A Genetic Algorithm for Discovering Process Trees

J.C.A.M. Buijs, B.F. van Dongen, W.M.P. van der Aalst
Faculty of Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, the Netherlands
Email: {j.c.a.m.buijs, b.f.v.dongen, w.m.p.v.daalst}@tue.nl

Abstract—Existing process discovery approaches have problems dealing with competing quality dimensions (fitness, simplicity, generalization, and precision) and may produce anomalous process models (e.g., deadlocking models). In this paper we propose a new genetic process mining algorithm that discovers process models from event logs. The tree representation ensures the soundness of the model. Moreover, as experiments show, it is possible to balance the different quality dimensions. Our genetic process mining algorithm is the first algorithm where the search process can be guided by preferences of the user while ensuring correctness.

I. INTRODUCTION

More and more events are being recorded. Over the last decade we have witnessed an exponential growth of event data. Information systems already record lots of transactional data. Moreover, in the near future an increasing number of devices will be connected to the internet and products will be monitored using sensors and RFID tags. At the same time, organizations are required to improve their processes (reduce costs and response times) while ensuring compliance with respect to a variety of rules. Process mining techniques can help organizations facing such challenges by exploiting hidden knowledge in event logs. Process mining is an emerging research discipline that provides techniques to discover, monitor and improve processes based on event data [2].

Starting point for process mining is an event log. All process mining techniques assume that it is possible to sequentially record events such that each event refers to an activity (i.e., a well-defined step in the process) and is related to a particular case (i.e., a process instance). Event logs may store additional information such as the resource (i.e., person or device) executing or initiating an activity, the timestamp of an event, or data elements recorded with an event (e.g., the size of an order). Typically three types of process mining are distinguished. The first type is process discovery where a process model is discovered using only the behavior observed in the event log. Another application of process mining is that of conformance checking. Here an existing process model is compared with an event log of the same process. This comparison shows where the execution of the process deviated from the process model. The third type is that of enhancement where a process model is extended or improved using information obtained from the event log.

In this paper we focus on process discovery. However, we would like to stress that process discovery is only the starting point for other types of analysis. After linking events to process model elements it becomes possible to check conformance, analyze bottlenecks, predict delays, and recommend actions to minimize the expected flow time.

To illustrate the notion of process discovery see Figure 1a-b. Based on the event log shown in Figure 1a, we can discover the Petri net shown in Figure 1b. For simplicity we use a rather abstract description of the event log: process instances are represented by sequences of activity names (traces). For example, there are 30 cases that followed trace abde, 38 cases that followed trace acde, 20 cases that followed trace adbe, and 12 cases that followed acde. This small event log consists of 400 events describing 100 process instances. There are 50 events corresponding to the execution of activity b.

Here we abstract from additional information such as the person executing or initiating an activity, the timestamp of an event, and associated data elements. The Petri net shown in Figure 1b describes a process model able to explain the observed behavior.

When using Petri nets to describe process models, the search space consists of all possible Petri nets. However, even when we ignore the event log, we can identify Petri nets that are clearly undesirable. Figure 1c-d shows two additional candidate models. Model $N_2$ has two potential deadlocks.
After executing $a$ and $b$ we reach the state with just a token in place $p2$. Transition $e$ is not enabled because there have to be tokens in both input places ($p2$ and $p3$) in order for $e$ to occur. Hence, $N_2$ gets “stuck” after executing the partial trace $ab$. A similar deadlock is encountered after executing $ad$. Only after executing partial trace $ac$, transition $e$ becomes enabled and the process can successfully terminate with a token in end.

$N_3$ in Figure 1d has another problem. It is possible to execute trace $abe$ that puts a token in place end. However, a token is left in $p2$. Although the process seems to have completed (token in end), it is still possible to execute $d$. Whereas $N_2$ was unable to replay the event log in Figure 1a, $N_3$ is able to replay the event log but none of the 100 cases reach the desired final state with just a token in place end.

The anomalies illustrated by $N_2$ and $N_3$ in Figure 1c-d are not specific to Petri nets. Any of the main business process modeling languages (EPC, BPMN, UML, YAWL, etc.) [17] allows for deadlocks, livelocks, and improper termination. These anomalies exist independent of the event log, e.g., the event log is not needed to see that $N_2$ has a deadlock. Nevertheless, most process discovery techniques consider such incorrect models as possible candidates. This means that the search space is composed of both correct and incorrect models. It is not easy to limit the search space to only correct models. For common notations such as Petri nets, EPCs, BPMN, UML activity diagrams, and YAWL models it is only possible to check correctness afterwards. Note that deadlocks and livelocks are non-local properties. Therefore, it is very hard to ensure soundness during construction.

Therefore, we propose to use process trees for process mining. Figure 1e shows an example of such a tree. The root node of the process tree is a sequence node (seq) with three children. This means that $a$, the subtree with root and, and $e$ are executed sequentially. The and subtree executes its two children in any order. It will therefore execute $d$ followed or preceded by either $b$ or $c$. Hence, the process tree is able to reproduce the event log and is trace equivalent to $N_1$. Process trees such as the one shown in Figure 1e cannot have any of the anomalies mentioned before (deadlocks, livelocks, etc.) because of their block-structure [8].

Even when unsound process models are not considered, there are still multiple process models that can describe the behavior observed in an event log. Figure 2 shows two process trees that also describe the behavior in the event log of Figure 1a. The process tree shown in Figure 2a consists of a loop containing the choice between all possible activities. This process tree can produce any trace consisting of the activities $a$ to $e$. This type of model is called a ‘flower’ model since the Petri net representation looks like a flower [12]. Because the process tree of Figure 2a can produce (many) more traces than are recorded in the event log, the process model is too generic, i.e., the model is “underfitting”. Another example is shown in Figure 2b. This process tree simply enumerates all traces in the log. This may seem reasonable for this small event log.

However, for event logs with thousands of different traces this is infeasible and results in a so-called “overfitting model”.

The two process trees in Figure 2 show that besides soundness various subtle quality aspects play a role. As shown in Figure 3a, four quality dimensions can be identified for process discovery [2], [12]. The (replay) fitness quality dimension describes to which extent a process model allows for the behavior observed in an event log. Both process trees in Figure 2 are able to replay the behavior of the eventlog. The simplest model that can explain the behavior seen in the log is the best model (Occam’s Razor), e.g., based on simplicity, we may prefer the flower tree (Fig. 2a) over the other two trees (Fig. 1e and Fig. 2b). The precision quality dimension is related to the desire to avoid “underfitting”, i.e., the flower tree allows for behavior unrelated to the observed behavior (traces like bbcb and eeaa are allowed in Fig. 2a). The generalization dimension is related to the desire to avoid “overfitting”. In general it is undesirable to have a model that only allows for the observed traces. Remember that the log contains only example behavior and that other possible traces may not have been seen yet. This dimension cannot be illustrated using the small event log in Figure 1a. However, consider a process with 10 parallel activities allowing for 3,628,800 (10!) different traces, or a process with a loop allowing for infinitely many different traces. Although we would like to discover such models, we will never be able to find logs that exhaustively list all possible traces.

The desired balance between the four quality dimensions depends on the properties of the event log and the purpose of the process model, e.g., for events logs with a lot of variability one gets a spaghetti-like model if one insists on a perfectly fitting precise model. Different discovery algorithms tend to focus on a particular quality dimension, e.g., the ILP Miner [16] guarantees “perfect replay fitness” but often results in a very complex process model whereas the Fuzzy Miner [7] is designed to produce comprehensible, simpler process models, sacrificing replay fitness.

Process discovery can be seen as a search process, i.e., given an event log, search for the model that describes the observed behavior ‘best’. One of the first requirements for the process models is that they are sound, see Figure 3b. However, this still leaves a large number of candidates to be investigated. For example, if an event log consists of 10 activities there are $4.6 \cdot 10^{15}$ possible process trees [3]. Which of these sound
process trees is ‘best’ can then be expressed using the four quality dimension as shown in Figure 3a. However, it is not feasible to inspect all possible process trees.

In earlier work, we used genetic algorithms to discover process models [4], [10]. However, these algorithms suffered from the problem that the majority of process models considered during the search process has anomalies such as deadlocks, livelocks, and improper termination. Using process trees as a new representational bias we limit the search space of the genetic algorithm. And by implementing all four process model quality dimensions we can emphasize desired aspects of the process tree.

One of the main benefits of genetic algorithms is their flexibility. A genetic algorithm applies random modifications to candidates, in our case process trees. The quality of each candidate is then calculated. Since better candidates survive until the next round of changes, the chances of finding an even better candidate improve. Although finding the best candidate is not guaranteed, good candidates are often returned. The flexibility of this approach allows for the discovery of virtually any process tree, guided by the fitness definition.

The remainder of this paper is organized as follows. Section II introduces the process trees used to represent sound process models. Section III introduces a new genetic algorithm that uses process trees and steers for different quality dimensions. Experimental results evaluating our approach are presented in Section IV. Section V concludes the paper.

II. PROCESS TREES

In this paper we use a representational bias called process trees. The main property of this representation is that all process trees correspond to sound models [8].

A process tree is a directed connected graph without cycles. A node \( V \) in the graph is either a branch node or a leaf node. Each leaf node represents an activity from the collection of activities \( \mathcal{A} \). Each branch node, or operator node, has one or more children. These children can be other operator nodes or leaf nodes. The labeling function \( \ell \) assigns each operator node an operator from \( \mathcal{O} \) and each leaf node an activity from \( \mathcal{A} \). Currently, we have defined operators for the sequence (\( \rightarrow \)), exclusive choice (\( \times \)), parallel (\( \land \)), or (\( \lor \)), and loop (\( \bigcirc \)) constructs. The first three operators cover all five basic Control Flow Patterns [1]. All operators require at least two branches, except for the loop operator which only takes one branch. Each of the operators can be translated to parts in other well-known process modeling languages. Figure 4 shows the translation of each operator to a section of a Petri net. For the sequence operator the order of the subtrees is important. Therefore, the children of an operator node are ordered using a sorting function \( s \). All operators, except the sequence operator, represent both the split and the join construction in other process modeling languages. Therefore, the process tree always describes a block-structured process model [13], [14]. Furthermore, as there is no restriction on the number of leaf nodes, the operators allow to express any event log by representing an activity more than once.

III. GENETIC PROCESS DISCOVERY

As discussed in Section I we propose the use of a genetic programming algorithm for the discovery of process models from event logs. Evolutionary algorithms have been applied to process mining discovery before in [4], [10]. Our approach follows the same high-level steps as [6], which are shown in Figure 5. Main improvements with respect to [4], [10] are the internal representation and the overall fitness calculations. By using process trees as our internal representation we only consider sound process models. This drastically reduces the search space and therefore improves the performance of the genetic algorithm. Furthermore, we can apply standard tree change operations on the process trees to evolve them further. Finally, in our overall fitness calculation we consider all four quality dimensions for process models: replay fitness, precision, generalization and simplicity. The user can specify
the relative importance of each dimension beforehand. The genetic algorithm will then favor those candidates that have the correct mix of the different quality dimensions.

In general, our genetic algorithm follows the process as shown in Figure 5. The input of the algorithm is an event log describing observed behavior. In the initial step a population of random process trees is generated where each activity occurs exactly once in each tree. Next the four quality dimensions are calculated for each candidate in the population. Using the weight given to each dimension the overall fitness of the process tree is calculated. In the next step certain stop criteria are tested such as finding a tree with the desired overall fitness. If none of the stop criteria are satisfied, the candidates in the population are changed and the overall fitness is again calculated. This is continued until at least one stop criterion is satisfied and the fittest candidate is then returned as output of the genetic algorithm.

Our genetic algorithm has been implemented as a plug-in for the ProM Framework [15]. Details with respect to the overall fitness calculations and the genetic operators are described in the remainder of this Section.

The goal is to find a process model that describes the observed behavior ‘best’. This is tested by calculating an overall fitness value for each candidate. We calculate the overall fitness of a candidate by combining the values of each of the four individual quality dimensions. Each individual quality value ranges from 1 (worst) to 0 (best). The overall fitness is the weighted mean of these four values.

The most important dimension is that of replay fitness. The replay fitness indicates how well the event log can be replayed on the process model. The replay fitness is calculated by applying techniques from [5] on process trees. The basic idea of the approach is to find the optimal alignment between the event log and the process model. This alignment relates the activities of the process tree with the events in the event log. A mismatch in the alignment occurs when the execution of an activity in the process model cannot be matched with an event in the trace or vice versa. For these mismatches costs are calculated. The algorithm finds for each trace in the event log the alignment with the least costs. The replay fitness of the event log is the average cost per trace normalized to a value between 0 and 1.

The quality dimensions precision and generalization are calculated using the alignment and additional information calculated by the replay fitness. Precision is the fraction of states visited in the alignment with respect to all states that were enabled in the state space [11]. If only a small fraction of states is used, then the process tree allows for much more behavior than is observed. The process tree is thus underfitting which results in a poor precision.

Generalization can be interpreted as the probability that a new, unseen, trace exhibits behavior not possible according to the model. This is difficult to quantify as it deals with unseen behavior. However, if all nodes are visited frequently, then the likelihood of encountering new behavior is smaller than if most nodes have only been visited a few times. Therefore, we characterize generalization as follows: \( \frac{\sum_{\text{nodes}} \text{executions of } n}{\text{#nodes}} \). A value close to 0 indicates that all nodes are visited often and that the model is not overfitting. A value close to 1 indicates that new unseen traces are likely to deviate for the model.

Our simplicity metric combines multiple aspects. First, ideally, each activity occurs exactly once in the process tree. Duplicating or leaving out activities makes the process model more difficult to understand. Second, the fewer alternations between operator types, the easier the model is to read. And third, \( \circ \) and \( \lor \) operators are more difficult than \( \rightarrow, \times \) and \( \land \). Therefore, simplicity is calculated as follows: first it is counted how many activities are duplicated and missing in the process tree, with respect to the event log. Then the number of alternations of operator types between two connected operator nodes are counted. Finally, the use of \( \lor \) and \( \circ \) operators is punished, where loops get a punishment of 1 and OR’s get a punishment of 2. The final simplicity score is calculated by adding the number of missing and the number of duplicated activities, the number of operator alternations and the punishment for loops and OR’s. This number is then divided by the total number of nodes in the process tree.

Now that the overall fitness for each candidate in the population is calculated, the stop criteria are verified as can be seen in Figure 5. There are three stop criteria that will stop the algorithm as soon as one of them is satisfied. The first criterion is reaching a minimum fitness level. This means that when a fit enough process tree is discovered, the algorithm is stopped and that candidate is returned. The second criterion is a test if the best candidate did not change for a specified number of generations. This indicates that the algorithm was unable to find a better candidate for a specified number of generations. Therefore, the existence of a better candidate is uncertain thus the search can be stopped. The final stop criterion is reaching a maximum number of generations.

For each generation, new candidate models are computed. First, the best candidates of the current generation are put in
an elite group. These candidates are simply copied to the next generation and will remain constant. This is done to prevent relapses in overall fitness and to keep the best candidates for the future. The remainder of the population for the next generation is created by randomly selecting candidates from the current generation on which change operations are applied. Preference is given to fitter candidates using a tournament selection. The candidates that are changed will, together with the elite group of the current generation, form the new generation. It is important to note that candidates can be selected multiple times and candidates in the elite group can also be selected but will be copied before they are changed.

We define three different types change operations: random creation of new process trees, crossover and mutation. The first, random creation, is a rather extreme change operation since it completely replaces an existing process tree with a randomly created new one. This operator is used to replace the worst candidates in a population with fresh trees. Random trees are generated in such a way that each activity occurs exactly once. For the root a random operator type is chosen. The list of activities is randomly spread over the left and right subtrees. For each of these subtree roots, again a random operator is chosen and the remaining activities are divided. This is repeated until the activity list for a subtree contains only one activity and thus becomes a leaf node.

Subtree crossover is applied on two process trees and swaps two randomly selected subtrees between the parent trees.

The third change operation, that of mutation, consists of different types of small mutations on process trees. We use three mutations: node mutation, subtree removal and node addition. In the case of node mutation (or point mutation) a single node is changed. For operator nodes this means that the operator type is changed and sometimes the order of the children is also changed. In the case of a leaf node, the activity it represents is changed. Subtree removal (or shrink mutation) means that one node is randomly selected and removed from the tree, together with all its children. Node addition means that a random leaf node is created and added to an existing operator node. After applying these change operations the process tree is reduced without changing its behavior. For instance, a loop containing a loop does not add any behavior.

The ratio settings for the random- and crossover operations indicate to which proportion of the population that change operation is applied. We make sure that we never consider the same tree twice by caching trees. A tree is mutated until a tree is encountered that was not seen before. This of course does not apply to the group of elite candidates which is maintained between generations. The size of this elite group, relative to the total population size, is set by the elite ratio.

IV. EXPERIMENTS

Three experiments were conducted to evaluate our new discovery algorithm. The first experiment varies a selection of settings to investigate the optimal settings for the most important parameters. These settings are used in the other two experiments. In the second experiment the algorithm runs for more generations. In this experiment we investigate the long term behavior of the average and best overall fitness values and the tree size. In the third and last experiment we changed the weights given to the four quality dimensions. The goal of this experiment is to see the effects of each quality dimension on the resulting process model. The settings used in each experiment are shown in Table I.

As input for our experiments we used three process models and corresponding event logs from [9]: \textit{a12}, \textit{a12All5pcNoise} and \textit{HerbstFig6p34}. These processes are of medium size (around 14 activities) and have been used to evaluate many other process mining algorithms. The \textit{a12} process model only contains sequence, exclusive choice and parallel constructs, and can be represented by a tree without duplicating tasks. The \textit{a12All5pcNoise} is the same process model as \textit{a12} but in the event log noise is added. For 5\% of the traces noise is introduced by removing, adding or swapping events. The \textit{HerbstFig6p34} is a different process model that also contains loops and duplicate activities.

A. Experiment 1: Finding the Best Settings

As can be seen in Table I, in the first experiment we applied 324 setting combinations. Each combination was tested on the three event logs which means that in total 972 experiments were run. The ratios for random tree insertion into the population and the elite- and crossover rates were tested with values of 0.1, 0.2 and 0.3. The population size was tested with sizes of 10, 50, 100 and 200. Finally, the steady state count was tested with values of 10, 20 and 100.

Figure 6 shows the average overall fitness value of the best candidate for different population size and steady state values. The error bars indicate the 95\% confidence interval.

\begin{table}[h]
\centering
\begin{tabular}{ |c|c|c|c| }
\hline
Setting & Experiment 1 & Experiment 2 & Experiment 3 \\
\hline
Log \textit{a12} & X & X & X \\
Log \textit{a12At5pcNoise} & X & - & - \\
Log \textit{HerbstFig6p34} & X & - & - \\
\hline
Population size & 10 / 50 / 100 / 200 & 200 & 200 \\
Max. Generations & 100 & 10,000 & 1,000 \\
Steady states & 10 / 20 / 100 & 10,000 & 1,000 \\
Target overall fitness & 0.00005 & 0.00005 & 0.00005 \\
Elite rate & 0.10 / 0.20 / 0.3 & 0.3 & 0.3 \\
Random rate & 0.10 / 0.20 / 0.3 & 0.3 & 0.3 \\
Crossover rate & 0.10 / 0.20 / 0.3 & 0.1 & 0.1 \\
Replay fitness weight & 1 & 1 & 1 / 10 / 100 \\
Precision weight & 1 & 1 & 1 / 10 / 100 \\
Generalization weight & 1 & 1 & 1 / 10 / 100 \\
Simplicity weight & 1 & 1 & 1 / 10 / 100 \\
\hline
\end{tabular}
\caption{Settings per experiment}
\end{table}
Figure 6a shows the effects on the overall fitness of the fittest candidate of changing the population size. A larger population means that in each generation more candidates are created and considered. It is therefore not surprising that a larger population size in general results in the discovery of better process trees. In Figure 6b the influence of the number of steady states on the overall fitness of the best candidate is shown. The steady state setting stops the algorithm if for the specified number of generations the best candidate did not change. Here again it is clear that performing more generations while the fittest candidate does not change, results in the possible discovery of better candidates.

Figure 7 shows the average overall fitness of the best candidate for each ratio setting. The error bars indicate the 95% confidence interval of the average. Using the information from Figures 6 and 7, the best settings for the other experiments are determined. For the population size this means that we use a size of 200 candidates in the next experiments. The number of steady states was set equal to the maximum number of generations to let the algorithm run as long as possible (cf. Table I). In the next experiments we used the slightly better values of 0.3 for the random- and elite ratios and 0.1 for the crossover ratio (cf. Table I).

B. Experiment 2: Long Run Behavior

The second experiment investigates the behavior when the algorithm is run for more generations. One would expect that tree size and average overall population fitness remain fairly constant, even when the algorithm runs for more generations. The top line in Figure 8a shows the evolution of the average overall fitness of the whole population. The grey dots in this figure show averages of individual generations while the line shows the overall trend. The overall fitness drastically improves in the first 150 generations. The average overall fitness of the population then gets slightly worse, which is mainly caused by the fact that trees that were already considered are not added to the population any more. We do however see a stabilization of the overall fitness over time. The bottom line in Figure 8a shows the overall fitness of the best candidate. As expected this improves drastically in the first 150 generations after which it stabilizes but does keep slightly improving until generation 4,000.

Figure 8b shows the average tree size over the generations. The process tree for this process model should ideally contain 27 nodes if no activities are duplicated or left out. Here we also see a stabilization after roughly 150 generations. This shows that the simplicity quality dimension ensures that trees do not shrink or grow drastically over time unless other quality dimensions are improved significantly.

Based on the results we can conclude that, although the best candidate keeps getting better over time, the main improvements occur in the first 150 generations. And, because we take the growth and shrinking of process trees into account during our overall fitness calculation, the average tree size remains relatively constant.
C. Experiment 3: Steering for Quality

Using the settings obtained in experiments 1 and 2, the quality dimensions are tested in this experiment. The weights for each of the four quality dimensions are set to 1, 10 and 100 (cf. Table I). For each of the 81 weight combinations we let the algorithm run for 1,000 generations with a population size of 200.

Table II shows the details of the original tree and the trees that were discovered by our genetic algorithm. The original process tree is shown in Figure 9. For each of the quality dimensions we applied combinations of weights of 1, 10 and 100. Let us first look at the process trees that were focused on one of the four quality dimensions. The dimension on which was focused had a weight of 100 and a weight of 1 for the other dimensions.

In the case where we steer for replay fitness, the best tree starts with four sequential operators. There is also a large block of XOR-operators in a loop. This construct allows the arbitrary execution of the activities in the loop of XOR’s, comparable to the flower model. This causes both a good replay fitness and a good precision. Generalization however is relatively bad when compared to the other three process trees that focused on another dimension. Simplicity is in line with the other discovered trees. Punishment is mainly caused by the presence of the loop operator and the alternation from the sequence to the loop operator.

The tree where the precision dimension was set to a weight of 100, while all the others had a weight of 1, contains only sequence operators. The replay fitness of this tree, focussed on precision, is relatively low with a value of 0.276, since the process tree only allows for a single trace. The original process tree however contains choices and parallelism resulting in different traces. Although this is bad for the replay fitness, since choices and parallelism cannot be correctly replayed, from a precision point of view the process tree is perfect. Furthermore, the simplicity of this tree is perfect since it only contains sequence operators.

Focusing on generalization does not result in a tree with a perfect score of 0 for generalization, which is caused by the definition of the metric. Since generalization is calculated by dividing by the number of nodes in the tree, adding a loop operator increases the number of nodes which improves the generalization. Multiple loop nodes in succession are removed by our behavior reduction algorithm. The resulting tree contains mainly sequence operators and a small block of two activities in a loop with an XOR-operator. This results in a bad replay fitness caused by the limited number of traces the process tree can produce. Precision is perfect since the process tree does not allow for much behavior. Simplicity is not that good which is caused by the alternation of operator type and the introduction of a loop operator.

Finally, the tree where we focus on simplicity results in a tree with only sequence operators. This results in a bad replay fitness, but since the model only allows for a single trace the precision is perfect. Because the tree only contains sequence operators, and all activities exactly once, the simplicity is perfect. Most of the quality dimensions of this process tree have the same values as the process tree that is discovered for the precision dimension. The main difference is a worse replay fitness value, because the order of the activities is worse.

Next, we tried to rediscover the original process model. Looking at the process tree that was used to generate the event log, it can be seen that it does not have a perfect overall fitness. Although the replay fitness and precision dimension are perfect, the model is not very good in the generalization and simplicity dimensions, according to our notions. Since the model contains alternating operator types the simplicity is relatively bad. The good but not perfect generalization score can be explained by our way of calculation that causes generalization to never become perfect.

Figure 10 shows the process tree that was discovered using weights of 100 for replay fitness and precision, and 1 for
generalization and simplicity. These weights are chosen guided by the scores for the quality dimensions of the original process tree. The goal was to see if we could rediscover the original process tree. The characteristics of the discovered tree are shown in the third column in Table II. The discovered process tree has a perfect replay fitness and precision, and the generalization and simplicity dimensions are better than the original model. Although the process trees are not similar, the discovered process tree has better values for each quality dimension than the original process tree. This is mainly caused by a relatively large block of XOR-operators within a loop. This results in a high replay fitness without punishment in precision. Generalization is also good since the loop and its containing XOR-operators are executed multiple times. Furthermore, simplicity is good because of the large part of the process model that only contains XOR-operator nodes, despite the punishment for the loop operator. The algorithm was able to discover the exclusive choice between activities B and C, the precedence of these activities by A and the sequence of L and END.

Looking at the settings used for this experiment we can calculate the total number of process trees created and evaluated by the algorithm. In the first generation 200 process trees are randomly created. In the following 9,999 generations 140 new process trees are created by applying one of the change operations. The 60 fittest process trees are placed in the elite group and are not changed in the next generation. The total number of process trees considered in this experiment therefore is 140,060. In total there are $4.3 \cdot 10^{24}$ process trees where each of the 14 activities occurs exactly once [3]. However, this is an underestimation since in our algorithm activities can occur multiple times. Furthermore, we did not consider the loop operator in the calculation. This demonstrates that the genetic algorithm is able to discover a high quality process tree while only visiting a small fraction of the total search space.

It should be noted that the replay fitness plays an important role in the calculation of the other dimensions. In the calculation of the generalization and precision dimensions the alignment of the replay fitness is used. A bad replay fitness thus has effects on these dimensions. Therefore, the replay fitness should be the main criterion. Calculation of the generalization and precision dimensions are only meaningful if the replay fitness is good.

This last experiment shows that emphasizing one of the quality dimensions results in a process tree that scores very well in that dimension. Furthermore, emphasis on a selection of the quality dimensions result in process trees that have a mixture of those qualities. We were unable to rediscover the original process tree for the given event log. This is mainly caused by the implementation of the quality dimensions. The current implementation favors certain structures, such as a loop of choices (e.g. the flower model), too much. By further refining the metrics for the different quality dimensions it is possible to steer the search process better and rediscover Figure 9.

V. Conclusion

The genetic process discovery algorithm presented in this paper is the first algorithm that ensures correctness while incorporating all four quality dimensions. The user can indicate the importance of all dimensions.

Future work will focus on the refinement of the four quality metrics and the genetic operators used. Moreover, the approach will be used to compare the processes of the ten Dutch municipalities involved in the CoSeLoG project.

Acknowledgment

This research has been carried out as part of the Configurable Services for Local Governments (CoSeLoG) project (http://www.win.tue.nl/coselog/).

References

[1] W.M.P. van der Aalst. Workflow Patterns. In L. Liu and M. Tamer Özsu, editors, Encyclopedia of Database Systems, pages 3557–3558. Springer-Verlag, Berlin, 2009.
[2] W.M.P. van der Aalst. Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer-Verlag, Berlin, 2011.
[3] W.M.P. van der Aalst, J.C.A.M. Buijs, and B.F. van Dongen. Improving the Representational Bias of Process Mining Using Genetic Tree Mining. In SIMPDA 2011 Proceedings, Springer-Verlag, 2011.
[4] W.M.P. van der Aalst, A.K. Alves de Medeiros, and A.J.M.M. Weijters. Genetic Process Mining. In G. Cardo and P. Darondeau, editors, Applications and Theory of Petri Nets 2005, volume 3536 of Lecture Notes in Computer Science, pages 48–69. Springer-Verlag, Berlin, 2005.
[5] A. Adriansyah, B. van Dongen, and W.M.P. van der Aalst. Conformance Checking using Cost-Based Fitness Analysis. In IEEE International Enterprise Computing Conference (EDOC 2011), pages 55–64. IEEE Computer Society, 2011.
[6] A.E. Eiben and J.E. Smith. Introduction to evolutionary computing. Springer Verlag, 2003.
[7] C.W. Günther and W.M.P. van der Aalst. Fuzzy Mining: Adaptive Process Simplification Based on Multi-perspective Metrics. In International Conference on Business Process Management (BPM 2007), 2007.
[8] Oliver Koppen, Daniel Martin, Daniel Wutke, and Frank Leymann. The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages. Enterprise modelling and Information Systems Architecture, 4(1):3–13, 2009.
[9] A.K. Alves de Medeiros. Genetic Process Mining, PhD thesis, Eindhoven University of Technology, 2006.
[10] A.K. Alves de Medeiros, A.J.M.M. Weijters, and W.M.P. van der Aalst. Genetic Process Mining: An Experimental Evaluation. Data Mining and Knowledge Discovery, 14(2):245–304, 2007.
[11] J. Moran-Gama, A. Adriansyah, J. Carmona, R. van Dongen, and W.M.P. van der Aalst. Alignment Based Precision Checking in Process Mining. In SAC2012 Proceedings, Riva del Garda, Italy, 2011.
[12] A. Rozinat and W.M.P. van der Aalst. Conformance Checking of Processes Based on Monitoring Real Behavior. Information Systems, 33(1):64–95, 2008.
[13] Guido Schinn. Mining most specific workflow models from event-based data. In Proceedings of the 2003 international conference on Business process management, BPM’03, pages 25–40, Berlin, Heidelberg, 2003. Springer-Verlag.
[14] J. Vanhatalo, H. Völzer, and J. Koehler. The Refined Process Structure Tree. Data and Knowledge Engineering, 68(9):793–818, 2009.
[15] H.M.W. Verheooq, J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. XES, XESame, and ProM 6. In P. Soffer and E. Proper, editors, Information Systems Evolution, volume 72 of Lecture Notes in Business Information Processing, pages 60–75. Springer-Verlag, Berlin, 2010.
[16] J.M.E.M. van der Werf, B.F. van Dongen, C.A.J. Hurkens, and A. Serebrinik. Process Discovery using Integer Linear Programming. Fundamenta Informaticae, 94:387–412, 2010.
[17] M. Weske. Business Process Management: Concepts, Languages, Architectures. Springer-Verlag, Berlin, 2007.