
Production Systems and Information Engineering
Volume 12 (2), pp. 40–50.
https://doi.org/10.32968/psaie.2024.2.3

EFFICIENCY ANALYSIS OF FUZZY LOOP DETECTION METHODS

LÁSZLÓ KOVÁCS
University of Miskolc, Hungary Institute of Information Technology
kovacs@iit.uni-miskolc.hu

Abstract: Automated Process Discovery is aimed at determining the most fitting business process schema model from the event logs. One of the most difficult schema components is the loop structure and current process mining methods lack efficient loop mining. Our paper presents two novel approaches to fuzzy loop mining, where in fuzzy loops, the repeating kernel may have different variations. On of the proposed methods uses item-wise direct comparisons, while the other one is based on CNN neural network architecture. The performed evaluation tests show which are the benefits or drawbacks of the different loop mining approaches providing a guide for the implementation of efficient schema mining frameworks.

Keywords: *Robotic Process Automation, Robotic Process Mining, CNN neural network*

1. Introduction

Automating repetitive business and office tasks is an essential tool for increasing the efficiency of corporate processes. Automation reduces execution time, error rates, and overall operational costs. The goal of Robotic Process Automation (RPA) is to achieve high-level automation of business processes by creating and running software robots (bots) that perform the work processes of employees. These robots handle high-volume, repetitive, and template-based tasks with greater accuracy and efficiency than humans [1].

The concept of RPA is an umbrella term that encompasses all IT solutions that interact with other IT systems as humans would [2]. While some interpretations suggest that RPA systems do not integrate internally with the managed system but control it through its existing user interface, this interpretation is not entirely accurate, as demonstrated by the approach of the Gartner Institute [3]. According to Gartner's definition, RPA solves control through a combination of classic user interfaces and direct program-level API interfaces.

Robotic Process Mining (RPM) is a new research area closely related to RPA, aiming to discover automatable processes using machine learning tools. Developing process mining methods is still a largely unexplored problem area. The field closely related to robotic process mining is Automated Process Discovery (APD), which

involves determining the most fitting business process model through data analysis of the logs from the examined processes. It is crucial to ensure the completeness of the analysis, addressing exceptional cases and errors. The main goal of APD processes is to produce a model that supports the control and planning tasks of processes. The primary criteria for measuring goodness [4] are: a) the ability to generate the recorded operational steps, b) the ability to generalize the examined processes and predict similar processes, and c) the ability to identify non-conforming processes.

The most general formalism of process models includes automata, state transition systems, and operational flow models. The model has three main elements:

- a set of states (S), with special emphasis on the starting and ending states,
- state transitions (T),
- actions (A).

There are several methods available for developing operational flow models; the following briefly reviews the more common methods.

Petri Nets. The most widespread process modelling tool [5]. In the model, there are two types of nodes:

- normal state places (place),
- state transitions (transition).

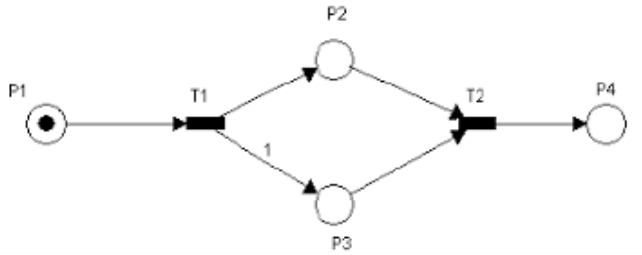

Figure 1. Petri net formalism

Connections can only link normal state and state transition nodes. At the transitions, we can thus talk about input and output places. In addition to normal states and transitions, status indicator elements known as tokens also appear. The tokens indicate the fulfilment of the associated state by their presence. A specific condition is required for the transition to occur in the model. A transition only happens (fires) if there is at least one token in every input place. During the transition, tokens move to the output places. The law of token conservation does not necessarily apply here. We can also assign capacity values to the edges, indicating the number of tokens involved in the transitional transformation.

BPMN (Business Process Model Notation) [6]. This modelling language, similar to the YAWL system, is widely used in industry, primarily due to its functional richness. The main groups of building elements are:

- EVENT descriptors
- ACTIVITY descriptors

- GATEWAY descriptors
- CONSTRAINT descriptors

The ACTIVITY group contains the structure of elementary events, including the LOOP unit.

Process Tree [7]. General graph-based methods for process descriptions do not necessarily provide valid descriptions and cannot effectively control higher-level forms. In contrast, process tree-based descriptions use a structural, containment-based hierarchical model, resulting in a less flexible but more controlled model. In the tree, internal nodes represent structural units, while leaves denote elementary operations.

In the more complex modelling languages, the loop (LOOP) structure plays a prominent role. However, according to analyses, uncovering and inducing the LOOP structure is not a simple task, and existing schema discovery procedures are not fully capable of handling it. In our current research, we analyze the reasons for the difficulty in uncovering LOOP structures and the loop-detecting algorithms.

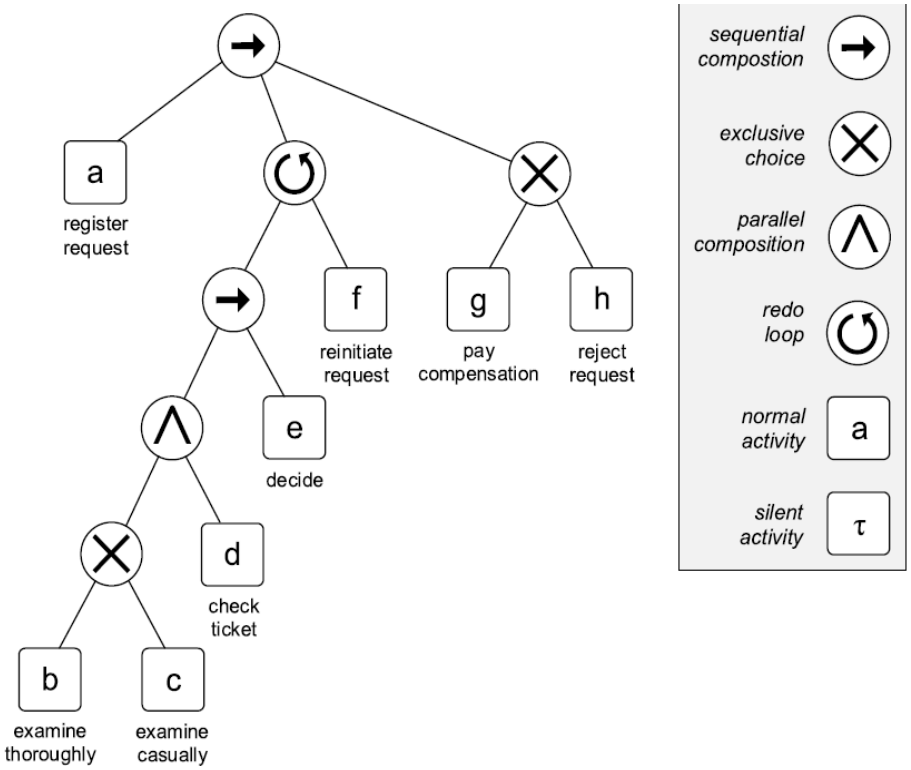

Figure 2. Process tree formalism (source: [7])

2. Complexity of fuzzy loop mining

The loop structure means the repetition of some event sequences. Having an alphabet Γ, the simple loop L is defined as

$$L = s\, s \ldots s$$

where $s \subseteq \Gamma$. The sequence s is the kernel of the loop.

Beside the exact loop, we can define fuzzy loops where there is no fix kernel, it may have some limited variations:

$$L = s_1\, s_2, \ldots, s_m$$

where

$$d(s_i, s_j) \leq \varepsilon$$

In the case of complex problem domains, the kernel *s* is not just an elementary sequence, but a complex structure. The complex loop is defined with

$$L = X_1\, X_2, \ldots, X_m$$

where X_i denotes a sequence, loop or XOR structure units.

The loops are local properties inside a sequence. We have exact definitions for both the fuzzy loops, which enable us to develop a loop detection algorithm. On the other hand, the direct implementation of the loop mining is based on a brute force approach. Thus there is a need for more efficient methods, but this problem domain has some inherent difficulties:

- there are too large number of options to be tested;
- the internal structure of a loop can be very complex (XOR branches or inner LOOP);
- there is no standard algorithms and code implementation for loop detection task;
- there is no machine learning approach for this problem domain;
- there are very few investigation in the literature on the fuzzy loop and complex loops.

Based on these challenges, our paper focuses on the presentation of some possible solution alternatives and on the evaluation of the proposed algorithms. The investigated algorithms are implemented in Python using the TensorFlow/Keras framework.

3. Algorithms for fuzzy loop

3.1. Brute force algorithm

In the case of brute force approach, we test all possible positions and kernel sets. Taking a starting position p_s in the sequence, we test all possible starting kernels which are substrings starting at p_s. All possible variants are saved into a queue of candidates (Q). In the next phase, we process the current candidates (k_c) pulled from

Q. Next, we try to extend k_c with another new kernel resulting a new candidate (k_n) which is saved in Q. Only those loop candidates are stored in Q which meet the similarity constraints, i.e. the edit distance between any two kernels is less than a given threshold. The result set contains only those loops which are maximal, i.e they can not be extended with new kernels.

The simplified pseudocode of the brute force algorithm:

```
def find_fuzzy_loops_A (seq, dlim):
  # seq_ imput sequence
  # dlim : distance limit releative length

   for p1 in range(len(seq)-1):
      s2 =seq[p1:]
      cloops = []
      for p in range(1, int(len(s2)/2+1)):
         w = s2[:p]
         cloops.append({"start":p1,"kernels":[w],"len":p})

      while len(cloops) > 0:
         citem = cloops.pop()
         p0 = citem["start"] + citem["len"]
         s3 = seq[p0:]
         for p in range(1,int(len(s3)/2+1)):
            w3 = s3[:p]
            db = 1
            for j in range(len(citem["kernels"])):
               wk = citem["kernels"][j]
               dv = levenshteinDistance(wk,w3)
               dl = min(int(dlim[0]*len(wk)),dlim[1])
               if dv > dl:
                  db = 0
            if db == 1:
               ncitem = copy_citem(citem)
               ncitem["kernels"].append(w3)
               ncitem["len"] = ncitem["len"] + p
               cloops.append(ncitem)
```

3.2. Method DC algorithm

The next investigated algorithm is a novel proposal to improve the efficiency of the brute force method. The method uses special storage units to optimize the processing costs. The main storage variables:

- Poss: hash table to store the occurrence positions for each character (sequence item)

- Reps: the list of hash tables to manage the distances between the positions with the same character

The algorithm first fill in the variables Reps and Poss. The main program loop processes the item (positions) of the sequence one by one. For the current position we check whether the current character and the previous character may belong to the same kernel or not. In order to speed up the similarity checking, we use a Hamilton-distance instead of the Levenstein distance. Similar to the brute-force approach, a starting kernel may have several adjacent kernels, thus the loop variants can be structured into a tree. During the candidate generation, the valid loop candidates must meet the predefined distance conditions. Only the maximal loops are returned in the results, thus smaller loops covered by other loops are removed from the result.

The simplified pseudocode of the algorithm:

```
def test_rep (seq, Mp, Md):

   N = len(seq)   # input sequence
   poss = dict()    # dictionary where does the given character occur in the list
   reps = [[] for _ in range(N)]   # the list of rep lengths
   kernels = []     # list of valid loops

   for p in range(N):    # loop on the start position
      for k0 in reps[p]:   # loop on possible kernel length
         k1 = k0
         dist = k1-1 # distance
         p1 = 1

         maxp1 = min (k0 + min (Md,int(k0*Mp)) + 1, N-p)
         while p1< maxp1:      # go forward to test the candidate kernels
            s = [ (abs(k-k1),k)  for k in reps[p+p1]]  # differences in gap
            if len(s) > 0:
               s.sort()
               k2 = s[0][1]
               if k2 == k1:
                  dist -= 1
               if k2 < k1:
                  dist ,k1= dist+ (k1-k2), k2
               if k2 > k1:
                  dist.k1  = dist + 1.k1+1
            if dist < dmin:
               dist = levenshteinDistance(seq[pmin-k0min:pmin],
seq[pmin:pmin+p1min+1])
               dmin = dist
```

```
    p1 += 1
kernels.append((dmin,pmin-k0min,pmin+p1min+1))
```

3.3. Method CNN algorithm

The neural networks are the dominating machine learning mechanism where a neural network can be considered as an universal function approximator. The main motivations to use NN for problem solving are

- no need to develop domain specific algorithm
- shorter development time
- existing standard universal tools
- integration with other neural network architectures

The application of NN in the different optimisation and pattern matching problems is hindered by the fact that NN is designed to optimize a lost function calculated for a given fixed size input [10]. In order to copy with complex problem domains, different new NN architectures were developed in the recent decades. While MLP [11] is for a function approximation with input of fixed size, the RNN [12] networks are able to process sequences and CNN [9] perform prediction on high dimensional cubes.

In our approach, we proposed a CNN-based solution based on the following components.

1. The sequence is first converted into a two dimensional square matrix describing the similarity between the items at the different positions in the sequence.
2. The kernels can be recognized as special patterns in the matrix.
3. The CNN is trained to discover these patterns.

The 2D representation matrix shows which section in the sequence are similar to each others. For example, taking the sequence s = “adogdogdughua”, we get the following matrix:

```
M matrix
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 1. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 1. 0. 0. 1. 0. 0. 0.]
 [0. 1. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 1. 0. 0. 1. 0. 0. 0.]
 [0. 1. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0.]
 [0. 0. 0. 1. 0. 0. 1. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

Figure 3. Sample similarity matrix

The main steps in building up the CNN matrix are summarized in the followings:

```
model = Sequential()
model.add(Conv2D(64, kernel_size=(3, 3), activation='linear',
                 input_shape=(in1, in2,1),
                 padding='same'))
model.add(LeakyReLU(alpha=0.1))
model.add(MaxPooling2D((2, 2), padding='same'))
model.add(
   Conv2D(128, (3, 3), activation='linear', padding='same'))
model.add(LeakyReLU(alpha=0.1))
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))
model.add(
   Conv2D(256, (3, 3), activation='linear', padding='same'))
model.add(LeakyReLU(alpha=0.1))
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))
model.add(Flatten())
model.add(Dense(256, activation='linear'))
model.add(LeakyReLU(alpha=0.1))
model.add(Dense(out, activation='relu'))

# cnn.compile(loss=keras.losses.binary_crossentropy,
# optimizer=keras.optimizers.Adam(),metrics=['mse'])
model.compile(loss=keras.losses.mean_squared_error,
              optimizer=keras.optimizers.Adam(),
              metrics=['mse'])
```

4. Comparison Tests

In the performed tests, we compared the execution time complexity of the presented algorithms. In order to get the $O(n^5)$ complexity, we performed the loop mining on sequences with different lengths. In the case of brute force algorithm, we got the execution times presented in *Table 1*.

Table 1. Time cost of the brute-force algorithm

sequence length	execution time [s]l
50	0.2
100	5.9
150	42.9
200	169.4
250	627.0
300	1214.1

Table 2. Time cost of the Method DC algorithm

sequence length	execution time [s]l
200	0.2
300	0.5
400	0.9
500	1.6
600	2.3
700	3.9
8000	6.1

In the case of the CNN-based solution, we measure the execution time both for training and prediction.

Table 3. Training time cost of the Method CNN algorithm

sequence length	execution time [s]l
50	12.9
100	38.3
150	90.3
200	173.1
250	287.0
300	488.2

Table 4. Prediction time cost of the Method CNN algorithm

sequence length	execution time [s]l
50	0.002
100	0.003
150	0.005
200	0.006
250	0.006
300	0.007

Regarding the accuracy of the loop prediction, we calculated item level accuracy as

$$acc = \frac{\#\text{ of correctly predicted items}}{\#\text{ of all items}}$$

The accuracy test results is summarized in the next table:

Table 5. Accuracy results of the methods

method	accuracy	recall
Brute-force	100%	100%
Method DC	96%	93%
Method CNN	81%	72%

5. Conclusion

We proposed two novel algorithms for detection of fuzzy loops, one of them is based on the usual direct comparison of the sequence items and the other one uses a CNN neural network to perform the loop prediction. In the case of CNN-based solution, the sequence is first converted into a similarity matrix which is the input of the CNN module. In the efficiency comparison tests, we also involved a brute force algorithm as a baseline method.

The test results can be summarized into the following conclusions:

1. The best accuracy is achieved by the brute-force method, but it has a very high execution cost, nearly $O(n^5)$.
2. The proposed Method DC provides an acceptable $O(n^{2.5})$ cost.
3. The training cost of CNN-based method is in $O(n^{2.3})$ while the prediction cost is very low $O(n)$.
4. The absolute time for CNN is very high, it is not worth to use CNN for a small number of predictions.
5. Another problem of CNN is that we should fix the sequence length for the training and prediction, it is not suitable to apply it for variable sequence length domains.
6. The brute force can be used if we require very high precision and the sequence length is a small value.
7. The CNN can be used for stream-based loop search when the length of the sequence is fixed.
8. The proposed Method A provides a good balance with a good accuracy and good time efficiency values.

References

[1] Aguirre, Santiago, and Alejandro Rodriguez (2017). Automation of a business process using robotic process automation (RPA): A case study. *Applied Computer Sciences in Engineering*. 4th Workshop on Engineering Applications, WEA, pp. 65–71. https://doi.org/10.1007/978-3-319-66963-2_7

[2] Van der Aalst, W. M. P., Bichler, M., and Heinzl, A. (2018). Robotic process automation. *Business & information systems engineering*, 60, pp. 269–272. https://doi.org/10.1007/s12599-018-0542-4

[3] Tornbohm, C., Gartner (2017). *Market guide for robotic process automation software*. Report G00319864.

[4] Russell, N., van der Aalst, W. M. P., and Ter Hofstede, A. (2016). *Workflow patterns : the definitive guide*. MIT Press.

[5] Jensen, K., and Kristensen, L. M. (2015). Colored Petri nets: a graphical language for formal modeling and validation of concurrent systems. *Communications of the ACM*, 58, 6, pp. 61–70. https://doi.org/10.1145/2663340

[6] Recker, Jan et al. (2009). Measuring method complexity: UML versus BPMN. *Proceedings of the Fifteenth Americas Conference on Information Systems*. Association for Information Systems, pp. 1–9.

[7] Leemans, Maikel, Van Der Aalst, W. M. P., and Van Den Brand, M. G. J. (2018). Hierarchical performance analysis for process mining. *Proceedings of the 2018 International Conference on Software and System Process*, pp. 96–105. https://doi.org/10.1145/3202710.3203151

[8] Asadollahfardi, Gholamreza (2015). Artificial neural network. *Water Quality Management: Assessment and Interpretation*, pp. 77–91. https://doi.org/10.1007/978-3-662-44725-3_5

[9] Li, Zewen et al. (2021). A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems*, 33, 12, pp. 6999–7019. https://doi.org/10.1109/TNNLS.2021.3084827

[10] El Alani, Omaima et al. (2021). Short term solar irradiance forecasting using sky images based on a hybrid CNN–MLP model. *Energy Reports*, 7, pp. 888–900. https://doi.org/10.1016/j.egyr.2021.07.053

[11] Zainuddin, Zarita, and Pauline, Ong (2008). Function approximation using artificial neural networks. *WSEAS Transactions on Mathematics*, 7, 6, pp. 333–338.

[12] Guo, Jianmin et al. (2021). Rnn-test: Towards adversarial testing for recurrent neural network systems. *IEEE Transactions on Software Engineering*, 48, 10, pp. 4167–4180. https://doi.org/10.1109/TSE.2021.3114353