Mining Business Process Stages from Event Logs

Hoang Nguyen[1(✉)], Marlon Dumas[2], Arthur H.M. ter Hofstede[1], Marcello La Rosa[1], and Fabrizio Maria Maggi[2]

[1] Queensland University of Technology, Brisbane, Australia
huanghuy.nguyen@hdr.qut.edu.au, {a.terhofstede,m.larosa}@qut.edu.au
[2] University of Tartu, Tartu, Estonia
{marlon.dumas,f.m.maggi}@ut.ee

Abstract. Process mining is a family of techniques to analyze business processes based on event logs recorded by their supporting information systems. Two recurrent bottlenecks of existing process mining techniques when confronted with real-life event logs are scalability and interpretability of the outputs. A common approach to tackle these limitations is to decompose the process under analysis into a set of stages, such that each stage can be mined separately. However, existing techniques for automated discovery of stages from event logs produce decompositions that are very different from those that domain experts would produce manually. This paper proposes a technique that, given an event log, discovers a stage decomposition that maximizes a measure of modularity borrowed from the field of social network analysis. An empirical evaluation on real-life event logs shows that the produced decompositions more closely approximate manual decompositions than existing techniques.

Keywords: Process mining · Decomposition · Clustering · Modularity · Multistage

1 Introduction

Process mining offers numerous opportunities to extract insights about business process performance and conformance from event logs recorded by enterprise information systems [1]. Among other things, process mining techniques allow analysts to discover process models from event logs for as-is analysis, to check the conformance of recorded process executions against normative process models, or to visualize process performance indicators. Process mining techniques however suffer from scalability issues when applied to large event logs, both in terms of computational requirements and in terms of interpretability of the produced outputs. For example, process models discovered from large event logs are often spaghetti-like and provide limited insights [1].

A common approach to tackle this limitation is to decompose the process into stages, such that each stage can be mined separately. This idea has been successfully applied in the context of automated process discovery [2] and performance mining [3]. The question is then how to identify a suitable set of stages and how to

E. Dubois and K. Pohl (Eds.): CAiSE 2017, LNCS 10253, pp. 577–594, 2017.
DOI: 10.1007/978-3-319-59536-8_36

map the events in the log into stages. For simpler processes, the stage decomposition can be manually identified, but for complex processes, automated support for stage identification is required. Accordingly, several automated approaches to stage decomposition have been proposed [4–6]. However, these approaches have not been designed with the goal of approximating manual decompositions, and as we show in this paper, the decompositions they produce turn out to be far apart from the corresponding manual decompositions.

This paper puts forward an automated technique to split an event log into stages, in a way that mimics manual stage decompositions. The proposed technique is designed based on two key observations: (i) that stages are intuitively fragments of the process in-between two milestone events; and (ii) that the stage decomposition is modular, meaning that there is a high number of direct dependencies inside each stage (high cohesion), and a low number of dependencies across stages (low coupling) – an observation that has also been applied in the context of process model decomposition [7] and more broadly in the fields of systems design and programming in general. For example, a loan origination process at a bank has multiple stages such as the application is assessed (accepted/rejected milestone), offered (offer letter sent milestone), negotiated (agreement signed milestone), and settled (agreement executed milestone). There may be many back-and-forth or jumps inside a stage, but relatively little across these stages.

The proposed technique starts by constructing a graph of direct control-flow dependencies from the event log. Candidate milestones are then identified by using techniques for computing graph cuts. A subset of these potential cut points is finally selected in a way that maximizes the modularity of the resulting stage decomposition according to a modularity measure borrowed from the field of social network analysis. The technique has been evaluated using real-life logs in terms of its ability to approximate manual decompositions using a well-accepted measure for the assessment of cluster quality.

The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 presents the proposed technique and Sect. 4 describes its empirical evaluation. Finally, Sect. 5 summarizes the contributions and outlines future work directions.

2 Related Work

The problem of automated decomposition of event logs into stages has been approached from multiple perspectives. For example, Carmona et al. [4] extract a transition system from an event log and apply a graph cut algorithm over this transition system to identify stages. A formal divide and conquer framework has been defined and formalized in [5], which has led to several instantiations and applications in case studies [2,6,8]. The key idea of this framework is to cluster activities in event logs by first constructing an activity causal graph from the logs and then searching for regions of heavy connected edges (edges with high weights) as activity clusters. A recent work of local process model discovery [9]

also seeks to cluster activities into subsets in order to speed up its performance as well as to increase the quality of the detected models. It uses three heuristics based on Markov clustering, log entropy and maximal relative information gain.

The above decompositions have been applied to automated process discovery. Other decomposition techniques have been proposed in the context of performance mining. For example, the Performance Analysis with Simple Precedence Diagram plug-in ProM [10] uses a medoid-based approach to find activity clusters. Given a similarity measure between activities, this technique identifies possible medoids and a membership function to determine to which medoid an activity should be assigned. A similar approach has been proposed in the context of queue mining from event logs [11].

None of the above techniques has been designed and evaluated in the view of producing stage decompositions that approximate manual ones. In the experiments reported later, we assess the performance of [10] and [5,6,8] with respect to manual decompositions, and compare it to the approach proposed in this paper.

Other related work deals with the problem of identifying sub-processes in an event log [12,13]. The output of these techniques is a log of the top-level process and a set of logs of sub-processes thereof. This output is not a stage decomposition. In a stage decomposition, every activity label in the log must be assigned to exactly one stage, i.e. the stages must form a partition of the set of activity labels, whereas the techniques described in [12,13] do not ensure that every activity label belongs to only one sub-process. In fact, these techniques do not guarantee that any sub-process will be found at all.

3 Stage Decomposition Technique

The proposed technique for extracting stages from an event log proceeds in two steps. In the first step, we construct a weighted graph from the event log capturing the direct-follows relation between activities in the process. In the second step, we split the nodes in the graph (i.e. the activities) into stages with the aim of maximizing a modularity measure. Below we introduce each of these two steps in detail.

3.1 From Event Log to Flow Graph

Table 1 shows an example event log of a loan origination process. An event log consists of a set of *cases*, where a case is a uniquely identified execution of a process. For example, the loan application identified by c_2 is a case. Each case consists of a sequence of *events*. An event is the most granular element of a log and is characterized by a set of attributes such as *timestamp* (the moment when the event occurred), *activity label* (the name of the action taken in the event), and *event types* relating to the activity lifecycle, such as "schedule", "start", and "complete".

Definition 1 (Event Logs). *An event log EL is a tuple (E, ET, A, C, time, act, type, case), where E is a set of events,* $ET = \{start, complete\}$ *is a set of*

Table 1. Example event log.

Case ID	Event ID	Event type	Timestamp	Activity label
c_1	e_1	Start	05.10 09:00:00	Update application
	e_2	Complete	05.10 10:00:00	Update application
c_2	e_3	Start	06.10 09:00:00	Update application
	e_4	Complete	06.10 10:00:00	Update application
	e_5	Start	08.10 09:00:00	Check application
	e_6	Complete	08.10 10:00:00	Check application
	e_7	Start	09.10 08:30:00	Check application
	e_8	Complete	09.10 09:00:00	Check application
c_3	e_9	Start	08.10 09:00:00	Update application
	e_{10}	Complete	08.10 10:00:00	Update application
	e_{11}	Start	09.10 09:00:00	Check application
	e_{12}	Complete	09.10 09:15:00	Check application
	e_{13}	Start	11.10 09:00:00	Follow-up offer
	e_{14}	Complete	11.10 10:00:00	Follow-up offer

event types, A is a set of activity labels, C is a set of cases, time: $E \rightarrow \mathbb{R}_0^+$ is a function that assigns a timestamp to an event, act: $E \rightarrow A$ is a function that assigns an activity label to an event, type: $E \rightarrow ET$ is a function that assigns an event type to an event, and case: $E \rightarrow C$ relates an event to a case. We write $e \lesssim_E e'$ iff $time(e) \leq time(e')$. In this paper, we only use "complete" events, denoted as E^c, where $E^c = \{e \in E | type(e) = complete\}$.

A *process graph* is a directed graph in which *nodes* represent activities and *edges* represent direct-follows relations between activities. For example, if activity b occurs after activity a in a case, the graph contains a node a, a node b and a directed edge from a to b. In addition, edges carry *weights* representing the frequency of the direct-follows relation between two related activities in the log.

Definition 2 (Process Graph). *A process graph of an event log EL=(E, ET, A, C, time, act, type, case) is a graph $G_{EL} = (V_{EL}, F_{EL}, W_{EL})$, where:*

- *V_{EL} is a set of nodes, each representing an activity, i.e. $V_{EL} = A$.*
- *F_{EL} is a set of directed edges, each representing the direct-follows relation between two activities based on "complete" events. Activity a_2 directly follows activity a_1 if there is a case in which the "complete" event e_2 of a_2 follows the "complete" event e_1 of a_1 without any other "complete" events in-between, i.e. e_1 is in a direct "complete" sequence with e_2. Event e_1 is in a direct "complete" sequence with e_2, denoted $e_1 \longrightarrow e_2$, iff $e_1 \in E^c \wedge e_2 \in E^c \wedge e_1 \neq e_2 \wedge case(e_1) = case(e_2) \wedge e_1 \lesssim_E e_2 \wedge \nexists e_3 \in E^c[e_3 \neq e_1 \wedge e_3 \neq e_2 \wedge case(e_3) = case(e_1) \wedge e_1 \lesssim_E e_3 \wedge e_3 \lesssim_E e_2]$. Thus, $F_{EL} = \{(a_1, a_2) \in V_{EL} \times V_{EL} | \exists e_1, e_2 \in E^c[act(e_1) = a_1 \wedge act(e_2) = a_2 \wedge e_1 \longrightarrow e_2]\}$.*

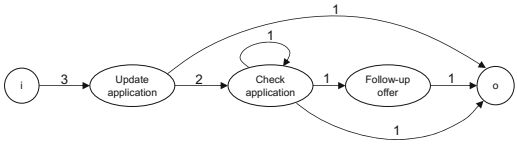


Fig. 1. Flow graph created from the event log in Table 1.

– W_{EL} *is a function that assigns a weight to an edge,* W_{EL}*:* $F_{EL} \rightarrow \mathbb{N}_0^+$*. The weight of an edge connecting node* a_1 *to node* a_2*, denoted* $W_{EL}(a_1, a_2)$*, is the frequency of the direct-follows relation between* a_1 *and* a_2 *in the log, i.e.* $W_{EL}(a_1, a_2) = |\{(e_1, e_2) \in E^c \times E^c | act(e_1) = a_1 \wedge act(e_2) = a_2 \wedge e_1 \longrightarrow e_2\}|$.

The process graph constructed above has a set of start nodes called *firstacts* containing the first activities of all cases, and a set of end nodes called *lastacts* containing the last activities of all cases, i.e. $firstacts(V_{EL}) = \{a \in V_{EL} | \exists e \in E^c \colon [act(e) = a \wedge \nexists e' \in E^c | e' \longrightarrow e]\}$, and $lastacts(V_{EL}) = \{a \in V_{EL} | \exists e \in E^c \colon [act(e) = a \wedge \nexists e' \in E^c | e \longrightarrow e']\}$.

From a process graph, we can derive a corresponding *flow graph*, which has only one source node i and one sink node o.

Definition 3 (Flow Graph). *The flow graph of a process graph* $G_{EL} = (V_{EL}, F_{EL}, W_{EL})$ *is a graph* $FL(G_{EL}) = (V_{EL}^{FL_G}, F_{EL}^{FL_G}, W_{EL}^{FL_G})$*, where:*

– $V_{EL}^{FL_G} = V_{EL} \cup \{i, o\}, \{i, o\} \cap V_{EL} = \varnothing$.
– $F_{EL}^{FL_G} = F_{EL} \cup \{(i, x) | x \in firstacts(V_{EL})\} \cup \{(x, o) | x \in lastacts(V_{EL})\}$
– $$W_{EL}^{FL_G}(a_1, a_2) = \begin{cases} W_{EL}(a_1, a_2) & \text{if } a_1 \neq i \wedge a_2 \neq o \\ |\{e \in E^c | act(e) = a_2 \wedge [\nexists e' \in E^c | e' \longrightarrow e]\}| & \text{if } a_1 = i \\ |\{e \in E^c | act(e) = a_1 \wedge [\nexists e' \in E^c | e \longrightarrow e']\}| & \text{if } a_2 = o \end{cases}$$

Figure 1 illustrates a flow graph constructed from the example log in Table 1, while Fig. 2 shows a flow graph created from a simulated log.

3.2 Stage Decomposition and Quality Measure

We assume that a process stage exhibits a *quasi-SESE* (single entry single exit) fragment on a flow graph. A quasi-SESE fragment is a MEME (multi-entry multi-exit) fragment, which has one entry point with high inflow and one exit point with high outflow (see Fig. 2), where inflow (outflow) is the total weight of the incoming (outgoing) edges. The entry and exit points are *transition nodes* between stages. We aim at developing a technique to extract a list of stages from a flow graph called a *stage decomposition*, where stages are sets of nodes.

In order to measure the quality of stage decompositions, we use *modularity* [14], which was proposed for detecting community structures in social networks. A community structure is characterized by a high density of edges within a community and a low number of edges connecting different communities. The

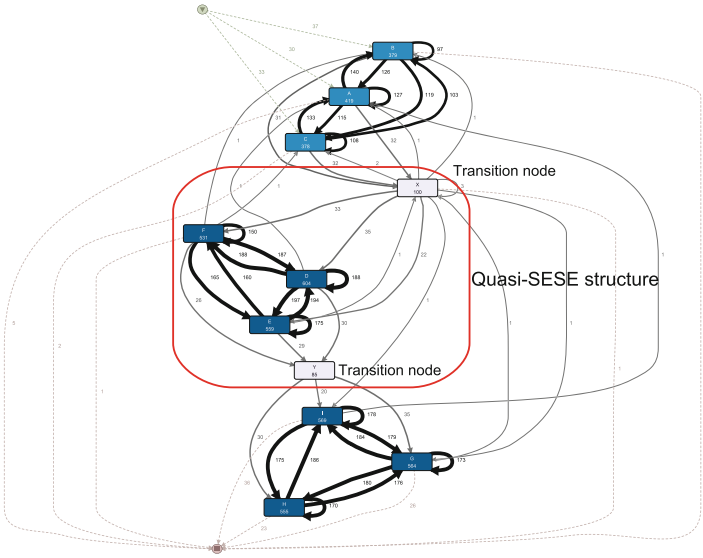


Fig. 2. Example quasi-SESE fragments.

higher the modularity is, the more a network exhibits a community structure. In this paper, we use a variant of modularity for weighted and directed graphs which are the characteristics of the flow graphs defined above.

Let S be a stage decomposition extracted from a flow graph based on an event log EL, and $S_i \in S$, where $i = 1 \ldots |S|$, be a stage. Let $W_{EL}^{FL_G}(S_i, S_j)$ be the total weight of edges connecting S_i to S_j (excluding self-loops), $W_{EL}^{FL_G}(S_i, S_j) = \sum_{a_1 \in S_i, a_2 \in S_j, a_1 \neq a_2} W_{EL}^{FL_G}(a_1, a_2)$. Let W^T be the total weight of all edges in the graph excluding self-loops, $W^T = \sum_{a_1, a_2 \in V_{EL}^{FL_G}, a_1 \neq a_2} W_{EL}^{FL_G}(a_1, a_2)$. The modularity of a stage decomposition is computed based on a *modular graph* which is the flow graph with a special treatment for transition nodes (see Fig. 3). Every transition node in the stage decomposition is split into two child nodes, one as an end node of one stage and the other as a start node of the next stage. The edges connected to the transition node are connected to the child nodes accordingly. The child nodes are also connected between each other through a new edge with weight equal to zero. In this way, the weight of edges in the modular graph remains the same as in the original graph. The modular graph is used for computing modularity because it can well reflect the quality of stage decomposition.

Let $W_{EL}^{FL'_G}(S_i, S_j)$ be the total weight of edges connecting S_i to S_j in the modular graph. The modularity of a stage decomposition S is computed as follows.

$$Q = \sum_{i=1}^{|S|} (E_i - A_i^2) \tag{1}$$

where $E_i = \frac{W_{EL}^{FL'_G}(S_i,S_i)}{W^T}$ is the fraction of edges that connect nodes within stage S_i and $A_i = \frac{\sum_{j=1}^{|S|} W_{EL}^{FL'_G}(S_j,S_i)}{W^T}$ is the fraction of edges that connect to stage S_i, including those within stage S_i and those from other stages.

Stage decomposition may become overly fragmented as a fragment could in principle be composed of only two strongly connected activities. Therefore, we introduce *minimum stage size* indicating the smallest number of activities in any given stage as a user parameter for stage decomposition. Through this, one can decide on what level of granularity they may want to look at stages.

3.3 Stage Decomposition Algorithm

Given an event log, we seek to find a stage decomposition that can maximize modularity. To this end, we propose a technique that starts from the flow graph constructed from the log, and recursively decomposes it into sets of nodes using the notion of min-cut as calculated by Ford-Fulkerson's algorithm. Note that the min-cut here is the one found in the graph after a node has been removed. The set of edges in that min-cut is called a *cut-set* associated with the removed node, and the total weight of edges in the cut-set is called *cut-value*. Together, a node and its cut-set form a border between two graph fragments. The lower the cut-value is, the more the related fragments will resemble quasi-SESE fragments. Therefore, if we find a set of nodes with low cut-values, we can take multiple graph cuts on those nodes and their cut-sets to obtain a stage decomposition that can approximate the maximum modularity.

Transition nodes intuitively have lower cut-values than the min-cut found by Ford-Fulkerson's algorithm in the original flow graph (called *source-min-cut*). Thus, we can use the source-min-cut as a threshold when selecting a candidate list of cut-points, i.e. nodes with cut-values smaller than that of the source-min-cut will be selected. Further, in a flow graph, the source-min-cut can be computed in constant time as it is equal to the set of outgoing edges of the source node of the graph or the set of incoming edges of the sink node.

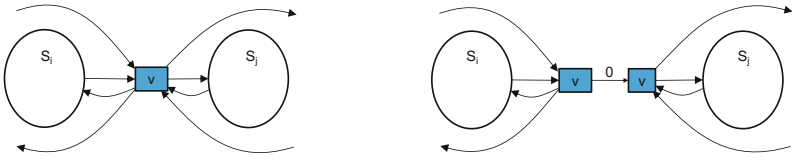
Once we have a candidate list, the key question is how to find a subset of nodes to form a stage decomposition that can maximize modularity. One way is to generate all possible subsets from the list, create stage decompositions based on all subsets, and select the one that has the highest modularity. However, this approach may suffer from combinatorial problems if the number of candidate nodes is large. For example, if we assume that the flow graph has 60 nodes and the candidate list has 30 nodes, the total number of subsets would be $\binom{30}{1} + \binom{30}{2} + \ldots + \binom{30}{30} = 1{,}050{,}777{,}736$. We thus propose two algorithms (Algorithms 1 and 2) to find a stage decomposition that can approximate the maximum modularity. The inputs to the algorithms are an event log and a minimum stage size.

Algorithm 1 is a greedy algorithm. The main idea (Lines 9–22) is to search in the candidate list for a cut-point that can result in a stage decomposition with two stages and of highest modularity. Then it removes the node from the candidate list (Line 20) and searches in the list again for another cut-point that

can create a new decomposition with three stages and of highest modularity, i.e. higher than the former decomposition and the highest among all decompositions with three stages, and so on until it cannot either find a stage decomposition of higher modularity or all new decompositions have a stage of smaller size than the minimum stage size. Note that stage decomposition is recursive meaning a stage in the current decomposition will be decomposed into two sub-stages based on a selected cut-point (Line 14). Modularity is computed according to Eq. 1 based on the modular graph as described above (Line 15).

Algorithm 2 has the same structure as Algorithm 1, but uses the lowest cut-value as a heuristic. Firstly, it sorts the candidate list in ascending order of cut-values, then it sequentially picks every node from the list to create recursive stage decompositions until the modularity is not increased or all new decompositions have a stage of smaller size than the minimum threshold.

The worst-case time complexity of functions used in the algorithms can be computed as follows. The *create_flow_graph* function is $O(V + F)$, where $V = V_{EL}^{FL_G}$ and $F = F_{EL}^{FL_G}$. The *node_min_cut* function removes a node from the graph and uses Ford-Fulkerson's algorithm to find a min-cut; it is $O(Fw)$, where w is the maximum weight of edges in the flow graph [15]. The *source_min_cut* function is $O(1)$ since it only computes the total weight of edges originating from the source node. The *find_cut_stage* function searches a stage that contains the current node in the current stage decomposition; it is $O(V)$. The *cut_graph* function (Algorithm 3) is $O(V + F)$, which performs a depth-first search to find disconnected components in the graph [15]. The *copy_sd* function is $O(V)$ (replace a stage with two sub-stages). The *modularity* function is $O(V + F)$, which involves copying the original graph to a new one with a special treatment for cut-points ($O(V + F)$) and computing the modularity based on Eq. 1 ($O(F)$). The *get_activity_labels* function is $O(V)$ (extract activity labels from nodes). The *sort* function (Algorithm 2) is $O(V \log V)$. Based on these observations, the complexity of Algorithm 1 is $O(V^2(V + F)))$, and Algorithm 2 is $O(V(V + F))$.



(a) Before split (b) After split

Fig. 3. Treatment for transition nodes in computing modularity.

4 Evaluation

We call our technique *Staged Process Miner* (SPM) and implemented it as a ProM plug-in as well as a stand-alone Java tool[1]. We evaluated the technique

[1] Available from http://apromore.org/platform/tools.

Algorithm 1. Highest Modularity Stage Decomposition

Input: *EL*: an event log
minStateSize: minimum number of activities in a stage
Output: A sequence of stages, each is a set of activity labels

```
1  G = create_flow_graph(EL)
2  CandidateNodes := {}
3  forall v in V_EL^FL_G \ {i, o} do
4      <v.mincut, v.cutset>:= node_min_cut(G, v)
5      if v.mincut < source_min_cut(G) then
6          CandidateNodes := CandidateNodes ∪ {v}
7  CurrentBestSD := [V_EL^FL_G \ {i, o}]
8  NewBestSD := CurrentBestSD
9  while CandidateNodes ≠ {} do
10     forall v in CandidateNodes do
11         CutStage := find_cut_stage(CurrentBestSD, v)
12         <PreStage, SucStage>:= cut_graph(G, v, CutStage)
13         if |PreStage| ≥ minStateSize and |SucStage| ≥ minStateSize then
14             NewSD := copy_sd(CurrentBestSD, CutStage, PreStage, SucStage)
15             if modularity(NewSD, G) > modularity(NewBestSD, G) then
16                 NewBestSD := NewSD
17                 BestCutPoint := v
18     if NewBestSD ≠ CurrentBestSD then
19         CurrentBestSD := NewBestSD
20         CandidateNodes := CandidateNodes \ {BestCutPoint}
21     else
22         break // stop when modularity is not increased
23 return get_activity_labels(CurrentBestSD)
```

Algorithm 2. Lowest Cut-value Stage Decomposition

Input: *EL*: an event log
minStageSize: minimum number of activities in a stage
Output: A sequence of stages, each is a set of activity labels

```
   // Line 1-7 is the same as Algorithm 1
8  Candidates_sorted := sort(CandidateNodes, min_cut, asc)
9  while Candidates_sorted ≠ [] do
10     v := head(Candidates_sorted)
11     CutStage := find_cut_stage(CurrentBestSD, v)
12     <PreStage, SucStage>:= cut_graph(G, v, CutStage)
13     if |PreStage| ≥ minStateSize and |SucStage| ≥ minStateSize then
14         NewSD := copy_sd(CurrentBestSD, CutStage, PreStage, SucStage)
15         if modularity(NewSD, G) > modularity(CurrentBestSD, G) then
16             CurrentBestSD := NewSD
17         else
18             break // stop when modularity is not increased
19     Candidates_sorted := tail(Candidates_sorted)
20 return get_activity_labels(CurrentBestSD)
```

through a range of real-life logs and against two baselines. The input to the technique is an event log and the minimum stage size; the output is an ordered set of activity sets, where each activity set represents a stage. The ProM plug-in also offers a visualization of the stage decomposition as *staged process maps*, where boxes represent stages and list all activities that belong to a given stage,

Algorithm 3. *cut_graph*

Input: G: a flow graph
v: a node
$CutStage$: a node set containing v
Output: a pair of subsets of $CutStage$

```
1 G_aftercut := remove_edges(remove_node(G, v), v.cutset)          // Graph cut
2 G_source := source_graph(G_aftercut)          // The subgraph containing the source
3 PreStage := (CutStage ∩ V_{G_source}) ∪ {v}
4 SucStage := CutStage \ PreStage
5 return <PreStage, SucStage>
```

and arcs between stages report the frequency of handover from one stage to the other (the thicker the arc, the higher the frequency) – see Fig. 5 for an example.

The purpose of this research is to determine if it is possible to algorithmically produce stage decompositions of event logs that mimic decompositions produced manually by domain experts. Accordingly, we define the quality of a stage decomposition in terms of its similarity relative to a manually produced ground truth, the evaluation aims at addressing the following top-level question:

Q1. How does the quality of the stage decomposition produced by our technique compare to that of existing baselines?

The decomposition produced by our technique depends on the selected minimum stage size. Accordingly, we also address the following ancillary question:

Q2. How does the quality of the decomposition produced by our technique vary depending on the minimum stage size?

4.1 Datasets

We used seven publicly available, real-life event logs. These include two logs from the Business Process Intelligence (BPI) Challenge 2012 and 2013, and five logs from the BPI Challenge 2015. Table 2 reports descriptive statistics on the size of these datasets.

BPI12[2] is a loan origination process in a Dutch bank. Its stages are: (i) pre-assess application completeness, (ii) assess eligibility, (iii) offer & negotiate loan packages with customers, iv) validate & approve. As a ground truth, these stages are marked in the log by milestone events occurring at the end of each stage, such as A_PREACCEPTED (stage i) and A_ACCEPTED (stage ii), where "A" stands for Application. We preprocessed this log by replacing a group of milestone events occurring usually simultaneously at the end of a stage with one representative milestone event only.

BPI13[3] is an IT incident handling process at Volvo Belgium. A stage in this process reflects the IT helpdesk level (team) where an IT incident ticket is processed. The IT department has three levels from 1 to 3. The ground truth of stages thus is the helpdesk level of the resource who initiates an event. This log

[2] doi:10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f.
[3] doi:10.4121/uuid:500573e6-accc-4b0c-9576-aa5468b10cee.

is preprocessed by selecting only complete cases, i.e. cases that have completed all stages.

BPI15[4] is a set of five logs from five Dutch municipalities relating to a building permit application process. This process has many stages, such as: (i) application receipt, (ii) completeness check of the application, (iii) investigation leading to a resolution (e.g. accept, reject, ask for more info), (iv) communication of the resolution, (v) public review, (vi) decision finalization, and (vii) objection and complaint filing. The ground truth of stages in this process is encoded in the action_code field which has a generic format 01_HOOFD_xyy, where x indicates the stage number and yy indicates the activity code within the stage. This log is preprocessed by selecting only events of the main process (i.e. events with HOOFD code), and then selecting cases that have completed stage 1 to stage 4, which show strong quasi-SESE fragments.

Table 2. Statistics on the datasets used in the evaluation.

Dataset	Business process	Number of cases	Number of events	Event classes
BPI12	Loan origination	13,087	127,290	19
BPI13	IT incident handling	175	1,996	27
BPI15-1	Building permit application	834	11,451	61
BPI15-2		618	8,979	52
BPI15-3		1,013	13,929	60
BPI15-4		792	10,710	50
BPI15-5		951	13,682	56

4.2 Baselines

We used two baseline techniques in our evaluation: the *Divide and Conquer framework* (DC) and the *Performance Analysis with Simple Precedence Diagram* (SPD) presented in Sect. 2. They are both available in ProM.

DC consists of a set of ProM plug-ins run in sequence: Discover Matrix, Create Graph, Create Clusters and Modify Clusters. These plug-ins require one to configure many parameters, notably the number of clusters and the target cluster size. This tool-chain is designed to be used in an interactive manner where users can see how their selected parameters affect the decomposition through visualizations. Since clusters must be disjoint in a stage decomposition, we select parameters for this tool-chain in such a way to only generate disjoint clusters.

SPD takes as input an event log and a number of clusters to be produced. The output is a diagram called *Simple Precedence Diagram* where nodes are clusters of activities. The plug-in uses medoid-based fuzzy clustering. In order to obtain disjoint clusters, we adapted the membership function such that given an activity it only returns one medoid with the highest membership value.

[4] doi:10.4121/uuid:31a308ef-c844-48da-948c-305d167a0ec1.

4.3 Accuracy Index

To assess the accuracy of a stage decomposition against the ground truth, we experimented with three well-known external indexes of clustering quality: *Rand*, *Fowlkes–Mallows* and *Jaccard* [16]. These indexes are used to evaluate the similarity of two clusterings. The higher the index is, the more similar the two clusterings are. In our tests, the Rand Index was very high even for less similar clusterings while Jaccard was often low even for very similar clusterings. Fowlkes–Mallows provided more reasonable results between those returned by the other two indexes. Thus, we decided to report the results using the Fowlkes–Mallows index only, given that Rand and Jaccard also showed consistent results across all datasets and techniques.

The formula for Fowlkes–Mallows is provided below, where n_{11} is the number of activities that are in the same stage in both decompositions, and $n_{10}(n_{01})$ is the number of activities that are in the same stage in the first (second) decomposition but in different stages in the second (first) decomposition.

$$\text{Fowlkes}-\text{Mallows} = \frac{n_{11}}{\sqrt{(n_{11}+n_{10})(n_{11}+n_{01})}} \tag{2}$$

4.4 Results

We present the evaluation results in light of the two questions defined above.

Q1. How does the quality of the stage decomposition produced by our technique compare to that of existing baselines?

We run DC, SPD and SPM with different parameter settings and chose for each technique the configuration that achieves the highest accuracy in terms of the Fowlkes–Mallows index. The best configuration for each technique is reported in Table 3. Further values used for DC are: Modify Clusters Miner = "Incremental using Best Score (Overlapping Only)"; Cohesion/Coupling/Balance/Overlap Weight = 100/100/0/100, while all other parameters we used default values, e.g. Discovery Matrix Classifier = Activity.

Table 4 shows the Fowlkes–Mallows index for the three techniques, for each log. SPM, in either of its two variants (highest modularity and lowest cut-value) consistently outperformed the two baseline techniques across all datasets, with slightly higher results achieved by the highest modularity algorithm. These results attest the appropriateness of the modularity measure for stage decomposition, with lowest cut-value being a good approximation of the ground truth. In addition, our heuristics-based techniques with highest modularity and lowest cut-value can approximate the optimal selection of cut-points when comparing with the exhaustive technique for BPI12 and BPI13 logs. For BPI15-x logs, the exhaustive technique does not finish after running for several hours due to the large number of combinations of cut-points. For example, BPI15-1 has 61 activities and 30 candidate cut-points and, for minStageSize = 4, the total number of combinations of cut-points is 614,429,471 ($\binom{30}{1} + \binom{30}{2} + ... + \binom{30}{15}$).

Table 3. Parameters configuration for the evaluated techniques.

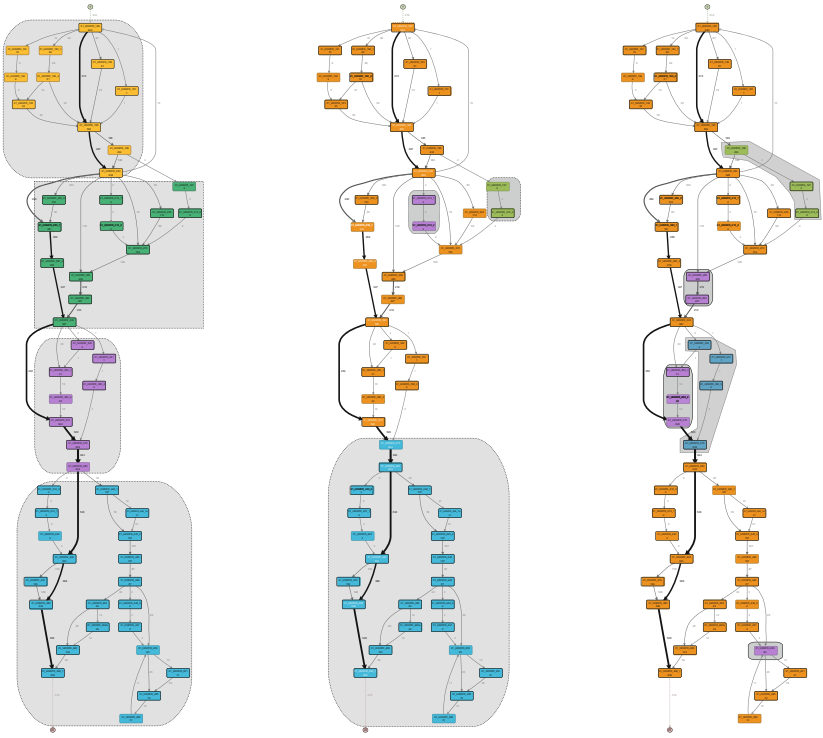
Dataset	DC			SPD	SPM
	No. of clusters	Target cluster size	Weight threshold	No. of clusters	Minimum stage size
BPI12	4	5	0.943	4	3
BPI13	3	5	0.834	3	5
BPI15-1	4	12	0.432	4	4
BPI15-2	4	12	0.425	4	4
BPI15-3	4	12	0.527	4	4
BPI15-4	4	12	0.597	4	5
BPI15-5	4	12	0.507	4	5

Table 4. Fowlkes–Mallows index for the evaluated techniques.

Dataset	Stages	DC	SPD	SPM		
				Highest modularity	Lowest cut-value	Exhaustive
BPI12	4	0.30	0.49	**1.0**	0.92	1.0
BPI13	3	0.36	0.73	**0.78**	**0.78**	0.78
BPI15-1	4	0.40	0.54	0.90	**0.92**	Timed-out
BPI15-2	4	0.40	0.52	**0.92**	0.76	Timed-out
BPI15-3	4	0.42	0.50	**0.86**	**0.86**	Timed-out
BPI15-4	4	0.45	0.57	**0.72**	**0.72**	Timed-out
BPI15-5	4	0.46	0.49	**0.83**	**0.83**	Timed-out

As an example, Fig. 4 shows the decomposition identified by our technique (highest modularity) and by the two baselines, for the BPI2015-2 log, on top of the direct-follows graph of the event log. Here activities have been color-coded (or marked in shaded areas) based on the clusters they belong to. We can observe that in both the baselines, stage boundaries are not sharply defined, with many activities being mixed between stages.

The low accuracy of the two baselines is due to the underlying clustering approach used. DC searches for clusters starting from heavy edges (edges with high weights) and growing the cluster to other connected edges with weight over a threshold. This is the reason why it can detect some regions that cover an actual stage, but fails to determine exactly where to stop clustering. SPD searches for clusters based on medoids, i.e. a central node in a direct-follows graph that is close to all other nodes in a cluster, where closeness is measured by the frequency of the direct-follows relation between activities. SPD thus tends to produce a large cluster covering several actual stages because stages are usually



(a) SPM (b) DC (c) SPD

Fig. 4. Stage decomposition for the BPI15-2 log. Shaded areas are activity clusters. Activities not in any shaded areas belong to one big cluster.

strongly connected via transition nodes. In general, both baseline techniques are unable to detect stage boundaries.

In addition, only our technique can retrieve stages in the correct order, while ordering is not part of the results provided by the two baseline techniques. We can see this, for example, in Figs. 5 and 6, which show the stage decomposition for the BPI12 and BPI13 logs provided by our ProM plug-in.

To complement our comparison with baselines, we also experimented with three clustering techniques proposed in local process model discovery [9]. They are based on well-established heuristics used in data mining, such as Markov clustering, log entropy, and maximal relative information gain. However, the results obtained are very different from the ground truth, with Fowlkes–Mallows Index always being below 0.5.

In terms of runtime performance, both our technique and the two baselines perform within reasonable bounds, in the order of seconds (see Table 5). However, the exhaustive technique could not finish for BPI15-x logs after running for several hours.

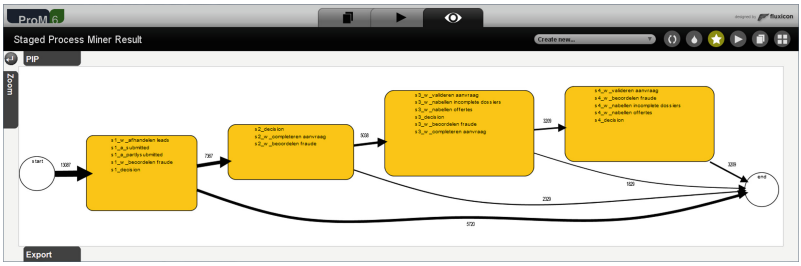


Fig. 5. Stage decomposition produced by SPM for the BPI12 log.

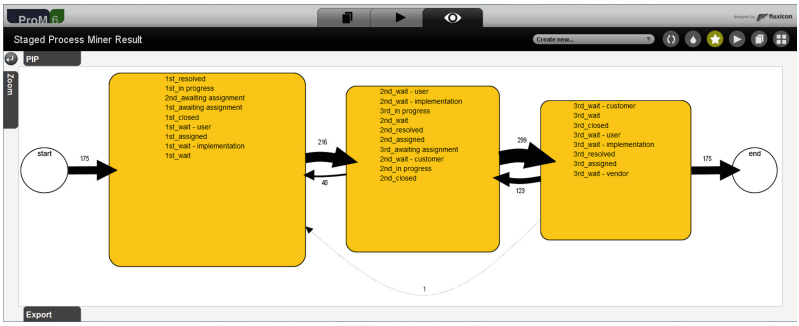


Fig. 6. Stage decomposition produced by SPM for the BPI13 log.

Q2. How does the quality of the decomposition produced by our technique vary depending on the minimum stage size?

To answer this question, we run our technique with the highest modularity algorithm using different values of minimum stage size (minSS), from 2 to half of the total number of activities in an event log. Table 6 provides the characteristics of different stage decompositions, each for a minSS value. It shows that the modularity is higher when minSS is small and peaks when minSS is equal to 2. This is because, when minSS is small, the technique is allowed to decompose the graph into stages as much as possible to increase modularity. For example, for minSS = 2, the best stage decomposition for BPI15-1 log has 7 stages, in which two stages have size 2 (i.e. two activities), one has size 3, one has size 5, one has size 11, one has size 14, and one has size 24.

Notably, in Table 6, for each dataset, one resulting decomposition is very close to the ground truth, such as minSS = 3 for BPI12 (FM = 1.0), minSS = 9 for BPI13 (FM = 0.85), minSS = 4 for BPI15-1 (FM = 0.90), minSS = 4 for BPI15-2 (FM = 0.92), and minSS = 4 for BPI15-3 (FM = 0.86). This suggests how to use our technique for stage-based analysis. Users may decide not to fix the minimum stage size, run the technique for different sizes (as we did in this second experiment) and choose the stage decomposition that best suits their needs. For example, for BPI15-1 log, they can vary the minSS parameter to view different

Table 5. Run-time performance (in seconds)

Dataset	DC[a]	SPD	SPM		
			Highest modularity	Lowest cut-value	Exhaustive
BPI12	2	0.563	10	5	158
BPI13	2	0.019	0.36	0.31	1
BPI15-1	2	0.096	1	0.85	Timed-out
BPI15-2	2	0.069	1	0.85	Timed-out
BPI15-3	2	0.070	1	0.87	Timed-out
BPI15-4	2	0.050	1	0.64	Timed-out
BPI15-5	2	0.072	1	0.73	Timed-out

[a]Estimated due to manual use of plugins

stage decompositions as shown in Table 6. They can then rely on the number of stages and the associated modularity as a recommendation to choose the best stage decomposition. However, a good balance between optimal number of stages and high modularity needs to be identified manually. For example, for BPI15-1 log, the process with seven stages has high modularity but probably too many stages. On the other hand, the process with three stages has low modularity that also indicates that the result may not be good candidate for stage decomposition.

5 Conclusion

Given a business process event log, the technique presented in this paper partitions the activity labels in the log into stages delimited by milestones. The idea is to construct a direct-follows graph from the log, to identify a set of candidate milestones via a minimum cut algorithm, and to heuristically select a subset of these milestones. The paper considered two greedy heuristics: one that selects at each step the milestone with the lowest cut-value, and another that selects milestones that maximize modularity, using a modularity measure originally designed for social networks.

The technique has been implemented as a plug-in in the ProM framework, which splits an event log into stages and generates a staged process map. Experimental results on seven real-life event logs show that: (i) both heuristics significantly outperform previously proposed event log decomposition techniques in terms of the concordance of the produced decompositions relative to manual decompositions; and (ii) the stage decompositions generated by maximizing modularity outperform those based on cut-value. The latter result confirms previous empirical observations in the field of process model decomposition [7], while demonstrating the applicability of a modularity measure for social networks in this setting.

The proposed technique has a range of applications in the field of process mining. For example, stage decompositions can be used to scale up automated process discovery techniques [5,6] or to produce decomposed metrics and visual-

Table 6. Highest Modularity SPM with different minimum stage sizes (MinSS = Minimum Stage Size, Mod = Modularity, FM = Fowlkes–Mallows).

MinSS	BPI12			BPI13			BPI15-1			BPI15-2			BPI15-3		
	Stages	Mod	FM	Stages	Mod	FM	Stages	Mod	FM	Stages	Mod	FM	Stages	Mod	FM
2	6	0.70	0.75	6	0.67	0.56	7	0.80	0.82	7	0.79	0.82	7	0.82	0.76
3	4	0.59	**1.00**	4	0.61	0.72	5	0.77	0.83	5	0.77	0.84	6	0.80	0.75
4	4	0.57	0.81	4	0.61	0.72	4	0.73	**0.90**	4	0.72	**0.92**	4	0.73	**0.86**
5	3	0.55	0.82	3	0.58	0.78	4	0.73	0.90	4	0.72	0.92	4	0.73	0.86
6	3	0.44	0.70	3	0.58	0.78	4	0.73	0.90	4	0.72	0.92	4	0.73	0.86
7	2	0.40	0.67	3	0.58	0.78	4	0.73	0.90	4	0.72	0.92	4	0.73	0.86
8	2	0.40	0.67	3	0.58	0.78	5	0.69	0.61	4	0.68	0.72	4	0.73	0.86
9	2	0.34	0.54	3	0.52	**0.85**	5	0.69	0.61	4	0.68	0.72	4	0.73	0.86
10				2	0.38	0.65	4	0.68	0.73	4	0.68	0.72	4	0.73	0.86
11				2	0.38	0.65	4	0.68	0.73	4	0.68	0.72	4	0.68	0.67
12				2	0.38	0.65	4	0.68	0.73	3	0.58	0.66	4	0.68	0.67
13				2	0.38	0.65	3	0.68	0.73	3	0.58	0.66	3	0.57	0.65
14							3	0.57	0.66	3	0.58	0.62	3	0.57	0.65
15							3	0.57	0.66	3	0.58	0.62	3	0.57	0.65
16							3	0.56	0.63	2	0.49	0.74	3	0.57	0.65
17							3	0.56	0.63	2	0.49	0.74	3	0.56	0.62
18							2	0.49	0.74	2	0.49	0.74	2	0.49	0.69
19							2	0.49	0.74	2	0.49	0.74	2	0.49	0.69
20							2	0.49	0.74	2	0.49	0.74	2	0.49	0.69
21							2	0.49	0.74	2	0.49	0.74	2	0.49	0.69
22							2	0.49	0.74	2	0.49	0.74	2	0.49	0.69
23							2	0.49	0.74	2	0.49	0.74	2	0.49	0.69
24							2	0.49	0.74	2	0.48	0.73	2	0.49	0.69
25							2	0.49	0.74	2	0.48	0.73	2	0.49	0.69
26							2	0.49	0.74				2	0.48	0.68
27							2	0.49	0.74				2	0.48	0.68
28							2	0.48	0.73				2	0.48	0.68
29							2	0.48	0.73						

izations for performance analysis [3]. Investigating these applications is an avenue for future work.

Beyond the field of process mining, the proposed technique could find application in the realm of customer journey analysis, by allowing analysts to identify stages from customer session logs. With suitable extensions, the technique could also be used to compute abstracted views of large event sequences for interactive visual data mining.

Acknowledgments. This research is funded by the Australian Research Council (grant DP150103356) and the Estonian Research Council (grant IUT20-55).

References

1. van der Aalst, W.M.: Process mining: discovering and improving spaghetti and lasagna processes. In: Proceedings of CIDM. IEEE (2011)
2. Hompes, B.F.A., Verbeek, H.M.W., Aalst, W.M.P.: Finding suitable activity clusters for decomposed process discovery. In: Ceravolo, P., Russo, B., Accorsi, R. (eds.) SIMPDA 2014. LNBIP, vol. 237, pp. 32–57. Springer, Cham (2015). doi:10.1007/978-3-319-27243-6_2
3. Nguyen, H., Dumas, M., Hofstede, A.H.M., Rosa, M., Maggi, F.M.: Business process performance mining with staged process flows. In: Nurcan, S., Soffer, P., Bajec, M., Eder, J. (eds.) CAiSE 2016. LNCS, vol. 9694, pp. 167–185. Springer, Cham (2016). doi:10.1007/978-3-319-39696-5_11
4. Carmona, J., Cortadella, J., Kishinevsky, M.: Divide-and-conquer strategies for process mining. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 327–343. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03848-8_22
5. Van Der Aalst, W.M.: A general divide and conquer approach for process mining. In: Proceedings of FedCSIS, pp. 1–10. IEEE (2013)
6. Verbeek, H., van der Aalst, W.M., Munoz-Gama, J.: Divide and conquer. Technical report, BPM Center Report Series (2016)
7. Reijers, H.A., Mendling, J., Dijkman, R.M.: Human and automatic modularizations of process models to enhance their comprehension. Inf. Syst. **36**(5), 881–897 (2011)
8. Verbeek, H.M.W., Aalst, W.M.P.: Decomposed process mining: the ILP case. In: Fournier, F., Mendling, J. (eds.) BPM 2014. LNBIP, vol. 202, pp. 264–276. Springer, Cham (2015). doi:10.1007/978-3-319-15895-2_23
9. Tax, N., Sidorova, N., van der Aalst, W.M., Haakma, R.: Heuristic approaches for generating local process models through log projections. In: Proceedings of CIDM (2016)
10. Dongen, B.F., Adriansyah, A.: Process mining: fuzzy clustering and performance visualization. In: Rinderle-Ma, S., Sadiq, S., Leymann, F. (eds.) BPM 2009. LNBIP, vol. 43, pp. 158–169. Springer, Heidelberg (2010). doi:10.1007/978-3-642-12186-9_15
11. de Smet, L., van der Aalst, W., Verbeek, H.: Queue mining: combining process mining and queueing analysis to understand bottlenecks, to predict delays, and to suggest process improvements. Master thesis, Eindhoven University of Technology (2014)
12. Li, J., Bose, R.P.J.C., Aalst, W.M.P.: Mining context-dependent and interactive business process maps using execution patterns. In: Muehlen, M., Su, J. (eds.) BPM 2010. LNBIP, vol. 66, pp. 109–121. Springer, Heidelberg (2011). doi:10.1007/978-3-642-20511-8_10
13. Conforti, R., Dumas, M., García-Bañuelos, L., La Rosa, M.: BPMN miner: automated discovery of BPMN process models with hierarchical structure. Inf. Syst. **56**, 284–303 (2016)
14. Newman, M.E., Girvan, M.: Finding and evaluating community structure in networks. Phys. Rev. E **69**(2), 026113 (2004)
15. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2009)
16. Halkidi, M., Batistakis, Y., Vazirgiannis, M.: On clustering validation techniques. J. Intell. Inf. Syst. **17**(2–3), 107–145 (2001)