University of Queensland

Mathematics Honours Thesis

Graph Theoretic Analysis of Relationships Within Discrete Data

Author:
Ivan Lazar Miljenovic

Supervisor:
Professor Peter Adams

Second Reader:
Andriy Kvyatkovskyy

3 November, 2008

Abstract

Graph theory is a useful and powerful mathematical tool that can be utilised to analyse the relationships between discrete data points. However, there is a lack of general-purpose computing libraries to help people automate analysis of large data sets. This thesis covers the development of a library designed to fulfill this hole, as well as a sample application using this library to analyse the static code complexity of source code.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

We live in what is commonly referred to as the "Information Age". No single definition of this phrase is agreed upon and its origin is unknown, but Marvin Kranzberg stated that:

> ... the accumulation, manipulation, and retrieval of data by computerized electronic devices and their application to many facets of human life, [and] computer developments are transforming industry and society to produce a new "Information Age". [16, page 35]

This summarises what many perceive as being one of the critical factors in the development of the Information Age: the use of technology to more widely disseminate and utilise information. This information comes in many forms and from many sources. Broadly speaking, we can categorise information — which in its raw unprocessed form is referred to as *data* — into two types: about continuous systems and discrete items. However, whilst many tools and utilities abound for processing data that represents continuous systems (e.g. matrices for large scale computations), there is a lack of general-purpose utilities that can be used to examine and analyse the relationships inherent in discrete data.

Discrete data can come in many forms: authorship lists of scientific papers, address books, programme source code, etc. However, whilst most people focus on the contents of these data sources, it is quite often the *relationships* between individual datums that provide the more interesting source of information. For example, consider people's address books. By following the relationships from one person to another by their presence in the corresponding address book, the structure of the underlying social networks present in the initial data source become apparent.

The analysis of data is rooted deeply in mathematical theory. For example, mathematical tools such as matrices are often used to process large amounts of raw numeric data, and statistical techniques are used on data collated from experiments, etc. As such, when considering how to analyse relationships in discrete data it is natural to look for a branch of mathematics that fits. In this case, the area of combinatorics known as *graph theory* is eminently suitable, as it deals explicitly with mathematical structures known as *graphs* that are used to model the very kind of relationships that we wish to analyse. For a brief overview of graph theory (and more specifically the definitions used throughout this thesis), please see Section 2.

1.1 Focus of this Thesis

This thesis aims at developing a software library to assist people in using graph-theoretic techniques to analyse and visualise the relationships within their data. As a sample application of this library, a program to analyse source code is also developed. A major focus on the library is to make it extensible: only generic analysis algorithms are included, whilst still providing sufficient access to the internals of the system to allow the person utilising the library to create new analysis algorithms.

There are many possible algorithms and analysis techniques that could be implemented in the proposed software library. To simplify matters, the library will be initially constrained at providing algorithms that would be of use for a relatively sparse directed graph (by which we mean that is has a relatively low number of relationships per data point. This matches many types of discrete data sources of interest, such as the address book and source code examples listed above.

This is by no means the first usage of graph theory in a real world setting. However, no mention of any general-purpose analysis library or technique was able to be found. Some similar applications from which ideas were taken are:

Network Theory/Analysis Analyses various types of networks (technical, social, hierarchical, etc.) as graphs. Differing in terms of the focus of the analysis and in that these graphs *are* the dataset rather than a normally disregarded subcomponent of the datasets. [27]

Visualisation of source code Several tools have been developed to aid programmers in visualising the *Call Graph* of the functions in source code, however the only type of analysis normally used is with determining the heavily-used functions from profiling information. [30]

Graph-based optimisations Several programming language compilers/interpreters use a limited amount of graph theory to optimise the programs during compilation. [3]

2 Graph Theory

This thesis will predominantly follow standard graph theory terminology, with one exception: rather than using the term *vertex* to denote the data points, we shall use the more computer science-oriented term *node*, as this is the term used by the graph library (see Section 3.3.1). We will still refer to a set of nodes/vertices as V however. Other minor differences in terminology are explained below.

2.1 Graphs

A graph is a set of nodes V along with a set of *edges* E, where an edge is a tuple (v_1, v_2) with $v_1, v_2 \in V$. Thus, a graph G is defined as:

$$G = (V, E), \qquad \text{where} \quad E \subseteq V \times V \tag{1}$$

Note that here we do not require the commonly-used restriction of a graph being non-empty (i.e. containing at least one node), as several of the algorithms developed in Section 4.4 rely on allowing empty graphs

By default, we assume that all graphs are *directed*: that is, if $(v_1, v_2) \in E$, then we say that there exists an edge in our graph from v_1 to v_2. Undirected graphs can be simulated by enforcing the requirement that

$$\forall\, (v_1, v_2) \in E,\ v_1 \neq v_2 \rightarrow (v_2, v_1) \in E \tag{2}$$

Note that with this definition, loops are *not* duplicated. For an example of this, please see Figure 2.1.

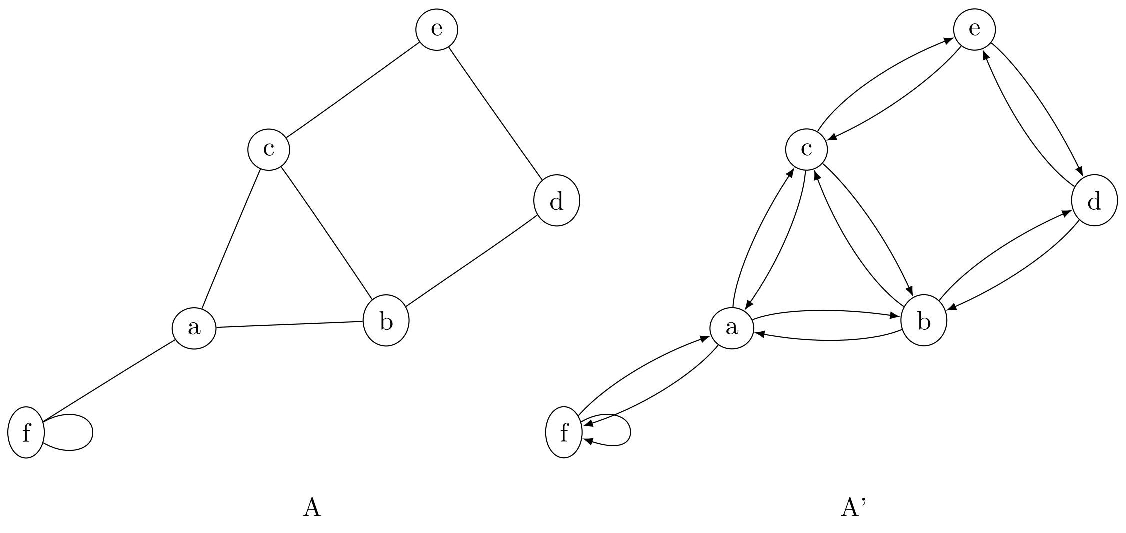


Figure 2.1: Simulating an undirected graph A using a directed graph A'

If we have have an edge $(v, v) \in E$, then this is called a *loop*. A graph with no loops and no repeated edges is called *simple*. In a directed graph, if we have a node r such that $\forall\, v \in R \setminus \{r\}$, $(v, r) \notin E$, then we call r a *root* of the graph (note that we allow the root node to have a loop on itself). Similarly, a node l is called a *leaf* if $\forall\, v \in R \setminus \{l\}$, $(l, v) \notin E$. An *isolated vertex* is both a root and a leaf of the graph.

An important type of graph is the *complete graph on n nodes*, denoted K_n. This is a simple undirected graph where each pair of distinct nodes has an edge between them, i.e. $\forall v_i, v_j \in V,\ v_i \neq v_j \rightarrow (v_i, v_j) \in E$.

2.2 Subgraphs

A graph $G' = (V', E')$ is called a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. Normally, however, we only consider subgraphs that have some underlying property. A common classification of subgraphs is whether or not they are *maximal*. A subgraph is maximal if it is not possible to add any more vertices from the set V to V' (along with any allowed edges) and maintain that subgraphs property. For example, it is possible to have a complete graph K_n as a subgraph of G. If this is the case, then all complete graphs of lower order (i.e. $K_{n-1}, K_{n-2}, \ldots K_1$) will also be subgraphs of G. However, if K_n has the largest value of n for all complete subgraphs in G, then it is the maximal complete subgraph, often called a *clique*. Note that it is possible to have multiple maximal subgraphs: we just require the difference of the two node sets to be non-empty (i.e. for each pair of maximal subgraphs, there is at least one vertex that is in each subgraph that isn't in the other). Another property of subgraphs is whether or not they are *spanning*, that is if $V' = V$.

A *path* is an ordered list of distinct nodes $v_1, v_2, \ldots, v_n$ such that for each v_i in the path (with $i < n$), there is an edge from v_i to v_{i+1} (i.e. $(v_i, v_{i+1}) \in E$). A *cycle* is a path with $v_1 = v_n$ (note that all other nodes must be distinct). Note that all cycles are thus paths, and all cliques are composed of cycles (if a clique has n vertices, then it has cycles of length $2, 3 \ldots n$). Also, when considering paths, cycles and cliques, we often require the graph to be simple to avoid duplicates and confusion (e.g. does a clique require all nodes to have loops?).

For a graph G, a subgraph G' is called *connected* if for any two distinct nodes in in G', there exists a path between the two (treating directed graphs as undirected). Furthermore, it is a *connected component* (or merely *component*) if it is a maximal connected subgraph of G. For directed graphs, a *strongly connected component* is a connected component in which the direction of the edges is used (i.e. there is a directed path between any two distinct nodes); a *weakly connected component* is equivalent to the connected component of an undirected graph.

2.3 Extra terminology

In Computer Science, there is often a need for an acyclic connected graph, that is a graph with no cycles (often required to have no multiple edges as well if it's a directed graph). Such a graph is known as a *tree*, and contains exactly one root. Whilst trees are often required in the literature to be undirected, the edges are normally implicitly directed from the root outwards. A collection of trees (that is multiple connected graphs with no connections between them) is called a *forest*. The *height* of a tree is the length of the longest path from the root to a leaf in the tree, and a *subtree* is a subgraph that is also a tree.

A graph *clustering* is a partition of the nodes of a graph into non-overlapping mutually disjoint subsets that contain some underlying trait. For example, if we consider a graph representing the contacts listed in an address book, then we may consider a clustering where one set of nodes contains friends, another work colleagues, a third for utilities, etc. Alternatively, we may cluster the nodes by contact name (that is, create an alphabetical ordering of all the contacts). This example shows that there may be more than one valid clustering on a graph.

3 Implementation

In this section, we examine some implementation specifics of the graph analysis library, such as what functionality it will include and how it is implemented.

3.1 Separation of Tasks

For any set of discrete datapoints whose relationships we want to analyse, there are four main stages (see also Figure 3.1 for a visual representation) to go through before raw data can be utilised as information:

Parse Read the data in from file, the internet, databases, etc. into a usable form.

Graph Creation Turn the parsed data into a graph to analyse.

Analysis Analyse the graph.

Reporting Report the information found from analysis to the user in an appropriate format.

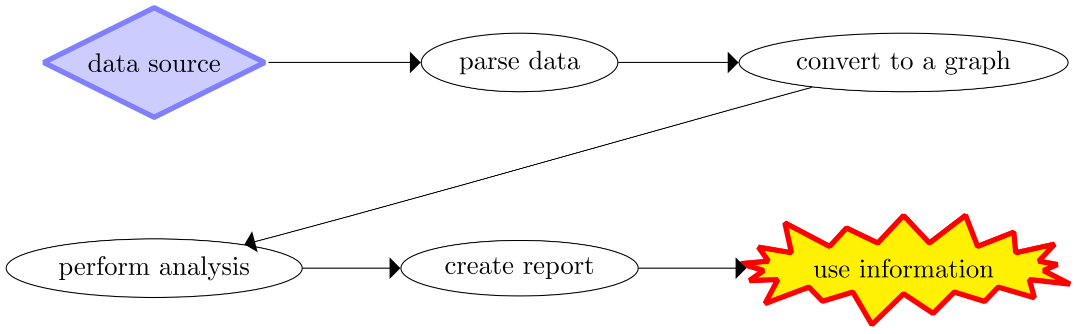


Figure 3.1: Process of converting raw data to information

As data parsing is strongly tied to the type of data that is to be parsed, the first stage above is relegated to the program that is using the library. On the other hand, graph creation is rather trivial: the calling application merely needs to indicate the sets of nodes and edges being used (as well as some possible extra information that might be of relevance to various analysis algorithms). As such, a function to create a graph in the required format from the supplied nodes and edges can be implemented in the library.

The library contains a number of algorithms that would be seen as being multi-use (i.e. suitable for a wide range of data sources). However, in many cases per-use algorithms will be required. As such, an "open" implementation format will be utilised where users of the library are able to implement their own custom algorithms and utilise them as well.

For reporting the graphs, people will quite often want to view the results in different formats: plain text, LaTeX documents, HMTL web pages, custom data types for use in other applications, etc. As such, the reporting format will also be "open": for those who want to produce a document of some kind, a simplified document structure will be provided with extensible output support; otherwise, the user can utilise the results from the analysis directly.

3.1.1 Implementation of tasks

When actually developing the software, these tasks were split into the GRAPHALYZE library and the SOURCEGRAPH programme. For more information on these pieces of software, please see Section 4 and Section 5 respectively.

3.2 Implementation Details

The chosen programming language for implementing the software is the purely functional language *Haskell* [22]. As a functional language, it looks more “mathematical” than standard imperative languages such as C, Fortran, etc. This arises because — like other functional languages — Haskell’s model of computation arises from Alonzo Church’s λ-Calculus rather than Turing Machine (though they are both equivalent to each other). Part of the reason for choosing Haskell was the available libraries in Section 3.3.

3.2.1 Haskell Syntax

Haskell syntax comes in two main yet related parts: type signatures and the actual funtion definitions. Due to Haskell’s strong static typing using the Hindley-Milner type system, explicit type signatures are usually optional as they can be derived from the actual function, but often serve as extra documentation. Type signatures also often look mathematical. To compare mathematical syntax to Haskell syntax, let us consider the function that calculates the Eulerian distance from the origin, with (3) showing the formal mathematical definition (though (4) has the more common usage):

$$\begin{aligned} d_E : \mathbb{R} \times \mathbb{R} &\to \mathbb{R}^+ \\ (x, y) &\to \sqrt{x^2 + y^2} \end{aligned} \tag{3}$$

$$d_E(x, y) = \sqrt{x^2 + y^2} \tag{4}$$

and compare this to the Haskell version found in Listing 3.1:

```
1  distE       :: (Double,Double) → Double
2  distE (x,y) = sqrt (x^2 + y^2)
3
4  -- Or in what's known as _Curried form_
5  distE'      :: Double → Double → Double
6  distE' x y = sqrt (x^2 + y^2)
```

Listing 3.1: Distance from origin

Note that in Haskell, `Double` covers floating point numbers (with the precision being dependent on the computer architecture), but we can’t restrict it to non-negative numbers like we do with $\mathbb{R}^+$ in (3). It should be obvious that line 1 of Listing 3.1 is almost identical to the statement of the domain, etc. in (3), but that the actual function syntax is more similar to “normal” function usage as in (4).

The *Curried form* that the comment in line 5 alludes to is a technique (named after Haskell Curry, which the Haskell language is also named after) where a function that takes in an n-tuple of arguments and converts it to an equivalent form where it is used as a chain of functions, each of which takes one argument. For example, the function `distE' 2` will take in a value for the `y` value, and return the distance from the origin for the position $(2, y)$ (i.e. it is equivalent to the function $f(y) = d_E(2, y)$).

As an example of a commonly used Haskell function, let us consider the definition of the `map` function, which applies a supplied function to every item in a list of values:

```
1 map          :: (a → b) → [a] → [b]
2 map _ []     = []
3 map f (x:xs) = f x : map f xs
```

Listing 3.2: The `map` function

Note that whilst we have lined up the two equal signs and the type signature, this is purely for readability purposes. This example demonstrates the often-used technique of *pattern matching* on a data structure: the first second line matches for empty lists denoted by the square brackets (and using an underline to denote that the value of the first parameter isn't utilised) whilst the last line matches on a non-empty list, which is recursively defined as an element (in this case `x`) appended onto another list (denoted by `xs`).

The `map` function also demonstrates two other common traits of functional languages: the use of higher-order functions (where functions can – and often do – take other functions as parameters) and usage of recursion as a looping parameter. Compared to other functional languages, however, Haskell is also *pure*, which means that — like variables in mathematical functions — the value of parameters to functions are immutable. This example also highlights one major difference between Haskell and mathematical syntax: the low usage of bracketing. Haskell code takes full use of function composition, Currying, etc. and as such brackets are generally only used for tuples and occasional grouping: even the non-Curried example in Listing 3.1 has brackets on the left hand side only because it takes in a tuple (in this case just a pair) as its only parameter, not because the brackets denote the function arguments.

Another attribute of Haskell compared to other languages is that it features *non-strict semantics*, where strict languages require all function parameters to be fully evaluated before the function can use them. Most implementations of Haskell implement this as *lazy evaluation*, where values are only evaluated when they need to be and as much as they need to be. Due to this, it is possible to implement infinite data structures in Haskell, such as the following:

```
1    -- The list of all positive integers.  The Integer data type allows
         ...infinite
2    -- precision integers, as opposed to the fixed size Int data type.
3    positiveInts :: [Integer]
4    positiveInts = [1..]
5
6    -- Use a list comprehension to list all positive even integers.
7    evenInts :: [Integer]
8    evenInts = [ n | n ← positiveInts, even n ]
```

Listing 3.3: Infinite lists in Haskell

Line nine above uses *list comprehensions* — the syntax of which are based upon mathematical set comprehensions — to create the list of all positive even integers, using the predicate `even`. Haskell can utilise infinite data structures because they are never fully evaluated: if we try and find the sum of all the positive integers using `sum positiveInts` and then print the result, the program will eventually terminate due to a lack of computer memory. Note that this is because we are evaluating a variable that is then kept in memory: if we instead wanted to evaluate the list without using an extra variable (that is, `sum [1..] :: Integer`), the program won't run out of memory as elements of the list are discarded as soon as their value is added to the running sum).

Haskell libraries/applications are commonly split up into logical groupings known as *modules* with one module per file. The standard — but by no means required — mapping from modules to files is that the module $A.B.C$ is found in the file — relative to the root of the source directory structure — *A/B/C.hs* (where "X/" denotes the directory "X"). The extension ".hs" denotes Haskell source code; alternatively, it can be ".lhs" if the module is coded using *Literate Haskell*, where rather than special delimiters used to separate comments from code, delimiters are required for the code.

3.3 Haskell Libraries used

A number of free/open-source libraries were used when developing the software. The major ones of these are listed below. All of these libraries are available from the Haskell software repository website *hackageDB* [12], which is built upon the Haskell Cabal library [14].

3.3.1 The Functional Graph Library

Martin Erwig's Functional Graph Library (FGL) [4] is based upon his paper about an alternative method of computationally representing graphs [6]. Normally, graphs are stored as either adjacency matrices or an adjacency list. However, he proposed that graphs can be stored and utilised in an *inductive* fashion. In the FGL library, a graph is either:

- empty
- a graph $G = (V, E)$ extended with a node $v \notin V$ and lists of edges to and from nodes in V (and also loops on v)

This extension on a graph is called a *context*, with undirected graphs being emulated by directed graphs (see Section 2.1). A simplified view of a context is that it is a three-tuple containing the following (actually, FGL seperates out the node ID from its label, and edges can also have labels):

1. A set of predecessor nodes P, with $P \subseteq V \cup \{v\}$
2. The node v, where $v \notin V$
3. A set of successor nodes S, with $S \subseteq V \cup \{v\}$

Typically, $v \notin S \cap P$ (i.e. if there is a loop on v, then v is either its own predecessor or successor, but not both). Thus, if we define the "&" symbol as the application of a context to a pre-existing graph (i.e. the composition function for graphs), then we have that if $G' = (P, v, S) \,\&\, G$, then $V' = \{v\} \cup V$, $\forall v' \in p, (v', v) \in E'$ and $\forall v' \in s, (v, v') \in E'$.

For example, consider the graph in Figure 3.2. One way we can represent this graph using a simplified form of FGL (where we ignore node and edge labels) as follows:

```haskell
g :: Graph
g = ([1],6,[6])   & (
    ([3,4],5,[]) & (
    ([],4,[2])   & (
    ([1],3,[2])  & (
    ([1],2,[1])  & (
    ([],1,[])    & empty)))))
```

Listing 3.4: FGL form of Figure 3.2

Note that the extra bracketing here is required because of the associativity of the `&` operator, and to ensure the types are satisfied. Figure 3.3 is a visual representation of the inductive process of building up this graph, with each box representing an intermediary inductive graph. The actual order of nodes chosen when constructing inductive graphs is arbitrary, as long as the edges listed in the contexts are already present in the graph.

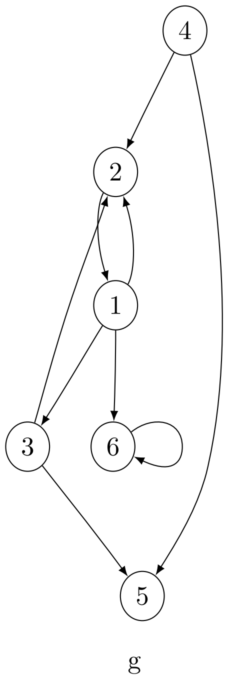


Figure 3.2: Sample directed graph

The inverse of graph composition is of course graph decomposition. FGL provides two different graph decomposition operators. The `match` function takes in a node and a graph, and returns the context containing that node (if that node is indeed in the graph) and the rest of the graph. Whilst `match` can be considered the *deterministic* inverse of `&`, the *non-deterministic* inverse of `&` is `matchAny`, which will return an arbitrary context and the remaining graph. Note that `matchAny` will throw an error if it is applied to an empty graph. [5]

Algorithms using the FGL library often utilise its inductive nature: decomposing a graph in FGL is akin to peeling an onion, taking off a context at a time and recuring until the empty graph is reached. However, most of the functions that come with the FGL library return the nodes in the relevant subgraph and not the subgraph itself, so have been re-implemented in Section 4.4. Another reason to do so is FGL doesn't store and manipulate nodes by their label, but by an arbitrary integer number used to represent that node. Thus, whenever the actual "name" of the node is required additional work is done to obtain it from the graph.

3.3.2 Haskell Parsing libraries

The *Haskell-Src* (often called *haskell-src*) library is a complete Haskell parser written in Haskell that parses code that conforms to the Haskell 98 Revised Standard [22] along with some extensions. However, the de-facto Haskell implementation — the *[Glorious] Glasgow Haskell Compiler*

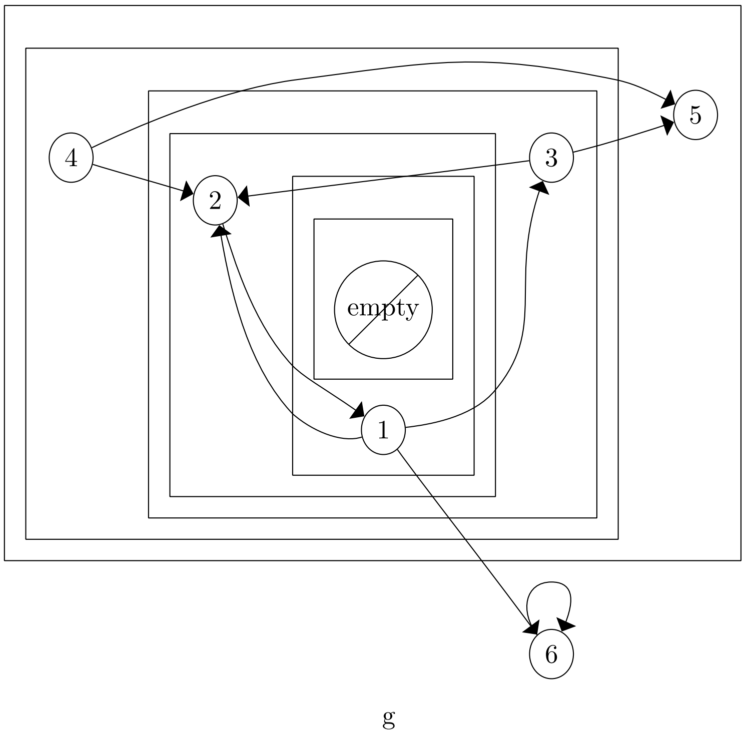


Figure 3.3: Inductive representation of Figure 3.2

(GHC) [15] — has numerous extensions available [7] that do not conform with the standard, and thus haskell-src won't parse them. However, the *Haskell-Src with Extensions* (HSE or *haskell-src-exts*) library is an extension to haskell-src that *can* parse these extensions as well as some extra ones. As such, the SourceGraph application uses this library to parse Haskell code. One major failing with haskell-src-exts is that it is poorly documented: for some reason, its *Haddock* [19] (a tool to generate documentation from annotated Haskell source code) documentation fails to build.

Another library utilised is *Cabal* [14], which stands for "Common Architecture for Building Applications and Libraries". Cabal is a library for specifying how to build a Haskell package, and is utilised by SourceGraph to determine which information about the piece of software to analyse.

3.3.3 Haskell graphviz library

It is arguable that the *GraphViz* [9] program is *the* method for computationally drawing graphs. However, whilst several Haskell libraries for GraphViz are available, none fulfilled all the criteria required for Graphalyze. The closest was the *graphviz* [24] (note the difference in capitalization) library by Matthew Sackmann, yet it had no support clustering or differentiating between directed and undirected graphs (however, unlike most of the other options available, it directly utilised FGL graphs). Thus — with Matthew's permission — I extended the library to include such support. A more thorough rundown on its features (as well as how it compares to the other available options) can be found in the announce email I sent to the `haskell@haskell.org` mailing list [21].

3.3.4 Pandoc

Markdown [10] is a lightweight markup language primarily designed to create html webpages from simple text files that are just as easily readable *as* plain text files. John MacFarlane's *Pandoc* [18] is a Haskell implementation of the original Markdown program that can not only convert (a slightly altered and extended) Markdown to html, but also supports exporting to numerous other filetypes such as LaTeX, rich text format (rtf), Open Document Text (odt), etc. as well as importing html, LaTeX, etc. Not only that, but it is a fully fledged document library as well (though output to Open Document Text via the library seems not to work, as it requires zip compression which Pandoc achieves via its executable rather than through its library).

4 The Graphalyze Library

Graphalyze is the name given to the graph analysis library (the name being a portmanteau of *Graph* and the American spelling of *analyze*), with four main areas of focus:

1. Graph creation
2. Graph analysis
3. Graph utility functions
4. Graph algorithms
5. Graph visualisation
6. Reporting results

Comparing the above to the areas listed in Section 3.1, items three and four above have been split off from the general analysis section. As such, Graphalyze can in this sense be considered as a general graph-manipulation library as well. Furthermore, item five can be considered an extension to the graphviz library (see Section 3.3.3).

Following the standard trend of Haskell library licensing (where libraries are released under a permissive free software license), Graphalyze is distributed under a 2-Clause BSD License (also known as the FreeBSD License or the BSD-2 License), which can be found in Appendix A. Graphalyze is available from hackageDB at:

`http://hackage.haskell.org/cgi-bin/hackage-scripts/package/Graphalyze`

with the source repository at:

`http://code.haskell.org/Graphalyze`

At the time of writing, the latest available version is 0.4. As the source code is publicly available, this thesis will not include a complete copy. However, we shall examine interesting snippets.

4.1 Graph Creation

The Graphalyze library utilises two different types of data for the given data source: the graph representing the relationships in that data source, and — if the relationships are directed — any expected root datums. For the sake of simplicity, edges aren't allowed to have labels attached to them.

```
-- We use graphs with node labels of type 'a' and no edge labels.
type AGr a = Gr a ()

-- We store a graph, and any expected roots.
data GraphData a = GraphData { graph        :: AGr a,
                               wantedRoots  :: [LNode a]
                             }
```

Listing 4.1: Graph data stored by Graphalyze

Here, `Gr` is a basic tree-based graph datatype, and `LNode` represents a labelled node.

To create the graph, the user needs to provide information on the relationships inherent in the data source. We require a list of data points, a list of relationships, any expected root nodes and whether or not the graph is directed. Note that if `directed` is true, then the value of `roots` is ignored.

```
data ImportParams a = Params { dataPoints    :: [a],
                               relationships :: [(a,a)],
                               roots         :: [a],
                               directed      :: Bool
                             }
```

Listing 4.2: Data to import into Graphalyze

The actual graph creation function `importData` then assigns an integer value to each data point (as this is what FGL uses to identify each node), converts the relationships to (label-less) edges and creates the graph. A simplified version that doesn't set the root node and ignores the direction parameter is (the `Ord` parameter is required to specify an ordering on the node labels):

```
1  importData        :: (Ord a) => ImportParams a → GraphData a
2  importData params = GraphData { graph = dataGraph }
3      where
4        -- Adding unique integer values to each of the data points.
5        lNodes = zip [1..] (dataPoints params)
6        -- Create a lookup map from the data point to its integer value.
7        nodeMap = M.fromList $ map (uncurry (flip (,))) lNodes
8        -- Find the integer value for the given data point.
9        findNode l = M.lookup l nodeMap
10       -- Validate a edge after looking its values up.
11       validEdge (v1,v2) = case (findNode v1, findNode v2) of
12                             (Just x, Just y) → Just $ addLabel (x,y)
13                             _                → Nothing
14       -- Add an empty edge label.
15       addLabel (x,y) = (x,y,())
16       -- The valid edges in the graph.
17       graphEdges = catMaybes $ map validEdge (relationships params)
18       -- Construct the graph.
19       dataGraph = mkGraph lNodes graphEdges
```

Listing 4.3: Importing data into Graphalyze

Note that one advantage of the method used for graph creation — specifically for creating the graph edges in lines eleven to thirteen — is that *the relationship list can reference non-existent nodes!*. This means that if an analysis is being performed on a subset of the data source, then the user doesn't need to ensure that all the data points listed in the list of relationships are also in the list of nodes, as this function will remove those relationships that are invalid. This also applies to the root node list.

4.2 Graph analysis

At the moment, Graphalyze only considers two general-purpose analysis functions. The first of these is to analyse the roots provided when constructing the graph with those found in the graph. The `classifyRoots` function in Listing 4.4 returns three different lists of nodes:

- Those roots that are expected and roots of the graph.
- Those roots that are expected but are not roots of the graph.
- Those roots of the graph that are not expected.

```
1  classifyRoots     :: (Eq a) => GraphData a
2                    → ([LNode a], [LNode a], [LNode a])
3  classifyRoots gd = (areWanted, notRoots, notWanted)
4      where
5        g = graph gd
6        -- The expected roots
7        wntd = wantedRoots gd
8        -- The actual roots of the graph
9        rts = rootsOf g
10       -- Find those nodes that are in both lists
11       areWanted = intersect wntd rts
12       -- The nodes that are only in the expected list
13       notRoots = wntd \\ rts
14       -- The nodes that are roots but not wanted
15       notWanted = rts \\ wntd
```

Listing 4.4: Root analysis

Note that in lines twelve and fourteen, \ is the Haskell list difference function, analogous to the mathematical set difference function.

The second analysis function does not work on graphs directly, but can be used to analyse the lengths of paths, cycles, etc. in the graph. It returns the mean and standard deviation in the lengths of these lists, as well as all lists that are longer than one standard deviation away from the mean with their lengths.

```
lengthAnalysis    :: [[a]] → (Int,Int,[(Int,[a])])
lengthAnalysis as = (av,stdDev,as'')
    where
      as' = addLengths as
      ls = map fst as'
      -- statistics' is a function that returns the rounded mean and
      -- standard deviation of a list of integers.
      (av,stdDev) = statistics' ls
      as'' = filter (λ(l,_) → l > (av+stdDev)) as'
```

Listing 4.5: Analysing lengths of lists

In general, however, there are many algorithms that you might wish to apply to your data. The `applyAlg` function is the preferred way of doing so rather than applying it directly, in case the layout of the `GraphData` datatype changes in future:

```
applyAlg   :: (AGr a → b) → GraphData a → b
applyAlg f = f ∘ graph
```

Listing 4.6: Using any algorithm for analysis

It is now possible to utilise any algorithm listed in Section 4.4 or indeed defined elsewhere on the relationships in the data being analysed.

4.3 Graph utility functions

Whilst the Functional Graph Library comes with a large number of functions for graphs, there are still many generic utility functions that might be required. Furthermore, in the case of the `undir` function to convert an directed graph to an undirected one (see Section 2.1), the FGL version duplicates *all* edges, even loops, which goes against the criteria in (2). As such, a modified version can be found in Graphalyze that duplicates all non-loop edges (`gmap` applies the given function to all `Context`s in the graph):

```haskell
undir :: (Eq b, DynGraph gr) => gr a b → gr a b
undir = gmap dupEdges
    where
      dupEdges (p,n,l,s) = (ps',n,l,ps)
          where
            -- All edges, predecessor and successor.
            ps = nub $ p ++ s
            -- Take only those edges that aren't loops.
            ps' = filter (not ∘ isLoop) ps
            isLoop (_,n') = n == n'
```

Listing 4.7: Creating undirected graphs

A pseudo-inverse to `undir` can also be found for those algorithms that do not work well when edges are duplicated. This function works by making all duplicated edges successor edges only.

```haskell
oneWay :: (DynGraph g, Eq b) => g a b → g a b
oneWay = gmap rmPre
    where
      -- Remove predecessor edges that are also successor edges.
      rmPre (p,n,l,s) = (p \\ s,n,l,s)
```

Listing 4.8: Pseudo-inverse to Listing 4.7

Another criteria that some algorithms require is that the given graphs are simple. The following function removes loops and multiple edges:

```haskell
mkSimple :: (DynGraph gr) => gr a b → gr a b
mkSimple = gmap simplify
    where
      -- Remove loops
      rmLoops n = filter ((/=) n ∘ snd)
      -- Remove multiple edges
      rmDups = nubBy ((==) `on` snd)
      -- Remove loops and edges
      simpleEdges n = rmDups ∘ rmLoops n
      -- Simplify the given Context
      simplify (p,n,l,s) = (p',n,l,s')
          where
            p' = simpleEdges n p
            s' = simpleEdges n s
```

Listing 4.9: Creating simple graphs

Various other utility functions are also defined, including the `statistics'` function used in Listing 4.5 and functions to find fixed-points of the given function with initial value (note that these functions assume that the given function does indeed converge).

4.4 Graph Algorithms

The five main criteria used when writing the algorithms found in Graphalyze are:

1. They should be easy to read.
2. The algorithm should follow the graph structure.
3. Be dependent solely upon the graph structure (i.e. graph-invariant).
4. Require no outside input.
5. Where possible, to be developed from scratch.

The only exceptions to these are the two clustering algorithms in Section 4.4.3, which are not original (and one of which requires a random seed). Note also that despite Graphalyze working on graphs with no edge labels, these algorithms should work for *any* graph.

Whilst FGL comes with quite a few different graph algorithms, most of them work on the level of *nodes* rather than on graphs. For example, the connected components algorithm in Section 4.4.1.

There are three types of algorithms included in Graphalyze: Common algorithms (work on all graphs), Directed algorithms and Clustering/Collapsing algorithms.

4.4.1 Common Algorithms

Connected Components

Unlike the FGL-provided function `components`, the Graphalyze function `componentsOf` returns a list of graphs that are the connected components of the original graph.

To find each connected component, we take a node from the remaining graph, then recursively extract its neighbours until the entire component has been extracted. We then choose another node and repeat the process. Note in particular the functions `extractNode` and `nodeExtractor` in Listing 4.10: they are *corecursive*, in that each one calls the other.

```
-- The unfoldr function used here will keep calling splitComponent
-- until it returns a `Nothing' value.
componentsOf :: AGr a → [AGr a]
componentsOf = unfoldr splitComponent

-- Attempt to extract a single component, returning that component and
-- the remaining graph.
splitComponent :: AGr a → Maybe (AGr a, AGr a)
splitComponent g
    | isEmpty g = Nothing
    | otherwise = Just ∘          -- Get the type right
                  first buildGr ∘ -- Create the subgraph
                  extractNode ∘   -- Extract components of subgraph
                  first Just ∘    -- Getting the types right
                  matchAny $ g    -- An arbitrary node to begin with

-- Recursively extract all neighbours of the given context.
extractNode :: ADecomp a → ([AContext a], AGr a)
extractNode (Nothing,gr) = ([],gr) -- Reached the end
extractNode (Just ctxt, gr)
    | isEmpty gr = ([ctxt], empty) -- No graph left to decompose
    | otherwise  = -- Add this context to the beginning of the list
                   first (ctxt:) ∘
                   -- Recursively extract out the neighbours.
```

```
                      foldl' nodeExtractor ([],gr) $ nbrs
      where
        -- The neighbours of this context.
        nbrs = neighbors' ctxt

-- Helper function for extractNode to perform graph decompositions.
nodeExtractor :: ([AContext a], AGr a) → Node → ([AContext a], AGr a)
nodeExtractor cg@(cs,g) n
      | gelem n g = first (++ cs) -- Append cs to the result.
                      -- Extract all neighbours of n from g.
                    ∘ extractNode $ match n g
      | otherwise = cg
```

Listing 4.10: Connected components algorithm

Finding Cliques, Cycles and Chains

A *chain* is a maximal path $v_1, v_2, \ldots, v_n$ where for each interior node v_i, v_{i-1} is its only predecessor and v_{i+1} is its only successor (the first condition also holds for v_n, and the second condition for v_1).

The algorithms for finding cliques, cycles and chains are all similar in how they work. They operate on both directed and undirected graphs, though chains use the result of `oneWay` form Listing 4.8 as it uses both successor and predecessor edges, whereas the other two only follow successor edges. All three results are returned as ordered lists of nodes (since the corresponding graph can be easily re-constructed from this list, whereas the graphs representing connected components for example can't be re-constructed as easily). Another common characteristic is that they all require the graph to be simple, and hence first call `mkSimple` to ensure this.

The outline of the functions are conceptually similar to how `componentsOf` operate: choose an initial node, then extract out all related nodes (note that for chains, we merely follow related nodes and don't remove them from the graph) and recurse. For cliques, we find all complete subgraphs of order two which contain the given node, and try to combine them into larger complete subgraphs. Cycles and chains just follow along for as long as they can, though a cycle is only deemed valid if it does indeed end at the same node it started at.

The clique detection algorithm undergoes post-processing to remove any complete subgraphs that aren't maximal. Cycle detection has an optional post-processing stage where cycles that are part of complete subgraphs (and hence part of a clique) are removed.

4.4.2 Directed Algorithms

End Nodes

Directed algorithms are all about dealing with *end nodes*, by which we mean either root, leaf or singleton nodes (i.e. those nodes which have either no predecessors, no successors or both). We can use a generic function to find all end nodes of a given type

By "End Nodes", we refer to either root, leaf or singleton nodes (recall that we allow loops on these nodes). We can use a generic function to cater for all three scenarios. A simplified version that ignores node labels can be found in Listing 4.11, using the utility function `filterNodes` that returns all nodes in the graph that match the given predicate.

```
-- Find all end nodes of the required type.
endBy   :: (Graph g) => (g a b → Node → [Node]) → g a b → [Node]
endBy f = filterNodes (endNode f)
```

```

-- Return true if the given node is an end node of the required type.
endNode       :: (Graph g) => (g a b → Node → [Node]) → g a b →
     ...Node → Bool
endNode f g n = case (f g n) of
                  []    → True
                  [n']  → n' == n -- Allow loops
                  _     → False
```

Listing 4.11: Finding end nodes

We can use the `endBy` function to find all roots, leaves and singletons of a graph:

```
-- Find all roots of the graph.
rootsOf :: (Graph g) => g a b → [Node]
rootsOf = endBy pre

-- Find all leaves of the graph.
leavesOf :: (Graph g) => g a b → [Node]
leavesOf = endBy suc

-- Find all singleton nodes in the graph.
singletonsOf :: (Graph g) => g a b → [Node]
singletonsOf = endBy neighbors
```

Listing 4.12: Types of end nodes

Core of the Graph

The "core" of the graph G is a subgraph $G' = (V', E')$ where there are no end nodes. To obtain the core of the graph, we can recursively remove all roots and leaves of the graph (note that a singleton is both a root *and* a leaf, and thus doesn't require a special case):

```
coreOf :: (DynGraph g, Eq a, Eq b) => g a b → g a b
coreOf = fixPointGraphs stripEnds
    where
      stripEnds gr' = delNodes roots ∘ delNodes leaves $ gr'
          where
            roots = rootsOf gr'
            leaves = leavesOf gr'
```

Listing 4.13: Finding the core of the graph

where `fixPointGraphs` is a utility function that finds the fixed point of a graph-based function (i.e. it finds a graph G such that $f(G) = G$).

For an example of why the core could be a useful thing to find for a graph, consider the hierarchy of a business. If a problem is found which someone can't solve, they may pass the problem on to their superiors. This could recursively happen until it reaches the central board of the company, who then pass the problem around to each other. As such, in certain scenarios the core of a graph could be where most of the "work" in the data source is done.

4.4.3 Graph Clustering/Collapsing

Two clustering algorithms have been implemented: *Chinese Whispers* [2] and *CLUSTER* [1] (henceforth, this latter algorithm will be called *Relative Neighbourhood* to avoid confusion).

Whereas most of the more commonly used clustering algorithms require some kind of input (e.g. expected number of clusters) from the user [8], these two algorithms require no input at all (save for a random seed for Chinese Whispers). Also implemented is an algorithm to collapse a graph.

When clustering the nodes in a graph, the cluster that a given node belongs to is stored in its node label. A simple type used for purposes of explanation during this section is (using integers to represent a given cluster):

```haskell
  type Cluster a = (a,Int)
```

Listing 4.14: A simple cluster label type

Chinese Whispers algorithm

The Chinese Whispers algorithm is a non-deterministic graph clustering algorithm for undirected graphs with weighted edges. GRAPHALYZE features an adapted version of the algorithm that also works on directed graph but doesn't utilise edge weightings. Instead, weighting is increased on nodes that are part of "interesting structures" such as cliques and root nodes.

A simplistic overview of the algorithms is:

1. Assign each unique cluster label.
2. Sort the nodes into some random order. For each node in turn, it joins the most popular cluster in its neighbourhood (where popularity is defined as the sum of the node weightings in that cluster).
3. Repeat Step 2. until a fixed point is reached.

This results in an algorithm that has a runtime that is $O(|E|)$.

Since the graph is non-deterministic, it is possible that for some graphs no fixed point may be reached (with the algorithm oscillating between a few different graph clusterings). The random sorting of the nodes in Step 2. should alleviate this problem, as well as by including an implicit loop on each node such that its own weighting is taken into account when calculating cluster popularity.

A simplified version of the implementation (with equal weightings for each node) is as follows (note that `StdGen` is Haskell's standard pseudo-random number generator type):

```haskell
-- The actual Chinese Whispers algorithm.
chineseWhispers       :: (Eq a, Eq b, DynGraph gr) => StdGen → gr a b
                      → gr (Cluster a) b
chineseWhispers g gr = fst -- drop the random seed
                        -- Find the fixed point of whispering
                        $ fixPointBy eq whispering (gr',g)
    where
      eq = equal `on` fst
      ns = nodes gr
      whispering (gr'',g') = foldl' whisperNode (gr'',g'') ns'
          where
            -- Shuffle the nodes to ensure the order of choosing a new
            -- cluster is random.
            (ns',g'') = shuffle g' ns
      gr' = addWhispers gr

-- Choose a new cluster for the given Node.  Note that this updates
-- the graph each time a new cluster value is chosen.
whisperNode          :: (DynGraph gr) => (gr (Cluster a) b, StdGen)
                     → Node → (gr (Cluster a) b, StdGen)
whisperNode (gr,g) n = (c' & gr',g')
    where
      (Just c,gr') = match n gr
      (g',c') = whisper gr g c -- Choose a new cluster.

-- Choose a new cluster for the given Context.
whisper :: (Graph gr) => gr (Cluster a) b → StdGen
        → Context (Cluster a) b → (StdGen,Context (Cluster a) b)
whisper gr g (p,n,al,s) = (g',(p,n,al {whisp = w'},s))
    where
      (w',g') = case (neighbors gr n) of
                  -- Singleton node: stay in the same cluster.
                  [] → (whisp al,g)
                  -- Include the current node's cluster when finding
                  -- the most popular cluster in the neighbourhood.
                  ns → chooseWhisper g (addLabels gr (n:ns))

-- Choose which cluster to pick by taking the one with maximum sum of
-- weightings.  If more than one has the same maximum, pick one of
-- these at random.
chooseWhisper       :: StdGen → [LNode (Cluster a)] → (Int,StdGen)
chooseWhisper g lns = pick maxWsps
    where
      -- This isn't the most efficient method of choosing a random list
      -- element, but the graph is assumed to be relatively sparse and
      -- thus ns should be relatively short.
      pick ns = first (ns!!) $ randomR (0,length ns - 1) g
      -- Find the popularity of each cluster.
      whispPop = map (second length) ∘ groupElems whisp $ map label lns
      -- Find all those clusters of maximum popularity.
      maxWsps = map fst ∘ snd ∘ head $ groupElems (negate ∘ snd) whispPop

-- Make the graph suitable for the Chinese Whispers algorithm.
addWhispers   :: (DynGraph gr) => gr a b → gr (Cluster a) b
addWhispers g = gmap augment g
    where
      -- Since each node has its own unique number, we can use that as
      -- its initial unique cluster.
      augment (p,n,l,s) = (p,n,(l,n),s)
```

Listing 4.15: Chinese Whispers algorithm

Relative Neighbourhood algorithm

The Relative Neighbourhood algorithm is aimed more at discrete points with a position then an actual graph, as it clusters points based the distances between them. Since nodes in a graph don't have an explicit spatial position, at first glance it seems that this algorithm isn't applicable. However, the Haskell graphviz library (see Section 3.3.3) includes a function where the it is possible to obtain the parsed output from the GraphViz programme, including where it places each node. As such, we can use these positions to partition the graph.

Fundamental to this algorithm is the concept of relative neighbours: in a point set P, two points $x, y \in P$ are relative neighbours with distance $d(x, y)$ if there is no node $z \in P \setminus \{x, y\}$ such that $d(x, z) \leq d(x, y)$ and $d(y, z) \leq d(x, y)$. Intuitively, this means that if we draw circles of radii $d(x, y)$ centred at x and y, there are no points in the overlapping region. From all the relative neighbours in P, it is possible to construct a *relative neighbourhood graph* (RNG) by having an edge between all relative neighbours with the edge weightings being the distance between the two points. The Relative Neighbourhood algorithm then clusters the RNG, and has a runtime that is $O\left(|P|^2\right)$.

One adaptation that GRAPHALYZE uses is to use a *fuzzy distance*. Since due to rounding issues and the inability of computers to accurately store and represent real numbers, we consider all distances to be *integers*. To calculate an integral distance, we take the ceiling of the standard Eulerian distance function (see (3) and (4) for this function). This fuzzy distance still seems to meet the criteria for a metric geometry.

Rather than show the entire implementation, Listing 4.16 shows just the function that directly clusters the RNG as per the original Relative Neighbourhood algorithm. Post-processing is then done on the list of nodes produced to actually assign each node in the original graph to a cluster.

```
-- | Performs the actual clustering algorithm on the RNG.
nbrCluster   :: (DynGraph gr) => gr a Int → [[Node]]
nbrCluster g
    | numNodes == 1 = [ns] -- Can't split up a single node.
    | eMax < 2*eMin = [ns] -- The inter-cluster relative neighbours
                           -- are too close too each other.
    | null thrs     = [ns] -- No threshold value available.
    | single cg'    = [ns] -- No edges meet the threshold deletion
                           -- criteria.
    | nCgs > sNum   = [ns] -- Over-fragmentation of the graph.
    | otherwise     = concatMap nbrCluster cg'
    where
      ns = nodes g
      numNodes = noNodes g
      sNum = floor (sqrt $ fI numNodes :: Double)
      les = labEdges g
      (es,eMin,eMax) = sortMinMax $ map eLabel les
      es' = zip es (tail es)
      sub = uncurry subtract
      -- First order differences.
      -- We only need the minimum and maximum differences.
      (_,dfMin,dfMax) = sortMinMax $ map sub es'
      -- We are going to do >= tests on t, but using Int values, so
      -- take the ceiling.
      t = ceiling $ (((fI dfMin) + (fI dfMax))/2 :: Double)
      -- Edges that meet the threshold criteria.
      thrs = filter (λ ejs@(ej,_) → (ej ≥ 2*eMin)
                     && (sub ejs ≥ t)) es'
      -- Take the first edges that meets the threshold criteria.
      thresh = fst $ head thrs
      -- Edges that meet the threshold deletion criteria.
      rEs = map edge $ filter ((≥ thresh) ∘ eLabel) les
      g' = delEdges rEs g
      -- Each of these will also be an RNG
      cg' = componentsOf g'
```

```
        nCgs = length cg'
```

Listing 4.16: The Relative Neighbourhood algorithm

(Note that `sortMinMax` is a function that sorts a list and also returns the minimum and maximum elements.) One thing that should be immediately obvious is that — despite this being written in a purely functional language — the implementation of `nbrCluster` is remarkably close to the original algorithm specification [1, page 3].

Graph collapsing

It is possible to "collapse" a graph down such that all "interesting" parts of a graph (e.g. cliques, cycles and chains) are replaced with a single node. As such, collapsing a graph converts it into an underlying tree (this is *not* a spanning tree, however, as the number of nodes may be reduced), and is a way of looking at the "big picture" of what the graph represents. Note that the collapsing function doesn't work on undirected graphs, as each pair of nodes form a K_2 complete subgraph (and so the entire graph would be collapsed to a single node).

4.5 Graph Visualisation

Graphalyze includes several extra wrapper functions providing extra functionality to the graphviz library (see Section 3.3.3) as well some utility functions to assist the programmer in pretty-printing different types of lists of nodes (e.g. print cycles with an arrow between each pair of nodes, but don't insert anything between nodes when printing a path). It is envisaged that the graphviz wrapper functions will be added upstream to the graphviz library at a later stage (as it stands, the graphviz library has no functions to aid the programmer in using the GraphViz programme to create images, but just converts graphs to GraphViz's dot format; these functions in Graphalyze act as wrappers around the GraphViz programme itself).

4.6 Reporting Results

Graphalyze includes a cut-down document structure representation:

```
data Document = Doc { -- Document location
                      rootDirectory :: FilePath,
                      fileFront     :: String,
                      -- Pre-matter
                      title         :: DocInline,
                      author        :: String,
                      date          :: String,
                      -- Main-matter
                      content       :: [DocElement]
                    }

-- Elements of a document.
data DocElement = Section DocInline [DocElement]
                | Paragraph [DocInline]
                | Enumeration [DocElement]
                | Itemized [DocElement]
                | Definition DocInline DocElement
                | DocImage DocInline Location
                | GraphImage DocGraph

```

```
-- Inline document elements.
data DocInline = Text String
               | BlankSpace
               | Grouping [DocInline]
               | Bold DocInline
               | Emphasis DocInline
               | DocLink DocInline Location

-- Where to save the graph to, and what the backup text should be in
-- case the graph can't be shown.
type DocGraph = (FilePath,DocInline,DotGraph)

-- Representation of a location, either on the internet or locally.
data Location = URL String | File FilePath
```

Listing 4.17: Document representation

Note that `FilePath` is a type alias for `String` used to denote locations. With this document structure, the analysis results can be converted into a simple document relatively easily.

However, this document structure by itself can't create documents. Rather than tying the programmer to a single type of document output, GRAPHALYZE specifies a `DocumentGenerator` class, that can be instantiated to create arbitrary document types:

```
class DocumentGenerator dg where
    -- Convert idealised 'Document' values into actual documents,
    -- returning the document file created.
    createDocument :: dg → Document → IO (Maybe FilePath)
    -- The extension of all document-style files created.  Note that
    -- this doesn't preclude the creation of other files, e.g. images.
    docExtension   :: dg → String
```

Listing 4.18: The `DocumentGenerator` class

GRAPHALYZE also provides several helper functions to assist in creating documents.

4.6.1 Creating documents with Pandoc

As a sample instantiation of the `DocumentGenerator` class in Listing 4.18, GRAPHALYZE includes functions that map its abstract document data types into those used by Pandoc. Furthermore, since Pandoc supports multiple output types, by default GRAPHALYZE includes wrappers to create html, LaTeX, Rich Text Format and Markdown files, with the ability to specify others.

4.7 Using Graphalyze

GRAPHALYZE is a fully documented library using the Haddock [19] documentation generator. Furthermore, even though it's split into different modules, all of these modules — except for the Pandoc document generator module — are re-exported by the overall `Analysis` module. As such, to use GRAPHALYZE in your Haskell project, all you need to include in your import list is:

```
import Data.Graph.Analysis
-- To use Pandoc to create documents
import Data.Graph.Analysis.Reporting.Pandoc
```

Listing 4.19: Importing GRAPHALYZE

Despite being a relatively new library (the first release being on 29 September 2008), there has already been at least one other person interested in using GRAPHALYZE for their own project [31].

5 The SourceGraph Programme

SourceGraph is an application that utilises the Graphalyze library to analyse the *static complexity* of Haskell source code, that is it analyses the source code itself and not how it behaves at run time. It analyses the code on a per-module basis, cross-module imports as well as the entire piece of software. Thus, SourceGraph could be an important utility in a programmer's toolkit to help them understand and visualise how their code actually fits together and what it's doing.

SourceGraph is licensed under the GNU General Public License Version 3[1], and is available from hackageDB at:

`http://hackage.haskell.org/cgi-bin/hackage-scripts/package/SourceGraph`

with the source repository at:

`http://code.haskell.org/SourceGraph`

At the time of writing, the latest available version is 0.2.

5.1 Using SourceGraph

To run SourceGraph, first navigate to the code base's root directory, then call the SourceGraph executable with the software's Cabal [14] file as the sole argument. This will produce the analysis report in the "SourceGraph" subdirectory. For example, here is how SourceGraph is run on itself:

```
ivan $ ls
Analyse/    _darcs/  Parsing/    setVersion.sh*
Analyse.hs  dist/    Parsing.hs  SourceGraph.cabal
COPYRIGHT   Main.hs  Setup.lhs   TODO

ivan $ SourceGraph SourceGraph.cabal
Report generated at: SourceGraph/SourceGraph.html

ivan $ ls
Analyse/    _darcs/  Parsing/    setVersion.sh*      TODO
Analyse.hs  dist/    Parsing.hs  SourceGraph/
COPYRIGHT   Main.hs  Setup.lhs   SourceGraph.cabal

ivan $ ls SourceGraph/
Analyse_collapsed.png               codeRNG.png
Analyse.Everything_collapsed.png    importCluster.png
Analyse.Everything.png              importCW.png
Analyse.Imports_collapsed.png       importRNG.png
Analyse.Imports.png                 imports.png
Analyse.Module_collapsed.png        Main_collapsed.png
Analyse.Module.png                  Main.png
Analyse.png                         Parsing_collapsed.png
Analyse.Utils_collapsed.png         Parsing.ParseModule_collapsed.png
Analyse.Utils.png                   Parsing.ParseModule.png
codeCluster.png                     Parsing.png
codeCollapsed.png                   Parsing.Types_collapsed.png
codeCW.png                          Parsing.Types.png
code.png                            SourceGraph.html
```

[1] `http://www.gnu.org/licenses/gpl-3.0.txt`

Listing 5.1: Running SourceGraph on itself

(where `ivan $` is just the prompt, emphasised to differentiate it from the results).

For comparison, we also show the output when running SourceGraph on the source code of Graphalyze:

```
ivan $ ls
_darcs/  dist/             LICENSE    setVersion.sh*
Data/    Graphalyze.cabal  Setup.lhs  TODO

ivan $ SourceGraph Graphalyze.cabal
Report generated at: SourceGraph/Graphalyze.html

ivan $ ls
_darcs/  dist/             LICENSE    setVersion.sh*  TODO
Data/    Graphalyze.cabal  Setup.lhs  SourceGraph/

ivan $ ls SourceGraph/
codeCluster.png
codeCollapsed.png
codeCore.png
codeCW.png
code.png
codeRNG.png
Data.Graph.Analysis.Algorithms.Clustering_collapsed.png
Data.Graph.Analysis.Algorithms.Clustering.png
Data.Graph.Analysis.Algorithms_collapsed.png
Data.Graph.Analysis.Algorithms.Common_collapsed.png
Data.Graph.Analysis.Algorithms.Common_core.png
Data.Graph.Analysis.Algorithms.Common.png
Data.Graph.Analysis.Algorithms.Directed_collapsed.png
Data.Graph.Analysis.Algorithms.Directed.png
Data.Graph.Analysis.Algorithms.png
Data.Graph.Analysis_collapsed.png
Data.Graph.Analysis.png
Data.Graph.Analysis.Reporting_collapsed.png
Data.Graph.Analysis.Reporting.png
Data.Graph.Analysis.Testing_collapsed.png
Data.Graph.Analysis.Testing.png
Data.Graph.Analysis.Types_collapsed.png
Data.Graph.Analysis.Types.png
Data.Graph.Analysis.Utils_collapsed.png
Data.Graph.Analysis.Utils_core.png
Data.Graph.Analysis.Utils.png
Data.Graph.Analysis.Visualisation_collapsed.png
Data.Graph.Analysis.Visualisation.png
Graphalyze.html
importCluster.png
importCW.png
importRNG.png
imports.png
```

Listing 5.2: Running SourceGraph on Graphalyze

Thus, when running SourceGraph on `Foo.cabal`, it will produce the resulting analysis report of the project at `SourceGraph/Foo.html`. It might appear that it takes a long time for SourceGraph to run, especially on larger code bases: this is mainly due to the time taken to create the various graph visualisations used in the report, rather than the algorithms or choice of language.

5.2 Analysing Haskell

There are several advantages in using Haskell as a sample language for analysis purposes, not least of which is the availability of ready-made parsers (see Section 3.3.2). It has the following properties in common with most other functional languages as opposed to the more popular languages such as C and Java:

1. Preference for smaller, more numerous functions over larger monolithic ones.
2. Each function typically calls a larger number of other functions.
3. Due to the high level of polymorphism and higher-order functions, functions are more re-usable and hence are called more often.
4. The common usage of recursion as a looping construct rather than iteration (e.g. for-loops) found in imperative languages.

When viewing source code in any language as a graph, we typically use functions/methods/subroutines/etc. (depending on the language being used) as nodes in the graph, and function calls as directed edges. As such, the above list transforms into the following graph properties (when compared to the graph representation of code written in other languages):

1. Larger number of nodes.
2. Each node typically has a larger out degree.
3. Nodes typically have a larger in degree.
4. Appearance of loops and cycles (non-functional languages generally will have little or no loops or cycles).

For example, a graph representation of the four functions found in Listing 4.10 on page 17 can be found in Figure 5.1. Note that whilst `componentsOf` recursively calls `splitComponent` until the remaining graph is empty, this recursive call is hidden by the higher-order function *unfoldr*. For more information on this, please see Section 6.

Thus, when representing Haskell code as a graph, the nodes are functions, directed edges are function calls and graph clusterings are the modules those functions belong in. Root nodes would be those that are exported in a given module (or exported overall for the entire code base: for a programme, this will be just the `main` function) and leaves are any variables or functions that are not dependent upon other functions/variables in the code base. However, when representing the module imports as a graph, the nodes are the modules, the directed edges are the module imports and the clusters are the directories that module is in (see Section 3.2.1 for more information). Root modules are those that are exported and available for use in a library, and the overall programme module in an application.

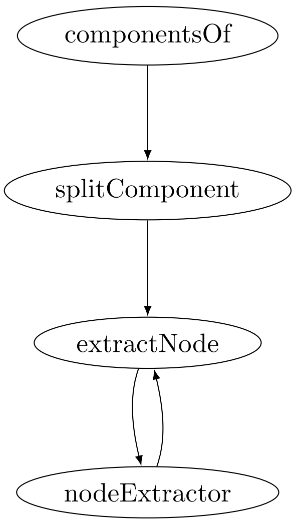


Figure 5.1: Graph representation of Listing 4.10

5.3 SourceGraph limitations

At the moment, SourceGraph isn't as nicely polished an application as one would like. There are currently four classes classes of limitations:

1. Parsing
2. Analysing
3. Reporting
4. Usage

The majority of these errors are able to be remedied and will be in future.

5.3.1 Parsing Limitations

Whilst Haskell-Src with Extensions provides parsing support for may extensions, as yet SourceGraph doesn't fully utilise it for the following classes of extensions:

- Template Haskell (allows meta-programming in Haskell)
- HaRP (regular expression pattern matching)
- HSX (embedding XML into Haskell source code)

At the moment, these types of extensions are silently ignored, mainly due to unfamiliarity with how they work. These extensions will be supported in a future release.

Many Haskell modules utilise a C pre-processor (cpp) to enable compile-time options (e.g. support of UTF-8 strings). HSE can't parse these pre-processing annotations, and attempting to silently ignore these could have adverse effects on the parsed source. For example, if a function can be defined one of several ways depending on which compile-time flag is chosen, then by ignoring the annotations more than one definition of the function will be defined. Also, if a function is commonly found within a library but a version exists if when compiled that library is chosen not to be utilised, then when parsing it is difficult to tell which version of the function should be utilised. However, there does exist a library version of the cpp utility: cpphs [28]. A future release of SourceGraph will utilise this library to be able to fully parse Haskell modules that require pre-processing.

One parsing limitation that will probably never be supported is what to do with what can be labelled as *data-oriented functions*. This covers functions that are part of class definitions or record statements. Class definition functions (also known as *methods* are ignored because when a function uses them, it isn't possible to tell for which instance (and hence which actual function definition) they are using. As for record statement functions, these are trivial functions that are analogous to getter/setter functions in languages such as Java.

5.3.2 Analysis Limitations

The only limitation due to analysis is that there are many more possible analyses that could be utilised than what are already present. For example, it would be interesting to examine the depth of each leaf function, using the shortest path from any root function. This would give an idea of the overall "depth" of the code. Another possible algorithm would be to examine which functions are the most "popular", that is those that have the highest number of function calls.

5.3.3 Reporting Limitations

The report output is limited by the `DocumentGenerator` class utilised (see Listing 4.18). At the moment, SourceGraph utilises the Pandoc instance, which has the unfortunate tendency to be a "jack of all trades and master of none", that is it can produce a wide variety of document types but a specific LaTeX generator for example would no doubt produce better output than Pandoc does. Most of the reporting problems arise directly from limitations in how Graphalyze converts its internal `Document` type to Pandoc's, and as such when the conversion is improved then SourceGraph's reporting capabilities will improve as well. One area where SourceGraph can be improved though is reporting of "boring" analysis: if only one parseable function exists in a module (the rest might be class instantiations, etc.) then there isn't much point in reporting information just about that module.

5.3.4 Usage Limitations

For more information on how to use SourceGraph, please see Section 5.1. As it stands however, SourceGraph only runs from the command line (which may or may not be a limitation), and only analyses Haskell software that utilises Cabal [14] to build themselves (so software such as GHC [15] which use Makefiles cannot be analysed). Furthermore, output is currently non-customisable: SourceGraph will produce a single html report file using Portable Network Graphics (png) images in a subdirectory called *SourceGraph* inside the software's root directory. There are also no help options, etc. provided, and in many cases if anything fails (e.g. a parsing error) then it does so silently with no explanation.

Despite all these usage limitations/errors, care has been taken to ensure that SourceGraph can't do any damage to either the code or to the filesystem in general.

5.4 How SourceGraph works

Following is a simplified outline of how SourceGraph works:

1. Parse the Cabal file, returning the exposed modules (or if it produces an executable, which file contains the `main` function).
2. Find every Haskell file (i.e. those with a `hs` or `lhs` extension) and attempt to parse it.
3. Convert the HSE representation of all parseable files into the one used by SourceGraph (which is fundamentally a mapping from a function name to the functions which it calls).
4. Analyse the modules, imports and overall code base, creating the `Document` representation.
5. Create the report.

Note that SourceGraph is smart enough to ignore "trivial" Haskell files, such as those generated by Cabal when building a Haskell package, and those that might be kept in side a Darcs [23] repository (Darcs is a distributed version-control system written in Haskell, and is thus very popular in the Haskell community).

5.4.1 Converting HSE representation

Note in particular that when SourceGraph converts HSE's representation into one that it can use, it attempts to assign all functions found to the modules they belong in, with any unknown functions (i.e. from other libraries or unparseable modules) are discarded. But to do so, it has to already have parsed all the files and found all the functions, which in turn requires that all possible functions are known so that functions can be assigned to modules, and so on *ad infinitum.* This kind of infinite loop processing is where lazy languages like Haskell (see Section 3.2.1) excel: whilst we have to know which functions are available from each module to be able to assign functions to modules, we *don't* have to have this mapping to find all functions defined in each module. Thus, we can utilise a programming idiom known as *Tying the Knot* [26] where we (lazily) partially evaluate a data structure before using the partial evaluations to finish evaluation.

When converting the parsed data, tying the knot is utilised twice:

- To assign module names to functions in the current module whilst finding all the functions defined in that module.
- To assign module names to functions from other modules utilised inside the current module before all those other modules have been parsed.

As an example of tying the knot, let us consider the second example, found in Listing 5.3. The tying the knot is performed in lines twenty-six and twenty-seven, which recursively rely on each other to evaluate.

```
 1 -- The file contents as well as its location.
 2 type FileContents = (FilePath,String)
 3
 4 -- A high-level viewpoint of a Haskell module.
 5 data HaskellModule = Hs { moduleName :: ModuleName
 6                         , imports    :: [ModuleName]
 7                         , exports    :: [Function]
 8                         , functions  :: FunctionCalls
 9                         }
10
11 -- A lookup map of HaskellModules.
12 type HaskellModules = Map ModuleName HaskellModule
```

```

-- The name of a module.  The 'Maybe' component refers to the possible
-- path of this module.
data ModuleName = M (Maybe String) String

-- Parse all the files and return the map, utilising Tying the Knot.
parseHaskell    :: [FileContents] → HaskellModules
parseHaskell fc = hms
    where
      ms = parseFiles fc
      -- The next two lines recursively rely on each other, using
      -- laziness to continuously partially evaluate one and then
      -- the other.  The createModuleMap and parseModule functions are
      -- defined elsewhere.
      hms = createModuleMap hss
      hss = map (parseModule hms) ms

-- Parse all the files that you can.
parseFiles :: [FileContents] → [HsModule]
parseFiles = catMaybes ∘ map parseFile


-- Attempt to parse an individual file using Haskell-Src with
     ...Extensions.
parseFile       :: FileContents → Maybe HsModule
parseFile (p,f) = case (parseModuleWithMode mode f) of
                    (ParseOk hs) → Just hs -- Able to parse the module
                            ....
                    _            → Nothing -- Unable to parse.
    where
      mode = ParseMode p
```

Listing 5.3: Simultaneously parsing all Haskell modules

5.4.2 Chosen analysis algorithms

The algorithms from Graphalyze chosen to analyse the static complexity of various parts of Haskell source code can be found in Table 5.1. Note that "Entire Code Base" refers to the functions from *all* modules. When visualising clusters, the current clusters in the graph are first visualised (for imports, this is the directory in which the module can be found; for the code base, functions are clustered by the module they're in) and then the two clustering algorithms found in Section 4.4.3 are used to visualise alternate clustering arrangements. Connected components are important when programming because they indicate that the current module/code base can be safely split up since the two sections are completely independent of each other, whereas chains indicate that the given functions/modules can be combined into one: to the rest of the code base, they only appear as one anyway.

There is one other analysis algorithm that is applied to each level of interest: calculating its *Cyclomatic Complexity* [20]. Note that technically the usage of cyclomatic complexity in these contexts isn't as intended: cyclomatic complexity is a software metric intended to measure the number of linearly different paths through an application written using a *structured programming language* (which Haskell — as a functional language — isn't). Such an application is represented

Algorithm	Per-Module	Imports	Entire Code Base
Graph Visualisation	✔	✔	✔
Collapsed Graph Visualisation	✔		✔
Graph Core Visualisation	✔		✔
Cluster Visualisation		✔	✔
Root Analysis	✔	✔	✔
Connected Components	✔	✔	✔
Clique Finding	✔		✔
Cycle Finding		✔	
Clique-less Cycle Finding	✔		✔
Chain Finding	✔	✔	✔

Table 5.1: Algorithms used to analyse Haskell code

as a graph where the nodes are functions but with edges between nodes indicating possible data-flow between those functions. For such a graph $G = (V, E)$, the cyclomatic complexity M is defined as:

$$M = |E| - |V| + 2P \tag{5}$$

where P is the number of connected components in G. This definition assumes that there is only one possible "exit" from the programme: if it is possible to exit the programme from multiple points, the definition then becomes:

$$M = |E| - |V| + P + R \tag{6}$$

where R is the number of possible exit points.

Despite not being the original intended use, SourceGraph still uses Cyclomatic Complexity as defined in (5) as a useful graph-based metric for all three levels of interest. No literature has been found using it as a metric in this fashion for functional languages (indeed, it seems to be oriented only at stand-alone applications and not libraries), and as such SourceGraph is unable to give any explanation of what the value returned actually represents from the code. The Cyclomatic Complexity is calculated as follows:

```
cyclomaticComplexity    :: GraphData a → Int
cyclomaticComplexity gd = e - n + 2*p
    where
      p = length $ applyAlg componentsOf gd
      n = applyAlg noNodes gd
      e = length $ applyAlg labEdges gd
```

Listing 5.4: Calculating the Cyclomatic Complexity

5.5 Caveats of using SourceGraph

Despite utilising and manipulating the source code at a graph level, SourceGraph is *not* a Haskell refactoring tool: instead, look at *HaRe* [17]. The reason for this is that all information on the structure of the source code is lost after converting the parsed output into SourceGraph's internal state, let alone when converting it to a graph. Furthermore, its output should be taken with a grain of salt, as it doesn't actually understand what its analysing. For example, it might recommend that you split a module up into several modules because there are numerous components in there, whereas you've created it like that since it's a library/utility module and all those functions are related in what they *do*, if not in how they do it (e.g. most standard math library modules in any language have a large number of functions which in no way depend upon each other).

6 Analysis of Results

This section examines sample results from the SourceGraph application. The results themselves can be found in Appendix B, and cover analysis of *XMonad* [25]. Note that we will be solely focusing on the results obtained, and not on the visual aspect or layout.

XMonad is an extensible tiling window manager for the X Window System written in Haskell. It is a combination application and library (note that it is usually utilised with an extension library known as *XMonad-Contrib* that is released separately). It might not be a very large project, but for the purposes of brevity it wouldn't be beneficial to include the entire SourceGraph report for a larger one (even SourceGraph itself has a generated report of roughly twenty-two pages).

One thing that's immediately obvious is how "wide" the graph quickly becomes, even in individual modules. This is due to the much larger number of smaller functions typically found in Haskell code (see Section 5.2). Also, individual modules get more and more complicated rather quickly.

As for proposed module groupings using the clustering routines, they tend to over-fragment the modules, though the Relative Neighbourhood algorithm (the second alternative clustering shown) seems to give slightly better results.

Present in the overall call graph is a function that calls another repeatedly. Without even examining the source code, this can probably be attributed to a multi-part function definition that utilises pattern matching (with each separate pattern in the first function calling the second). This is thus an example of information that can be extracted just from the visual representation of the code base.

Whilst it was said that Haskell code utilises recursion and function "cycles" a lot more than non-functional languages, none can be found in the XMonad source. This is because most recursive are hidden within *higher order functions*. For example, consider the `map` function in Listing 3.2: if a function uses `map`, then the recursion is "hidden away" in the higher order function. Thus, direct evidence of recursion is usually found in more complicated one-off examples that are not as easily covered by generic recursion routines. For a direct code example, `extractNode` and `nodeExtractor` in Listing 4.10 cyclically depend upon each other, but the recursive call of `splitComponent` by `componentsOf` is hidden away in the higher order function `unfoldr`. Even those functions that do directly utilise recursion typically do so within smaller functions defined within the larger one, which are thus not visible to the rest of the code base (and which SourceGraph will strip out since it isn't a defined function).

XMonad has five connected components (this is easier to see in the collapsed view). This, however, is indicative of the fact that it is not only an application, but also an extensible library. As such, many of the functions might not be directly utilised by the main program, but are available for end users to use to customise their installation.

Examining the cyclomatic complexity of, we find that there does indeed seem to be relationship between the "visual" complexity and the one calculated. The decision on whether or not it reveals useful information to the programmer will have to be delayed until more code bases are analysed using SourceGraph and the results compared.

One item that will definitely be useful, however, is the list of chains: it is quite possible that several of these chains can be compressed down to single functions, thus reducing the complexity of code.

7 Future Work

Whilst Graphalyze and SourceGraph are currently usable, work on them — like almost any other piece of software — will never truly finish. As it stands, there are already several outstanding issues and possible future areas of work that can be seen.

Whilst Graphalyze has a fair number of algorithms included in it, there is always room for more. Also, these algorithms can be further extended by more general-purpose analysis functions that try and extract something more than just raw data out of the graph. The reporting mini-library also needs more work. In particular for html output, it would be useful to have "zooming" support to click on an image and open up a larger version (images are currently hard-coded to be no larger than fifteen inches wide by ten inches high (since the GraphViz programme measures in inches) so as not to be too big). Furthermore, html output would be much nicer to read if it was split over multiple pages and utilised Cascading Style Sheets (CSS) to produce something other than plain html.

Furthermore, many of the visualisation helper functions should probably be moved to the graphviz library, as they are wrapper functions that were not included due to lack of time on Matthew Sackmann's part. Visualisation functionality that as yet isn't included however is to colour the nodes in the graphs, for example using node colour as a visual indication of a nodes degree. Some of the graph algorithms might also be more appropriate included in the Functional Graph Library.

As for SourceGraph, apart from those fixable limitations listed in Section 5.3, the reporting output could also be made "smarter", by detecting whether or not the analysis output is actually interesting. Some attempt at this is already made by not listing for example connected component analysis if there is already one component, but this should be extended to other analysis types (for instance, don't list a chain in the overall analysis section if it consists of nodes from a single module and hence would have already been listed).

However, possibly the most exciting possible extension that could be made to SourceGraph is to extend its language capabilities. Benedikt Huber recently released his Language.C library [13], which is a full C99 parser written in Haskell and analogous to Haskell-Src and Haskell-Src with Extensions. This will require tweaking of SourceGraph's internal state, but will allow greater usage of this utility.

Another possible adaption is to extend the scope of SourceGraph to use it to track *data flow* within a programme. This would be useful to visually follow the path data takes throughout the programme. Also, if profiling information exists then adding that to the graph would aid the programmer in finding bottlenecks in their code (note that there already exists a Haskell application that specifically performs this task [32]).

8 Conclusion

Mathematics is often used to assist people in analysing their raw data to convert it into usable information. As data sets become larger and larger, computational techniques and libraries are steadily increasing in importance for data analysis. However, whilst there are a large number and variety of utilities available to assist people in analysing continuous data, there is a dearth of resources for computationally analysing discrete data sources and — almost more importantly — the relationships inherent within such data sources (which indeed might even characterise the data source).

Graph theory is an extremely useful tool for people who wish to analyse the relationships between discrete data points. But as it stands there is no general-purpose graph-theoretic library available for people to use for their own analysis (though there are more specialised libraries available for tasks such as network theory and analysis).

In this thesis, we have examined a general purpose library aimed at assisting people in utilising graph theory to analyse discrete data. What's more, this library is *extensible*, allowing people to define and use their own custom analysis functions. This library — called GRAPHALYZE — is freely available under a two-clause BSD license. GRAPHALYZE is fully documented and comes with its own mini reporting library to help users produce documents containing their analysis results.

As a sample application of this library, the SOURCEGRAPH application analyses Haskell source code, using functions for nodes and functions calls for directed edges in the analysis graph. Whilst this application isn't quite polished, it is currently usable and returns interesting and valid results. It is hoped that in the future, SOURCEGRAPH becomes a vital part of every Haskell programmer's toolkit to help visualise how their code fits together, and find possible improvements.

8.1 Summary of work

Using David A. Wheeler's *SLOCCount* [29] program, we are able to find the total number of lines of source code produced, as well as use arbitrary values to calculate how much code of this size would cost to produce:

```
ivan $ ls
Graphalyze/  SourceGraph/

ivan $ sloccount --multiproject --wide Graphalyze SourceGraph
Creating filelist for Graphalyze
Creating filelist for SourceGraph
Categorizing files.
Finding a working MD5 command....
Found a working MD5 command.
Computing results.


SLOC    Directory   SLOC-by-Language (Sorted)
1226    Graphalyze      haskell=1222,sh=4
990     SourceGraph     haskell=986,sh=4


Totals grouped by language (dominant language first):
haskell:       2208 (99.64%)
```

```
sh:                8 (0.36%)



Total Physical Source Lines of Code (SLOC)                = 2,216
Development Effort Estimate, Person-Years (Person-Months) = 0.45 (5.35)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 0.32 (3.78)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Total Estimated Cost to Develop                           = $ 60,196
 (average salary = $56,286/year, overhead = 2.40).
SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
SLOCCount is Open Source Software/Free Software, licensed under the GNU
    ...GPL.
SLOCCount comes with ABSOLUTELY NO WARRANTY, and you are welcome to
redistribute it under certain conditions as specified by the GNU GPL
    ...license;
see the documentation for details.
Please credit this data as "generated using David A. Wheeler's 'SLOCCount
    ...'."
```

Listing 8.1: Calculating the amount of code produced

Furthermore, on top of this code was contributed to Matthew Sackmann's *graphviz* [24] library to enable it to differentiate between directed and undirected graphs, as well as drawing graph clusterings.

8.2 A final note

Whilst Graphalyze can help you turn your raw data into some type of information, it is important to note that information is only valid within the required context, and that the results might not always be useful. For example, as seen in Section 6, not all analysis results of SourceGraph are useful, because the application isn't a programmer and doesn't understand the internal logic of the code (nor does it analyse the resulting graph with any particular context).

This cautionary statement was said best by Arthur C. Clarke in an interview in 2003 [11], when he said:

> But it is vital to remember that information — in the sense of raw data — is not knowledge; that knowledge is not wisdom; and that wisdom is not foresight. But information is the first essential step to all of these.

References

[1] Sanghamitra Bandyopadhyay. An automatic shape independent clustering technique. *Pattern Recognition*, 37(1):33–45, 2004.

[2] C. Biemann. Chinese Whispers - an Efficient Graph Clustering Algorithm and its Application to Natural Language Processing Problems. In *Proceedings of the HLT-NAACL-06 Workshop on Textgraphs-06*, New York, USA, 2006.

[3] Zdenek Dvorak, Jan Hubicka, Pavel Nejedly, and Josef Zlomekj. Infrastructure for profile driven optimizations in GCC compiler. `http://www.ucw.cz/~hubicka/papers/proj/index.html`, 2002. accessed 5 July, 2008.

[4] Martin Erwig. Functional Graph Library/Haskell. `http://web.engr.oregonstate.edu/~erwig/fgl/haskell`. accessed 24 March, 2008.

[5] Martin Erwig. FGL/Haskell - a funtional graph library - user guide. `http://web.engr.oregonstate.edu/~erwig/fgl/haskell/old/fgl0103.pdf`, 12 April 2001. accessed 24 March, 2008.

[6] Martin Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.*, 11(5):467–492, 2001.

[7] Simon Peyton Jones et. al. GHC Language Features. `http://www.haskell.org/ghc/docs/latest/html/users_guide/ghc-language-features.html`. accessed 25 October, 2008.

[8] Guojun Gan, Chaoqun Ma, and Jianhong Wu. *Data Clustering: Theory, Algorithms, and Applications*. ASA-SIAM Series on Statistics and Applied Probability. SIAM, 2007.

[9] Emden R. Gansner and Stephen C. North. An Open Graph Visualization System and Its Applications. *Software - Practice and Experience*, 30:1203–1233, 1999. `http://graphviz.org`.

[10] John Gruber. Markdown. `http://daringfireball.net/projects/markdown/`.

[11] Nalaka Gunawardene. Humanity will survive information deluge. `http://ez.oneworldsouthasia.net/article/view/74591`, 5 December 2003. Interview with Sir Arthur C. Clarke.

[12] hackageDB. `http://hackage.haskell.org`. accessed 25 October, 2008.

[13] Benedikt Huber. Language.c - a c99 library for haskell . `http://www.sivity.net/projects/language.c`.

[14] Isaac Jones. The haskell cabal: a common architecture for building applications and libraries. In Marko van Eekelen, editor, *6th Symposium on Trends in Functional Programming*, pages 340–354, 2005. `http://www.haskell.org/cabal/`.

[15] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, 1993. `http://haskell.org/ghc/`.

[16] Melvin Kranzberg. The information age: Evolution or revolution? In Bruce R. Guile, editor, *Information Technologies and Social Transformation*, number 2 in Series on Technology and Social Priorities, pages 35–53. National Academy of Engineering, National Academy Press, October 1984.

[17] Huiqing Li and Simon Thompson. Formalisation of haskell refactorings. In *In Trends in Functional Programming*, 2005. `http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html`.

[18] John MacFarlane. Pandoc. `http://johnmacfarlane.net/pandoc/`.

[19] Simon Marlow. Haddock, a haskell documentation tool. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 78–89, New York, NY, USA, 2002. ACM. `http://www.haskell.org/haddock/`.

[20] Thomas J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[21] Ivan Lazar Miljenovic. ANNOUNCE: graphviz-2008.9.20. email sent to the haskell@haskell.org mailing list, 21 September 2008. `http://www.haskell.org/pipermail/haskell/2008-September/020631.html`.

[22] Simon Peyton Jones, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, John Hughes, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. Also available at `http://www.haskell.org/definition/` and through Cambridge Press.

[23] David Roundy. Darcs: distributed version management in Haskell. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 1–4, New York, NY, USA, 2005. ACM. `http://darcs.net/`.

[24] Matthew Sackmann and Ivan Lazar Miljenovic. graphviz: GraphViz wrapper for Haskell. `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/graphviz`.

[25] Don Stewart and Spencer Janssen. XMonad: A tiling window manager. In *Haskell '07: Proceedings of the 2007 ACM SIGPLAN Workshop on Haskell*, New York, NY, USA, Sep 2007. ACM Press.

[26] Tying the knot. `http://www.haskell.org/haskellwiki/Tying_the_Knot`.

[27] A. Vitalis, D. J. Monin, and P. Holland. *Network Analysis - A Practical Approach*. The Dunmore Press Limited, 1987.

[28] Malcolm Wallace. cpphs: A liberalised re-implementation of cpp, the C pre-processor. `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/cpphs`.

[29] David A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.

[30] Call Graph. `http://en.wikipedia.org/wiki/Call_graph`. accessed 5 July, 2008.

[31] Wirt Wolff. Interest in using graphalyze for a snakes and paths puzzle. Private Communication, 2 October 2008.

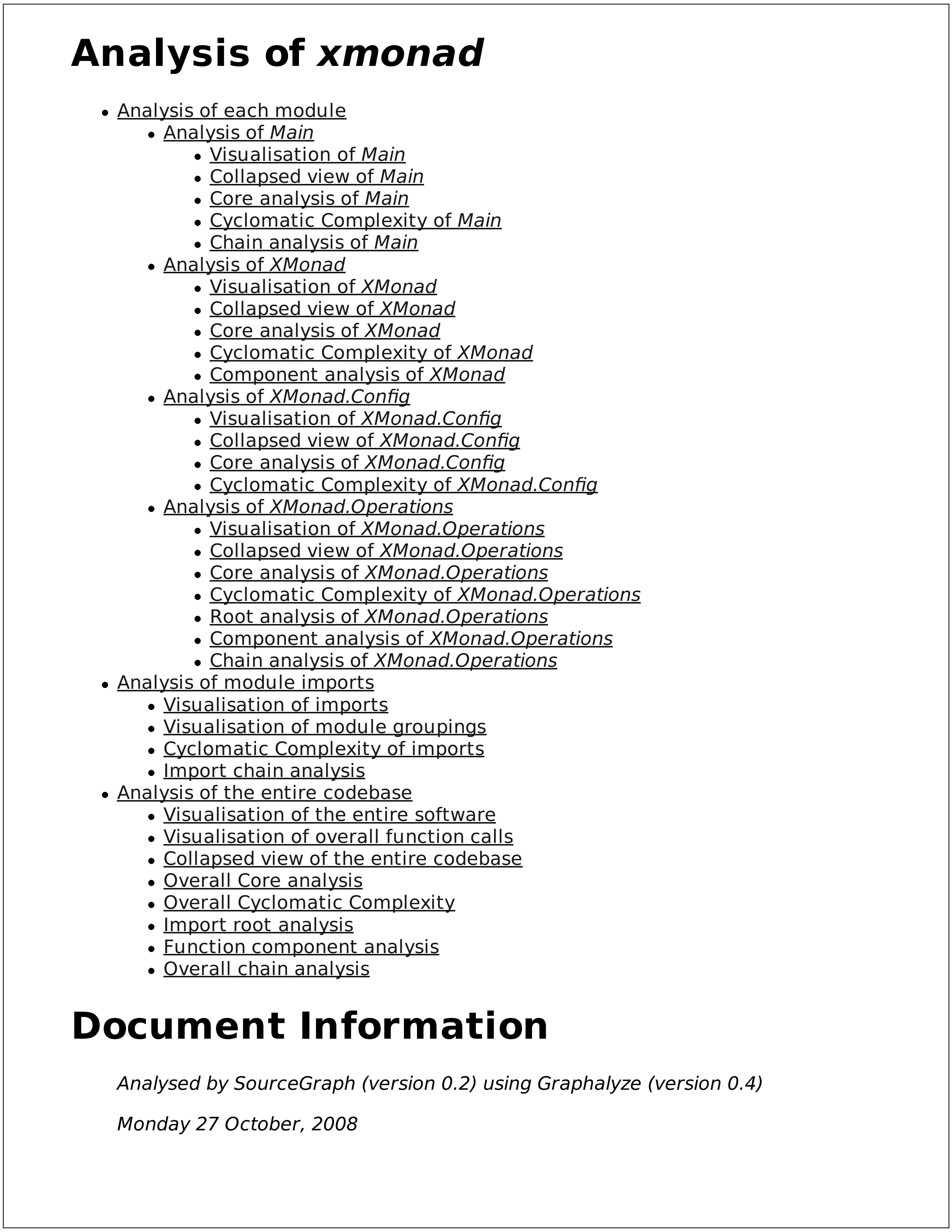
[32] Gregory Wright. prof2dot. `http://antiope.com/downloads.html`.

Appendices

Appendix A Licence of the Graphalyze library

Appendix B SourceGraph report on XMonad

Starting over the page is a sample output from SourceGraph when run on the source for XMonad [25]. Note that some quality has been lost due to the conversion from html to pdf format to let it be embedded into this document.;



Analysis of *xmonad*

Document Information

Analysed by SourceGraph (version 0.2) using Graphalyze (version 0.4)

Monday 27 October, 2008

Monday 27 October, 2008

Analysis of each module

Analysis of *Main*

Visualisation of *Main*

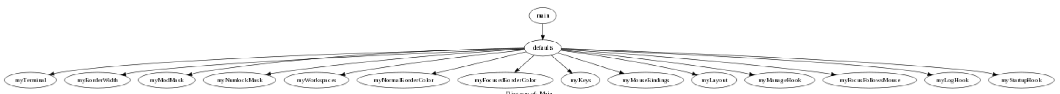
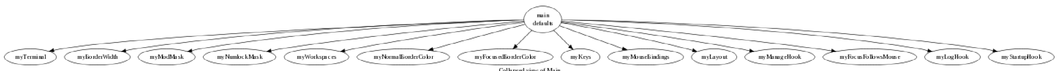


Diagram of: Main

Collapsed view of *Main*

The collapsed view of a module collapses down all cliques, cycles, chains, etc. to make the graph tree-like.



Collapsed view of Main

Core analysis of *Main*

The core of a module can be thought of as the part where all the work is actually done.

The module Main is a tree.

Cyclomatic Complexity of *Main*

The cyclomatic complexity of Main is: 1.

For more information on cyclomatic complexity, please see: Wikipedia: Cyclomatic Complexity

Chain analysis of *Main*

The module Main has the following chains:

main -> defaults

These chains can all be compressed down to a single function.

Analysis of *XMonad*

Visualisation of *XMonad*

Diagram of: XMonad

Diagram of: XMonad

Collapsed view of *XMonad*

The collapsed view of a module collapses down all cliques, cycles, chains, etc. to make the graph tree-like.

Collapsed view of XMonad

Core analysis of *XMonad*

The core of a module can be thought of as the part where all the work is actually done.

The module XMonad is a tree.

Cyclomatic Complexity of *XMonad*

The cyclomatic complexity of XMonad is: 0.

For more information on cyclomatic complexity, please see: Wikipedia: Cyclomatic Complexity

Component analysis of *XMonad*

The module XMonad has 0 components. You may wish to consider splitting it up.

Analysis of *XMonad.Config*

Visualisation of *XMonad.Config*

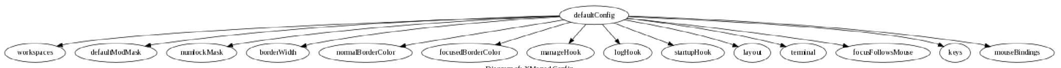
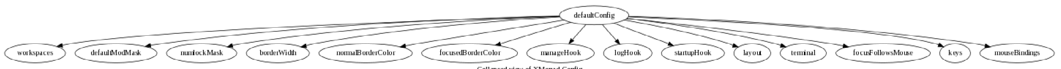


Diagram of: XMonad.Config

Collapsed view of *XMonad.Config*

The collapsed view of a module collapses down all cliques, cycles, chains, etc. to make the graph tree-like.



Collapsed view of XMonad.Config

Core analysis of *XMonad.Config*

The core of a module can be thought of as the part where all the work is actually done.

The core of a module can be thought of as the part where all the work is actually done.

The module XMonad.Config is a tree.

Cyclomatic Complexity of *XMonad.Config*

The cyclomatic complexity of XMonad.Config is: 1.

For more information on cyclomatic complexity, please see: Wikipedia: Cyclomatic Complexity

Analysis of *XMonad.Operations*

Visualisation of *XMonad.Operations*

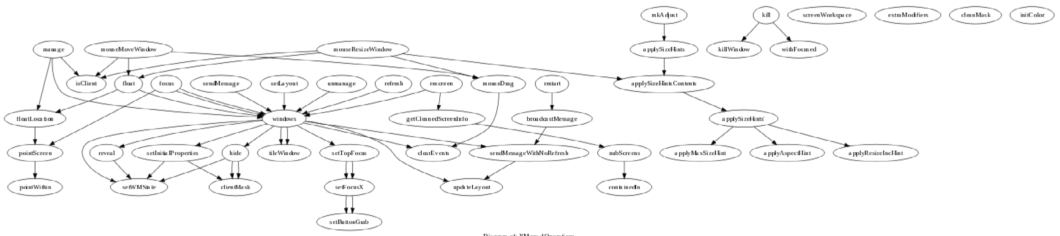
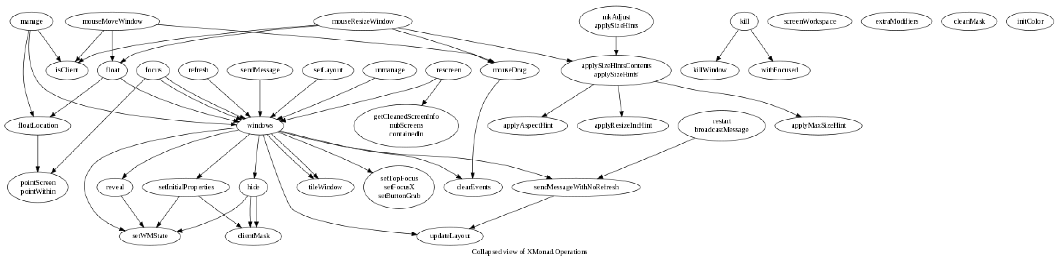


Diagram of: XMonad.Operations

Collapsed view of *XMonad.Operations*

The collapsed view of a module collapses down all cliques, cycles, chains, etc. to make the graph tree-like.



Collapsed view of XMonad.Operations

Core analysis of *XMonad.Operations*

The core of a module can be thought of as the part where all the work is actually done.

The module XMonad.Operations is a tree.

Cyclomatic Complexity of *XMonad.Operations*

The cyclomatic complexity of XMonad.Operations is: 22.

For more information on cyclomatic complexity, please see: Wikipedia: Cyclomatic Complexity

Root analysis of *XMonad.Operations*

These nodes are those that are in the export list and roots:

manage, unmanage, kill refresh, rescreen focus, sendMessage setLayout screenWorkspace extraModifiers cleanMask, initColor restart mouseMoveWindow mouseResizeWindow mkAdjust

These nodes are those that are not in the export list but roots:

killWindow, windows, setWMState hide, reveal, clientMask setInitialProperties, clearEvents tileWindow, containedIn, nubScreens getCleanedScreenInfo, setButtonGrab setTopFocus, setFocusX broadcastMessage sendMessageWithNoRefresh updateLayout, withFocused, isClient floatLocation, pointScreen pointWithin, float, mouseDrag applySizeHints applySizeHintsContents applySizeHints', applyAspectHint applyResizeIncHint, applyMaxSizeHint

Component analysis of *XMonad.Operations*

The module XMonad.Operations has 6 components. You may wish to consider splitting it up.

Chain analysis of *XMonad.Operations*

The module XMonad.Operations has the following chains:

getCleanedScreenInfo -> nubScreens -> containedIn

setTopFocus -> setFocusX -> setButtonGrab

restart -> broadcastMessage

pointScreen -> pointWithin

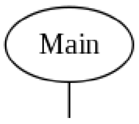
mkAdjust -> applySizeHints

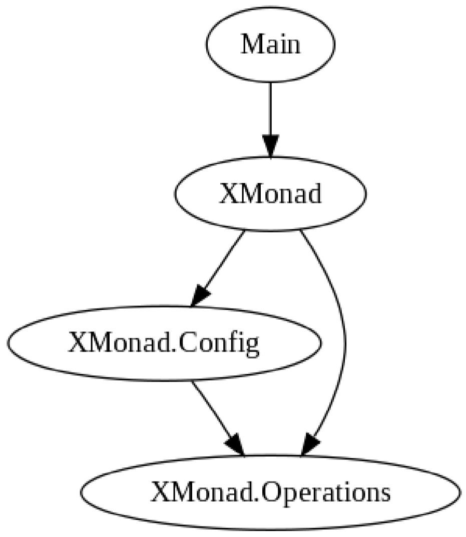
applySizeHintsContents -> applySizeHints'

These chains can all be compressed down to a single function.

Analysis of module imports

Visualisation of imports

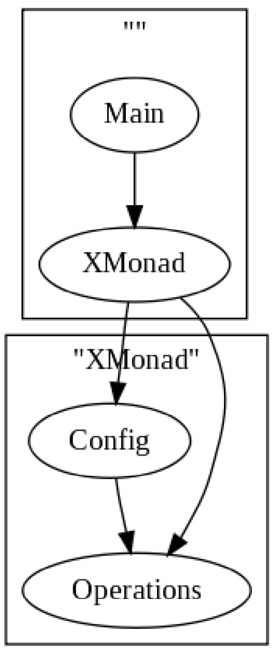




Import visualisation

Visualisation of module groupings

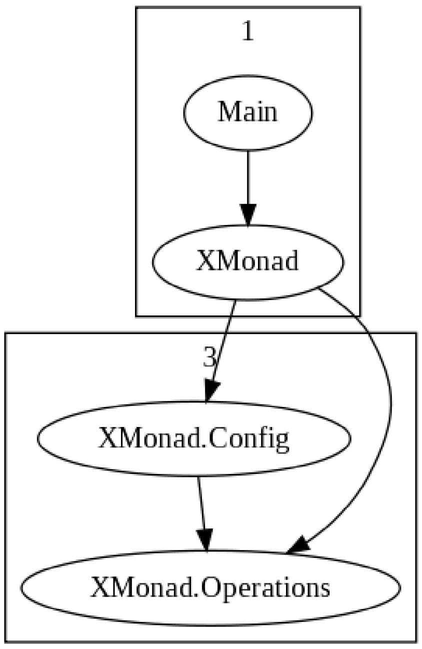
Here is the current module groupings:



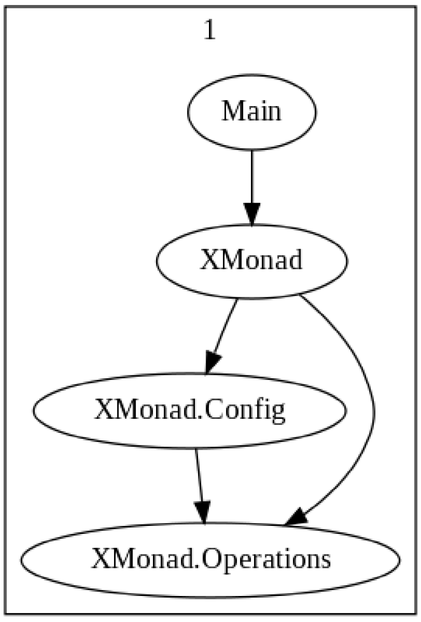
Module groupings

Here are two proposed module groupings:

Here are two proposed module groupings:



Chinese Whispers module groupings



Relative Neighbourhood module groupings

Cyclomatic Complexity of imports

The cyclomatic complexity of the imports is: 2For more information on cyclomatic complexity, please see:Wikipedia: Cyclomatic Complexity

Import chain analysis

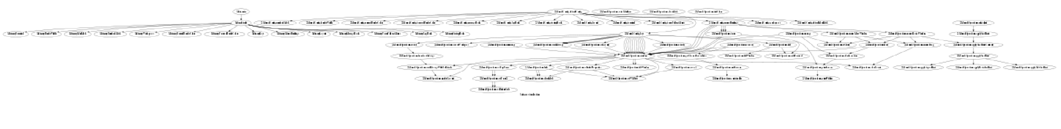
The imports have the following chains:

Main -> XMonad

These chains can all be compressed down to a single module.

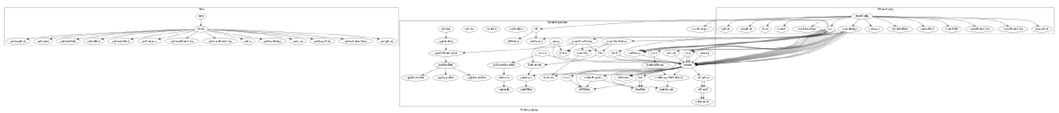
Analysis of the entire codebase

Visualisation of the entire software

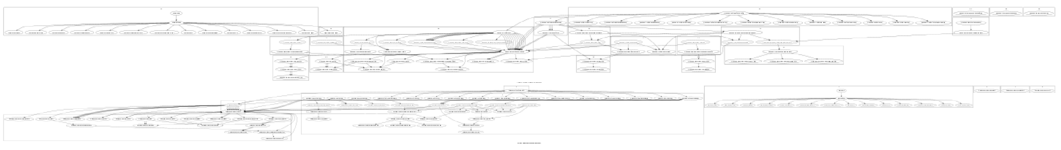


Visualisation of overall function calls

Here is the current module grouping of functions:

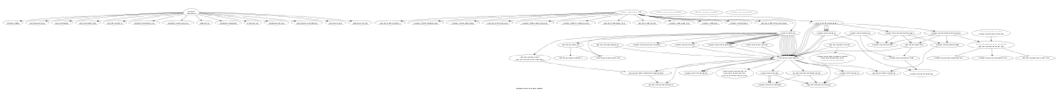


Here are two proposed module groupings:



Collapsed view of the entire codebase

The collapsed view of code collapses down all cliques, cycles, chains, etc. to make the graph tree-like.



Overall Core analysis

The core of software can be thought of as the part where all the work is actually done.

The code is a tree.

Overall Cyclomatic Complexity

The overall cyclomatic complexity is: 48For more information on cyclomatic complexity, please see:Wikipedia: Cyclomatic Complexity

Import root analysis

These functions are those that are available for use and roots:

Main.main

These functions are those that are available for use but not roots:

XMonad.Config.defaultConfig XMonad.Operations.manage XMonad.Operations.unmanage XMonad.Operations.rescreen XMonad.Operations.extraModifiers XMonad.Operations.cleanMask XMonad.Operations.initColor XMonad.Operations.mkAdjust

XMonad.Operations.extraModifiers XMonad.Operations.cleanMask
XMonad.Operations.initColor XMonad.Operations.mkAdjust

Function component analysis

The functions are split up into 5 components. You may wish to consider splitting the code up into multiple libraries.

Overall chain analysis

The functions have the following chains:

Main.main -> Main.defaults

XMonad.Operations.getCleanedScreenInfo -> XMonad.Operations.nubScreens -> XMonad.Operations.containedIn

XMonad.Operations.setTopFocus -> XMonad.Operations.setFocusX -> XMonad.Operations.setButtonGrab

XMonad.Operations.restart -> XMonad.Operations.broadcastMessage

XMonad.Operations.pointScreen -> XMonad.Operations.pointWithin

XMonad.Operations.mkAdjust -> XMonad.Operations.applySizeHints

XMonad.Operations.applySizeHintsContents -> XMonad.Operations.applySizeHints'

These chains can all be compressed down to a single function.