A Graph Annotation Based Algorithm for Transducer Modification Inference

Gábor Guta

May 18, 2011

Abstract

Grammatical Inference is a new branch of (symbolic) learning algorithms. In this field most of the algorithms infer automata or transducers from a set of examples.

In this paper we propose an inference algorithm which infers the modification of an existing a transducer according to the examples of desired input/output pairs instead of inferring such a transducer from scratch. The paper evaluates the effectiveness of the algorithm by analyzing the inferred solutions of examples. The solutions of the algorithm are also compared to the results of a previously described inference algorithm. The comparison showed that the newly proposed algorithm behaved superior.

Contents

1 Introduction

A transducer is a finite state machine with output [1]. Transducers can be used to model aspects of various real life systems, e.g. model to text transformation languages. In this report we focus on finite state string transducers with two minor extensions: we allow transition rules which require no reading from the input tape, and the output can contain an arbitrary number of output symbols.

We study a special field of grammatical inference in which transducer modifications are inferred from example. We presented the basic concepts and the related literature in [3]. In that report we described an algorithm and discussed its possible improvements. Based on these ideas we present in this report an improved algorithm. We evaluate the improved algorithm and compare it with the first algorithm.

The algorithm infers a modified transducer from a set of example pairs and an original transducer. An example pair contains an input and the corresponding output which is the expected result of the execution of the modified transducer.

Technically, the only correctness requirement is that the algorithm must produce the expected output by executing the example input while the behavior related to the rest of the inputs is undefined. Informally we can formulate two further requirements: the modification on the transducer shall be as small as possible; and the behavior of the transducer shall be preserved with respect to the specified input as much as possible.

We can modify this transducer in infinitely many ways to produce the new output. One extreme solution is that we construct a transducer from scratch to accept the specified input and to produce the expected output, but the informal requirements provide us with guidelines that prevents this simple approach. To formalize the informal requirements, we have defined in [3] some metrics; that are to be minimized.

This report is structured as follows: in Section 2 we present the definitions used to describe the algorithm; in Section 3 we describe the algorithm and explain the idea behind it; in Section 4 we present extensions of the described algorithms; finally, in Section 5 we evaluate and compare the properties of the newly described algorithm to the algorithm described in [3].

2 Definitions

In this section we discuss the pseudo code notations and define the concepts used in this work. The algorithms are represented in a Python-like syntax.

The main difference between our pseudo language and Python is that we declare types to make easier to follow the data passed between the algorithms. Comments start with #. The notation has four kinds of data structures: record, sequence (list) and function (dictionary). Operators used over the data structures are defined as follows:

- Records are used to represent heterogeneous data structures.
  - They can be constructed by writing the name of the type and enumerating the values in the order of their field definitions enclosed in parentheses.
  - Values of the record can be accessed by the "dot and the name of the field" notation.
- Lists are used to enumerate ordered values with an equivalent type. They can be constructed by enumerating their elements. The signatures of the accesses functions are as follows:
  - list(_): name of T $\rightarrow$ T^* (empty list)
  - head(_): $T^* \rightarrow T$ (first element of the list)
  - tail(_): $T^* \rightarrow T^*$ (a list contains all but the first element)
  - last(_): $T^* \rightarrow T$ (last element of the list)
  - allButLast(_): $T^* \rightarrow T^*$ (a list contains all but the first element)
  - length(_): $T^* \rightarrow \mathbb{N}$ (the length)
  - _ + _ : (T^*, T^*) $\rightarrow T^*$ (concatenation of lists)
  - [_]: (T) $\rightarrow T^*$ (one element list)
- Functions are to represent mappings.
  - func(_ , _): (name of T_1, name of T_2) $\rightarrow$ ($T_1 \rightarrow T_2$) (empty function)
  - _ [_]= _ : ($T_1 \rightarrow T_2$, T_1, T_2) $\rightarrow$ ($T_1 \rightarrow T_2$) (define function value for a key)
  - _ [_]: (($T_1 \rightarrow T_2$, T_1) $\rightarrow T_2$ (the value belongs to a key)
  - keys(_): ($T_1 \rightarrow T_2$) $\rightarrow T_1^*$ (sequence of keys on which the function is interpreted)
- Directed multi graphs are used to represent the transition graphs.

- graph(_ , _ , _): (name of T_1, name of T_2, name of T_3, name of T_4) $\rightarrow$ Graph(T_1, T_2, T_3, T_4)
- addNode(_ , _): (Graph(T_1, T_2, T_3, T_4), T_1) $\rightarrow$ Graph(T_1, T_2, T_3, T_4)
- addNodesFrom(_ , _): (Graph(T_1, T_2, T_3, T_4), T_1^*) $\rightarrow$ Graph(T_1, T_2, T_3, T_4)
- addEdge(_ , _ , _ , _): (Graph(T_1, T_2, T_3, T_4), T_1, T_1, T_2) $\rightarrow$ Graph(T_1, T_2, T_3, T_4)
- getNode(_ , _): (Graph(T_1, T_2, T_3, T_4), T_1) $\rightarrow$ T_3
- getEdge(_ , _ , _ , _): (Graph(T_1, T_2, T_3, T_4), T_1, T_1, T_2) $\rightarrow$ T_4

In principle the definitions are the same as the ones that can be found in [3]. *Input*, *Output*, and *State* are types of un-interpreted symbols from which the input alphabet, output alphabet, and states formed, respectively.

- A *Transition Key* (or *Guard*) is a record of a source state and an input symbol.

  TransitionKey := (source: State, input: Input)

- A *Transition Data* element is a record of a target state and output symbols.

  TransitionData := (target: State, output*: Output)

- *Transition Rules* (or *Rules* or *Transitions*) are represented by a map of *Transition Keys* to *Transition Data*.

  Rules := TransitionKey $\rightarrow$ TransitionData

- A *Transducer* is a record of input symbols, strings of output symbols, states, initial state, transition rules.

  Transducer := (input: set(Input), output: set(Output*), state: set(State), init: State, rules: Rules)

- A *Trace Step* is a state transition record.

  TraceStep := (source: State, input: Input, target: State, output: Output*)

- A *Trace* is a sequence of state transition records.

Trace := TraceStep*

- A *Difference Sequence* is a sequence of records each of which consists of a string of output symbols and its relation to the source string and the target string. The relation can be represented by " ", "-", or "+", which means that the output appears in both the source and the target, appears in the source, or appears in the target, respectively.

  Diff := (output: Output$^+$, mod: {" ", "-", "+"})*

- A *Node ID* is a State.

  NodeID := State

- A *Edge ID* is a tuples of two States and an Input symbol.

  EdgeID := (State, State, Input)

- A *List of the Processed Nodes* is a sequence of States.

  ProcNodes := NodeID*

- A *List of the Processed Edges* is a sequence of tuples of two States and an Input symbol.

  ProcEdges := EdgeID*

- An *Edge Label* is a label used to denote incoming or outgoing edges. In case of a multi graph we need the state and the input symbol to determine the edge uniquely.

  EdgeLabel := (State, Input)

- A *Vector Label* is a label used to denote incoming or outgoing edges. This is the same as the edge label extended with the direction information.

  VectorLabel := (State, Input, {"in", "out"})

- A row of *Difference at a Node* is a record of the source edge label, target edge label, and the modified output.

  NodeD := (source: EdgeLabel, target: EdgeLabel, output: Output*)

- A *Difference Table at a Node* is a sequence of rows of differences.

NodeDT := NodeD*

- A *Difference Table at an Edge* is a sequence of rows of differences (modified outputs).

  EdgeDT := List(Output*)

- An *Edit Script* contains a sequence of modifications of a transducer.

  EditScript := (type: {"e", "ni", "no"}, nodeid: NodeID, edgeid: EdgeID, output: Output*)

- A *Data at a Node* is a record that corresponds to a node.

  NodeData := (pass: nat, diffTab: NodeDT)

- A *Data at an Edge* is a record that corresponds to an edge.

  EdgeData := (output: Output*, pass: nat, diffTab: EdgeDT)

- An *Inference Algorithm* is an algorithm which takes a *Transducer* and an input and output pairs as parameter and returns a transducer.

  InferenceAlgorithm := (Transducer, Input*, Output*)) → Transducer

3 The Graph Annotation Based Inference Algorithm

In this section we describe our advanced inference algorithm. The algorithm starts by recording the trace of the transducer execution on a given input. The output is extracted from the trace and is compared with the expected output. The algorithm builds a transition graph from the transition rules of the transducer. According to the result of the comparison, the algorithm annotates the transition graph. The annotations are assigned to the nodes containing the input edge, the new output, and the output edge. The input and output edges are the preceding and following edges in the trace around the new output. The annotations assigned to the edges express that the edge must be deleted or must be changed to print an empty element. To make the transducer capable of reproducing the modified trace, new transition rules are added or existing transition rules are modified according to the annotations. During the processing of the annotation, the algorithm may ask for further information or alert the user of contradictions.

Example An example transducer is depicted in Figure 1. The nodes and the edges represent the states and the transitions, respectively. The arrow labeled "a/x" between the node "A" and "B" means that if "a" exists on the input and the transducer is in state "A", an "x" is produced on the output and the new state is "B". Empty (or λ) input means that the system changes its state without reading anything from the input. This transducer produces "xyywzxywz" from input "aabaaba". We can specify the requested modification by providing a single example pair: the input "aabaaba" and the expected output "xyyvzxyvz". The algorithm is expected to infer a new transducer from the specified transducer and from the example pair.

Overview of the Algorithm In Figure 2 we depict the steps of the algorithm. The rectangles with rounded corner (yellow) represent the functions. The objects between rectangles represent the data passed between the functions by considering the example described in the beginning of Section 3. In the top of the figure the input parameters can be seen: the graphical representation of the transducer *trans*, the *input* ("aabaaba") and the *output* ("xyyvzxyvz"). The algorithm computes the modified transducer *trans'* as the result.

1. The transducer (*trans*) executes the *input* to produce a trace (*tr*). From the trace the output and the graph representation (*g*) of the transducer can be computed. The computed output can be compared to the expected output and the result of the comparison is represented in the diff sequence.

2. The modifications in the difference sequence are recorded as data ele-

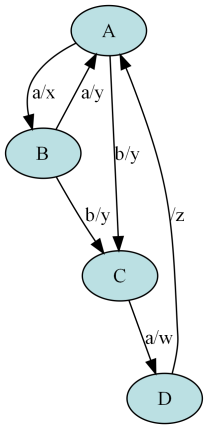


Figure 1: An Example Transducer

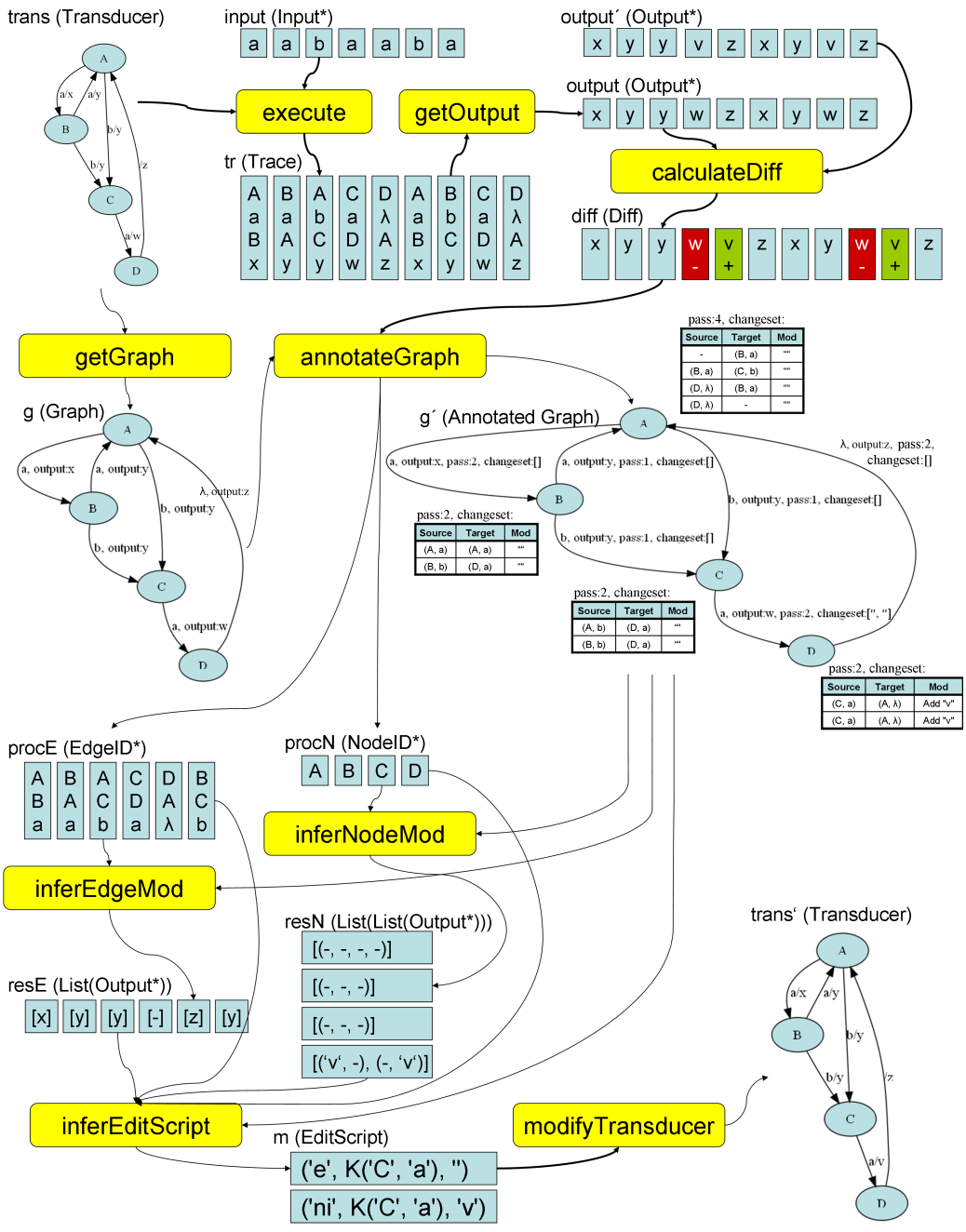


Figure 2: An Example of the Execution of the Algorithm

```
1  inferTransducer(Transducer trans, Input* input,
2        Output* output'): Transducer
3    Trace tr=execute(trans, input)
4    Output* output=getOutput(tr)
5    Diff diff=calculateDiff(output, output')
6    Graph g=getGraph(trans)
7    (g',procN, procE)=annotateGraph(tr, diff, g)
8    List(List(Output*)) resN=inferNodeMod(g', procN)
9    List(Output*) resE=inferEdgeMod(g', procE)
10   EditScript m=inferEditScript(g, resN, procN, resE, procE)
11   Transducer trans'=modifyTransducer(trans, m)
12   return trans'
```

Listing 1: The Main Algorithm

ments of the nodes and the edges of the graph by the annotateGraph function. The function also produces a list of the processed nodes and a list of the processed edges.

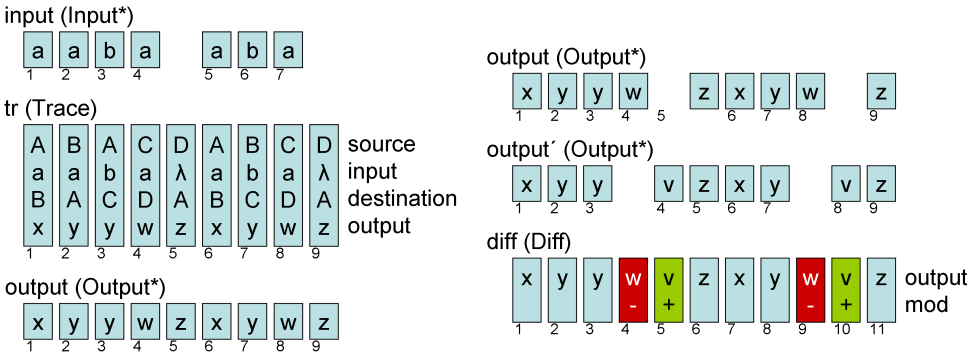
3. The functions inferNodeMod and the inferEdgeMod infer the solution from the annotated graph for each element of the list of the processed nodes and the list of processed edges, respectively.

4. The function inferEditScript creates an edit script (a list of modification steps) from the list of inferred solutions belonging to nodes and to edges. The transducer is finally modified by the function modifyTransducer resulting the transducer *trans'*.

The Main Function The main function computes the modification of the transducer according to the supplied example by executing the part of the algorithm. It takes the following inputs: a transducer *trans*, an example input *input* and an expected output *output'* and returns the modified transducer *trans'*.

> inferTransducer(Transducer trans, Input* input, Output* output'): Transducer

The main function can be seen in Listing 6. The algorithm starts with the execution of the input on the transducer which produces the trace *tr* (line 3). The trace is a sequence of trace steps corresponding to the executed transition rules. A trace step records the source state, the processed input



(a) The Trace (b) The Difference Sequence

Figure 3: Processing Steps of the Example

symbol, the target state and the produced output symbol of the corresponding transition. The output is the sequence of the output elements of the trace sequence (line 4). Then the sequence *diff* is calculated which describes the difference between the output and the expected output using a commonly used comparison algorithm, described in [5] (line 5). Then the transducer *trans* is converted to a graph representation *g* (line 6). The annotated graph *g'* is calculated from the original trace *tr* and the difference sequence *diff* (line 7). The lists of the processed edges *procN* and nodes *procE* are created along this calculation. The inference function infers the lists of modifications of the nodes *resN* and edges *resE* (line 8, 9). The edit script *m* is assembled from the lists of modifications and the list of processed edges and nodes (line 10). Finally, the new transducer *tr'* is built from the modified trace (line 11).

The function *execute(trans Transducer, input Input): Trace* initializes its internal state variable according to the *init* element of the *trans* parameter. Then the function executes the rules in the *trans* parameter. After each transition a new trace step is appended to the trace sequence. The *getOutput(tr Trace): Output* function returns the sequence of the output elements of the trace sequence. The *getGraph(trans Transducer): Graph* functions builds a directed multi graph from the transducer. The nodes of the graph are the states of the transducer. The edges are composed of the source and target states of the rules, the keys of the edges are the input symbols and the data elements contain the output symbols and the annotations.

Annotating the Transition Graph with the Modification This function converts the information from the diff sequence into the difference tables of the nodes and the edges with the help of the trace. The function takes

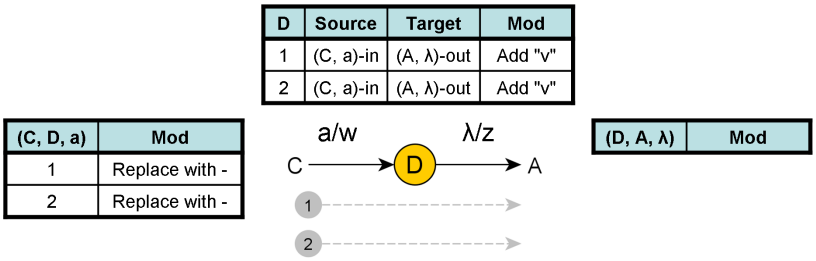


Figure 4: A Part of the Annotated Graph with Traces and Difference Tables

the graph, the diff and the trace as input; and returns the annotated graph, the list of the annotated nodes and the list of the annotated edges.

> annotateGraph(Graph graph, Trace trace, Diff diff): (Graph, NodeID*, EdgeID*)

The annotated graph is a copy of the original graph containing difference tables at the nodes and the edges. The function marks if a certain node or edge is represented in the trace. If it is represented, then we also mark whether it is left unmodified, or new tokens are inserted, or existing ones are removed. The transition which produced a specific list of the output symbols can be identified by simultaneously processing the diff sequence and the trace itself. As soon as the transition which produced (or removed) the output symbols is known, the position of the output symbols can be located in the annotated graph.

In Figure 4 the difference tables for Node D are shown, which are generated from the trace of Figure 3a. The node and the edges are labeled in the same way as it can be seen in Figure 1; the edges are additionally marked with the tuples of the source state, target state and input symbol. The trace elements which traverse Node D are represented by (numbered) gray arrows. The connection between the figures can be understood as follows (each line starts with the number of the trace in Figure 4, followed by the corresponding element numbers in Figure 3b, and is closed by the informal description of the trace elements):

- 1 (elements 4-6): Node D is passed for the first time, when a ”v” is printed (C) and then a ”z” is printed (A);
- 2 (tokens 9-11): Node D is passed for the first time, when a ”v” is printed (C) and then a ”z” is printed (A);

```
1 annotateGraph(Graph graph, Trace trace, Diff diff):
2     (Graph, NodeID*, EdgeID*)
3   (graph', diff')=processPrefix(graph, diff, head(trace))
4   (graph', proceN, procE)=annotateGraphImpl(graph', trace,
5       diff', [head(trace).source], [])
6   return (graph', procN, procE)
```

Listing 2: The annotateGraph Algorithm

The insertion is stored in the difference table of the Node D and the deletion is stored in the difference table of Edge (C, D, a) as a replacement of the original output symbols.

In a more precise way the tables in the nodes and in the edges of the trace are generated as follows. Each row in the tables represents one continuous sequence of difference elements corresponding to the trace passing the edge or the node. In case the tables are located at the nodes, the source edge and target edge are recorded, besides the elements of the difference sequence. The difference tables are constructed by the function *annotateGraph*. This function calls two sub-functions: the function *procPrefix* which processes possible insertion at the beginning of the difference sequence (line 3); and the function *annotateGraphImpl* which actually do the processing of the trace steps in a recursive manner (line 4-5).

The function *procPrefix* searches additions at the beginning of difference sequence (lines 6-8). Then it modifies the difference table at the node according to the result of the search (line 9-12). The target node and the source node of two successive trace steps are the same. If we pre-process the first source node, than we can uniformly process all the trace steps by processing the modifications belonging to the edge and to the target node, because we can assume that the source node was already processed.

The function *annotateGraphImpl* processes the trace steps if the trace sequence is not empty (lines 3-25). The modifications belonging to the first step of the trace are identified by calling *procTraceStep* which returns the modified output belonging to the edge (if it is modified), the modified output belonging to the target node, and the rest of the difference sequence (lines 4-5). If further trace step exists (one after the currently processed), it is used as the target state of the edge, otherwise the unknown label (?, ?) is used (lines 6-8). The target node of the trace step and the edge represented by the trace step are selected for annotation (line 9-11). The counters which count how many times the transitions and the states referenced by the trace are increased (line 12-13). The output corresponding to the node is recorded

```
1  procPrefix(Graph graph, DiffSeq diff, TraceStep startT):
2      (Graph, Diff)
3    State startN=startT.source
4    Nat i=0
5    Output* outputN=""
6    while diff[i].mod=="+":
7      outputN+=diff[i].output
8      i+=1
9    getNode(graph, startN).pass=1
10   if i!=0:
11     getNode(graph, startN).diffTab+=((?, ?),
12         (startnode.target, startnode.input), nodeOutput)
13   return (graph,diff[i:])
```

Listing 3: The processPrefix Function

even if they have not been modified (lines 14-15). The output corresponds to the edge is recorded only if modification occurred (line 16-17). The processed node and edge are added to the processing sequence if they are not processed already by once (line 18-23). The rest of the trace and diff sequence are processed recursively by calling the function itself (24-25).

Inferring the Possible Modifications These functions infer the possible transformation modifications from the difference tables in the annotated graph. The tables in the nodes and in the edges are processed in different ways. To process the annotated graph the inferEdgeMod and inferNodeMod functions take the list of the processed edges and the list of the processed nodes, respectively. They produce the lists of the inferred solutions.

inferNodeMod(Graph graph, ProcNodes procN): List(List(Output*))

inferEdgeMod(Graph graph, ProcEdges procE): List(Output*)

- **Processing node tables:** Inferring modifications in case of the nodes can deliver three kinds of results: one solution is possible (the ideal case), several modifications are possible and a contradiction has occurred. Processing node tables are done in two steps: first, they are converted to a new representation in which the columns represent edges connected to the node; then solutions are inferred from the new representation. The detailed algorithm will be explained after the description of an example below.

```
annotateGraphImpl(Graph graph, Trace trace, Diff diff,
    NodeID procN, EdgeID procE): (Graph, NodeID*, EdgeID*)
  List(NodeID) procN
  List(EdgeID) procE
  if length(trace) > 0:
    (outputE, outputN, diff')=procTraceStep(head(trace),
        diff)
    targetLabel=(?, ?)
    if length(trace)>1:
      TargetLabel=(trace[1].target, trace[1].input)
    nodeData=getNode(graph, head(trace).target)
    edgeData=getEdge(graph, head(trace).source,
        head(trace).target, head(trace).input)
    nodeData.pass+=1
    edgeData.pass+=1
    append(nodeData.diffTab, (head(trace).source,
        head(trace).input), targetLabel, outputN))
    if outputE!=head(trace).output:
      append(edgedict.diffTab, outputE)
    if head(trace).target not in procN:
      append(procN, head(trace).target)
    if (head(trace).source, head(trace).target,
        head(trace).input) not in procE:
      append(procE, (head(trace).source,
          head(trace).target, head(trace).input))
    (graph, procN, procE)=annotateGraphImpl(trace[1:],
        diff', graph, procN, procE)
  return (graph, procN, procE)
```

Listing 4: The annotateGraphImpl Function

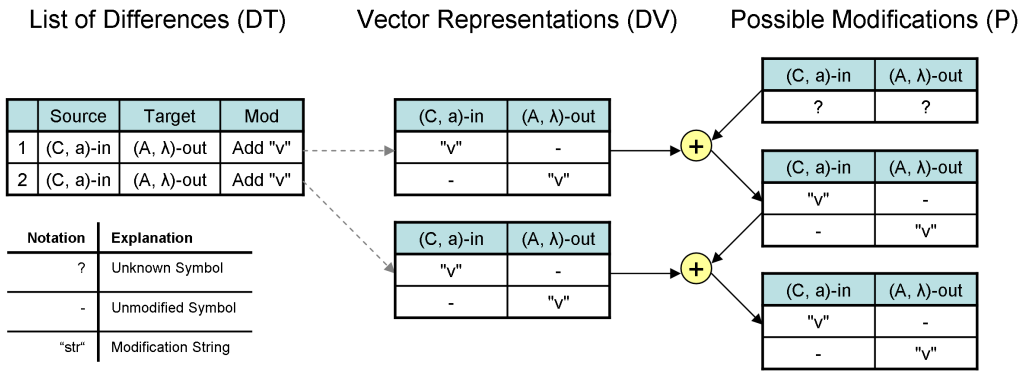


Figure 5: Resolution of difference tables

- **Processing edge tables:** Inferring modifications from the differences in case of the edges are simple: the modifications imposed by all lines in the table must be identical in the table in order to deduce a consistent modification. A difference between the lines means that the same parts of the output instructions are modified in an inconsistent way by the user.

The leftmost table of the Figure 5 shows the *difference table (DT)* of Node 1 presented in Figure 4. The tables in the middle of the figure represent the *difference vectors (DV)* that corresponds to the lines of the left table. The tables on the right side of the figure show the changes prescribed by the *possible modifications (P)* after the vectors are processed. The difference vector (DV) contains a column for each input and output edge, which is labeled with the identifier of the edge and its direction. The column structure of the possible modification tables (P) are the same.

The correspondence between the lines of the list of differences and their vector representation is shown by the dashed (grey) arrows. (1) The first line of the lists of difference table (DT) is represented by two vectors (DV), because the token can be added either before traversing the node or after traversing it. (4) The second element of the difference list represents the same difference again and this element is processed in the same way as the first one.

The uppermost table (P) in the right column represents the initial state of the possible modifications. The second table (P) shows the combination of the initial table (P) and the first difference vector (DV). The third table (P) shows the combination of the second table (P) and the second difference vector (DV); and so on. The bottommost table of right column (P) is the final possible modification table which belongs to Node 1.

The inference of the modification of nodes is implemented by the function *inferNodeMod*. It iterates over the list of the nodes *procN* which have to be processed (lines 3-39). A mapping *vecMapping* is created, which maps the labels of the edges to a position in the difference vector (line 4). Then a difference vector (DV) *defaultDV* is initialized with unknown values (line 5). This vector is used to initialize the possible modification table (P) (line 6). The difference table (DT) is processed in a loop, in which we distinguish four cases (lines 7-39):

- processing a row which is recorded at the start of the trace (source edge is (?, ?));
- processing a row which is recorded at the end of the trace (target edge is (?, ?));
- processing a row which contains no modification;
- processing a row which contains modification.

The element of the difference vector can be marked with the unknown symbol (*?*) i.e. we have no information about the element; the unmodified symbol (in the algorithm marked with *""* and otherwise with - for better readability); and a difference string (*"a string"*). A difference vector can be formed for each line of the difference list as follows:

- If the row is recorded at the start of the trace, a single vector is created with the output symbol of the row for the target edge and the remaining columns will contain the unknown symbol (lines 9-11);
- If the row is recorded at the end of the trace, a single vector is created with the output symbol of the row for the source edge and the remaining columns will contain the unknown symbol (lines 14-16);
- If no difference is recorded in the list, a single vector is created with the unmodified symbol for the source and target edges and the remaining columns will contain the unknown symbol (lines 19-23);
- If a difference is recorded in the list, two vectors are created: the first one contains an unmodified symbol for the source edge and the modification string for the target edge, and in the other case, vice versa. All other elements will contain the unknown symbol (lines 26-30, 32-37).

In the initial state, the table of possible modifications (P) contains a single row filled with the unknown symbol (line 6). Each row of the difference

vectors table (DV) is combined with the rows of the possible modifications table (P) and the result becomes the new possible modifications table (P) (lines 12, 17, 24, 31, 38). The combination of the rows of the tables is carried through by merging each line of the tables (DV, P). Merging two lines mean that two rows are combined element-wise. The rows can be merged successfully if each pair of the elements is the same or one of the elements is the unknown symbol (in this case the result row contains the other element). Otherwise, the merging fails and no result is produced. The tables are combined by forming the union of the result of the row merges (line 38). The final possible modification table (P) is stored in result sequence *resN*.

The inference of the modification of edges is implemented by the function *inferEdgeMod*. This function is also iterates over the edges (lines 3-14). The loop deals with three possible cases:

- if the output of the edge was left unchanged, the result will be the original output of the edge, the value of which indicates no change (lines 5-7);
- if the output of the edge was changed in some cases, the result will be empty, the value of which indicates the contradiction (lines 8-10);
- if the output of the edge was changed in all cases, depending on whether the changes are contradictory, the result will be empty or the value will be changed (lines 11-18).

Modifying the Transformation Two functions are responsible for the modifications of the transducer according to the list of possible modifications. The function *inferEditScript* creates an edit script by selecting one of the modification for each nodes and edges from the possible modifications. The function *modifyTransducer* modifies the transducer according to the edit script.

> inferEditScript(Graph g, ProcNodes resN, List((Output*)*) procN, ProcEdges resE, List(Output*) procE): EditScript
>
> modifyTransducer(Transducer trans, EditScript m): Transducer

The edit script which is used to update transformation is assembled from the sequence of possible modification according to rules as follows:

- If a *consistent modification occurred in an edge*, the original output is replaced by the modified output.

```
1  inferNodeMod(Graph graph, ProcNodes procN): NodeDT
2    resN=list(list(Ouput*))
3    for node in procN:
4      vecMapping=buildInOutEdgesMapping(graph, node)
5      (Output*)* defaultDV=initUnknownDV(length(vecMapping))
6      possibleM=[defaultDV]
7      for row in node(graph, node).diffTab:
8        if row.source==(?, ?):
9          newRow=defaultDV
10         newRow[vecMapping((row.target[0], row.target[1],
11               'o'))]=row.output
12         possibleM=mergeSoultions(possibleM,newRow)
13       elif row.target==(?, ?):
14         newRow=defaultDV
15         newRow[vecMapping((row.source[0], row.source[1],
16               'i'))]=row.output
17         possibleM=mergeSoultions(possibleM,newRow)
18       elif row.output=="":
19         newRow=defaultDV
20         newRow[vecMapping[(row.source[0], row.source[1],
21               'i')]]=""
22         newRow[vecMapping[(row.target[0], row.target[1],
23               'o')]]=""
24         possibleM=mergeSoultions(possibleM,newRow)
25       else:
26         newRowA=defaultDV
27         newRowA[vecMappings[(row.source[0], row.source[1],
28               'i')]]=row.output
29         newRowA[vecMappings[(row.target[0], row.target[1],
30               'o')]]=""
31         newpossibleMA=mergeSoultions(possibleM,newRowA)
32         newRowB=defaultDV
33         newRowB[vecMappings[(row.source[0], row.source[1],
34               'i')]]=""
35         newRowB[vecMappings[(row.target[0], row.target[1],
36               'o')]]=row.input
37         newpossibleMB=mergeSoultions(possibleM,newRowB)
38         possibleM=union(newSolutionA, newSolutionB)
39     resN=resN+[possibleM]
40   return resN
```

Listing 5: The inferNodeMod Algorithm

```
1  inferEdgeMod(Graph graph, ProcEdges procE): List(Output*)
2    resE=list(Output*)
3    for edge in procE:
4      edgeData=getEdge(graph, edge[0], edge[1], edge[2])
5      if length(edgeData.diffTab)<edgedata.pass
6           length(edgeData.diffTab)==0
7           resE=res+[[edgeData.output]]
8      elif length(edgeData.diffTab)<edgeData.pass
9           length(edgeData.diffTab)!=0:
10          resE=res+[list(Output*)]
11     else
12       solution=[?]
13       for row in edgeData.diffTab
14         if solution!=[row.output]:
15           solution=list(Output*)
16         if solution==[?]:
17           solution=[row.output]
18       resE=res+[solution]
19   return resE
```

Listing 6: The inferEdgeMod Algorithm

- If *inconsistent modifications occurred in an edge*, an error message can be emitted or the designer can be prompted to resolve the conflict.
- If a *consistent single modification was inferred in a node*, it is appended to the beginning or the end of the output of the appropriate transition.
- If a *modification occurred in a node and multiple solutions were inferred*, the one with smaller number of modifications is selected (if solutions with the same number of modifications are available, the first one is chosen).
- In case of *inconsistent modifications in a node*, an error message can be emitted or the designer can be prompted to manually resolve the conflict.

As a result, our error correction technique is automated in case of consistent minor modifications, and it becomes semi-automated when conflict resolution requires input by the designer.

The edit script of the described example are the following: replace with " the output of the transition from state C reading 'a'; insert 'v' to the beginning of the output of the transition from state C reading 'a'. The

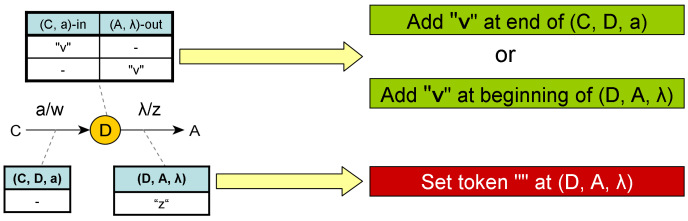


Figure 6: Creating the Edit Script from Possible Modifications Tables

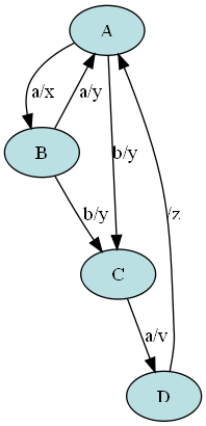


Figure 7: The Inferred Transducer

relation between the possible modifications and the inferred edit script of the modified part of the transducer can be seen in Figure 6. The final transducer can be seen in Figure 7.

4 Extension of the Algorithm

In the description of the algorithms we described for clarity the processing of a single input/output pair as an example. This section describes a minor modification of the algorithm described in the previous section (called GABI - Graph Annotation Based Inference) to make it more usable for practical problem solving. To make the comparison of the two algorithms easier we also sketch the extension of the algorithm previously described in [3] (called DTMBI - Direct Trace Modification Based Inference).

We present the extended version of the inferTransducer in Listing 7. This algorithm differs from Listing 6 in three points:

- the function returns a list of possible solutions and expects lists of input and expected output combinations;

```
1  inferMultipleTransducer(Transducer trans, List(Input*)
2        inputL, List(Output*) outputL'): Transducer*
3    Graph g=getGraph(trans)
4    List(NodeID) procN
5    List(EdgeID) procE
6    while length(inputL)>0:
7      input=head(inputL)
8      output'=head(outputL')
9      inputL=tail(inputL)
10     outputL'=tail(outputL')
11     Trace tr=execute(trans, inuput)
12     Output* o=getOutput(tr)
13     Diff diff=calculateDiff(output, output')
14     (g',procN', procE')
15           =annotateGraph(tr, diff, g, procN, procE)
16     g=g'
17     procN=procN'
18     procE=procE'
19   List(List(Output*)) resN=inferNodeMod(g', procN)
20   List(Output*) resE=inferEdgeMod(g', procE)
21   List(EditScript) mL
22         =inferEditScriptList(g, resN, procN, resE, procE)
23   List(Transducer) transL'
24   while length(mL)>0:
25     m=head(mL)
26     Transducer trans'=modifyTransducer(trans, m)
27     transL'=transL'+[trans']
28   return transL'
```

Listing 7: The Main Algorithm Returning Multiple Solutions

- capability to handle multiple input/output pairs (lines 3-18);
- capability to return all possible solutions (lines 21-27).

Handling Multiple Examples The algorithms need slight modification to handle multiple input/output pairs. This feature is necessary to create test cases which can produce specific values of the selected metrics e.g. to reach complete transition coverage in a transducer which has a tree structure.

In the GABI algorithm the function annotateGraph is executed successively on each diff sequence by keeping the previous annotated graph. Till that point each input/output pair can be processed separately in the same way as in the simple version of the algorithm (lines 11-13). To handle multiple pairs we have to summarize the inputs in some point of the algorithm execution (lines 14-15). The annotateGraph function is extended with two extra parameters to get the list of the processed nodes and edges instead of initializing them with an empty list internally. Consequently the annotations are summarized in a single annotation graph.

In the DTMBI algorithm the result can be inferred by producing the modified trace for each input separately and then the function modifyTransducer is executed successively on each modified traces. The extra constraint is that we allow only to modify the transitions contained in the processed trace and we do not allow to change the initial state of the transducer.

Producing Multiple Solutions The described version of the GABI algorithm selects the inferred solution from all the possible versions with a quite simple heuristics (i.e.: selecting the first combination). To check the correctness of the algorithm we have to verify that the correct solution is always listed among the solution candidates. This feature is also necessary to create better heuristic by evaluating the properties of the inferred solutions. The function annotateGraph is modified between the lines. The function inferEditScriptList returns a list of solutions instead of a single solution.

Multiple Output Symbol Handling The DTMBI algorithm is implemented in a way that the transition rules are allowed to produce a single output symbol only. Producing multiple output symbols can be simulated by a sequence of transitions each of which produces a single output (normalization), e.g.: transition [(A, a,B, xyz)] can be encoded by the transitions [(A, a, N1, x), (N1, λ, N2, y), (N2, λ, z, B)]. To make the result of the DTMBI algorithm comparable to the result of the GABI, we have to reverse the effect of the normalization. This can be done by merging the transition

into the preceding transition in case of reading empty input symbols (de-normalization). An example of the normalization and the de-normalization can be seen in Figure 9a, 9b and Figure 9c, 9d, respectively.

Similar algorithm is described in [1] in connection with the problem of eliminating lambda input in PDFAs. The detailed description of the de-normalization algorithm is as follows:

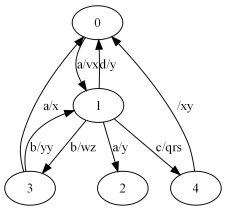
- A transition is selected for de-normalization if it contains multiple output symbols.
- New transitions are created to output the additional output symbols. The transitions are connected to each other through new unique states and they read no input symbols. The target state of the last newly created transition equals with the target state of the selected transition.
- The selected transition is modified to output only its first output symbol and its target state is the source state of the first newly created transition.

The detailed algorithm of the normalization is as follows:

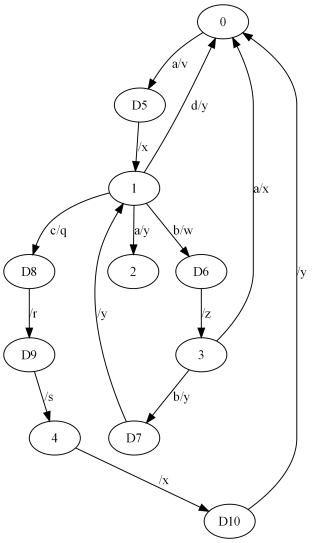
- A transition is selected for elimination if it reads no input (a), its source state is not equal with the initial state of the transducer (b), and the number of incoming edges is one.
- The selected transition is merged into the preceding transition by appending its output to the predecessors' output and by overwriting the target-state of the predecessor with its target-state.
- The selected transition and the intermediate state are deleted.

5 Evaluation of the Algorithm

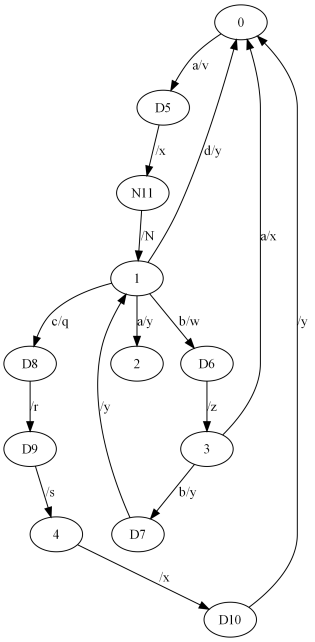
In this section we evaluate the algorithms GABI and DTMBI. The qualitative evaluation can be performed by evaluating examples and comparing the inferred results to the expected results. The quantitative evaluation can be carried through by generating several examples and calculating metrics over them.



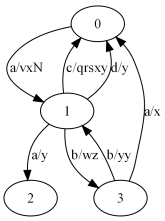
(a) Multiple output symbol per transition



(b) Single output symbol per transition

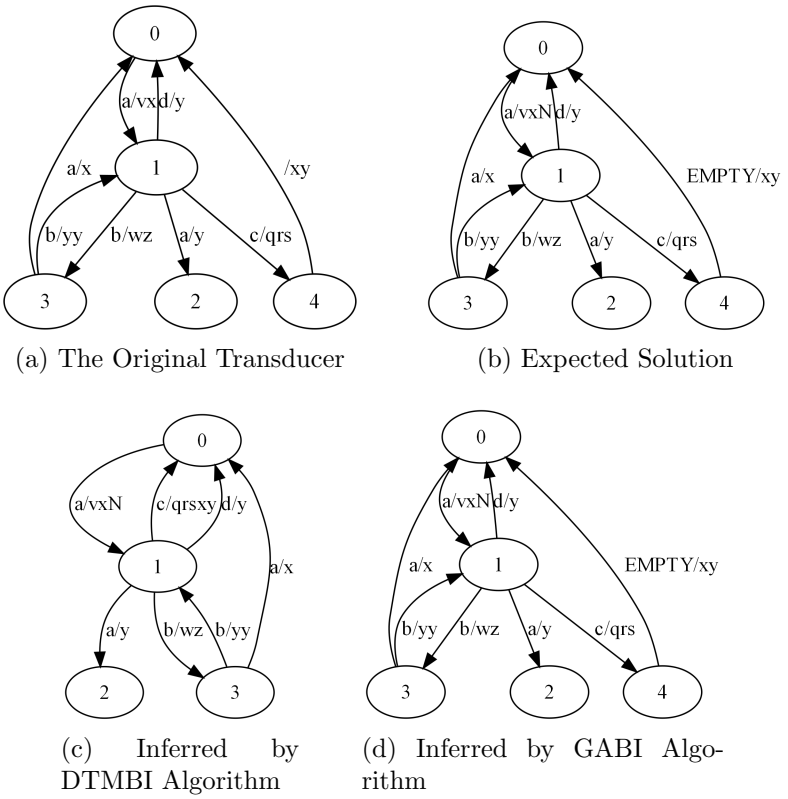


(c) Single output symbol per transition



(d) Multiple output symbol per transition

Figure 8: Example for the Normalization/De-normalization



(a) The Original Transducer

(b) Expected Solution

(c) Inferred by DTMBI Algorithm

(d) Inferred by GABI Algorithm

Figure 9: Example 1

5.1 Qualitative Evaluation

In this section we evaluate manually the behavior of the algorithms on three examples. We present the input/expected-output pair, the original transducer, the expected modification, and the result of the two inference algorithms.

The expected result was derived from a survey which was conducted among PhD students in computer science. The survey contained the same examples and the participants had to choose between the possible modifications. There were no uniquely preferred solutions. A major critique of the students was that they would select these preferences much more surely if the examples would contain some domain specific labels and not meaningless letters and numbers. In the current experiment most of the students have selected the modifications which cause the smallest change in the transducer.

The first example demonstrates a single point modification of the transducer which can be learned from a single input/output pair. The algorithms

take the inputs as follows:

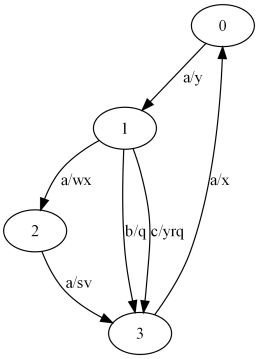
- the transducers can be seen in Figure 9a,
- the input string is "abb" and
- the expected output string is "vxNwzyy".

The original transducer produces "vxwzyy" for the specified input string. During the processing of the input string the transducer reaches the states: 0, 1, 3, and 1. We know that the additional output N must be produced in one of the following transitions: the transition between 0 and 1 or the transition between 1 and 3. The human solutions prefer appending the solution which appends N to the end of the output of the transition between 0 and 1 (Figure 9b). We can see that in this case the two algorithms select the same solution (Figure 9c, 9d). The difference is caused by the normalization/denormalization process which eliminated the node 4.

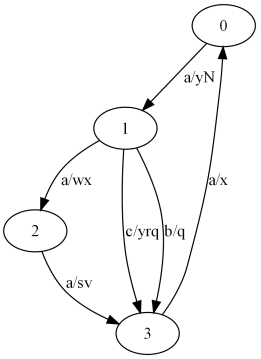
The second example demonstrates modifications of multiple paths; here multiple input/output pairs are specified. The algorithms take the inputs as follows:

- the transducers can be seen in Figure 10a,
- the first input string is "aaa",
- the first expected output string is "yNwxsv",
- the second input string is "aba",
- the second expected output string is "yNqx".

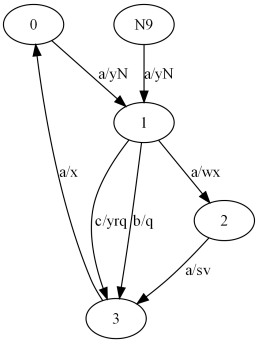
The original transducer produces "ywxsv" and "yqx" for the first and second input string, respectively. During the processing of the first input string the transducer reaches the states: 0, 1, 2, and 3. During the processing of the second input string the transducer reaches the states: 0, 1, 3, and 1. The transducers can be seen in Figure 10. In this example the DTMBI algorithm stops during the processing. The termination is caused by the constraint that the initial state can be modified only once. The renamed states of the reoccurring trace elements do not match to the renamed edges, therefore the algorithm tries to modify the initial state for a second time (which causes the termination). We present in Figure 10c the partially modified form of the transducer before the algorithm terminates (in this transducer the initial state must be modified to N9 to produce the right output to the second input/output pair).



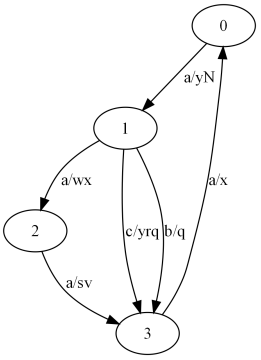
(a) The Original Transducer



(b) Expected Solution

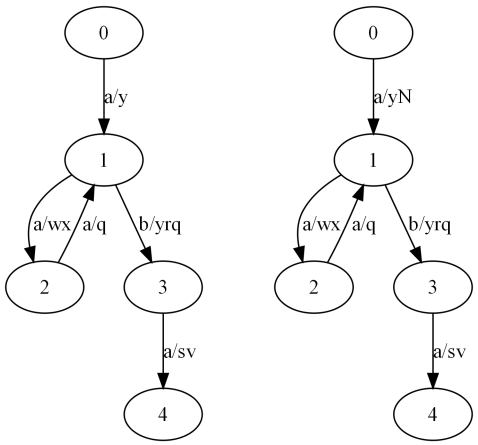


(c) Intermediate result of the DTMBI Algorithm



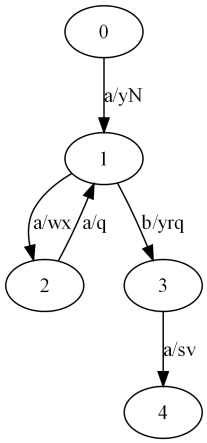
(d) Inferred by GABI Algorithm

Figure 10: Example 2

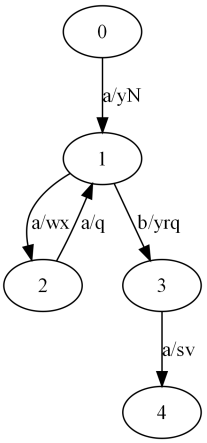


(a) The Original Transducer

(b) Expected Solution



(c) Inferred by DTMBI Algorithm



(d) Inferred by GABI Algorithm

Figure 11: Example 3

The third example demonstrates a single modification of a cycle in the transducer; single input/output pairs are specified. The algorithms take the inputs as follows:

- the transducers can be seen in Figure 11a,
- the input string is "aaaba",
- the expected output string is "yNwxqyrqsv".

The original transducer produces "ywxqyrqsv" for the input string. During the processing of the input string the transducer reaches the states: 0, 1, 2, 1, 3 and 4. The transducers can be seen in Figure 11. Both algorithms deliver the expected solution.

The fourth example demonstrates a single modification of a cycle in the transducer; single input/output pairs are specified. In this example we cover partially the same path twice. The algorithms take the inputs as follows:

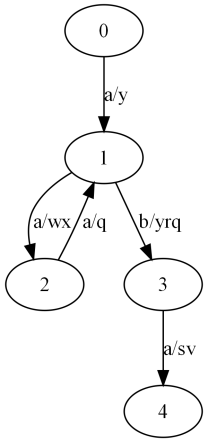
- the transducers can be seen in Figure 12a,
- the input string is "aaaaaba",
- the expected output string is "yNwxqNwxqyrqsv".

The original transducer produces "ywxqwxqyrqsv" for the input string. During the processing of the input string the transducer reaches the states: 0, 1, 2, 1, 3, 1, 3 and 4. The transducers can be seen in Figure 12. By covering the cycle between state 1 and 2 the DTMBI algorithm delivers a completely different result than the GABI algorithm.

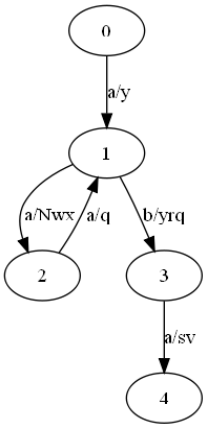
5.2 Quantitative Evaluation

In this section we compute and compare some metrics of the execution of the two algorithms. First we generate the metrics from the examples presented in the previous section. Than we generate random graphs and inject random mutations to demonstrate the metrics over a larger set of examples.

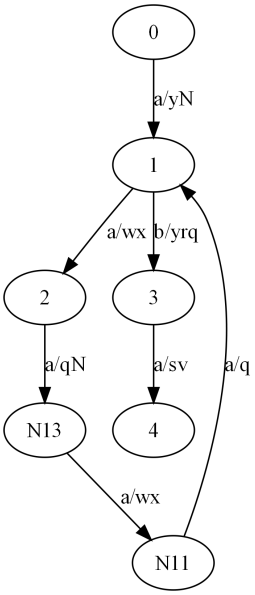
Metrics of the Hand-Crafted Examples In Figure 13 we present the metrics calculated from the presented examples. The presented metrics are described in Section 4 of [3]. The first and the second rows contain the number of the exercises and the selected the algorithms (D - DTMBI; G - GABI), respectively. The further rows contain the metrics grouped by they types:



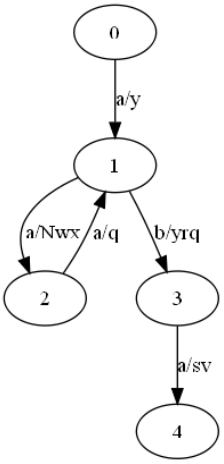
(a) The Original Transducer



(b) Expected Solution



(c) Inferred by DTMBI Algorithm



(d) Inferred by GABI Algorithm

Figure 12: Example 4

Example	1		2		3		4	
Algorithm	D	G	D	G	D	G	D	G
No. of new states	0	0	-	0	0	0	2	0
No. of deleted states	1	0	-	0	0	0	0	0
No. of new states / No. of original states	0.00	0.00	-	0.00	0.00	0.00	0.40	0.00
No. of deleted states / No. of original states	0.20	0.00	-	0.00	0.00	0.00	0.00	0.00
No. of new trans.	2	2	-	1	1	1	4	1
No. of deleted trans.	3	2	-	1	1	1	2	1
No. of new trans. / No. of original trans.	0.25	0.25	-	0.16	0.20	0.20	0.80	0.20
No. of deleted trans. / No. of original trans.	0.37	0.25	-	0.16	0.20	0.20	0.40	0.20
Modified Output	1.00	1.00	-	0.68	1.00	1.00	1.00	0.86
Average length till the first difference	2.0	2.0	-	4.3	1.0	1.0	1.0	1.0
Average number of changed characters	3.2	7.9	-	1.5	1.0	1.0	5.9	2.8
State Coverage	0.38	0.38	-	0.83	1.00	1.00	1.00	1.00
Transition Coverage	0.60	0.60	-	0.80	1.00	1.00	1.00	1.00
No. of possible interpretations	2	2	-	1	2	2	1	1

Figure 13: Summary of the Metrics

- Structural Metrics of the Transducer Modification,
- Behavioral Metrics of the Transducer Modification,
- Coverage Metrics of the Transducer Modification,
- Metric of the Intention.

We interpret the modifications as additions and deletions. The GABI algorithm can deliver several alternative solutions, which we display in separate columns. If the algorithm cannot produce evaluable results we omit the metrics in the specific column.

In case of the first and third examples there are no significant differences. In case of the second example we have no comparable results. In the fourth case we can see significant difference between the two solutions: the DTMBI delivers a much larger transducer: the number of the states is increased by 40% and the number of the transitions is increased by 40% (80% addition and 40% deletion).

Random Transducer Generation To produce a comparable amount of examples for the measurement we generate transducers randomly. Our random transducer generation is based on the combination of random graph generator algorithms [4].

- We first generate a random connected direct graph. This ensures that we do not have unreachable nodes. Than we generate a random undirected graph of which edges are added to the first graph as directed edges pointing form higher to lower node id numbers. Adding such edges introduce loops.
- After the creation of the graph random mutations are introduced. Mutation craeted by selecting edges randomly in the graph and by modifing their output.
- Random paths are created: the shortest path is selected between the node 0 and the mutated edges. This path also contains random nodes of the graph.
- The input/output pairs are generated: the input is calculated from the path; the output is calculated by executing the mutated transducer with the generated input.
- The two algorithms are used to generate the solutions from the original transducers and the input/output pairs.

Example Set	1		2	
Algorithm	D	G	D	G
No. of examples with mutations	97	97	83	83
No. of successful inferences	3	69	17	95
No. of unsuccessful inferences	97	31	83	5
No Output Case 1	17	-	2	-
No Output Case 2	12	-	9	-
No Output Case 3	-	10	-	1
No. of inferences produced results	68	21	72	4

Figure 14: Inference Results of the Generated Transducers

- The results are compared to mutated transducers.
- The metrics are computed from the results.

Results of the Generated Transducers We generated two sets of the examples with one hundred transducers each. In the first set we have introduced a high number of mutations per transducer (on average 4.5 edges are modified; the maximum is 5). In the second set we have introduced low number of mutations per transducers (on average 1.5 edges are modified; the maximum is 2). The first set and second set contain three and 17 examples with no mutations. The result of the experiment is summarized in Figure 14 and the complete list can be found in [2]. Comparing the results we can state that the GABI algorithm delivers much better results (comparing only examples with modifications). From the first set (97 examples) and the second set (83 examples) the GABI was capable to infer the expected result in 64.02 percent and 64.74 percent of the examples, respectively. The DTBMI was not capable to infer anything except those transducers that have no modifications.

We studied why the algorithms have calculated no or not the expected solutions. The reasons for the inference algorithms producing no solutions are as follows:

- No Output Case 1: The DTMBI algorithm produces no results for multiple example pairs: the goal of the algorithm is to produce the lowest difference in the output while keeping the constraints to produce the right solution for the already presented examples. This makes the algorithm to decide "early" about the preferred results. In the case of single input/output pairs, this behavior leads in the worst case to a solution which only accepts the specified example. In case of multiple

input/output pairs the algorithm in the worst case requires two different initial states for the same transducer which is impossible. This is an expected behavior.

- No Output Case 2: This is a defect in the described DTMBI algorithm: the algorithm does not handle the special case in which inserting transitions having empty inputs can form a circle (a transition sequence that has the same source and destination state) that requires no input. This renders the transducers to be non-deterministic, because the "loop" can be executed arbitrarily many times without reading from the input.

- No Output Case 3: The GABI algorithm produces no results if the examples are inconsistent. This case is described in details in Section 3.

In some cases, the GABI algorithm does not produce output (because it finds the input inconsistent - No Output Case 3) or it produces output which differs from the expected outcome. We have investigated this problem, because all the examples were generated by valid mutations of the transducers on which the algorithm expected to work correctly. The result of the investigation was that the used differentiation algorithm sometime provides "misleading" difference sequence. If the result of the differentiation algorithm is replaced with a hand crafted difference sequence (i.e. one which reflects the order of modifications) and the rest of the computation is fed with this difference sequence the algorithm works as it is expected. To demonstrate the problem consider a character sequence 'x', 'y', 'z', 'v' and replace 'y' with '0' and replace 'z' with '1'; the result will be 'x', '0', '1', 'v'. The expected difference sequence by the algorithm is 'x', '-y', '+0', '-z', '+1', 'v'. In practice the selected differentiation algorithm produces 'x', '-y', '-z', '+0', '+1', 'v', which does not reflects the order of the modifications. This behavior explains why more mutations lead to more unsuccessful inferences: in case of a random selection of transitions for modification, the more transitions have been selected, the higher is the chance that two successive transitions have been selected. The difference between start and end states contains incomplete information about the intermediate states, thus in general it is not possible to create a better difference sequence generation algorithm. However we could modify the described algorithm in such a way that it takes as input the difference sequence instead of the expected result, e.g. the modification steps which are recorded by an editor.

Metrics of the Generated Transducers The average values of the metrics are displayed in Figure 15. By comparing the structural metrics we can

Example Set	1		2	
Algorithm	D	G	D	G
No. of new states	5.4	0	2.0	0
No. of deleted states	0.6	0	0.5	0
No. of new states / No. of original states	0.54	0.00	0.20	0.00
No. of deleted states / No. of original states	0.06	0.00	0.05	0.00
No. of new trans.	8.2	4.4	3.2	2
No. of deleted trans.	4.7	4.4	2.8	2
No. of new trans. / No. of original trans.	0.33	0.18	0.13	0.08
No. of deleted trans. / No. of original trans.	0.19	0.18	0.12	0.08
Modified Output	0.72	0.67	0.64	0.41
Average length till the first difference	3	3.2	3.1	3.4
Average number of changed characters	3.3	2.5	3.3	2.0
State Coverage	0.44	0.44	0.23	0.23
Transition Coverage	0.77	0.77	0.53	0.53
No. of possible interpretations	2.4	2.4	1.6	1.6

Figure 15: Average of the Metrics of the Generated Sets

see that GABI is the much superior algorithm, because it produced a lower number of changes.

- The DTBMI algorithm added and deleted several states and transitions.
- The GABI algorithm behaved as it was expected i.e. it did not introduce or remove any states and modified only transitions (that is why the number of transition additions and deletions are equal).

The behavioral metrics of the algorithms (the differences in the output of the original and the inferred transducer for a bounded length enumeration of the possible inputs) are surprising: we expected a lower number in case of DTBMI. In case of the second set the GABI algorithm has a significantly lower number. This means that with respect to the behavioral metrics the GABI algorithm produces better results too. The coverage metrics and the number of possible interpretations correlate with the number of modification introduced in the modified transducers. These metrics contain information about the quality of the examples. In our experiment the GABI algorithm was superior independently from the quality of the examples.

5.3 Summary of the Evaluation

We have described and evaluated four manually crafted examples. The results have demonstrated that in two cases out of four the GABI algorithm behaved significantly better than the DTBMI algorithm and in no case behaved worse. Furthermore, we generated and modified two hundred example transducers randomly. The results of the inference over the random examples shows that in case of larger examples the DTMBI algorithm as it is described in the previous technical report is unusable.

6 Conclusion

We have described an algorithm which infers transducer modifications from an example set of input/output pairs. We demonstrated the efficiency of the algorithm by comparing it to our previous algorithm. We have evaluated the algorithms with two types of examples: four hand-crafted examples and two sets of one hundred autmatically generated examples. The proposed algorithm solves the described inference problem correctly according to the tests on large random graphs. The result of comparing the newly proposed and the previously described algorithm was that the newly proposed algorithm

behaved superior. We are going to further test our algorithm in the context of XSLT transformation modifications by example.

Developing new algorithms always results in several new open research questions. In our case completely new open topics are the psychological aspects of such modifications by example problems i.e. which solution is preferred by the user of an algorithm in the case of several possible outcomes.

References

[1] Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA, 2010.

[2] Gabor Guta. Results of DTMBI and GABI algoritms on Random Transducer Sets. `http://www.risc.jku.at/people/gguta/res_random_tr.xls`.

[3] Gabor Guta. Finite State Transducer Modification by Examples. Technical Report 00-00, RISC Report Series, University of Linz, Austria, 2010.

[4] P. L. Krapivsky and S. Redner. Organization of Growing Random Networks. *Physical Review E*, 63(6):066123+, 2001.

[5] John W. Ratcliff and David Metzener. Pattern Matching: The Gestalt Approach. *Dr. Dobb's Journal*, (July):46, 1988.