Chapter 2
Control-Flow Perspective

The control-flow perspective is a type of analysis that focuses on the discovery of the sequence of activities in a business process. The idea is that by analyzing how tasks follow each other in the event log, it should be possible to come up with a model that describes the overall behavior of the process.

There are several algorithms to discover the sequential behavior of a process, with notable examples being the \( \alpha \)-algorithm [19], the heuristics miner [25], the genetic miner [9], and the fuzzy miner [4]. These algorithms employ different approaches to arrive at essentially the same result, which is a model that depicts the transitions between tasks.

The simplest way to do this is as follows: every time task \( a \) is followed by task \( b \), we count that transition. We do this for all pairs of consecutive tasks within the same case id. (Transitions between tasks in different case ids do not count.) Doing this across the whole event log will provide a count of how many times each transition has occurred. Then, it is possible to combine these transitions in order to generate an output graph that captures the sequential behavior of the process.

This idea is the essence of many control-flow algorithms. Rather than looking at a range of different algorithms and their specific details, here we will focus on this single fundamental idea. Armed with a good understanding of how this idea can be implemented, the interested reader will find it easier to get acquainted with more advanced algorithms in the field of process mining.

2.1 The Transition Matrix

As stated above, we will be looking at a simple version of a control-flow algorithm. This algorithm will work with case ids and tasks. The algorithm will be described mainly in abstract terms, meaning that we will refer to examples of tasks such as
a and b without implying a connection to the purchase process from the previous chapter. In the present context, a, b, c, etc., are just some arbitrary tasks.

Let us think for a moment on how we should store the information about the transitions between these tasks. Before analyzing the event log, we do not know which transitions have actually occurred, so we can only assume that any transition between those activities is possible. If we have \( N \) activities, then there are \( N^2 \) possible transitions between these activities. For example, with three activities \( \{a, b, c\} \) there are nine possible transitions, namely:

\[
\{a \to a, a \to b, a \to c, b \to a, b \to b, b \to c, c \to a, c \to b, c \to c\}
\]

To store the count of how many times each transition has occurred, it becomes more convenient to represent these transitions in matrix form:

\[
\begin{array}{ccc}
   & a & b & c \\
   a & & & \\
b & & & \\
c & & & \\
\end{array}
\]

The nine cells in this matrix can be used to store the count of each transition. This is called the transition matrix. The goal of the control-flow algorithm is to go through the event log and to fill in this transition matrix with a count in each cell.

In particular, the transition matrix should be read in the following way: if in row \( i \) we find activity \( a \) and in column \( j \) we find activity \( b \), then the cell \((i, j)\) contains the number of times that transition \( a \to b \) has been observed.

To formally describe the algorithm, it becomes more convenient to use the notation \( a_i \) for the activity in row \( i \) and \( a_j \) for the activity in column \( j \). The activities are then \( \{a_1, a_2, a_3, \ldots\} \) and the transition matrix has the following form:

\[
\begin{array}{cccc}
   & a_1 & a_2 & a_3 & \ldots \\
a_1 & & & & \\
a_2 & & & & \\
a_3 & & & & \\
\ldots & & & & \\
\end{array}
\]

### 2.2 The Control-Flow Algorithm

Let \( T \) be the set of distinct tasks recorded in an event log, and let \( |T| \) be the size of that set. For example, if \( T = \{a, b, c, d, e, f, g, h\} \) then \( |T| = 8 \).

In mathematical terms, the transition matrix is a function \( f : T \times T \to \mathbb{N}_0 \) which gives the number of times that each possible transition between a pair of activities
2.3 Implementation in Python

Algorithm 1 Control-flow algorithm

1: Let $F$ be a square matrix of size $|T|^2$
2: Initialize $F_{ij} \leftarrow 0$ for every position $(i,j)$
3: for each case id in the event log do
4:    for each consecutive task transition $a_i \rightarrow a_j$ in that case id do
5:        $F_{ij} \leftarrow F_{ij} + 1$
6:    end for
7: end for

in $T$ has been observed. The objective of the control-flow algorithm is to find all the values for this function.

This can be done by initializing a transition matrix of size $|T|^2$ with zeros. As we go through the event log, every time a transition $a_i \rightarrow a_j$ is observed, we increment the value at position $(i,j)$ in the matrix. Algorithm 1 describes this procedure.

Since the values in matrix $F$ are obtained through a counting procedure, we will refer to those values as the transition counts.

2.3 Implementation in Python

There are several ways to implement the above algorithm in Python. The main decision is which data structure should be used to store the transition matrix. The matrix is bi-dimensional, so it makes sense to use a data structure that can be indexed twice (for rows and columns). With the built-in data structures available in Python, the natural choices are: a list of lists, or a dictionary of dictionaries.

Using a list of lists would involve indexing by position and it would require having a value stored at each position, even if the corresponding transition never occurs in the event log (those positions would remain with zero). This is what could be called a dense representation of the transition matrix.

On the other hand, using a dictionary of dictionaries allows us to index by task name and we have to introduce only the keys that correspond to the transitions that actually occur in the event log. This is what could be called a sparse representation of the transition matrix.

Given that the size of a full transition matrix is $|T|^2$ but probably only a subset of all possible transitions will be observed, it makes sense to use a sparse representation which avoids having to store a relatively large amount of zeros. With these issues in mind, here we will show how to implement the transition matrix as a dictionary of dictionaries.

Also, we assume that the event log has been read into a dictionary as in Listing 8 on page 11. After that, we could write the code shown in Listing 11.

At the beginning of this code, the matrix $F$ is initialized as a dictionary. Then the script iterates through each case id in the log, and also through the list of events for that case id. Here, $a_i$ and $a_j$ are two variables that hold a pair of consecutive tasks.
Listing 11  Implementing the control-flow algorithm in Python

```python
F = dict()
for caseid in log:
    for i in range(0, len(log[caseid])-1):
        ai = log[caseid][i][0]
        aj = log[caseid][i+1][0]
        if ai not in F:
            F[ai] = dict()
        if aj not in F[ai]:
            F[ai][aj] = 0
        F[ai][aj] += 1
for ai in sorted(F.keys()):
    for aj in sorted(F[ai].keys()):
        print ai, '->', aj, ':', F[ai][aj]
```

Listing 12  Output of the previous script

```plaintext
a -> b: 3
b -> c: 1
b -> d: 2
d -> e: 1
d -> g: 1
e -> f: 2
f -> g: 1
f -> h: 1
g -> e: 1
g -> h: 1
```

If ai is not present in the matrix (line 6), then that row is initialized as a dictionary. If aj is not present in that row (line 8), then that position is initialized with zero. Immediately after this, and regardless of any initialization that might have been done before, the value at that position is incremented (line 10).

The rest of the script shows how to iterate through the matrix and print its contents. For every row ai and column aj (with both being sorted in alphabetical order), the script prints the value at position F[ai][aj]. The output of this code for the event log in Listing 1 on page 7 is shown in Listing 12.

2.4 Introducing Graphviz

Graphviz\(^1\) is a wonderful piece of software. It can save enormous amounts of work when creating graphs, since it takes care of the graph layout automatically. Graphviz is very often used to visualize the results of process mining techniques.

To provide an idea of what Graphviz does, we will start with a simple example in Listing 13. This is a text-based definition of a directed graph. Graphviz supports both directed and undirected graphs, and this is a matter of specifying digraph or graph at the beginning of the definition.

---

\(^1\)http://www.graphviz.org/.
2.4 Introducing Graphviz

Listing 13  Definition of a directed graph in Graphviz’s DOT language

```dot
digraph G {
  rankdir=LR;
  node [shape=box];
  a -> b [label="3"];
  b -> c [label="1"];
  b -> d [label="2"];
  d -> e [label="1"];
  d -> g [label="1"];
  e -> f [label="2"];
  f -> g [label="1"];
  f -> h [label="1"];
  g -> e [label="1"];
  g -> h [label="1"];
  }
```

The graph has a name (in this case, \texttt{G}) and its structure comprises a series of statements enclosed in curly braces (\{\ldots\}) and separated by semicolons (;). Each statement adds a piece of information to the graph definition.

The first statement \texttt{rankdir=LR} establishes the graph orientation from left to right (the default is \texttt{TB}, i.e. top to bottom). The second statement says something about the nodes in this graph. In particular, it says that the shape of nodes is a box (rectangle). Technically, \texttt{shape} is an attribute of node, and it can be specified individually for each node (in order to have nodes with different shapes, for example). However, here the shape is being specified globally for every node.

The remaining statements define the edges in the graph. It should be noted that in this example the nodes being are defined implicitly by the edges, i.e. the statement \texttt{a->b} defines an edge between nodes \texttt{a} and \texttt{b} and, implicitly, it also defines the nodes \texttt{a} and \texttt{b} since they have not been defined before.

Nodes and edges can also be defined separately. A common practice is to first define nodes and their attributes, and only then define the edges between those nodes. In the example of Listing 13, the edges and their attributes are being defined. In this simple example, only one edge attribute (\texttt{label}) is being used.

With the \texttt{label} attribute, we are attaching a label to each edge. A possible use for that label is to annotate the edges with the transition counts provided by the control-flow algorithm.

Generating a graph from the definition in Listing 13 is as simple as running a command such as: \texttt{dot -Tpng listing13.gv -o graph.png.2} In this command, \texttt{dot} is the Graphviz tool that calculates the graph layout and produces an image. Several image formats are supported, including both raster graphics (e.g. PNG, JPEG) and vector graphics (e.g. SVG).

Figure 3 shows the output generated by Graphviz. Note how Graphviz has automatically decided on the positioning of each node and has also carefully rendered the edges and their labels without any crossings or overlaps.

\footnote{To be able to run this command, you may have to install Graphviz first. In Ubuntu, Graphviz can be installed with: \texttt{sudo apt-get install graphviz}.}
Finally, note how this graph depicts the behavior of the process shown in Fig. 1 on page 2. Naturally, this graph is not as expressive as a full-fledged process modeling language, but it certainly captures the run-time behavior of the process from the information recorded in the event log.

### 2.5 Using PyGraphviz

There are several ways in which one can use Python and Graphviz together. Python is a good language to implement process mining algorithms, and Graphviz is a great tool to visualize the results. The question now is how to plug these tools together in order to generate the graph from the results of the control-flow algorithm.

The simplest solution would be to modify the Python code in Listing 11 on page 18 to print the graph definition. After all, that Python script is already generating the output in Listing 12. With a few tweaks, it could as well generate the graph definition in Listing 13, which is not much different.

However, it can be a bit cumbersome to have complex graph definitions being generated with `print` instructions in Python. In addition, this would still require running `dot` manually in the command line in order to generate the graph.

A more elegant solution is to use a Python interface for Graphviz, such as pydot³ or PyGraphviz.⁴ Here, we use PyGraphviz which, at the time of this writing, has been in active development in recent years.⁵

Listing 14 shows how to build and generate the graph, assuming that the transition matrix has already been created by Listing 11 on page 18.

³[https://pypi.python.org/pypi/pydot/](https://pypi.python.org/pypi/pydot/).
⁴[https://pypi.python.org/pypi/pygraphviz/](https://pypi.python.org/pypi/pygraphviz/).
⁵In order to use PyGraphviz, you may have to install it first. In Ubuntu, it can be installed with:
sudo apt-get install python-pygraphviz.
The script starts by importing PyGraphviz and then creates a directed graph, as indicated by `directed=True`. The `strict` argument, if true, imposes certain restrictions, such as not allowing self-loops and multiple edges between the same pair of nodes. However, here we may have self-loops (i.e. transitions between the same activity), so we do not impose such restrictions.

In lines 5–6, the script sets the graph attribute `rankdir` and the node attribute `shape` in a similar way to what was done in Listing 13.

The most important part comes in lines 8–10 where the script iterates through the rows and columns in the transition matrix $F$ and adds an edge for each transition. The edge is labeled with the transition count stored in the matrix. Finally, in line 12 the script draws the graph by invoking the `dot` program, and saves it into an image file.

Behind the scenes, PyGraphviz generates a graph definition that is very similar to the one presented in Listing 13. The interested reader may want to try adding the instruction `print G.string()` to the code in Listing 14 to see the graph definition generated by PyGraphviz, and compare it to Listing 13.

### 2.6 Edge Thickness

It often happens in practice that the event log to be analyzed is quite large and the resulting graph has a lot of edges with different transition counts, some being relatively large and others being relatively small.

By labeling each edge with the corresponding transition count, as in Fig. 3, it is possible to identify the most frequent transitions, but it still requires us to have a look at every label in order to compare those transitions and to determine, for example, which transition is the most frequent one.
Consider what would happen if the event log had several thousand instances. Figure 4 shows how the output graph could look like.

Here, it becomes a bit difficult to compare the transitions and determine which one is the most frequent. Of course, if we have some prior knowledge about the process, we could expect that $a \rightarrow b$ is the most frequent transition, but it would still take us a moment to confirm that in Fig. 4.

Fortunately, there is a simple way to improve the graph in order to provide a better idea of the relative frequency of transitions at first glance. This can be done by adjusting the thickness of each edge according to the corresponding transition count, as shown in Fig. 5.

A quick look at Fig. 5 suggests that $a \rightarrow b$ is the most frequent transition based on the thickness of that edge when compared to others. We can look at the transition counts to confirm this, but in any case we need to compare the labels of only the thickest edges, without having to worry about the thinner ones.

More importantly, an attentive look at Fig. 5 leads to the conclusion that the most frequent path in this process is $a \rightarrow b \rightarrow d \rightarrow g \rightarrow e \rightarrow f \rightarrow h$. This is another type of insight that can be gained by analyzing the event log.

By looking at the thickness of edges, it also becomes apparent that the number of purchase requests that are not approved (i.e. the ones that follow through activity $c$) is noticeably smaller than the ones which do get approved (i.e. the ones that go through activity $d$). From the edge labels, one can estimate that about 25% of the purchase requests do not get approved.

Edge thickness is therefore an important feature that can provide a better perception of the results of process mining techniques.

In terms of implementation, edge thickness can be controlled with the `penwidth` attribute provided by Graphviz. Basically, `penwidth` is an edge attribute just like `label`, so both of these attributes can be applied to an edge, for example as follows: `a->b [label="3", penwidth=1.0]`. 

---

**Fig. 4** Output graph generated from an event log with a large number of instances

**Fig. 5** Adjusting the edge thickness according to the transition count
Typically, label is a string (if it is a number, it will be converted to a string), but penwidth is a numerical value. The default value for penwidth is 1.0. In Fig. 4, all edges have this default value.

To produce a similar graph to that in Fig. 5, the edge thickness must be increased for those transitions which have a higher transition count. However, we do not want the edge thickness to become excessively large or excessively small. Therefore, a good practice is to define minimum and maximum values for the edge thickness, and associate them with the minimum and maximum transition counts.

Let $x$ denote a transition count, and let $y$ denote the corresponding edge thickness. If we want to have a linear relationship between $x$ and $y$, we can use the following expression:

$$y = y_{\text{min}} + \left( y_{\text{max}} - y_{\text{min}} \right) \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}$$

It is easy to verify that when $x = x_{\text{min}}$ (minimum transition count), the expression yields $y = y_{\text{min}}$ (minimum edge thickness), and when $x = x_{\text{max}}$ (maximum transition count), the expression yields $y = y_{\text{max}}$ (maximum edge thickness).

Listing 15 shows how to set the edge thickness according to this expression. The differences in comparison to Listing 14 are the following:

- Line 8 puts all the transition counts in a list. A Python technique known as list comprehension is being used here to do this in a single line of code.
- Lines 9 and 10 compute the minimum and maximum values in that list (i.e. the minimum and maximum transition counts found in the matrix).

Listing 15 Setting the edge thickness according to the transition count

```python
import pygraphviz as pgv
G = pgv.AGraph(strict=False, directed=True)
G.graph_attr['rankdir'] = 'LR'
G.node_attr['shape'] = 'box'
values = [F[ai][aj] for ai in F for aj in F[ai]]
x_min = min(values)
x_max = max(values)
y_min = 1.0
y_max = 5.0
for ai in F:
    for aj in F[ai]:
        x = F[ai][aj]
        y = y_min + (y_max-y_min) * float(x-x_min) / float(x_max-x_min)
        G.add_edge(ai, aj, label=x, penwidth=y)
G.draw('graph.png', prog='dot')
```
Fig. 6  Output graph generated from the previous script

Fig. 7  Including the activity counts in each node

- Lines 12 and 13 define the minimum and maximum edge thickness, respectively.
- Line 17 stores each transition count in a variable, to be used in subsequent lines.
- Line 18 calculates the edge thickness for the given transition count (x).
- Line 19 adds an edge to the graph, now with the extra attribute `penwidth`.

The output of this code for the event log in Listing 1 on page 7 is shown in Fig. 6.

2.7 Activity Counts

In addition to transition counts, sometimes it is useful to display also activity counts, i.e. the number of times that each activity (same task name) appears in the event log. Such activity counts are also useful to identify the most common paths in the process, especially when there is a large number of transitions in the graph.

For example, consider the graph shown in Fig. 7. This is the same graph of Fig. 5, with the difference that now it includes the activity count in each node.

A quick look at this graph reveals that there are actually two groups of instances: those that end in c (1866 instances) and those that follow the other branch and end in h (5683 instances). When the graph becomes more complex, this kind of conclusion may be more difficult to reach by looking at the transition counts alone.

Therefore, it is useful to calculate the activity counts and include them in the graph. We will show how to do this in two separate listings. First, Listing 16 shows how to calculate the activity counts.
Listing 16  Calculating the activity counts
1 A = dict()
2 for caseid in log:
3     for i in range(0, len(log[caseid])):
4         ai = log[caseid][i][0]
5         if ai not in A:
6             A[ai] = 0
7             A[ai] += 1

Listing 17  Including the activity counts in the graph
1 import pygraphviz as pgv
2 G = pgv.AGraph(strict=False, directed=True)
3 G.graph_attr['rankdir'] = 'LR'
4 G.node_attr['shape'] = 'box'
5 for ai in A:
6     text = ai + '\n(' + str(A[ai]) + ')
7     G.add_node(ai, label=text)
8 values = [F[ai][aj] for ai in F for aj in F[ai]]
9     x_min = min(values)
10    x_max = max(values)
11    y_min = 1.0
12    y_max = 5.0
13 for ai in F:
14    for aj in F[ai]:
15        x = F[ai][aj]
16        y = y_min + (y_max-y_min) * float(x-x_min) / float(x_max-x_min)
17        G.add_edge(ai, aj, label=x, penwidth=y)
18 G.draw('graph.png', prog='dot')

After reading the event log (see Listing 8 on page 11), we create a dictionary \( A \) to store the activity counts. A loop goes through each case id (line 2), and another loop goes through every event in that case id (lines 3). For each event, the task is stored in variable \( ai \). If this task has not been seen before, it is inserted in the dictionary with an count of zero. Then, its count is incremented by 1.

This is a simple way to count the number of occurrences of each task. Now let us look at how to include this count in the graph. Listing 17 shows how to do this.

This is the same code as in Listing 15 except for lines 8–10. In these lines, we go through each task in the dictionary \( A \) and add a node to the graph (line 10). The node name is equal to the task name (\( ai \)). However, its label includes additional information. Specifically, the label is the result of appending the task name with the activity count inside parenthesis, and with a newline character in between (line 9).

For the event log of Listing 1 on page 7, the resulting graph is shown in Fig. 8.
Fig. 8  Output graph generated from the previous script

It should be noted that in Listing 17 the graph is being built by first adding the nodes and only then adding the edges. Naturally, the node names that are used when adding the nodes must be the same that are used when adding the edges. The additional information about the activity counts is being included in the node labels, not in the node names.

2.8 Node Coloring

A further improvement that can be done to the graph is to color the nodes according to their activity counts. Graphviz provides an extensive set of colors, including 100 different shades of gray. This is what we will be using here. A lighter shade of gray will correspond to a lower activity count, and a darker shade of gray will correspond to a higher activity count.

To get maximum contrast, we will make the minimum activity count correspond to white, and the maximum activity count correspond to black. Any values in between will correspond to some intermediate shade of gray. With this correspondence, the graph for a large event log could look like the one in Fig. 9.

The fill color of each node now provides a visual cue of which nodes have similar activity counts. In particular, the two groups of instances mentioned in the previous section (i.e. the ones that go through activity c and the ones that go through activity d) are now easily distinguishable by their color shading.

Activities a and b have maximum shading since they have the maximum activity count. Also, note that the font color in these nodes has been changed to white in order to be readable over a dark background.
2.8 Node Coloring

Fig. 9 Adjusting the node color according to the activity count

Listing 18 Adding fill color and setting the font color of nodes
1    import pygraphviz as pgv
2    G = pgv.AGraph(strict=False, directed=True)
3    G.graph_attr['rankdir'] = 'LR'
4    G.node_attr['shape'] = 'box'
5    x_min = min(A.values())
6    x_max = max(A.values())
7    for ai in A:
8        text = ai + '
9        gray = int(float(x_max - A[ai]) / float(x_max - x_min) * 100.)
10       fill = 'gray' + str(gray)
11       font = 'black'
12       if gray < 50:
13          font = 'white'
14       G.add_node(ai, label=text, style='filled', fillcolor=fill, fontcolor=font)
15    values = [F[ai][aj] for ai in F for aj in F[ai]]
16    x_min = min(values)
17    x_max = max(values)
18    for ai in F:
19       for aj in F[ai]:
20          x = F[ai][aj]
21          y = y_min + (y_max-y_min) * float(x-x_min) / float(x_max-x_min)
22          G.add_edge(ai, aj, label=x, penwidth=y)
23    G.draw('graph.png', prog='dot')

Listing 18 shows how the graph in Fig. 9 has been generated. The new block of code is in lines 8–18.

As before, the node label includes the task name and the activity count in parenthesis (line 12). Then, according to the activity count, a gray level is chosen (line 13). In Graphviz, gray0 corresponds to black and gray100 corresponds to white. Therefore, we must convert the activity count into a gray level between 0 and 100, where 0 (black) corresponds to the maximum activity count, and 100 (white) corresponds to the minimum activity count.
If the activity count is denoted as $x$ and the gray level is denoted as $y$, then the expression to convert an activity count to a gray level is:

$$y = \frac{x_{max} - x}{x_{max} - x_{min}} \times 100$$

It is easy to check that when the activity count is maximum ($x = x_{max}$), the expression yields 0 (black), as desired. On the other hand, if the activity count is minimum ($x = x_{min}$), the expression yields 100 (white).

The minimum and maximum activity counts have been computed before in lines 8–9. In line 13, the script applies the expression above and converts the result to an integer between 0 and 100. In line 14, the script uses this result to pick the correct shade of gray, as a Graphviz color between gray0 and gray100.

The choice of font color happens in lines 15–17. Line 15 sets the font color to the default value of black but, if the gray level is below 50 (meaning that the fill color is dark), the font color is switched to white in line 17.

Finally, the script adds the node to the graph in line 18. Several attributes are being specified: the node label, the style (to ensure that the node is actually filled), the fill color, and the font color.

### 2.9 Summary

Here is a brief recap of what we have learned in this chapter:

- The aim of the control-flow perspective is to extract a model of the sequence of activities from the event log. This is done by counting the transitions between successive tasks with the same case id.
- Such transition counts can be stored in a transition matrix, which is the basis for generating an output graph.
- Graphviz is a tool for drawing graphs. It takes care of the layout of nodes and edges automatically, so the minimum information required to draw a graph is a list of edges between nodes.
- Graphviz has its own language to define graphs, but there is no need to write such definitions by hand. PyGraphviz provides a convenient interface to generate such definitions from Python code.
- Both nodes and edges have attributes that can be used to improve the graph. For better visualization, edge thickness can be adjusted according to transition counts, and node color can be adjusted according to activity counts.
- Both transition counts and activity counts can be used to discover the most frequent paths (i.e. the typical behavior) in the process.

Listing 19 shows a complete script with what we have learned in this chapter.
Listing 19  Complete script for reading the event log and generating the control-flow graph

```python
import pygraphviz as pgv

f = open('eventlog.csv', 'r')
log = dict()
for line in f:
    line = line.strip()
    if len(line) == 0:
        continue
    [caseid, task, user, timestamp] = line.split(';')
if caseid not in log:
    log[caseid] = []
event = (task, user, timestamp)
    log[caseid].append(event)
f.close()

F = dict()
for caseid in log:
    for i in range(0, len(log[caseid]) - 1):
        ai = log[caseid][i][0]
        aj = log[caseid][i+1][0]
        if ai not in F:
            F[ai] = dict()
        if aj not in F[ai]:
            F[ai][aj] = 0
        F[ai][aj] += 1

A = dict()
for caseid in log:
    for i in range(0, len(log[caseid])):
        ai = log[caseid][i][0]
        if ai not in A:
            A[ai] = 0
        A[ai] += 1

G = pgv.AGraph(strict=False, directed=True)
G.graph_attr['rankdir'] = 'LR'
G.node_attr['shape'] = 'box'

x_min = min(A.values())
x_max = max(A.values())
for ai in A:
    text = ai + '
    (' + str(A[ai]) + ')
    gray = int(float(x_max - A[ai]) / float(x_max - x_min) * 100.)
    fill = 'gray' + str(gray)
    font = 'black'
    if gray < 50:
        font = 'white'
    G.add_node(ai, label=text, style='filled', fillcolor=fill, fontcolor=font)

values = [F[ai][aj] for ai in F for aj in F[ai]]
x_min = min(values)
x_max = max(values)
y_min = 1.0
y_max = 5.0
for ai in F:
    for aj in F[ai]:
        x = F[ai][aj]
        y = y_min + (y_max - y_min) * float(x - x_min) / float(x_max - x_min)
        G.add_edge(ai, aj, label=x, penwidth=y)
G.draw('graph.png', prog='dot')
```
A Primer on Process Mining
Practical Skills with Python and Graphviz
Ferreira, D.R.
2017, VIII, 96 p. 37 illus., 15 illus. in color., Softcover
ISBN: 978-3-319-56426-5