UCLouvain

epl

École polytechnique de Louvain

Summarising and Visualising Large Acyclic Graphs

Author: **Philippe Stormme**
Supervisors: **Kim Mens, Guillaume Maudoux**
Reader: **Peter Van Roy**
Academic year 2018–2019
Master [120] in Computer Science

Contents

Chapter 1

Introduction

The purpose of this thesis is to explore ways of summarising large acyclic graphs. Past a certain threshold in the amount of nodes or edges, a human being will not be able to extract relevant information from a graph visualisation. This threshold may depend, among other elements, on the graph layout and the "kind" of relevant information. Aside from human limitations, a computer simply cannot build a visual representation of a graph past a certain threshold in the amount of nodes or edges. Once again the threshold depends on different factors, such as the required layout, the amount of edges per node, and the computer building the visualisation. We are here interested in building visual representations such that the amount of nodes and edges is below the human threshold, which is unsurprisingly (usually) inferior to the computer threshold.

Of course, summarising, be it a graph, a book, or an event, comes at a cost. If we were to write a 20 pages long summary of World War I, it would be unlikely to mention that when the war started, the colours of the French uniform, bright red and blue, made the soldiers easy targets for their opponents and was therefore replaced (several times) by an outfit more suited for modern warfare. That is, of course, if the purpose of the summary is to give the reader a general understanding of the war. This seemingly irrelevant information could find its place in the summary if we were to follow the French point of view, it could even become a key part of the summary if our focus was the evolution of the equipment of foot soldiers. Therefore, determining the set of relevant information is not trivial and depends on the purpose of the summary. For any non-trivial graph, there isn't a single summary (we're done with WW1) and we might very well be unable to build an exhaustive, strict hierarchy between the possibly many summaries.

To make large graphs understandable by a human being, we'll explore ways of summarising an original graph by applying to this graph a sequence of filters. A filter is used to merge similar nodes, drop parts of the graph that are deemed uninteresting, or enrich information about nodes and edges.

1.1 Related work

Summarising graphs isn't exactly an entirely new matter, we'll now dig into the work of those who have taken or are still taking interest in this topic. The following list isn't exhaustive but should constitute a decent sample of approaches aiming at making graphs understandable.

1.2 Polymetric views

Polymetric views are to be used on relatively small graphs. The idea is to use the position, dimensions, and colour of a two-dimensional node or the dimensions and colour of an edge to visualise information, such as software source code metrics. Figure 1.1 is reproduced based on "CodeCrawler - polymetric views in action" [7] and describes the node features on which polymetric views rely. Our work offers some support for creating such views.

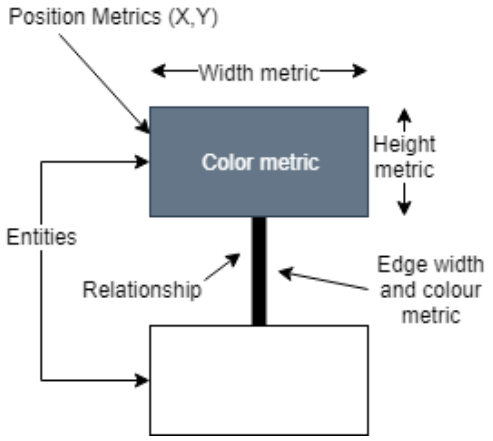

Figure 1.1: The principles of a polymetric view [7] (reproduction)

1.3 Graph Visualisation systems

Software such as ASK-GraphView [1] allows to manipulate large graphs and break down an original graph into a set of larger pieces through clustering operations. It also provides some support for labelling, colouring, etc. The clustering makes it possible to display a very large graph, as many nodes are represented as a single one. The software allows itself a few seconds to generate the layout of a cluster when requested. ASK-GraphView [1] is meant to be used with undirected graphs,

but it shows some degree of similarity with our own approach. Figure 1.2 shows the main panel of this software.

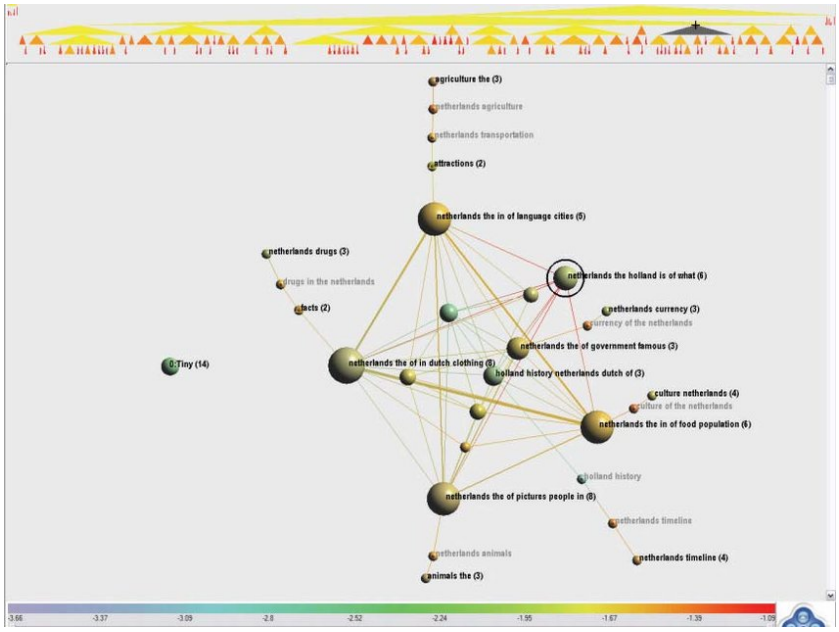

Figure 1.2: The main panel of ASK-GraphView [1]

1.4 Edge bundling

Edge bundling aims at reducing visual clutter. It also highlights high-level edge patterns. Figure 1.3 illustrates this. Edge bundling is not explored in this thesis but we'll briefly distinguish between 3 edge bundling techniques.

Force directed edge bundling This method, used by Holten and van Wijk in their paper "Force-directed edge bundling for graph visualization" [5], does not require the graph to contain a hierarchy nor the construction of a control mesh. The specificity of the result of this method is a reduction in curvature variation and thus smoother bundles.

Image-Based Edge bundling This approach is used by Telea and Ersoy in their paper "Image-based edge bundles: Simplified visualization of large graphs" [8]. This method involves the computation of a hierarchy. The result of this method "displays a given graph as a small set of overlapping shaded edge bundles". Colours hold information about edge type, density and similarity.

Figure 1.3: US migration graph (1715 nodes, 9780 edges) [5]

Geometry based edge clustering This approach, used by Cui et al in their paper "Geometry-based edge clustering for graph visualization" [3], uses a control mesh to "guide the edge clustering process" [3]. This approach is claimed to be "intuitive, flexible and efficient" [3] by its authors.

1.5 Highlighting local graph structures

The paper "Summarizing and understanding large graphs" [6] aims at finding important local graph structures in large graphs and highlight those local structures. Local structures consist of bipartite cores, stars, cliques and chains. Taking those local structures into account when building the visualisation can make it more compact and or understandable. Figure 1.4 shows the process by which VoG, the method proposed in "Summarizing and understanding large graphs" to highlight local structures, summarises a graph.

1.6 Acyclic graphs

To conclude this introduction, we will ask ourselves what are acyclic graphs and list a few of their applications. As it is still rather general, we'll rely heavily on Wikipedia. On mathematical matters, Wikipedia articles constitute a decent introduction to many subjects and we'll just use quotes from "Directed Acyclic Graph" [2] for some applications. Some of these applications will be considered in the next chapters.

Why Acyclic Graphs? Our initial concern was only summarising dependency graphs. Dependency graphs are necessarily acyclic. Extending the topic from

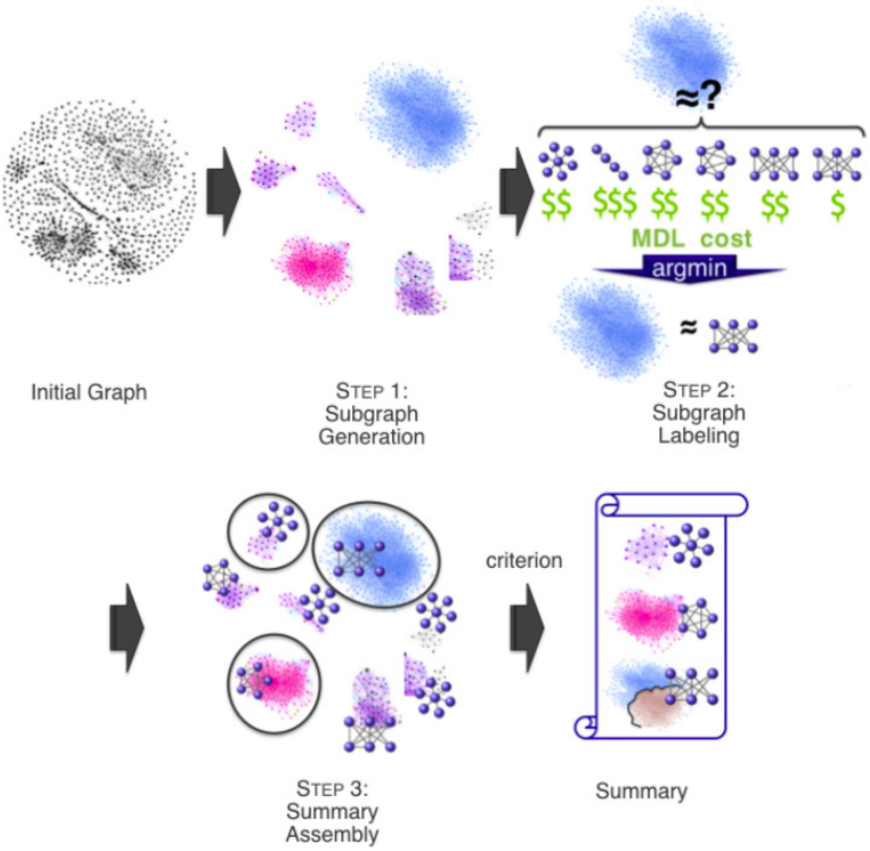

Figure 1.4: Illustration of VoG step-by-step [6]

dependency graphs to acyclic graphs in general introduced several concrete types of graphs and summarisation options as well as new possible uses of the results. A filter designed to summarise one type of acyclic graph is usually applicable to other types as well. Therefore the decision of extending the topic made sense as it served our initial purpose. The choice of not widening the topic any further, to all directed graphs, was motivated by the fact that some filters exploit the absence of cycles and that directed graphs have properties inapplicable to acyclic graphs when it comes to manipulating dependency graphs.

Acyclic Graph "In mathematics, particularly graph theory, and computer science, a directed acyclic graph (DAG), is a finite directed graph with no directed cycles. That is, it consists of finitely many vertices and edges (also called arcs), with each edge directed from one vertex to another, such that there is no way to start at any vertex v and follow a consistently-directed sequence of edges that eventually loops back to v again. Equivalently, a DAG is a directed graph that has

a topological ordering, a sequence of the vertices such that every edge is directed from earlier to later in the sequence." [2]

Tree A tree is a graph in which any pair of nodes is connected by exactly one path. In the case of a directed graph, all the edges must point away from the root (the root being the node without any incoming edge).

1.7 Applications

Scheduling "Directed acyclic graphs representations of partial orderings have many applications in scheduling for systems of tasks with ordering constraints. An important class of problems of this type concern collections of objects that need to be updated, such as the cells of a spreadsheet after one of the cells has been changed, or the object files of a piece of computer software after its source code has been changed." [2]

Data processing network "A directed acyclic graph may be used to represent a network of processing elements. In this representation, data enters a processing element through its incoming edges and leaves the element through its outgoing edges." [2]

Causal structures A directed acyclic graph may be used to represent events and causality relations between those events. Bayesian networks are a more concrete application of this type.

Genealogy and version history Genealogy trees, which are usually not trees (see figure 1.5), are always acyclic. This is also true for version history once there have been merges and branches in the history.

Citation graphs "In a citation graph the vertices are documents with a single publication date. The edges represent the citations from the bibliography of one document to other necessarily earlier documents" [2]

Data compression "Directed acyclic graphs may also be used as a compact representation of a collection of sequences. In this type of application, one finds a DAG in which the paths form the given sequences. When many of the sequences share the same subsequences, these shared subsequences can be represented by a shared part of the DAG, allowing the representation to use less space than it would take to list out all of the sequences separately." [2]

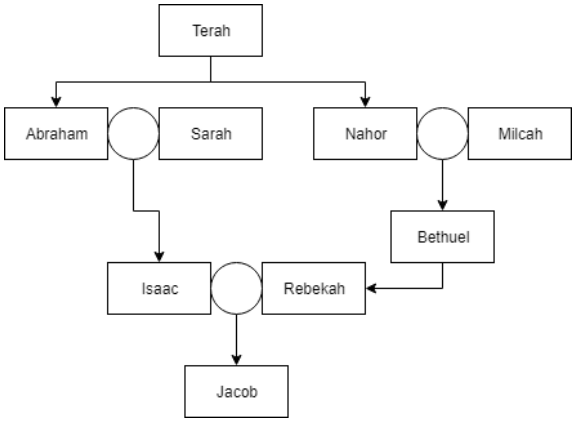

Figure 1.5: Partial genealogy of Abraham

Hierarchical data Proper trees (unlike genealogy trees) can also be seen as acyclic graphs and an additional application of acyclic graphs becomes the representation of hierarchical data such as syntax trees or directory tree hierarchies.

1.8 DAG manipulation through GraphTool

It is possible to build a summary of a given graph by writing a linear piece of code. However, we stated earlier that usually, there isn't a single summary for a non-trivial graph. But it's okay, we can comment a few lines of our piece of code and replace them to meet our new expectations. However, things quickly get messy if we are to try more than 2 or 3 summaries. Fine, enough with comments, why not isolate the independent pieces of code responsible for one (step) of the summarisation process. Then we can specify a sequence of those independent pieces of code (that we'll call filters from now on) as program parameters and try different summaries more easily. This seems like a reasonable solution. However, it is still necessary to know the sequence of filters from the start. Making a change in the sequence implies that we need to re-run the whole process from the start. This is where our Java application, that we'll call GraphTool, comes in. GraphTool allows the user to manipulate acyclic graphs stored in different file formats, by applying a sequence of filters. The graphical interface allows to add, remove and swap filters, usually without repeating the whole summarisation process. But more on that in the next chapter.

Chapter 2

Technology

To achieve our goal, we used some existing software and libraries, as well as a Java application: a small tool used to apply filters to an original graph in order to produce as summary as a textual representation in dot language. This textual representation being then used either independently or directly by the application to produce a visualisation with the help of GraphViz.

2.1 GraphViz

GraphViz[1] is an open source graph visualisation software. GraphViz is used to turn a textual definition of a graph (in dot language) into a visualisation in SVG, PNG, PDF, etc. The dot language allows to specify many node, edge, or graph features to customise the output. GraphViz uses different layout algorithms such as dot (probably not unrelated to the dot language but it is something different), neato, or circo. Most visualisations used in this document, such as figure 2.1 are produced with GraphViz.

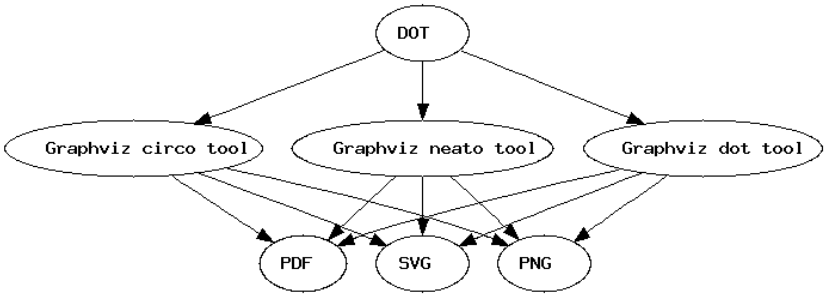

Figure 2.1: Producing graph visualisations with Graphviz

[1] https://www.graphviz.org/

2.2 GraphStream

GraphStream[2] is an open source graph handling Java library. It allows to manipulate directed and undirected graphs, read and write in various formats. The next few paragraphs describe why, at the beginning, GraphStream was used in GraphTool but was discarded later.

GraphStream is convenient The library contains a Java representation of graphs, which can stored and retrieved to and from different kinds of files (among which files in the dot language used by GraphViz). It allows to generate visualisations, with different layouts. It was convenient to start working.

GraphStream has a weak community GraphStream is an open source project for which the code is split in 31 repositories, some of which were not updated in the last five years. One of those has a README with this content: "TODO". While the project grew large, its development today seems to be almost frozen.

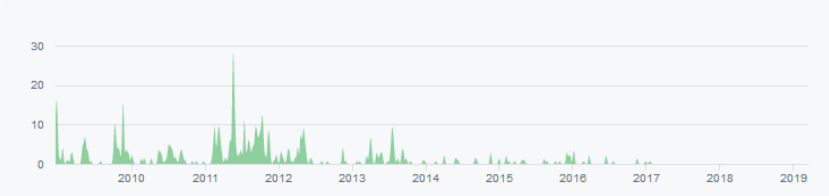

Figure 2.2: Contributions to the main GraphStream repository

GraphStream releases Between two GraphStream releases, there are certainly improvements. On the other hand, it is unfortunate that the documentation sometimes describes methods which are removed or renamed in the latest release. Not deprecated, just gone. Of course, the site documentation does not describe how things are done in the latest release.

GraphStream is large GraphStream offers many possibilities, just like a Swiss Army Knife. But after using such a knife for some time, with the sole purpose of cutting your vegetables, you start thinking you'd be better off with an appropriate, sharp kitchen knife.

[2] `http://graphstream-project.org`

GraphStream and the dot language GraphStream allows to read and produce dot language. However, at some point, it appeared that parsing some dot graphs led to a failure. Expecting a fix wouldn't have been reasonable. Personally fixing the GraphStream dot parser would have been an option but would have required some time to find the problem and work on a solution. It was faster to write a new dot parser, outside of the library.

GraphStream discarded All GraphStream classes uses were replaced by personal classes without major apparent loss. Of course, GraphStream had an influence on the classes written to replace its own.

2.3 GraphTool

As stated before, GraphTool is the Java application used to manipulate graphs and create different summaries. Figure 2.3 shows a schema of the graphical interface of GraphTool, the next paragraphs are a brief description of each element represented in the schema.

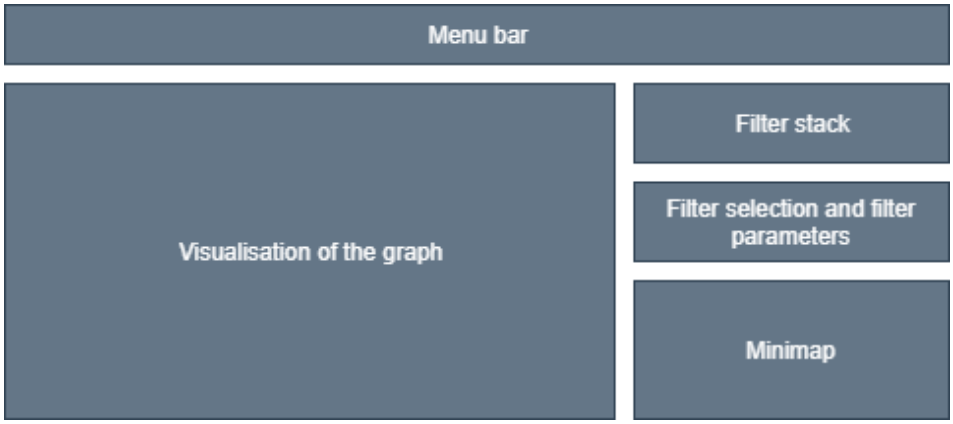

Figure 2.3: Schema of the interface of GraphTool

Menu bar The menu bar is where the user should go to open a graph, save it, export the current visualisation as a PNG image, undo or redo an action, etc.

Visualisation of the graph The visualisation panel displays a visualisation of the graph in its current state, considering the application of the filters on the filters stack. This visualisation is purely static.

Filter stack The filter stack displays the sequence of filters that were applied to the current graph. But it isn't only there to show information, it is also possible to remove a filter from the stack or move it up or down on the stack. It is also in this area that the user may save the session: the current graph as well as the sequence of filters. The session is restored when GraphTool starts.

Filter selection The filter selection panel is where the user may select a new filter to apply to the graph, some filters also have parameters that need to be specified by the user. Once the selection is made, the filter is applied and added on the top of the filter stack. The proposed filters will be discussed further in this document.

Minimap The minimap displays a down-scaled visualisation of the visualisation panel. It can be used to navigate in the visualisation, which can be rather large during summarisation.

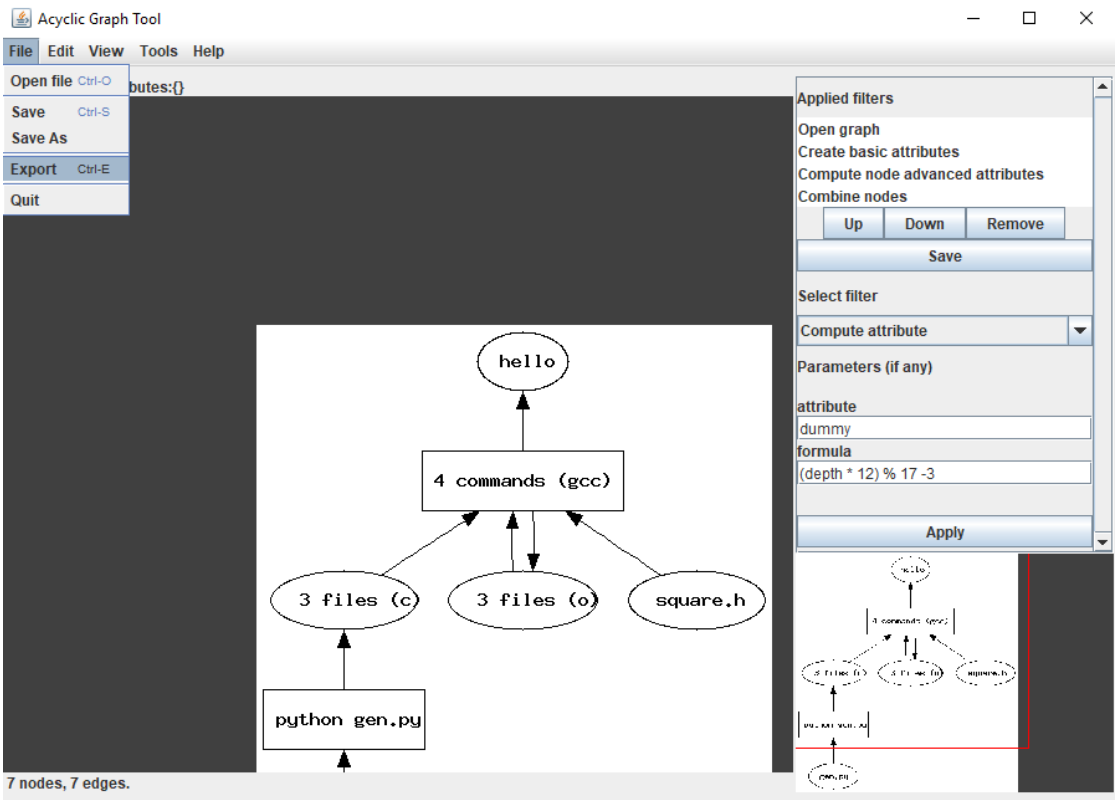

Figure 2.4: Screenshot of GraphTool running on Windows 10

2.3.1 The filters

As mentioned before, in order to summarise a graph, we make use of filters. In GraphTool, a filter is a class implementing the following interface:

```
public interface Filter {
    /**
     *
     * @param src: an input graph.
     * @param parameters: the filter parameters.
     * @param copy: whether changes should be applied
     *            directly to 'src' or to a copy.
     * @return the input graph after the filter was
     * applied to it.
     */
    Graph apply(Graph src, Map<String, String> parameters, boolean copy);

    /**
     *
     * @return the name of the filter,
     * to be displayed on the GUI.
     */
    String getName();

    /**
     *
     * @return the list of parameters of the filter,
     * to be used on the GUI.
     */
    FilterParameter[] getParameters();
}
```

A filter should precompute new information to the nodes or edges of the input graph to make it available for other filters, change the visual representation of nodes and edges, merge similar nodes, or remove insignificant nodes.

The filter selection

The user can select the appropriate filter from a drop-down list. Some filters take additional parameters. The objects returned by the "getParameters" method of the filter interface are used to generate text fields to specify those parameters. Once completed by the user, their content is passed to the "filter.apply" method.

2.3.2 The filter stack

The purpose of the filter stack is to allow the user to remove an unnecessary filter from the stack or to move a filter up or down the stack. Each of those operations requires to rebuild part of the stack. Unless of course if the operation is the removal of the top of the stack.

The FilterContainer object The filter stack does not really only contain filters. The objects on the filter stack are called "FilterContainer". A FilterContainer object has 3 attributes and getters to recover them. The 3 attributes in question are a graph, the filter by which it was generated, and the parameters passed to this filter.

The filter stack is convenient, not necessary GraphTool does not need the filter stack, the end result would be the same without it. But thanks to this stack, it is possible to fix mistakes and try different summaries for a graph. Saving the stack allows the user to recover the session in the same state as it was when the stack was last saved.

Keeping multiple copies of large graphs For very large graphs, keeping several copies of the graph can use too much memory. It is possible to "disable the stack". In that case, it will still be possible to save the stack, but not to modify it to fix mistakes.

2.3.3 GraphViz usage

GraphTool relies on GraphViz to generate visualisations. GraphTool can read and write to dot language. In order to produce a visualisation, the following process is executed:

1. A temporary file in dot language, representing the graph in its current state, is generated.
2. GraphTool uses GraphViz to produce the visualisation. *dot* is one of the command-line executables of GraphViz. It produces visualisations with the dot layout. *dot* is used to produce visualisations for directed graphs as hierarchies.

```
dot temp.dot -Tpng:gd -o temp.png
```

3. The GraphTool GUI displays the generated image.

Algorithm 1 Filter stack operations

```
1: procedure APPLY_OPERATION(stack, f, operation)
2:     temp ← Stack()
3:     while stack.size() > 1 and stack.peek() != f do
4:         temp.push(stack.pop())
5:     if stack.size() = 1 then
6:         return
7:     operation(stack, temp)
8:     while not temp.empty() do
9:         fc ← temp.pop()
10:        g ← stack.peek().graph
11:        fc ← fc.filter.apply(g, fc.parameters)
12:        stack.push(fc)
13: procedure REMOVE(stack, temp)
14:    stack.pop()
15: procedure MOVE_UP(stack, temp)
16:    if temp.size() > 0 then
17:        fc ← stack.pop()
18:        fc1 ← temp.pop()
19:        temp.push(fc)
20:        temp.push(fc1)
21: procedure MOVE_DOWN(stack, temp)
22:    if stack.size() > 1 then
23:        fc ← stack.pop()
24:        fc1 ← stack.pop()
25:        temp.push(fc)
26:        temp.push(fc1)
```

If a graph has too many nodes or edges, GraphTool does not try to produce the visualisation, as *dot* would run "forever" (forever being at least a few minutes). Sometimes it is not possible to produce a visualisation even though the amount of nodes and edges seems reasonable. An example of such a situations would be the construction of a small graph with a node having dozens of children, with the additional constraint according to which the child nodes should be represented as very large rectangles.

2.4 I/O formats

GraphTool can read various input format and write to a widely used graph representation as output. GraphTool can also export visualisations as PNG images.

2.4.1 A single output format

dot GraphTool can only produce files in dot language. Countless programs can process dot files and the format is documented online, therefore it shouldn't be a problem to reuse graphs produced with GraphTool.

2.4.2 A few input formats

It is not a problem to have a single widely used format as output. But as input, it seems better to handle various standard formats. That way, it is possible to work on different types of graphs without spending time to make them fit a single representation at the cost of some time. One of the input formats is of course the dot language, but there are 3 additional graph representations handled by GraphTool. Those additional representations can be easily generated with tool probably already accessible on the user's system.

dot Files in dot language are accepted by GraphTool as input. To be able to apply filters on the graph, it should by acyclic and strict (for a given pair of nodes and a direction, there is only one edge), but no other restrictions are imposed. Most filters expect some attributes to be present on nodes, such as the depth of the node, but this information can be added once the graph is loaded.

tree GraphTool can read graph representations produced by the *tree* command. *tree* can produce the list of files in a directory and can include the size of each file if the "–du" (disk usage) parameter is used. The output of this command is directly usable as input for GraphTool.

git log The output of the *gitlog* command can be parsed as a graph object by GraphTool, if the command was used with the right set of parameters:

```
git log --all --date-order --pretty="%H|%P|%d" --numstat
```

The output of this command matches the following structure:

```
<commit id> | <parent commit 1> ... <parent commit n> | [(<note>)]

<plus> <minus> <file 1 path>
...
<plus> <minus> <file n path>
```

That way the output of the command is the complete list of commits in a git project, specifying the parent(s) of each commit and the changes related to the commit (the numbers in "plus" and "minus" place).

Java project Given a directory containing a Java project, GraphTool can produce a graph with the packages and Java source files. The nodes correspond to a source file and contain the amount of lines and methods as attributes.

Chapter 3

Methods and Results

3.1 Applications of acyclic graphs

Depending on the given concrete use, the appropriate summarisation of a graph is expected to be different. We will now consider some of the applications considered in the introduction: scheduling, causal structures, genealogy and version history, and hierarchical data (Data processing networks, citation graphs and data compression related graphs won't be discussed).

3.1.1 Scheduling

Among the different types of scheduling graphs, we will consider dependency graphs. To summarise such a graph, we will regroup similar nodes, with the hope of drastically reducing their numbers. The nodes are then assigned an equivalence class, the edges then representing the relations between those equivalence classes rather than between individual nodes. Depending on how we define those classes, this process will produce different summaries that may or may not hold any meaning. Of course this choice depends on the purpose behind the creation of the summary. A disadvantage of this process is that the summary will not necessary be acyclic.

Example The figure 3.1 is a toy dependency graph. The nodes have been coloured in red(ish) and blue. Red nodes also have an oval shape, while blue ones are rectangular. Red nodes represent files and the blue ones represent commands. If the nodes were to be split in 2 different equivalence classes depending on whether the node represents a file or a command, the visualisation representing the relation between those classes would be the result on figure 3.2. Defining the equivalence classes differently will result in different summaries, as can be seen on figure 3.3. For example, in the top left summary, equivalence is based on the depth of the

nodes and in the bottom right summary, the equivalence is based on the command name (for commands) and the file extension (for files). Unfortunately, most of these summaries are not acyclic anymore. It does not necessarily make the visualisation less meaningful but makes the use of subsequent filters impossible if those rely on the acyclic nature of the graph.

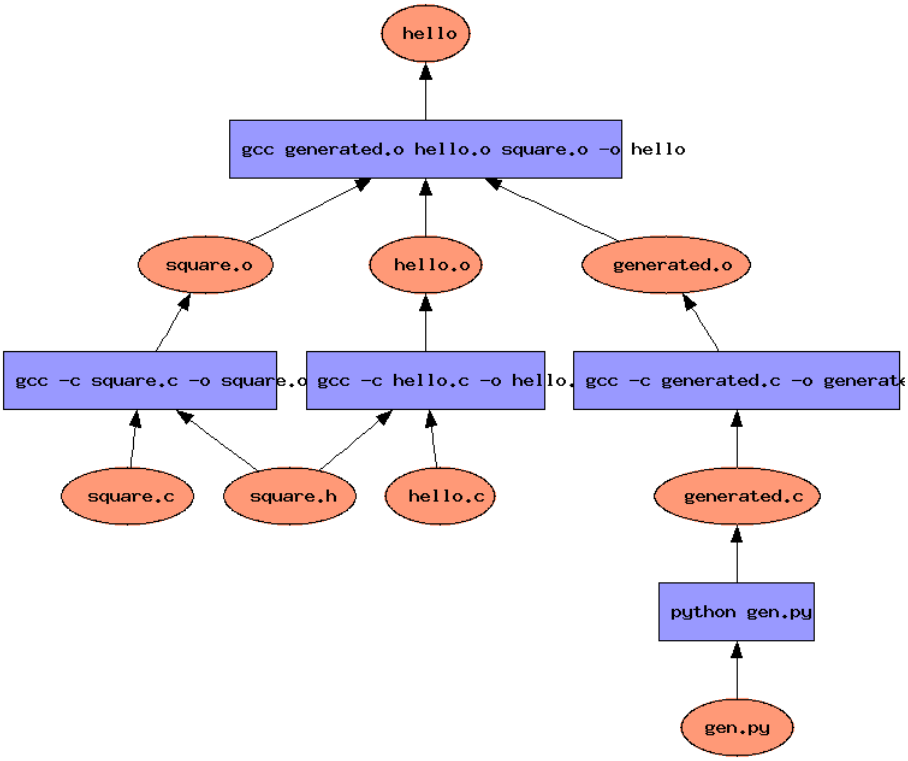

Figure 3.1: Toy dependency graph

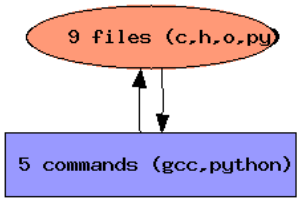

Figure 3.2: Summary of the dependency graph on figure 3.1

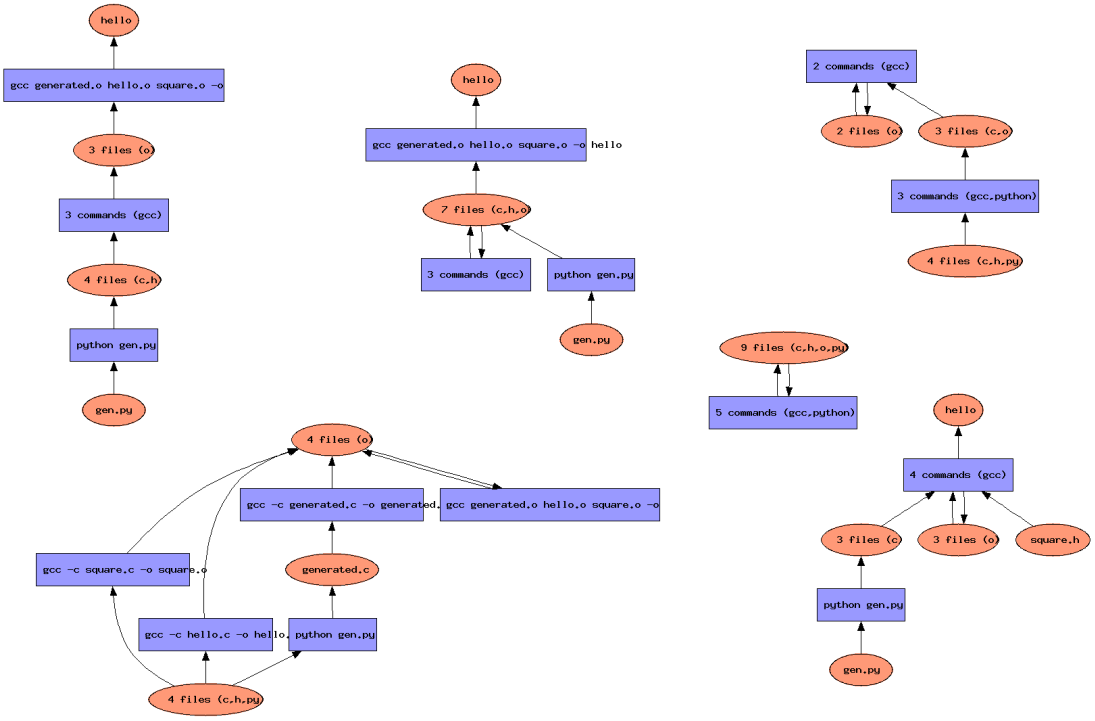

Figure 3.3: Summaries of the dependency graph on figure 3.1

3.1.2 Causal structures

An interesting example of causal structure would be “Bayesian causal networks”. One way to reduce the amount of nodes and edges in such a graph would be to remove unlikely events.

3.1.3 Genealogy and version history

Genealogy tree Summarising genealogy trees will not be considered in this thesis as it is most likely both a manual process requiring interactive visualisations and a matter widely covered by genealogy software.

Version history History created by version-control systems such as git can be seen as a type of acyclic graph. Each commit is represented by a node and each edge represents the parenthood relationship between 2 nodes. A summary of such graphs could use the amount of changes in a commit to only preserve important changes. This is the approach used in the following example.

Example Given a small git graph (5 contributors, 127 nodes, 141 edges), figure 3.4 is a representation of the graph produced by GraphTool. This small git graph is still too large to be represented in this document. Figure 3.5 is the visualisation of a summary of the graph in figure 3.4. On both figures, the size of a node represents the amount of changes in the commit. Even though the original graph probably looks blurry in this document, the general look of the graph do match. The biggest commits seem to tend to involve merges around them and long chains get considerably reduced in the summary. The summary is obtained by hiding the commits modifying less than 100 lines and thus contains only 32 nodes and 39 edges.

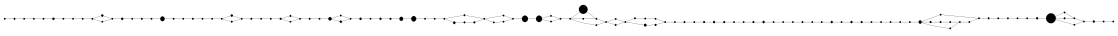

Figure 3.4: Original "small" git graph

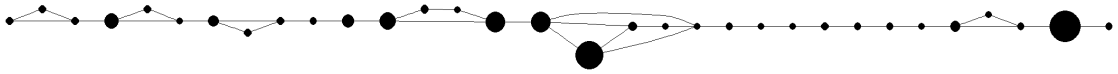

Figure 3.5: Reduced "small" git graph

In figure 3.5, insignificant nodes are hidden. More specifically, when an insignificant node N is removed, the set of the nodes pointing to N are linked to the set of nodes pointing from N by an edge. This way of summarising the graph preserves the reachability relations between the remaining nodes and usually preserves a similar global structure. However, the summarisation creates local aberrations. On figure 3.6, we can see more clearly a piece of the original graph above the corresponding piece of the summary. The paths connecting a few chosen nodes are highlighted, the purpose of colours is to show the correspondence between the nodes in the original graph (above) and the summary (below). We can see that the purple path suffered significant deformations, yet the "bigger picture" is similar.

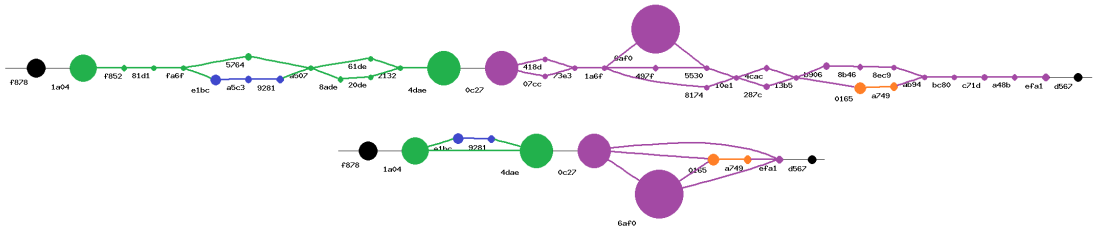

Figure 3.6: Local aberrations in git graph summary

3.1.4 Hierarchical data

An example of hierarchical data (data represented as a tree) is directory tree hierarchy. In such a graph, each node represents a file or a directory and each edge represents a parenthood relationship between two nodes. In order to summarise this kind of graph, we could cut some nodes based on some characteristic (like its weight or depth in the tree) and maybe make use of the shape of the remaining nodes to display relevant information.

Example 1 The visualisation in figure 3.7 is a summary of the tree representing the directory containing the code of GraphTool. The summary contains 17 nodes and 16 edges while the original graph contains around 300 nodes and edges. This original graph is visible in figure 3.8. The size of a node in the summary represents the average size of the files contained in the node (directory). A node is missing in the summary if it was hidden by application of 2 filters. The first one hid all the nodes with a depth greater than 3 and the second one hid files and directories containing a single file. 8 other filters were applied for visual purposes.

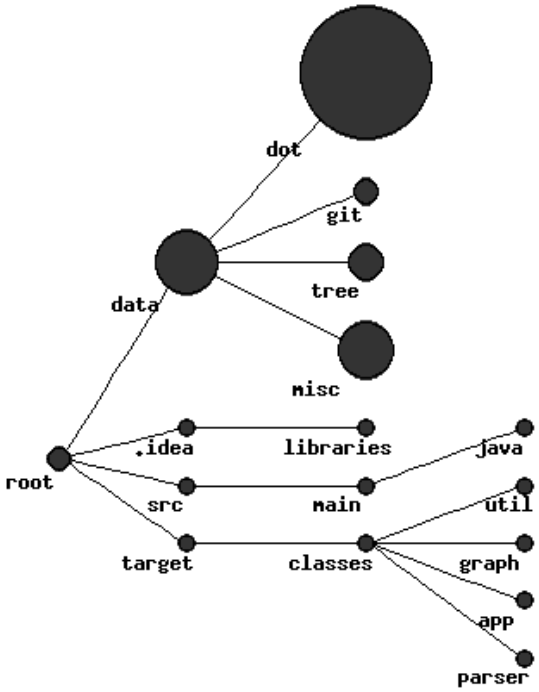

Figure 3.7: GraphTool directory summary

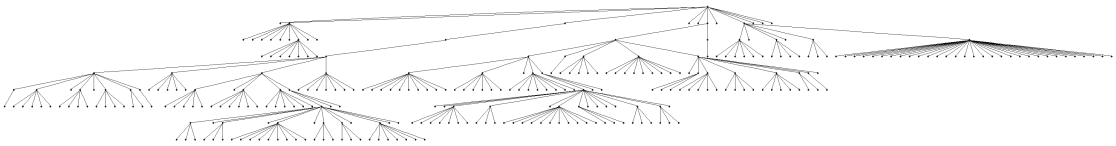

Figure 3.8: GraphTool directory

Example 2 Let's consider a digital comic book library with only 3 series represented by 3 directories. The owner of the library wishes to start reading one of

these series. He wants to base his choice on the time necessary to read a chapter and the time necessary to read the whole series, for some reason. Each of the 3 directories contains all chapters of its series, stored as cbz archives (cbz is zip for comic books). A chapter is either 20 or 40 pages long in average, depending on the series. This information is not available in the input graph. But the size and amount of files representing chapters is known. If we assume that all pages have the same quality, we can deduce the relative amount of pages between series from the size of the file. The 3 series contain 19, 598, and 940 chapters (files).

Figure 3.9 is the visualisation of the summary of the library. The original tree representing the library contains almost 1600 nodes and is therefore not displayed in this document. The summary is produced by application of the following sequence of filters:

1. Hide nodes with depth above 1, leaving depths 0 and 1, the root directory and the directories corresponding to each series. The nodes still hold two values: the size on the hard-drive (an approximation of the amount of pages in the series) and the amount of files in the directory (the amount of chapters).

2. Give a rectangular shape to the nodes, use the width, height and area of the nodes to represent information.

3. Compute the amount of pages by chapter of each series. The exact amount of pages is unknown, but a multiple of this amount is obtained by dividing the size of the series directory by the amount of its children. An approximation of the exact value is good enough for what it'll be used for. To compute this value, the filter can evaluate a simple formula using other attributes of the node as variables.

4. Set the node width based on the amount of chapters.

5. Set the node's height based on the computed amount of pages.

6. Give a colour to the nodes based on their level, just to make them pretty.

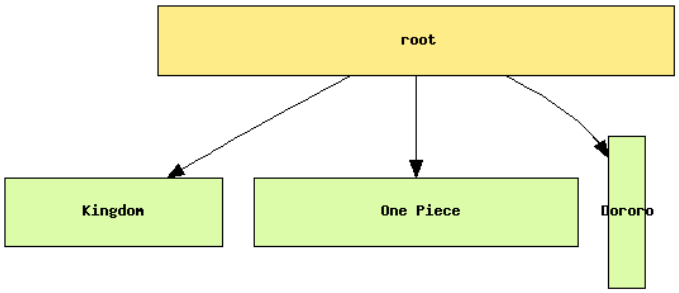

Figure 3.9: Toy comic book library

After applying this sequence of filters the area of a node corresponds to the total amount of pages in the series (Not exactly, as nodes have a minimum and a maximum size). In figure 3.9, we can see that “Kingdom” and “One Piece” have chapters of the same length, while “Dororo” only has longer chapters. Also “Dororo” only has a few chapters while the 2 other series have many. If we assume that the owner of the library thinks about reading a complete series during his one week long vacation, he should pick “Dororo” but plan longer reading sessions than for the other 2. The reshaping of the nodes is unsurprisingly based on the polymetric views [7] approach.

Chapter 4

Implementation

In this chapter we will discuss the implementation of several filters and parsers used in GraphTool.

4.1 Filters

We will discuss the implementation of filters of different categories. The categories considered here are based on the effect produced by the filters. We distinguish between filters computing hidden information based on node attributes or graph structure, filters grouping nodes together, filters hiding nodes deemed insignificant, and filters "dressing" nodes, modifying their visual representation.

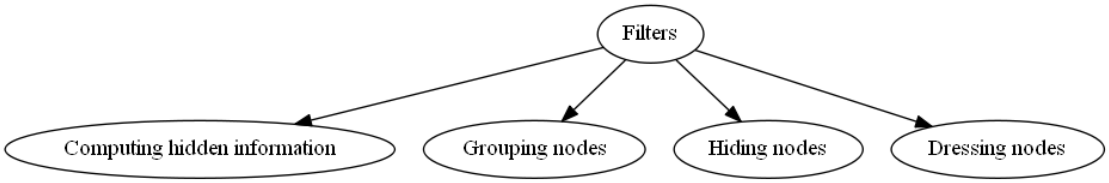

Figure 4.1: Filter categories

4.1.1 Filters grouping nodes

The most interesting example of this kind of filter is probably the one assigning an equivalence class to each node and summarising the graph by preserving equivalence classes and relations between those classes but dropping the nodes themselves. This filter is described by algorithm 2. The steps are roughly the following:

1. Each node receives an equivalence class identifier, named "hash" in algorithm 2.

2. All nodes but one per class are added to a removal stack. The preserved node now represents its entire class.

3. Nodes on the stack are removed one after another, adding their edges towards another class to the remaining node (if the edge is not already present) right before removal.

Algorithm 2 Combine

```
1:  procedure COMBINE_NODES(graph, compute_hash)
2:      hashes ← Map()
3:      reversed ← Map()
4:      for node in graph do                ▷ computing hash for each node in graph
5:          h ← compute_hash(node)
6:          hashes.put(node, h)
7:          if reversed.contains(h) = false then
8:              reversed.put(h, node)
9:      groups ← Map()
10:     removal ← Stack()
11:     for node in graph do                ▷ Finding groups of similar nodes
12:         if groups.containsKey(hashes.get(node)) then
13:             removal.add(n)
14:             rep ← reversed.get(hashes.get(node))
15:             for edge in node.enteringEdges() do
16:                 temp ← reversed.get(hashes.get(edge.from()))
17:                 if rep.hasEdgeFrom(temp) = false then
18:                     graph.addEdge(temp, rep)
19:             for edge in node.leavingEdges() do
20:                 temp ← reversed.get(hashes.get(edge.to()))
21:                 if rep.hasEdgeToward(temp) = false then
22:                     graph.addEdge(rep, temp)
23:             groups.replace(rep, groups.get(rep) + 1)
24:         else
25:             groups.put(node, 1)
26:     while not removal.empty() do        ▷ Removing non-rep. nodes
27:         graph.remove(removal.pop())
```

Example Figure 4.2 shows our earlier toy dependency graph (figure 3.1), on which labels were replaced with the nodes class identifiers. The equivalence classes

are here based on the node's file extension (for files) and command name (for commands).

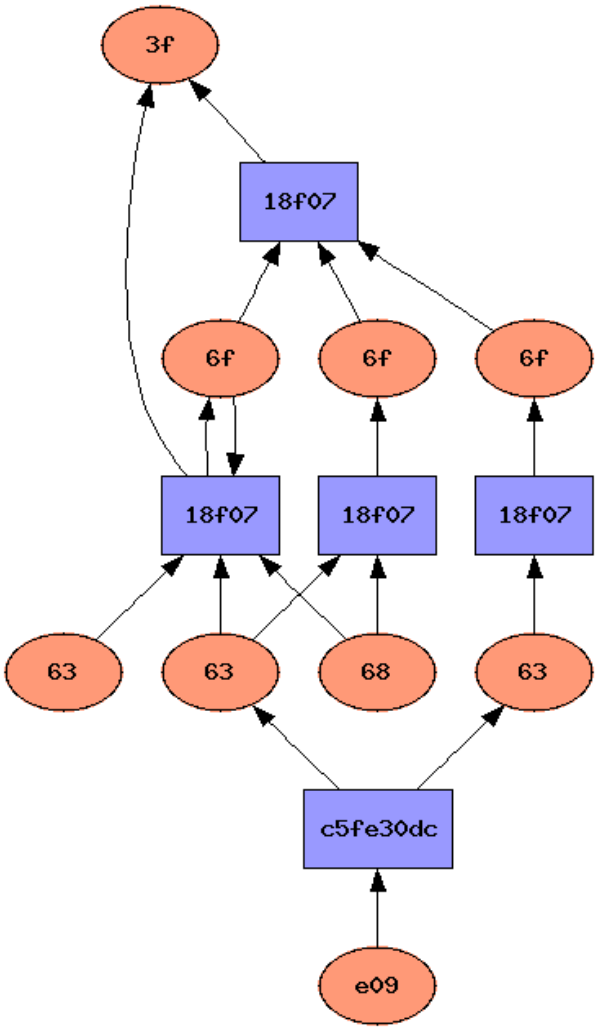

Figure 4.2: Toy dependency graph with hash values as labels

Figure 4.3 shows the summary of the toy dependency graph. The hash values representing the classes are still visible for comparison with figure 4.2. The summary shows relations between those classes rather than between nodes.

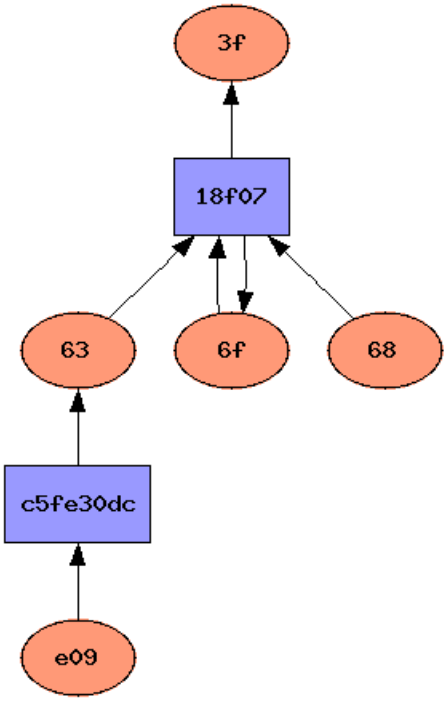

Figure 4.3: Summary of the toy dependency graph with hash values as labels

4.1.2 Filters hiding nodes

The filter described in algorithm 3 hides nodes for which the numerical value of a chosen attribute is above (or below) a certain threshold. The edges to and from the hidden edges are connected together in such a way that each edge towards a hidden node is connected to every edge from this hidden node.

Algorithm 3 Hide

```
 1: procedure HIDE(graph, attribute, value, valid)
 2:     stack ← Stack()
 3:     for node in graph do
 4:         if valid(n, attribute, value) then
 5:             stack.push(node)
 6:     while stack.empty() = false do
 7:         node ← stack.pop()
 8:         for edge1 in node.enteringEdges() do
 9:             for edge2 in node.leavingEdges() do
10:                 graph.addEdge(edge1.from(), edge2.to())
11:         graph.remove(node)
                                                        ▷ hide above
12: procedure VALID(node, attribute, value)
13:     return node.attribute(attribute) > value
                                                        ▷ hide below
14: procedure VALID(node, attribute, value)
15:     return node.attribute(attribute) < value
```

4.1.3 Filters dressing nodes

Algorithm 4 is used to assign a width, height, or colour to every node in the graph. The assigned value for a node n is based on the numerical value of an attribute a, or more exactly the value and the range of values for this attribute. The information provided by this filter allows to easily compare nodes with each other. However, for some graphs, it is impossible to generate a satisfying visualisation. The only concrete example of such a graph involved a nodes with many edges towards very large rectangular nodes. Of course this limitation is not exactly related to our work, but to the dot layout algorithm.

Algorithm 4 Weighted nodes

```
1: procedure WEIGHTED_NODE(graph, attribute, min, max, apply)
2:     maxVal ← MIN_VALUE
3:     for node in graph do
4:         if node.attribute(attribute) > maxVal then
5:             maxVal ← node.attribute(attribute)
6:     multiplier ← max/maxVal
7:     for node in graph do
8:         apply(node, min, max, multiplier, attribute)
9: procedure APPLY_WIDTH(node, min, max, multiplier, attribute)
10:    node.attribute("fixed_size", true)
11:    node.attribute("width", min + node.attribute(attribute) ∗ multiplier)
12: procedure APPLY_HEIGHT(node, min, max, multiplier, attribute)
13:    node.attribute("fixed_size", true)
14:    node.attribute("height", min + node.attribute(attribute) ∗ multiplier)
15: procedure APPLY_COLOR(node, BLACK, WHITE, multiplier, attribute)
16:    node.attribute("style", "filled")
17:    c ← asColor(node.attribute(attribute) ∗ multiplier)
18:    node.attribute("fillcolor", c)
```

In algorithm 4, the apply parameter is meant to be one of the "*apply_something*" procedures below the main procedure of the block.

4.2 Parsing

In GraphTool, parsing textual data is necessary in 4 occasions: parsing formulae, graphs stored in dot language, graphs stored as custom git log format, and graphs stored as the output of the *tree* command. The general process is similar for the first 3 data types:

1. The textual data is stored in memory as a string of characters.
2. The scanner goes through each character to build a sequence of tokens. Each token has a type and a value.
3. The parser interpretes the sequence of tokens and returns a graph (dot and git log files) or a numerical value (formulae).

This parsing process does not rely on any preexisting library. Using a tool such as ANTLR simply would've... killed part of the fun. That is about all there is to say about this choice.

Figure 4.4 represents the parsing of a very short dot file. The scanners, as well as the parsers, to a lesser extent, are heavily inspired by the j– compiler.

Figure 4.4: Illustration of the dot parsing process

4.2.1 Parsing formula

The formula parser can interprete simple formulae containing multiplications, divisions, additions, substractions and the modulo operation. These formulae are used to produce new node attributes based on existing numerical attributes.

Example In a git graph, the amount of lines added and the amount of lines removed by a commit are directly available as the "plus" and "minus" attributes. The sum of these values is the amount of changes made by the commit. GraphTool has a filter allowing the user to create a new node attribute with the evaluation of a formula as its value. The existing attributes of the node can be used as variables (or rather constants) in the formula.

Scanner The scanner goes through each characters one after another, building tokens as pairs made of a character sub-sequence and a type. As we are parsing mathematical expressions, the types of tokens are the following:

- left (right) parenthesis
- multiplication, division, addition, substraction and modulo operators
- equals sign
- variable
- value, a number

Parser Our parser function takes as parameters a sequence of tokens, a set of variables, and a stack of values. This function returns a numerical value. A variable obviously has a name and a value. First, the type of the first token is considered:

- Value: the value is pushed on the stack, we move to the next token and make a recursive call.
- Variable: the value of the variable is pushed on the stack, we move to the next token and make a recursive call.
- Left parenthesis: we move to the next token, we make a call to the parser function with an empty value stack. The value returned by the call is pushed on the stack and we make a recursive call.
- Right parenthesis: the value stack should contain a single value, we move to the next token and return this value.
- One sign among $\{*, /, \%, +, -\}$: the corresponding helper parser function is called and its returned value is pushed on the stack. We make a recursive call.
- "END" token: the value stack should contain a single value, we return this value.

We didn't and won't describe how we check if the expression is valid, because it involves uninteresting look ahead. But we have yet to describe the helper functions mentioned in the penultimate item of the previous list. Those functions are fairly similar to each others, therefore describing only one of those seems reasonable.

Plus helper parser function At this point, the current token should be the + sign. We move to the next one and consider its type:

- Value: the value is pushed on the stack, we move to the next token.
- Variable: the value of the variable is pushed on the stack, we move to the next token.
- Left parenthesis: we move to the next token, we make a call to the parser function with an empty value stack. The value returned by the call is pushed on the stack.
- Minus sign: the minus helper parser function is called and its returned value is pushed on the stack. The minus sign in this case is the unary negation of the expression beginning with the next token.

Now, we have (at least) 2 values on the value stack, we could pop those 2 values and return their sum. But we need to deal with the precedence of operators first. The next token might be the sign of an operator with higher precedence. If it is the case, we make a call to the corresponding helper parser function and push the returned value on the stack. We now pop the 2 values on the top of the stack and return their sum.

Example With a variable A equal to 2 and an empty value stack, the expression $(40 + A)$ is evaluated by our parser according to the following table. With the "END" token and a single value on our stack, the expression $(40 + 2)$ is evaluated to 42.

Expression	**Variables**	**Value stack**	**First token type**
(40 + A)	{A=2}	S1={}	Left parenthesis
40 + A)	{A=2}	S1={}, S2={}	Value
+ A)	{A=2}	S1={}, S2={40}	+ sign
A	{A=2}	S1={}, S2={40}	Variable
<END>	{A=2}	S1={}, S2={2, 40}	END
<END>	{A=2}	S1={42}	END

4.2.2 Parsing dot

The dot language allows to store directed and undirected graphs and subgraphs. The grammar defining this language won't be discussed but is available on the GraphViz website [9]. Figure 4.4 was a representation of the dot parsing process in GraphTool. The scanner is still heavily inspired by the j– compiler and the parser simply adds elements to the graph object as they come.

4.2.3 Parsing git

It is possible to consult the recent commit history by using the "git log" command, but this simplest call produces some irrelevant information, fails to provide some of the information we need, and is mostly intended to be human readable. Therefore the input data we want to feed GraphTool should be generated with such a command:

```
git log --all --date-order --pretty="%H|%P|%d" --numstat > commits.txt
```

The output of the command, stored in a text file, contains a list of commits. Each commit contains the identifier of the commit, the identifiers of the parent commits, an optional note (such as the name of a release), and the set of modified files, along with the amount of added and removed lines. The structure of a commit in this format matches the following pattern:

```
<commit id> | <parent commit 1> ... <parent commit n> | [(<note>)]

<plus> <minus> <file 1 path>
...
<plus> <minus> <file n path>
```

Chapter 5

A large dependency graph

In this chapter, we will consider a large dependency graph, containing 33580 nodes and 1168658 edges. GraphTool can parse such a graph in a reasonable time (about 1 minute). It is impossible to produce a visualisation in a reasonable time for this initial graph, at least with the layout algorithm used so far. But we can create a summary and create a visualisation of the summary. For this graph, we will only consider 1 filter, the one regrouping nodes in equivalence classes.

Classes based on node type First, we consider two classes. Nodes representing a file belong to the first and nodes representing a command to the second. We already used this summary for our toy graph (see figures 3.1 and 3.2). Figure 5.1 is the visualisation for our large graph. The result is the same as for the toy graph (if we ignore the file extensions and command names in the node labels). This summary shows us little more than the files types and commands involved.

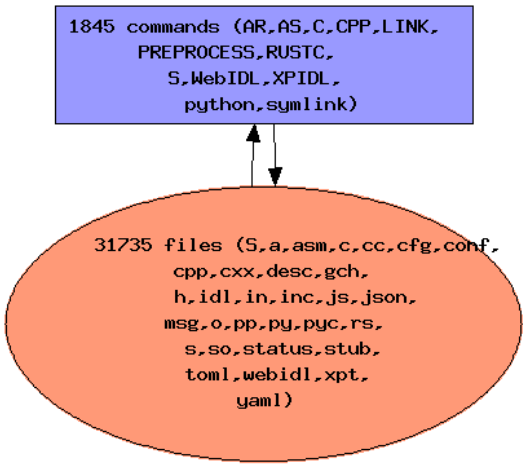

Figure 5.1: Large dependency graph summary, classes based on node type

Classes based on node depth Figure 5.2 shows the summary of our graph if equivalence classes are based on the depth of the nodes (a node depth is 0 if the node has no parent, otherwise its depth is 1 plus the smallest depth of its parents).

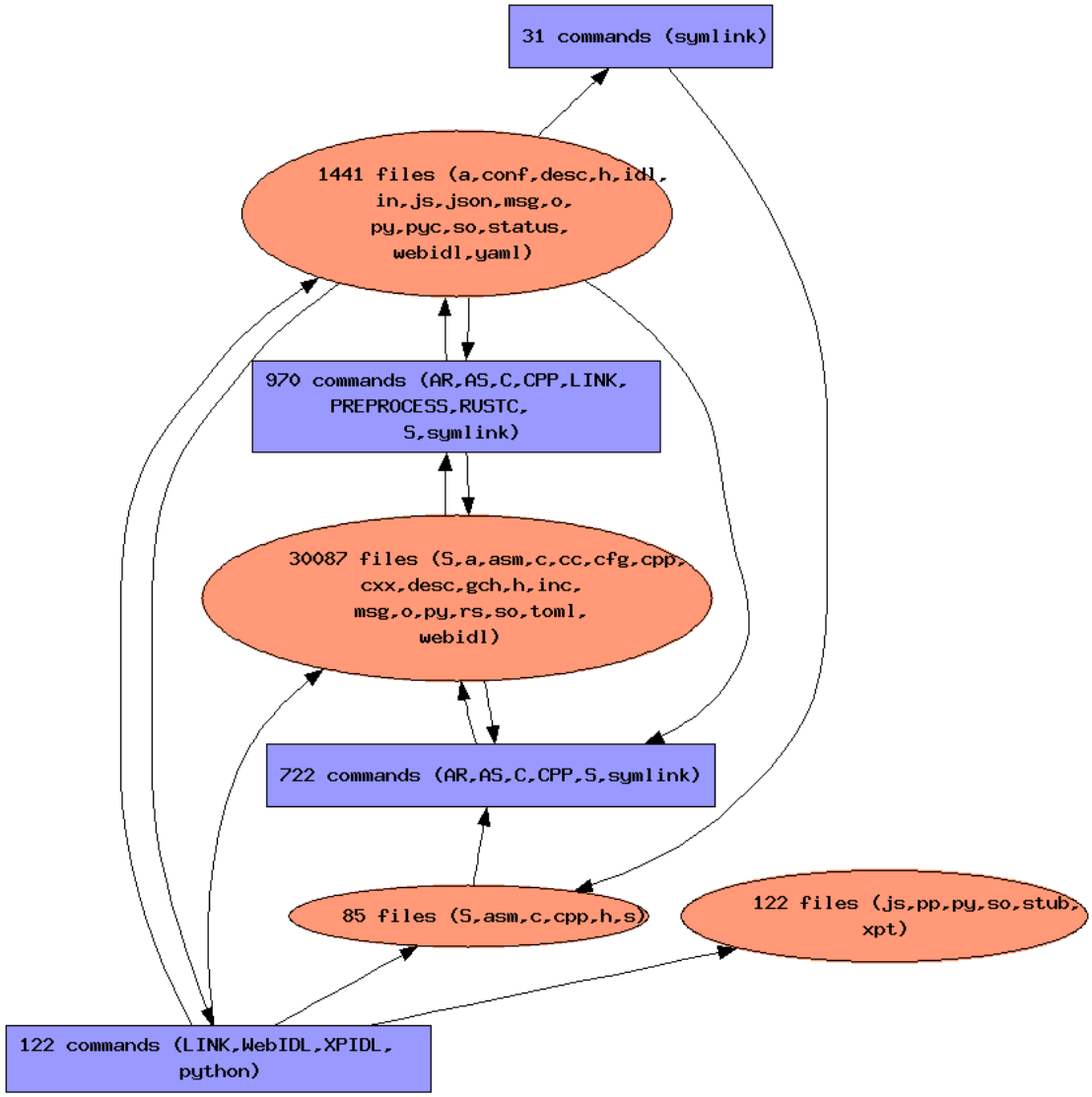

Figure 5.2: Large dependency graph summary, classes based on node depth

Classes based on node subtype We call subtype of a node the file extension of the file represented by the node (for nodes representing files) or the name of the command (for nodes representing commands). Figure 5.3 shows the corresponding summary.

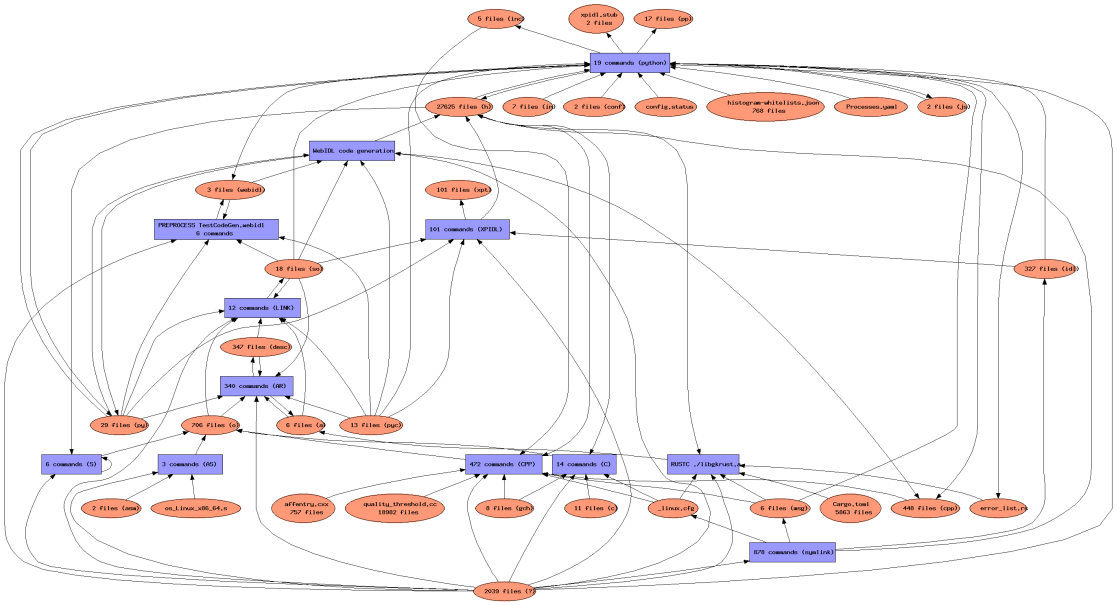

Figure 5.3: Large dependency graph summary, classes based on node subtype

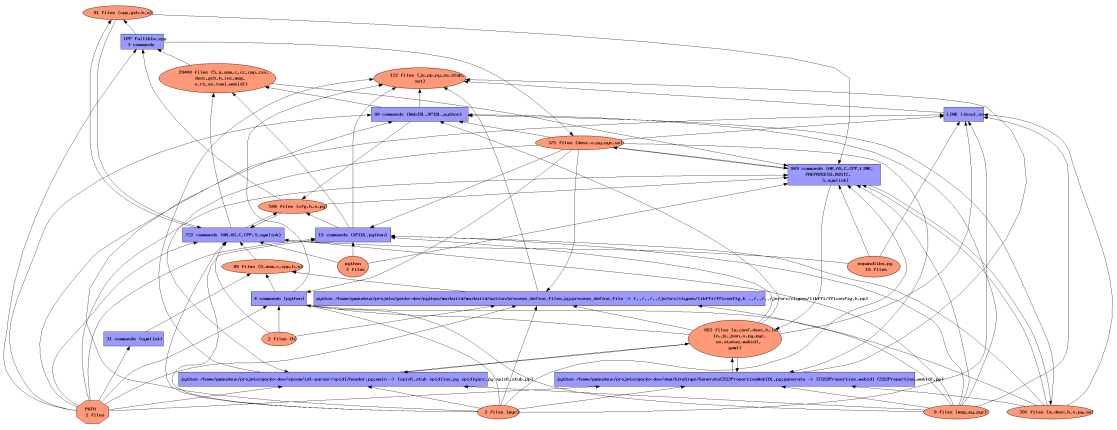

Figure 5.4: Large dependency graph summary, classes based on the depth of the parent nodes

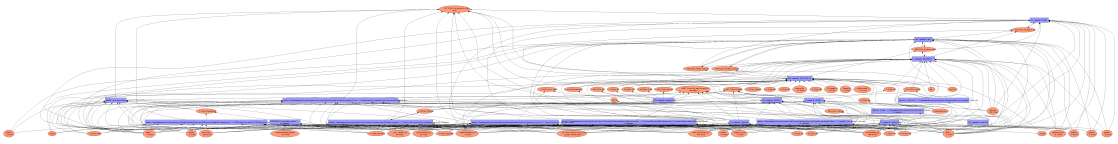

Figure 5.5: Large dependency graph summary, classes based on the subtype of the parent nodes

Classes based on parent depth or subtype Figures 5.4 and 5.5 are summaries in which the classes are based on parent nodes. These 2 summaries are rather messy but could be reduced further by adding a condition when assigning classes. Figures 5.6 and 5.7 are parts of figure 5.5.

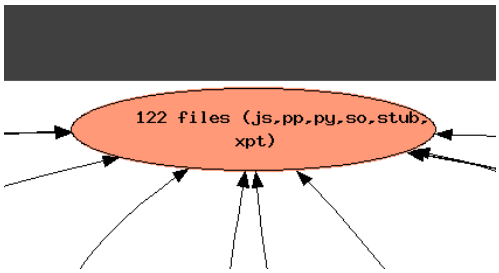

Figure 5.6: Large dependency graph summary, classes based on parent subtype, top of the visualisation

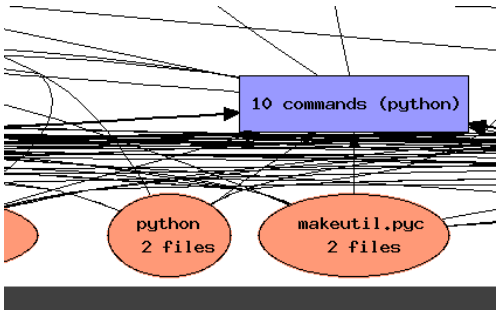

Figure 5.7: Large dependency graph summary, classes based on parent subtype, bottom of the visualisation

Valuable summaries Figures 5.2, 5.3 and 5.8 are valuable summaries, preserving part of the information of the original graph, condensed in only a few nodes. Producing those different summaries is possible by only repeating the application of 1 filter with different parameters. The application of the filter taking only a fraction of the time necessary to repeat the whole process.

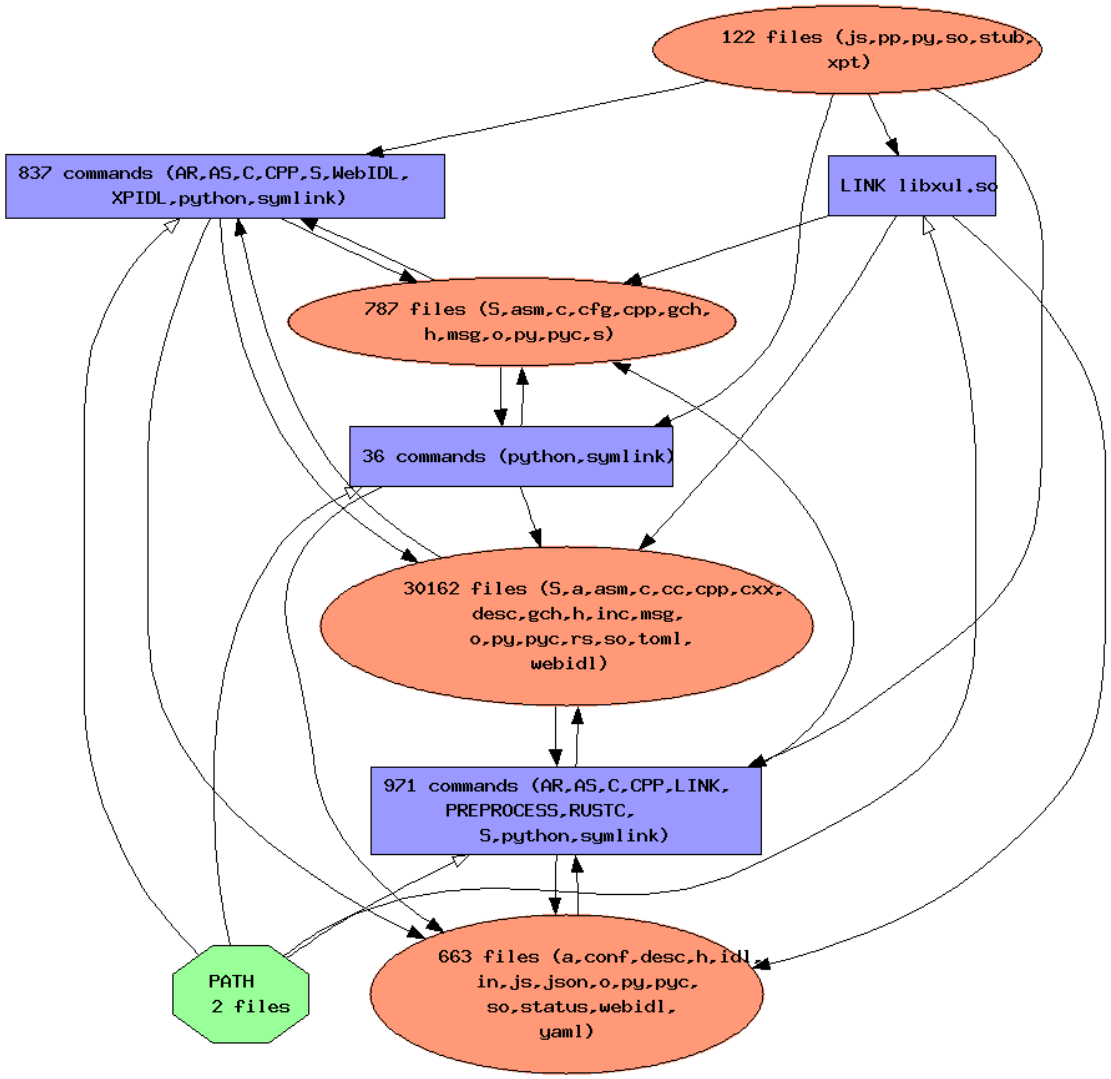

Figure 5.8: Large dependency graph summary, classes based on maximum parent depth

Chapter 6

Conclusion

In this conclusion, we will state some of the strengths and weaknesses of GraphTool, in order to identify the desired qualities of such a tool. Then we will evaluate to which degree we succeeded in fulfilling our initial goal.

Strengths of GraphTool GraphTool has room for improvement, but its filter stack carries out its initial purpose. It allows to easily try different combinations of filters and immediately see the resulting visualisation. It provides rather generic filters that can be used to produce different summaries for an initial acyclic graph. It handles several input formats, produces graphs representation in a generic and convenient format, and exports PNG visualisations. The computations performed in GraphTool are also reasonably efficient.

Weaknesses of GraphTool GraphTool only allows limited and indirect interactivity with the graphs. It would be convenient to be able to directly edit a summary with just a few clicks on its visualisation. Unfortunately this feature is not available in GraphTool. The available filters are not necessarily applicable to graphs containing cycles, even though some summaries are not acyclic. It is of course a limitation of the current state of GraphTool, it could be extended to handle any directed graph. Some filters need to perform some computation relying on the value of some node attribute, for example. But it would be convenient to be able to feed far more complex rules or conditions to GraphTool through a short piece of code at run-time.

How successful have we been? GraphTool is far from perfect but it fulfilled its initial goal. It produces various and satisfying(ish) summaries of large acyclic graphs such as dependency graphs and trying variations of a summary is by far less painful than it would be without manipulation of the filter stack.

Bibliography

[1] James Abello, Frank Van Ham, and Neeraj Krishnan. Ask-graphview: A large scale graph visualization system. *IEEE transactions on visualization and computer graphics*, 12(5):669–676, 2006.

[2] Wikipedia community. Directed acyclic graph. `https://en.wikipedia.org/wiki/Directed_acyclic_graph/`. [Last accessed on May 13th, 2019].

[3] Weiwei Cui, Hong Zhou, Huamin Qu, Pak Chung Wong, and Xiaoming Li. Geometry-based edge clustering for graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1277–1284, 2008.

[4] Ozan Ersoy, Christophe Hurter, Fernando Paulovich, Gabriel Cantareiro, and Alex Telea. Skeleton-based edge bundling for graph visualization. *IEEE transactions on visualization and computer graphics*, 17(12):2364–2373, 2011.

[5] Danny Holten and Jarke J Van Wijk. Force-directed edge bundling for graph visualization. In *Computer graphics forum*, volume 28, pages 983–990. Wiley Online Library, 2009.

[6] Danai Koutra, U Kang, Jilles Vreeken, and Christos Faloutsos. Summarizing and understanding large graphs. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 8(3):183–202, 2015.

[7] Michele Lanza. Codecrawler-polymetric views in action. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 394–395. IEEE Computer Society, 2004.

[8] Alexandru Telea and Ozan Ersoy. Image-based edge bundles: Simplified visualization of large graphs. In *Computer Graphics Forum*, volume 29, pages 843–852. Wiley Online Library, 2010.

[9] Unknown. Directed acyclic graph. `https://graphviz.gitlab.io/_pages/doc/info/lang.html`. [Last accessed on June 3rd, 2019].

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**