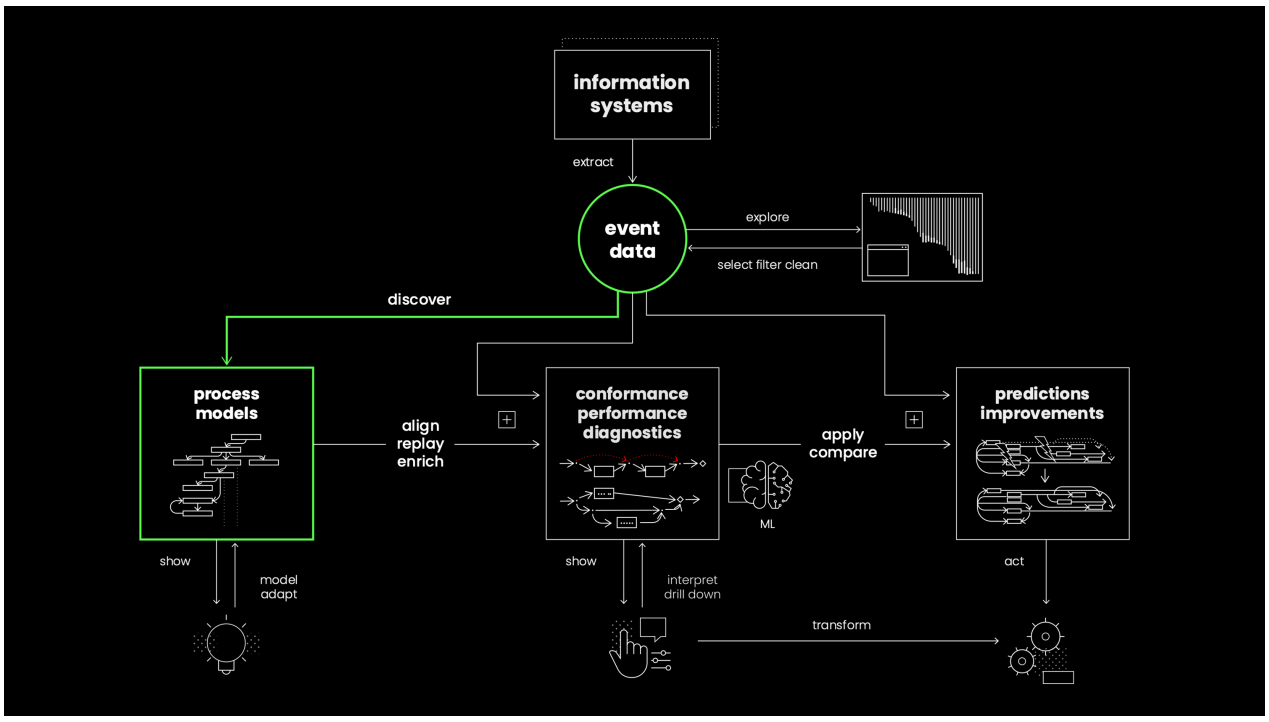
Process Discovery

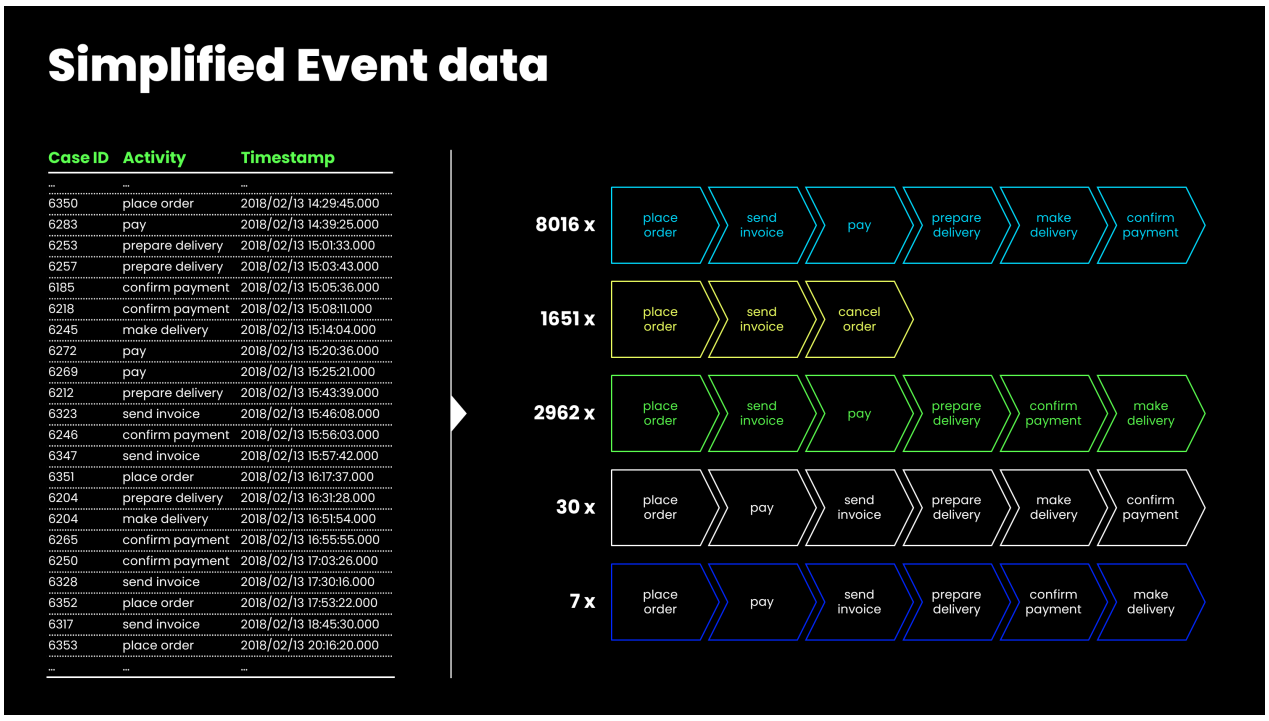
▼ Overview

- **Process mining** involves tasks like *process discovery*, *conformance checking*, and *process improvement*, but process discovery is particularly challenging and interesting. It starts with extracting event data from source systems.
- **Process discovery algorithms** use event data to automatically build a process model that describes observed behaviors. This is complex because event data are often incomplete, showing only examples of behaviors. Conflicting requirements, like desiring a simple model that excludes infrequent behaviors, add to the challenge.



- The input for process discovery is an **event log**, created after exploring, selecting, filtering, and cleaning event data from information systems. Events are defined by three main attributes:
  - case (a process instance)
  - activity (the task performed)
  - timestamp (when the task occurred).
- In control-flow-focused process mining, attention centers on
  - the **activity attribute** and the sequence of activities within each case.
  - Events are organized per case
  - ordered based on timestamps to form sequences.

  These sequences are then projected based on activity names to create traces—sequences of activities representing each case.
- Multiple cases may produce the same trace, resulting in different cases generating identical traces with varying frequencies. This is visually represented in process mining diagrams.



- **Simplified event logs**, technically called multisets of traces, are essential for deriving process models. These models, such as

- Directly-Follows Graphs (DFGs),
- Petri nets,
- BPMN models,
- UML activity diagrams

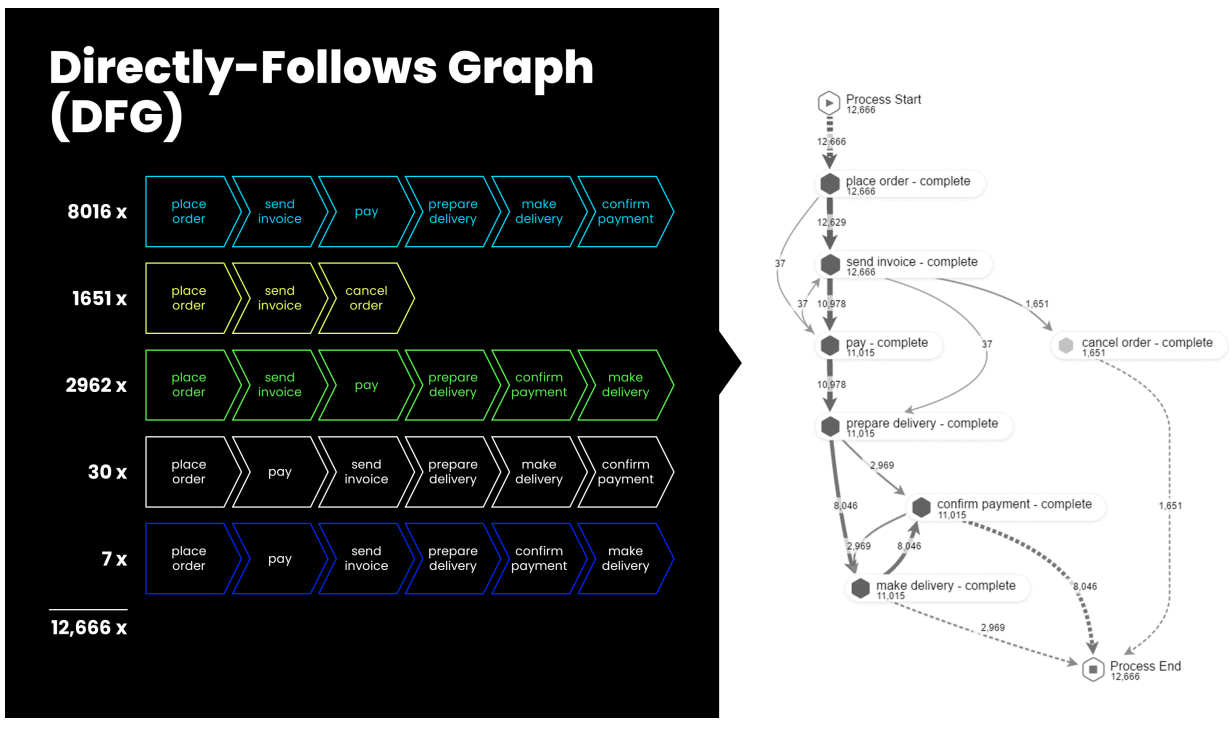
  outline possible and impossible sequences of activities.

- Process models are typically graphical and describe the set of possible process behaviors based on observed event data. They facilitate stakeholder inspection and diagnostic projections, such as identifying bottlenecks and analyzing frequencies.

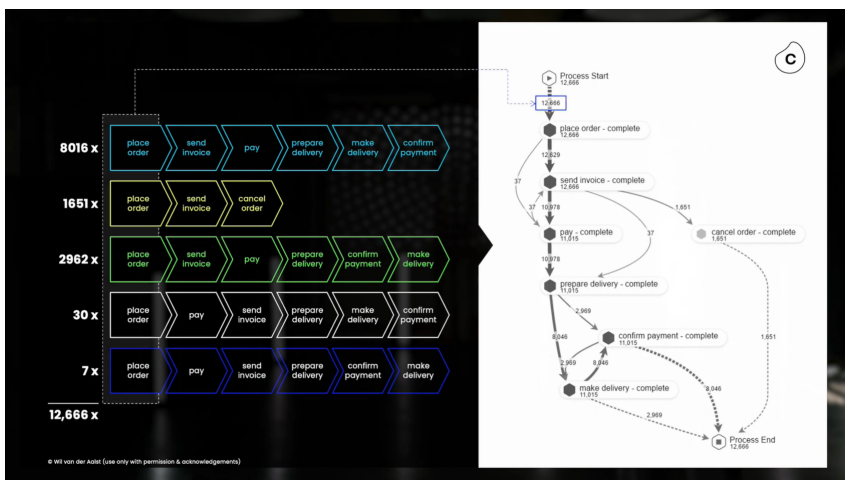
▼ Directly Follows Graphs - DFG

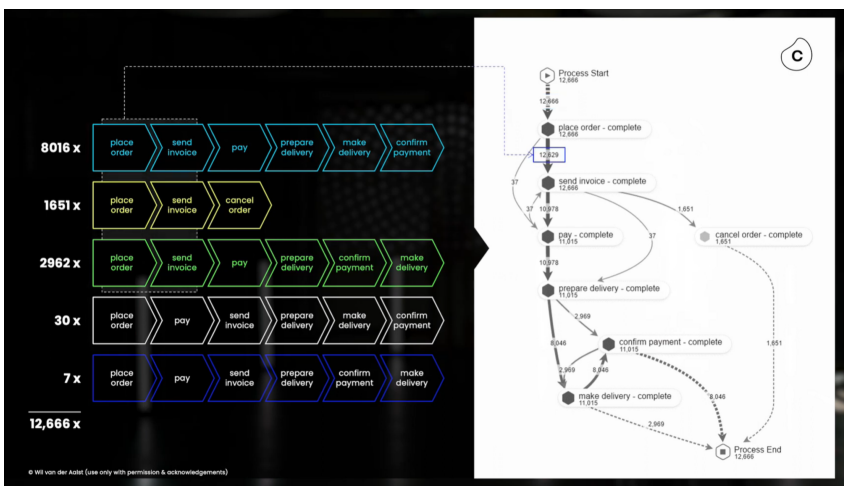
▼ Concept

The most basic discovery approach that is supported by most (if not all) process mining tools is a so-called *Directly-Follows Graph (DFG)* by simply counting how often one activity is followed by another activity within cases. A DFG is a graph composed of nodes and directed edges. The *nodes* correspond to *activities*, and the *edges* correspond to *directly-follows relations*. *Artificially* added activities are added to clearly indicate the *start* and *end* of the process. These dummy activities do not appear in the event log but are added to see where the process starts and ends. Semantically, one can extend each case with one *dummy start event* at the beginning and one *dummy end event* at the end. If an activity is followed by an activity , there is a directed edge from node to node . Moreover, the edge can be annotated by the *frequency* of the relation. Hence, if the edge connecting to has a frequency of 2987, then was followed by 2987 times. Note that for these directly-follows relations, we only consider events within the same case. Therefore, we can simplify event data into a multiset of traces, as described before. Based on the example event log containing 12,666 cases and seven unique activities, we obtain the following DFG:



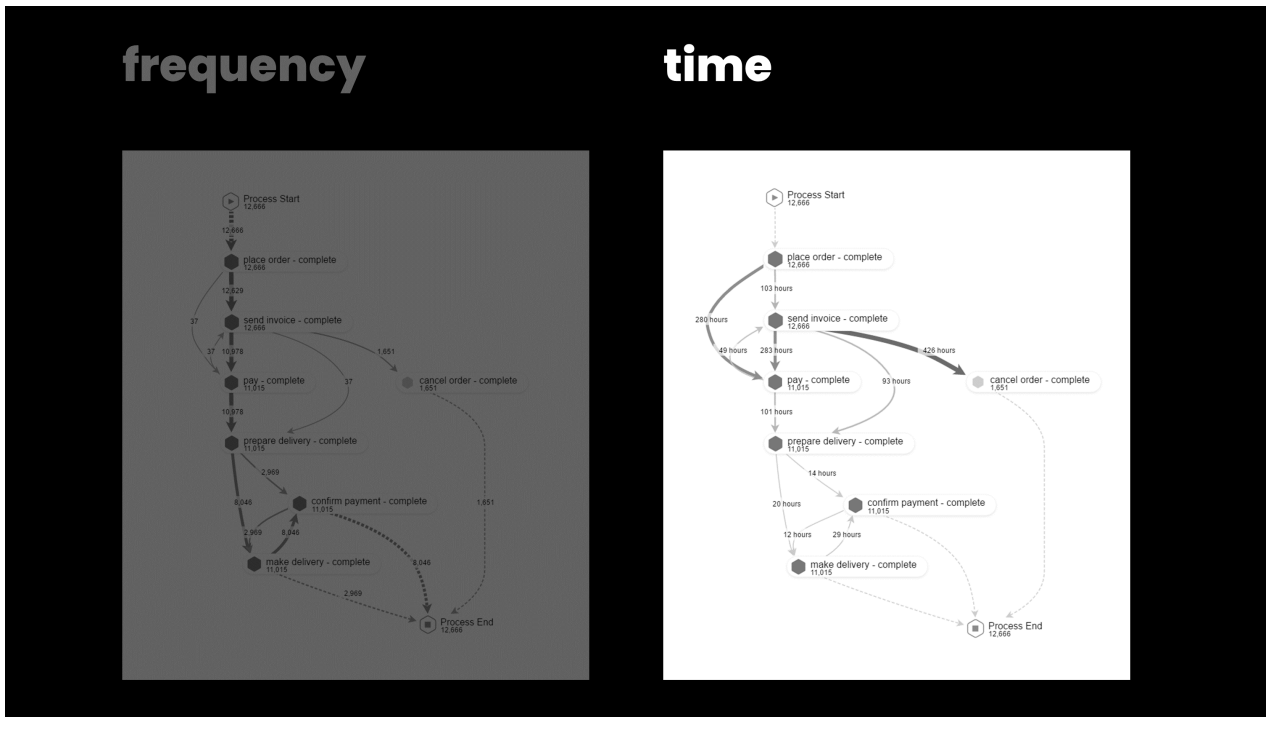
All 12,666 cases start with activity **place order**. Therefore, the arc connecting the artificial start has a frequency of 12,666. Activity **place order** can be followed by activity **send invoice** or activity **pay**. Therefore, there is an arc connecting **place order** to **send invoice** and an arc connecting **place order** to activity **pay**. The frequency of the first arc is 8,016 + 1,651 + 2,962 = 12,629. The frequency of the second arc is 30 + 7 = 37. For this simple example, it is possible to compute all frequencies by hand. For example, activity **send invoice** is directly followed by activity **prepared delivery** 37 times and activity **send invoice** is directly followed by activity **cancel order** 1651 times. For more complex processes, this is, of course, impossible. However, process mining software can compute these frequencies extremely fast (one pass through the event log suffices).





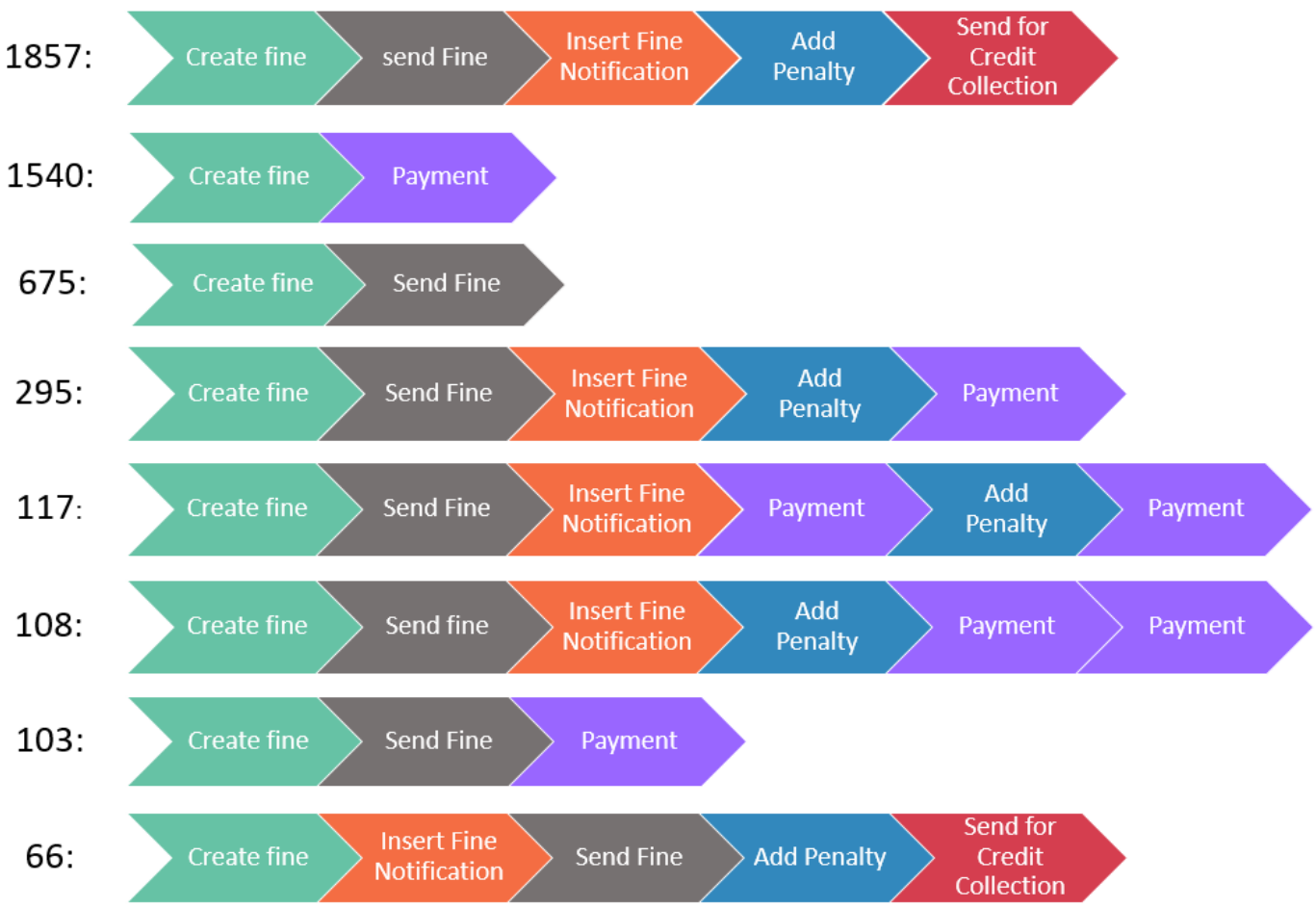
Although we abstracted from timestamps to create a DFG (they were only used to order events), each event has a *timestamp*. Therefore, If an activity is followed by an activity for a specific case, we know the *time difference* between both events. For example, activity **send invoice** is directly followed by activity **prepare delivery** 37 times, this means that we can produce 37

measurements, i.e., 37 times we can measure the time difference between two subsequent events for the same case. Using these 37 observations, we can compute the mean, median, minimum, maximum, variance, etc. The above DFG shows the mean time between activities. For example, the mean time between activity **send invoice** and activity **prepare delivery** is 93 hours.



To summarize, given event data it is fairly simple to produce a DFG showing frequencies and time information. The approach presented is *scalable* and diagrams are *easy to interpret* as long as one realizes that frequencies and times *refer to directly-follows relationships only*. The mean time between **send invoice** and **prepare delivery** of 93 hours only relates to the cases where people paid before getting an invoice. In most cases, it takes longer because people still need to pay after receiving the invoice.

▼ Exercice



Question 1

1/1 point (ungraded)

Which of the following statements are correct?

- ☐ The DFG starts with *Create Fine* 4072 times.
- ☐ The DFG ends with *Payment* 1835 times
- ☐ **Create Fine is the only start activity.**
- ☐ **There are three end activities, which are *Payment*, *Send for Credit Collection*, and *Send Fine*.**
- ☐ *Send Fine* is followed by *Insert Fine Notification* 2443 times.

- **Starts with Created fine**: 1857 (Created fine) + 1540 + 675 + 295 + 117 + 108 + 103 + 66 = **4761**

- **Ends with Payment**: 1540 + 295 + 117 + 108 + 103 = **2163**
- **Send Fine is followed by Insert Fine Notification**: 1857 + 295 + 117 + 108 = **2377**

Correct (1/1 point)

Question 2

1/1 point (ungraded)

Which of the following traces are allowed by the directly-follows graph (general process path)?

- ☐ Trace 1: *Create Fine, Send Fine, Insert Fine Notification, Send for Credit Colllection.*
- ☐ **Trace 2: *Create Fine, Send Fine, Insert Fine Notification, Payment, Payment***
- ☐ **Trace 3: *Create Fine, Payment, Add Penalty, Send for Credit Colllection***

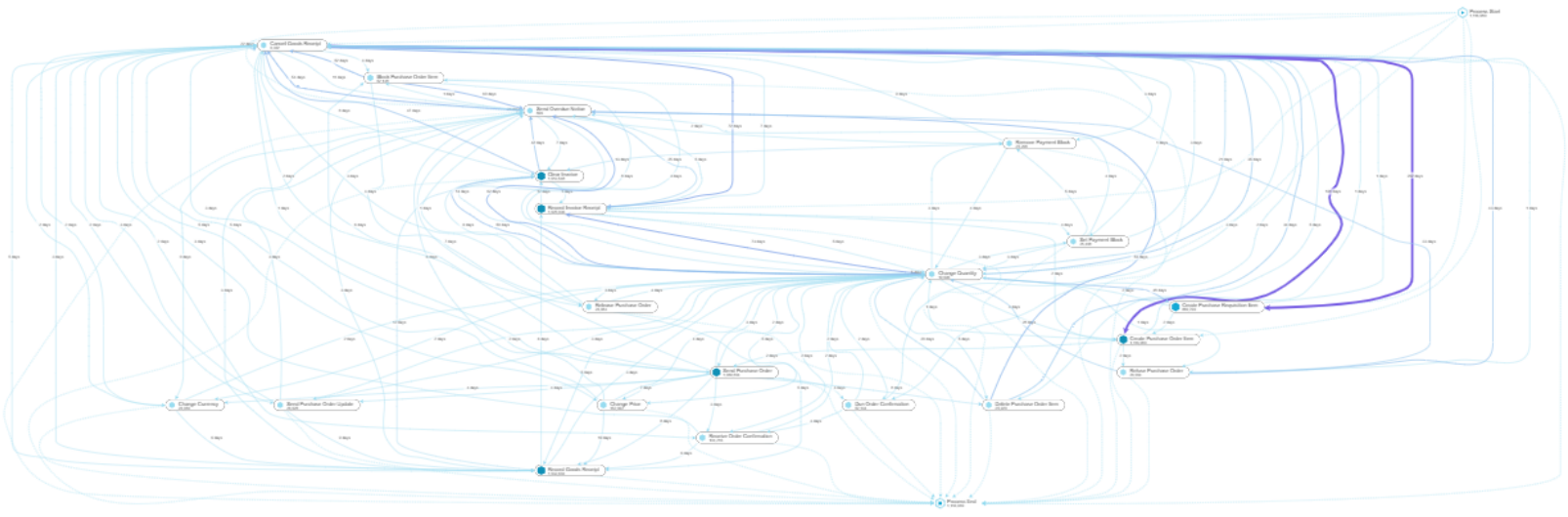
Correct (1/1 point)

▼ Simplifying DFGs Using Filtering

Using the baseline DFG-based discovery algorithm, an activity appears in the discovered DFG when it occurs at least once and two activities and are connected by a directed arc if is directly followed by at least once in the log. Often, we do not want to see the process model that captures all behavior. Instead, we would like to see the dominant behavior. For example, we are interested in the most frequent activities and paths. Therefore, we would like to filter the event log and model. Here, we consider the three basic types of filtering:

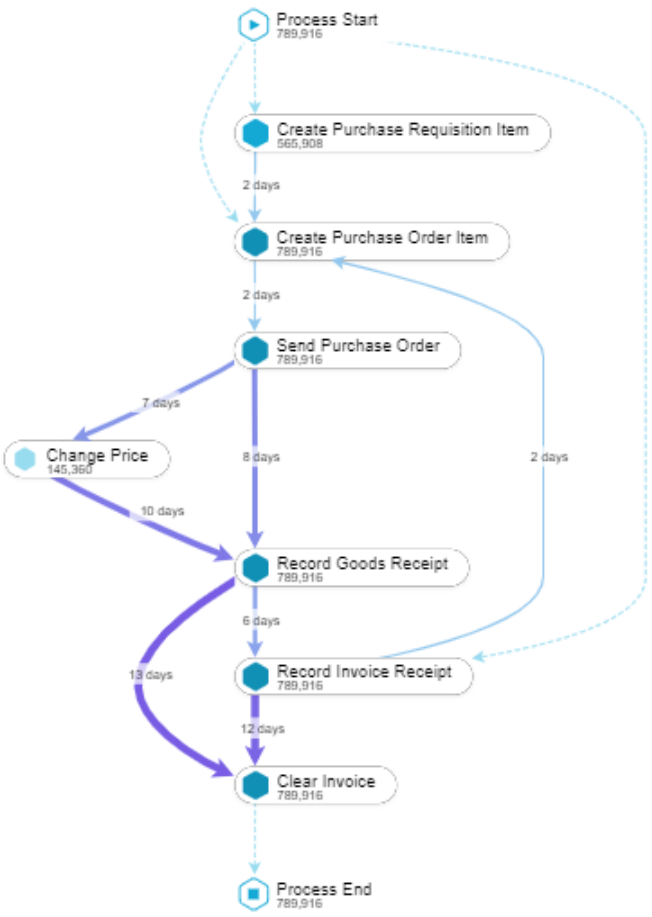
- *Activity-based filtering*: project the event log on a subset of activities (e.g., remove the least frequent activities).
- *Arc-based filtering*: remove selected arcs in the DFG (e.g., delete arcs with a frequency lower than a given threshold).
- *Variant-based filtering*: remove selected traces (e.g., only keep the most frequent variants).

Filtering is often necessary because process models are too complex or underfitting. DFGs tend to be "Spaghetti-like" when there are many unique activities and trace variants. The complexity is not caused by the volume of data but by the variability or behavior. Even in simple P2P (Purchase to Pay) and O2C (Order to Cash) processes we may find thousands of different traces including traces that are unique (i.e., only one case followed this path through the process.

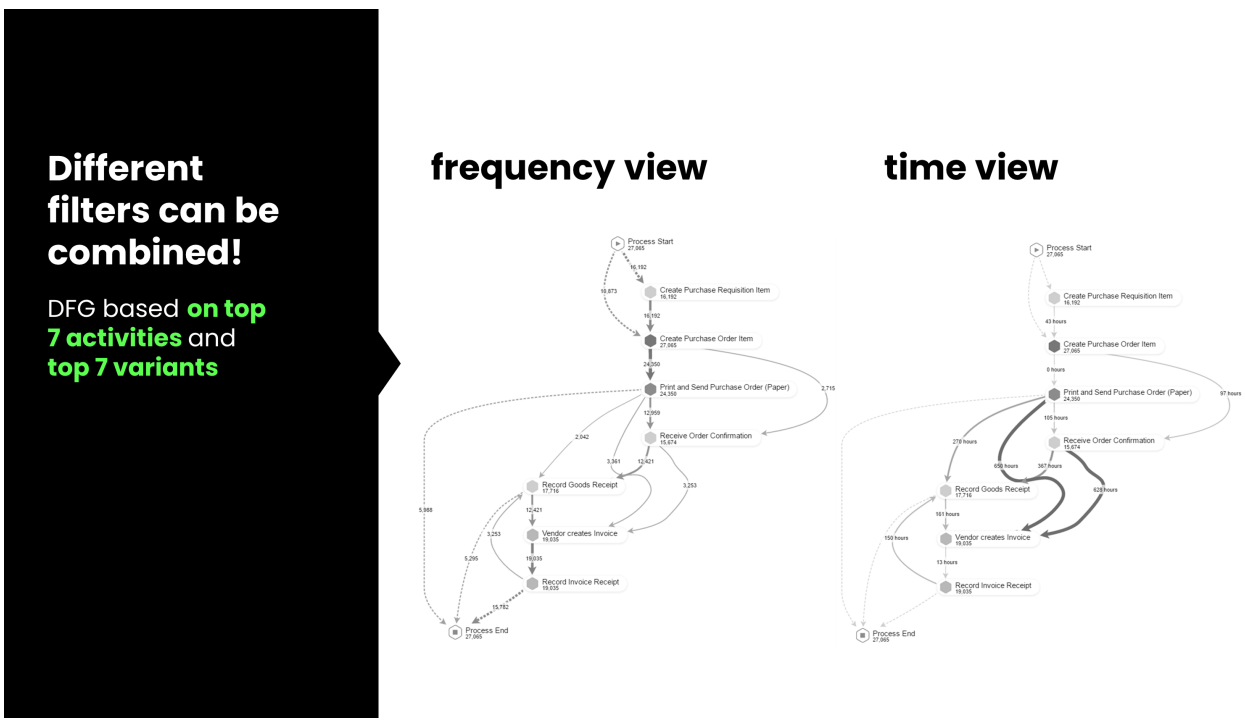


- Using *activity-based filtering*, we simply remove specific activities from the event data. For example, we remove all activities that happened less than 30,000 times. After removing these activities, we construct a new event log and create a DFG in the usual manner. Note that activity-based filtering does not remove cases, only events corresponding to selected (e.g., infrequent activities).

- *Variant-based filtering* removes complete traces instead of selected events. Each case corresponds to a trace variant (i.e., a sequence of activities). Trace variants can be sorted by frequency. One may choose to remove all variants that happened less than 10 times. This means that we removed the corresponding cases from the event data and created a DFG in the usual manner. Both activity-based and variant-based filtering have a clear interpretation because they relate to a simplified event log where events corresponding to selected activities and cases corresponding to selected traces are removed.



- *Arc-based filtering* is less recommendable. In this approach, arcs in the DFG that are infrequent are removed. However, this does not have a clear relationship to the event data. Moreover, arc-based filtering is often combined with removing disconnected activities and other repair actions. As a result, the semantics are less clear. Also, note that the frequency of an activity does not need to match the frequencies of the ingoing and outgoing arcs. Given an activity that happened 800 times, the sum of the frequencies of the ingoing edges may be 456, and the sum of the frequencies of the outgoing edges may be 611. Therefore, arc-based filtering should only be used to get a first impression. *For a proper analysis of the process, one should use a combination of activity-based and variant-based filtering.*



For our event logs with 12,666 cases, we can select the 7 most frequent activities followed by the 7 most frequent variants. The resulting DFG is depicted above using both the frequency and time views. The DFG is much simpler, but still captures most of the behavior. Note that most processes have a Pareto distribution, e.g., 80% of the cases can be described by only 20% of the process variants. It is often easy and desirable to create a process model describing these 80% of cases. If the distribution of cases over variants does not follow a Pareto distribution, then it is best to first apply activity-based filtering. When applying activity-based filtering, the number of variants typically decreases dramatically, and a Pareto distribution emerges.

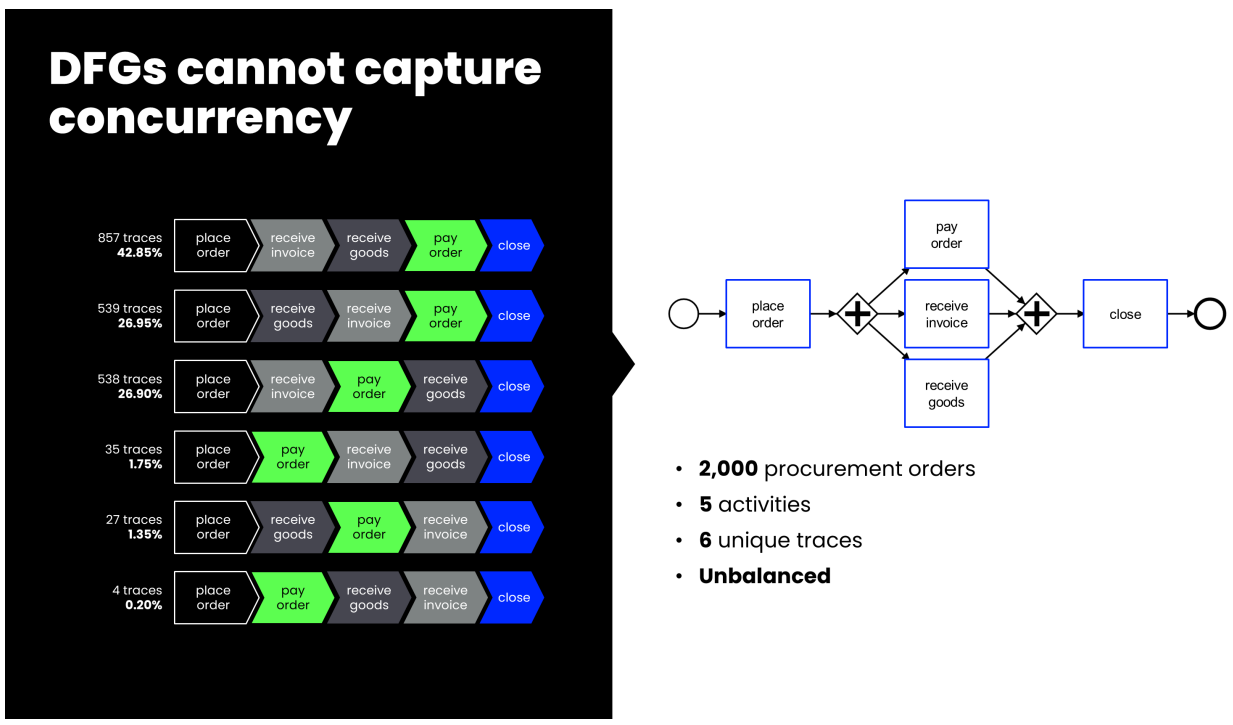
Activity-based and variant-based filtering can be combined with any process-discovery technique because a new simplified event log is created. *Hence, it is a generic approach independent of the process-model representation and discovery technique.*

▼ Limitations of DFGs

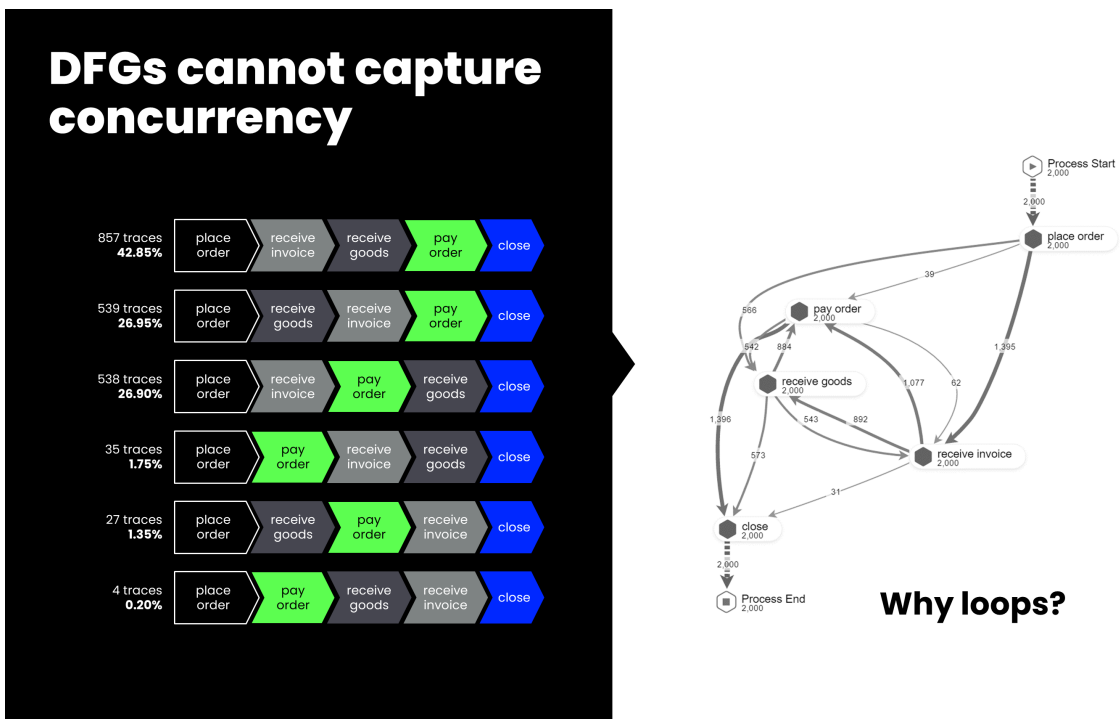
Concurrency means that activities can be executed in any orders or in the same time for different cases like in the image bellow in left side, this situation can be represented with BMP technique to design the situation like in the right side of image below

DFGs *cannot capture concurrency*. A DFG is a sequential model, i.e., it only considers the directly-follows relation and not the causal relation between activities. Whenever two activities do not happen in a fixed order, the DFG creates loops, This is best

illustrated using an example:



The process consists of five activities. All five activities are executed once for all 2000 cases. The corresponding traces and their frequencies are shown. Each case starts with **place order** and ends with **close**. The three other activities can be executed in any order, ***resulting in 6 trace variants***. This behavior can easily be presented using a ***BPMN model or a Petri net***. However, when discovering a DFG, one finds the following model:



The DFG contains many loops despite the fact that the five activities are executed precisely once for each case. The DFG allows for skipping the activities in the middle or executing these activities any number of times. Based on the DFG, it is possible that for a single order, 30 invoices were sent, and the goods were received 31 times, but the customer never paid. *There is no DFG that describes the observed behavior properly.* Due to the loops introduced, the model is underfitting and Spaghetti-like. This clearly shows the ***need for more advanced discovery techniques.***

> 💡 *Inductive mining approche - solve* **Concurrency**

▼ Exercice

Question 1

Which of the following statements are correct?

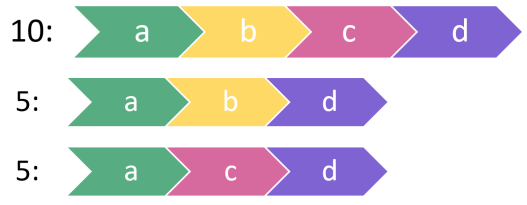
- ☐ After applying arc-based filtering on a DFG, the frequencies of a node always match the add-up frequencies of the node's outgoing arcs.
- ☐ **There are three types of filtering techniques, which are activity-based, arc-based, variant-based filtering.**
- ☐ One cannot combine different kinds of filtering techniques for DFGs.
- ☐ **Hidden activities might influence directly-follows frequency.**
- ☐ **DFGs cannot capture concurrency.**

Correct (1/1 point)

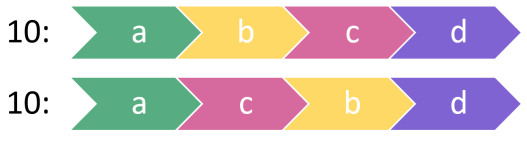
Question 2

Consider the following event logs.

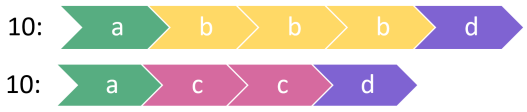
Event log 1:



Event log 2:



Event log 3:



Draw a directly-follows graph for each of the event logs. Which of them shows the weakness of illustrating concurrency.

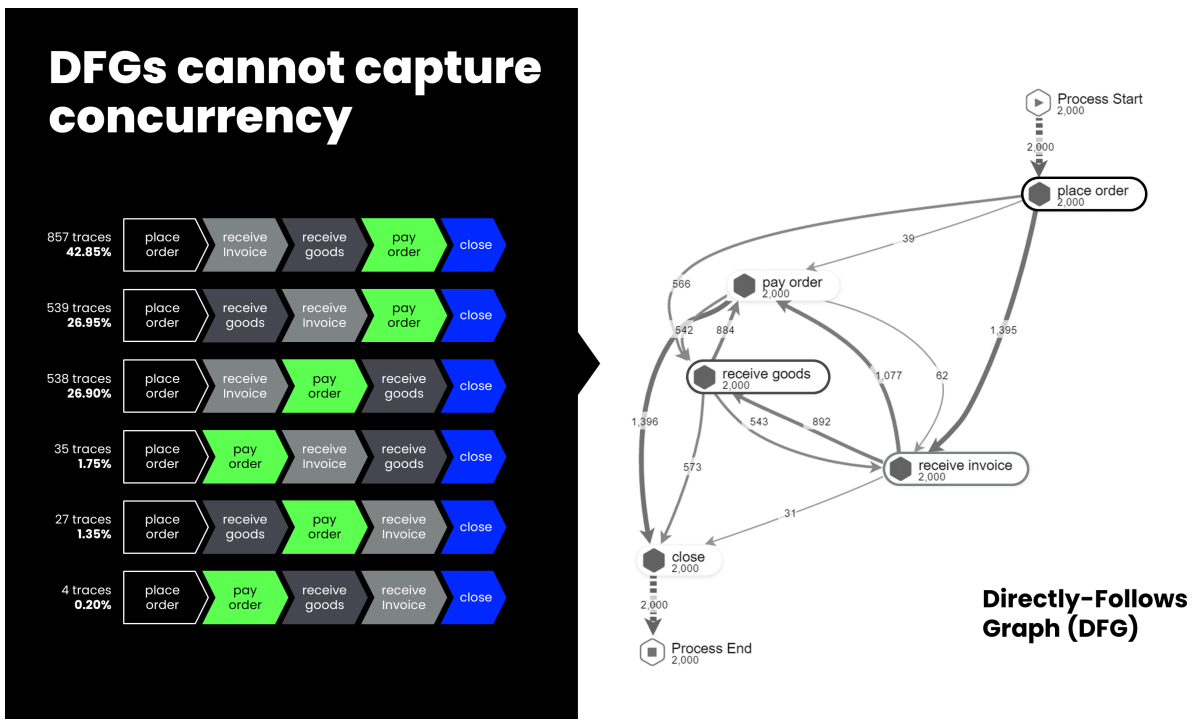
- ☐ Event log 1
- ☐ **Event log 2**
- ☐ Event log 3

Correct (1/1 point)

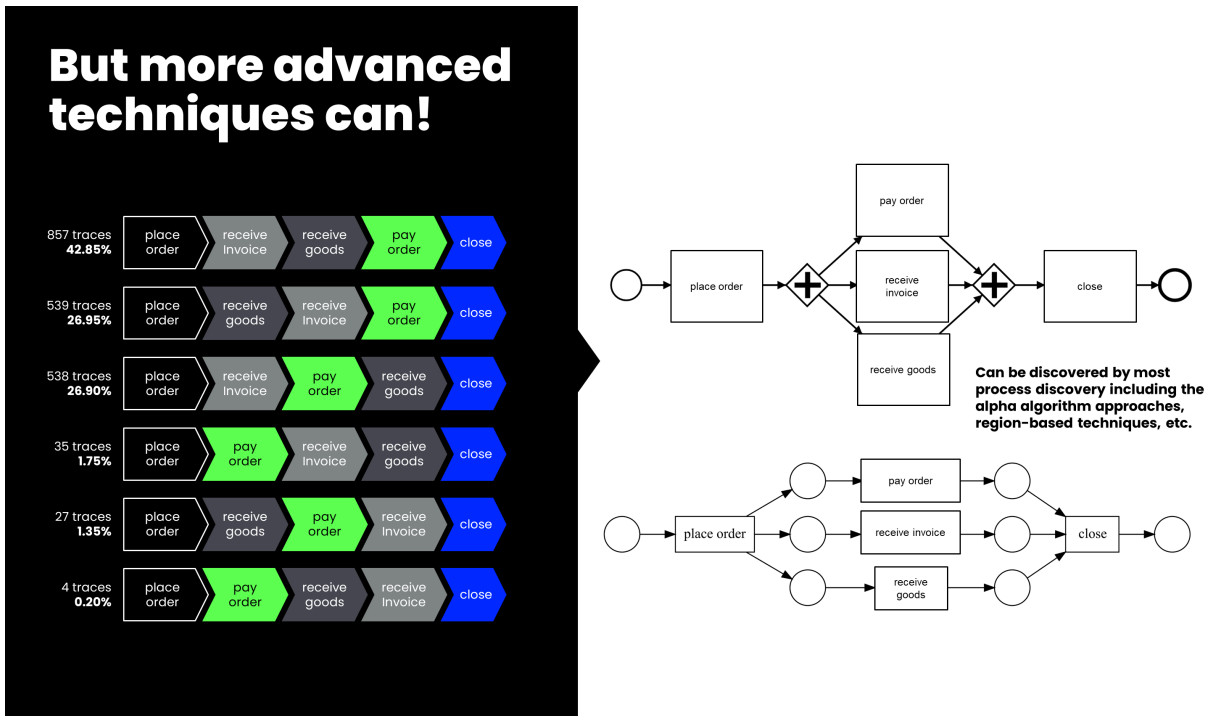
▼ Discover Sophisticated Process Models

▼ Capturing Concurrency

Two activities are *concurrent* if they may be executed in any order or even at the same time. When cooking a meal involving cooking rice and cutting unions, we may sometimes cook the rice first and then cut the unions, but also do it in reverse order, or cut the unions while the rice is being cooked. In a DFG, this results in loops because there is a connection in both directions. This is illustrated by the following example:

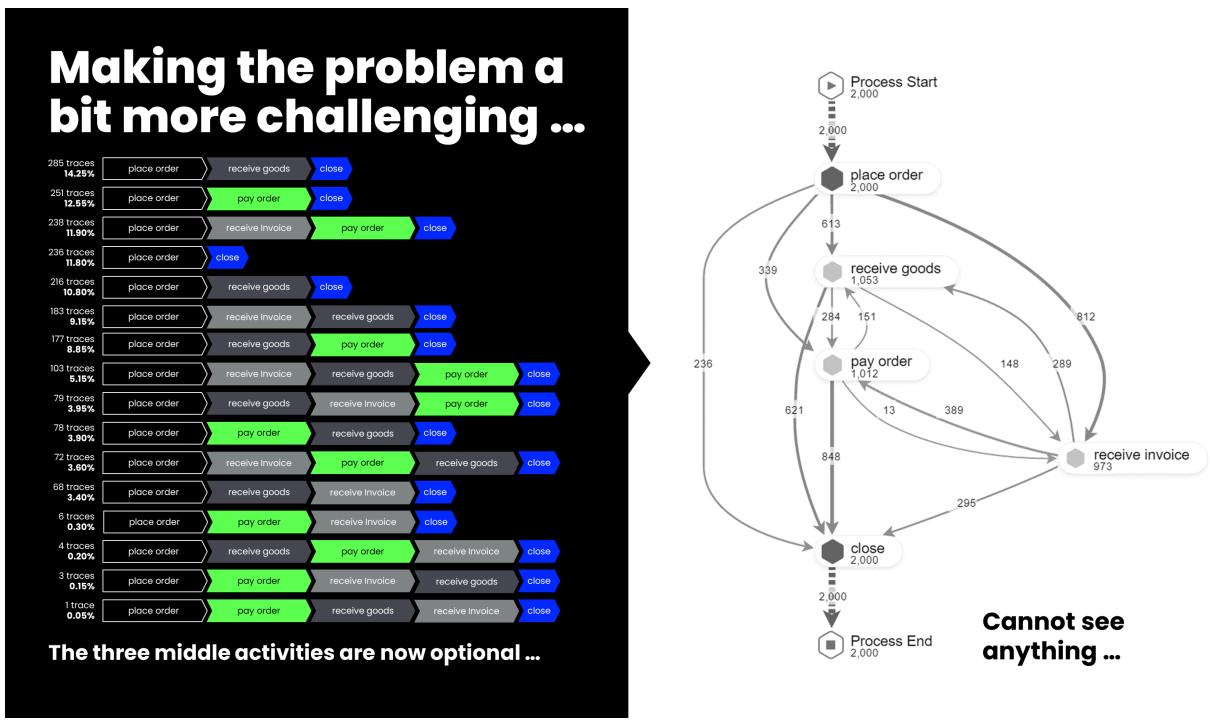


Activity **pay order** is followed by activity **receive invoice** and vice versa. Activity **pay order** is followed by activity **receive goods** and vice versa. Activity **receive goods** is followed by activity **receive invoice** and vice versa. Considering the three activities in the middle, there are many loops involving two or three activities. However, in the event log, activities happen only once per case. However, using *Business Process Model and Notation (BPMN)* or *Petri nets*, the process can be described perfectly for the sole reason that they are able to express concurrency:

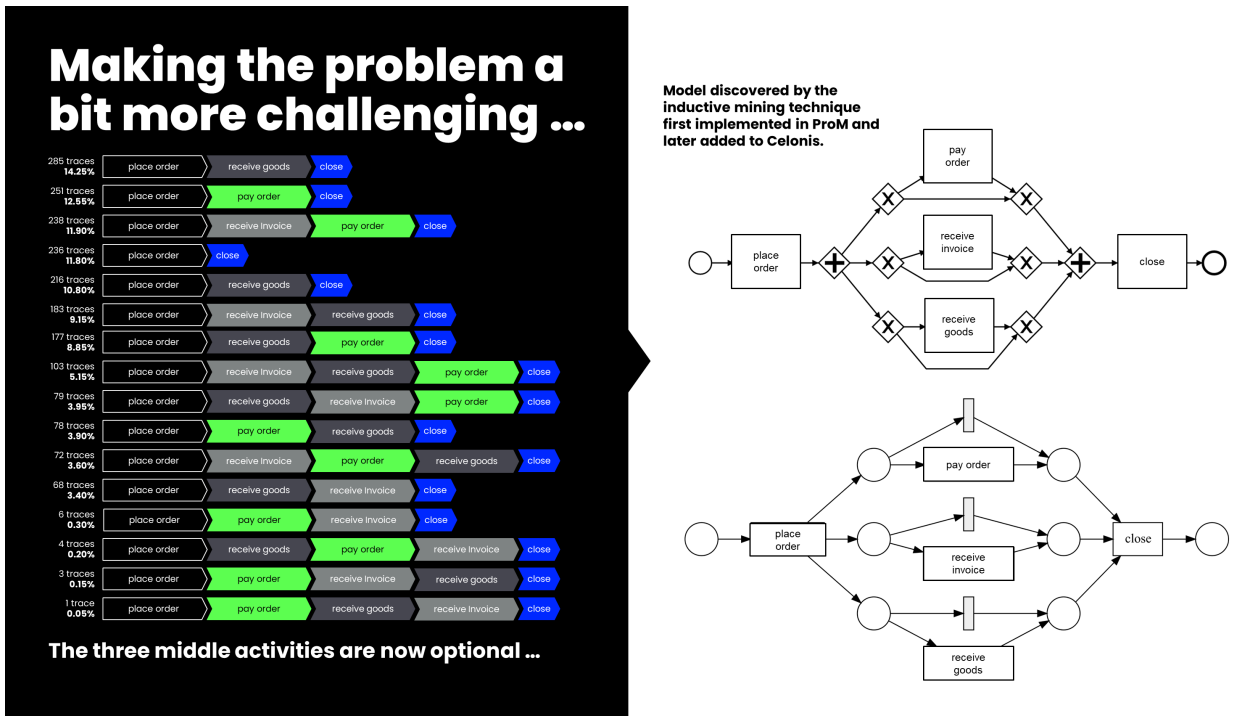


Both the BPMN model and Petri net describe a process that allows for the six traces observed in the data. Note that the DFG allows for infinitely many traces due to the loops.

Let's make the example a bit more interesting. We now make the activities **pay order**, **receive invoice**, and **receive goods** optional. Again, we have an event log with 2000 cases.



We see similar problems when looping at the DFG. There are loops despite the fact that each activity was executed at most once per case. It is also challenging to see what the difference is with the previous DFG. **Using more sophisticated discovery techniques,** we can discover a BPMN model or Petri nets that both capture the observed behavior much better:



Business Process Model and Notation (BPMN) is the de facto representation for business process modeling in industry. The BPMN standard is maintained by the Object Management Group (OMG), is supported by a wide range of vendors, and is used by most of the larger organizations. For process mining, the constructs for control-flow are most relevant. Moreover, most tools only support

a small subset of the BPMN standard, and an even smaller subset is actually used on a larger scale. Therefore, we only consider *start* and *end events*, *activities*, *exclusive gateways*, *parallel gateways*, and *sequence flows*. The BPMN model shows the five activities starting with **place order** and ending with **close**. The two parallel gateways are indicated by the **+** symbol and correspond to an *AND-split* and *AND-join*. There are six exclusive gateways indicated by the x symbol. Three exclusive gateways represent *XOR-splits* and three exclusive gateways represent *XOR-joins*. These gateways allow skipping activities **pay order**, **receive invoice**, and **receive goods**. Together, these gateways model exactly what was seen in the event log.

Petri nets are the oldest and best-known notation to represent concurrency. Languages like BPMN are based on ideas and concepts first developed in Petri nets. It is possible to convert one into the other. Compared to BPMN, Petri nets have fewer concepts and a stronger theoretical foundation. Petri nets are, therefore, the de facto standard in process mining research. However, most of the commercial tools use BPMN representations rather than Petri nets for visualization. Yet, internally, often Petri nets are used. In the context of process mining and business process management, one can view BPMN as "syntactic sugaring" for Petri nets. Therefore, it is important to know both.

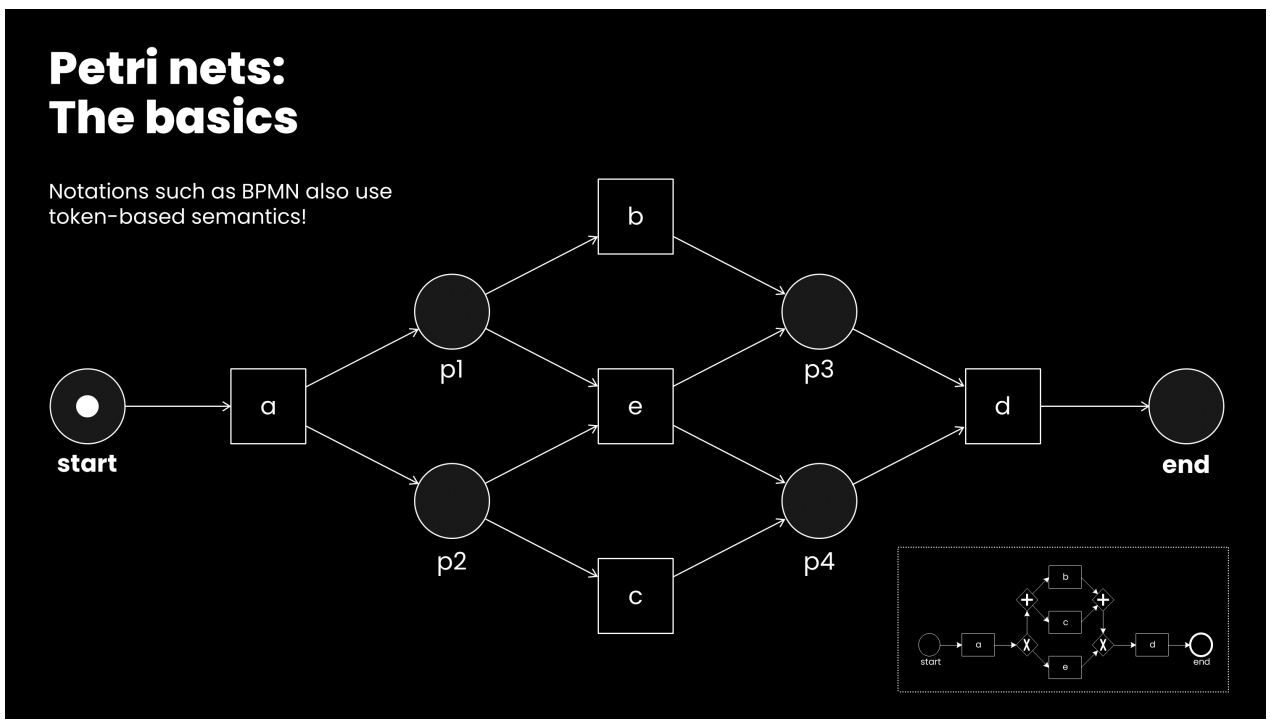
The Petri net in the diagram above describes the same process as the BPMN model and allows for the traces seen in the event log. The circles correspond to places (to model states) and the rectangles correspond to transitions (to model activities). To model the skipping of the activities **pay order**, **receive invoice**, and **receive goods**, so-called silent transitions are used. These are depicted using the smaller grey rectangles without an activity label.

These examples show that DFGs are great for obtaining a first impression of the process, but for a more fine-grained analysis, it is better to use a notation able to express concurrency. The examples also show the need for more sophisticated discovery techniques, like inductive mining.

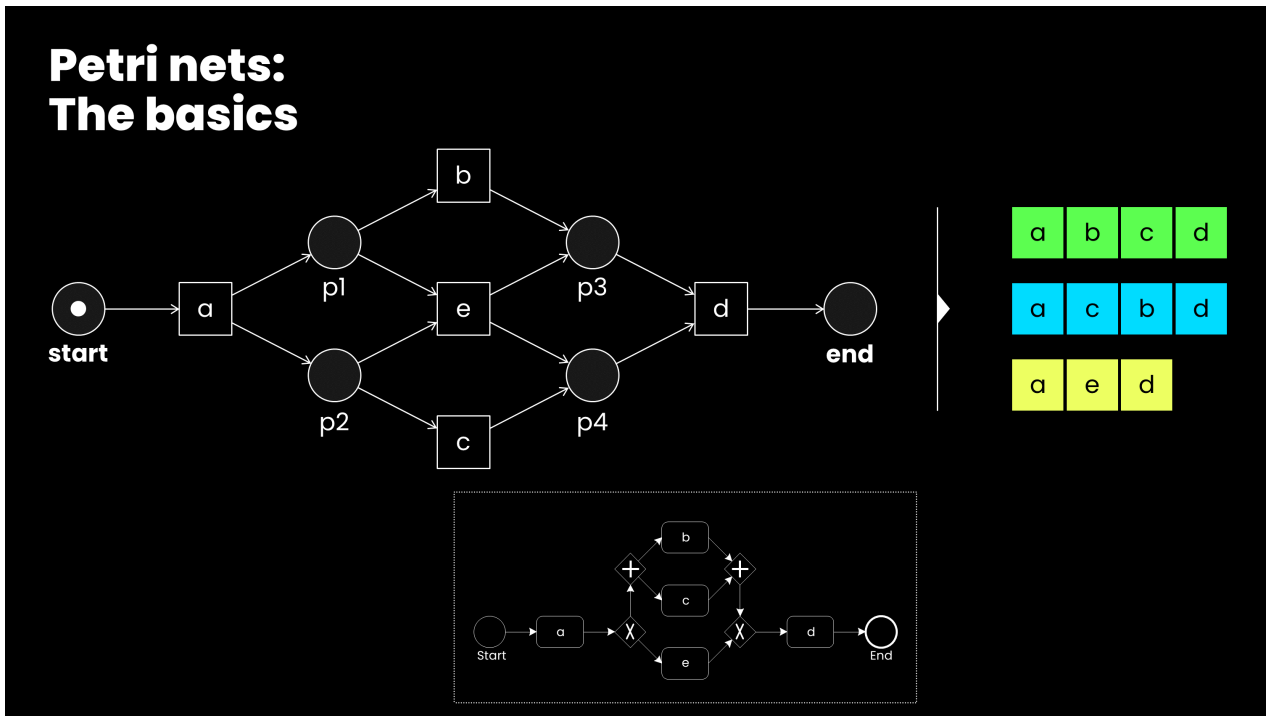
> 💡 Using sophisticated discovery techniques, we can uncover BPMN models or Petri nets that capture observed behavior more effectively. BPMN and Petri nets can perfectly describe processes due to their ability to express concurrency. ***In Petri nets, diagrams use tokens to represent input and output flows, ensuring that the number of tokens in the input matches the number in the output, in BPMN the notation*** X ***express 1 choice & + express parallels choices***

▼ Petri Nets Introduction

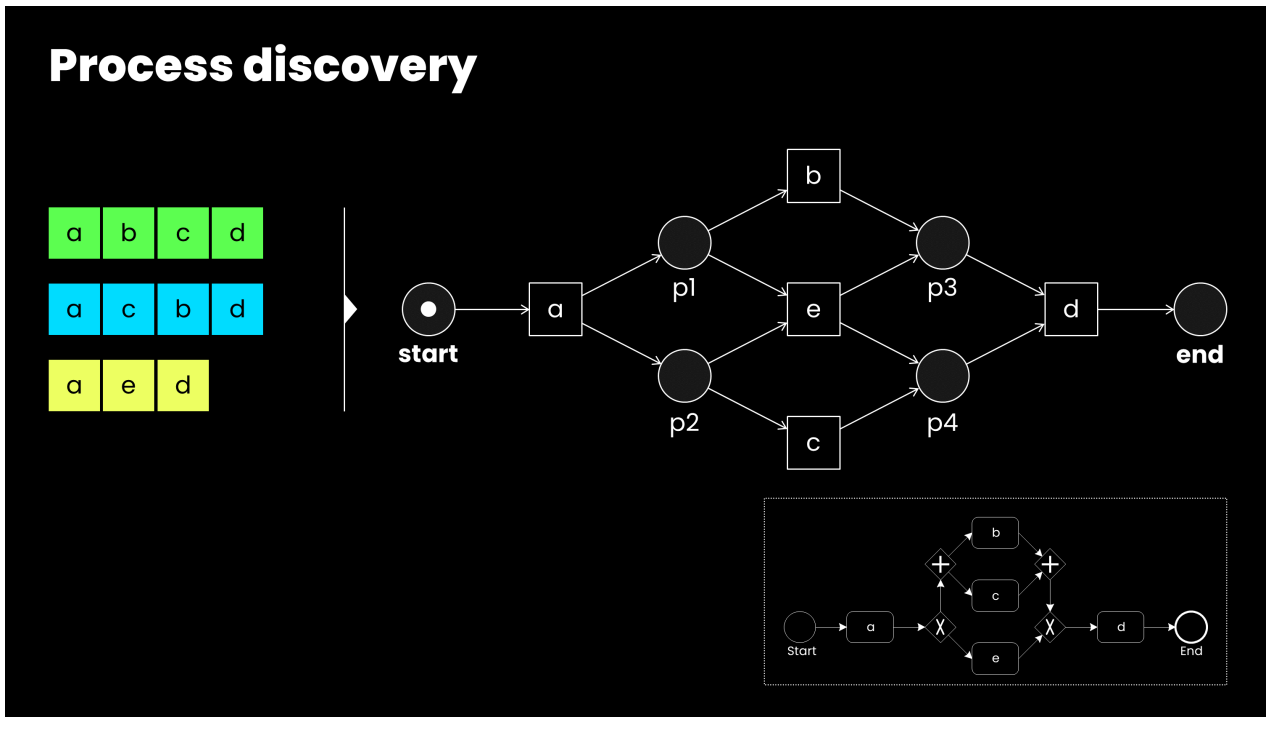
Notations such as BPMN also use the token-based semantics of Petri nets. To explain Petri nets, we use a simple abstract example. The corresponding BPMN model is also shown in small.



The Petri net consists of six *places* $(start, p1,\ p2,\ p3, p4\ , end)$ and five *transitions* $(a,\ b,\ c,\ d, and\ e)$. The transitions (i.e., squares) correspond to the five activities seen in the event log (or the activities one expects to see). The places (i.e., circles) constrain the behavior. States in Petri nets are called *markings* that mark certain places (represented by circles) with tokens (represented by black dots). In the example, only the $start$ place is marked with a token. A transition is called enabled if each of the input places has a token. An enabled transition may fire (i.e., occur), thereby consuming a token from each input place and producing a token for each output place.



The Petri net allows for three traces: (a, b, c, d), (a, c, b, d), and (a, e, d). Initially, only transition a is enabled. When a fires (i.e., occurs), a token is consumed from the input place and a token is produced for each of the two output places. As a result, transitions b, c, and e become enabled. If e fires, both tokens are removed and two tokens are produced for the input places of d . If b fires, only one token is consumed and one token is produced. After b fires, c is still enabled, and c will fire to enable d. Transition c can also occur before b, i.e., b and c are concurrent and can happen at the same time or in any order. There is a choice between e and the combination b of c and . The start of the process is modeled by the token in the source place labeled $start$. The end of the process is modeled by the sink place labeled end.

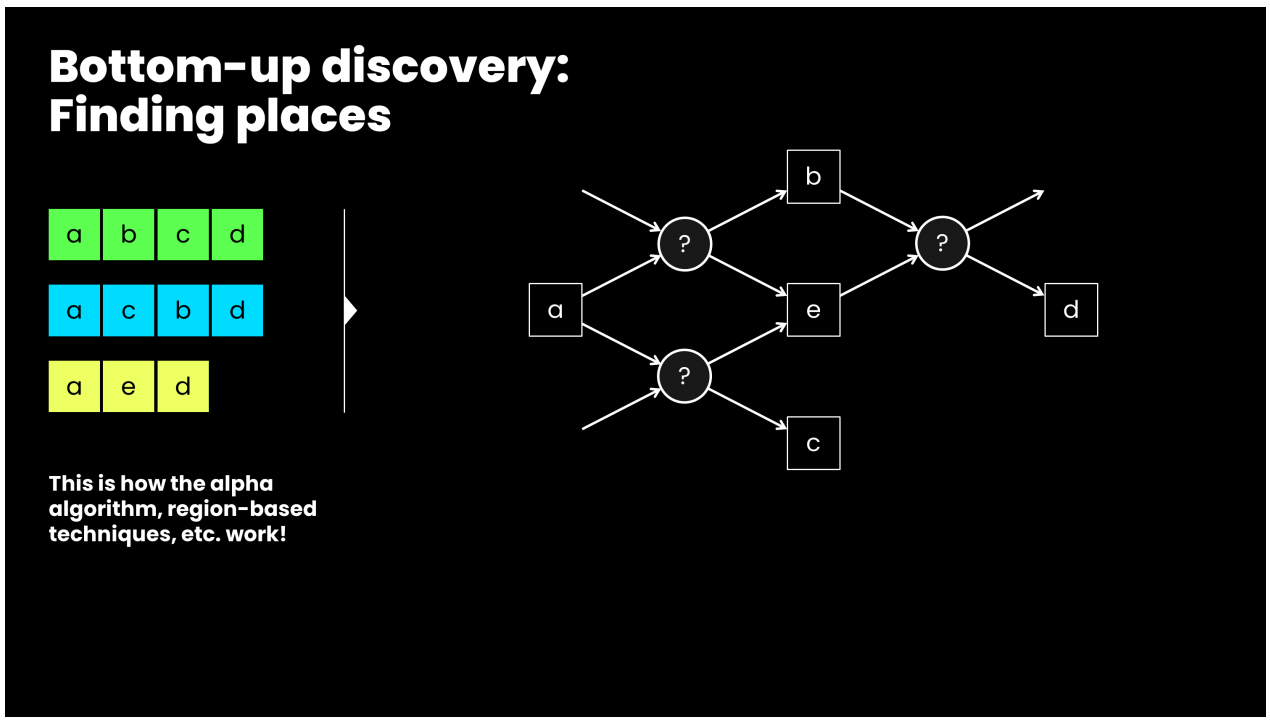


Using process discovery, the goal is to discover a Petri net that describes the traces in the event log. Next to the problem of discovering concurrency, we face the problem that we cannot assume that what did not happen cannot happen, i.e., we do not have negative examples and cannot assume that the event log is complete. This also makes evaluation difficult.
Nevertheless, there is consensus in the process mining community that there are the following four quality dimensions to evaluate a process mode in the context of an event log.

- *Recall*, also called (replay) fitness, aims to quantify the fraction of observed behavior in the event log that is allowed by the process model (e.g., Petri net).
- *Precision* aims to quantify the fraction of behavior allowed by the model that was actually observed (i.e., avoids "underfitting" the event data).
- *Generalization* aims to quantify the probability that new unseen cases will fit the model (i.e., avoids "overfitting" the event data).
- *Simplicity* refers to Occam's Razor and can be made operational by quantifying the complexity of the model (number of nodes, number of arcs, understandability, etc.).

There exist various measures for recall. The simplest one computes the fraction of traces in the event log possible according to the process model. It is also possible to define such a notion at the level of events. There are many simplicity notions. These do not depend on the behavior of the model, but measure its understandability and complexity. Most challenging are the notions of precision and generalization. Also, these notions can be quantified, but there is less consensus on what they should measure. The goal is to strike a balance between precision (avoiding "underfitting" the sample event data) and generalization (avoiding "overfitting" the sample event data). A detailed discussion is outside the scope of this course.

Returning to Petri nets as representation, we would like to stress that the *places in a Petri net should be seen as constraints*.



Apart from the start and end places, the other places are empty at the beginning, should be empty at the end of the trace, and at no point in time have a negative number of tokens (i.e., attempt to consume a token that is not present). Each place models a *constraint* related to the input and output transitions.

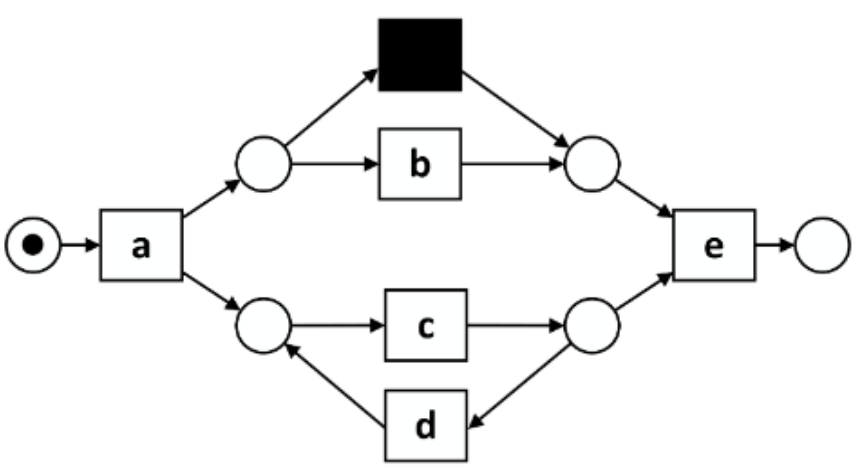
Some discovery approaches use a so-called *bottom-up* approach. They start with a Petri net without any place allowing for any behavior involving the activities represented as transitions. Such models are called "flower models". Then places are added one-by-one further constraining the behavior. Techniques such as the alpha algorithm and the region-based approaches use such an approach. For more details, we refer to the book Process Mining: Data Science in Action and the book chapter Foundations of Process Discovery.

▼ Exercice

Question 1

1/1 point (ungraded)

Consider the following Petri net model and select the correct statements.

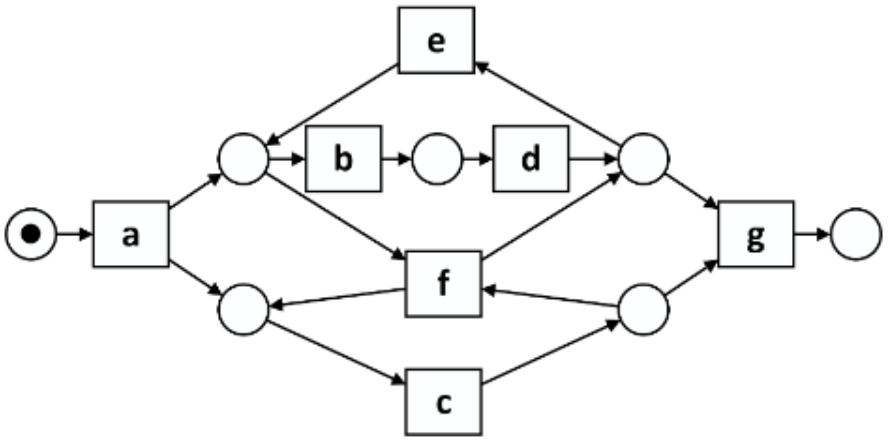


- [x] The number of traces that are allowed by this model is infinite.
- [ ] The trace $\langle a, b, c, d, e \rangle$ is allowed by the model.
- [x] The trace $\langle a, c, d, c, b, d, c, e \rangle$ is allowed by the model.
- [ ] The shortest trace that is allowed by this model has a length of 4.
- [x] A complete trace always starts with activity *a* and ends with activity *e* .

Question 2

1/1 point (ungraded)

Consider the following Petri net model and select the correct statements.



- [ ] After executing *a,b,c,d* activities g, f, or e can occur.
- [x] The trace $\langle a, c, f, c, g \rangle$ is allowed by the model.
- [ ] The trace $\langle a, b, c, d, e, b, d, c, g \rangle$ is allowed by the model.

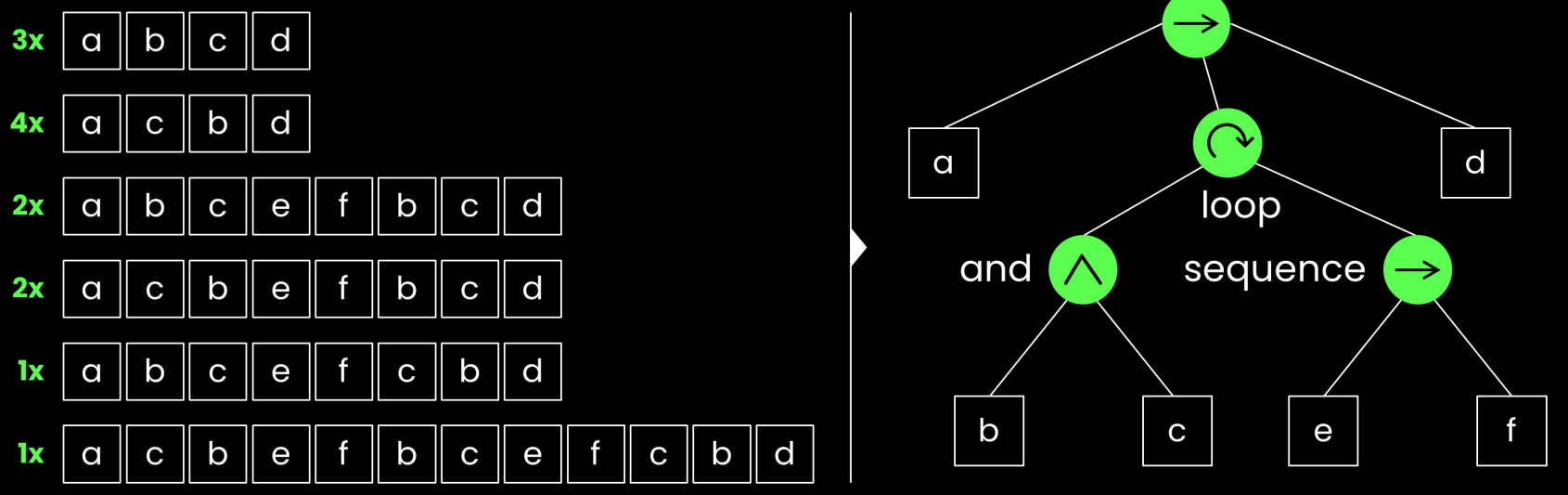
✔

▼ Discovery and Advantages of Higher Level Process Models

Inductive mining techniques use a *top-down approach*, i.e., an event log is repeatedly decomposed into smaller parts. In this process, a so-called *process tree* is created. Process trees are not commonly used as a modeling language. However, state-of-the-art process discovery techniques use process trees as an *internal* representation. The behavior of process trees can be easily visualized using Petri nets, BPMN, UML activity diagrams, EPCs, etc. However, they also have their own graphical representation. The main reason for using process trees is that they have a hierarchical structure and are *sound by construction*, i.e., they do not have deadlocks, livelocks, or disconnected parts. This does not automatically hold for other notations such as Petri nets and BPMN.

A process tree is a *tree-like structure* with one *root node*. The leaf nodes correspond to *activities* (including the silent activity τ, which is similar to a silent transition in Petri nets). Four types of operators can be used in a process tree: $\rightarrow$ (*sequential composition*), $\times$ (*exclusive choice*), $\wedge$ (*parallel composition*), and $\circlearrowleft$ (*redo loop*).

Top-down discovery: Inductive miner



The process tree (right) was discovered based on an event log (left). The tree has one root node and six leaf nodes corresponding to activities in the event log. The process tree can also be described textually: $\rightarrow (a, \circlearrowleft (\wedge (b, c), \rightarrow (e, f)), d)$. The fragment $\rightarrow (e, f)$ models a sequence of two activities: e is followed by f. The fragment $\wedge (b, c)$ models the concurrent execution of b and c, i.e., b and c are executed once, but in arbitrary order. The redo loop $\circlearrowleft$ operator is a bit more involved. The left-hand-side child is executed at least once. After executing the left-hand-side child, the subprocess may end, or the right-hand-side child is executed. In the latter case, the left-hand-side child is executed again, etc. Hence, the left-hand-side child is executed n times, the right-hand-side child is executed n-1 times, and both alternate. For example, $\circlearrowleft (x, y)$ corresponds to the traces $\langle x \rangle$, $\langle x, y, x \rangle$, $\langle x, y, x, y, x \rangle$, $\langle x, y, x, y, x, y, x \rangle$, $\langle x, y, x, y, x, y, x, y, x \rangle$, etc. The operators can be used to combine subprocesses where a single activity is the simplest subprocess possible. Hence, x and y can also be executions of complete subprocesses. Therefore, $\circlearrowleft (\wedge (b, c), \rightarrow (e, f))$ corresponds to traces such as $\langle b, c \rangle$, $\langle c, b \rangle$, $\langle b, c, e, f, b, c \rangle$, $\langle c, b, e, f, b, c \rangle$, $\langle b, c, e, f, c, b \rangle$, $\langle c, b, e, f, c, b \rangle$, $\langle c, b, e, f, c, b, e, f, c, b \rangle$, etc. $\rightarrow (a, \circlearrowleft (\wedge (b, c), \rightarrow (e, f)), d)$ models a sequential process that starts with a and ends with d, therefore, $\rightarrow (a, \circlearrowleft (\wedge (b, c), \rightarrow (e, f)), d)$ corresponds to $\langle a, b, c, d \rangle$, $\langle a, c, b, d \rangle$, $\langle a, b, c, e, f, b, c, d \rangle$, $\langle a, c, b, e, f, b, c, d \rangle$, $\langle a, b, c, e, f, c, b, d \rangle$, $\langle a, c, b, e, f, c, b, d \rangle$, $\langle a, c, b, e, f, c, b, e, f, c, b, d \rangle$, etc.

The redo loop $\circlearrowleft$ operator may seem a bit odd. However, it is a combination of a while-do and repeat-until used in programming. Let S be a subprocess. $\circlearrowleft(S, \tau)$ corresponds to a repeat-until of subprocess S, i.e., S is in the "do part" and is executed one or more times. $\circlearrowleft(\tau, S)$ corresponds to a while-do of subprocess S, i.e., S is in the "redo part" and is executed zero or more times. For example, $\circlearrowleft(x, \tau)$ corresponds to the traces $\langle x \rangle$, $\langle x, x \rangle$, $\langle x, x, x \rangle$, $\langle x, x, x, x \rangle$, etc., and $\circlearrowleft(\tau, x)$ also includes the empty trace.

The example does not include the exclusive choice $\times$ operator. However, the behavior of this operator is easy to understand; the operator simply picks one of its children. Hence, $\times(x, y)$ makes a choice between x and y. Operators can also have more than two children, e.g., $\rightarrow(a, b, c, d)$, $\wedge(a, b, c, d)$, $\times(a, b, c, d)$, and $\circlearrowleft(a, b, c, d)$. The semantics of $\circlearrowleft(a, b, c, d)$ is the same as $\circlearrowleft(a, \times(b, c, d))$. For the other operators, it is simply the repeated application, i.e., $\rightarrow(a, b, c, d) = \rightarrow(a, \rightarrow(b, \rightarrow(c, d)))$, $\wedge(a, b, c, d) = \wedge(a, \wedge(b, \wedge(c, d)))$, and $\times(a, b, c, d) = \times(a, \times(b, \times(c, d)))$. The key thing to see is that the operators may compose complete subprocesses corresponding to subtrees.

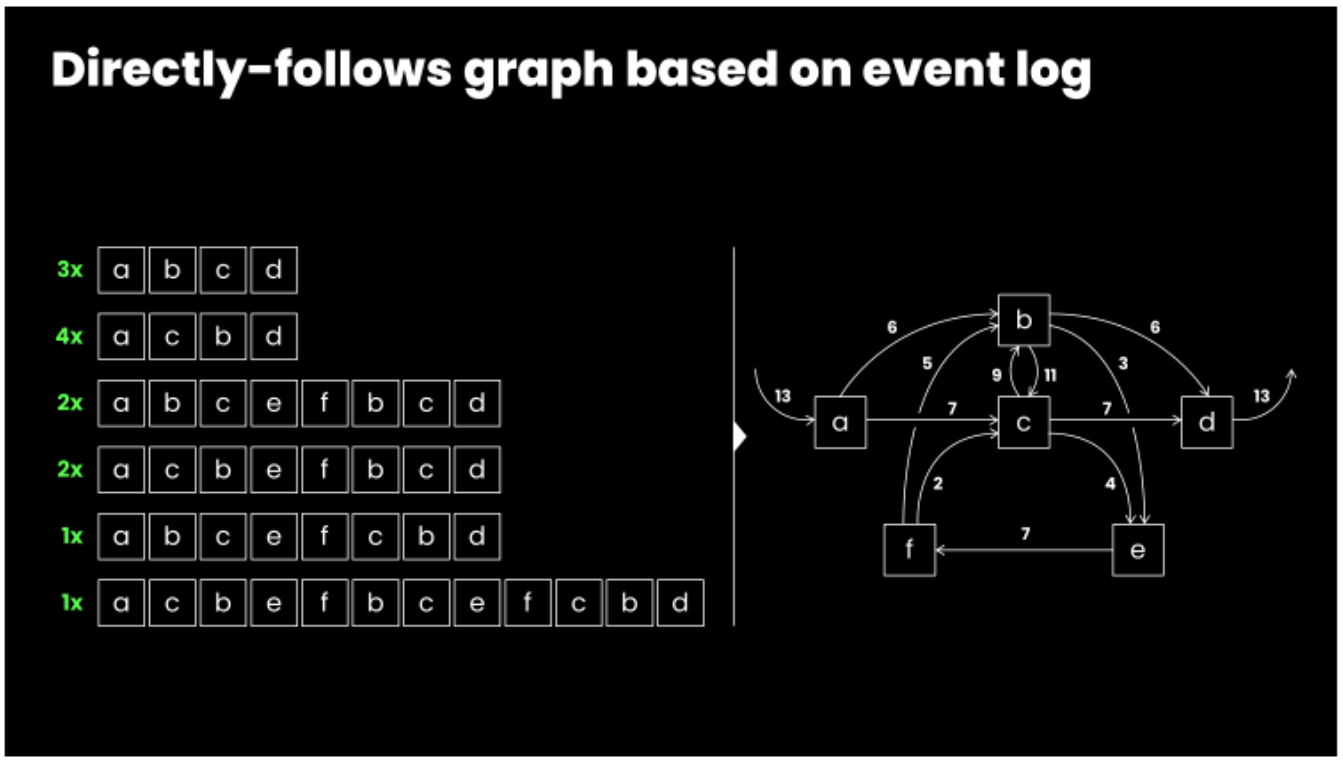
Some more examples:

- $\wedge(\rightarrow(a, b), c, \rightarrow(d, e))$ corresponds to the process tree allowing for traces such as $\langle a, b, c, d, e \rangle$, $\langle a, c, b, d, e \rangle$, $\langle c, a, b, d, e \rangle$, $\langle d, e, c, a, b \rangle$, $\langle d, c, e, a, b \rangle$, $\langle a, c, d, e, b \rangle$, $\langle d, a, e, b, c \rangle$, etc.
- $\times(\rightarrow(a, b), c, \rightarrow(d, e))$ corresponds to the process tree allowing for the traces $\langle ab \rangle$, $\langle c \rangle$, and $\langle de \rangle$.
- $\circlearrowleft(\rightarrow(a, b, c), \rightarrow(d, e))$ corresponds to the process tree, allowing for traces such as $\langle a, b, c \rangle$, $\langle a, b, c, d, e, a, b, c \rangle$, $\langle a, b, c, d, e, a, b, c, d, e, a, b, c \rangle$, etc.
- $\rightarrow(\circlearrowleft(a, b), c, \times(d, e))$ corresponds to the process tree allowing for traces such as $\langle a, c, d \rangle$, $\langle a, c, e \rangle$, $\langle a, b, a, c, d \rangle$, $\langle a, b, a, c, e \rangle$, $\langle a, b, a, b, a, c, d \rangle$, $\langle a, b, a, b, a, c, e \rangle$, $\langle a, b, a, b, a, b, a, b, a, c, d \rangle$, etc.

Process trees can be trivially converted to Petri nets and BPMN models. Therefore, they are mostly used as an *internal* representation and visualized as a BPMN model. However, to understand top-down approach discovery approaches such as inductive mining, it is important to understand the notation and why it is useful.

▼ Inductive Mining

The Alpha algorithm is an example of a bottom-up discovery approach that tries to add places to the Petri net to locally constrain behavior. Top-down discovery approaches try to recursively decompose the event log into smaller sublogs until these sublogs get trivial (e.g., refer to just one activity). The *inductive mining algorithm* is an example of a top-down approach. It is actually a *family of approaches* with many possible variants. Here, we just consider the basic principle of inductive mining. For each sublog, a DFG is created, and the structure is analyzed to see how the process and sublog can be decomposed further. We illustrate this using an example:

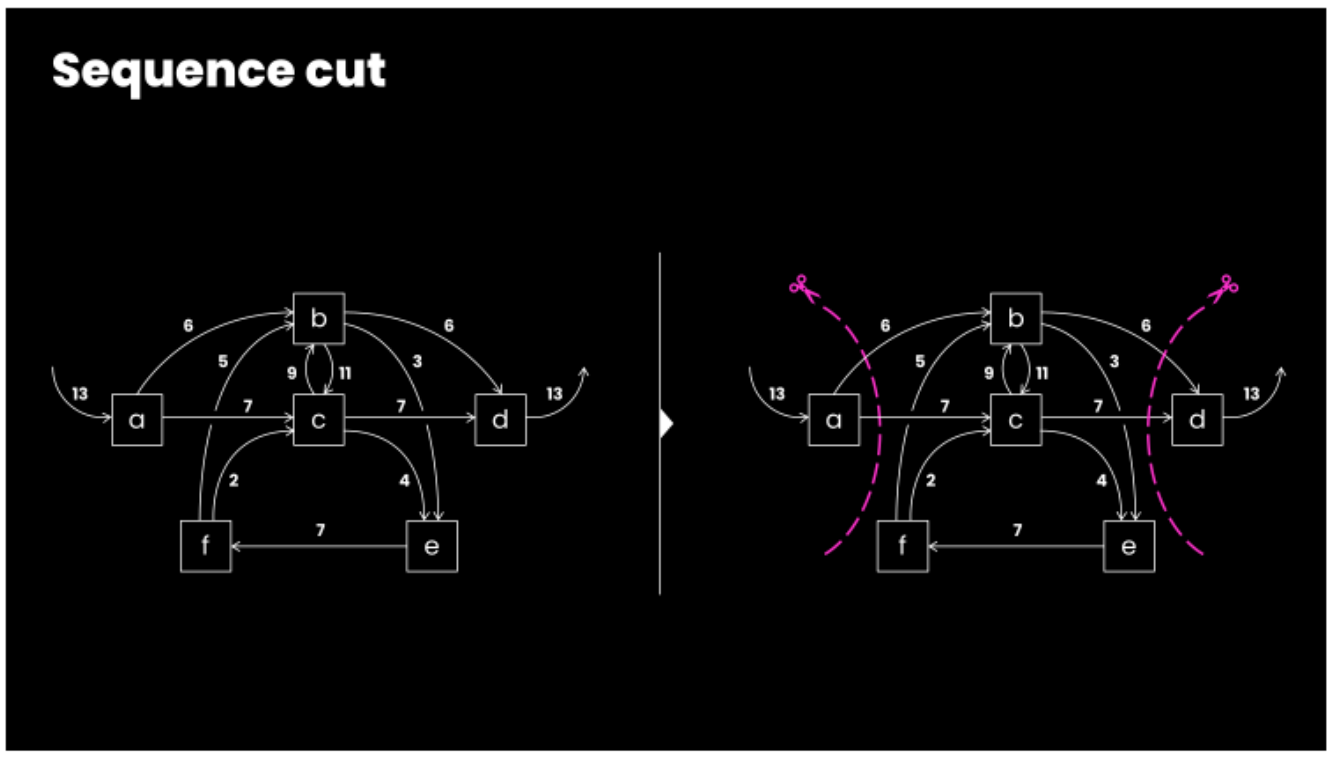


Based on the event log on the left (again abstracted into a multiset of traces), we create a DFG in the standard way. (Note that the DFG visualization is slightly different, because the start and end nodes are not depicted explicitly) . As long as the DFG has multiple activities, we try to find a pattern in the DFG that matches one of the process tree operators: $\rightarrow$ (*sequential composition*), $\times$ (*exclusive choice*), $\wedge$ (*parallel composition*), and $\circlearrowleft$ (*redo loop*). Each operator corresponds to a so-called *cut* of the tree. Hence there are four types of cuts:

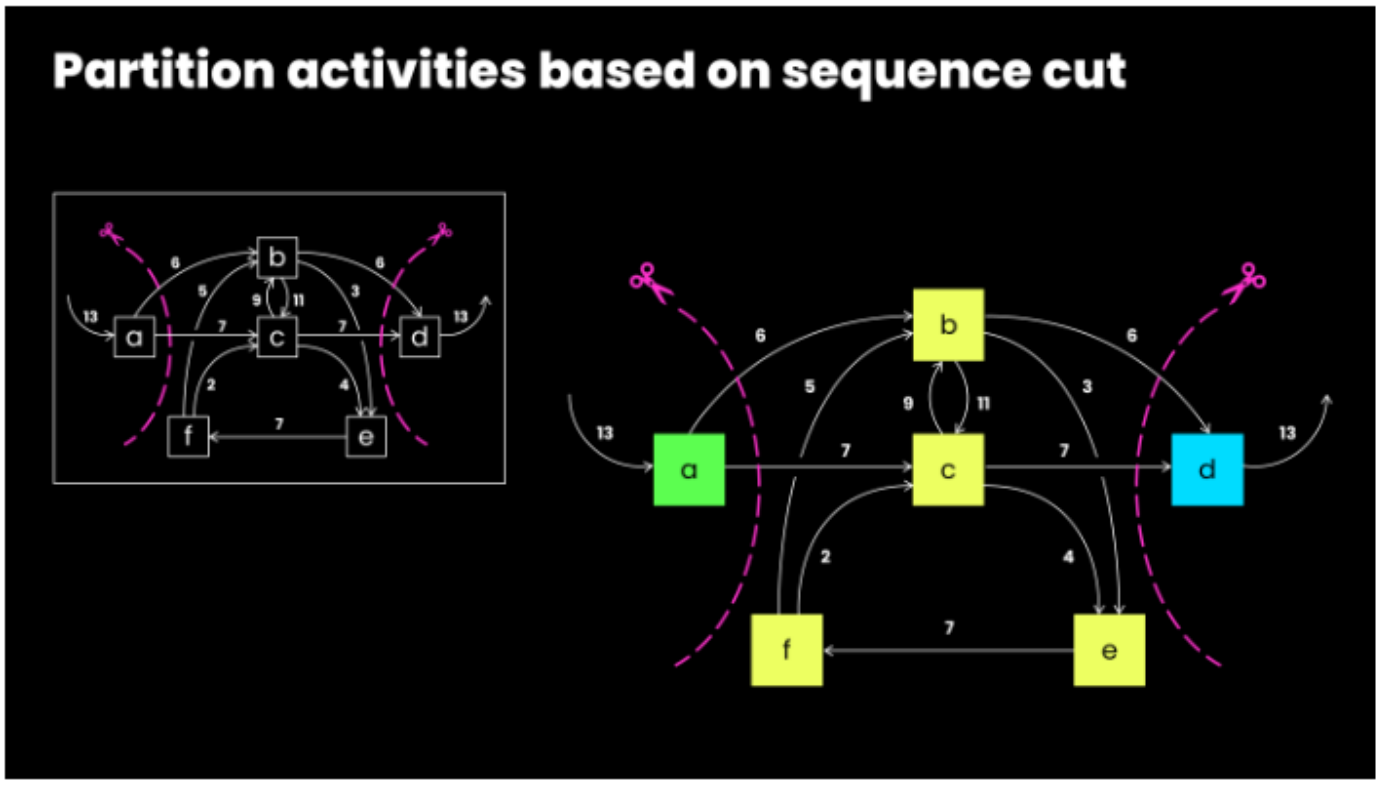
- *Exclusive-choice cut*: The process can be split into two or more parts that are in an exclusive choice relation, meaning that activities in different parts never follow each other.
- *Sequence cut*: The process can be split into two or more parts that are executed in sequence.
- *Parallel cut*: The process can be split into two or more parts that are in parallel relation, meaning that activities in different parts can follow each other because they are fully independent.
- *Redo-loop cut*: The process can be split into a "do part" and a "redo part".

The cuts are investigated in the order indicated. If a sublog refers to just one activity, it is trivial to pick the corresponding construct. If all traces in the sublog are of the form a (i.e., activity a is executed once for every trace), then the subtree is simply the activity node a. If a occurs at most once (i.e., activity a is sometimes executed once and sometimes not at all), then the subtree $\times\left(a, \tau\right)$ is produced. If a occurs at least once, then the subtree is $\circlearrowleft\left(a, \tau\right)$. If a occurs zero or more times, then the subtree is $\circlearrowleft\left(\tau, a\right)$. If the log still has multiple activities and no cut is possible, there are so-called fall-throughs that ensure that all behavior in the log is still possible. This only occurs when the behavior is not expressible as a process tree. In any case, the resulting process tree is *always* able to generate *all* behavior in the event log.

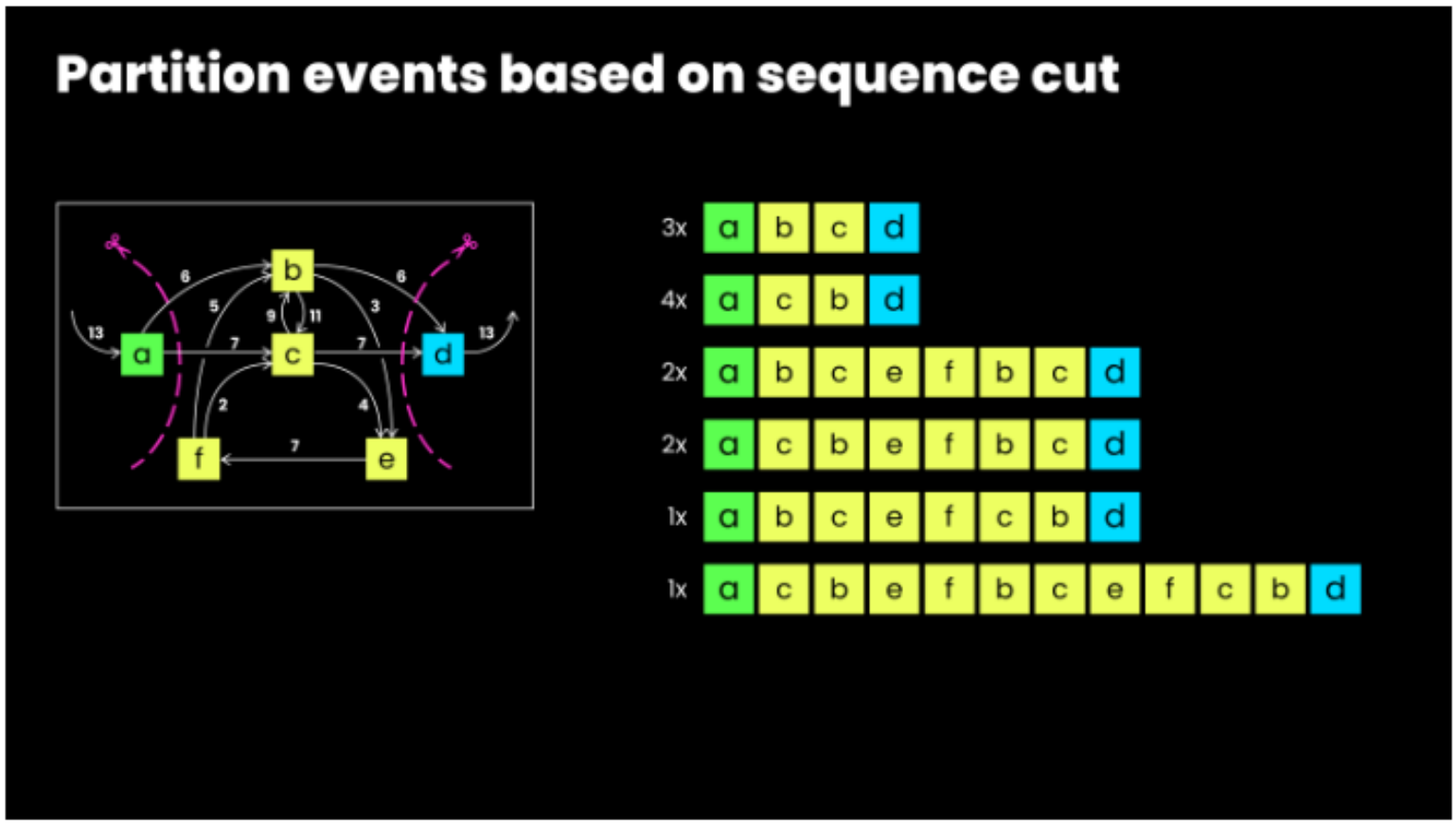
There is a sequence cut when the DFG can be split into sequential parts where only "forward connections" are possible. This is the case in our running example:



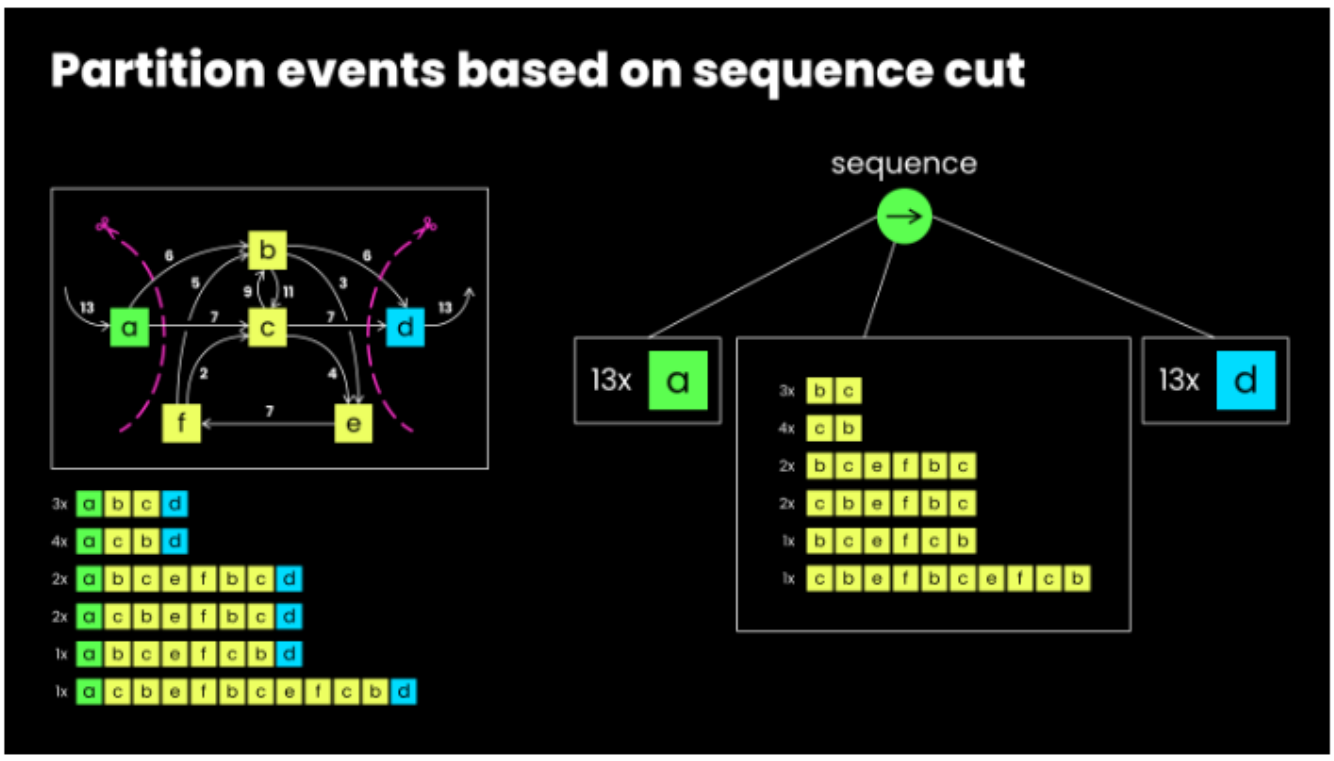
This particular *sequence cut* splits the set of activities into three disjoint subsets such that arcs only go from left to right. Take $X = \{a\}$, $Y = \{b, c, e, d\}$, and $Z = \{d\}$ as the three sets of activities. Activities in X can be followed by all activities in Y, but not vice versa. Activities in Y can be followed by all activities in Z, but not vice versa. Activities in X can be followed by all activities in Z, but not vice versa.



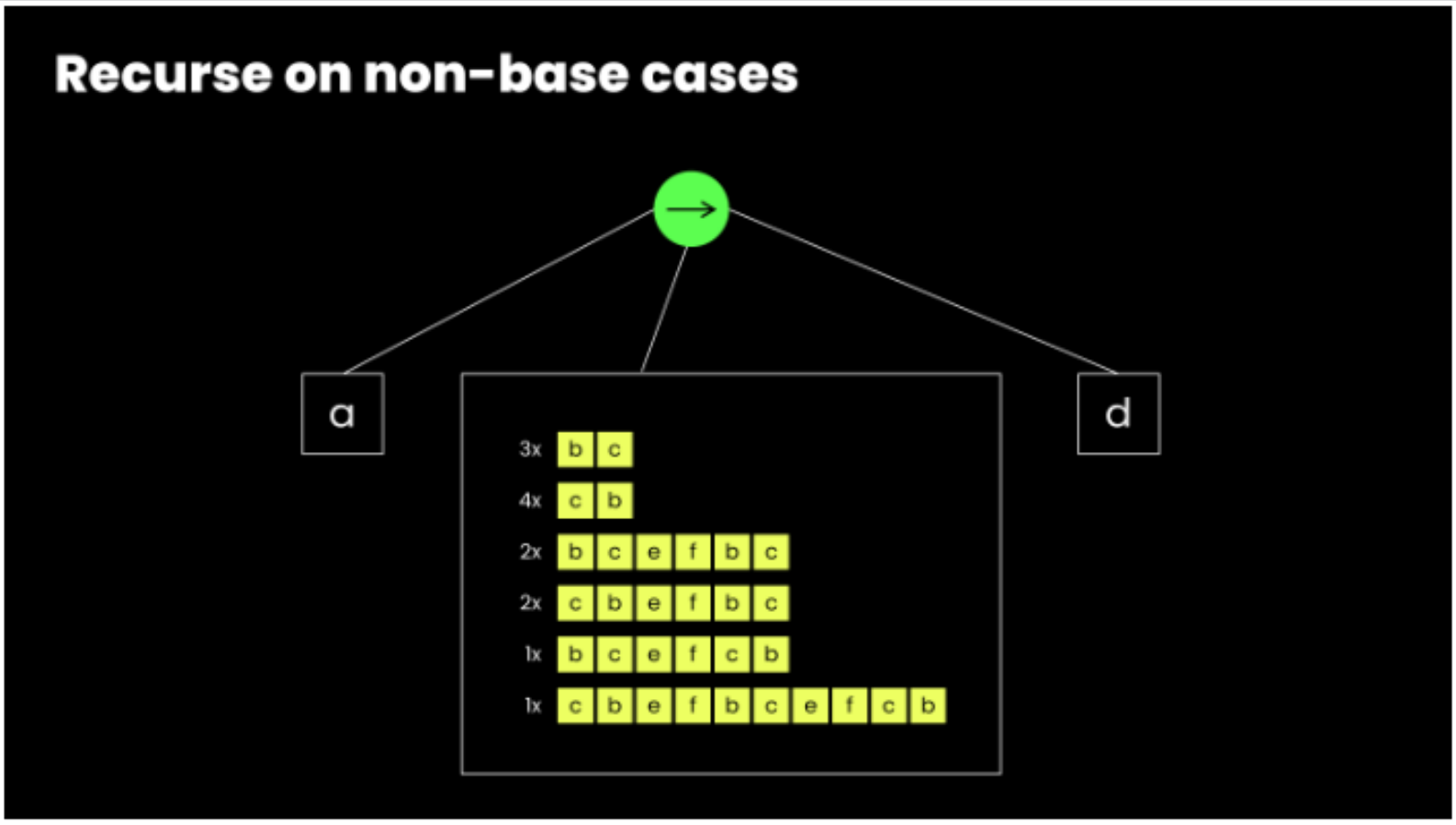
Based on $X = \{a\}$, $Y = \{b, c, e, d\}$, and $Z = \{d\}$, we partition the event log.



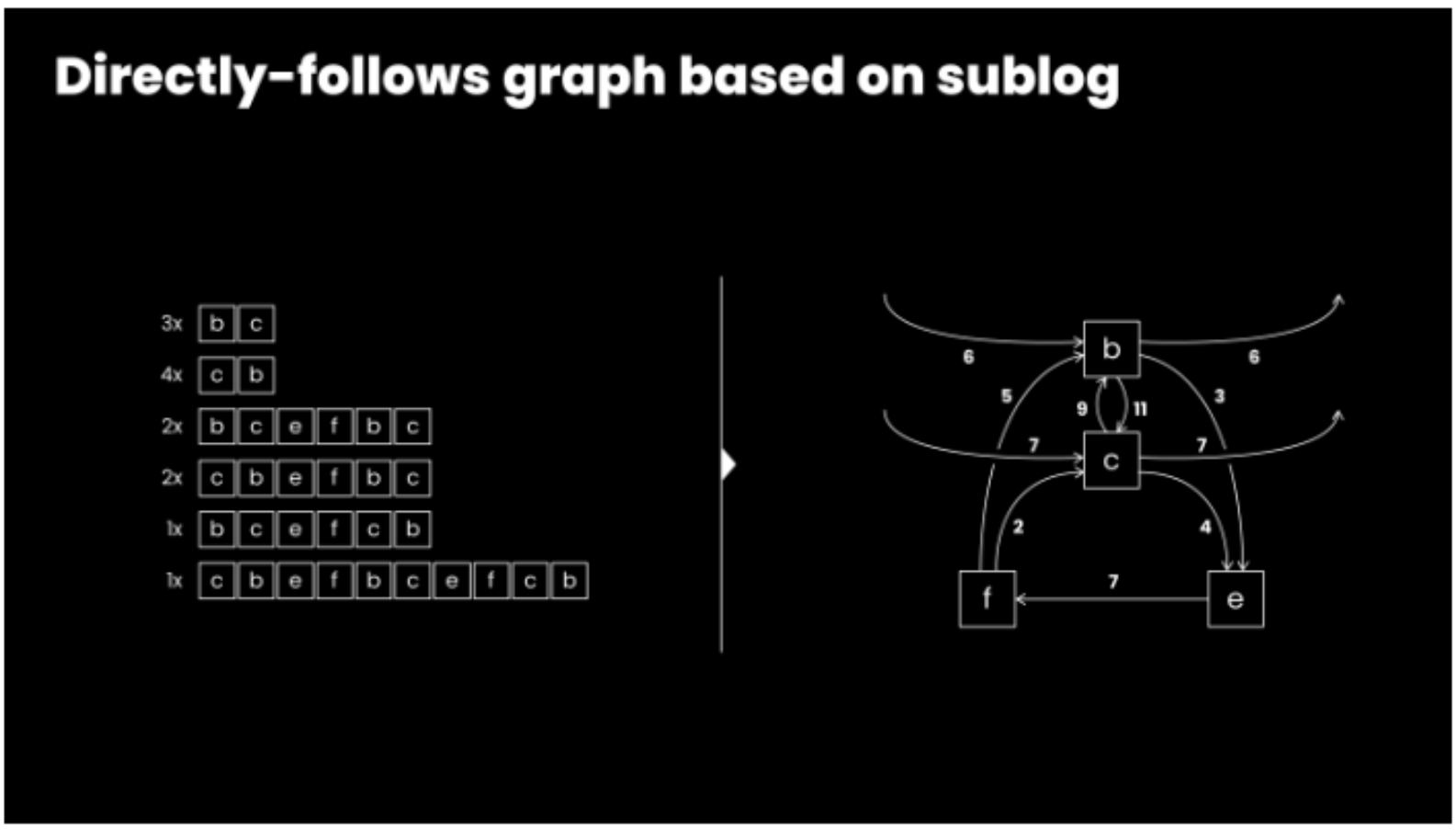
This results in three new event logs, called *sublogs*.



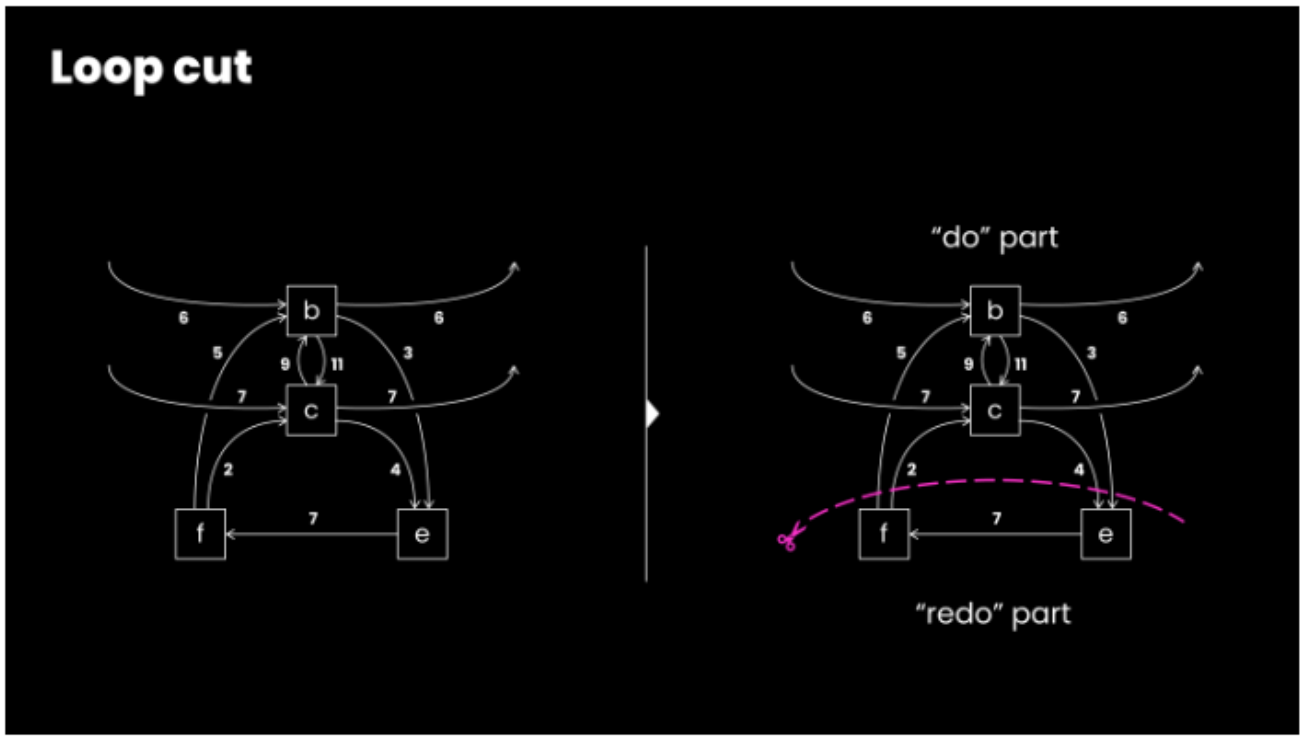
The sublog corresponding to $X = \{a\}$ is trivial because a always occurs once. The sublog corresponding to $Z = \{d\}$ is trivial because d always occurs once. The event log corresponding to $Y = \{b, c, e, d\}$ is not trivial and needs to be investigated further.



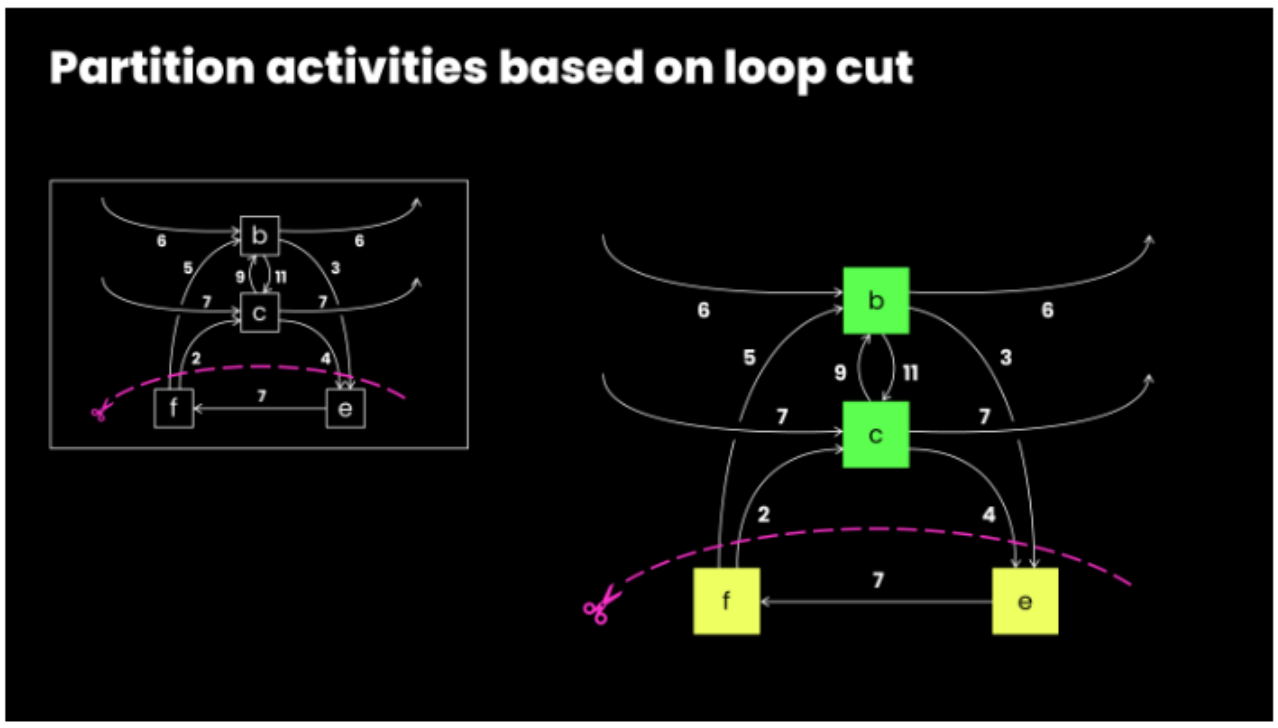
For the remaining sublog based on $Y = \{b, c, e, d\}$, we create another DFG.



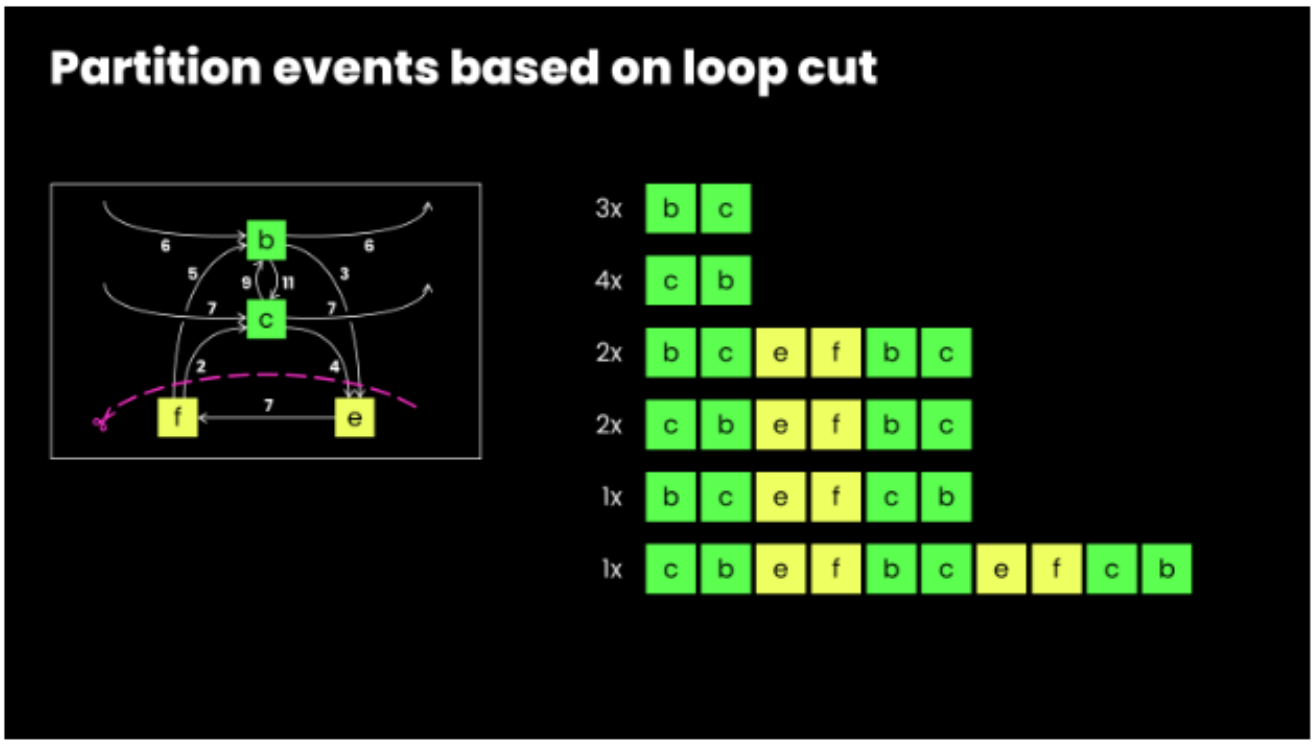
We cannot apply an exclusive-choice cut, we cannot apply a sequence cut, and we cannot apply a parallel cut. However, we can apply a *redo-loop cut,* splitting the activities into two sets: $X = \{b, c\}$ and $Y = \{e, f\}$. The following requirements are satisfied. All start and end activities should be in X (the "do part") and none of the "redo parts" can have start or end activities. Moreover, the "redo part" Y is only connected through the "do part" X and cannot contain start or end activities. The start activities of the "redo part" are connected to the end activities in the "do part" and the end activities of the "redo part" are connected to the start activities in the "do part". Also all end activities of the "do part" are connected to all start activities of the "redo part" and all end activities of the "redo part" are connected to all start activities of the "do part". All these requirements are satisfied, hence, we apply the redo-loop cut depicted.



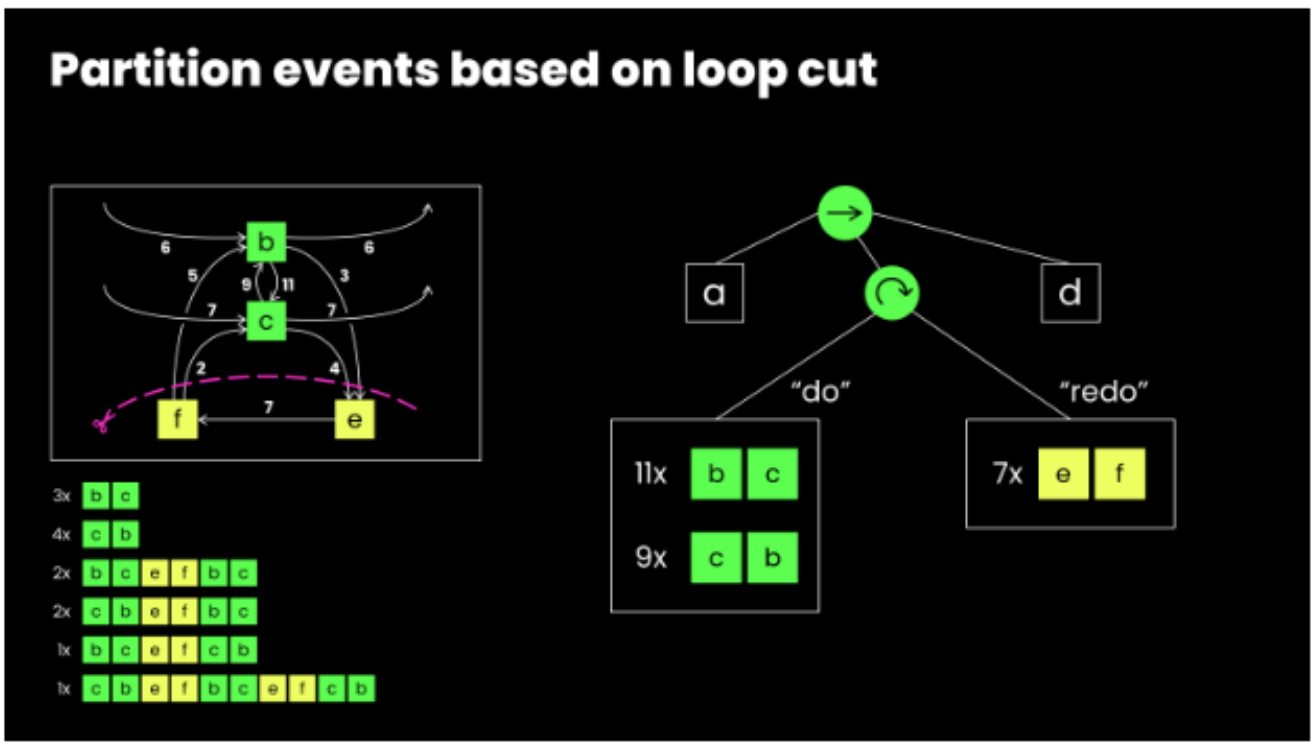
We partition the activities into $X = \{b, c\}$ and $Y = \{e, f\}$.



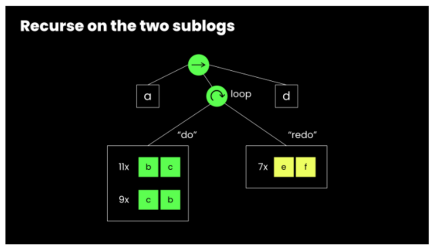
For each of the two sets of activities, we create new sublogs. However, due to the nature of the loop cut, one trace can be split into smaller traces. In other words, the way that we create sublogs depends on the operator. When using a sequence cut, the number of cases in each sublog is the same as in the original event log. When using a redo-loop cut, the number of cases in a sublog depends on how often the loop was taken.



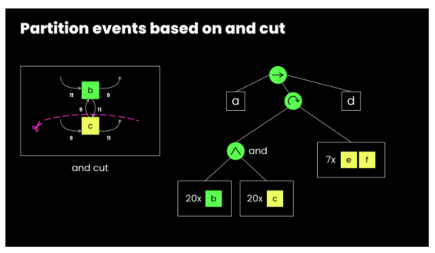
The sublog corresponding to $X = \{b, c\}$ has 20 traces, and the sublog corresponding to $Y = \{e, f\}$ has seven traces.



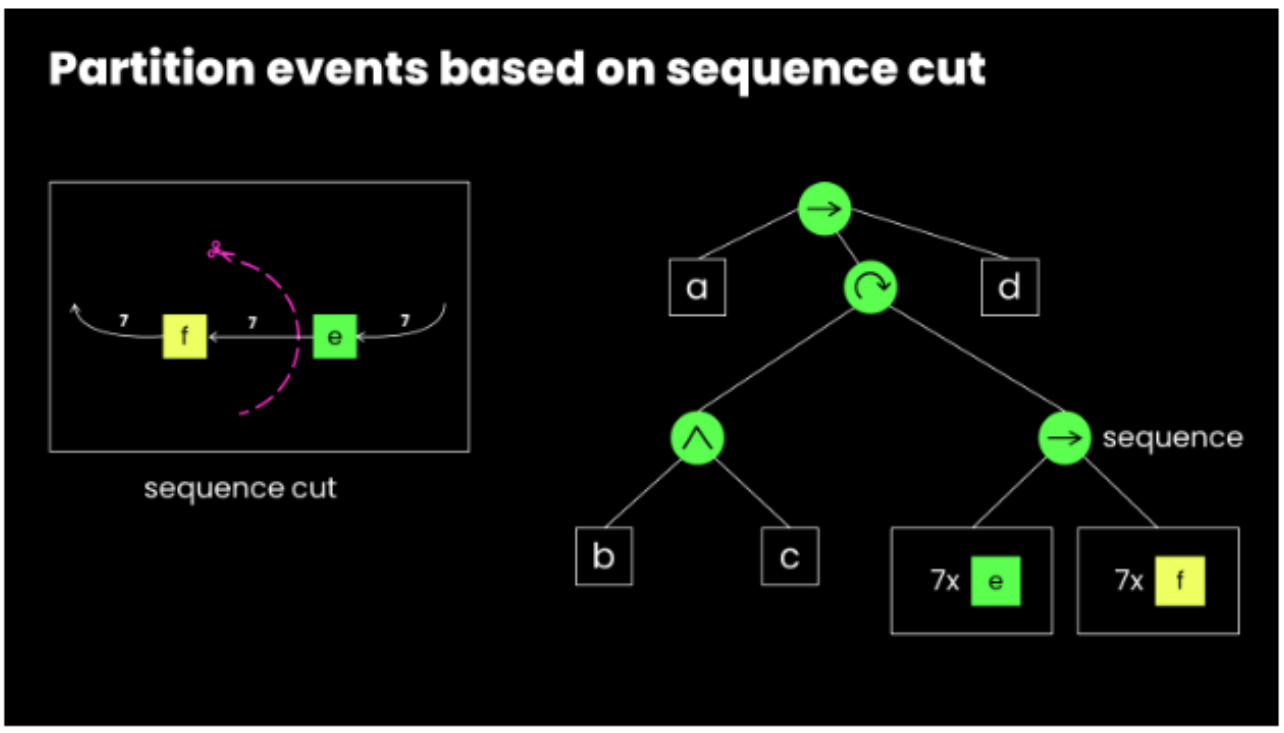
Both sublogs contain multiple activities and need to be processed further.



The DFG based on the sublog consisting of activities b and c shows that we can apply a parallel cut. There is a parallel cut when the DFG can be split into concurrent parts where any activity in one part can be followed by any activity in another part.

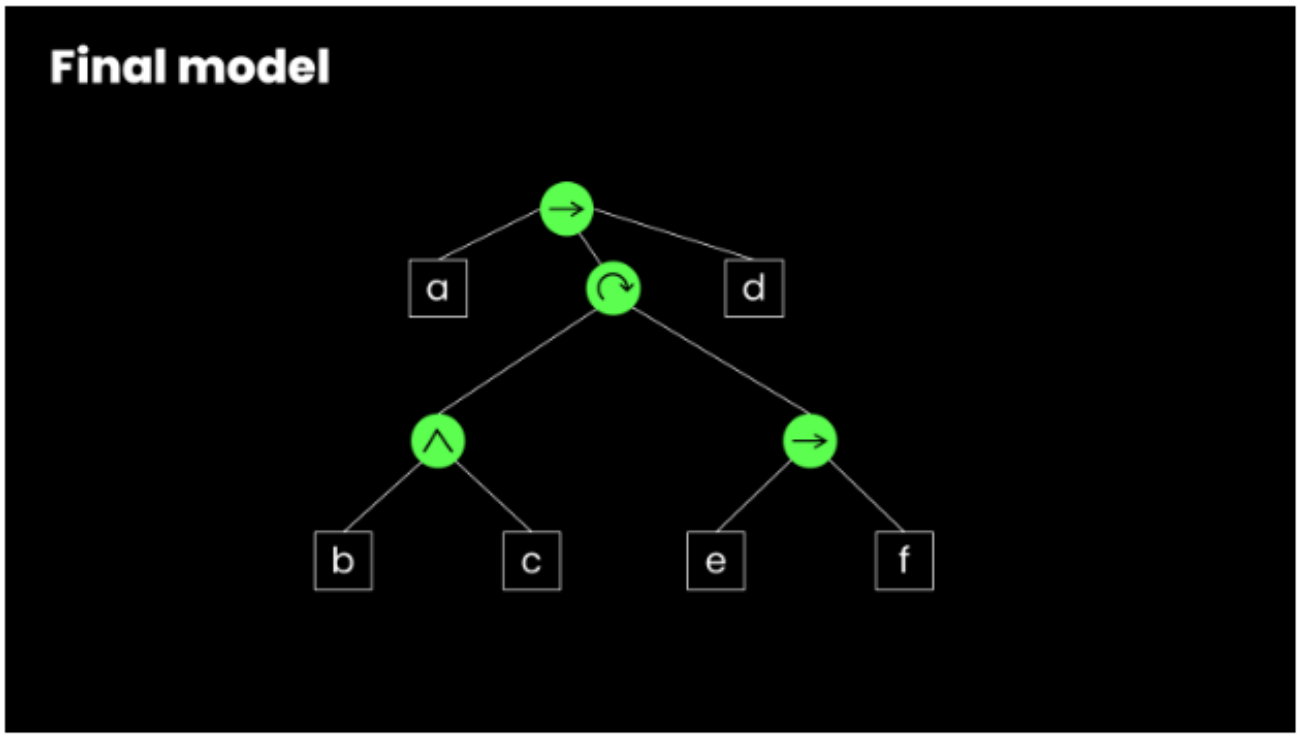


After creating two new sublogs for both b and c, we see that these are trivial. In each sublog, the respective activity is executed only once. The DFG for the sublog based on activities e and f, reveals another sequence cut.



After creating two new sublogs for both e and f, we see that these are also trivial.

Hence, we obtain the following process tree: $\rightarrow (a, \circlearrowleft (\wedge (b, c), \rightarrow (e, f)), d)$.



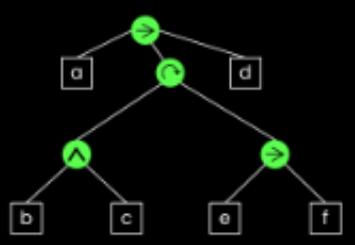
To create the model, we could not apply an exclusive-choice cut. However, this is the simplest cut to apply. If the DFG can be split into disconnected parts, then the activities in the respective parts never follow each other, clearly signaling an exclusive choice.

Using the set notation for activities in the parts of the tree not fully explored (highlighted in red), we can summarize this particular run of the inductive mining algorithm as follows:

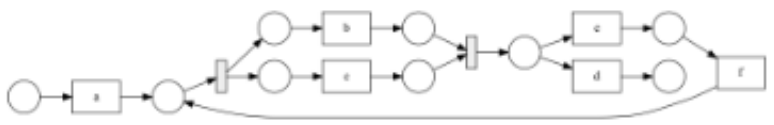
1. $\{a, b, c, d, e, f\}$: detect sequence cut,
2. $\rightarrow (\{a\}, \{b, c, e, f\}, \{d\})$: replace the two trivial sublogs by their respective activity,
3. $\rightarrow (a, \{b, c, e, f\}, d)$: detect redo-loop cut,
4. $\rightarrow (a, \circlearrowleft (\{b, c\}, \{e, f\}), d)$: detect parallel cut,
5. $\rightarrow (a, \circlearrowleft (\wedge (\{b\}, \{c\}), \{e, f\}), d)$: replace the two trivial sublogs by their respective activity,
6. $\rightarrow (a, \circlearrowleft (\wedge (b, c), \{e, f\}), d)$: detect sequence cut,
7. $\rightarrow (a, \circlearrowleft (\wedge (b, c), \rightarrow (\{e\}, \{f\})), d)$: replace the two trivial sublogs by their respective activity,
8. $\rightarrow (a, \circlearrowleft (\wedge (b, c), \rightarrow (e, f)), d)$: final process model.

The process tree can be converted into a more standard notation.

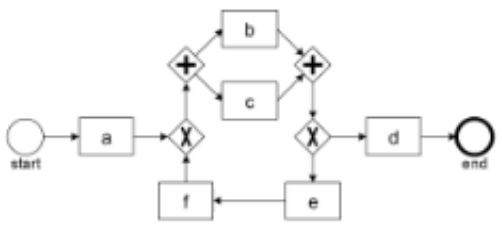
Different visualizations of the same discovered model



Petri net



BPMN



Any discovered process tree can be mapped onto a BPMN model or Petri net with exactly the same behavior.

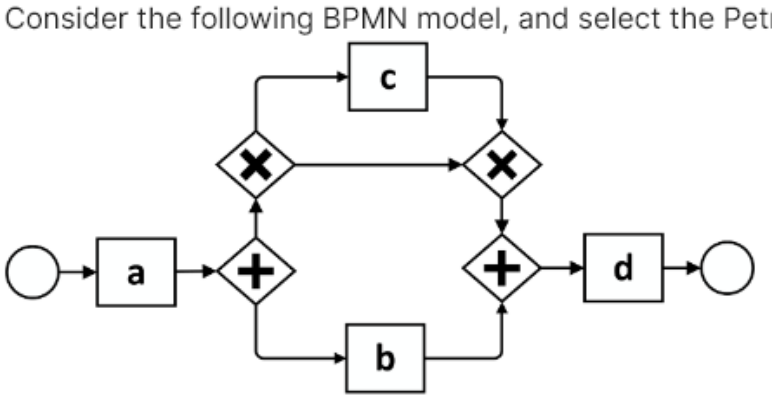
As the example shows, inductive mining follows a *divide-and-conquer approach* using process trees that are sound by construction. Each subtree of the final process tree corresponds to a particular sublog. The approach ensures that the trace in the original event log can be generated by the resulting process model. This also applies to all subtrees and sublogs.

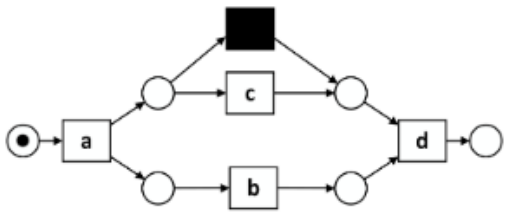
▼ **Exercice**

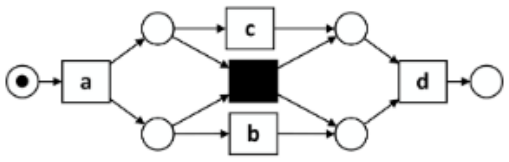
Question 1

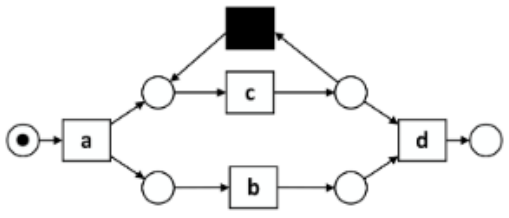
1/1 point (ungraded)

Consider the following BPMN model, and select the Petri net that has the same behavior.





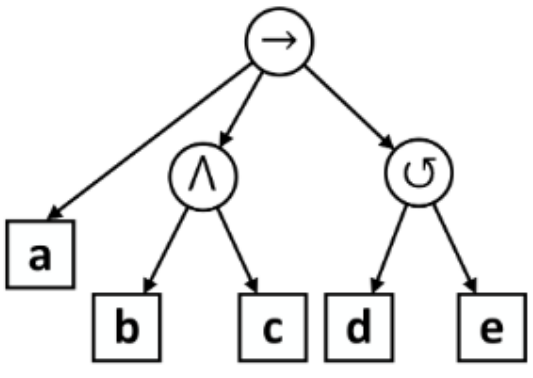




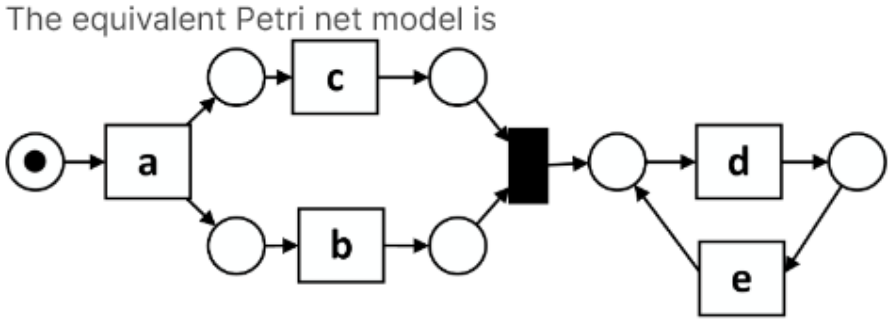
Question 2

1/1 point (ungraded)

Consider the following process tree. Which of the following statements are correct?



- [ ] The trace $\langle a, b, c, d, e \rangle$ is allowed by this model.
- [x] Activities *b* and *c* can occur in any order.
- [x] Activity *d* can occur several times in one trace.
- [ ] A complete trace always starts with activity *a* and ends with activity *e* .
- [x] The equivalent Petri net model is

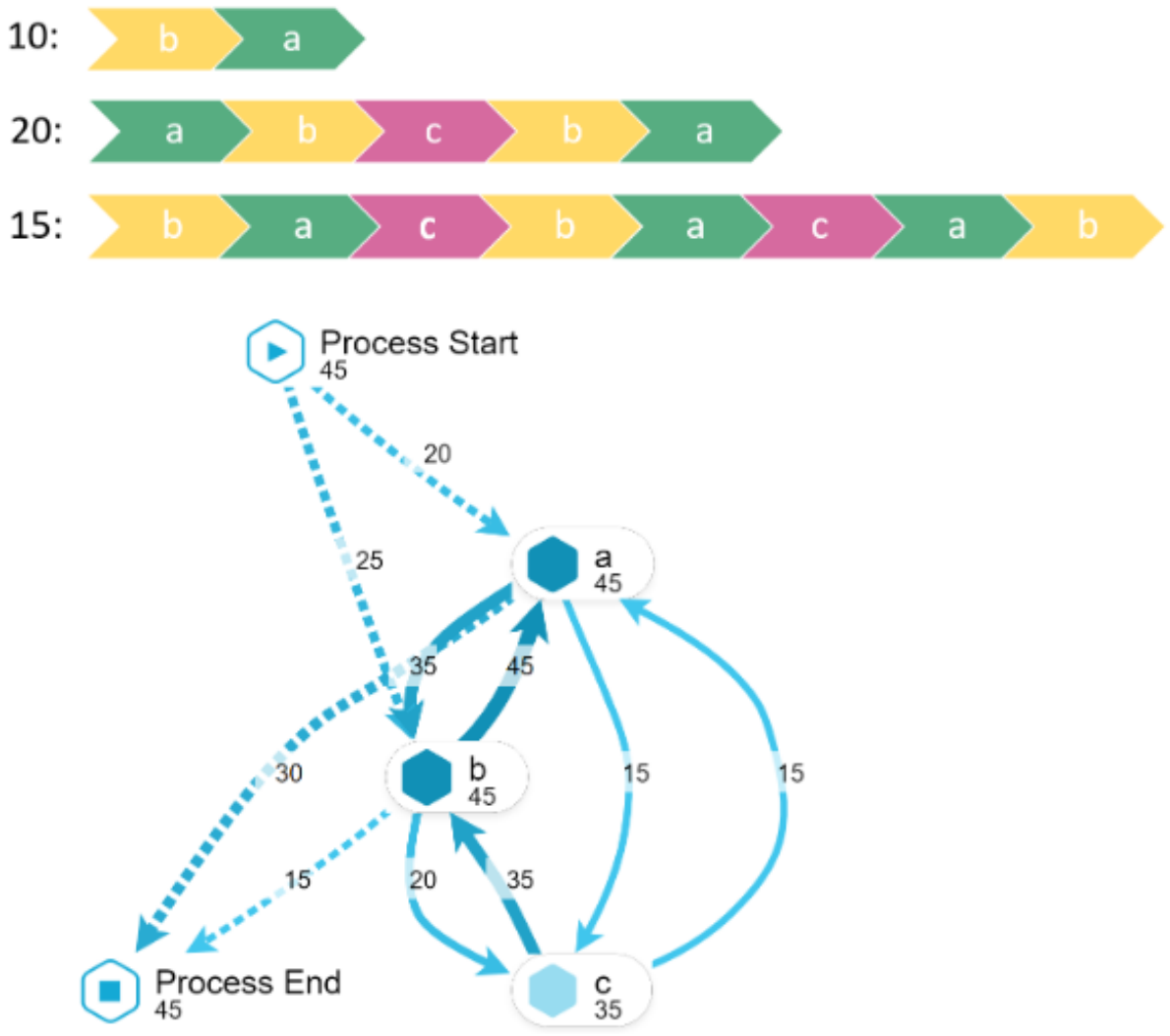


✔

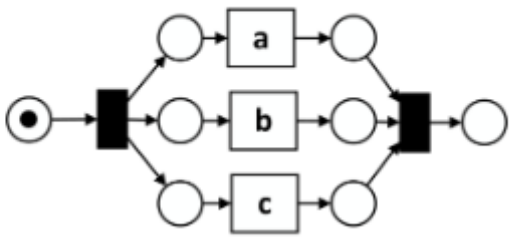
Question 3

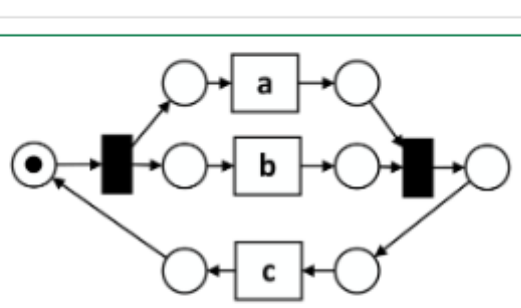
1/1 point (ungraded)

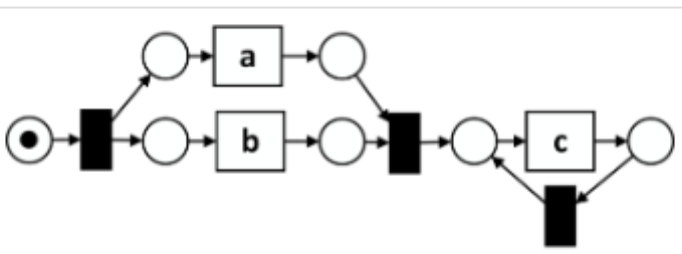
Consider the following event log and the corresponding directly-follows graph.



Which of the following models is discovered by the inductive miner algorithm?







> 💡 Process discovery, a key task in process mining, involves creating process models from event logs, often facing challenges like representing concurrency. Higher-level notations such as Petri nets, process trees, and BPMN help overcome these limitations. Inductive process discovery techniques, particularly those based on process trees, produce sound models with guaranteed recall but may underfit in some cases. Various other approaches like the Alpha algorithm, heuristic mining, and region-based methods offer different advantages and challenges in discovering process models.

▼ Celonis - Process Discovery

how to go from a raw data set to a Directly Follows Graph (DFG) representation in Celonis. DFGs are the most commonly used process visualizations in process discovery. They are very intuitive and add a story-telling aspect to the process visualization as they illustrate the process one step at a time. DFGs have the downside of getting complex very quickly (as compared to BPMN models or Petri nets). In this lecture, we will also see how activity, arc-based, and variant-based filtering can be used in a simple order management example to break down complexity.

If you want to test out the Celonis software yourself, you may also access the Celonis Training Cloud to upload the classroom data. Instructions on how to obtain software access, as well as the demo data set used in this lecture, can be found below:

- **How to get a Celonis license & upload data :** https://courses.edx.org/assets/courseware/v1/b5281a907af1c274d349873f7986b039/asset-v1:RWTHx+PM+1T2023+type@asset+block/How_to_get_a_Celonis_license___upload_data.pptx.pdf
- **Data :** https://courses.edx.org/assets/courseware/v1/b5281a907af1c274d349873f7986b039/asset-v1:RWTHx+PM+1T2023+type@asset+block/How_to_get_a_Celonis_license___upload_data.pptx.pdf
- https://www.youtube.com/watch?v=rbOGUtlpb-8

We could already see that DFGs are very handy in practical use. However they have a number of shortcomings when it comes to parallelisms or concurrency. Therefore Petri nets and BPMN models can be a useful, yet less intuitive, alternative. will now look at two other more complicated variations of an Order Management process including skips and concurrency. We will investigate what a DFG for these examples looks like and what it can capture in comparison to a BPMN model.

You can find the corresponding data sets for download below together with a tutorial on how to upload data in the new Celonis user interface:

- **Data :** https://courses.edx.org/assets/courseware/v1/a9e90947dbfceebce2feb64edd6af070/asset-v1:RWTHx+PM+1T2024d+type@asset+block/concurrency-matters.xlsx
- https://www.youtube.com/watch?v=CZVhRTDHS5M