output symbols, which is associated with the corresponding transition.

In the description of our algorithms, we use the graph terminology because it describes a more general approach. However, it is important to emphasize that the purpose of our algorithms is to maintain a test sequence of a system, whose description is given in the form of an FSM.

In the rest of the paper we consider strongly connected, deterministic FSMs.

2.3. Representing changes to FSMs

In order to develop incremental algorithms for FSM testing we need to discuss how to represent changes to FSMs.

We consider the following atomic graph operators:

- adding or removing an edge,
- adding or removing an isolated node,
- adding or removing the label associated with an edge.

The changes of the FSMs can be described by FSM fault models – see [5] and its extension [3]. Based on the fault model of [3], we consider the following FSM fault operators, which can be derived from applying atomic graph operators consecutively:

- Adding or removing a transition corresponds to adding or removing an edge and the label associated with it. To add a label to a new transition we add a pair of an input symbol $i \in I$ and an output symbol $o \in O$, so that the input symbol can be any symbol that is not used in any transition originating from the state the new transition originates from. This assumption is necessary to preserve the deterministic property of the FSM.
- Removing a state corresponds to removing an isolated node and all the edges with their associated labels that originate from or lead to this node.
- Changing the output label of a transition corresponds to removing the label of the transition and adding a new label that has the same input symbol as but a different output symbol than the removed label.
- Changing the next state of a transition can be achieved by composing two previously discussed FSM fault operators: removing a transition and then adding a transition with a same start state and same input–output label.

- Adding a state with one incoming and one outgoing transition can be produced by the composition of adding a state, adding a transition that originates from this state, and changing the next state of a transition so that it leads to the added state.

The changing of the output label of a transition and the changing of the next state of a transition are frequently applied steps in changing system specifications, because they preserve the deterministic property of the FSM. Adding a state with one incoming and one outgoing transition is also a frequently applied step as it prevents the added state from becoming isolated. However, some changes described above may not preserve the strongly connected condition of the FSM. This is not a problem in our case, because our algorithm presented in Section 4.2 is able to show if the strongly connected condition does not hold after a series of changes is applied to the graph of specification FSM M.

In the following we use the graph representation of FSMs to describe our algorithms, thus we use the FSM fault operators discussed above in the graph representation form. In the following we assume that if an edge is added or removed then its edge label is also added or removed in the way described above for the case of adding a transition.

2.4. Euler tour

An *Euler tour* is a cycle that traverses every edge exactly once in a given graph. In directed graphs there exists an Euler tour iff the graph is strongly connected and every node of the graph is balanced, i.e., $\forall s_j : d^-(s_j) = d^+(s_j)$ [12].

There are two different representations of Euler-tours: edge-pairing representation and next-node representation [8]. In the *edge-pairing representation*, a list of ordered pairs of consecutive directed edges is created for each node. In the *next-node representation*, an order of the edges is defined for each node and the node is left after the l^{th} visit across the edge labeled with l. Both representations have a corresponding algorithm that finds an Euler tour, they are called end-pairing and next-node pairing algorithm, respectively [8].

We use the next-node representation in Section 4.2 to maintain an Euler tour across updates, so the next-node pairing algorithm [8] is shortly discussed:

- **STEP 1:** Select any node $s_r \in G_E$. Find a spanning tree T of Eulerian graph G_E, for which edges are directed toward the root s_r of T. Thus, every node s_k (except $k = r$) of this tree has one edge directed away from s_k.
- **STEP 2:** For any node s_k (except $k = r$) specify arbitrary order L_k of the edges directed away from s_k with the restriction that the edge of the spanning tree T is the last in the ordering. For the root s_r, specify an arbitrary order L_r of the edges directed away from s_r.
- **STEP 3:** Finally, the Euler tour of G_E can be found by starting from the node s_r and always traversing an unused edge that has the lowest sequential number in the ordered list of edges L_x of the corresponding node s_x. If the start node s_r is reached and all edges originating from node s_r have been traversed, then terminate.

2.5. The Chinese Postman Problem and the Transition Tour method

The objective of the Chinese Postman Problem (CPP) is to find a minimum cost walk that covers each edge of the given graph at least once and return to the start node. The resulting sequence is called the postman tour of the given graph. The original problem was addressed in [17] and different

variations derived from the original CPP and their solutions are summarized in [9]. There are several applications of the CPP. It can be used to optimize routing problems, such as mail delivery, snow removal or garbage collection, where streets of a given city correspond to edges of the graph [9].

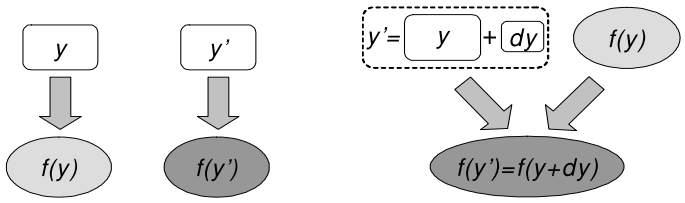
A special case of the CPP is the Directed Chinese Postman Problem (DCPP), where each edge of the graph is directed. The DCPP is used in the field of testing [4, 29, 32] to generate an optimal test sequence that traverses every transition of the specification machine at least once. With unit cost for each edge this is known as the Transition Tour (TT) method [21]. The TT is the shortest possible test sequence, which provides 100 percent transition coverage and 100 percent state coverage of the specification FSM [33].

The algorithms that solve the DCPP [8, 22, 20] have two main parts: (1) Augmenting the original graph G by duplicating some edges to make the graph Eulerian. (2) Creating an Euler tour of the Eulerian graph G_E. The resulting Euler tour of the augmented graph G_E is the postman tour of the original graph G.

There are several algorithms that produce an optimal Eulerian graph (i.e. the total cost of added augmenting edges is minimal). The algorithm presented in [8, 22] reduces the original problem to a minimum cost network flow problem or an equivalent transportation problem and has $O(n^3)$ time complexity. The algorithm presented in [20] also uses a similar procedure with time complexity $O(k \cdot n^2)$, where "constant k is closely related to the structure of the network. For some problem instances, such as the sparse network, k could be much smaller than m and n" [20]. However, complexity calculations of these solutions assume that the graph is simple, i.e. the graph does not have parallel and loop edges. If this assumption does not hold – a graph of an FSM may contain loops and parallel edges – we cannot have an $m < n^2$ upper bound for the number of edges. Thus, in the worst case, the algorithm presented in [20] requires $(m - n + 1)(3 \cdot n^2/2 + 3 \cdot m)$ steps of computation [20], i.e., its complexity is $O(m \cdot (n^2 + m))$. For the same reason the solutions presented in [8] and [22] have the complexity of $O(n^3 + m)$.

After graph G has been augmented to an Eulerian graph G_E, an Euler tour in G_E can be found. Let x denote the number of augmented edges used by the previous algorithm that transformed graph G to an Eulerian graph G_E. An Euler tour then can be created with a complexity of $O(m + x)$ with the next-node pairing algorithm discussed earlier in Section 2.4.

3. Incremental Test Generation and Incremental Testing



(a) Batch algorithm (b) Incremental algorithm

Figure 1. Batch and incremental algorithms

A *batch algorithm* for a given problem is an algorithm that computes the solution $f(y)$ of the problem on some input y without assuming any previous results; if input y has been changed into y', it generates the new output $f(y')$ from scratch – see Figure 1(a). Virtually, all FSM-based test generation algorithms [18, 4, 6] are batch algorithms. The input of these methods is the system specification

as an FSM model and the output consists of test sequences for the given specification. An *incremental algorithm* assumes that the same problem has been solved already on a slightly different input. It uses previous input y, along with the changes dy and the previous output $f(y)$ to generate the new output $f(y') = f(y + dy)$ – see Figure 1(b). In the case of test generation, an incremental algorithm would utilize existing test sequences to generate new test sequences for the updated system version. As a result, it may generate the solution for the given problem significantly faster than a batch algorithm.

3.1. Related Work

An incremental approach in the field of testing has been first introduced by El-Fakih et al. [10]. The method generates test cases only for the modified parts of the system, but it is not capable of maintaining a complete test set across changes. The method of El-Fakih et al. [10] presumes that the parts of the implementation which correspond to the unmodified parts of the specification have not been changed accidentally or intentionally.

An interesting aspect of incremental test generation for evolving systems is when only a part of the given specification is implemented. Jääskeläinen [16] presents a solution in which the unimplemented parts of the specification can be filtered out to support incremental testing: the remaining parts can be used to generate test cases for the implemented portion of the product. Simão and Petrenko [25] investigate the case how a predefined test set could be incrementally extended until the desired fault coverage is achieved, but they do not deal with evolving, changing specifications.

In [28] a framework has been developed to analyze the effects of changes on the test sequences by identifying the consistency between the test cases and the specification machine. Following this work, an incremental algorithm was proposed in [23] for keeping a complete test set up-to-date after an FSM specification has been modified by a change, if the original test cases are given in the form of the HIS-method [24, 34]. The total complexity of the solution introduced in [23] is $O(p \cdot n \cdot \Delta)$, where $p = |I|$ denotes the number of inputs and $\Delta \leq n^2$ denotes the number of test sequences that have to be modified, i.e., the number of test cases to be generated in worst case is equivalent to those generated by a batch algorithm with $O(p \cdot n^3)$ complexity.

Although the solution discussed in [23] makes the HIS-method much more efficient in case of changing specifications because of the structure of the test cases, both the test generation and the test execution still require a huge amount of time. For complex systems the test set would be simply unacceptably large to execute. Thus, a test designer would rather select a less exhaustive coverage criteria such as state and transition coverage [2, 26], which is also adequate in the typical software testing framework [2]. The TT covers all of the transitions and states of the specification FSM and its test execution requires only $O(m + x)$ complexity. The generation of a TT test sequence however requires $O(n^3 + m + x)$ or $O(m \cdot (n^2 + m) + x)$ complexity depending on the method used, either [8], [22] or [20], as described in Section 2.5. In the case of software testing [1], when the new code is compiled and tested only few times, test generation requires much more time than execution. Thus, more efficient test generation is required. In Section 4 we propose incremental algorithms, which utilize existing information of the previous system version and generate the test sequence for the new system much more efficiently than its counterparts. Moreover, these algorithms require fewer assumptions about FSM specifications and they can handle multiple and many more types of changes than the one proposed in [23].

4. Incremental Maintenance of Transition Tour

Similarly to previous approaches solving the Directed Chinese Postman Problem (DCPP), our solution – which keeps the TT test sequence up to date across changes – contains two parts: Keep Graph Eulerian (KGE, presented in Section 4.1) that maintains a graph to keep it Eulerian after changes, and Keep Euler Tour (KET, presented in Section 4.2) that maintains the Euler tour in this updated Eulerian graph.

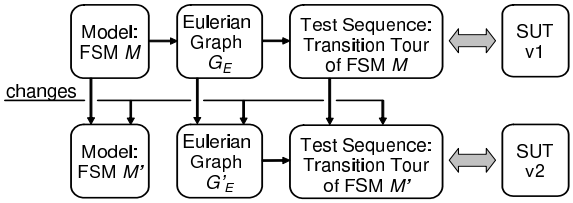


Figure 2. The block diagram of the incremental maintenance of Transition Tour

The high-level view of our approach is presented in Figure 2. The upper part of the figure shows the traditional generation of the TT test sequence: FSM M is transformed into an Eulerian graph G_E and an Euler tour in G_E is generated. The Euler tour of graph G_E corresponds to the TT test sequence of FSM M, which can be applied for the system under test (SUT). The lower part of the figure shows how the sequence of changes affects the generation of TT. The changes turn FSM M into FSM M'. Instead of repeating the previous operations on M', our KGE algorithm (proposed in Section 4.1) maintains graph G_E across the given sequence of changes to keep it Eulerian; the resulting graph is denoted by G'_E. The KET algorithm (introduced in Section 4.2) maintains the Euler tour of G'_E, which corresponds to the TT sequence of the modified FSM M'. Then this TT test sequence can be applied to the modified system (SUT v2).

4.1. An algorithm to keep the graph Eulerian

In this section we propose an algorithm called Keep Graph Eulerian (KGE) to keep graph G'_E Eulerian through changes if G_E had been Eulerian. An edge of the original graph G or G' is called an *original edge*. An edge is called an *extra edge* if it is a duplication of an edge of the original graph to be transformed in an augmented Eulerian graph G_E or G'_E. In the rest of this paper we denote the number of extra edges in the augmented graph by x.

Our algorithm consists of two main logical units, which solve two different subproblems:

1. A procedure that balances a given pair of nodes which have become unbalanced.
2. A main task that determines how the different types of changes turn the nodes of graph G_E unbalanced, and uses the procedure above to balance them.

First, a quick overview of the KGE algorithm is given along with a demonstration through an example. Detailed solutions for Subproblems 1 and 2 are presented in Sections 4.1.1 and 4.1.2, respectively.

In a nutshell, our KGE algorithm creates new *virtual extra edges* between nodes, which have become unbalanced after the changes. The KGE algorithm substitutes these virtual extra edges into the sequence of "real" extra edges – i.e. it duplicates edges that are present in the original graph. First, the path between two unbalanced nodes is found by breadth-first search, the edges of the path are

called *on-path edges*. After that, the algorithm checks whether the new on-path edges produce cycles with other extra edges. If they do, then the algorithm deletes these cycles, i.e. eliminates their edges[1]. To determine and delete the cycles consisting of extra edges, the KGE algorithm identifies the SCCs of the graph made up of extra edges using Tarjan's algorithm [30].

When the KGE algorithm terminates, all unbalanced nodes have become balanced[2]. The resulting graph G'_E will become Eulerian and the subgraph on the extra edges is again acyclic.

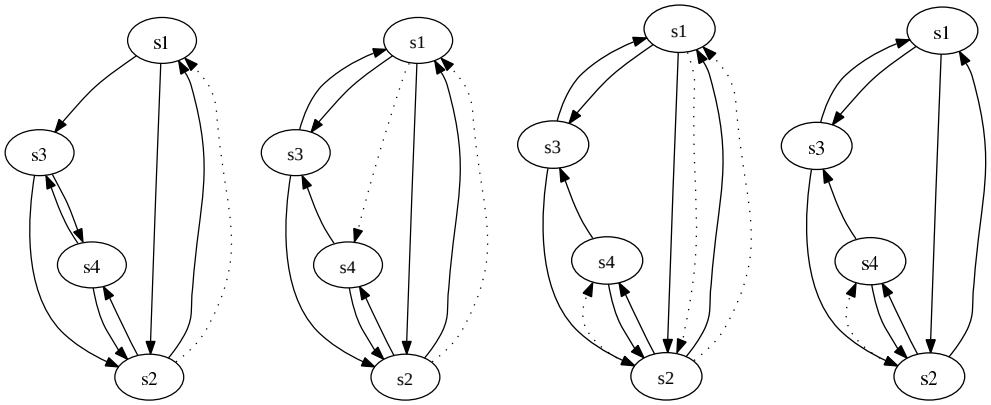


Figure 3. KGE algorithm example

KGE algorithm example: Here we demonstrate how our KGE algorithm keeps the graph balanced after a change has been made. We use the following notations in the figures: Continuous lines represent original edges (i.e. the transitions of the FSM). Extra edges, which make the graph balanced, are shown with dotted lines.

We have an Eulerian graph G_E of FSM M – see Figure 3(a) – and an edge change $s_3 \to s_4 \Rightarrow s_3 \to s_1$ that turns it into G'_E of FSM M'.

The KGE algorithm creates a virtual extra edge that originates from the new destination node (s_1, where $d^+(s_1) < d^-(s_1)$) and leads to the old destination node (s_4, where $d^+(s_4) > d^-(s_4)$) of the changed edge – see Figure 3(b). Then it runs a breadth-first search in order to substitute the virtual extra edge. The $s_1 \to s_2 \to s_4$ path is selected – see Figure 3(c). In the next step, the algorithm identifies the SCCs consisting of extra edges. $SCC_1 = \{s_1, s_2\}$, $SCC_2 = \{s_3\}$, $SCC_3 = \{s_4\}$ components were found. Only SCC_1 contains more than one node. Thus, the algorithm runs a depth-first search in SCC_1 of extra edges from the last node s_2 of the component's path-section in order to find cycles made up of extra edges. The $s_2 \to s_1$ edge is found, which forms a cycle with the $s_1 \to s_2$ on-path edge, making the algorithm delete this cycle. As no more extra edges remain in SCC_1, the algorithm terminates – the result is shown in Figure 3(d).

[1]The reason why the cycle can be eliminated is that each extra edge has a corresponding original edge. Thus each edge of the original graph will be covered by the Euler tour of G'_E and the length of this tour can be reduced with the elimination of cycles formed by extra edges.

[2]If it is not possible to balance all unbalanced nodes (because the given sequence of changes broke the strongly connected condition), this is also highlighted by the algorithm.

4.1.1. Balancing unbalanced nodes

In this section we discuss the procedures that substitute the virtual extra edge between two unbalanced nodes and eliminate cycles consisting of extra edges in more detail.

Procedure A: Substituting the virtual edge $s_y \rightarrow s_x$:

- **STEP 1:** Create a new virtual extra edge from s_y to s_x.
- **STEP 2:** Make a depth-first search on extra edges from s_x to s_y to check if the new virtual extra edge forms a cycle with other extra edges.
  - **STEP 2.1:** If **Yes:** Delete the whole cycle. Then the procedure terminates.
  - **STEP 2.2:** If **No:** Substitute the virtual extra edge step by step:
    - **STEP 2.2.1:** Run a breadth-first search from s_y to s_x on the original edges of the graph. Select a shortest path from s_y to s_x, and create extra edges – referred to as *on-path edges* – for each edge along this $s_y \rightarrow \ldots \rightarrow s_x$ path. If such path does not exists, then the algorithm shown that the given changes have broken the strongly connected property of the graph and the algorithm terminates.
    - **STEP 2.2.2:** Identify the SCCs of the graph of extra edges using Tarjan's algorithm [30].
    - **STEP 2.2.3:** In each non-singleton SCC eliminate cycles with a depth-first search on extra edges except on-path edges using procedure B to be defined bellow.

Procedure B: Eliminating cycles in SCC_C:

- **STEP 1:** Mark all nodes that lie on on-path edges – referred to as *on-path nodes* – in SCC_C with C.
- **STEP 2:** Let the consecutive on-path edges in SCC_C found by procedure A be denoted by p_C. Get the last C node s_b of p_C while it is to be found; if no more C nodes remain, the procedure terminates.
  - **STEP 2.1:** Run a depth-first search from s_b on extra edges of SCC_C that have not yet been visited, except for on-path edges p_C.
    - **STEP 2.1.1:** If a path $q_{b\rightarrow a}$ from node s_b to an on-path node s_a marked with C is found, then the edges of $q_{b\rightarrow a}$ create a cycle with consecutive on-path edges $p_{a\rightarrow b}$. Thus, delete the edges of $q_{b\rightarrow a}$ and $p_{a\rightarrow b}$. Remove the C marking for all on-path nodes of $p_{a\rightarrow b}$ except for the first on-path node s_a. If only one node remains in C (the first on-path node in SCC_C) then remove its C marking.
    - **STEP 2.1.2:** Otherwise, all investigated edges are assigned a $visited$ marker. If all edges reachable from the C node s_a are marked as $visited$, then remove the C marking from node s_a.

In order to show the correctness of the KGE algorithm, we prove the following two claims.

Lemma 4.1. After STEP 2.2.1 of procedure A, a non-singleton SCC of extra edges contains consecutive subset of on-path edges.

Proof:
The extra edges formed a DAG before changes. The reason for this is that Chinese Postman Algorithms generate optimal solutions and we assume unit cost for each edge, thus extra edges cannot

form cycles (otherwise cycles consisting of extra edges can be deleted, which makes a better solution, thus the original solution was not optimal). Thus, each non-singleton SCC of extra edges has at least one on-path edge.

For the consecutive property, assume that this is not true, and the $v_1 \to \ldots \to u_1$ and $v_2 \to \ldots \to u_2$ ($v_1 \neq u_1 \neq v_2 \neq u_2$) sections of consecutive on-path edges, which do not overlap, are found in the same SCC, but the $u_1 \to \ldots \to v_2$ section is not. Let this SCC be denoted by SCC_X. There is a directed path from each node of SCC_X to any other node of SCC_X. Thus, there is a directed path from u_1 to any other node of SCC_X and v_2 is reachable from any other node of SCC_X. Conversely, from each node of the missing $u_1 \to \ldots \to v_2$ section any other node of SCC_X is accessible, and from SCC_X all nodes of the missing $u_1 \to \ldots \to v_2$ section are accessible. Thus, the $u_1 \to \ldots \to v_2$ section of consecutive on-path edges belongs to SCC_X. This contradicts the initial assumption. □

Lemma 4.2. After procedure A terminates, the graph of extra edges remains DAG.

Proof:
As the graph of extra edges had been a DAG before the changes took place, cycles can only be formed with new extra edges added by procedure A described above. If a virtual extra edge has formed a cycle with other extra edges it is eliminated in step 2.1 of procedure A. Otherwise procedure A substitutes the virtual extra edge into a sequence of "real" extra edges and identify SCCs of extra edges. The new extra edges that lie between different SCCs do not form cycles with other extra edges, otherwise they would belong to a non-singleton SCC. All new extra edges that used to belong to a non-singleton SCC have been deleted in the cycle elimination phase of procedure B or verified not to form a cycle with other extra edges. Thus, no cycles are left when procedure A terminates. □

When procedure A terminates, the subgraph on the extra edges remains acyclic. Our solution does not guarantee that it finds a minimal cost Eulerian graph, however our simulations in Section 4.4 show that in practice it is not far from the optimal solution produced by traditional, batch algorithms even in the case of extensive changes.

4.1.2. Handling changes

In the following we investigate how the different types of changes discussed in Section 2.3 unbalance the nodes of G_E and how we can handle this using the procedures described in Section 4.1.1.

The algorithm goes through the given sequence of changes and handles them in the following ways:

Adding an edge: If the added edge is a loop, i.e. its source and the destination nodes are the same, then the graph remains balanced, thus no further action is required. Otherwise, two nodes become unbalanced: the source and the destination of the new edge. Let these nodes be denoted by s_x and s_y, respectively. Using procedure A (described in Section 4.1.1) create a new virtual extra edge between these unbalanced nodes – from node s_y where $d^-(s_y) > d^+(s_y)$ to node s_x where $d^-(s_x) < d^+(s_x)$ and substitute it.

Removing an edge: If the removed edge is a loop, i.e. its source and the destination nodes are the same, then the graph remains balanced, thus no further action is required. Otherwise, two nodes are unbalanced after a given edge has been removed: the source and the destination of the removed

edge, denoted by s_y and s_x, respectively. Create a new virtual extra edge from node s_y to node s_x and then substitute it. If the removed edge has extra copies, remove them as well and create virtual extra edges for these extra copies[3]. If one or more nodes have become unreachable after the change (i.e. the strongly connected assumption does not hold anymore), then the algorithm discovers this when it substitutes the virtual extra edge $s_y \rightarrow s_x$ step by step: it visits every edge with breadth-first search from node s_y and there are still unreachable nodes.

Changing the output symbol in the label of an edge: The given change modifies only an edge label of the graph, but does not affect its structure; the graph remains Eulerian.

Changing the destination of an edge: After the destination of an edge is changed, two nodes become unbalanced: the old s_x and the new destination s_y of the changed edge. Create a new virtual extra edge from node s_y to node s_x and substitute it. If the changed edge has extra copies, change their destination also and create a virtual extra edge for each of these extra copies[3].

Adding a node with one incoming and one outgoing edge: A new node is added to the graph with a new edge originating from it, and the destination of an existing edge is changed so that it leads to the new node. Let the destination node of the new edge originating from the added node be denoted by s_y and the previous destination node of that edge leading now to the newly added node be denoted by s_x. If $s_y = s_x$, then the graph remains balanced, thus no further action is required. Otherwise, two nodes are unbalanced: s_y and s_x. Create a new virtual extra edge from s_y to s_x and substitute it.

Removing a node: After the selected node and all edges originating from or leading to it are deleted, the following method is applied to each removed edge:

- If an edge originating from the removed node, then the in-degree $d^-(s_j)$ of node s_j that the removed edge was leading to is decreased by one: $d^-(s_j) := d^-(s_j) - 1$.
- If an edge was leading to the removed node, then the out-degree $d^+(s_k)$ of node s_k that the removed edge originated from is decreased by one: $d^+(s_k) := d^+(s_k) - 1$.

For all unbalanced node-pairs one by one[4]:

- Create a new virtual extra edge from a node s_y where $d^-(s_y) > d^+(s_y)$ to a node s_x where $d^-(s_x) < d^+(s_x)$ and then substitute it.

The algorithm terminates when all nodes of the graph have become balanced.

4.2. An algorithm to maintain the Euler tour

In this section we propose an algorithm called Keep Euler Tour (KET) to keep an Euler tour update in an Eulerian graph. The input of the algorithm consists of (1) an Euler tour in graph G_E before the changes in next-node representation (see Section 2.4), (2) the list of changes and (3) an Eulerian graph G'_E generated by the KGE algorithm proposed in Section 4.1. As the Euler tour is given in next-node representation, its maintenance can be reduced to the maintenance of a spanning tree T (the edge of which are directed towards the root). Without loss of generality we assume that the spanning tree T comprises exclusively of original edges.

[3] In this case we do not need to run breadth-first search again as the shortest path from s_y to s_x was found previously.

[4] It is possible that after the change has been applied the graph remains balanced (for instance the removed node was an isolated node or for each node, where incoming edges have been removed the same number of outgoing edges have been also removed). In this case no virtual edge is added and substituted.

First, we introduce terms and notations used in the description of the algorithm. Let the ordered list of edges originating from node s_x be denoted by L_x and the ordered lists of edges for all nodes by $\mathcal{L} = \{L_1, \ldots, L_n\}$. We refer to the edges of the spanning tree T as *spanning edges*. We denote by MOD the following set of nodes: (i) nodes for which the directed path in the spanning tree are removed or changed during the update process, (ii) nodes which are added to the graph. A work set is also used to keep track of these nodes, for which the new spanning edge has not yet been generated – denoted by MOD_{Work}. As a result an edge is a *valid spanning edge* of node s_x (except for root node s_r) if it has the last sequential number in L_x, and s_x does not belong to the set MOD_{Work}. A subtree of the spanning tree T having node s_i as root will be referred to as T_{s_i}.

Our algorithm consists of three main logical units:

1. Procedures that describe frequently used steps that can be applied to update the Euler tour through edge changes.
2. A main task that determines how the different types of changes affect the next-node representation of the Euler tour and uses Procedures 1 to maintain it.
3. A task that constructs a subgraph for MOD_{work} nodes in order to create new spanning edges for them.

A quick overview of the KET algorithm is given along with a demonstration through an example, the details of the three units of the KET algorithm are described in Sections 4.2.1, 4.2.2 and 4.2.3, respectively.

The KET algorithm maintains for each node $s_k, k = 1, \ldots, n$ the sequential numbers in L_k so that the spanning edge remains the last one in the ordered list of edges L_k. If a spanning edge was removed, the algorithm tries to find a new spanning edge, which leads to a node, that has a valid spanning edge. If this is not possible, the node is put into the set MOD_{work}. Another procedure is introduced to maintain $\mathcal{L}$ when a virtual extra edge was substituted by the KGE algorithm. After the given sequence of changes have been handled, the KET algorithm creates spanning edges for MOD_{work} nodes.

When the KET algorithm terminates, the resulting $\mathcal{L}$ with root node s_r gives a valid Euler tour of G'_E in the next-node representation form.

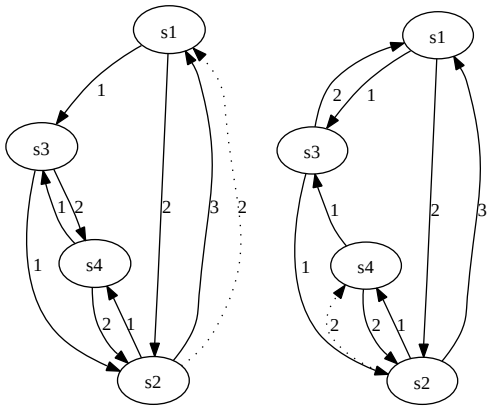


Figure 4. KET algorithm example

KET algorithm example: We demonstrate the KET algorithm with the example introduced in Section 4.1: edge change $s_3 \rightarrow s_4 \Rightarrow s_3 \rightarrow s_1$ turns Eulerian graph G_E of FSM M (Figure 4(a)) into G'_E of FSM M' (Figure 4(b)). Again, the original edges (i.e. the transitions of the FSM) are represented with continuous lines, and the extra edges – which make the graph balanced – are shown with dotted lines. The labels of edges represent the sequential numbers of edges in $\mathcal{L}$ used by the TT test sequence.

The KET algorithm maintains an Euler tour with the root s_1 and the spanning edges $\{s_2 \rightarrow s_1, s_3 \rightarrow s_4, s_4 \rightarrow s_2\}$ – see Figure 4(a) – as they have the last sequential numbers in L_2, L_3 and L_4, respectively. The changed $s_3 \rightarrow s_4$ edge was a spanning edge in the Eulerian graph G_E, so the starting node s_3 is put into the set MOD. The algorithm discovers that the $s_3 \rightarrow s_1$ edge leads to a non-MOD_{Work} node, so it marks $s_3 \rightarrow s_1$ as a spanning edge, i.e. this edge receives the last sequential number in L_3. When the virtual edge was substituted with the KGE algorithm, one extra edge was created and one extra edge was removed, which did not affect the spanning tree, but modified the order of edges in L_2 – the final solution is shown in Figure 4(b). The TT test sequence of the modified FSM M' is started at root node s_1 of the spanning tree T and always uses an unused edge that has the lowest sequential number in the ordered list of edges, until s_1 is reached and all edges of node s_1 are traversed.

4.2.1. Maintaining the ordered list of edges

In this section a detailed description is given for the procedures that handle frequently used steps in the maintenance of the Euler tour given in next-node representation form.

After the graph is balanced with the KGE method proposed in Section 4.1 we need the following procedures to maintain the Euler tour of G'_E:

1. **Adding a new non-spanning edge to node** s_x: The new edge is added to the graph as follows: the order of old edges in L_x remains unchanged, except for the last edge, the spanning edge, which would be raised by one to remain the last in L_x. The new edge gets a sequential number in L_x that is lower by one than that of the spanning edge.
2. **Adding a spanning edge to node** s_x: The edge is added as the last element of L_x.
3. **Removing a non-spanning edge from node** s_x: If the removed edge was the last non-spanning edge in L_x, then the sequential number of the spanning edge in L_x is decreased by one. Otherwise, the last non-spanning edge gets the sequential number of the removed edge in L_x and the sequential number of the spanning edge is decreased by one in L_x.
4. **Removing a spanning edge from node** s_x: The algorithm adds all nodes of the subtree T_{s_x} to the sets MOD and MOD_{Work}. Then it checks whether there is another edge originating from node s_x that leads to a non-MOD_{Work} node.
   - If **Yes:** The edge is set as a new spanning edge, i.e. it gets a last sequential number in L_x. All nodes of the previously used subtree T_{s_x} are removed from the set MOD_{Work}.
   - If **No:** The algorithm creates new spanning edges for MOD_{Work} nodes after all changes have been applied to the pattern that is described in Section 4.2.3.
5. **Virtual edge substitution handling:** As a virtual extra edge was substituted by the KGE algorithm, two cases could have occurred: extra edges (on-path edges) were added or some extra edges (that formed a cycle with on-path edges) were removed. As the spanning tree contains only original edges, the algorithm maintains the sequential numbers in $\mathcal{L}$ using procedure 1 and 3.

4.2.2. Handling changes

In this section the main task is described in detail which determines, how the different types of changes introduced in Section 2.3 affect the next-node representation of the Euler tour and how they can be handled with the procedures discussed in Section 4.2.1.

The KET algorithm goes through the given sequence of changes and handles them in the following ways:

Adding an edge: We check whether the new edge is originating from a MOD_{Work} node – a node which does not have a valid spanning edge[5] – and is leading to a node that does not belong to the set MOD_{Work}. If it does, the new edge is marked as a spanning edge using procedure 2 and the source node of this edge is removed from MOD_{Work}. Otherwise, the new edge is added using procedure 1. If a virtual extra edge has been substituted by the KGE algorithm, it must be handled using procedure 5.

Removing an edge: Check, whether the removed edge was a spanning edge and then the given edge is removed using procedure 3 or 4. If a virtual extra edge has been substituted by the KGE algorithm, it must be handled using procedure 5.

Changing the output symbol in the label of an edge: No action is required.

Changing the destination of an edge: Check whether the changed edge originating from node s_x was a spanning edge.

- If it was, and it still leads to a non-MOD_{Work} node, it remains a spanning edge; nodes of subtree T_{s_x} are added only to the set MOD.
- If the changed edge was a spanning edge, and after the change it leads to a MOD_{Work} node, then procedure 4 is used.
- Otherwise, do nothing.

As a virtual extra edge has been substituted by the KGE algorithm, it must be handled using procedure 5.

Adding a node with one incoming and one outgoing edge: We add the new node to the set MOD. The edge that originated from the new node is a new spanning edge if this edge leads to a node which is not in the set MOD_{Work}. If this edge leads to a node that belongs to the set MOD_{Work}, then the new node is added to the set MOD_{Work}, too. For the case when the destination of an edge is changed in order to lead to the new node apply the case that handles the change of the destination of an edge. If a virtual extra edge has been substituted by the KGE algorithm, it must be handled using procedure 5.

Removing a node: Check whether there are any nodes that have a spanning edge leading to the removed node. Let s_i denote these nodes.

- If **No:** Do nothing.
- If **Yes:** We add these s_i nodes and all nodes in subtrees T_{s_i} to the sets MOD and MOD_{Work}. For all i we check whether there is another edge originating from s_i to a non-MOD_{Work} node.
  - If **Yes:** We set this edge as a spanning edge, i.e. this edge gets the last sequential number in L_i. The investigated node s_i and other nodes in T_{s_i} are removed from the set MOD_{Work}.

[5] This case may be realized if a spanning edge has been removed in a previous step.

– If **No:** The algorithm creates new spanning edges for MOD_{Work} nodes after all changes have been applied to the pattern described in Section 4.2.3.

If a virtual extra edge has been substituted by the KGE algorithm, it must be handled using procedure 5.

4.2.3. Creating new spanning edges

In this section, we discuss the task in detail which creates new spanning edges for MOD_{Work} nodes.

After all changes have been applied, we construct a subgraph of G'_E denoted as R in the following way: (I) For each MOD_{Work} node there is a corresponding node in R. (II) For each original edge between MOD_{Work} nodes there is an edge in R. (III) If there is an edge in G'_E originating from a MOD_{Work} node leading to a non-MOD_{Work} node, the given MOD_{Work} node in R is marked with the corresponding edge. We select the marked nodes, set the marked edges as new spanning edges and remove the marked nodes from MOD_{Work}. Then we run a breadth-first search in R from the first marked node using the opposite edge directions. If we reach a MOD_{Work} node, we remove it from MOD_{Work} and set the edge that was used to reach that node as a new spanning edge using procedure 2. We continue the breadth-first search using the opposite edge directions until the set of MOD_{Work} nodes becomes empty or until every edge in R (that can be traversed from marked nodes using breadth-first search) is traversed.

When the KET algorithm terminates, the nodes that belong to the set MOD_{Work} are unreachable nodes from the root of the spanning tree T. Thus, the algorithm is able to show if the strongly connected condition do not hold after a series of changes is applied to the augmented graph G_E of FSM M.

4.3. Calculating complexity

Let the number of nodes and edges in the original – non-augmented – graph be denoted by n and m, respectively. Let the number of extra edges in the augmented graph be denoted by x.

4.3.1. Complexity calculation of the KGE algorithm

First, we discuss the complexity of procedures described in Section 4.1.1. To create a virtual extra edge from s_y to s_x requires $O(1)$ step. To make a depth-first search on extra edges and to check if the virtual extra edge forms a cycle with other extra edges both have complexity of $O(x)$. The breadth-first search from s_y has complexity $O(m+n)$. Tarjan's algorithm [30] identifies the SCCs of extra edges in $O(n+x)$ complexity. To run depth-first search in each SCC and to delete cycles formed by extra edges requires $O(n+x)$ steps. Thus, the total complexity of virtual edge substitution described in Section 4.1.1 is $O(m+n+x) = O(m+x)$.

Let us investigate the complexity of the KGE algorithm across different type of changes (see Section 4.1.2). Adding an edge or adding a node with one incoming and one outgoing edge requires $O(m+x)$ steps as at most one virtual extra edge substitution is required. If an edge is removed or the destination of an edge is changed and no extra copies of the changed edge were present then $O(m+x)$ steps are required for the reason as previously. If the changed edge has extra copies, the algorithm creates and substitutes a virtual extra edge for each extra copy of the changed edge, but in this case no breadth-first search is performed during virtual edge substitution (see footnote 3 of Section 4.1.2). Thus, the complexity is $O(m + p_1 \cdot (n+x))$, where p_1 denotes the number of modified edges

(including the original edge and its extra edges). In case a node is removed, the algorithm creates at most p_2 virtual extra edges, where p_2 denotes the sum of original and extra edges leading to the removed node from other nodes. Hence the complexity of removing a node is $O(p_2 \cdot (m + x))$. Changing the output symbol in the label of an edge requires $O(1)$ step, because this change does not affect the structure of the graph.

Thus, the total complexity of the KGE algorithm is $O(p \cdot (m + x))$, where p denotes the number of changed edges in G_E.

4.3.2. Complexity calculation of the KET algorithm

Like in the previous section, we begin discussing the complexity of procedures described in Section 4.2.1. Procedures 1-3 have $O(1)$ complexity for obvious reasons. Procedure 4 has $O(n)$ complexity, because if a spanning edge was removed from a node, $n - 1$ nodes were added to the sets MOD and MOD_{Work} in the worst case. The complexity calculation for procedure 5 is as follows: When a virtual edge is substituted by the KGE algorithm, at most $n - 1$ extra edges are added or removed, so procedure 5 updates the sequential numbers in $\mathcal{L}$ for at most $n - 1$ edges, each one with $O(1)$ complexity using procedure 1 or 3. Thus, procedure 5 has $O(n)$ complexity in the worst case.

The complexity of handling changes depends on the type of the change (see Section 4.2.2). Changing the output symbol in the label of an edge requires $O(1)$ step for the same reason as discussed in Section 4.3.1. Adding an edge or adding a node with one incoming and one outgoing edge requires $O(n)$ steps because of procedure 5. If an edge was removed or the destination of an edge was changed and no extra copies of the changed edge were present, $O(n)$ steps are required because of procedures 4 and 5. If the changed edge has $p_1 - 1$ extra copies, the first algorithm has also created and substituted a virtual extra edge for each of them, so the complexity of this case is $O(p_1 \cdot n)$. When a node was removed, the first algorithm created and substituted p_2 virtual extra edges, each of them must be handled with procedure 5 with $O(n)$ sets, and a maximum of $n - 1$ nodes are added to the sets MOD and MOD_{Work}, thus the complexity of this case is $O(p_2 \cdot n)$.

Finally, the KET algorithm creating a subgraph R of G'_E for MOD_{Work} nodes as discussed Section 4.2.3, which requires $O(\Delta)$ complexity, where $\Delta < m$ denotes the number of edges in R.

Thus, the total complexity of the KET algorithm which maintains an Euler tour of G'_E is $O(\Delta + p \cdot n)$, where p denotes the number of changed edges in G_E and $0 \leq \Delta < m$.

4.3.3. Overall Complexity

The total complexity of the KGE and KET algorithms that incrementally maintain a Transition Tour test sequence is therefore $O(p \cdot (m + x))$ in the worst case, where p denotes the number of changed edges in G_E.

4.4. Simulation results

We implemented the traditional TT-method and our algorithms as well in C++ using the LEMON [19] library. Our incremental algorithms utilize the auxiliary information generated initially by batch algorithms – as shown previously in Figure 2. To be able to compare batch and incremental algorithms for the same system version, test cases for the modified machine M' were generated with the batch method as well. For the simulations we used operating system Linux Gentoo 3.4.6, which

runs on an AMD Opteron 246 processor with 1993 MHz clock frequency, 1024 KB cache and with 4 GB memory.

We generated random FSMs to investigate the practical performance of the traditional and the incremental TT-method. The random FSM generator ensures the strongly connected property in the following way: A directed cycle is created that traverses each state of the FSM, then transitions are added randomly until the predefined number of transitions has been attained.

We investigated the case when transitions were added to or removed from the original FSM, because these types of changes can be easily handled, and it gives representative results. Each data point in the figures had been obtained by averaging 10 simulation runs.

In the following we analyze how our algorithms perform against the traditional TT-method. We investigate two different scenarios: In Scenario 1 each state of the FSM before changes – denoted with M – has 5 transitions in average, and in Scenario 2 each state of the FSM M has 25 transitions on average.

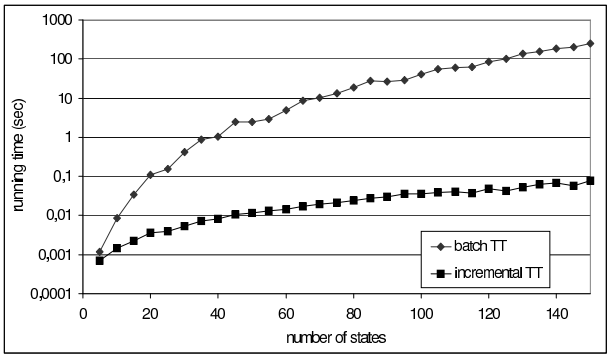


Figure 5. Running times of incremental TT and batch TT in Scenario 1

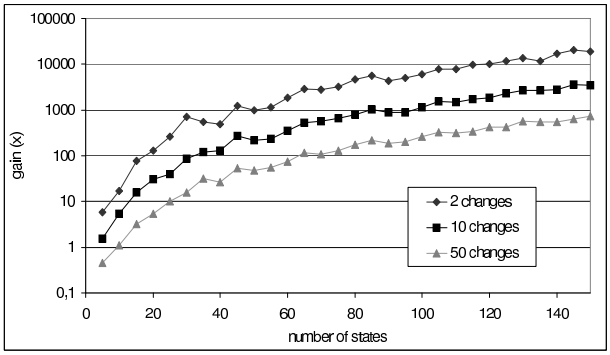


Figure 6. The gain of the incremental TT compared to the batch TT in Scenario 1

First, consider Scenario 1. Figure 5 shows the running time of the traditional TT-method and the incremental TT-method in the case of 10 changes. Instead of investigating running times, it is more rewarding to analyze how the running times of the traditional and the incremental TT-method correlate. We define the variable $Gain$, which describes the gain of our incremental TT-method compared to the traditional TT-method and it is calculated as follows: $Gain = \frac{t_{trad_{TT}}}{t_{inc_{TT}}}$, where $t_{trad_{TT}}$ is the

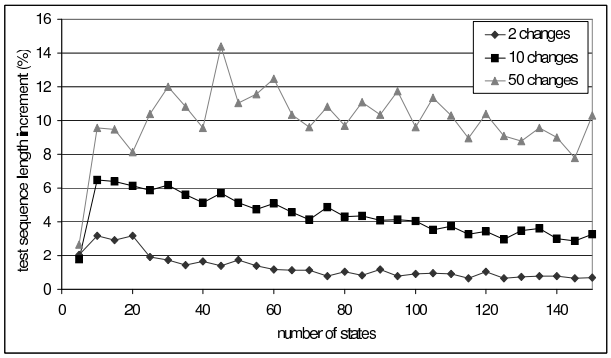


Figure 7. The difference in the number of transition traversals compared to the optimal solution in Scenario 1

running time of the traditional TT-method, and $t_{inc_{TT}}$ is the running time of our incremental TT-method measured in seconds. Thus, the higher the $Gain$ value, the better the performance as compared to the traditional approach. For instance $Gain = 100$ means that our incremental TT-method generates the test sequence a hundred times faster than the traditional TT-method. The results for 2, 10 and 50 changes are shown in Figure 6. The cost for this high gain is that our KGE algorithm does not guarantee an optimal solution, however our simulations in Figure 7 indicate that the difference from the optimal solution is not significant. An another important aspect is that the difference of the length of the test sequence compared to the optimal solution does not decline against the number of states.

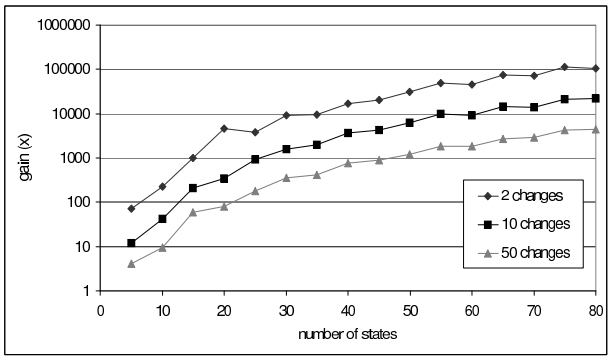


Figure 8. The gain of the incremental TT compared to the batch TT in Scenario 2

We also investigate Scenario 2, when the given FSM M is denser; 25 transitions on average are set for each state of M. Figure 8 shows the gain of our incremental algorithm compared to the traditional one. Figure 9 depicts the difference of the number of transition traversals compared to the optimal solution. We found that our algorithm is even more efficient compared to the traditional one than in the case of sparse graphs and the generated solution is even closer to the optimal one.

We also investigate how the number of changes affect the increase in the length of the test sequence compared to the traditional TT-method. The number of states was fixed at 50, and each state of the machine has 5 transitions on average as in the case of Scenario 1. Each data point in the figure was obtained by averaging 50 simulation runs. The results are presented in Figure 10. We observe that even in the case of thousands of edge changes, the difference from the optimal solution is typically

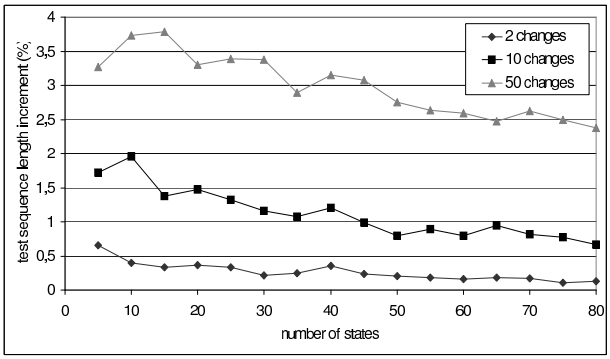


Figure 9. The difference in the number of transition traversals compared to the optimal solution in Scenario 2

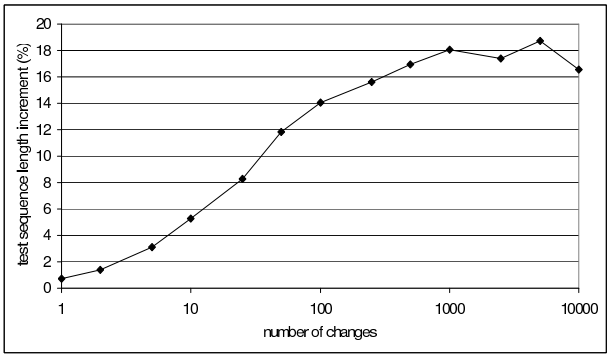


Figure 10. The difference from the optimal solution against the number of changes

less than 20 percent, which is an acceptable cost in the light of the increased performance in test generation.

The results show that our solution is not only a theoretical approach, but it is far more efficient than the traditional TT-method in case of changing specifications. The algorithms require much less time to keep the test sequence up-to-date than the TT-method and it performs even better as the size of the system increases. It has been shown, that the generated test case is not far from the optimal one even in the case of thousands of changes.

5. Conclusion

We have presented bounded incremental algorithms to maintain the Transition Tour test sequence for finite state machine models across updates. Nevertheless, our solution can be used in different routing problems too. Our approach assumes a changing specification and utilizes an existing test set of the previous version to keep a complete test set efficiently up-to-date. For each update of the system a complete test set is generated with the same fault detection capability as that of a traditional TT-method. Both analytical calculations and experiments indicate that our approach is very efficient in case of large, changing specifications. Therefore, our algorithms are serviceable in real testing environments – such as software testing – through development phase, because they

provide much faster test generation than the traditional approach, while the increment in the length of the test sequence remains acceptable. As a result, our approach assures a good balance between the very efficient generation of test sequence through updates and the nearly optimal length of this test sequence.

Acknowledgments

This work is connected to the scientific program of the "Development of quality-oriented and cooperative R+D+I strategy and functional model at BME" project. This project is supported by the New Hungary Development Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

References

[1] Ammann, P., Offutt, J.: *Introduction to Software Testing*, 1 edition, Cambridge University Press, New York, NY, USA, 2008.

[2] Binder, R. V.: *Testing object-oriented systems: models, patterns, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[3] Bochmann, G. v., Das, A., Dssouli, R., Dubuc, M., Ghedamsi, A., Luo, G.: Fault Models in Testing, *Proceedings of the IFIP TC6/WG6.1 Fourth International Workshop on Protocol Test Systems IV*, North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 1992.

[4] Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., (Eds.), A. P.: *Model-Based Testing of Reactive Systems*, Springer, 2005.

[5] Chow, T.: Testing software design modeled by finite-state machines, *IEEE Transactions on Software Engineering*, **4**(3), May 1978, 178–187.

[6] Dorofeeva, R., El-Fakih, K., Maag, S., Cavalli, A. R., Yevtushenko, N.: FSM-based conformance testing methods: A survey annotated with experimental evaluation, *Information and Software Technology*, **52**(12), 2010, 1286–1297.

[7] Dove, R.: *Response Ability - The Language, Structure and Culture of the Agile Enterprise*, John Wiley and Sons, 2001.

[8] Edmonds, J., Johnson, E. L.: Matching, Euler tours and the Chinese postman, *Mathematical Programming*, **5**(1), 1973, 88–124.

[9] Eiselt, H. A., Gendreau, M., Laporte, G.: Arc routing problems, part I: The Chinese postman problem, *Operations Research*, **43**(2), 1995, 231–242.

[10] El-Fakih, K., Yevtushenko, N., von Bochmann, G.: FSM-Based Incremental Conformance Testing Methods, *IEEE Transactions on Software Engineering*, **30**(7), 2004, 425–436.

[11] Friedman, A. D., Menon, P. R.: *Fault Detection in Digital Circuits*, Prentice-Hall, 1971.

[12] Gibbons, A.: *Algorithmic graph theory*, Cambridge University Press, 1985.

[13] Highsmith, J. A.: *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, Dorset House Publishing Co., Inc., New York, NY, USA, 2000.

[14] Holzmann, G. J.: *Design and Validation of Protocols*, Prentice-Hall, 1990.

[15] ITU-T: Recommendation Z.100: Specification and Description Language, 2000.

[16] Jääskeläinen, A.: Filtering test models to support incremental testing, *Proceedings of the 5th international academic and industrial conference on Testing - practice and research techniques*, TAIC PART'10, Springer-Verlag, Berlin, Heidelberg, 2010.

[17] Kwan, M.: Graphic programming using odd or even points, *Chinese Math*, (1), 1962, 273–277.

[18] Lee, D., Yannakakis, M.: Principles and Methods of Testing Finite State Machines – A Survey, *Proceedings of the IEEE*, **84**(8), 1996, 1090–1123.

[19] LEMON: *A C++ library for efficient modeling and optimization in networks*, Technical report, Avaliable: http://lemon.cs.elte.hu.

[20] Lin, Y., Zhao, Y.: A new algorithm for the directed chinese postman problem, *Computers & operations research*, **15**(6), 1988, 577–584.

[21] Naito, S., Tsunoyama, M.: Fault detection for sequential machines by transition-tours, *Proceedings of the 11th IEEE Fault-Tolerant Computing Conference (FTCS 1981)*, IEEE Computer Society Press, 1981.

[22] Orloff, S.: A Fundamental Problem in Vehicle Routing, *Networks*, **4**, 1974, 35–64.

[23] Pap, Z., Subramaniam, M., Kovács, G., Németh, G. A.: A Bounded Incremental Test Generation Algorithm for Finite State Machines, *19th IFIP Testing of Communicating Systems (TestCom/FATES)*, Tallinn, Estonia, 2007.

[24] Petrenko, A., Yevtushenko, N., Lebedev, A., Das, A.: Nondeterministic State Machines in Protocol Conformance Testing, *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI*, 1994.

[25] Simão, A., Petrenko, A.: Fault Coverage-Driven Incremental Test Generation, *Comput. J.*, **53**, November 2010, 1508–1522.

[26] Simao, A., Petrenko, A., Maldonado, J.: Comparing finite state machine test coverage criteria, *Software, IET*, **3**(2), april 2009, 91–105.

[27] Smith, P., Reinertsen, D.: *Developing Products in Half the Time: New Rules, New Tools*, John Wiley and Sons, 1998.

[28] Subramaniam, M., Pap, Z.: Analyzing the Impact of Protocol Changes on Tests, *Proceedings of the IFIP International Conference on Testing Communicating Systems, TestCom*, 2006.

[29] Takahashi, J., Kakuda, Y., Ohba, M.: Extended-model Based Testing by Using Directed Chinese Postman Problem, *Journal of Reliability Engineering Association of Japan*, **25**(3), 2003, 267–278.

[30] Tarjan, R.: Depth-first search and linear graph algorithms, *SIAM Journal of Computing*, **1**(2), jun 1972, 146–160.

[31] TC97/SC21, I.: Estelle – A Formal Description Technique Based on an Extended State Transition Model. International Standard 9074, 1988.

[32] Thimbleby, H.: The directed chinese postman problem, *Software Practice and Experience*, **33**, 2003, 1081–1096.

[33] Utting, M., Legeard, B.: *Practical Model-Based Testing: A Tools Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[34] Yannakakis, M., Lee, D.: Testing finite state machines: fault detection, *Selected papers of the 23rd annual ACM symposium on Theory of computing*, 1995.