TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Power graph visualizations for event logs

Sonke, W.M.

Award date:
2015

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Technische Universiteit
Eindhoven
University of Technology

Power graph visualizations for event logs

Master's thesis

Willem Sonke

Supervisor: **prof. dr. Bettina Speckmann**
Tutor: **dr. ir. Eric Verbeek**

August 31, 2015

Contents

1 Introduction **4**
1.1 Mining algorithms 6
1.2 The directly-follows graph 7
1.3 Concurrency 9
1.4 Edge and vertex bundling 11
1.5 Results in this thesis 11

2 Confluent drawings **13**
2.1 Definition 13
2.2 Confluent graphs 14
2.3 Algorithm 16
2.4 Results 17

3 Strict power graphs **19**
3.1 Definition 20
3.2 The greedy algorithm 22
3.3 Results 23

4 Fuzzy power graphs **26**
4.1 Definition 27
4.2 A naive algorithm 27
4.3 Problems with this approach 28
4.4 An improved algorithm 29
4.5 Results 32

5 Implementation **34**
5.1 WebCola 34
5.2 Bug fixes 34
5.2.1 Browser incompatibility 35
5.2.2 Edge reversal bug 35
5.2.3 Crash in the grid router 35
5.3 Fuzzy power graph algorithm 36

5.4 Results for the repair shop example 37

6 Evaluation **39**
6.1 Data sets 39
6.1.1 Repair shop 39
6.1.2 Generated data sets 40
6.1.3 Financial log 41
6.2 Generated data sets 41
6.3 The financial log 43

7 Conclusion **44**

References **46**

A Screenshots of results **48**
A.1 Generated data sets 48
A.2 The financial log 55

Chapter 1

Introduction

Nowadays, there are many complicated processes that play important roles in our society. Examples include the process by which support requests are handled by a company and the process by which a county decides whether or not to grant a building permit. Let us consider an example process in some more detail. The example is based on a workshop that repairs broken telephones. In this workshop, the process of repairing phones works as follows.

1. A customer delivers a broken phone; it gets registered in the system.

2. An employee analyzes what the cause of the defect is.

3. The phone is dispatched to one of two repair teams. One of the teams is able to perform simple repairs; the other team handles complex repairs. After the repair is done, someone tests whether the repair was successful.

4. While the phone is being repaired, an employee informs the customer of the cause of the defect.

5. If the repair was successful, the phone is returned to the customer and the case is filed in an archive. If the defect has not been fixed, the phone is dispatched back to one of the repair teams, after which a new test is performed, and so on.

Often it is necessary to study a process like these, for example for quality control, or to discover how the process could be improved. In order to investigate a process, we need to know how it works in practice. Therefore, it is useful to record a log of what happens during the execution of the process.

Let us give a formal definition of such a log file. We assume that the process is executed often, and that within the process we can distinguish distinct activities, that

we can represent by some name. In our repair shop example, we identify the following activities: *Register*, *Analyze Defect*, *Repair (Simple)*, *Repair (Complex)*, *Test Repair*, *Restart Repair*, *Inform User* and *Archive Repair*. To collect log data about the process, we just write down, for a series of executions, a list of all activities that happened. This gives rise to the following definition.

Definition 1.1 (Traces and logs)**.** A *trace* is an ordered list of activities. A particular occurrence of an activity within a trace is called an *event*. A *log* is a multiset of traces.

We model a log as a multiset of traces since a log may contain multiple traces that consist of exactly the same series of activities.

In Figure 1.1, three possible traces from our repairshop example are shown. Furthermore, to show what a log looks like in practice, we show a tiny part of a real log (only three events from a single trace) from the repairshop in Figure 1.2. We see that there is a wealth of information in this log. It goes much beyond a simple list of activities: for every event a timestamp and a resource (the entity performing the event) is stored. However, in this thesis we do not consider those additional attributes.

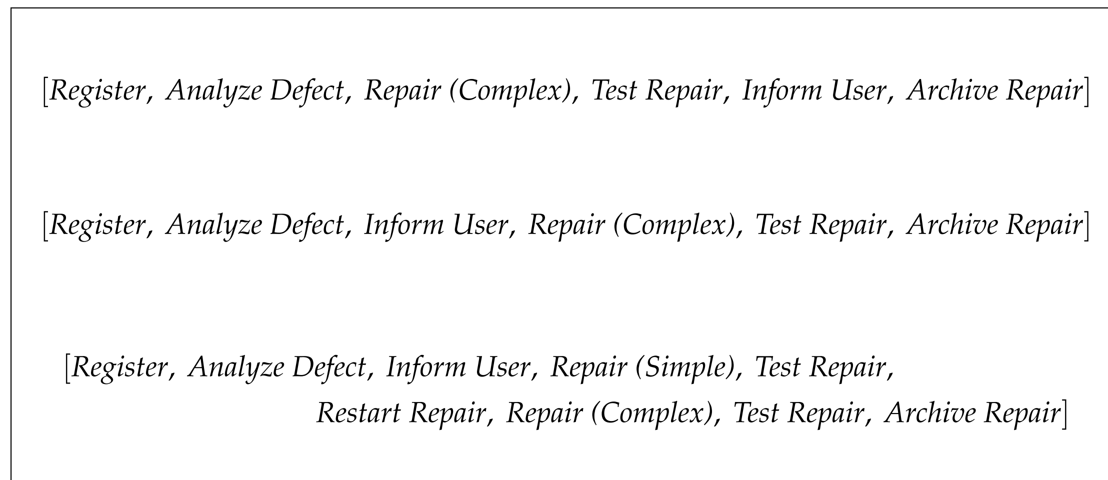


Figure 1.1: Three example traces from the repairshop example.

From the three traces in Figure 1.1 we can already draw some conclusions. For example, all traces start with *Register*, and all of them end with *Archive Repair*. Also, we can see from the traces that there are two types of repairs: simple and complex ones. Inspecting real-world log data by hand is not easy however, since real-world logs are huge: every single event that happens during the process is logged.

Hence, we want to analyze logs in an automated fashion. More precisely, for the process at hand, we want to find a description of the process from a log of its execution. This is called *process mining*. For a broad overview of process mining, we refer to the seminal work by Van der Aalst *et al.* [2].

```
<event>
    <string key="org:resource" value="System"/>
    <date key="time:timestamp" value="1970-01-02T12:23:00.000+01:00"/>
    <string key="concept:name" value="Register"/>
    <string key="lifecycle:transition" value="complete"/>
</event>
<event>
    <string key="org:resource" value="Tester3"/>
    <date key="time:timestamp" value="1970-01-02T12:23:00.000+01:00"/>
    <string key="concept:name" value="Analyze Defect"/>
    <string key="lifecycle:transition" value="start"/>
</event>
<event>
    <string key="defectType" value="6"/>
    <string key="org:resource" value="Tester3"/>
    <string key="lifecycle:transition" value="complete"/>
    <string key="phoneType" value="T2"/>
    <date key="time:timestamp" value="1970-01-02T12:30:00.000+01:00"/>
    <string key="concept:name" value="Analyze Defect"/>
</event>
```

Figure 1.2: A small part of a log file as it is stored on disk.

1.1 Mining algorithms

An algorithm that takes a log and produces a process description is called a *mining algorithm* or a *discovery algorithm*. The resulting process description can be expressed in various formalisms. A formalism that is very well-known in the process mining community is the *Petri net*. We use the definition by Van der Aalst [1] here.

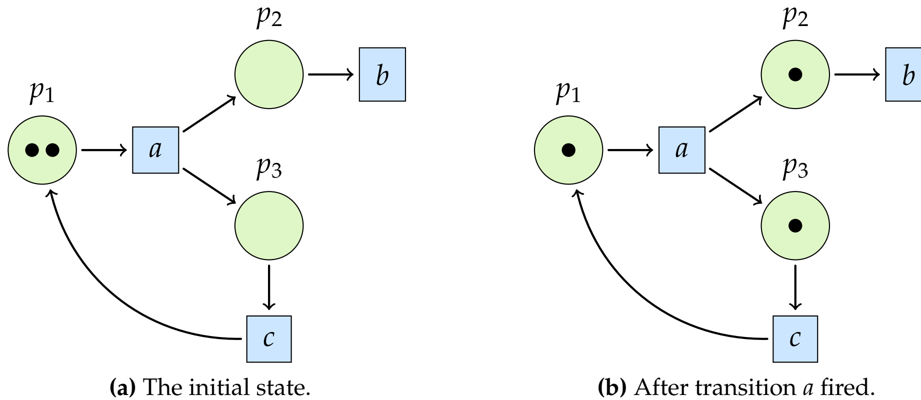
Definition 1.2 (Petri net)**.** A *Petri net* is a directed graph $(P \cup T, F)$ where P is a set of *places*, T is a set of *transitions* and $F \subseteq (P \times T) \cup (T \times P)$ is a set of edges.

Note that edges can exist between places and transitions and between transitions and places, but not between places and places or between transitions and transitions. In diagrams, we depict places as circles and transitions as rectangles. An example Petri net is shown in Figure 1.3a.

A Petri net describes a process by representing states the process can be in by a *marking*. A marking is a function $M : P \to \mathbb{N}$ that assigns to every place a non-negative number of *tokens*. In drawings of Petri nets, we visualize the tokens by drawing $M(p)$ dots in every place p. In Figure 1.3a for example, the place p_1 contains two tokens.

The transitions in the Petri net model the way the process can go from one state into another. If at least one token is present in all places p with an edge (p, t), a transition t can *fire*, meaning that it changes the marking by taking away one token from all those places p, and it puts one new token in all places p' with an edge (t, p'). In Figure 1.3a transition a can fire; the result of this operation is shown in Figure 1.3b.

A well-known example of a mining algorithm is the *α-algorithm* [3], that produces process descriptions in the form of Petri nets. Just like most other mining algorithms, it



(a) The initial state.

(b) After transition *a* fired.

Figure 1.3: An example of a simple Petri net.

is based on the *directly-follows relation*. This relation captures which activities happen directly after each other in the log.

Definition 1.3 (Directly-follows relation)**.** Let L be a log file and A the set of activities occuring in it. The *directly-follows relation* of L is a relation $>_L \subseteq A \times A$ where $(a_1, a_2) \in >_L$ iff

$$\exists t \in L : a_2 \text{ directly follows } a_1 \text{ in } t.$$

The α-algorithm is based on a set of rules that define a Petri net based on the directly-follows relation. For every activity that occurs in the log, the algorithm produces a transition in the Petri net. The places in the resulting Petri net correspond to pairs of subsets of activities. A full description of the algorithm would be quite lengthy and is out of the scope of this thesis; for more details we refer to Van der Aalst *et al.* [3].

There are many more mining algorithms. Examples include a mining algorithm based on heuristics [21] and the so-called *inductive visual miner* [14, 15]. Like the α-algorithm, those algorithms both use the directly-follows relation to produce the resulting process model.

The process mining community has created implementations of mining algorithms. A large software package called *ProM*[1] [19] implements many mining algorithms, and is also able to visualize the resulting process models. In Figure 1.4, a screenshot of ProM is shown.

1.2 The directly-follows graph

As noted in the previous section, the usual workflow in process mining is to use some mining algorithm on the log to obtain a model of the process, and then inspect this

[1] ProM can be downloaded from `http://promtools.org`.

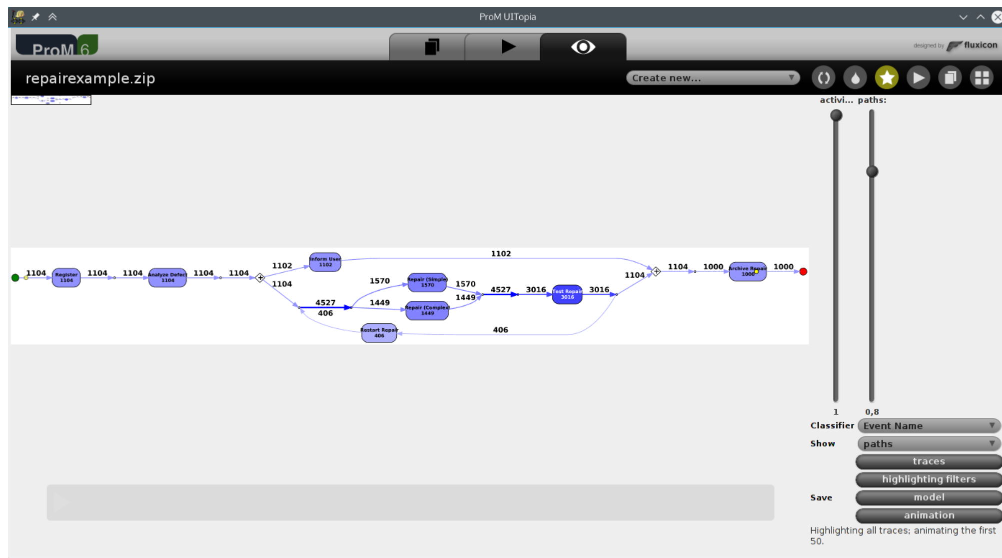


Figure 1.4: A screenshot of the ProM workbench.

process model. This approach has a disadvantage, however. Every mining algorithm assumes that the analyzed process satisfies the formalism used by the algorithm to express the process model. For example, as we saw in the previous section, the α-algorithm produces Petri nets. Hence, it is only able to reconstruct useful models for processes that can actually be represented as a Petri net. This phenomenon that patterns that are not supported by the formalism will not be represented correctly in the resulting process model, is called a *representational bias* [2].

To avoid this representational bias, it can be useful to inspect the log directly, without the use of a mining algorithm. Still, we do not want to do this inspection by reading the raw log file. Hence, it can be useful to visualize the directly-follows relation.

To visualize the directly-follows relation $>_L$, we can represent it as the graph $(A, >_L)$. So, we use a vertex for every activity, and insert a directed edge (a_1, a_2) if activity a_2 directly follows activity a_1 in some trace from the log. We call this graph the *directly-follows graph*. The classic way of visualizing a graph is a *node-link drawing*, where we represent the vertices by dots, and the edges by arrows from one dot to another. As an example, in Figure 1.5, the directly-follows graph is drawn for the event log that consists of the three traces from the telephone repair process shown in Figure 1.1.

However, such a visualization is not often used in practice. The reason for this is that the directly-follows graph is usually very dense and non-planar. Hence, the drawing becomes cluttered and hard to read. As an example, in Figure 1.6 the directly-follows graph of the real-world repairshop log is shown. While this process is not very complicated, the drawing becomes quite cluttered already.

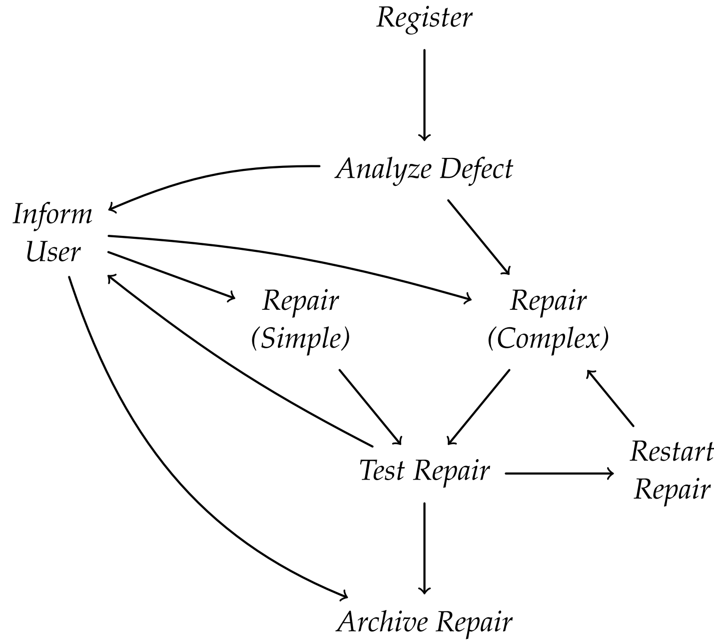


Figure 1.5: The directly-follows graph for the three traces in Figure 1.1.

1.3 Concurrency

We note that the clutter in Figure 1.6 is mainly caused by the large number of edges from and to *Inform User*. In fact, *Inform User* has such a large degree because it does not matter when *Inform User* is executed during the repair, as long as it is done. We say that *Inform User* is executed *concurrently* with *Repair (Simple)*, *Repair (Complex)*, *Restart Repair* and *Test Repair*.

More generally, this pattern, named *concurrency* or *parallellism*, means that two or more parts of the process are executed at the same time, without any synchronization. This means any possible ordering between those parts can occur in a trace. For example, consider two separate teams that are going to execute activities a_1, a_2, a_3 and b_1, b_2, b_3, respectively. Then, although a_1 must always happen before a_2, and likewise, b_1 must happen before b_2, there is no fixed ordering between, say, a_2 and b_2. Hence, resulting traces can include $[a_1, a_2, a_3, b_1, b_2, b_3]$, $[b_1, b_2, b_3, a_1, a_2, a_3]$ and $[a_1, b_1, a_2, a_3, b_2, b_3]$.

In the directly-follows graph concurrency manifests itself as a large number of edges between $\{a_1, a_2, a_3\}$ and $\{b_1, b_2, b_3\}$ (see Figure 1.7), corresponding to all switches between $\{a_1, a_2, a_3\}$ and $\{b_1, b_2, b_3\}$ that are present in the log. For example, with the above traces we get the edges (a_3, b_1), (b_3, a_1), (a_1, b_1), (b_1, a_2) and (a_3, b_2). We can also see this pattern occurring in Figure 1.5: all edges between *Inform User* and *Repair (Simple)*, *Repair (Complex)*, *Restart Repair* and *Test Repair* are present.

In the ideal case, any edge between $\{a_1, a_2, a_3\}$ and $\{b_1, b_2, b_3\}$ will be present. How-

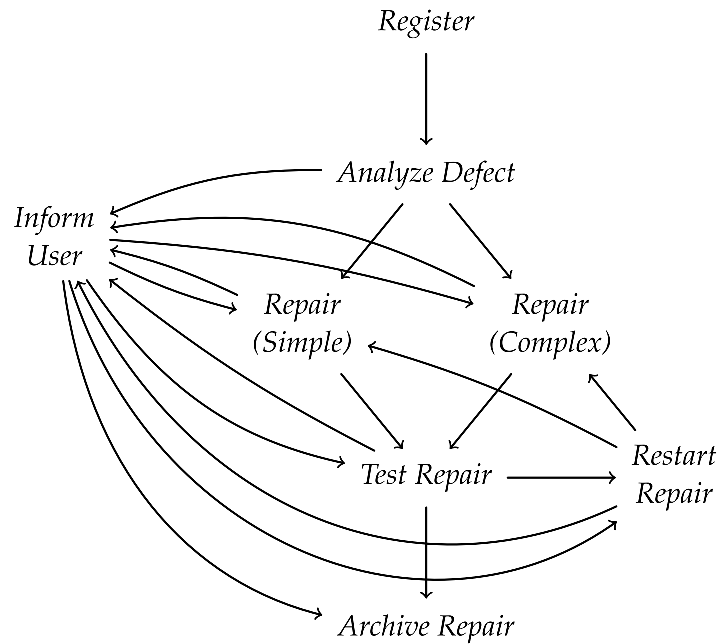


Figure 1.6: The directly-follows graph for the repairshop example log.

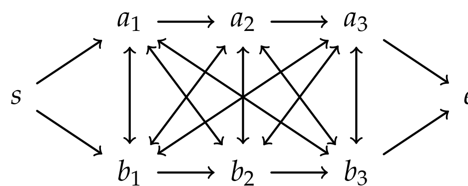


Figure 1.7: Concurrency in the directly-follows graph.

ever, this will happen only if we have enough traces that happen to cover all of the switches. Especially if the concurrent parts of the process contain a lot of activities, this will probably not happen in practice. For example, in Figure 1.5 a lot of the edges between *Inform User* and the other activities are missing, while those activities could certainly happen directly after each other in the underlying process, as evidenced by Figure 1.6.

Concurrency often occurs in real-world processes, so a node-link drawing of the directly-follows graph, like the one in Figure 1.6, will often be cluttered. Hence it makes sense to try to simplify node-link drawings like these. This is what we are investigating in this thesis.

1.4 Edge and vertex bundling

The problem of making drawings of dense, non-planar graphs easier to read has been studied extensively. There are many well-known techniques to accomplish this.

A major approach is called *edge bundling*. This means that instead of drawing edges all as separate line segments or curves, edges are grouped together. This reduces the amount of clutter in the drawing. For determining which edges are grouped together and how the grouping is performed, many techniques have been invented.

Many edge bundling techniques introduce *ambiguity*, which means that it is not possible to correctly reconstruct the edges from the original graph by using just the resulting drawing. There are also ambiguity-free techniques, that produce drawings from which we can completely reconstruct the original graph. This necessarily leaves more information in the drawing. Hence, ambiguity-free techniques are generally suited best to drawings of smaller graphs. Techniques that do introduce ambiguity, on the other hand, throw away information and are suited to drawings with a larger amount of edges.

Some examples of the many proposed edge bundling techniques that introduce ambiguity are the following. Holten [10] presented a technique, called *hierarchical edge bundles*, that visualizes graphs with a hierarchical structure using any tree layout algorithm. Holten and Van Wijk [11] also presented *force-directed edge bundling*, where edges are positioned by a force-directed method. Lambert *et al.* [13] proposed a method that routes the edges over a grid graph. All of these methods result in drawings where edges are represented by smooth curves that are grouped in bundles.

On the other hand there are ambiguity-free edge bundling techniques. Luo *et al.* [16] proposed a method to do this, that is based on making bundles in which all edges share a common node. Another technique, *confluent drawings*, was invented by Dickerson *et al.* [4]. In this thesis, we discuss confluent drawings, including related work, in Chapter 2.

A radically different approach is used by *vertex bundling* methods. Instead of grouping edges together, like the edge bundling methods, vertex bundling methods create groups of vertices to reduce the amount of clutter in the drawing. In particular, we consider a technique called *power graph analysis*. This ambiguity-free technique was pioneered by Royer *et al.* [18] in the context of protein networks and later refined by Dwyer [6, 5]. We study power graph analysis and related work on it in Chapter 3.

1.5 Results in this thesis

In this thesis, we set out to investigate edge and vertex bundling techniques for the visualization of directly-follows graphs. In particular, we study confluent drawings in Chapter 2 and power graph analysis in Chapter 3. We see that both techniques can be useful for visualizing a directly-follows graph. However, the drawings created by power graph analysis seem to be less cluttered than the confluent drawings; hence, we think

that the former are the most useful in practice.

Power graph analysis is an ambiguity-free technique. As already noted in the previous section, ambiguity-free techniques are generally best suited for small graphs. Indeed in practice turns out that the drawings of large directly-follows graphs are not readable anymore because the large complexity of the drawing (we show examples of this in the experimental evaluation in Chapter 6). As far as we know, there is no vertex bundling technique known yet that allows ambiguity. Therefore we invented a variation on power graph analysis that we call *fuzzy power graphs*. We present this technique in Chapter 4.

To experiment with power graph analysis and fuzzy power graphs, we made an implementation of those algorithms. We discuss this in more detail in Chapter 5. Then, we ran the implementation on several logs to see how well the methods worked. The results are presented in Chapter 6.

Chapter 2

Confluent drawings

In this chapter, we study confluent drawings in more detail. Confluent drawings are an edge bundling technique to make drawings of dense graphs clearer. However, unlike most edge bundling techniques, confluent drawings are ambiguity-free.

Confluent drawings were first proposed by Dickerson *et al.* [4]. After they were introduced, confluent drawings have been extensively researched. As we will see in Section 2.2, not all graphs admit a confluent drawing. Therefore, in particular, researchers have also studied the problem of recognizing which graphs admit a confluent drawing. Hui *et al.* [12] proposed a variant on confluent drawings, called *strong confluent drawings*, that is defined slightly differently. They showed that it is possible to test whether a graph admits a strongly confluent drawing is in *NP*. Eppstein *et al.* [7] introduced the Δ-*confluent drawings*, and showed that the graphs that admit a Δ-confluent drawing are exactly the distance-hereditary graphs, that can be recognized in linear time. Furthermore, Eppstein *et al.* [9] introduced another class of graphs, namely the graphs that admit a so-called *strict confluent drawing*, and showed that testing whether a graph is in this class is *NP*-complete. To our knowledge however, the complexity of deciding whether a general graph admits a confluent drawing is still open.

2.1 Definition

In this section, we follow the definitions for confluent drawings as given in the paper by Dickerson *et al.* [4] that originally introduced them. Confluent drawings can be used for both undirected and directed graphs. We will give both definitions.

A *locally-monotone curve* is a smooth curve that does not self-intersect in any point.

Definition 2.1 (Undirected confluent drawing)**.** Let $G = (V, E)$ be an undirected graph. A *confluent drawing* of G is a drawing A consisting of curves in $\mathbb{R}^2$ where

1. every $v \in V$ is represented by a point $v' \in A$,
2. $(v_i, v_j) \in E$ iff there is a locally-monotone curve in A that connects v'_i and v'_j, and
3. curves are allowed to overlap, but curves are not allowed to intersect each other.

If G admits a confluent drawing, we call G *confluent*.

The definition for confluent drawings of directed graphs is very similar.

Definition 2.2 (Directed confluent drawing)**.** Let $G = (V, E)$ be a directed graph. A *confluent drawing* of G is a drawing A consisting of directed curves in $\mathbb{R}^2$ where

1. every $v \in V$ is represented by a point $v' \in A$,
2. $(v_i, v_j) \in E$ iff there is a directed locally-monotone curve in the drawing that runs from v'_i to v'_j,
3. curves are allowed to overlap, but curves are not allowed to intersect each other, and
4. if curves overlap, they must share the same direction at the overlapping part.

If G admits a confluent drawing, we call G *confluent*.

The idea behind confluent drawings is to merge edges together. A confluent drawing represents every edge from the original graph by a smooth curve; see Figure 2.1. While the original graph is non-planar and cluttered, the confluent drawing does not contain intersecting curves and contains much less clutter.

Figure 2.1: The idea of confluent drawings: we represent every edge from the original directed complete bipartite graph $K_{3,3}$ *(left)* by a smooth curve in the confluent drawing *(right)*.

2.2 Confluent graphs

Not all graphs are confluent. For the undirected case, Dickerson *et al.* presented several graph classes of which all members are confluent. However, they also showed that there exist non-confluent graphs. For example, the Petersen graph (see Figure 2.2) is non-confluent. Dickerson *et al.* do not discuss whether there exist non-confluent directed graphs. However, the next lemma confirms this.

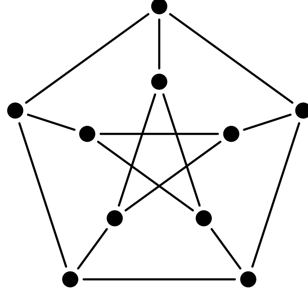


Figure 2.2: The Petersen graph, an example of a non-confluent undirected graph.

Lemma 2.3. *Let $G = (V, E)$ be an undirected graph, and let $G' = (V, E')$ be a directed graph obtained by directing the edges of G in an arbitrary way. If G is non-confluent, then G' is non-confluent either.*

Proof. We prove that if G' is confluent, G is also confluent. Let A' be a confluent drawing of G'. We claim that the drawing A, obtained by replacing all of the curves in A' by undirected curves, is a confluent drawing of G, hence proving the lemma.

To prove the claim, we show that A satisfies the three requirements of Definition 2.1.

1. This follows trivially from requirement 1 of Definition 2.2.

2. We prove both directions separately.

   Let $\{v_i, v_j\} \in E$ be an edge in G. Then either (v_i, v_j) or (v_j, v_i) is in E'. Without loss of generality, assume $(v_i, v_j) \in E'$. By requirement 2 of Definition 2.2 there is a locally-monotone curve from v_i' to v_j' in A'. Since we constructed A by removing the directions from the curves in A', there is a locally-monotone curve between v_i' and v_j' in A.

   For the other direction, take some $\{v_i, v_j\} \notin E$. Then neither (v_i, v_j) nor (v_j, v_i) are in E'. Hence, again by requirement 2 of Definition 2.2, in A' there is no locally-monotone curve from v_i' to v_j' and no locally-monotone curve from v_j' to v_i'. Now assume a locally-monotone curve between v_i' and v_j' does exist in A. Then in A' not all parts of this curve can have the same direction – otherwise, there was a locally-monotone curve from v_i' to v_j' or from v_j' to v_i'. That means that there must be some point in A' where curves of the opposite direction meet. This is disallowed by requirement 4 of Definition 2.2, hence, a locally-monotone curve between v_i' and v_j' cannot exist in A.

3. This follows trivially from requirement 3 of Definition 2.2. □

Since the Petersen graph is non-confluent, any directed graph obtained by directing its edges is also non-confluent. Furthermore, for any graph $G = (V, E)$ we can create a log that has G as its directly-follows graph. This can be done, for example, by making a

trace $[a_1, a_2]$ for every edge $(a_1, a_2) \in E$. Hence, we can create a log that has a directed version of the Petersen graph as its directly-follows graph. From this we conclude that we cannot guarantee that the confluent drawing of a directly-follows graph exists. Clearly, this is not desirable.

However, there is a simple solution to this problem. In the remainder of this chapter, we drop requirement 3 (that curves are not allowed to intersect) from Definitions 2.1 and 2.2. Now every graph becomes confluent: any (non-necessarily planar) node-link drawing is already a valid confluent drawing. The same approach of dropping the planarity requirement is also taken for the same reason by Eppstein *et al.* [8] for producing *confluent layered drawings*.

2.3 Algorithm

Dickerson *et al.* proposed simple heuristic algorithms to produce undirected and directed confluent drawings [4]. We consider only the version for directed confluent drawings here, The algorithm iteratively replaces all edges of a maximal directed complete bipartite subgraph by a series of curves; basically it repeatedly applies the transformation in Figure 2.1.

The original algorithm by Dickerson *et al.* includes the planarity restriction that we dropped. Hence, it produces planar confluent drawings only, and returns *fail* if it cannot find one. Since we dropped the planarity restriction, we modified the algorithm so that it just tries to perform as many replacements as possible, and stops when there are no complete bipartite subgraphs anymore to replace.

Also, the original algorithm never replaces bipartite subgraphs with only one vertex at a side. This is done since performing such a replacement never helps to make the drawing planar. However, since we are not interested in obtaining a planar drawing anymore, we also replace such small bipartite graphs. This results in the algorithm CONFLUENT-DRAWING that is given as pseudocode in Figure 2.3.

The algorithm first replaces the bipartite subgraph it found by a single vertex. Then it performs a recursive call to find more bipartite graphs in the remainder of the graph; during this search it can use the new vertex. Only when the recursive call returns, the new vertex is replaced again by switches.

To see why this is useful, see Figure 2.4. First the algorithm finds a bipartite subgraph and replaces it by a single vertex v. After that, there is a new bipartite subgraph that is replaced by a new vertex v', after which the algorithm is ready and replaces all new vertices by pairs of switches. Note that if v would not have been inserted after the first replacement, the algorithm would not be able to find the second replacement, which would enable less simplification.

Algorithm CONFLUENT-DRAWING($G = (V, E)$)
1 find a maximal bipartite complete subgraph
2 **if** there is none:
3   **return**
4 $L, R \leftarrow$ the left and right side of the subgraph
5 remove all edges from L to R
6 add a new vertex v; add edges from L to v and from v to R
7 CONFLUENT-DRAWING(G)
8 remove v and its adjacent edges again
9 add curves between L and R as shown in the right part of Figure 2.1

Figure 2.3: A heuristic algorithm to find a confluent drawing of a directed graph. This algorithm is adapted from algorithm HEURISTICDRAWUNDIRECTED by Dickerson *et al.* [4].

2.4 Results

In Figure 2.5, a confluent drawing of the repair shop example is shown. We did not implement the algorithm. Hence, we obtained this drawing by executing the algorithm by hand on the directly-follows graph of the log file.

Some properties of the process are clear in this drawing: we can see that it is possible to loop from *Test Repair* back to the repair activities via *Restart Repair*. Unfortunately, the result still seems rather messy and unstructured, even while we layouted the graph by hand to obtain a clear drawing. Also, the concurrency between *Inform User* and *Repair (Simple)*, *Repair (Complex)*, *Restart Repair* and *Test Repair* is not clearly visible.

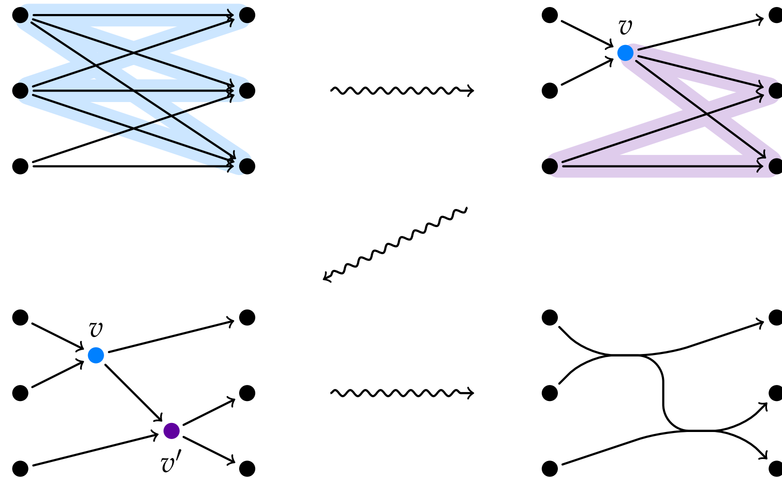


Figure 2.4: An example of the execution of algorithm CONFLUENT-DRAWING.

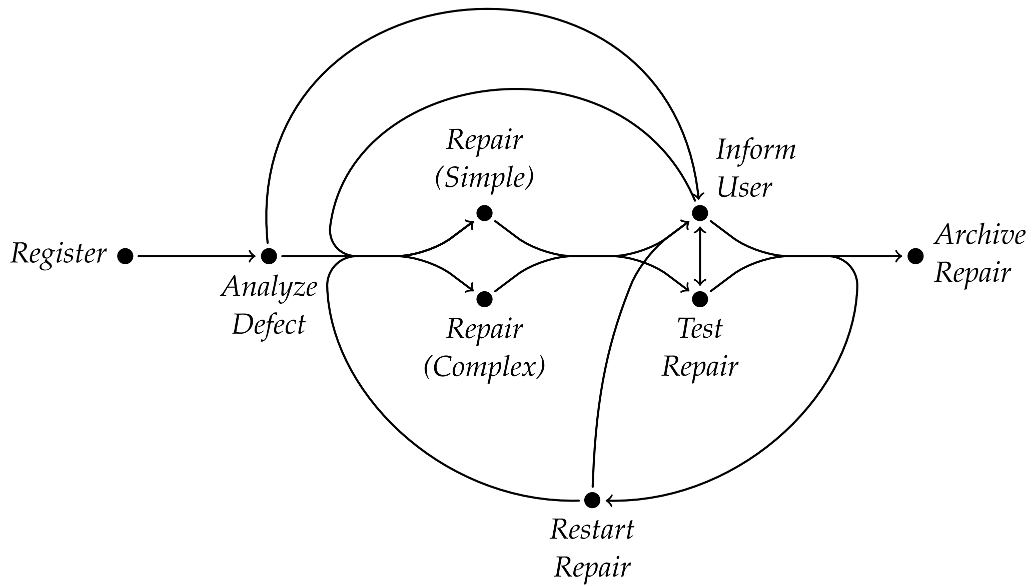


Figure 2.5: A confluent drawing for the repair shop data set. This figure was constructed manually using algorithm CONFLUENT-DRAWING.

Chapter 3

Strict power graphs

In this chapter we study power graphs in more detail. As we saw in the introduction, this is an approach to the problem of drawing dense graphs in a clear way, just like confluent drawings. However, power graphs use vertex bundling instead of edge bundling.

Power graph analysis was first discussed in 2008 by Royer *et al.* [18] for the analysis of biological protein networks. After that, Dwyer *et al.* refined this new concept. They performed a user study showing that inexperienced users are able to use power graphs to accurately answer shortest-path queries [6]. They also expanded on the technique by developing algorithms to compute power graphs [5].

The idea of power graph analysis is to draw the vertices in groups, and allow edges to attach to entire groups of vertices; see Figure 3.1. Hence, while edge bundling methods reduce the number of edges in the drawing by bundling them, vertex bundling methods achieve this by representing many edges in the original graph by one edge in the drawing.

Figure 3.1: The idea of power graph analysis: instead of grouping edges like with confluent drawings *(left)*, we group vertices *(right)*.

3.1 Definition

In this section, we give formal definitions for power graphs. We mostly follow the definitions by Dwyer *et al.* [6]. Note that power graphs can also be used for undirected graphs, just like confluent drawings. Again, we discuss only the directed variant here.

Definition 3.1 (Power graph configuration)**.** Let V be a set of vertices. A *power graph configuration*, or just *configuration*, on V is a set $M \subseteq 2^V$ of *modules*, where every module is a subset of the vertices, satisfying the following requirements: $\{v\} \in M$ for all $v \in V$, and for all modules $m, n \in M$: if $m \cap n \neq \emptyset$, then $m \subseteq n$ or $n \subseteq m$.

This means that the modules in a power graph configuration are not allowed to overlap partially: if modules overlap, one of them needs to completely contain the other. Hence, modules are ordered in a hierarchical, tree-like fashion. See Figure 3.2 for an example.

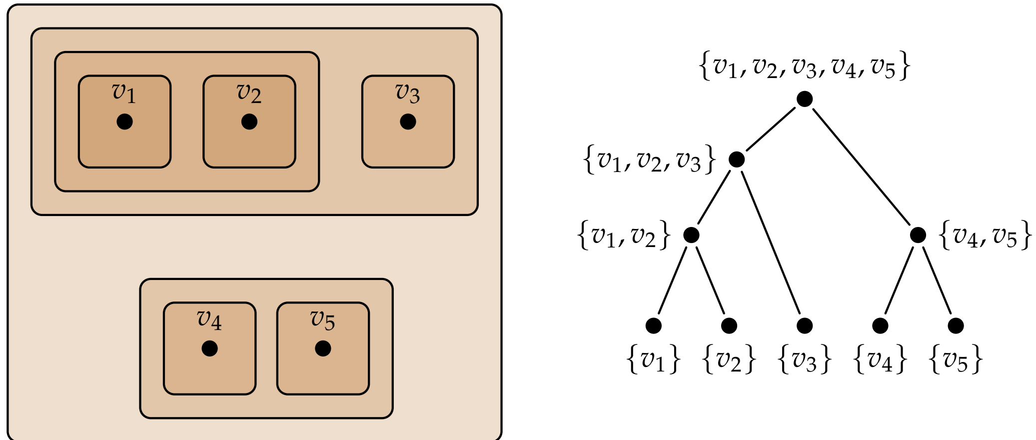


Figure 3.2: An example of a valid power graph configuration on five nodes, and its representation as a tree of modules.

Definition 3.2 (Power graph)**.** Let $G = (V, E)$ be a directed graph. A *power graph* on V is a pair $P = (M, R)$ where M is a configuration on V and $R \subseteq M \times M$ is a set of *power edges*. We say that P *represents* G if

- for every edge $(u, v) \in E$, there exists a power edge $(m, n) \in R$ with $u \in m$ and $v \in n$ ("every edge in G is represented by a power edge in P");
- for every power edge $(m, n) \in R$, for any $u \in m$ and $v \in n$, $(u, v) \in E$ ("every power edge in P represents a complete set of edges in G from all vertices in m to all vertices in n").

To avoid confusion with fuzzy power graphs later on, we also refer to power graphs as *strict power graphs*.

In Figure 3.3 a simple graph G is shown, along with three power graphs that represent G. It is clear that for any graph G, there are many power graphs representing G. Since our goal is to simplify the drawing, we are interested in obtaining power graphs with a low number of power edges.

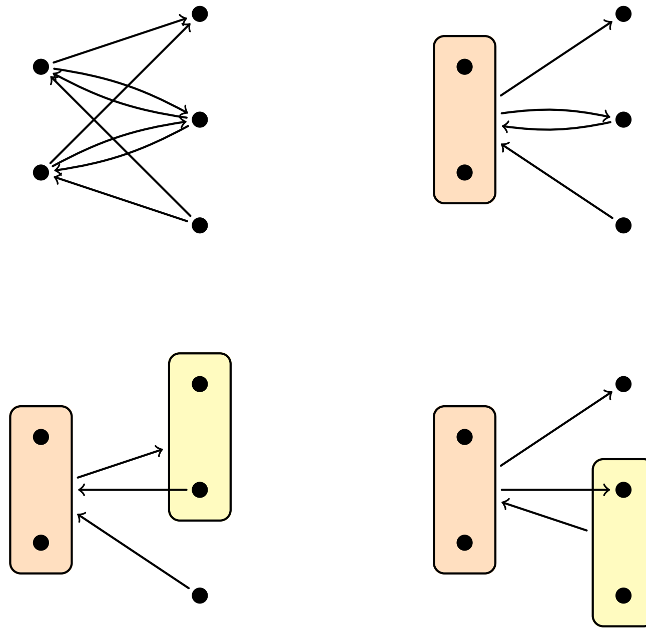


Figure 3.3: An example of a graph G *(top left)*, along with three power graphs representing G.

Note that in the power graph drawings in Figure 3.3, we left out the modules for individual vertices (just like in Figure 3.1). Instead, power edges arriving at such a single-vertex module are connected directly to the original vertex. In the remainder, we do the same for all drawings of power graphs, since it reduces clutter.

Finally, note that it is allowed for power edges to cross module boundaries. This happens, for example, in the lower two power graphs in Figure 3.3. This is essential to the definition of power graphs. A power graph without power edges that cross module boundaries is called a *modular decomposition* [6]. We will not discuss modular decompositions further here, since they allow for much less simplification than power graphs.

3.2 The greedy algorithm

Dwyer *et al.* [5] proposed a greedy algorithm to produce power graph configurations. This algorithm tries to minimize the number of power edges needed; however it is not optimal. Dwyer *et al.* present strong evidence that the problem of finding an optimal power graph configuration is *NP*-hard, although as far as we know a complete proof has not been found yet. Therefore, we focus on the greedy algorithm in this thesis.

In this section we look at the greedy algorithm in detail. Later, in Chapter 4, we will use a modified version of it to find fuzzy power graphs.

The greedy algorithm builds the tree of modules bottom-up. It starts with single-vertex modules for every vertex, that form the leaves in the tree. The algorithm then starts executing *merges*. For two top-level modules m and n, merging m and n means that a new module $m \cup n$ is introduced in the module tree. By performing a merge, any pair (m, v) and (n, v) of power edges can be replaced by a single power edge $(m \cup n, v)$, and similar for pairs (v, m) and (v, n); see Figure 3.4. The algorithm greedily picks the merge that eliminates the largest number of power edges. This is repeated until no merge step can reduce the number of power edges anymore.

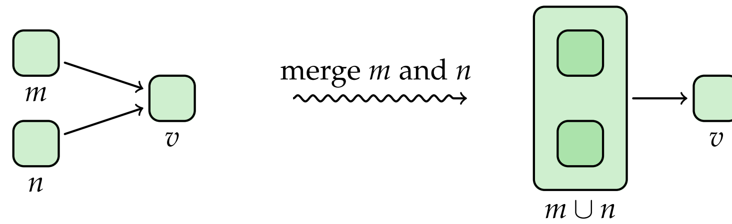


Figure 3.4: During a merge, we replace pairs of power edges (m, v) and (n, v) by a single power edge $(m \cup n, v)$.

To implement this algorithm, we need a way to determine how many power edges can be eliminated by merging m and n. We could just perform the merge and count the number of eliminated power edges. We can also just count the number of outgoing and incoming neighbours that m and n share:

$$\begin{aligned} \mathit{nedges}(m, n) &:= \text{the number of eliminated power edges when merging } m \text{ and } n \\ &= |N^+(m) \cap N^+(n)| + |N^-(m) \cap N^-(n)|, \end{aligned}$$

where $N^+(m)$ and $N^-(m)$ are the sets of outgoing and incoming neighbours of m, respectively.

Finally, we note that merging m and n might leave m or n without any associated power edges. Such modules do not serve any purpose while adding clutter to the drawing. Hence, we check for this situation after the merge step by removing m or n when they do not have any power edges attached to them. Alternatively, we can perform a post-processing step after the entire algorithm is ready.

The complete procedure is described in pseudocode in Figure 3.5. In Figure 3.6 an example execution of the algorithm is shown.

```
Algorithm POWER-GRAPH-GREEDY(G = (V, E))
 1  P ← a set of modules with a module {v} for every vertex v ∈ V
 2  R ← the set of power edges: {({v1}, {v2}) | (v1, v2) ∈ E}
 3  while improved:
 4     find the top-level modules m, n ∈ P with the largest nedges(m, n)
 5     P ← P ∪ {m ∪ n}
 6     for all v ∈ P with (m, v), (n, v) ∈ R:
 7        R ← R \ {(m, v), (n, v)} ∪ {(m ∪ n, v)}
 8     ...   ▶ repeat line 6–7 for incoming power edges
 9     if there are no more power edges in R incident to m or n:
10        remove that module from P
11  return P, R
```

Figure 3.5: The greedy algorithm proposed by Tim Dwyer *et al.* for generating a power graph decomposition.

Note that different implementations of this algorithm do not necessarily come up with the same power graph drawing. In line 4, there may be several pairs of modules which all have the same *nedges*. In such a case, the algorithm can pick any of those pairs, resulting in different power graphs.

3.3 Results

As we saw in Section 1.3, concurrency manifests itself in the directly-follows graph as a group of edges between the sets of activities that are carried out in parallel. In a power graph, a complete set of edges between two groups of vertices is really simple to represent: it just consists of two power edges (in both directions) between two sets of vertices. So, in the ideal case concurrency would be visualized like in Figure 3.7, which is a much clearer visualization than the node-link drawing in Figure 1.7.

To see whether this works in practice, in Figure 3.8 the result of running algorithm POWER-GRAPH-GREEDY on the directly-follows graph of the repair shop example (see Figure 1.6) is shown. The resulting drawing has 9 power edges. For comparison, the original directly-follows graph contains 19 edges.

We note that the choice between *Repair (Simple)* and *Repair (Complex)* is visible clearly in the drawing. However; the concurrency between *Inform User* and the other nodes is not really clear, just like in the confluent drawing in Section 2.4.

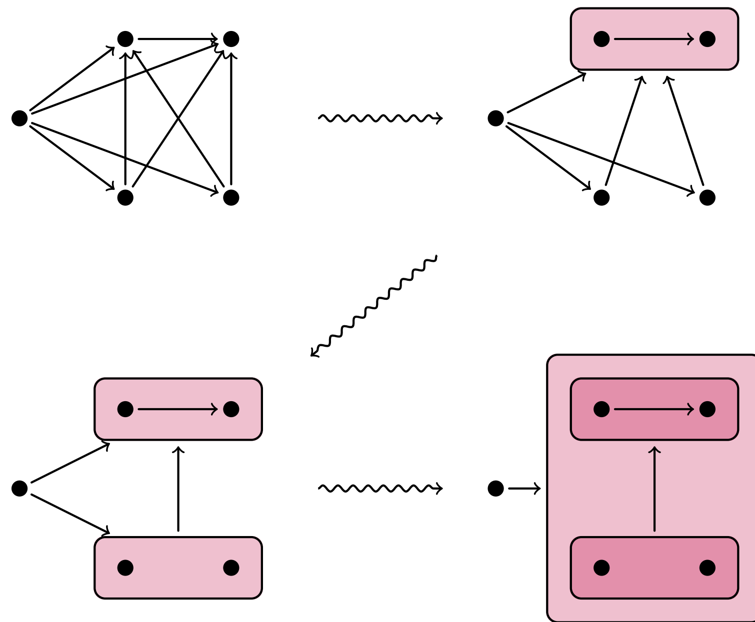


Figure 3.6: An example of executing algorithm POWER-GRAPH-GREEDY. The situation is shown after every iteration of the **while** loop.

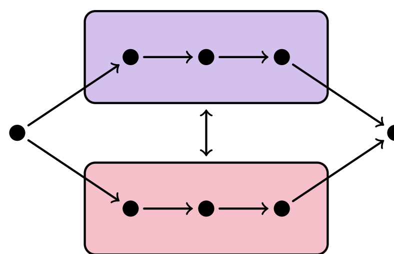


Figure 3.7: The directly-follows graph of a process containing concurrency visualized as a power graph drawing.

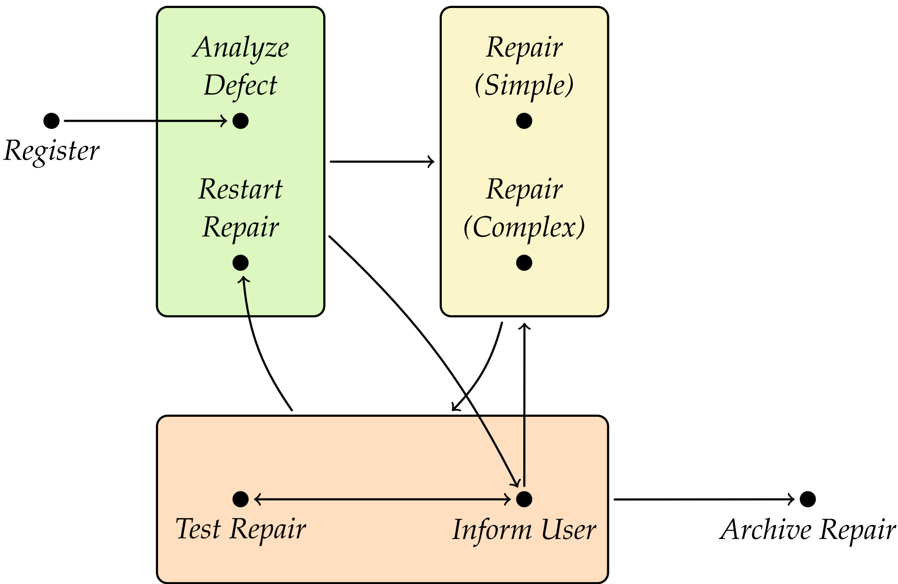


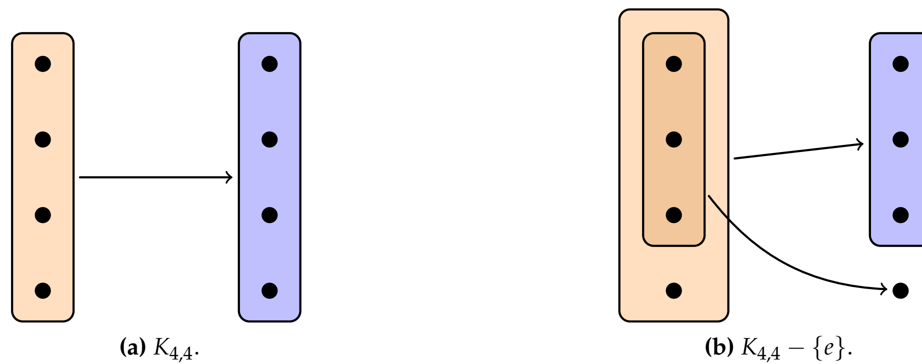
Figure 3.8: A strict power graph drawing for the repair shop example as generated by the algorithm POWER-GRAPH-GREEDY.

Chapter 4

Fuzzy power graphs

As we saw in the previous chapter, strict power graphs work well to make drawings of directly-follows graphs easier to read. However, for large logs, the resulting drawings are still very large and hard to read. Hence, we want to investigate whether it is possible to reduce the complexity of the drawing more than what strict power graphs are able to achieve.

It turns out that we can do this, by allowing our method to introduce ambiguity in our drawing. As an example, consider the bipartite graph $K_{4,4} - \{e\}$. In Figure 4.1b the result of POWER-GRAPH-GREEDY on this graph is shown; compare this to the drawing of $K_{4,4}$ in Figure 4.1a. By removing one edge from $K_{4,4}$, we need an additional edge and an additional module to represent the graph.



(a) $K_{4,4}$. **(b)** $K_{4,4} - \{e\}$.

Figure 4.1: Strict power graph representations of two directed bipartite graphs.

If we allow some ambiguity, it makes sense to represent $K_{4,4} - \{e\}$ by the drawing in Figure 4.1a. In that case, we are not able to see which of the 16 edges is not present,

but we save one edge and one module. In this chapter we introduce an extension to the concept of strict power graphs, based on this idea, called *fuzzy power graphs*, that allows ambiguity

4.1 Definition

Definition 4.1 (α-fuzzy power graph)**.** Let $G = (V, E)$ be a directed graph, and $\alpha \in [0, 1]$. An *α-fuzzy power graph* on V is a pair $P = (M, R)$ where M is a configuration on V and $R \subseteq M \times M$ is a set of *power edges*. We say that P *represents* G if

- for every edge $(u, v) \in E$, there exists a power edge $(m, n) \in R$ with $u \in m$ and $v \in n$ ("every edge in G is represented by a power edge in P");
- for every power edge $(m, n) \in R$, $|E \cap (m \times n)| \geq \alpha \cdot |m \times n|$ ("at least a fraction of α of the edges from the vertices in m to the vertices in n exists in E")

Note that the definition for fuzzy power graphs is almost identical to the definition for strict power graphs. The only distinction is in the second condition. With strict power graphs, we required that a power edge (m, n) indicates that all edges (u, v) with $u \in m$ and $v \in n$ exist in the original graph. Said differently, a power edge represents a complete bipartite subgraph in G. Now, with fuzzy power graphs, we require only that a power edge represents a partial bipartite subgraph. Such a partial bipartite subgraph needs to contain at least a fraction of α of the edges of the corresponding complete bipartite subgraph. Thus, the definition of 1-fuzzy power graphs is equivalent to the one for strict power graphs.

This definition solves the problem where $K_{4,4} - \{e\}$ is not drawn clearly: for $\alpha \leq 15/16$ the drawing in 4.1a is a valid α-fuzzy power graph representing $K_{4,4} - \{e\}$. While the disadvantage of fuzzy power graphs is that they are ambiguous, by setting α to values between 0 and 1, we can trade off the ambiguity and the simplicity of the drawing.

4.2 A naive algorithm

To obtain a simple algorithm that generates fuzzy power graphs, we take algorithm POWER-GRAPH-GREEDY from Section 3.2 as a starting point.

The basic idea is as follows. During a merge operation, instead of replacing only pairs of power edges (m, v) and (n, v) by a single power edge $(m \cup n, v)$ (as in Figure 3.4), we can sometimes also move a single power edge (m, v) to $(m \cup n, v)$, even when (n, v) does not exist (see Figure 4.2). Of course, we cannot always do this: we need to check if the new power edge satisfies the second condition of Definition 4.1.

To be able to perform this check, we need to know for every power edge e how many edges it represents in the original graph G. To do that we simply store these counts along with the power edges; we will denote by $r(e)$ the number of edges represented by

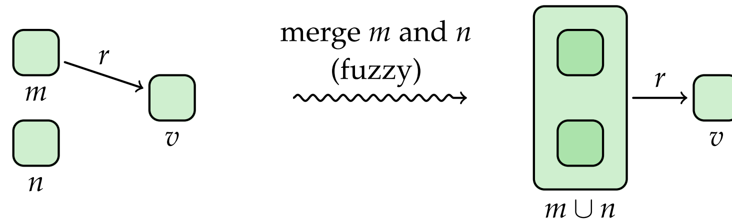


Figure 4.2: During a fuzzy merge, we can sometimes replace a single power edge (m, v) by the power edge $(m \cup n, v)$, even when (n, v) does not exist. Compare this to the step for strict power graphs (Figure 3.4).

a power edge e. Every time we replace a pair of power edges e_1, e_2 by a single power edge e, we take the sum of the counts of the old power edges: $r(e) = r(e_1) + r(e_2)$.

Now, to perform the check, we compute which fraction of all possible edges is present between the new module $m \cup n$ and v. The total number of possible edges is $|(m \cup n) \times v| = |m \cup n| \cdot |v|$. Thus, we are allowed to execute the replacement step iff

$$\frac{r((m, v))}{|m \cup n| \cdot |v|} \geq \alpha.$$

Of course, instead of the power edge (m, v) we can also replace (n, v) in the same way. Also the same can be done for power edges that run in the other direction (that is, from v to m or n). This results in the pseudocode for the algorithm that is given in Figure 4.3; the additions to Power-Graph-Greedy are shown in red.

4.3 Problems with this approach

Although the algorithm Fuzzy-Power-Graph-Naive always produces valid α-fuzzy power graphs, it turns out that it has major problems finding an optimal one, or even one that comes close to being optimal.

For example, consider again the graph $K_{4,4} - \{e\}$. As we saw in Section 4.1, for $\alpha \leq 15/16$ we would expect to obtain the power graph from Figure 4.1a. Unfortunately, Fuzzy-Power-Graph-Naive will come up with this drawing only for $\alpha \leq 3/4$: for $\alpha > 3/4$ it returns the drawing from Figure 4.1b. To see why this happens, we take a more detailed look what the algorithm is doing in this case.

First, the algorithm merges $\{a_1\}$ and $\{a_2\}$ to produce $\{a_1, a_2\}$, and then it merges $\{a_1, a_2\}$ and $\{a_3\}$ to produce $\{a_1, a_2, a_3\}$. This is exactly the same as in the non-fuzzy case.

Now, the algorithm will start merging $m := \{a_1, a_2, a_3\}$ and $n := \{a_4\}$; see Figure 4.4a. Of course, the pairs of power edges from m and n to $\{b_1\}$, $\{b_2\}$ and $\{b_3\}$ are all replaced by single power edges from $m \cup n$. So, we get the situation depicted in Figure 4.4b.

```
Algorithm FUZZY-POWER-GRAPH-NAIVE(G = (V, E), α)
1   P ← a set of modules with a module {v} for every vertex v ∈ V
2   R ← the set of weighted power edges: {({v1}, {v2}, 1) | (v1, v2) ∈ E}
3   while improved:
4     find the top-level modules m, n ∈ P with the largest nedges(m, n)
5     P ← P ∪ {m ∪ n}
6     for all v ∈ P with (m, v, r1), (n, v, r2) ∈ R:
7       R ← R \ {(m, v, r1), (n, v, r2)} ∪ {(m ∪ n, v, r1 + r2)}
8     ...   ▶ repeat line 6–7 for incoming power edges
9     for all v ∈ P with (m, v, r):
10      if r / (|m ∪ n| · |v|) ≥ α:
11        R ← R \ {(m, v, r)} ∪ {(m ∪ n, v, r)}
12    ...   ▶ repeat line 9–11 for incoming power edges and with m replaced by n
13    if there are no more power edges in R incident to m or n:
14      remove that module from P
15  return P, R
```

Figure 4.3: A simple modification to algorithm POWER-GRAPH-GREEDY to generate α-fuzzy power graphs. The additions to POWER-GRAPH-GREEDY are shown in red.

If $\alpha \leq 3/4$, the power edge $(m, \{b_4\})$ will be replaced by the power edge $(m \cup n, \{b_4\})$. When we execute the algorithm further, $\{b_1, b_2, b_3, b_4\}$ will be merged into one module, and we get the power graph from Figure 4.1a.

If however $\alpha > 3/4$, we cannot replace this power edge. Since the algorithm will try to merge top-level modules only, and m is not a top-level module anymore, the algorithm will never be able to replace $(m, \{b_4\})$ by $(m \cup n, \{b_4\})$ anymore. This results in the drawing in Figure 4.1b.

The main problem here is that the algorithm, at the moment it merges m and n, does not know yet that $\{b_1, b_2, b_3, b_4\}$ will all be merged in later steps of the algorithm. This limitation is inherent in using a greedy algorithm. We can however fix the problem by accepting the limitation, but doing the merge step in a different way.

4.4 An improved algorithm

In this section, we present an improved variant of FUZZY-POWER-GRAPH-NAIVE that overcomes the problem mentioned in the previous section. The basic structure of the algorithm stays the same; we only do the merge step in a different way.

Consider the situation where we are merging two top-level modules m and n. In the naive algorithm, we would look at every other top-level module v to see if both m or

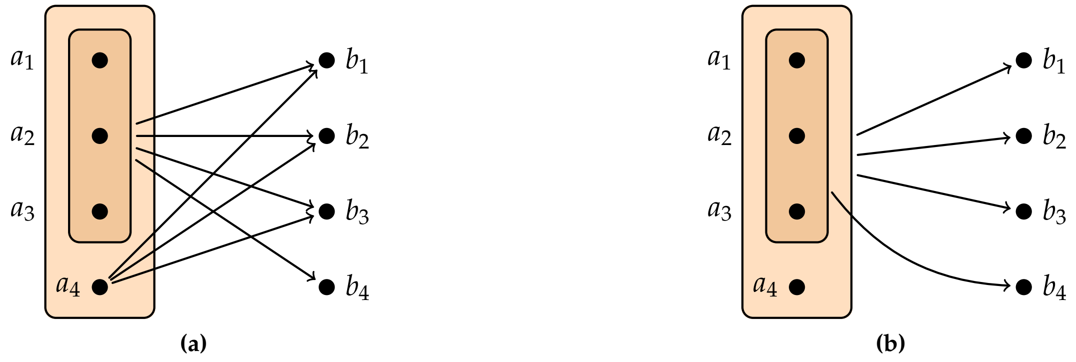


Figure 4.4: Two situations during the execution of FUZZY-POWER-GRAPH-NAIVE on the graph $K_{4,4} - \{e\}$.

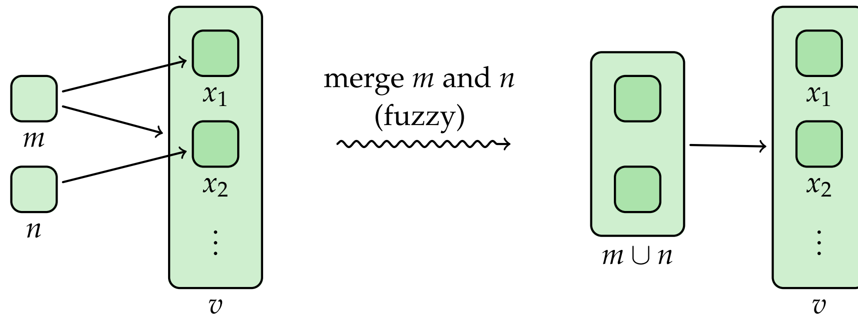


Figure 4.5: In the improved algorithm for finding fuzzy power graphs, we also consider power edges from m or n to submodules of v. We can sometimes replace all of those power edges by one power edge $(m \cup n, v)$. Compare this to Figures 3.4 and 4.2.

n had power edges towards v, and if so, replace those power edges by a single power edge from $m \cup n$ (see again Figure 4.4). Hence, we did not look at power edges towards submodules of v.

However, those power edges represent edges from G that are also a part of the same bipartite subgraph between $m \cup n$ and v. Therefore it is beneficial to look at all power edges to submodules of v, and – if possible – replace them by a single power edge $(m \cup n, v)$ (see Figure 4.5).

To determine whether we are allowed to perform this replacement, we compute the sum N of the weight of all power edges from m or n to any module $x \subseteq v$:

$$N := \sum \{r(e) \mid e \in R \text{ from } m \text{ or } n \text{ to } x \subseteq v\}.$$

Now if

$$\frac{N}{|m \cup n| \cdot |v|} \geq \alpha,$$

we can replace all of the power edges by a single power edge $(m \cup n, v)$ with weight N.

Algorithm FUZZY-POWER-GRAPH- $(G = (V, E), \alpha)$

1 $P \leftarrow$ a set of modules with a module $\{v\}$ for every vertex $v \in V$
2 $R \leftarrow$ the set of weighted power edges: $\{(\{v_1\}, \{v_2\}, 1) \mid (v_1, v_2) \in E\}$
3 **while** improved:
4   **for all** pairs of top-level modules $m, n \in P$:
5       $P_{m,n}, R_{m,n} \leftarrow$ MERGE(α, P, R, m, n)
6   take m, n with the smallest $|R_{m,n}|$
7    $P \leftarrow P_{m,n}$, $R \leftarrow R_{m,n}$
8   **if** there are no more power edges in R incident to m or n:
9   remove that module from P
10 **return** P, R

Subroutine MERGE(α, P, R, m, n)

1 $P \leftarrow P \cup \{m \cup n\}$
2 **for all** $v \in P$:
3    $e \leftarrow$ a list of all power edges (m, x, r) or (n, x, r) in R where $x \subseteq v$
4    $N \leftarrow \sum_{(a,x,r) \in e} r$
5   **if** $\frac{N}{|m \cup n| \cdot |v|} \geq \alpha$:
6       $R \leftarrow R \setminus e \cup \{(m \cup n, x, N)\}$
7   ... ▶ *repeat line 3–6 for incoming power edges*
8 **return** P, R

Figure 4.6: A variant on algorithm FUZZY-POWER-GRAPH-NAIVE that does not suffer from the problem described in Section 4.3.

The full algorithm is given as pseudocode in Figure 4.6. For clarity we moved the merge step into a separate subroutine. Also, since there is no simple expression for *nedges* anymore, we just execute all merges and take the one with the smallest number of power edges.

We now look again at the problem discussed in Section 4.3 and see how FUZZY-POWER-GRAPH-GREEDY solves this. For $\alpha = 15/16$, the algorithm still arrives at the situation depicted in Figure 4.4b, and it is still unable to replace the power edge $(m, \{b_4\})$ by $(m \cup n, \{b_4\})$. However, after some more merge steps the algorithm merges $m := \{b_1, b_2, b_3\}$ and $n := \{b_4\}$ (see Figure 4.7). The list e now contains both power edges in the graph, which have a total weight of 15. Since $|m \cup n| \cdot |v| = 16$, we are now allowed to replace both power edges by a single edge $(v, m \cup n)$, resulting in the power graph depicted in Figure 4.1a.

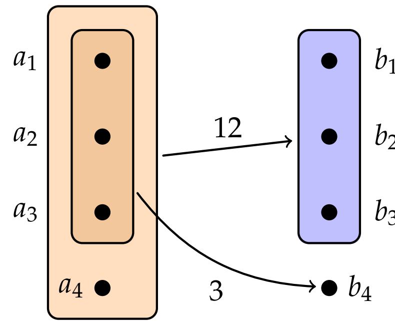


Figure 4.7: The situation before merging $m := \{b_1, b_2, b_3\}$ and $n := \{b_4\}$ during algorithm FUZZY-POWER-GRAPH-GREEDY on the graph $K_{4,4} - \{e\}$.

4.5 Results

In this section we apply algorithm FUZZY-POWER-GRAPH-GREEDY to the repair shop data set to see whether it works in practice. In Figure 4.8 we show the result of running the algorithm for $\alpha = 0.6$ on the directly-follows graph of the repair shop example.

Due to the complexity of this algorithm we did not execute the algorithm by hand, since that would be very error-prone. Instead, we used the implementation discussed in Chapter 5. We used a version of the implementation that outputs a debug message for every step it takes. Using those messages, we verified manually that the algorithm was executed correctly according to the pseudocode in Figure 4.6.

We see that the general structure of the drawing is quite similar to the strict power graph drawing in Figure 3.8, but there are more nested modules and less power edges. The strict power graph drawing contains 9 power edges, while the fuzzy power graph contains 7 power edges.

However, as expected, also a lot of structure is lost. In the drawing, we do not see anymore that there is a choice between *Repair (Simple)* and *Repair (Complex)*. Instead, if we interpret this drawing as a strict power graph, there seems to be a choice between four activities: *Repair (Simple)*, *Repair (Complex)*, *Restart Repair* and *Archive Repair*.

There is an interesting edge from *Restart Repair* to the module containing *Repair (Simple)*, *Repair (Complex)* and *Restart Repair* itself. If we assume that the original graph does not have any self-loops, such power edges to parent modules will never happen with strict power graphs. However, in a fuzzy power graph they do happen. In this case, the directly-follows graph contains edges from *Restart Repair* to both *Repair (Simple)* and *Repair (Complex)*. Hence, with $\alpha = 0.6 < 2/3$ we are allowed to use a single power edge from *Restart Repair* to its parent module to represent those two edges in the original graph.

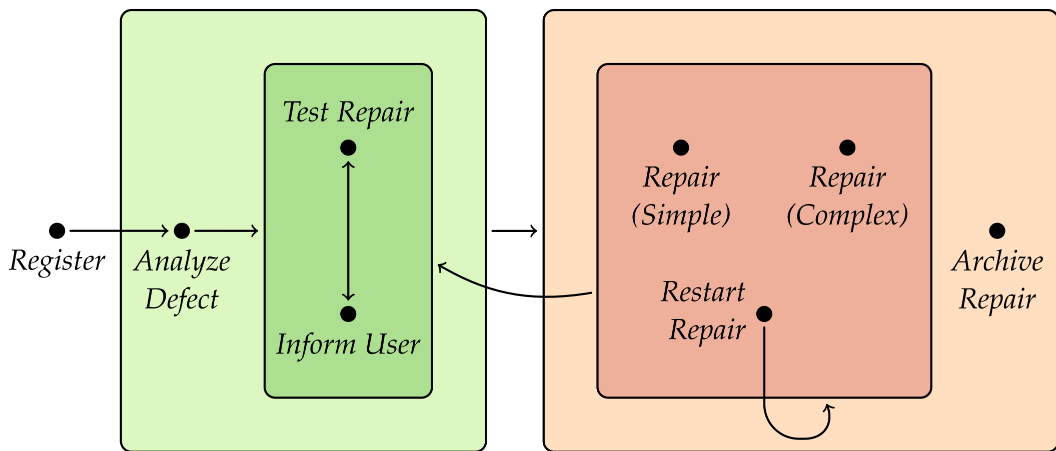


Figure 4.8: A 0.6-fuzzy power graph drawing for the repair shop example as generated by the algorithm FUZZY-POWER-GRAPH-GREEDY. Compare this to the strict power graph drawing in Figure 3.8.

Chapter 5

Implementation

In this chapter, we discuss our proof-of-concept implementation of the algorithms for strict and fuzzy power graphs. The proof of concept is available on `http://wimiso.nl/research/logvis/demo`.

5.1 WebCola

To experiment with strict and fuzzy power graphs, we used the *WebCola* library[1] by Dwyer *et al*. This is a library for layouting graphs with constraints on the layout. Supported constraints are *alignment constraints* (vertices need to have the same x- or y-coordinate) and *separation constraints* (the distance between vertices on the x- or y-axis needs to be above some minimum value).

WebCola is written in TypeScript, a language that compiles to JavaScript, so it runs inside the user's web browser. Since OpenXES, the library that reads log files (see Section 6.1), is written in Java and we already wrote Java code to call this library, we added functionality to export a log file to a format that WebCola is able to read.

WebCola already contained an implementation of the greedy algorithm from Section 3.2. The library also contained a *grid router*, which is a function to layout the edges on a rectangular grid, but this was experimental.

5.2 Bug fixes

We needed to make a few bug fixes to WebCola. They have all been incorporated in the library. In this section we give descriptions of those bug fixes.

[1]WebCola can be downloaded from `http://marvl.infotech.monash.edu/webcola`.

5.2.1 Browser incompatibility

In the greedy algorithm, the JavaScript `sort()` function was used to find the merge with the largest *nedges*. However, this sort function is not guaranteed by the JavaScript specification to be stable: some browsers implement a stable sorting algorithm, others do not. Hence, the output of the greedy algorithm varied between browsers. For the sake of reproducibility, we made the sort stable, by modifying the comparison function to compare based on the original ordering if *nedges* is identical.

5.2.2 Edge reversal bug

The grid router reverses edges, so that bundled edges all share the same direction. However, it forgot to reverse the edges back again after the algorithm, resulting in incorrect drawings like in Figure 5.1. We fixed this by adding code to reverse the edges back.

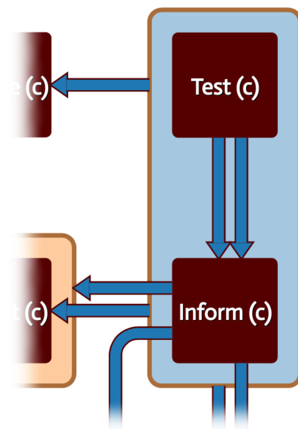


Figure 5.1: A double edge between two modules. This is an effect of the edge reversal bug: actually one of the arrows would have to be reversed to obtain a correct drawing.

5.2.3 Crash in the grid router

This issue is a bit more complicated; we give only a high-level overview here. When routing edges over the grid, edges will often share the same grid line. Since a drawing with overlapping edges would not be clear, the grid router pushes the edges apart a bit (this is already visible in Figure 5.1). However, when doing this it is important to pick a good ordering of edges within a single bundle, since otherwise we can get a lot of unnecessary edge crossings.

Therefore WebCola's grid router contains a subroutine `orderEdges()` that takes care of determining a smart ordering of two edges in a single bundle. The heart of the subroutine is a case distinction:

1. if the two edges are completely identical, it takes an arbitrary ordering;

2. otherwise, it looks at one of the endpoints where the two paths diverge, and bases the ordering on that.

The problem was that there is a third case that does not normally occur when rendering graphs, but does happen with power graphs: the case where one of the edges is entirely covered by another edge (see Figure 5.2). In this case, the algorithm would decide that the edges are not identical, so it would go to the second case, and try to find an endpoint where the paths diverge. Obviously, the paths do not diverge on either endpoint, which eventually resulted in an out-of-bounds array access, causing a crash.

The fix for this was surprisingly simple. Note that in Figure 5.2, the ordering of the two edges does not matter, just like in the first case of the case distinction. Hence, we modified the guard on the first case to include all possible cases where the order does not matter. This indeed solved the problem.

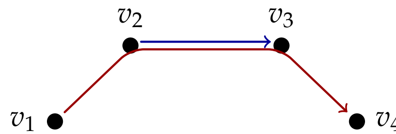


Figure 5.2: Sketch of a situation that would crash the grid router. When drawing normal graphs, this situation would never happen, since the grid router avoids vertices when routing edges, so the edge from v_1 to v_4 could never visit other vertices on its path. However, in power graphs, this situation can happen when v_1 and v_4 are submodules of v_2 and v_3, respectively.

5.3 Fuzzy power graph algorithm

We implemented the FUZZY-POWER-GRAPH-GREEDY algorithm from Section 4.4 in WebCola. To do this, we extended the function `powerGraphGroups()` to accept an additional, optional parameter α. If $\alpha = 1$, or if the parameter is omitted, the original implementation of POWER-GRAPH-GREEDY is run. If $\alpha < 1$, the new implementation is run.

While the implementation works, it is not completely stable yet, and could use further improvements. Therefore we made no attempt to get this change included in the WebCola project.

5.4 Results for the repair shop example

To test whether the implementation works correctly, we used the example data set from the telephone repair shop. In this section we present screenshots of the results.

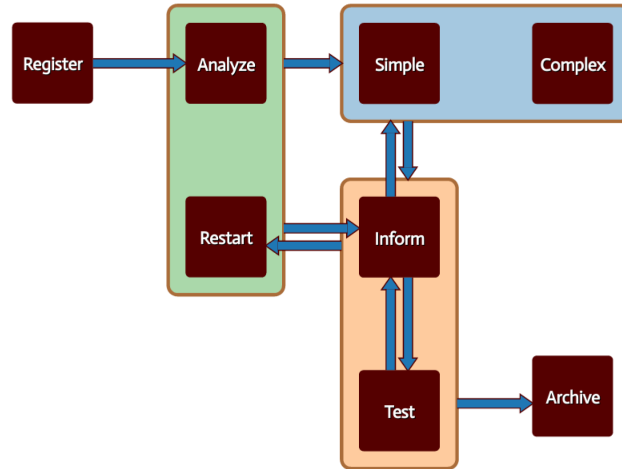
To make the drawings of the repair shop example more readable, we shortened the activity names: *Simple* corresponds to *Repair (Simple)*, *Analyze* corresponds to *Analyze Defect*, and so on. This was necessary because the grid router in WebCola has problems layouting non-square vertices, so we had to fit the labels in square vertices. Hence, long labels like *Analyze Defect* would cause the vertices to become very large.

Strict power graphs First, we ran the strict power graph algorithm on the directly-follows graph. Figure 5.3 presents two strict power graph drawings for this data set.

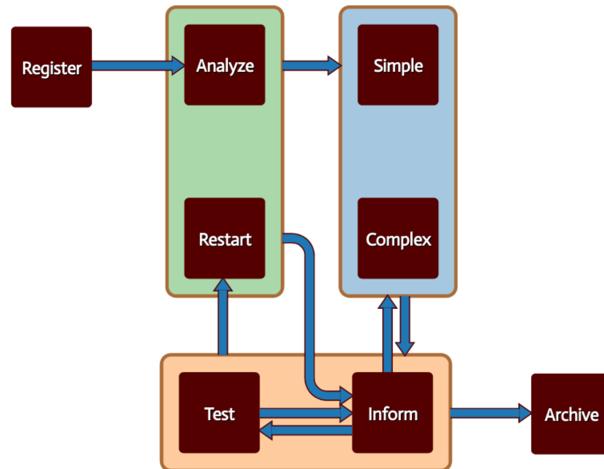
Figure 5.3a presents the layout that is generated initially by WebCola. WebCola offers the possibility to make vertices in a graph draggable, so that the layout can be improved by hand. To demonstrate this capability, Figure 5.3b presents the layout after dragging the vertices so that the layout corresponds to the drawing in Figure 3.8. We see that the constructed power graph is identical to the power graph that was constructed by hand.

Fuzzy power graphs Figure 5.4 presents the result of running the implementation for fuzzy power graphs with $\alpha = 0.6$. The output is identical to the drawing in Figure 4.8. This is logical, because we used the implementation to generate the power graph shown in the figure. However, as described in Section 4.5, we manually verified the results.

The layout was manually dragged again to correspond to Figure 4.8. An interesting point is the edge from *Restart* to its parent module. WebCola draws this edge as a very short arrow directly to the module boundary. In our manual drawing we used an arrow that leaves the module and reaches the module boundary from the outside.



(a) The original layout.



(b) After manual layouting.

Figure 5.3: The repairshop example visualized as a strict power graph.

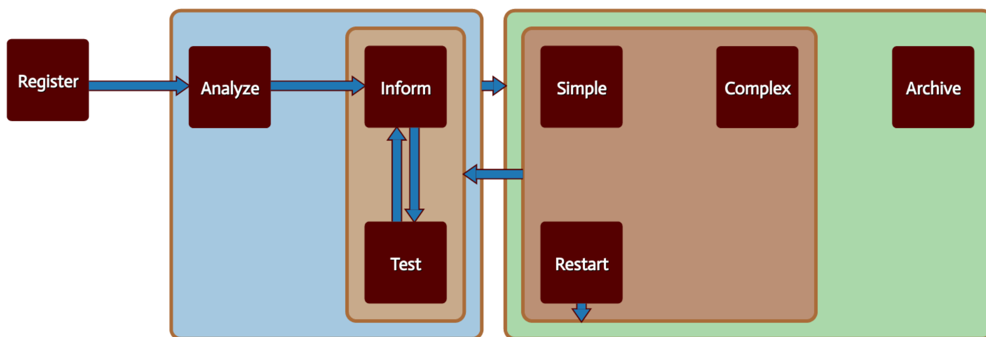


Figure 5.4: The repairshop example visualized as a 0.6-fuzzy power graph.

Chapter 6

Evaluation

In this chapter we evaluate how well the power graph visualizations work in practice. We provide screenshots of the strict and fuzzy power graphs generated by the implementation on several example data sets.

6.1 Data sets

In this section we start by describing the data sets used.

To store log data, the process mining community developed a standardized file format named *XES* (*Extensible Event Stream*) [20], that is also used by ProM. The file format is based on XML and is able to store a lot of information about events. For example, the log file shown in Figure 1.2 is in the XES format. All example data sets are stored in XES files, so we need a way to read them. Luckily, there is an existing, open-source library called *OpenXES*[1] to read XES files. We wrote a small Java program that uses this library to read the files and export a representation of the directly-follows graph that we can use as input for our visualizations.

6.1.1 Repair shop

First of all, we use the data set from a telephone repair shop, that was already used as the running example in this thesis. There are 1104 traces, with 11 855 events in total. As we saw already in Chapter 1, the data set contains examples of both choice and concurrency: there is a choice between *Repair (Simple)* and *Repair (Complex)*, and *Inform User* is performed in concurrency with *Repair (Simple)*, *Repair (Complex)*, *Test Repair* and *Restart Repair*.

[1]OpenXES can be downloaded from `http://www.xes-standard.org/openxes/start`.

In the original log file, some activities (*Analyze Defect*, *Repair (Simple)*, *Repair (Complex)* and *Test Repair*) have both a *start* and a *complete* variant. This is done to model activities that take a long time. Those sub-activities are called *lifecycle transitions*. For example, a trace from the log file, with lifecycle transitions, could be

$$[\textit{Register},\ \textit{Analyze Defect/start},\ \textit{Analyze Defect/complete},\ \textit{Repair (Simple)/start},\ \textit{Repair (Simple)/complete},\ \textit{Inform User},\ \textit{Test Repair/start},\ \textit{Test Repair/complete},\ \textit{Archive Repair}].$$

In the examples in this thesis we do not consider those *start* and *complete* variants as separate activities. Instead, in a preprocessing step, we merged all activities with the same name and distinct lifecycle transitions, and removed the resulting self-loops. This removal of self-loops is necessary because, for example, the sequence [*Test Repair/start*, *Test Repair/complete*] would create a self-loop on *Test Repair*, that does not actually occur in the process (*Test Repair* was not executed twice).

6.1.2 Generated data sets

We also use a series of data sets with 12, 22, 32 and 42 activities, that we call *a12*, *a22*, *a32* and *a42*, respectively. Those data sets are created by Măruşter *et al.* [17] and do not contain real-world data; instead they have been generated to evaluate process mining algorithms.

The logs have been generated by drawing Petri nets by hand; this has been done on purpose to contain many examples of concurrency. After that, traces were generated that conform to the Petri net. The Petri net of the *a12* log is given in Figure 6.1.

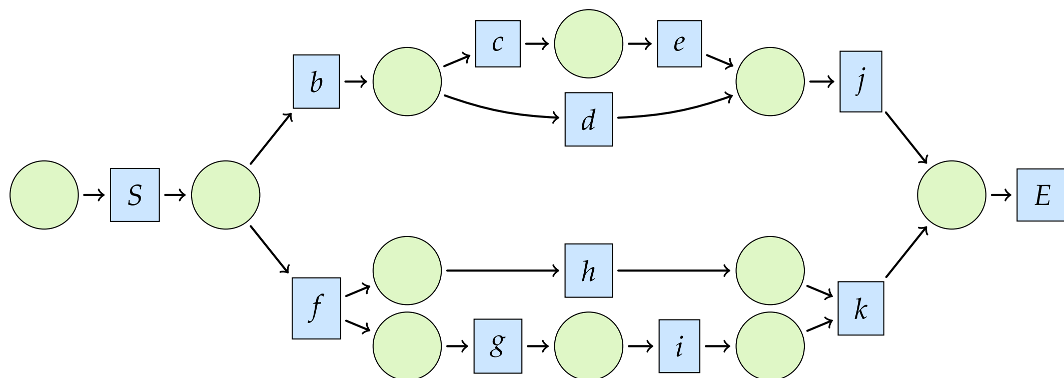


Figure 6.1: A drawing of the Petri net used to generate the *a12* data set. This is a modified version of Figure 1 in the paper by Măruşter *et al.* [17].

We see that, according to the Petri net, the process starts by doing S, and then there is a choice between two possibilities. In the first possibility, we do either b and then we can

choose between doing either c and e, or d. After that we do j and finish. In the second possibility, we get two concurrent processes: one performs g and i; the other performs just h. After that we do k and finish.

The method of generating the logs is explained in more detail in the paper by Măruşter *et al.* [17, § 3]. From this paper, we used the logs with 1000 lines, no noise and no imbalance of execution properties.

6.1.3 Financial log

Finally, to evaluate how well the algorithms work for very large event logs, we use the event log from the 2012 Business Processing Intelligence Challenge[2]. This event log contains data from a financial institution, and consists of 13 087 traces containing 262 200 events in total.

6.2 Generated data sets

First, we look at the generated data set *a12*. In Figure 6.2, the result as generated by our implementation is depicted. The drawing is very readable; it contains only 13 power edges. The drawing clearly shows the choice between the two possibilities, and the choice between $[c, e]$ and d inside the first possibility. The concurrency between $[g, i]$ and h is less visible in the drawing, however.

Since the strict power graph drawing is rather simple already, we do not expect that using fuzzy power graphs would really help in this case. To see if this is indeed the case, a 0.5-fuzzy power graph of the data set is given in Figure 6.3. The drawing contains 12 power edges, one less than the strict power graph. Note that the drawing contains several edges between vertices and their parent modules, just like the drawing of the repair shop example (see Section 4.5 and 5.4).

We then tried the implementation on the larger data sets *a22*, *a32* and *a42*. Screenshots of the generated strict and fuzzy power graphs can be found in Appendix A.1.

For *a22*, the generated strict power graph is still reasonably readable; we can see quite well what is going on in the process. The 0.5-fuzzy power graph has a lot less edges than the strict power graph, but it also loses a lot of information about the process.

For *a32*, the strict power graph has so many edges that it is quite unreadable. In the 0.5-fuzzy power graph we notice something strange: the module $\{b, n, h9, i, f, k10\}$ overlaps with the module next to it (see Figure A.4). It seems like $h9$, i, f and $k10$ are actually in two modules, which would of course be totally wrong.

Luckily, in fact, that is not what is going on. The algorithm works fine; those vertices belong only to the module $\{b, n, h9, i, f, k10\}$ in the underlying data structure. The problem is the way WebCola layouts power graphs. The layout algorithm uses

[2] More information about this challenge and the corresponding data set can be found on `http://www.win.tue.nl/bpi/2012/challenge`.

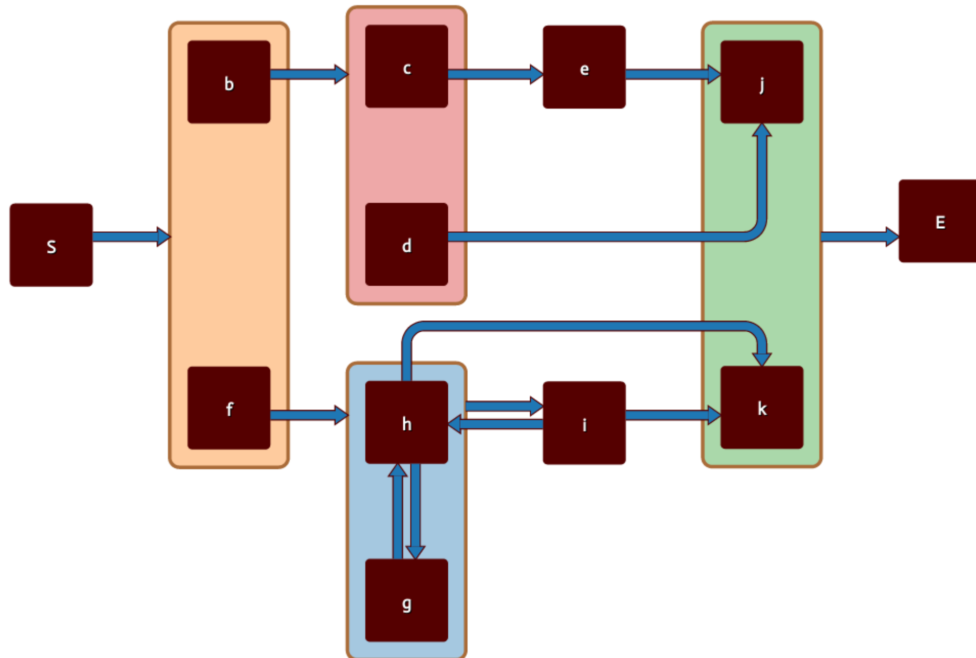


Figure 6.2: A strict power graph drawing of the *a12* data set, as generated by our implementation.

constraints to enforce that modules do not overlap, but the constraints do not take into account the padding between the contents of a module and its boundary. Usually this is not a problem since the padding is only small compared to the minimum distance enforced between modules.

However, for deeply nested module hierarchies this creates a problem, since there are a lot of paddings next to each other, so that modules can overlap each other with their padding areas. Figure A.4 presents an extreme case of this problem. This issue could probably be fixed by defining the constraints in a different way. However, since this is only a proof-of-concept implementation, we did not pursue this further.

For *a42*, the strict power graph is a total mess due to the large amount of edges. (In particular, the two strange sloped edges are caused by a bug in the grid router.) So, let us direct our attention to the 0.5-fuzzy power graph. Although this drawing is still large, it has a lot less edges and is actually quite readable. On the other hand, it seems that we cannot get a lot of information about the process from this drawing anymore. Hence, it makes sense to try increasing the α-value here. Therefore we tried generating a 0.8-fuzzy power graph (see Figure A.7). This drawing seems to retain much more of the structure of the original log, while still removing a lot of edges compared to the strict power graph.

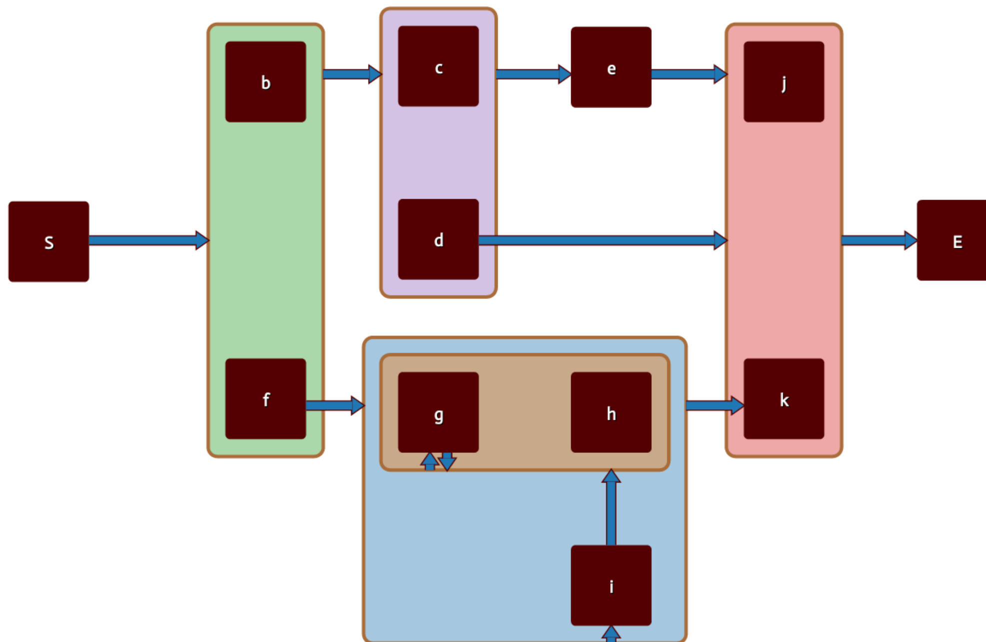


Figure 6.3: A 0.5-fuzzy power graph drawing of the *a12* data set, as generated by our implementation.

6.3 The financial log

Finally, we looked at the financial event log described in Section 6.1. We tried running the algorithm for strict power graphs on the directly-follows graph of this data set. This results in a huge drawing that is not clear at all. While using the algorithm for fuzzy power graphs helped a bit to make the graph more manageable, even at $\alpha = 0.5$ the number of edges is very large. We conclude that our visualization simply does not work on such enormous log files.

Several screenshots of the generated strict and fuzzy power graphs can be found in Appendix A.2.

Chapter 7

Conclusion

In this thesis, we described two ways to visualize event logs using the directly-follows graph. Firstly, we studied confluent drawings; secondly we looked at power graph analysis. Of those methods, power graph analysis seems to be the most promising, since confluent drawings result in a rather cluttered drawing.

We introduced a modification of power graph analysis, called fuzzy power graphs, to reduce the complexity of drawings, at the cost of losing some information about the original graph. We developed an algorithm to find fuzzy power graphs, based on an existing algorithm for power graph analysis.

Our approach also has some limitations. All methods we studied are not able to visualize edge weights. For visualizing directly-follows graphs, this means that it does not matter whether some activity occurs in one single trace or in a hundred traces. Of course, this throws away a lot of useful information. Furthermore, because we currently use a greedy algorithm, the behaviour is a bit unpredictable.

We implemented the fuzzy power graph algorithm in a proof of concept, together with the strict power graph algorithm that was already present in WebCola, and evaluated the results. We found out that strict power graphs generally work well for small log files, but fail to result in a clear drawing for large logs. Fuzzy power graphs are able to reduce the number of power edges in the drawing, also for larger log files. Unfortunately, a rather small α is needed to obtain a significant reduction in the number of edges, so we need to sacrifice a lot of accuracy to get useful results.

Future work While the method using fuzzy power graphs is promising, there is still a lot of work to do before it becomes useful for analyzing real-world log files.

First of all, we need to be able to visualize edge weights, for example by extending the concept of power graphs with power edge weights. Also, it seems to be worthwhile to try to improve the algorithm used for finding fuzzy power graphs. The greedy

algorithm that we use is not accurate enough, and sometimes comes up with very strange drawings. It would be interesting to look at other algorithms, for example beam search (as proposed by Dwyer *et al.* for strict power graphs [5]).

Alternatively, maybe the definition of α-fuzzy power graphs could be improved. For example, with $\alpha = 1/2$, the algorithm allows for any edge (a_1, a_2) in the original graph to be replaced by a power edge $(\{a_1, b\}, a_2)$, for any arbitrary activity b, also if the edge (b, a_2) does not exist. This seems undesirable, since there is no evidence at all that b has anything to do with a_1, which would justify such a grouping. Hence, a different definition that prohibits groupings like these may be better suited to visualizing event logs.

Acknowledgments I would like to thank Tim Dwyer, the inventor of the greedy algorithm for power graphs and the main developer of WebCola. He helped a lot in troubleshooting WebCola problems, implemented vertex dragging for power graphs, and gave very useful advice about which direction to take.

Furthermore I want to thank Eric Verbeek, who, as my tutor, gave the necessary input from the process mining side. Without him, I would not have known what is needed in a good log visualization. Also, thanks to Eric for providing useful feedback in general.

Thanks to Thom Castermans for the useful discussions we had, and for helping with the layout of the cover page.

Last but not least, thanks go to Bettina Speckmann, who was my supervisor during this project. She managed to understand my vague ideas during meetings and helped me to make them concrete. Also, she steered the project in the right direction, for which I am very grateful.

References

[1] Wil van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[2] Wil van der Aalst. *Process mining: discovery, conformance and enhancement of business processes*. Springer, 2011.

[3] Wil van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: discovering process models from event logs. *Knowledge and Data Engineering, IEEE Transactions on*, 16(9):1128–1142, 2004.

[4] Matthew Dickerson, David Eppstein, Michael Goodrich, and Jeremy Yu Meng. Confluent drawings: visualizing non-planar diagrams in a planar way. In *Graph Drawing*, pages 1–12. Springer, 2004.

[5] Tim Dwyer, Christopher Mears, Kerri Morgan, Todd Niven, Kim Marriott, and Mark Wallace. Improved optimal and approximate power graph compression for clearer visualisation of dense graphs. In *Pacific Visualization Symposium (PacificVis)*, pages 105–112. IEEE, 2014.

[6] Tim Dwyer, Nathalie Henry Riche, Kim Marriott, and Christopher Mears. Edge compression techniques for visualization of dense directed graphs. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2596–2605, 2013.

[7] David Eppstein, Michael Goodrich, and Jeremy Yu Meng. Delta-confluent drawings. In *Graph Drawing*, pages 165–176. Springer, 2006.

[8] David Eppstein, Michael Goodrich, and Jeremy Yu Meng. Confluent layered drawings. *Algorithmica*, 47(4):439–452, 2007.

[9] David Eppstein, Danny Holten, Maarten Löffler, Martin Nöllenburg, Bettina Speckmann, and Kevin Verbeek. Strict confluent drawing. In *Graph Drawing*, pages 352–363. Springer, 2013.

[10] Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):741–748, 2006.

[11] Danny Holten and Jarke van Wijk. Force-directed edge bundling for graph visualization. In *Computer Graphics Forum*, volume 28, pages 983–990. Blackwell Publishing Ltd, 2009.

[12] Peter Hui, Michael Pelsmajer, Marcus Schaefer, and Daniel Stefankovic. Train tracks and confluent drawings. *Algorithmica*, 47(4):465–479, 2007.

[13] Antoine Lambert, Romain Bourqui, and David Auber. Winding roads: Routing edges into bundles. In *Computer Graphics Forum*, volume 29, pages 853–862. Wiley Online Library, 2010.

[14] Sander Leemans, Dirk Fahland, and Wil van der Aalst. Discovering block-structured process models from event logs – a constructive approach. In *Application and Theory of Petri Nets and Concurrency*, pages 311–329. Springer, 2013.

[15] Sander Leemans, Dirk Fahland, and Wil van der Aalst. Exploring processes and deviations. In *Business Process Management Workshops*, pages 304–316. Springer, 2014.

[16] Sheng-Jie Luo, Chun-Liang Liu, Bing-Yu Chen, and Kwan-Liu Ma. Ambiguity-free edge-bundling for interactive graph visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 18(5):810–821, 2012.

[17] Laura Măruşter, Ton Weijters, Wil van der Aalst, Antal van den Bosch, and Walter Daelemans. A rule-based approach for process discovery: Dealing with noise and imbalance in process logs. *Data Mining and Knowledge Discovery*, 13(1):67–87, 2006.

[18] Loïc Royer, Matthias Reimann, Bill Andreopoulos, and Michael Schroeder. Unraveling protein networks with power graph analysis. *PLoS Computational Biology*, 4(7):e1000108, 2008.

[19] Eric Verbeek, Joos Buijs, Boudewijn van Dongen, and Wil van der Aalst. ProM 6: The process mining toolkit. In *Proc. of BPM Demonstration Track 2010*, volume 615, pages 34–39. CEUR-WS.org, 2010.

[20] Eric Verbeek, Joos Buijs, Boudewijn van Dongen, and Wil van der Aalst. XES, XESame, and ProM 6. In *Information Systems Evolution*, pages 60–75. Springer, 2011.

[21] Ton Weijters and Wil van der Aalst. Rediscovering workflow models from event-based data using Little Thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.

Appendix A

Screenshots of results

In this appendix we present some more screenshots of power graphs generated by our implementation. All screenshots in this appendix show the layouts generated by WebCola; they have not been improved manually by dragging vertices.

A.1 Generated data sets

In Figure A.1, A.3 and A.5, we show strict power graph drawings of the *a22*, *a32* and *a42* data sets, respectively. In Figure A.2, A.4 and A.6, we show 0.5-fuzzy power graph drawings of those data sets. Finally, we show a 0.8-fuzzy power graph drawing of the *a42* data set in Figure A.7.

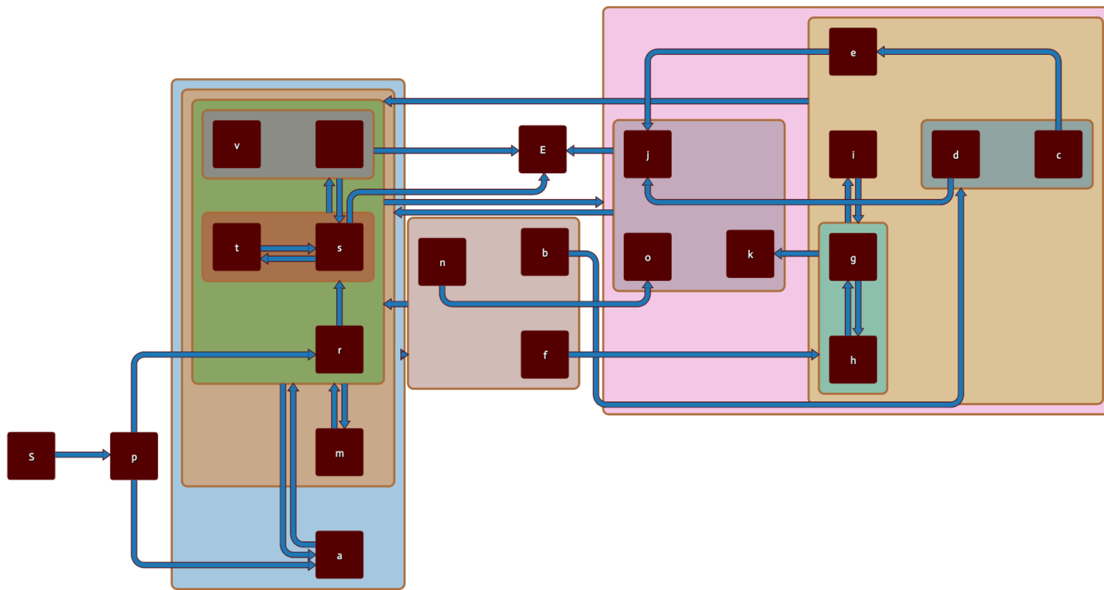


Figure A.1: A strict power graph drawing of the *a22* data set, as generated by our implementation.

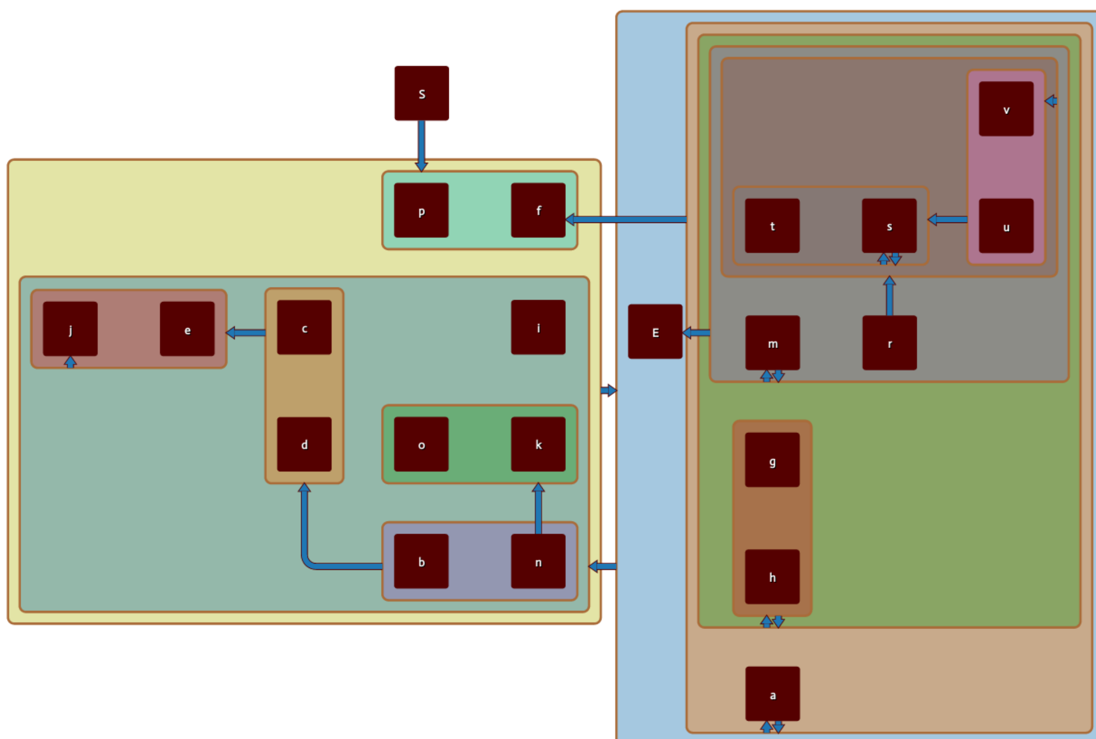


Figure A.2: A 0.5-fuzzy power graph drawing of the *a22* data set, as generated by our implementation.

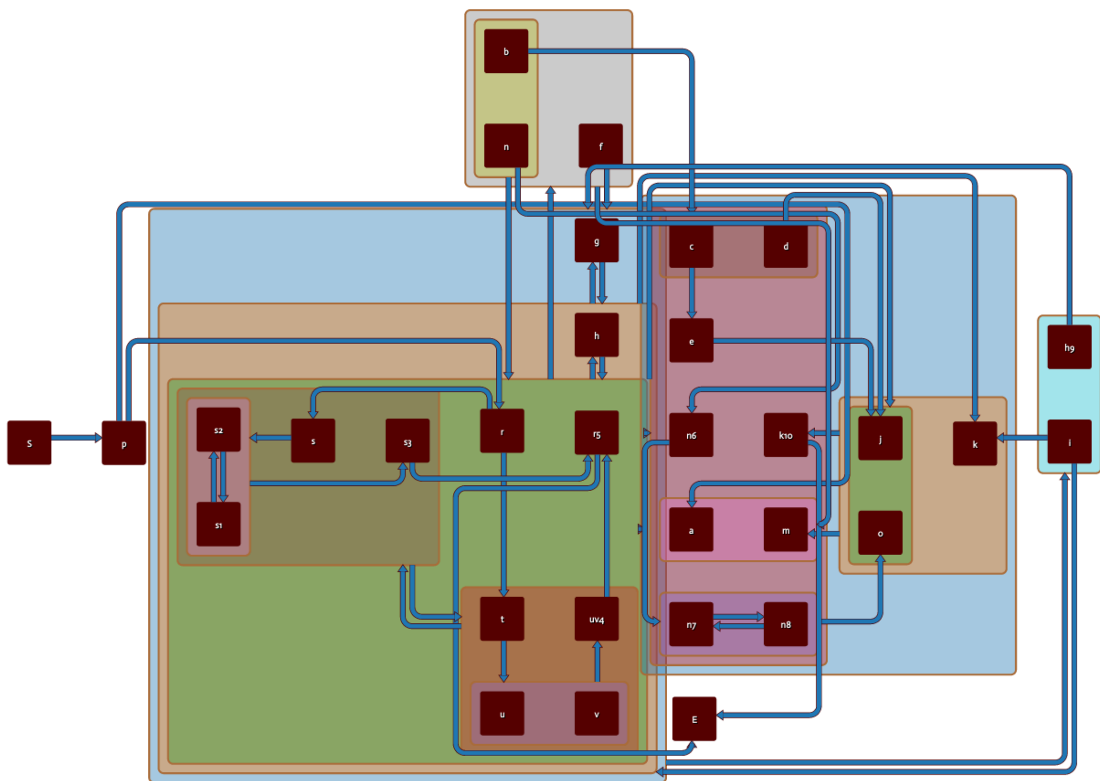


Figure A.3: A strict power graph drawing of the *a32* data set, as generated by our implementation.

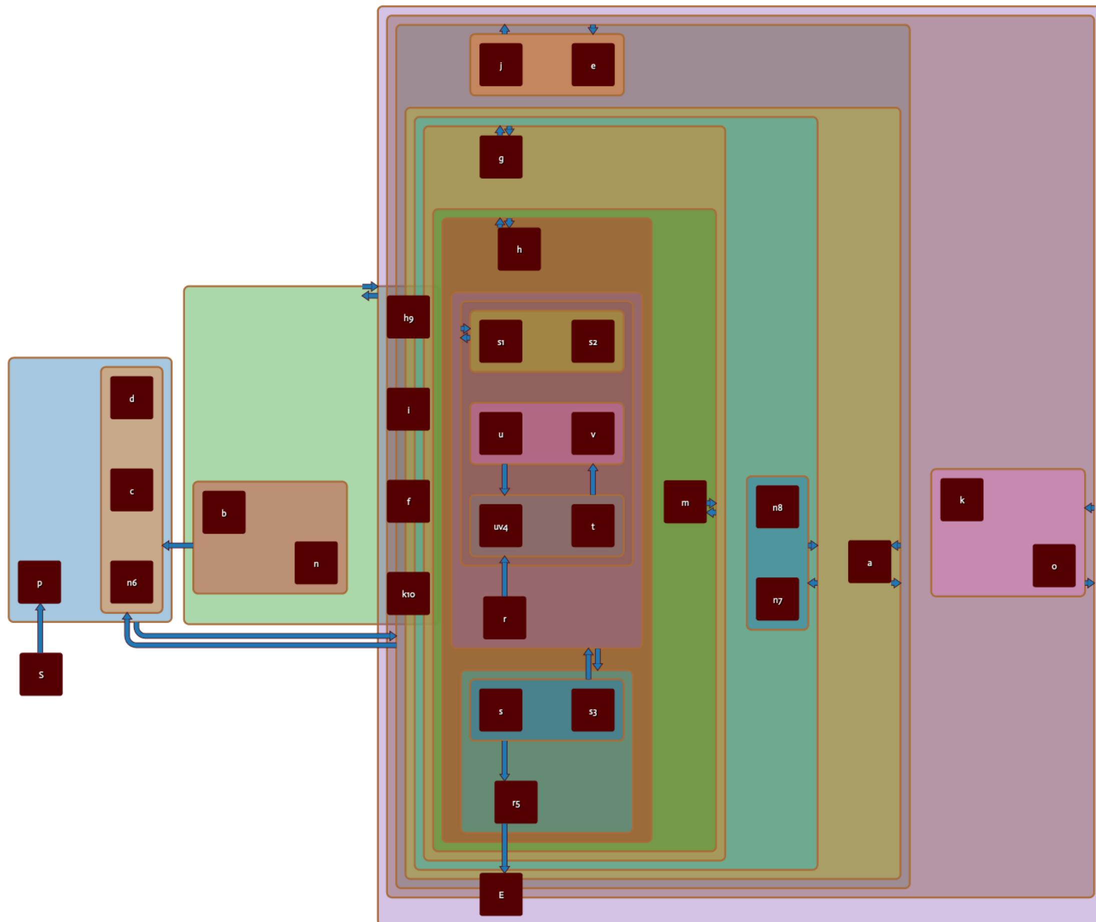


Figure A.4: A 0.5-fuzzy power graph drawing of the *a32* data set, as generated by our implementation.

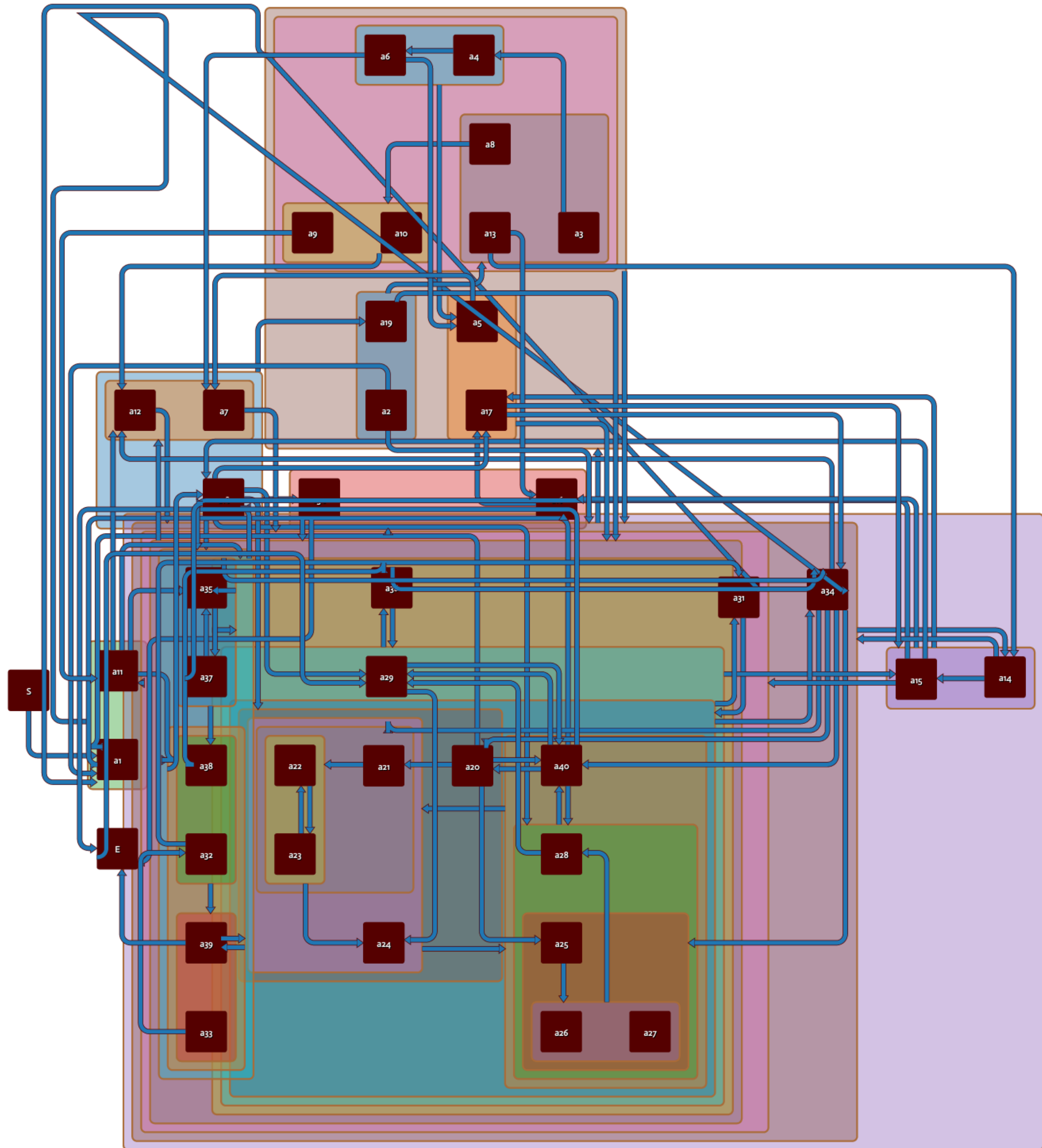


Figure A.5: A strict power graph drawing of the *a42* data set, as generated by our implementation.

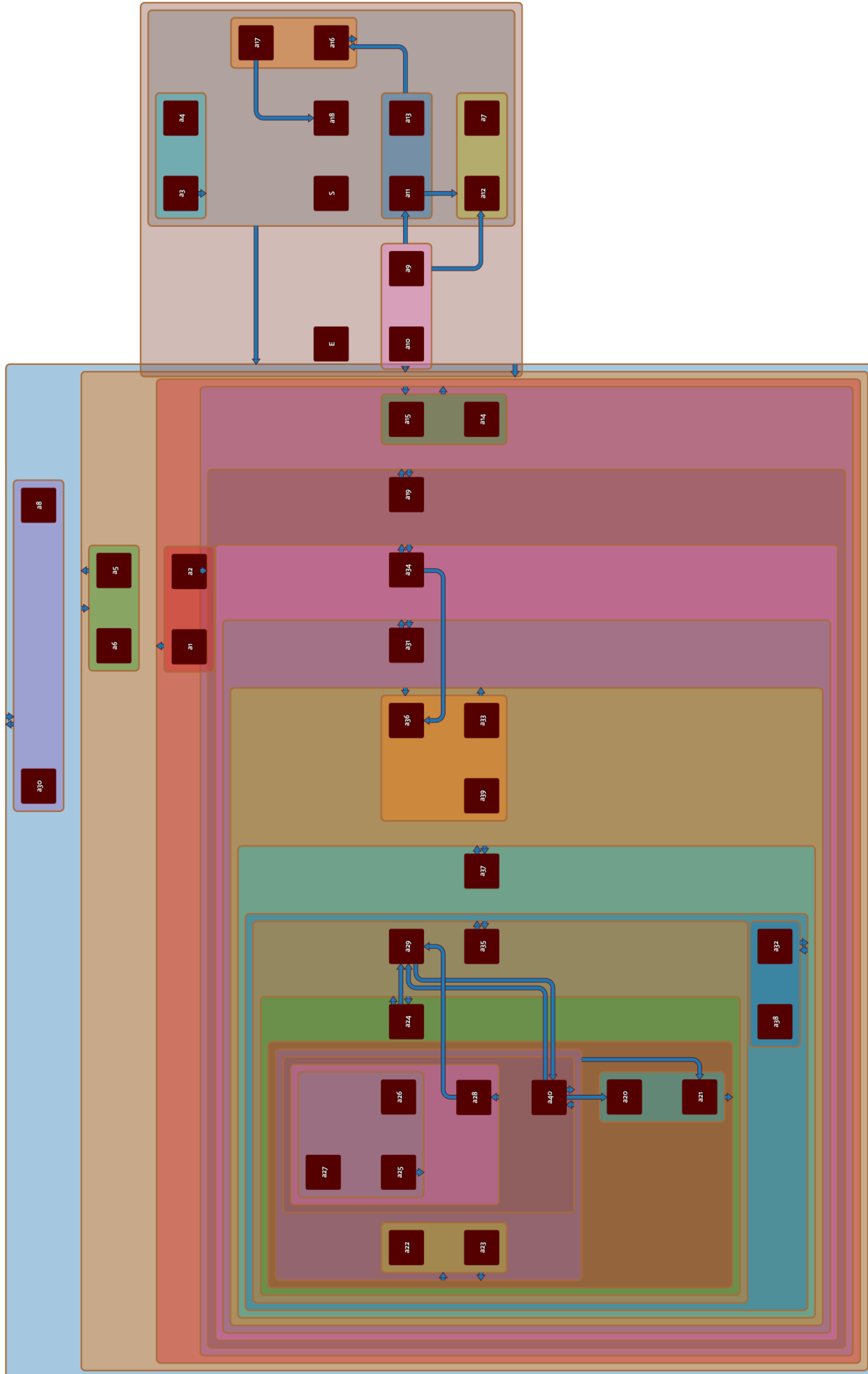


Figure A.6: A 0.5-fuzzy power graph drawing of the *a42* data set, as generated by our implementation.

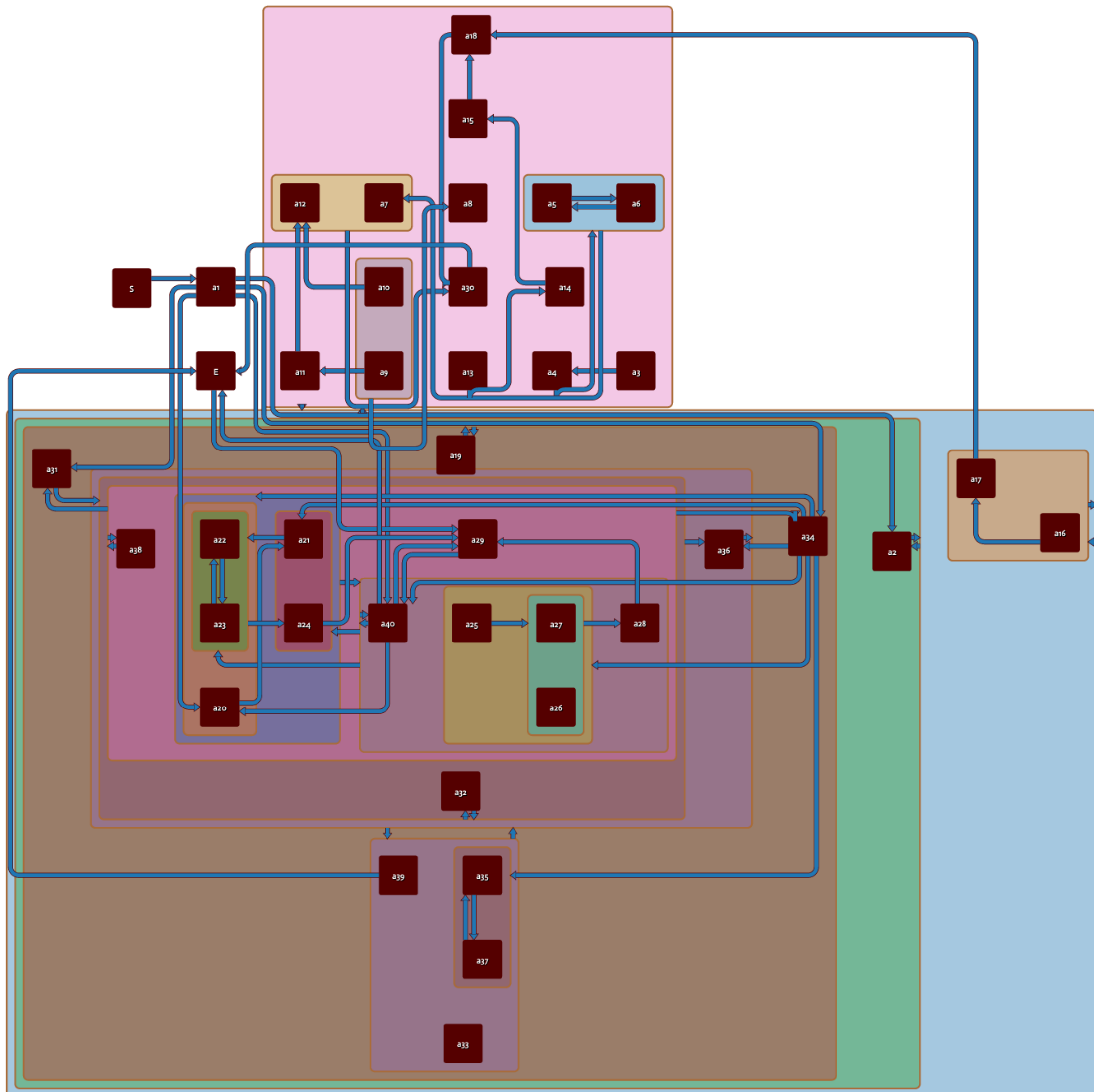


Figure A.7: A 0.8-fuzzy power graph drawing of the *a42* data set, as generated by our implementation.

A.2 The financial log

In Figure A.8, a strict power graph of the financial data set is shown. In Figure A.9 and A.10 α-fuzzy power graphs are shown for $\alpha = 0.8$ and $\alpha = 0.5$, respectively.

For comparison, we show a node-link drawing of the directly-follows graph in Figure A.11. In this drawing, the thickness of the edges encodes the number of times activities directly follow each other.

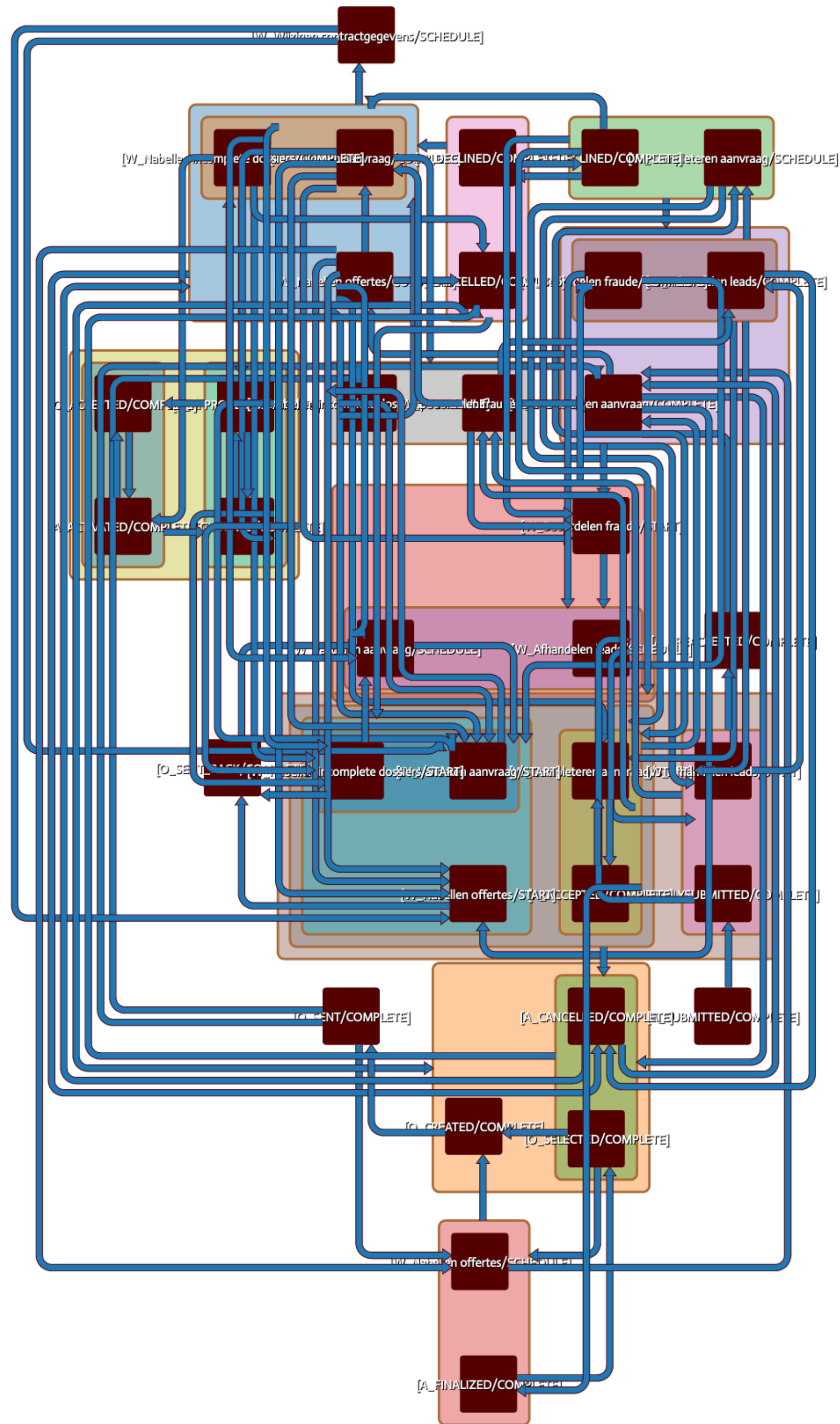


Figure A.8: A strict power graph drawing of the financial data set, as generated by our implementation.

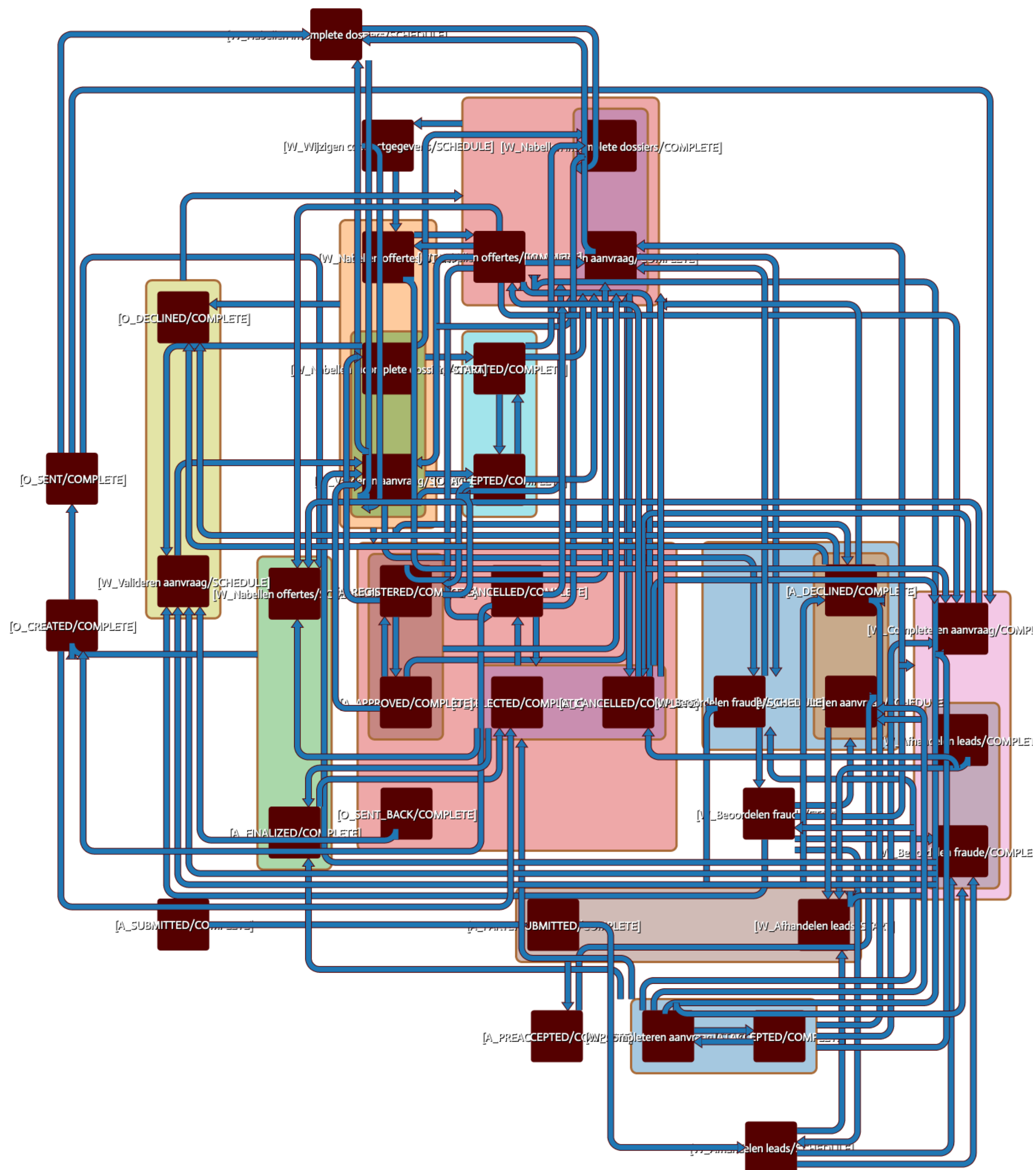


Figure A.9: A 0.8-fuzzy power graph drawing of the financial data set, as generated by our implementation.

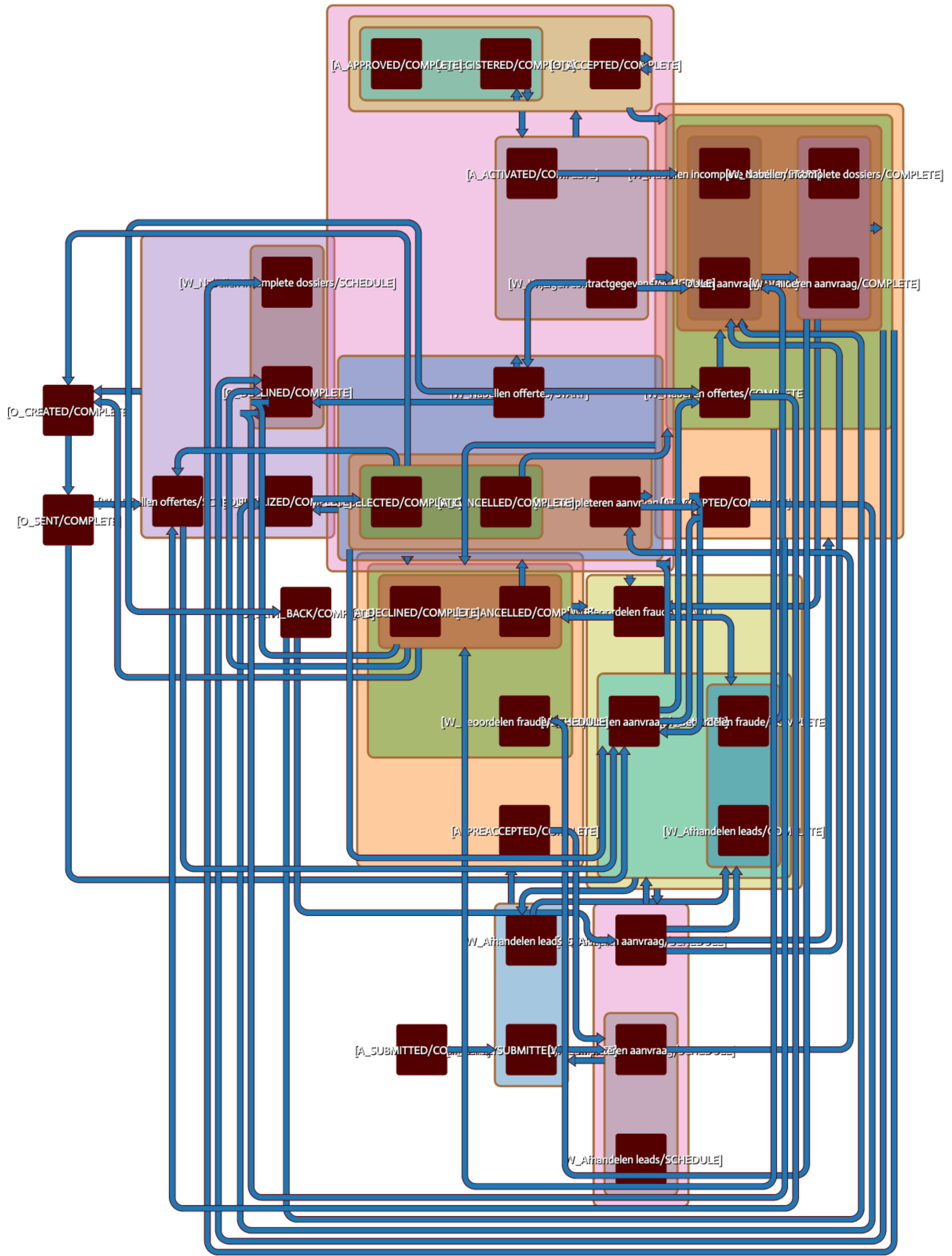


Figure A.10: A 0.5-fuzzy power graph drawing of the financial data set, as generated by our implementation.

Figure A.11: The directly-follows graph of the financial data set.