Call Graphing in C

Patrick Onyeyiri

Master of Computer Science with Honours
The University of Bath
May 2007
This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed:
Call Graphing in C

Submitted by: Patrick Onyeyiri

COPYRIGHT

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop).

This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:
Abstract

Source code visualisation is the act of using some form of graphical representation of the source code of an application. The intention is to make it easier to understand its structure, allowing a designer to improve on its design, or a maintenance programmer to understand what is being maintained. This project specialises on presenting the call graph of an application written in C (with the provision for expansion to other languages), with the intent of departing from the commonly found 2-dimensional diagrams, into the 3D world.
Contents

1 Introduction 1
  1.1 Project Structure ............................................. 1
  1.2 Aim .......................................................... 1
  1.3 Objectives ..................................................... 2

2 Literature Survey 3
  2.1 Source Code Visualisation ................................. 3
    2.1.1 Common Methods of Visualisation ..................... 3
    2.1.2 Reasons why they Fail .................................. 5
    2.1.3 Important Factors ....................................... 6
    2.1.4 How Many Dimensions? ................................. 7
  2.2 A Visualisation Tool Decomposed ......................... 8
    2.2.1 Lexical Analysis ....................................... 8
    2.2.2 Parsing ................................................ 8
    2.2.3 Internal Representation ............................... 10
    2.2.4 Creating The Graph .................................... 10

3 Requirements 14
  3.1 Functional Requirements ................................. 14
  3.2 Non-Functional Requirements ............................ 15
  3.3 Data Requirements .......................................... 16

4 Design 17
  4.1 Stage 1 : Parsing ........................................... 17
CONTENTS

4.2 Stage 2: Graph Layout ........................................ 18
   4.2.1 Layout Algorithms ........................................ 19
4.3 Stage 3: Visualisation ........................................ 25
   4.3.1 Graphics Library ........................................ 25
   4.3.2 Interface Design ...................................... 25
   4.3.3 User Interaction .................................... 26

5 Implementation .................................................. 27
   5.1 Programming Language .................................. 27
   5.2 Graph Data Structure ................................... 27
      5.2.1 The Sparse Graph ................................... 29
   5.3 The Parser .................................................. 29
      5.3.1 Processing a new file ................................ 29
      5.3.2 Processing a function definition .................. 30
      5.3.3 Processing a function call ......................... 30
      5.3.4 Resolving references .............................. 32
   5.4 Generating the graph layout ............................ 32
      5.4.1 High Dimensional Embedding ....................... 32
      5.4.2 Modified Spring Embedder ......................... 35
   5.5 Rendering .................................................. 36

6 Testing .......................................................... 37
   6.1 Testing Methodology ....................................... 37
   6.2 Testing File Processing ................................ 37
      6.2.1 Syntax Parsing ....................................... 37
      6.2.2 Multiple File Processing ......................... 38
   6.3 Graph Output ............................................... 39
      6.3.1 Differences in Layout Algorithm Aesthetics .... 39
      6.3.2 Processing Speed .................................. 40

7 Conclusions ...................................................... 41
   7.1 Requirements vs. Achievements ........................ 41
## CONTENTS

7.1.1 Function and Non-Functional Requirements ........................................... 41  
7.1.2 Achievement of Important Factors ...................................................... 41  
7.2 Problems to Overcome .............................................................................. 42  
  7.2.1 Speed vs. Aesthetics ............................................................................ 42  
  7.2.2 Complexity vs. Memory Usage ............................................................. 43  
7.3 Strengths in the System ........................................................................... 44  
7.4 Weaknesses in the System ........................................................................ 44  
7.5 Improvements ........................................................................................... 46  
  7.5.1 Application Analysis ........................................................................... 46  
  7.5.2 User Interface ....................................................................................... 46  
7.6 Was it Successful? ..................................................................................... 47  

A Design Diagrams ......................................................................................... 50  
  A.1 Overview of IRIS Application Structure ................................................ 50  

B User Documentation .................................................................................... 53  
  B.1 Prerequisites for Compilation .................................................................. 53  
  B.2 Command Line Execution ....................................................................... 53  
  B.3 Understanding the Visualisation ............................................................... 53  
  B.4 Manipulating the Graph .......................................................................... 54  
    B.4.1 Using the Mouse ............................................................................... 54  
    B.4.2 Using the Keyboard ......................................................................... 55  
  B.5 Problems with IRIS ................................................................................. 56  

C Raw results output ....................................................................................... 57  

D Code ............................................................................................................ 60  
  D.1 File: Flex Input ....................................................................................... 61  
  D.2 File: Bison Input ..................................................................................... 63  
  D.3 File: iris_extractor.c ............................................................................. 69  
  D.4 File: iris_extractor.h ............................................................................. 72  
  D.5 File: iris_fudger.c ................................................................................. 73
<table>
<thead>
<tr>
<th>File</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.6 File: iris_fudger.h</td>
<td>75</td>
</tr>
<tr>
<td>D.7 File: iris_functionata.c</td>
<td>76</td>
</tr>
<tr>
<td>D.8 File: iris_functionata.h</td>
<td>77</td>
</tr>
<tr>
<td>D.9 File: iris_graph.c</td>
<td>77</td>
</tr>
<tr>
<td>D.10 File: iris_graph.h</td>
<td>79</td>
</tr>
<tr>
<td>D.11 File: iris_hashtable.c</td>
<td>80</td>
</tr>
<tr>
<td>D.12 File: iris_hashtable.h</td>
<td>82</td>
</tr>
<tr>
<td>D.13 File: iris_lde.c</td>
<td>82</td>
</tr>
<tr>
<td>D.14 File: iris_lde.h</td>
<td>93</td>
</tr>
<tr>
<td>D.15 File: iris_heap.c</td>
<td>94</td>
</tr>
<tr>
<td>D.16 File: iris_heap.h</td>
<td>98</td>
</tr>
<tr>
<td>D.17 File: iris_list.c</td>
<td>99</td>
</tr>
<tr>
<td>D.18 File: iris_list.h</td>
<td>100</td>
</tr>
<tr>
<td>D.19 File: iris_maxqueue.c</td>
<td>100</td>
</tr>
<tr>
<td>D.20 File: iris_maxqueue.h</td>
<td>101</td>
</tr>
<tr>
<td>D.21 File: iris_params.c</td>
<td>102</td>
</tr>
<tr>
<td>D.22 File: iris_params.h</td>
<td>104</td>
</tr>
<tr>
<td>D.23 File: iris_queue.c</td>
<td>105</td>
</tr>
<tr>
<td>D.24 File: iris_queue.h</td>
<td>106</td>
</tr>
<tr>
<td>D.25 File: iris_rbtree.c</td>
<td>106</td>
</tr>
<tr>
<td>D.26 File: iris_rbtree.h</td>
<td>114</td>
</tr>
<tr>
<td>D.27 File: iris_refine.c</td>
<td>115</td>
</tr>
<tr>
<td>D.28 File: iris_refine.h</td>
<td>126</td>
</tr>
<tr>
<td>D.29 File: iris_spring.c</td>
<td>127</td>
</tr>
<tr>
<td>D.30 File: iris_spring.h</td>
<td>132</td>
</tr>
<tr>
<td>D.31 File: iris_stack.c</td>
<td>132</td>
</tr>
<tr>
<td>D.32 File: iris_stack.h</td>
<td>133</td>
</tr>
<tr>
<td>D.33 File: iris_util.c</td>
<td>133</td>
</tr>
<tr>
<td>D.34 File: iris_util.h</td>
<td>134</td>
</tr>
<tr>
<td>D.35 File: iris_visualiser.c</td>
<td>134</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Text Editor screen shot of gedit. ................................................. 4
2.2 Source Code Browser example of LXR output. ................................. 4
2.3 Extract from a Ctags file. ......................................................... 4
2.4 Flowchart taken from Wikipedia. ................................................ 5
2.5 Sequence Diagram taken from Unified Modelling Language: Superstructure. 5
2.6 Extract from a Lex/Flex input file. ............................................... 8
2.7 Extract from a YACC/Bison input file. ........................................ 9
2.8 Fisheye view, taken from Munzner. ........................................... 11
2.9 Cluster graph, red circles indicate clusters. ................................. 11
A.1 Flowchart detailing IRIS program structure. .................................. 51
A.2 Mockup design of IRIS interface ............................................... 52
C.1 Debug graph using HDE for layout ............................................. 58
C.2 Debug graph using MSE for layout ............................................. 58
C.3 Debug graph using HDE with only 50 nodes .................................. 59
List of Tables

6.1 Table of processing times for HDE and MSE . . . . . . . . . . . . . . . . . . 40
B.1 Table of IRIS command line options . . . . . . . . . . . . . . . . . . . . . . 54
B.2 Table of IRIS keyboard actions . . . . . . . . . . . . . . . . . . . . . . . . . 55
Acknowledgements

I would like to thank Prof. John Fitch for supervising my project.
Chapter 1

Introduction

Source code visualisation is the use of graphical presentations to represent code in a way that makes it easier to understand (Ball and Eick, 1996). It has been proven that people are able to make connections, and notice features of complex information that they would otherwise be unable to do (Price, Small and Baekcer, 1992), which makes effective means of visualisation a worthwhile goal.

There has already been a lot of work done on the topic of program visualisation. Most research papers give solutions for visualisation of multiple aspects of the software, such as the code, process execution, call graphs and, in some cases, differences between revisions of software. In this project, only the call graphing aspect of software visualisation will be focused on.

1.1 Project Structure

An iterative approach to software development has been used throughout this project. Iterative development permits small improvements to the design as the implementation encounters problems. This is beneficial because it is unclear from the beginning exactly what problems will be encountered - some of which may force major redesigns in the system. For this document, a less formal writing style has been chosen, with a view to, hopefully, facilitate easier understanding of the requirements, design and implementation.

1.2 Aim

The aim of this project is to design and build a new call graph visualisation system for the language C, capable of displaying an input application in a manner as simple as possible to understand.
1.3 Objectives

The objectives of this project are:

- To research how current source code visualisation systems display call graph information.
- To design and implement a new way of representing the call graph of an application.
- To analyse the implementation in order to find out how the idea can be improved.
Chapter 2

Literature Survey

2.1 Source Code Visualisation

2.1.1 Common Methods of Visualisation

The most common method of visualising source code is using some form of text editor (see Figure 2.1). Using such tools dates back to the first terminals, with terminal computers capable of showing as few as 24 lines, each with 80 characters. Consoles have had good success as computer languages are designed for text input, where terminals are a good candidate. They have the advantage of being able to facilitate both the viewing and editing of source code exactly; that is, every detail of the software is described, what you see is what the program does. However, consoles are not without their drawbacks. Only small portions of the code are visible at any one time, all of which is very fine detail, meaning an overview of the application can only be created after all of the code has been read and understood. For most production systems the volume of code would be almost impossible for a single developer to retain in memory, so overviews are not really feasible. Most recent console/text based source code visualisation tools provide syntax highlighting, which makes the text a bit more graphical, by colouring various parts of the code depending on their meaning, thereby assisting viewing.

Another common source code visualisation tool is the Source Code Browser (see Figure 2.2). It works by producing HTML web pages that contain the source code, but all references to things such as functions, files, etc. are represented by hyperlinks, which direct the viewer to the function body, file, etc. The advantage of this is the same as text editors, with the addition of easier navigation or following function calls, similarly, the disadvantages are the same as those encountered in text editors.

Ctags is a utility that, when given source files, creates a tag file (see Figure 2.3) which contains entries for every function and object, that brings Source Code Browser like navigation to text editors. This provides the programmer with the abilities of Source Code Browsers, with none of their disadvantages.
CHAPTER 2. LITERATURE SURVEY

Flow charts (see Figure A.1) have been around since the 1940’s (Shneiderman, Mayer, McKay and Heller, 1977). They provide a layer of abstraction above that of the source code, and are typically created as part of the design, and used for implementation. Soon after their conception they were found to be very useful, and quickly became standard practise.

Three types of flow chart are the Macro, Detailed and Structured. All types, by representing the program at a higher level than the code, allow for easier understanding of the algorithms being implemented, enabling designers to design systems without being concerned with the details of the implementation language. The charts can also be kept around as part of the software documentation for maintenance programmers later in the development cycle. Each of the 3 aforementioned types of flow chart have their own advantages and disadvantages.

Macro flow charts have the advantage of being small and compact, giving a clear and concise overview of what the code is supposed to be doing, however, because of this, you cannot use the flow chart to directly write the code. Detailed flow charts give very precise descriptions of the process carried out by a program, which means they can be used to write the code, as well as being easily understood for maintenance purposes. However the Shneiderman et al. study showed that detailed flow charts do not aid the programmer, and require more pages to represent the same process as well written code. Finally, the Structured flow charts have the following advantages according to Nassi and Shneiderman (Nassi and Shneiderman, 1973):

1. The scope of iteration is well defined and visible.

```c
#define DBCA_ACTION_NONE 0
#define DBCA_ACTION_EMAIL 1
#define DBCA_ACTION_CMD 2

typedef struct _EmailApplet EmailApplet;

typedef struct _EmailData EmailData;

struct _EmailApplet

PanelApplet* applet;

int acctype;
```

Figure 2.1: Text Editor screen shot of gedit.

```c
nw_device_close nw.c /* void nw_device_close(nw_device *device) */

nw_device_exists nw.c /* gboolean nw_device_exists(nw_device *device) */

nw_device_list nw.c /* GSLIST *nw_device_list() */

nw_device_open nw.c /* nw_device *nw_device_open(gchar *path) */

nw_get_track nw.c /* gboolean nw_get_track(nw_device *device, nw_track *track) */
```

Figure 2.2: Source Code Browser example of LXR output.

Figure 2.3: Extract from a Ctags file
2. The scope of IF-THEN-ELSE clauses is well defined and visible; moreover, the conditions on process boxes embedded within compound conditionals can be easily seen from the diagram.

3. The scope of local and global variables is immediately obvious.

4. Arbitrary transfers of control are impossible.

5. Complete thought structures can and should fit on no more than one page (i.e. no off-page connectors).

6. Recursion has a trivial representation.

Structured flow charts closely represent the layout of the code to be created, however, they are geared more specifically towards structured programming languages.

Sequence diagrams (see Figure 2.5), part of the UML standard (Group, 2005), are designed for modelling behaviours in systems. Vertical lines in the diagram represent the various entities involved in the process being described while the horizontal lines show how the entities interact. The abstraction used means that a programmer can quite easily understand what the source code is doing, however, it does not give enough detail to decipher how it is done. Another drawback of this method is that it can only show processes and is therefore only suitable in situations where process paths are relatively simple and predictable.

### 2.1.2 Reasons why they Fail

Many other source code visualisation tools have been developed apart from the ones listed above, however, they are not accepted by the developer community for various reasons. Some of these reasons could be the following:
• **Lack of scalability:** When the visualisation systems are designed, they are only tested on small programs to demonstrate what the output of the program is like. When given a production system, which is often comprised of hundreds of thousands of lines of code, they tend to generate output which is difficult to understand.

• **Navigation:** The majority of visualisation systems create 2-dimensional diagrams. Not enabling the viewers to manipulate the diagram, such as move nodes, zoom, pan, etc. can lead to situations where the users would find it more useful for data to be arranged differently to the generated output, but are forced to use what they are given.

• **Restriction on data types:** Applications touted as software visualisation tools need to be able to represent any given program in a number of ways. For example, call graphs, execution traces and bottleneck analysis. Failing to provide these features leads to low uptake of the utility. This should not be an issue with this project, as its primary aim is to restrict visualisations to that of the call graph.

### 2.1.3 Important Factors

There are 5 factors that must be considered when designing a visualisation system if it is to be effective. Each of them have been described below:

• **Layout:** The layout of a graph is the way in which its nodes are positioned and how its edges are drawn. With any given graph, a good layout would be easier to understand when compared to a bad one. A good layout is one in which the nodes are organised into a regular pattern, so that it exhibits qualities such as symmetry and minimal crossed edges (Purchase, 1997).

• **Colour:** Price et al. mention five effective ways of using colour. It can be used to indicate the state of an algorithm, to unite multiple views, emphasise patterns or capture history. For example, often used nodes could be coloured in a brighter shade of a colour (to indicate importance) than lesser used nodes. There is a lot of freedom with respect to the range of colours available, however, there are some colours, or some combination of colours, that should be avoided in certain situations; for example, people with red-green colour blindness would not be able to distinguish between red text on green backgrounds (or vice versa) (Rigden, 1999), similarly, red on blue is difficult to read even for those without vision deficiencies. Colour also has meaning. Red is often used to indicate high intensity, or warn of some danger, green is used for the opposite. Where possible, keeping to well established conventions could help speed up the time it takes a user to become proficient with a visualisation system.

• **Navigation:** The exact way navigation of a visualisation is implemented would depend on the diagrams created, however, there are some aspects that are essential
regardless of the type of graph, such as the number of dimensions being used, etc. These are basic operations of pan and zoom. At the very least, there will be a requirement to zoom in to parts of the diagram to make it legible, which, for software of any real size would mean not all of the diagram is visible at once. If the system is to be interactive, it is possible to build upon these basic operations, and allow for the user to navigate to sections of the visualisation by, for example, clicking on some entity. With respect to the problem of the visualisation system scaling to large software systems, techniques may need to be developed to reduce the data exposed to the user, such as selectively showing information.

- **Organisation:** How information is organised in a visualisation is important as it directly affects a person’s ability to correctly interpret the graph. Graphs could be arranged such that their data is grouped into relevant/related sub-sections, enabling the viewer to understand parts of the graph without having to know the entire content.

- **Scalability:** Software where visualisation tools would be most useful are large pieces of software, in which reading all of the source code to understand operations would be infeasible. Unfortunately, these same systems are the hardest for a visualisation system to effectively represent. The only way to cope with large scale applications, is with good organisation, layout and navigation algorithms.

### 2.1.4 How Many Dimensions?

A source code visualisation is an N-dimensional graphical representation of the source code. There are 3 obvious values for N, ranging from 2, 3 and T. All of the graphs are represented by \( G = \{ V, E \} \) (where G is the graph, V is the set of vertices (nodes) and E is the set of edges). The dimension comes into effect when considering how they are rendered.

**2-Dimensional** graphs are generated by arranging all of the nodes on the same plane, and connecting them by edges. This has the advantage of the whole graph being visible at once, which allows for printing, and is similar to how most common visualisation systems work, meaning less time for a user to become acquainted with the system.

**3-Dimensional** graphs have their nodes existing on multiple planes. Doing this allows for a reduction in the number of intersected edges (in comparison to the same graph represented in 2-Dimensions). However, it will produce graphs where it could be difficult to view the entire graph at once with a 2D projection, making printing all of the information impossible. However, it gives way to more interactive viewing approaches, such as modelling the graph as a 3D scene, and allowing the viewer to manipulate and move through it.

**The T-Dimension** is that of "time". It can be incorporated into either 2 or 3 dimensional graphs. In the case of call graphing, time could be represented as the call trace through the program, that is, some indication is given as to what function is currently being executed, then by some interaction by an observer, the visualisation changes to indicate which function execution has passed to.
CHAPTER 2. LITERATURE SURVEY

2.2 A Visualisation Tool Decomposed

A visualisation tool can be broken down into four main components, *Lexical Analysis*, *Parsing*, *Internal Representation* and *Creating the Graph*. As you can see, this resembles the various stages in the process of compiling source code. If you take a compiler to be a program which reads in a computer program in some language S, and creates an output of the same computer program in a language T, then the visualisation tool could be considered a compiler, as, in this case, the source language S is ANSI C, and the target language T is some form of graph.

2.2.1 Lexical Analysis

Lexical analysis is the process of reading in an input stream of characters, and by following certain rules, producing an output stream of tokens. A token is a block of text that is assigned some special meaning, such as mathematical operators, numbers, etc.

Lexical analysers (lexers) often use finite state automatas to process input. A finite state automaton is a system which has a finite number of states that it can exist in, with rules for transitions between states. In the case of lexing, transitions would be caused by a character read from an input stream putting the automaton in a state closer to one of its finishing states, which would be when a token has been read in.

A commonly used utility for automatically creating a lexer is Lex (Lesk and Schmidt, 1975) (and its more recent incarnation Flex). Lex works by reading in an input file (see Figure 2.6), which lists all of the character combinations (specified by regular expressions) that should be made into tokens, along with some C code that should be run - for most applications, it is acceptable for this C code to simply be a return statement specifying the type of the token.

```
"for" { return T_FOR; }
[0-9]* { return T_INT; }
"&&" { return T_OP_AND; }
```

Figure 2.6: Extract from a Lex/Flex input file.

The output of Lex is C source code for a lexer that will process the language specified in the input. After setting up the stream to read from, calls to `yylex` retrieve the next token (if there is one).

2.2.2 Parsing

Parsing is the process of interpreting the tokens found by the lexer, to produce a parse tree representing the program. There are two main categories of parser:
• **Top Down:** Parsing is done starting at the top of the parse tree, and hypothesise the structure of elements, and then walk down the tree, matching the found nodes to a previous hypothesis.

Top down parsing is a breadth first search for matching grammar rules, in which all possible routes are explored at once. The advantage of this is that there is never a need to backtrack. However, breadth first has the disadvantage of requiring a look-ahead of at least 1 to decide which path to take when more than one is available.

An example of a top down parser is the Recursive Decent Parser. It is usually implemented by having a procedure for each of the production rules in the grammar, and calling them recursively until the ”goal” is achieved.

• **Bottom up:** Parsing begins at the bottom of the tree upwards. At each juncture, one of the possible routes to take to the root is chosen, and an attempt is made to match the input to the grammar rules.

This is a depth first search for grammar rule matches. Where there are more than one possible matches, one of them is chosen, and an attempt is made to continue matching until either the goal is achieved, or it fails to match, in which case, it will backtrack and try another route. This has the advantage of knowing what elements are available before choosing an appropriate grammar rule, but backtracking can slow down the process.

An example of a bottom up parser is the *Cocke-Younger-Kasami algorithm*. In this, every possible sub-string is matched against production rules. For any match M greater than 2 characters in length, it attempts to match every partition of M such that it matches a rule $P \rightarrow AB$, where A is the first half of the partition and B is the second, in this case, M is matched against P.

Yacc (Yet Another Compiler Compiler) and its GNU equivalent Bison are commonly used tools for generating parsers. Similar to Lex/Flex, YACC and Bison read in files that define the grammar they are parsing (see Figure 2.7), and produce a C program which contains a function `yyparse` which works with `yylex` to parse a language.

```plaintext
multiplicative_expression :
  cast_expression |
  multiplicative_expression '*' cast_expression |
  multiplicative_expression '/' cast_expression |
  multiplicative_expression '%' cast_expression |

;  
```

Figure 2.7: Extract from a YACC/Bison input file.
2.2.3 Internal Representation

The Internal Representation is the structure used to represent the software in memory, between the stages of lexing/parsing and final output. In a compiler, this is required to store every detail of the application, and as such, a tree structure is often used (Purtilo and Callahan, 1989). When visualising function calls, this level of detail is not required, and a graph like structure is more suitable. This would allow for the creation of the graph to follow on logically from the data recorded. Essential data items include:

- **Function Names**: Obviously, these are a requirement, so that links in the visualisation can be mapped to the source code.
- **Called Functions**: Which functions are called by any function X.
- **Groups**: If grouping functions by some criteria - the group a function belongs too.

A directed graph, defined mathematically is \( G = \{V, A\} \), where:

- **G** is the graph.
- **V** is the set of vertices.
- **A** is a set of ordered pairs of vertices.

With this in mind, there are two ways to represent a graph in memory on a computer. C structures or equivalents can be used by having the structures that store information about a function, as well as pointers to other structures, detailing called functions. Using structures allows the data structure in memory to directly mimic the visualisation, however, it does not provide a convenient way of checking if there are edges between the functions. A method that is better suited to that task is a matrix, where the rows and columns represent the functions and data in the matrix dictate the relationship between them - the problem with this approach is that it does not allow for extra information to be stored, so in reality a combination of the two may be necessary.

2.2.4 Creating The Graph

It is a relatively simple task for a person to hand draw a graph for a small computer system in such a manner that it is easy to understand, however, in production systems, there are likely to be many thousands of lines of code, as well as a large number of functions. This leads to large graphs, which are extremely difficult to draw by hand, requiring some form of automated drawing system. When the graphs reach thousands of nodes, such that the resolution of the monitor they are being observed on is no longer capable of legibly displaying the graph in its entirety, other methods of visualisation need to be explored.
The simplest way to deal with a very large graph is to partially draw the graph. Only part of the graph would be visible at once, but what is visible would be clearly presented. Unfortunately, the purpose of the graph is to allow the developer to gain an overall view of the system, so it is essential that all of the graph be displayed at once.

Modelling the graph in hyperbolic space (Munzner, 1998) (see Figure 2.8) overcomes the problems of partial drawing, as the whole graph is visible at all times. The central area of the graph is enlarged, making it easy to read, while the rest of the graph is shrunk in size, getting smaller as it approaches the edge of the display area.

Another method is to use multi-level drawings (Eades and Feng, 1996). These are well suited to clustered graphs (see Figure 2.9), that is, graphs with areas of linked nodes and minimal links between clusters. With respect to source code, the clusters could be source code files.

**Graph Layout Strategies**

Many graph layout strategies have been developed, most of which are designed for a specific purpose (e.g. PCB layout). The ones discussed here are meant for generic purposes.

One class of graph drawing algorithm is called force directed (Quigley, 2006). With these algorithms, the graph is treated as a physical system. An example of this is would be treating edges like springs, and nodes as electrically charged particles. Particles with similar charges would be repelled from one another, separating them on the graph, but if they have an edge (spring) connecting them they will be held together, the distance dependant on the strength of the spring.

Gajer et al (Gajer, Goodrich and Kobourov, 2000) describe a multidimensional algorithm for graph layout. It has the advantage of allowing for realisation in multiple dimensions, so a single implementation would allow for both 2D and 3D graphs. The first step in the algorithm is to create an MIS filtration - a set of graphs containing the maximal
independent set, where for each $V_i$ in an MIS of cardinality $k$, as $i \rightarrow k$, $V_i$ contains the maximal independent set where distances between nodes are $\geq i$. It creates a scheduling function, then starting from the smallest graph in the filtration, processes them by finding an initial placement for its elements, then adjusting this based on assigned heats of elements. Hachul et al (Hachul and Jünger, 2006) test this algorithm, and even in the case of a large graph find its performance to be good.

Another algorithm that performed well in Hachul et al tests is called *High Dimensional Embedding* (Harel and Koren, 2002) (HDE). HDE, like the previous algorithm, draws graphs in a high dimension, then projects it onto either 2 or 3 dimensions. The algorithm begins by randomly selecting a node $p$ from the graph, and then, runs the following $m$ times (where $m$ is the number of dimensions), for $i = 1$ to $m$:

- Compute $i$th co-ordinate using a breadth first search.
- Store the positions of every node with respect to $i$.
- Choose the next pivot point.

From this, $m$ $n$-diminsional co-ordinates are found, which must then be projected onto a lower dimensional space (2 or 3) so that they can be realised.

**Graph Colouring**

As previously stated, colour is an important factor in the generation of an aesthetically pleasing graph. The colour of nodes can be used to represent information, or even just distinguish one node from another. One possible colouring scheme is to have all functions within a group given the same hue, but their saturation is dependent some other information, such as how many times they are called. In a situation like this, you are presented with the graph colouring problem, as for clarity, no two connected groups should have the same colour (if possible).

Finding the colouring of a graph, assuming one is possible is an $O(N^2)$ problem where $N$ is the number of nodes to be coloured. This complexity is not desirable, as it does not scale well as $N$ increases. A potential algorithm, is:

1. Push the current graph onto the stack.
2. Find a node where $E \leq C - 1$ (where $E$ is number of edges at node, $C$ is number of colours). If no such nodes exist and graph is not empty, stop as there is no possible colouring.
3. Remove the node and associated edges.
4. If the graph is not empty, go to step 1, otherwise continue.
5. Pop a graph off the stack.

6. For the node without a colour, choose a colour for it that is different to all other nodes it is connected to.

7. If there is a graph on the stack, go to step 5, otherwise end.

When the above algorithm terminates, it may have found a colouring for the graph. If it succeeds, it will return a valid colouring, a failure does not mean there is no colouring, it means the path it attempted did not have a colouring.
Chapter 3

Requirements

3.1 Functional Requirements

During the literature survey I was able to identify some requirements that are common across all source code visualisation systems, and also relevant for this one. Other requirements were decided upon after careful consideration of what the system was to be capable of, to be able to process data it is given, and present this in a manageable way to the user.

1. **The system must accept all syntactically correct C programs.**
   The application must accept all syntactically correct C programs as input.  
   Rationale: A user may want to visualise any possible C application, so it should be able to parse anything that would compile.

   (a) **The system should be easily modifiable to accept other languages.**
       It should be easy to change the application to accept other languages for future expansion.  
       Rationale: In the future it could be used for the visualisation of multiple languages, so forward planning needs to be incorporated into the design

2. **The system should be capable of producing 3-dimensional graphs.**
   The output visualisation should at the very least be generated in 3-dimensions, however the ability to render in 2-dimensions would also be desirable.  
   Rationale: The concept being explored by the system is 3D call graph visualisations.

3. **The system should create aesthetically pleasing graph layouts.**
   The output given to the user should be easily readable, to facilitate quick analysis of the data.  
   Rationale: The goal of the system is to make it easier for the user to understand the program being analysed - an overly complicated graph would hinder this.
4. **The system must allow manipulation of the graph.**
   As a minimum, the user must be able to manipulate the graph in the following ways:
   
   (a) Rotate the graph
   (b) Zoom in/out of the graph
   (c) Translate the graph

   *Rationale:* When the graph is large, it will not be possible to see it all at once from any one angle, so there has to be a way to change the viewing angle.

5. **The system must have some way of specialising visible content.**
   There must be a way to reduce the amount of the system being displayed to just elements that the user is interested in viewing.
   
   *Rationale:* A large system would create a large graph - which can be distracting if all the user wants to see is just a small portion of it, for example, which functions are called by a particular function.

6. **The system must not modify the input program.**
   Source code parsed by the program must not be modified, as the system is designed for observation only.
   
   *Rationale:* As a visualisation tool, it is expected by the user that the system will not modify the source code it is parsing, so in doing so it could cause application errors.

7. **The system should be executed using a command line interface.**
   
   The application should be accessible from a command line.
   
   *Rationale:* Similar developer tools are command line based, so users are already accustomed to it. At the same time, using a command line interface would enable the system to be a drop in replacement for tools such as the compiler, to make it easier to integrate its use into a project.

### 3.2 Non-Functional Requirements

1. **The system should be able to process large input applications in a reasonable time.**
   
   A large application can be considered anything above 10,000 functions, and a reasonable time as under a minute.
   
   *Rationale:* The larger a system is, the more likely you are to want to use a visualisation system. So it must be capable of processing them quickly.

2. **The system should be able to process large input applications using a reasonable amount of memory.**
   
   A large application can be considered anything above 10,000 functions, and a reasonable amount of memory as under 50Mb.
3.3 Data Requirements

1. **The system must use reasonable representations for all data sets.**
   All types of data stored in the system must be done in such a way that minimal memory is wasted.
   
   *Rationale: Keeping system resource usage to a minimum would increase the potential user base for the application.*

2. **The system must be able to store information about functions.**
   The information the system must be able to store is the following:
   
   (a) Function names
   (b) Function calls

   Other information may also be stored as long as it is relevant to source code visualisation - and the information will be presented in some way to the user.
   
   *Rationale: At the very least you need to know the names of functions, and the functions they call.*

3. **The system must be able to store a graph.**
   As well as function information, the system must be able to store data in a way that supports the graph layout process.
   
   *Rationale: Without a reasonable graph representation, the process of generating a graph layout could be slowed considerably.*
Chapter 4

Design

IRIS parses the source code of an application, and produces some 3D output. This process is much like that of a compiler - which is why a similar structure was chosen, which should allow IRIS to be easily modified to work with other languages (see Figure A.1).

4.1 Stage 1 : Parsing

The first step in the process is parsing the input program to extract information. To achieve this, a parser will be made with the provision of being able to parse all C programs, but will only update the graph data structure when a function definition or a function call is found. To speed up the development of the parser, two utilities designed for making lexical analysers and parsers will be used.

**Flex**, is a lexical analyser generator tool, compatible with its predecessor **Lex**. It accepts an input file defining all the valid lexical tokens in the language, and generates C source code for locating these tokens on an input stream. An example of Flex input (for the C language) can be seen in listing 4.1.

```
Listing 4.1: Example Flex input
"auto" { count(); return(AUTO); }
"break" { count(); return(BREAK); }
"case" { count(); return(CASE); }
"char" { count(); return(CHAR); }
```

The parser is generated by **Bison**, a utility that accepts an input file describing a context-free grammar, and produces a C application which is a LALR(1) parser. Example input for Bison can be seen in listing 4.2.

```
Listing 4.2: Example Bison input
additive_expression : multiplicative_expression
```

17
The Flex and Bison grammar files will be sourced online, from the URLs stated below:

- **Flex**: http://www.lysator.liu.se/c/ANSI-C-grammar-l.html
- **Bison**: http://www.lysator.liu.se/c/ANSI-C-grammar-y.html

Each file of the input program will be processed independently of the rest, so to handle cross-file references, a layer has to exist between the graph data structures and the parser to remember references to functions where the definitions have not yet been found.

When a function call is found, a search will be performed to find out if the function being called is known (present in the graph). If it is, then the function call is recorded in the graph - and if no match was found, the reference is added to a queue for post-processing.

Post-processing will use the algorithm given in listing 4.3.

```
Listing 4.3: Post-processing algorithm

for each reference in the queue
{
    if the function name is known then
    {
        add the edge to the graph.
    }
    otherwise
    {
        ignore reference.
    }
}
```

### 4.2 Stage 2: Graph Layout

Once stage 1 has completed, IRIS now has information on all of the functions in the input files. However, before graph layout can begin, it needs to be stored in such a way that easily facilitates this task. There are two straightforward ways of representing this data:

1. **Adjacency Matrix**: An NxN matrix is used, where rows represent functions, and values in columns indicate which functions are called.

2. **C Structures**: Each function could be stored in a structure, with pointers to other structures to show which functions it calls.
A sparse adjacency matrix was chosen as the storage method. This has the benefit of fast lookup times of a regular NxN adjacency matrix, but due to its sparse nature, can provide significant reductions in memory usage, depending on the structure of the graph being represented. On the other hand, checking for edges requires a linear search, but the function call and processing overhead would be insignificant - especially when compared to potential gains in memory usage.

### 4.2.1 Layout Algorithms

IRIS requires 4 different algorithms for generating graph layouts, two of which are designed to produce a layout, one is designed to refine layouts, and the last is designed to remove special cases that cause problems with the refinement algorithm.

**High Dimensional Embedding**

*High Dimensional Embedding (HDE)* is a spectral graph drawing method, which calculates a layout using the eigenvectors of the laplacian matrix associated with the graph. The algorithm works by doing the following:

1. Create a matrix $\chi$ containing the embedding of the graph in 50 dimensions. This is done by choosing up to 50 pivot nodes, and plotting the $i$th dimension of nodes as their graph theoretic distance from the $i$th pivot node.
2. Mean shift $\chi$ - which has the effect of centering the graph around the origin.
3. Calculate $\chi^T$ the transpose of $\chi$.
4. Calculate the degree matrix $D$.
5. Calculate $U$, the $D$-normalised eigenvectors of the adjacency matrix.
6. Project the graph onto $k$ dimensions by calculating the first $k$ co-ordinates for nodes using the first $k$ eigenvectors $\chi^T U_{k_i}$.

The advantages of this algorithm are:

- The complexity of this algorithm is stated as $O(m \cdot |E| + m^2 \cdot n)$ where $m = 50$. Meaning it grows linearly with the size of the graph. Stated typical execution times are approximately 3 seconds for $10^5$-node graphs.
- It accommodates easy selection of the number of dimensions that the final output should be. That is, setting $k = 2$ would produce a 2-dimensional graph.
However it does fail in some respects, which are summarised below:

- It models nodes as infinitly small points in space, which means that when it comes to display nodes, they could be positioned too close to each other, leading to overlaps.
- By utilising a spectral drawing technique, it inherits their inability to generate a layout for some graphs.
- 2-Dimensional layouts may not be as good as force directed methods would produce.
- It is designed for drawing undirected graphs.

There are two potential bottle necks in the HDE algorithm. During step 1 of its execution, the algorithm states the use of a Breadth First Search to calculate the graph theoretic distances - however - as all of the graph weights will be positivly weighted, the more efficient Dijkstra’s algorithm is used in its place.

Dijkstra’s algorithm is a greedy algorithm, which calculates the shortest path between a vertex \( s \) and all over verticies. When the algorithm begins, a list of distances is stored, where \( s \rightarrow s = 0 \), and \( s \rightarrow \text{all other verticies} \) = infinity. When the algorithm completes, the shortest path to all verticies from \( s \) will be known or be infinity if no such path exists. By utilising the \textit{ExtractMin} operation of a Fibonacci heap, the running time of Dijkstra’s algorithm is \( O(|E| + |V| \cdot \log |V|) \).

From steps 2 onwards, there are many matrix multiplication operations performed. There are 3 possible cases:

1. **Known matricies**: The structure of matricies are known and would always be the same.
2. **Matrix-vector**: Multiplication is being performed between a matrix and a vector.
3. **Matrix-matrix**: Two matricies are being multiplied.

The straight forward matrix-matrix multiplication algorithm (see listing 4.4) can be used for all 3 cases, but does not provide optimum performance.

Listing 4.4: Matrix-matrix multiplication

```plaintext
matrix : m1, m2, result
for i = 0 to m1.m
    for j = 0 to m2.n
        result(i, j) = 0
        for k = 0 to m2.m
```
\texttt{result} (i,j) += m1(i,k) \times m2(k,j)

\textbf{return} \texttt{result}

Performance can be improved for known matrices by adjusting the algorithm to ignore matrix elements that are known to be irrelevant. Or with matrix-vector multiplication, but using an algorithm designed to do so such as the one stated in listing 4.5, which calculates matrix-vector results faster than the previous example.

\textbf{Listing 4.5: Matrix-vector multiplication}

\texttt{matrix}: \texttt{m1}, \texttt{result}  
\texttt{vector}: \texttt{v}  
\textbf{for} i = 0 \textbf{to} ml.m  
\{  
\texttt{result} (i,0) = 0  
\textbf{for} j = 0 \textbf{to} ml.n  
\{  
\texttt{result} (i,0) += m1(i,j) \times v(i)  
\}  
\}  
\textbf{return} \texttt{result}

\textbf{Fast Node Overlap Removal}

\textbf{Fast Node Overlap Removal (FNOR)} is an algorithm which is designed to overcome one of the disadvantages of HDE - nodes overlapping due to being positioned to close to one another. FNOR only works on one dimension at a time, so nodes are only separated along an axis if it is deemed to have the worst overlap.

The FNOR algorithm works as follows:

1. Generate event queue - Create a list of events which signal the beginning and end of each node for the scanline algorithm. This event queue needs to be sorted (using quicksort for speed) into an order representing the dimension being processed.

2. Generate overlap constraints - This is done using a scan line algorithm, which checks if the edges of nodes overlap. The scan line is stored in a red-black binary tree which provides $O(\log|V|)$ insert and remove functions.
3. Satisfying the overlap constraints - Constraints are merged into blocks of active constraints, meaning all nodes in the block are at least gap distance apart, adding offsets to blocks to allow mergers.

4. Solving overlap constraints - Some blocks may need to be split, modified and merged again, due to later merges in the satisfying process invalidating previous merges.

The first three steps in the algorithm are sufficient to produce a layout, the fourth is only required if you want to ensure the optimal layout. For the purpose of optimal speed, this final step will be omitted, giving a complexity of $O(n + m \cdot \log m)$, where $m$ is the number of constraints (at most $|V|$) and $n$ is the number of variables (at most $|V|$).

The advantages of using FNOR are:

- It is very fast.
- It allows you to specify the minimum distance between nodes.
- It can work in any number of dimensions.

This is perfect to use as a post-processing step for the output of HDE, however the scanline part of the algorithm fails (enters into an infinite loop) if two nodes have the same value along a dimension, i.e. $a = (1, 1, 1)$ and $b = (1, 2, 3)$, it would fail as the $X$ co-ordinate of both $a$ and $b$ is $1$.

**Fudger**

**Fudger** is an algorithm I created myself to overcome the infinite loop problem with FNOR. Before the graph is processed with FNOR, it would be adjusted by Fudger to remove duplicate values.

The primary data structure used is a modified version of a red-black binary tree. When inserting a value into the tree, if the value being inserted is already present in the tree, instead of inserting a new node, it increments a counter in the tree, and records which graph node the new value was being inserted for.

With this modification, the algorithm is as follows:

1. Insert an entry into the tree for all graph nodes.

2. Perform an in-order tree walk, restoring graph node position values. When a tree node with count greater than $1$ is found, process the node like so:

   (a) Set $\Delta = [\text{Value of next tree node}] - \alpha$, where $\alpha$ is the smallest representable value.
(b) Set $\delta = \Delta \div \text{count}$. If $\delta$ is exactly 0, then set increment the value of all the tree nodes after the current one by 1, and return to (a).

(c) Set $\Gamma = \text{[Value of previous tree node]} + \alpha$.

(d) Set the $i$th position to $\Gamma + i \cdot \delta$.

This makes sure there are no duplicate position values, as duplicates are shifted by a small amount. The complexity of this algorithm is $O(|V| + \log|V|)$.

Advantages of this algorithm are:

- It spreads out nodes between the previous node position and the next node position - so only overlapping nodes are modified, helping to keep the layout generated by a layout algorithm.
- It is fast.

Spring Embedder

The **Spring Embedder** is a force-directed graph layout algorithm. It works by modelling the graph as a physical system of magnetically charged rings, which repel each other, and edges are modelled as logarithmic springs pulling rings together.

Pseudocode for the algorithm can be found in listing 4.6.

**Listing 4.6: Pseudocode for a Spring Embedder**

```plaintext
for iter = 1 to iter_max
{
    for each node
    {
        force = 0;

        for each other node
        {
            force += repulsive force between this node to other node
        }

        for each edge
        {
            force += attractive force between this node and edge node
        }

        node.velocity = delta * force
    }
}
```
In this implementation, attractive and repulsive forces will be calculated by formulae \[4.1\] and \[4.2\] respectively.

\[
f_a(\text{distance}) = 2.0f + \log(\text{distance}) \quad (4.1)
\]

\[
f_r(\text{distance}) = 1.0f + \sqrt{\text{distance}} \quad (4.2)
\]

There are \(|V|^2 - |V|\) operations to calculate the repulsive forces, and \(|E|\) operations to calculate attractive forces, giving a complexity of \(O(|V|^2 + |E|)\) per iteration, and Eades suggests 100 iterations as a suitable count for most graphs.

This is clearly not fast enough, so I have attempted to speed it up by borrowing an idea from computer graphics. Volume Elements (Voxels) are used in rendering to represent an area on a regular grid in 3-dimensional space. They speed up rendering when ray tracing by allowing you to calculate if a ray intersects a voxel, before doing calculations to check for intersections on more complicated objects.

Voxels have been used to reduce the number of repulsive force calculations, by splitting the world up into cuboids \(n \times n \times \infty\) in size. Now, when calculating the repulsive forces acting on a node \(\sigma\), formula \[4.2\] is used on all nodes in the same voxel as \(\sigma\), and then used on the distance from \(\sigma\) to the center of the surrounding 8 voxels. A voxel’s repulsive force is taken to be \(f_r(\text{distance}) \cdot x\), where \(x\) is the number of nodes in the voxel.

Pseudocode for calculating the voxelid of a node is given in listing 5.10.

**Listing 4.7: Pseudocode for calculating a voxel id**

```plaintext
voxel_size = 5
grid_size = 0xFFFFFFFF / voxel_size
half_grid_size = grid_size / 2

column = ceiling of (x / voxel_size);
if x less than 0 then
    column = half_grid_size - column
else
    column = half_grid_size + column

row = ceiling of (y / voxel_size);
if y less than 0 then
    row = half_grid_size - row
else
    row = half_grid_size + row

return (row * grid_size) + column
```

Using this modification, and assuming that the number of nodes in a voxel is bound from
above by some value \( k \), the number of calculations per voxel becomes \((k^2 - k + 8)\) done for \((|V| \div k)\) voxels, giving a new complexity of \(O((|V| \div k) + |E|)\).

### 4.3 Stage 3 : Visualisation

#### 4.3.1 Graphics Library

The visualisation of the graph is to be created using OpenGL and the GLUT library. This provides a convenient easy-to-use 3-dimensional API for drawing, which is available freely on most platforms. It also provides fast, hardware accelerated rendering - advantageous when creating a realtime interactive environment.

The problem with using OpenGL is that it puts all the strain of visualisation onto the graphics card of the system, meaning a powerful - potentially expensive card is required to visualise large systems.

#### 4.3.2 Interface Design

There are 4 pieces of information that need to be presented to the user. How this will be done is summarised in the list below:

- **Functions**: Functions will be represented by spheres in the scene. Their names will be displayed in text labels located at some offset the sphere.
- **Function Calls**: When one function calls another, the connection will be displayed by a cylinder connecting the two spheres representing the functions.
- **Groups**: Groups will not be explicitly identified in the graph. Instead members of the group will be identified, through the use of colour. Functions that belong in the same group, will be given the same hue (chosen using the HSL colour system).
- **Call Counts**: Again, this is information that will not be explicitly given in the graph. The luminance value of a sphere will be set linearly according to the number of times a function is referenced, allowing the user to - at a glance, gauge relative reference counts.

Figure \( \text{A.2} \) is a labelled mockup detailing what the above would look like in reality (func3 represents a function that is called more times than the others). As this is a 3-dimensional model, the interface can be manipulated as designed in the requirements, using the keyboard and mouse.
4.3.3 User Interaction

Almost all computers likely to be running IRIS will be equipped with both a keyboard and a mouse. Most 3-dimensional first-person shooter games use a combination of the W, A, S and D keys and the mouse the navigate. As well as being a familiar arrangement for interaction, it is also convenient for movement through 3D space. The keyboard will move the viewer through the world while the mouse will allow the user to rotate the graph to view it from different angles.

On top of this, the visualisation will accept user clicking to modify what they are viewing - i.e. double clicking on a node will show/ghost out all nodes that are not directly called by, or call the function in question.
Chapter 5

Implementation

This chapter will discuss the implementation of the design given, giving details on any interesting parts of the code, as well as details on data structures and the chosen language.

5.1 Programming Language

There were many languages to choose from when it came to the implementation. Many calculations will be performed during the running of the system, so the chosen language had to be as fast as possible - which rules out some interpreted languages as well as have bindings for the OpenGL library.

In the end, C was the language used for implementation, as it has bindings for OpenGL, gives the developer a lot of control over the memory being used, allowing for quick implementation. However, the crucial point in favour of C was that the utilities Flex and Bison produce C source files - reducing the amount of work required on the parsing stage of operations to a minimum for the developer.

5.2 Graph Data Structure

Graph data is stored in a structure that contains a list of vertices, and a sparse matrix of edges (see listing 5.1). Each of the vertex structures contains a pointer to a iris_function_info structure, which contains all the recorded data about the function being represented. By organising data in this fashion, it has created a model-view type organisation, keeping core data separate from visualisation information.

Listing 5.1: Graph Data Structure

```
typedef struct iri_graph_sparse_element
{
    int col;
};
```
Adding functions or function calls are the only actions that populate this data structure.

- **Adding a function**: A new `iris_graph_vertex` object is created to store the vertex information. The sparse graph structure is also allocated extra memory to represent
the new vertex.

- **Adding a function call**: A new edge is included in the sparse graph.

### 5.2.1 The Sparse Graph

There were two representations of the sparse graph considered during implementation, which are described below.

- **\(<\text{Row}, \text{Column}, \text{Value}>\) tuple**: The edges could be represented by a list of structures containing the row, column of cells in the matrix containing data. For very sparse graphs, this is the most compact form, giving excellent memory usage. It does, however, have the disadvantage of requiring either two searches (possibly hashtable to find tuples for a particular row, and a linear search to match column) or a complex data structure to store them in order to retrieve the matrix values.

- **Hashtable-like buckets**: In this case, there is an array, where the \(i\)th element corresponds to row number \(i\) in the matrix, and each array element is a bucket containing structures of the form \(<\text{Column}, \text{Value}>\). This would provide good compression for the kind of sparse graph that may be encountered in call graphing, as each vertex is likely to have at least one edge, so the row number is not duplicated as with the previous representation.

The second representation was decided upon, because it does not have the duplicate data the first representation will have in this situation, and only requires one search to find elements - a linear search of the bucket for that vertex.

### 5.3 The Parser

The parsing of C source files is handled by the parser generated by Flex and Bison using the input files in listings D.1 and D.2 respectively.

The extractor is the module of the code that handles retrieving function information from the input files. When it processes input, it will only add a function to the graph if that function is actually defined somewhere in the input files. This stops functions such as those in the Standard C Library (e.g. malloc and free) from showing up in the graph. On the other hand, this complicates the task of reading in information, but how this is handled is described in the following sections.

#### 5.3.1 Processing a new file

In IRIS, files are referred to as groups to keep generality in the event of changing the language being parsed. When a new group is found, a new group structure (see listing 5.2)
is created, and set this as the current group, meaning that from then onwards, any function definitions found will be assigned to this group.

Listing 5.2: Function Group Data Structure

typedef struct _iris_function_group {
    char *name;
} iris_function_group;

Currently, the group structure only records one attribute of the group, but it would be trivial to give it the capacity to store more, for example, if IRIS were to be extended to object-orientated languages, you know have class names as well as files, so you may want to add another member to *iris_function_group* specifically for storing file paths.

5.3.2 Processing a function definition

When a new function definition is found, a new function structure (see listing 5.3) is created to store its information.

Listing 5.3: Function Information Data Structure

typedef struct _iris_function_info {
    char *name;
    int ref_count;
    int public;
    int line; // the line the function is on.
    iris_function_group *group;
} iris_function_info;

Just like the data structure for groups, this can also be easily expanded in the future to accommodate new pieces of information - for example, the function signature.

This structure is then pushed onto the top of a stack, and the current function id is set to -1 so indicate that we are currently not inside a function.

5.3.3 Processing a function call

When a function call is found, the first action performed is to check if we are inside a function by checking the current function id is greater than -1. If it is not, when we are
not inside a function, and the function on the top of the stack is popped, and added to the graph (see listing 5.4).

Listing 5.4: Handling function calls

```c
void iris_extractor_add_call(iris_extractor *ext, char *lexeme)
{
    int v;
    iris_extractor_func *func;

    if (ext->current_func == -1)
    {
        func = (iris_extractor_func*)iris_stack_pop(ext->func_stack);
        iris_extractor_add_func(ext, func->lexeme, func->line);
        free(func->lexeme);
        free(func);
    }

    v = iris_hashtable_get(ext->funcht, lexeme);

    if (v >= 0)
    {
        iris_graph_add_edge(ext->graph, ext->current_func, v);
        ext->graph->vertex[v].function->ref_count++;
    }
    else
    {
        ext->refs_count++;
        ext->refs = (iris_extractor_ref*)realloc(ext->refs,
        sizeof(iris_extractor_ref) * ext->refs_count);
        ext->refs[ext->refs_count - 1].caller = ext->current_func;
        ext->refs[ext->refs_count - 1].caller_group = ext->current_group_num;
        ext->refs[ext->refs_count - 1].callee = strdup(lexeme);

        if (v == HASHTABLE_NOT_FOUND)
            iris_hashtable_insert(ext->funcht, lexeme, -1);
    }
}
```

An attempt is then made to locate the function being called in the hashtable of known functions. If the function being called is found, a new edge is added to the graph. Otherwise, a new reference structure is created which stores the function we are currently in, and the name of the function being called, so that references can be resolved after processing has completed.
The reason for waiting till this point to include functions in the graph, is that HDE is designed to work with connected graphs (meaning a path exists from one node to any other node), so to prevent problems, only functions which have calls, or are called are included. The former are inserted into the graph while adding the relevant edges, and the later are handled by resolving references.

5.3.4 Resolving references

After all files have been read in, the processes of resolving references begins. The list of references is iterated over, and references found to known functions are included into the graph. Listing 5.5 shows the code for this.

Listing 5.5: Implementation of reference resolving

```c
void iris_extractor_resolve.refs (iris_extractor *ext)
{
    int i, v;
    for (i = 0; i < ext->ref_count; i++)
    {
        v = iris_hashhtable_get (ext->funtch, ext->refs[i].callee);
        if (v != HASHTABLE_NOT_FOUND)
        {
            iris_graph_add_edge (ext->graph, ext->refs[i].caller, v);
            ext->graph->vertex[v].function->ref_count++;
        }
    }
}
```

5.4 Generating the graph layout

5.4.1 High Dimensional Embedding

Almost all the the operations in the HDE process are performed on matrices - so it was important to create a convenient but efficient representation. The one implemented is shown in listing 5.6.

Listing 5.6: Matrix representation

```c
typedef struct iris_hde_matrix
{
    float **v; // matrix memory
```
int m; // number of columns
int n; // number of rows
int col; // active column
}

iris_hde_matrix;

The first 3 elements of the structure are self explanatory, however, the last one is a little more ambiguous. In some situations, a calculation is performed on a whole matrix of size MxN, but later on, only column vectors are required from it. To save time and memory, the col data item exists to specify if the matrix should be considered a column vector - and if so, which column to use.

There are 4 matrix operations performed, matrix-matrix, matrix-vector, special matrix and scalar-matrix multiplication. As mentioned in the design, due to the nature of memory access and cache design, matrix-vector multiplication is more efficient if you use a slightly modified version of the standard algorithm. My implementation of the regular matrix-matrix multiplication is shown in listing 5.7 and the modified matrix-vector version in listing 5.8.

Listing 5.7: Matrix-matrix multiplication

iris_hde_matrix *iris_hde_matrix_mult(iris_hde_matrix *X1,
    iris_hde_matrix *X2)
{
    int i, j, k;
    iris_hde_matrix *R;

    R = iris_hde_matrix_create(X1->m, X2->n);

    for (i = 0; i < X1->m; i++)
    {
        for (j = 0; j < X2->n; j++)
        {
            R->v[i][j] = 0;

            for (k = 0; k < X2->m; k++)
            {
                R->v[i][j] += X1->v[i][k] * X2->v[k][j];
            }
        }
    }

    return R;
}
CHAPTER 5. IMPLEMENTATION

Listing 5.8: Matrix-vector multiplication

```c
iris_hde_matrix *iris_hde_matrix_mult_vector(iris_hde_matrix *m,
  iris_hde_matrix *v)
{
  iris_hde_matrix *r;
  int i, j;
  r = iris_hde_matrix_create(m->m, 1);

  for (i = 0; i < r->m; i++)
  {
    r->v[i][0] = 0;
    for (j = 0; j < m->n; j++) r->v[i][0] += m->v[i][j] * v->v[j][0];
  }

  return r;
}
```

General Purpose GPU Parallelism

The Graphics Processing Unit (GPU) of modern day computers are now capable of being used for general purposes - that is, for uses other than rendering graphics. Many graphics operations involve matrix calculations, meaning they are designed to do them extremely quickly, which could benefit processing times for IRIS.

By splitting up the matrix multiplication tasks into two sections, it would be possible to execute half on the CPU, and the other half on the GPU, hopefully, giving an overall reduction in computation time.

The simplest method for doing matrix multiplications on a GPU is allow it to consider the problem as the merging of two textures. Two texture maps are created, with the colour values set to the values in the matrices being multiplied - passed to the GPU, and made to merge. The act of merging two textures together yields the same result as multiplying the matrices representing them.

In the end, it was decided that adding such functionality to IRIS would increase the complexity of the program too much for the amount of benefit it would provide. Instead more time was invested on discovering the bottle necks in the matrix code, which was found to be a call to `bzero`, to populate the matrices with zeros when they are initialised. Once removed, computation times for large matrices became so small, that it nullified the advantages of parallelism.
5.4.2 Modified Spring Embedder

The modification to the *Spring Embedder* as discussed in the design is to add the concept of voxels. Nodes exist inside these voxels - where the ID of the voxel they are in is calculated by the code shown in listing [5.9].

Listing 5.9: Voxel ID computation

```c
int iris_spring_get_voxel(iris_graph *graph, int v)
{
    int gridsz, hgridsz, col, row;

    gridsz = 0xFFFFFFFF / SPRING_VOXEL_SZ;
    hgridsz = gridsz / 2;

    col = (int)ceil(graph->vertex[v].trans[0] / SPRING_VOXEL_SZ);
    col = (graph->vertex[v].trans[0] < 0) ? (hgridsz - col) :
         (hgridsz + col);

    row = (int)ceil(graph->vertex[v].trans[1] / SPRING_VOXEL_SZ);
    row = (graph->vertex[v].trans[1] < 0) ? (hgridsz + row) :
         (hgridsz - row);

    return ((row * gridsz) + col);
}
```

The grid size is taken to be 0xFFFFFFFF as this is the largest value an unsigned 32 bit integer can handle, which is what is used by the OpenGL depth buffer (at the time of writing). Also, you may notice that the voxel ID is computed using the location of the node, meaning not all of them will have content.

To keep memory usage to a minimum, only non-empty voxels are stored in memory. A binary tree is used to store active voxels, as it allows for fast insertion and removal of voxels from the world, as well as quick retrieval of stored voxel structures (see listing [5.10]).

Listing 5.10: Voxel structure

```c
typedef struct _iris_spring_voxel
{
    int voxelid;

    float center[3];

    iris_queue *verts;
} iris_spring_voxel;
```

The list of nodes inside a voxel is stored in a queue, in the *verts* data item. Room for development of the algorithm has been built into the data structure with the presence of the *center* data item, which is used as the center of the voxel during computation. It would
be possible to extend the algorithm to give what could possibly be a better layout if this value was updated to reflect the center of mass in a voxel - at a cost of extra complexity.

5.5 Rendering

Rendering is done using OpenGL. When it comes to display the graph, an OpenGL window is created, with keyboard and mouse event handlers attached to accept user input. The visual graph (i.e. the spheres and cylinders) are created, compiled and stored in an OpenGL list for fast rendering during manipulation. The only time this model is updated is when the user selects to view a sub section of the graph.

Choosing the hue and saturation of nodes is the only calculation to be performed at this stage in program execution, using the following process:

- Check if the node belongs to a group already assigned with a hue. If it does not, select a new hue by incrementing the hue selector. Only hue values 10 to 360 are permitted, as values less than that can leave nodes too dark to see on the black background.

- The saturation of the node is calculated using the equation 5.1, where ref_count is the number of times the function being represented was referenced in the input application, and maxcnt is the value to compare this to.

\[
MIN\_SAT + ((\text{ref\_count}/\text{maxcnt}) \times (1.0f - MIN\_SAT)); \quad (5.1)
\]
Chapter 6

Testing

6.1 Testing Methodology

A black box method has been chosen to test IRIS, as intermediate stages in the process are all dealing with data structures that are too complicated to accurately describe in this document.

6.2 Testing File Processing

6.2.1 Syntax Parsing

Although IRIS only requires function definition and call information - for successful extraction it will need to be able to process all valid C statements. It was found that the parser failed to parse valid programs under certain circumstances.

The grammar incorrectly parsed preprocessor statements - it would accept all statements that did not involve brackets, meaning:

- \#define TEST 123 is accepted.
- \#define F(X) X is not accepted.

To test if the grammar would work for all code examples, a highly obfuscated program was used as input, sourced from ioecc.org: The International Obfuscated C Code Contest. The code is given in listing 6.1. IRIS failed to parse it correctly.

Listing 6.1: Obfuscated C program from IOCCC.org winner 1987 `wall`

```
#define iv 4
#define v ; ( void
```
# define XI(xi) int xi [iv*’V’];
# define L(c, l, i) c [d(1); m(i)];
#include <stdio.h>

int +cc, c, i, ix=’t’, exit(), X=’\n’*’d’; XI(VI) XI(xi) extern (*vi[])(), (*signal()()), (char*V, cm, D[’x’], M=’\n’, I, *gets()); L(MV, V, (c=’d’, ix)) m(x){v signal(X/’1’, vi[x]); d(x) char*x; (v) write(i, x, i); } L(MC, V, M+1) xv() {c>=i?m(c/M/M+M): (d(&M), m(cm)); } L(mi, V+cm, M) L(ml, V, M) M() {c=c&M; X=cm; m(ix); }

LXX() { gets(D)| (vi[iv])(); c=atoi(D); while(c>=X) {c=X; d(”m”); } V=”ivxlcdm” +iv; m[ix]; LV() {c=cc; while((i=cc[*D=getchar()])>1) i?(c?(c<i&&l(-c-c, ”%d”), l(i,”%d”)):l(i,”(a”)):l(D,”%e”)):c;i&&l(X,””); l (-i,”%c”); m(iv-!(i&&l)); } L(ml, V, ’f’) li() {m(cm+isatty(i=I)); } ii() {m(c=cm = ++I)v pipe(VI); cc=xi+cm++; for (V=”jWYmdFlXW”; V; V++) xi[*V ’=’;
=, c, xi[*V’=’]=xi[*V]=c+1; cc[-I]==-ix
v } close(*VI); cc[M=3M] main() {
(vi)(); for (; v) write(VI[I], V,M)); l(xl, lx) char*lx; {v) printf(lx, xl)v ffflush (stdout); } L(xx+V, l, c=’X/cm,) ix) int (*vi[])() ={i, li, LXX, LV, exit, 1, l, d, l, d, m, M, M, M, M, ml, MV, xx, xx, xx, xx, MV, mi};

It does however correctly parse simpler code examples such as the one given in listing 6.2 so it is unclear - due to the complexity of 6.1 if it is IRIS at fault, or syntactically flawed input.

Listing 6.2: Example input

int func(int x)
{
    int i;
    int a[10];
    zero(a, sizeof(a));
    for (i = 0; i < x - 1; i++)
    {
        if (a[i] < x)
        {
            return 1;
        }
    }
    return 0;
}

6.2.2 Multiple File Processing

In this section, the reference resolving algorithm implemented by IRIS is tested to ensure it can properly analyse an application spanning multiple files. Code listings 6.3 and 6.4 detail the two files being parsed. IRIS was able to resolve the information successfully.
Listing 6.3: Multi-file input, file 1

```c
int main()
{
    foo();
}

int bar()
{
    exit(0);
}
```

Listing 6.4: Multi-file input, file 2

```c
int foo()
{
    bar();
}
```

### 6.3 Graph Output

#### 6.3.1 Differences in Layout Algorithm Aesthetics

There are two layout algorithms implemented in IRIS, HDE and MSE. To test the way in which graphs are rendered by the two algorithms, a debug graph can be generated by IRIS (by running it with the `-debug` option) and rendered with either algorithm. Figures C.1 and C.2 show the output of HDE and MSE respectively. As you can see, using HDE gives a much more readable output.

Figure C.3 shows how HDE performs when given 50 nodes. As you can see, it creates clusters of nodes which are overlapping each other, making it difficult to read. This graph is however, the debug graph simulating one function calling 49 others - and nothing else - which is a worst case scenario for HDE, and a fairly unlikely situation in real world use.
CHAPTER 6. TESTING

6.3.2 Processing Speed

Execution times for HDE and MSE were recorded over a few runs and averaged, to discover how fast the algorithms are in comparison. Table 6.1 shows the results.

Table 6.1: Table of processing times for HDE and MSE.

<table>
<thead>
<tr>
<th>Node Count</th>
<th>HDE Time (s)</th>
<th>MSE Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.01</td>
<td>0.18</td>
</tr>
<tr>
<td>1000</td>
<td>0.19</td>
<td>5.54</td>
</tr>
<tr>
<td>10000</td>
<td>2.49</td>
<td>Stopped after 2 hours</td>
</tr>
<tr>
<td>100000</td>
<td>21.41</td>
<td>n/a</td>
</tr>
</tbody>
</table>

As you can see from the table, the increase in the processing time for MSE is exponential, making it become very slow very quickly. HDE however, exhibits a linear growth in processing time, that enables even very large graphs to be processed in a reasonable time.
Chapter 7

Conclusions

7.1 Requirements vs. Achievements

7.1.1 Function and Non-Functional Requirements

At the beginning of the project, an aim and 3 objectives were stated. The aim was to develop a new call graph visualisation system, which has been achieved, while the 3 objectives were met in the process.

Of the requirements stated in chapter 3 all have been met except for two, non-functional requirement 1 which has only been partially satisfied:

The system should be able to process large input applications in a reasonable time.

This is a result of having to sometimes use the MSE algorithm in place of HDE. The other requirement it failed to meet is functional requirement 1 which has only been partially satisfied:

The system must accept all syntactically correct C programs.

As it does not properly parse all pre-processor directives, as well as failing to parse the obfuscated code given to it.

7.1.2 Achievement of Important Factors

In the literature survey, some important factors were identified that need to be considered when developing a source code visualisation system. Below is a discussion to give the extent to which IRIS has met these requirements.
CHAPTER 7. CONCLUSIONS

- **Layout:** The idea of using spheres and cylinders to represent the graph give the user with a familiar looking interface even though it is 3-dimensional, instead of the usual 2. However, the positioning of the nodes in the diagram are decided at run time based on the input. Sometimes a nice, easily understood layout will be generated, and other times it will be a relative mess, and harder to understand - with HDE, this is true for the same input, due to a bug in the implementation. With this in mind, especially as it can be considered a very opinionated heuristic, it can be concluded that the layout will not always be good a one, and so, IRIS fails in this respect.

- **Colour:** To unobtrusively represent some data, colour is used. It was mentioned in the literature survey that use of colour had to be careful to allow colour blind people to successfully use the system. I believe this has been achieved, as the main focus of the application - which is the call graph, is visible regardless of node colouring. In fact, when running IRIS it is possible to disable the use of colour.

- **Navigation:** IRIS provides a familiar method for moving through 3D worlds, which should allow people to navigate the graph easily. By enabling the user to hide portions of the graph, and concentrate on smaller parts, navigation is made even easier. IRIS could be considered to be compliant with this requirement.

- **Scalability:** For given inputs, IRIS is able to use the extremely fast HDE algorithm, which allows IRIS to scale to show very large applications (e.g. $10^5$ functions), in a reasonable time - however it is unlikely that a computer will have the computational power to display this all at once, so interaction speed could suffer. Unfortunately, some graphs cannot be positioned by HDE, and MSE is used instead, which does not scale well at all, taking an unacceptably long time for $10^4$ nodes.

### 7.2 Problems to Overcome

The biggest problems encountered during the development of the implementation were ones that had two conflicting requirements that did not have a simple solution to achieve both simultaneously.

#### 7.2.1 Speed vs. Aesthetics

As mentioned in the literature survey, there are many graph layout algorithms available - however most have the drawback of being slow. Every algorithm generates a slightly (potentially wildly) different layout for the same graph, some of which are aesthetically pleasing and some produce layouts that are practical for our purposes.

The major difference between them is their execution times. One of the requirements was to make the generation of the graph as fast as possible, which has been achieved by utilising the *High Dimensional Embedding* algorithm. However, as is evident in the testing, HDE
has a tendency to spread most nodes out, but leave some in a cluster in the middle of the graph, making them difficult to see.

In an attempt to produce graphs with better aesthetics, a spring embedder based algorithm was implemented which had the property of creating a more uniform spread of nodes throughout the graph. This reduced the amount of incoherent clustering in the layout, but left nodes too far apart.

Node spacing could be reduced in the spring embedder by increasing the attractive force of springs, and should be fairly trivial to implement. An attempt was made to speed up the spring embedder algorithm, as described in the design chapter, however it was still much slower than HDE, so IRIS could not rely on it solely.

Eventually, the Fast Node Overlap Removal algorithm was implemented to try and remove as much of the overlap in HDE generated layouts as possible, making it an acceptable choice as the primary layout generator - given the fact that the spring embedder is very slow.

### 7.2.2 Complexity vs. Memory Usage

Complexity in this sense is taken to be the complexity (running time) of the algorithms used as opposed to how complicated the implementation is.

IRIS was developed on a low end laptop with what is (at the time of writing) considered to be only a mediocre amount of system memory and hard disk space - and only Intel\textsuperscript{TM} graphics accelerator, instead of a discrete graphics card. It was decided that this would be an acceptable target as a minimum system requirement.

The initial implementation of IRIS did not take into account system resource usage, and was coded to keep complexity to a minimum, and speed to a maximum. This was achieved, but at the cost of massive memory usage. For example, a graph containing 10,000 nodes would use approximately 381Mb for the adjacency matrix alone, and with function information on top of that - IRIS was utilising almost all of the system memory, forcing the computer to use the page file heavily, and slowing down computation. On the other hand, data structures were simple, and so was accessing them.

By using slightly more complex data structures for storage of information, memory usage was decreased significantly - for the same 10,000 node graph it now only uses around 8Mb of memory across the entire application. In some situations these new data structures decreased complexity, but in other parts - such as checking if an edge exists between two nodes (which instead of a direct memory access, now requires a linear search, whose actual execution time will depend on the structure of the graph).

In testing, it was found that the modifications had no negative effect on the execution time - in fact, execution was sped up due to the system not having to use the page file.
7.3 Strengths in the System

There are two main strengths with the system, where the requirements have been met as defined:

1. **Execution Time:** When using the primary graph layout algorithm, the execution time has been kept well below the 1 minute mark for a simulated program with $10^5$ functions in it - easily within acceptable limits.

2. **Navigation:** By using a navigation arrangement commonly found in 3-dimensional PC games, navigating the graph is fairly intuitive.

7.4 Weaknesses in the System

Although HDE has proven itself to be a fast and efficient means to generate a layout for a graph, it does have some weaknesses:

1. **Incoherent Layouts:** For some graphs, especially those with sub 200 nodes, HDE often generates a graph that is not optimal for conveying information to the user. One could argue that this essentially renders IRIS useless in some situations, however, to counteract this, IRIS allows the user to select bits of the graph and ghost other parts, so any part obscuring others can be made temporarily invisible.

MSE generates graphs with a more uniform distribution of nodes, spread apart, which can also in some situations lead to graphs that are difficult to read. For example, a graph with many functions called by one, would result in one node being positioned in the middle of a sphere of surrounding nodes - a perfectly valid layout, but not optimal for our purposes.

2. **Inability to layout some graphs quickly:** A major problem with HDE - as with all algorithms of this type, is its inability to generate a layout for some graphs, such as some trees, which leave the algorithms unable to select proper points of origin.

With HDE, this creates a graph layout where node locations are either extremely large, or invalid.

As soon as HDE has generated its output, the graph is checked to ensure that all values are valid - as whenever it fails to generate a layout, at least one dimensional value is set to $NaN$. If an error is detected, it will rerun the layout process, this time using the MSE algorithm.

3. **Bugs in HDE implementation:** A bug in the implementation of the HDE algorithm has brought about two weaknesses with the system, which have inadvertently been worked around in attempting to accommodate the aforementioned weaknesses.
However, they still cause instability in the system which make it crash randomly while generating graph layouts. These bugs are:

(a) **Changing graphs:** HDE calculates graph layouts directly using matrix multiplications, so given the same graph, should generate the same layout over and over again - however a bug in the implementation, most likely caused by matrix structures not having their memory zeroed before being used. Doing so is unfortunately not an option, as it severely damages efficiency. So it has been left in while (unsuccessful) attempts were made to work around the problem.

(b) **HDE only accepts graphs of 50 nodes or more:** In an attempt to tweak the performance of the algorithms implementation, while modifying from one to the most recent version of it, changes were made that mean the matrices in the final stage of the computation are sized such that only layouts for graphs with at least 50 nodes can be computed.

4. **MSE execution time:** The MSE algorithm is very slow, which means that when HDE fails to compute a layout, the user could be waiting a long time before any output is given. This, unfortunately, cannot be improved as it is in the nature of force-directed algorithms.

5. **Only connected graphs accepted:** The HDE and MSE algorithms were designed to work with connected graphs (meaning a path exists from a node $a$ to a node $b$ where $a, b \in V$) and fail otherwise. Due to time constraints I was unable to design and implement a fix for this, but it would have to analyse the adjacency matrix after file processing to find the separate sub graphs - most likely using a breadth first search in the following manner:

   (a) Pick a node not already visited, and begin a breadth first search, recording all nodes reachable from this one.
   (b) Consider all visited nodes to be in the same graph, and process with HDE/MSE.
   (c) Return to the first step, ignoring the processed nodes.
   (d) Offset all of the graphs from the origin so that none of their nodes overlap.

The worst case complexity of a breadth first search is $O(|V| + |E|)$, meaning there would be a negligible increase in processing time.

6. **Node labelling:** Currently nodes are labelled by simply putting text at a specific offset from the node itself. The problem with this is two fold. To begin with, because IRIS is working with 3D space, OpenGL, renders the scene such that rotations can cause a node to come in front of the its label - so you can see a node, but not its label. The other problem is that it can become cluttered with relatively dense areas in the graph, making it impossible to read the text.

A better way to do this would be to have an algorithm that dynamically calculated the position of function names, based on the amount of free space available - and is able to use labels and arrows to reduce clutter.
7.5 Improvements

Having used IRIS, I have found that there are a few features that it could be extended to have, that would make it a much more useful tool. These can be split into Application Analysis and User Interface improvements, as discussed below.

7.5.1 Application Analysis

In this section, improvements to the data extraction abilities of IRIS are discussed. The ideas may take the project slightly beyond its initial aim, but would be useful nonetheless.

1. **Dynamic call graphs:** Currently IRIS will only show static call graphs - however it would be a useful feature to IRIS to be able to show the dynamic call graph as well. It is however, feasible for this to be impossible to predict until runtime (e.g. if it computes a location to jump to in code using a function of the time), but it can be done in some cases (e.g. there is an array of function pointers, and that array is used to call functions).

2. **Spot potential bottlenecks:** It may be possible to spot some areas of the application - using static analysis - that would be bottlenecks during execution. For example, a function that has a `for` loop with many iterations, that is called several times, could potentially be a bottleneck in execution. Highlighting this in the visualisation could help in speeding up the execution of input programs.

3. **Function signatures:** At the moment only function names are available to the user. Extending this to give the whole function signature would enable the use of IRIS as a function reference tool.

4. **Support for other languages:** With a little refactoring the extractor portion of IRIS can be made to accommodate the use of multiple language parsers, making the tool useful to a wider audience (although this is outside of the initial targets of the project).

7.5.2 User Interface

1. **Better labelling:** Implementing an algorithm as previously discussed for label placement would improve usability.

2. **Caller-callee display:** Currently, the edges between graph nodes show information in an undirected fashion. However, what is really being represented is directional information, one function calls another, and ideally the visualisation would be modified to convey this.
3. **Call tracing:** By keeping track of the order in which function calls were found, it would be possible to trace execution paths through the program. This could be shown to the user, for example, by an animation, showing how control passes from one function to another. It would also require information to be recorded about selection statements, and allow the user to specify which branch they wish to follow.

4. **Higher level views:** With large systems, there would be many nodes being displayed at once, leading to clutter on the screen - and with systems with limited graphics power, very slow rendering times. If there was the ability to abstract to a higher level (e.g. show groups), and only show the functions of selected groups, it would overcome this issue.

5. **Makefile parsing:** IRIS can almost be run as a drop-in replacement for make - taking this one step further, by including Makefile parsing support, it would be possible to make running IRIS as simple as running `iris` instead of `make`.

6. **Click to edit:** A useful feature of IRIS would be to allow the user to select a node, and bring up their favourite code editor, displaying the function ready for editing.

7. **Serialise graphs to file:** Being able to serialise the graph to file so that it can be observed at a later date without having to redo the entire processing could be a useful feature.

8. **Ambient rotation:** Adding a slow rotation to the graph could make it easier to absorb all of the information it is displaying. This could either be an automatic rotation in arbitrary directions, or by allowing the user to spin the graph using the mouse, and leaving it rotating in around the same axis.

### 7.6 Was it Successful?

The answer to this question is both yes and no. Conceptually, it works. Displaying the call graph in 3-dimensional space, is successful in presenting the information to the user. The idea of making it interactive also works, as it is easy to navigate the graph, and does not take very long to learn to do so.

On the other hand, in practise, the fast graph layout techniques are not stable enough to support the rendering of all graphs that may be given to them, making the system unable to perform quickly at times. This is clearly a major drawback, and one that will have to be overcome for the system to become really useful. Graph layout generation is an active research area, and new algorithms are constantly being conceived, so it is a very real possibility for IRIS to achieve all of its goals in the future.
Bibliography


Harel, D. and Koren, Y. (2002), Graph drawing by high dimensional embedding, *in* ‘Proceedings Graph Drawing 2002 (GD’02)’.


Appendix A

Design Diagrams

A.1 Overview of IRIS Application Structure
Figure A.1: Flowchart detailing IRIS program structure.
Figure A.2: Mockup design of IRIS interface
Appendix B

User Documentation

B.1 Prerequisites for Compilation

To compile IRIS you will need GCC (or comparable compiler) with the development files for the OpenGL and GLUT libraries.

B.2 Command Line Execution

IRIS is run with a command line shown below. Pressing Esc while observing the visualisation will exit the application.

\[ iris \{options\} file1 \{file2 file3 \ldots filen\} \]

Table B.1 shows all of the command line values that can be passed to IRIS during normal operation. All of the files specified in the command line must be syntactically correct C source or header files for processing to be completed successfully.

B.3 Understanding the Visualisation

Making sense of the data presented to you by IRIS is fairly simple. There are 4 pieces of information given to the user:

1. **Available Functions**: Each function in the input application is represented by a sphere in the graph.

2. **Function Calls**: Each line in the graph represents a function call.
### Table B.1: Table of IRIS command line options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-w</td>
<td>Show the visualisation in a window.</td>
</tr>
<tr>
<td>-c</td>
<td>Disable group colour coding.</td>
</tr>
<tr>
<td>-v</td>
<td>Enable verbose output.</td>
</tr>
<tr>
<td>-s</td>
<td>Disable summary output.</td>
</tr>
<tr>
<td>-o</td>
<td>Apply orthonormalisation to the graph when using HDE.</td>
</tr>
<tr>
<td>-d DIM</td>
<td>Project the graph into DIM dimensions (DIM = ([2, 3])).</td>
</tr>
<tr>
<td>-r MIN</td>
<td>Set the minimum gap between nodes to MIN.</td>
</tr>
<tr>
<td>-m MAX</td>
<td>Set the reference count needed for maximum saturation.</td>
</tr>
<tr>
<td>-a ALG</td>
<td>Specify the layout algorithm to use [auto</td>
</tr>
<tr>
<td>-debug</td>
<td>Use the built in debug graph instead of file inputs.</td>
</tr>
</tbody>
</table>

3. **Function Grouping:** Functions are considered to be in the same group if they were extracted from the same source file. This is shown on the graph by spheres having the same hue.

4. **Call Count:** The greater the saturation of the colour of the sphere, the more times it is called.

### B.4 Manipulating the Graph

#### B.4.1 Using the Mouse

The mouse is the primary device for performing the following actions:

- **Rotating the view:** This is done by holding down the right mouse button, and moving the mouse. By default, the center of the graph is the point used as the origin for rotation, however, by holding down `Ctrl + Alt` while left clicking on a node, that node becomes the center of rotation. Selecting a blank area will reset the origin.

- **Translating the view:** By holding down the middle mouse button while moving the mouse, the whole graph will be translated around the viewing area.

- **Displaying sub-graphs:** If you want to highlight functions that call, or are called by a specific function, double clicking with the left mouse button on the node will highlight it in white - and ghost any unrelated functions. Return to the normal view by double clicking on blank space.
• **Highlighting nodes:** You can select to view particular nodes by either *Ctrl* clicking on the ones you want, or by holding down the left mouse button and creating a selection rectangle.

• **Zooming in/out:** Using the scroll wheel you can zoom in and out of the visualisation.

### B.4.2 Using the Keyboard

Table B.2 details what the keyboard inputs do.

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>Move forwards through space.</td>
</tr>
<tr>
<td>A</td>
<td>Move left through space.</td>
</tr>
<tr>
<td>S</td>
<td>Move backwards through space.</td>
</tr>
<tr>
<td>D</td>
<td>More right through space.</td>
</tr>
<tr>
<td>L</td>
<td>Show/hide function names.</td>
</tr>
<tr>
<td>F</td>
<td>Turn fullscreen mode on/off.</td>
</tr>
<tr>
<td>+</td>
<td>Zoom in.</td>
</tr>
<tr>
<td>-</td>
<td>Zoom out.</td>
</tr>
</tbody>
</table>
B.5 Problems with IRIS

There are a few bugs in the implementation of IRIS, that can cause random crashing during the graph layout generation. Most of the time, running IRIS again will present you with a graph.
Appendix C

Raw results output

The content on the following pages are screenshots of the output from IRIS. See section 6.3 of the paper for a description.
Figure C.1: Debug graph using HDE for layout

Figure C.2: Debug graph using MSE for layout
Figure C.3: Debug graph using HDE with only 50 nodes
Appendix D

Code

The following pages are the full source code for IRIS. Listings D.1 and D.2 show the input files to Flex and Bison respectively instead of the generated C files.
D.1 File: Flex Input

D L H F FS IS
[0–9] [a–zA–Z_] [a–fA–F–0–9] [Ee][+–]?[D]+ (u|U|l[L]*)

{ extern
  float
  for
  goto
  if
  int
  long
  register
  return
  short
  signed
  sizeof
  static
  struct
  switch
  typedef
  union
  unsigned
  void
  volatile
  while

#include <stdio.h>
#include "iris_bison.h"

void count();
void comment();
int check_type();
char *nnamdi;
#define ECHO
%
"/*
 "*/

"#define" "#include" 
"#pragma" 
"#ifdef" 
"#ifndef"

"auto" 
"break" 
"case" 
"char" 
"const" 
"continue" 
"default" 
"do" 
"double" 
"else"

"enum" 
"extern" 
"float" 
"for" 
"goto" 
"if"

int count();

{return (DEFINE);}

{return (INCLUDE);}

{return (PRAGMA);}

{return (IF);}

{return (IFDEF);}

{return (IFNDEF);}

{return (AUTO);}

{return (BREAK);}

{return (CASE);}

{return (CHAR);}

{return (CONST);}

{return (CONTINUE);}

{return (DEFAULT);}

{return (DO);}

{return (DOUBLE);}

{return (ELSE);}

{return (DEFINE);}

{return (EXTERN);}

{return (FLOAT);}

{return (FOR);}

{return (GOTO);}

{return (IF);}

{return (INT);}

{return (LONG);}

{return (REGISTER);}

{return (RETURN);}

{return (SHORT);}

{return (SIGNED);}

{return (SIZEOF);}

{return (STATIC);}

{return (STRUCT);}

{return (SWITCH);}

{return (TYPEDEF);}

{return (UNION);}

{return (UNSIGNED);}

{return (VOID);}

{return (VOLATILE);}

{return (WHILE);}

{return (STRING_LITERAL);}

{return (ELLIPSIS);}

{return (LEFT_ASSIGN);}

{return (RIGHT_ASSIGN);}

{return (ADD_ASSIGN);}

{return (SUB_ASSIGN);}

{return (MUL_ASSIGN);}
```c
int yywrap ()
{
    return (1);
}

void comment ()
{
    char c1 ;
    loop :
    if ((c1 = getchar ( ) ) == '/')
        goto loop;
    /* 
     * putchar ( c ) ;
     * unput ( c1 ) ;
     * goto , ;
   */
    if (c1 == 0)
        putchar (c1); /
}

int column = 0 ;
int line = 0 ;

void count ()
{
    int i ;
    yylval = strdup (yytext);
    for (i = 0 ; yytext [ i ] != '\n' ; i ++)
        if (yytext [ i ] == '\b')
            column -= 1 ;
        else if (yytext [ i ] == 't')
            column = 8 - column % 8 ;
        else
            column ++ ;
```
ECHO;
}

int check_type()
{
    /* pseudo code — this is what it should check */
    /* if (yytext == type_name) */
    /* return (TYPE_NAME); */
    /* */
    /* it actually will only return IDENTIFIER */
    return (IDENTIFIER);
}

int iris_lex_get_line()
{
    return line;
}

D.2 File: Bison Input

{%
#include <stdio.h>
#define YYSTYPE char*

int yylex(void);
void yyerror(char *msg);

#include <stdlib.h>
#include "iris_params.h"
#include "iris_function_data.h"
#include "iris_graph.h"
#include "iris_hashable.h"

#include "iris_stack.h"
#include "iris_extractor.h"
extern char *yytext;
extern int column;

void iris_bison_set_extractor(iris_extractor *ext);
void iris_bison_call(char *name);
void iris_bison_func(char *name);

iris_extractor *extractor;
%
%token IDENTIFIER CONSTANT STRING LITERAL SIZEOF
%token PTR OP INC OP DEC OP LEFT OP RIGHT OP LE OP GE_OP
%token AND OP OR
%token MUL ASSIGN DIV ASSIGN MOD ASSIGN
%token ADD ASSIGN LEFT ASSIGN RIGHT ASSIGN AND ASSIGN
%token XOR ASSIGN OR ASSIGN TYPE_NAME
%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE
%token VOLATILE VOID
%token STRUCT UNION ENUM ELLIPSIS
%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN
%token DEFINE INCLUDE PPRAGMA PPIF ENDIF IFDEF IIFDEF IENDIF

%start translation_unit

%{
primary_expression:
    IDENTIFIER
    | CONSTANT
    | STRING_LITERAL
    | "( expression )"
    :

postfix_expression:
    primary_expression
    | postfix_expression "[" expression "]"
    |

%}

%}
APPENDIX D. CODE

```plaintext
| postfix_expression '()' |
|-------------------------|------------------|
|                         | { | iris_bison_call($1); } |
|                         | | postfix_expression '( argument_expression_list ')' |
|                         | | { | iris_bison_call($1); } |
|                         | | postfix_expression ',' IDENTIFIER |
|                         | | postfix_expression PTR OP IDENTIFIER |
|                         | | postfix_expression INC_OP |
|                         | | postfix_expression DEC_OP |

argument_expression_list : assignment_expression |
| argument_expression_list ',' | assignment_expression |

unary_expression : postfix_expression |
| INC_OP unary_expression |
| DEC_OP unary_expression |
| unary_operator cast_expression |
| SIZEOF unary_expression |
| SIZEOF '(' type_name ')' |

unary_operator :
| '&' |
| '*' |
| '+' |
| '-' |
| '~' |
| '!' |

cast_expression : unary_expression |
| '(' type_name ')' cast_expression |

multiplicative_expression : cast_expression |
| multiplicative_expression '/' cast_expression |
| multiplicative_expression '%' cast_expression |

additive_expression : multiplicative_expression |
| additive_expression '+' |
| multiplicative_expression |
| additive_expression '-' |
| multiplicative_expression |

shift_expression : additive_expression |
| shift_expression LEFT_OP additive_expression |
| shift_expression RIGHT_OP additive_expression |

relational_expression : shift_expression |
| relational_expression '<' shift_expression |
| relational_expression '>' shift_expression |
| relational_expression '=' shift_expression |
| relational_expression '<=' shift_expression |
| relational_expression '>=' shift_expression |

equality_expression : relational_expression |
| equality_expression EQ_OP relational_expression |
| equality_expression NE_OP relational_expression |

and_expression : equality_expression |
| and_expression '&' equality_expression |

exclusive_or_expression : and_expression |
| exclusive_or_expression 'ˆ' and_expression |

inclusive_or_expression : exclusive_or_expression |
```
APPENDIX D. CODE

expression : assignment_expression
            | expression ',' assignment_expression

constant_expression : conditional_expression

conditional_expression : logical_or_expression
                        | logical_or_expression '?' ' expression ':' conditional_expression

format_expression : logical_and_expression

declaration : declaration_specifiers ';'
             | declaration_specifiers init_declarator_list ';'
             | preprocessor_statement

init_declarator_list : init_declarator
                     | init_declarator_list ',' init_declarator

init_declarator : declarator
                 | declarator '=' initializer

storage_classSpecifier :_TYPEDEF
                       |EXTERN
                       |STATIC
                       |AUTO
                       |REGISTER

specifier : VOID
          |CHAR
          |SHORT
          |INT
          |LONG

expr: expression
app

\begin{verbatim}
| FLOAT |
| DOUBLE |
| SIGNED |
| UNSIGNED |
| struct_or_union_specifier |
| enum_specifier |
| TYPE_NAME |
|
struct_or_union_specifier
: struct_or_union IDENTIFIER '{' |
| struct_declaration_list '}' |
| struct_or_union '{' struct_declaration_list '}' |
| struct_or_union IDENTIFIER |
|
struct_or_union
: STRUCT |
| UNION |
|
struct_declaration_list
: struct_declaration |
| struct_declaration_list struct_declaration |
|
struct_declaration
: specifier_qualifier_list struct_declarator_list ';
|
specifier_qualifier_list
: type_specifier specifier_qualifier_list |
| type_specifier |
| type_qualifier specifier_qualifier_list |
| type_qualifier |
|
struct_declarator_list
: struct_declarator |
| struct_declarator_list ',' struct_declarator |
|
struct_declarator
: declarator |
\end{verbatim}
APPENDIX D. CODE

; pointer
;   '∗'
;   type_qualifier_list
;   '∗'
;   pointer
;   '∗'
;   type_qualifier_list pointer
;

; type_qualifier_list
;   type_qualifier
;   type_qualifier_list type_qualifier
;

; parameter_type_list
;   parameter_list
;   ; ELLIPSIS
;

; parameter_list
;   parameter_declaration
;   ; ELLIPSIS
;

; parameter_declaration
;   declaration_specifiers declarator
;   declaration_specifiers abstract_declarator
;   declaration_specifiers
;

; identifier_list
;   IDENTIFIER
;   ; ELLIPSIS
;

; type_name
;   specifier_qualifier_list
;   specifier_qualifier_list abstract_declarator
;

; abstract_declarator
;   pointer
;   direct_abstract_declarator
;   pointer direct_abstract_declarator

; direct_abstract_declarator
;   '(' abstract_declarator ')
;   '('.
;   constant_expression ')
;   direct_abstract_declarator '
;   direct_abstract_declarator '
;   constant_expression ')
;   '('.
;   '('.
;   direct_abstract_declarator ')
;   direct_abstract_declarator ')
;   direct_abstract_declarator ')
;   direct_abstract_declarator ')
;   parameter_type_list ')
;

; initializer
;   assignment_expression
;   '{' initializer_list '}'
;   '{' initializer_list ', ' '}'
;

; initializer_list
;   initializer
;   ; ELLIPSIS
;

; statement
;   labeled_statement
;   compound_statement
;   expression_statement
;   selection_statement
;   iteration_statement
;   jump_statement
;   preprocessor_statement
;

; labeled_statement
;   IDENTIFIER ': ' statement
;   CASE constant_expression ': ' statement
;   DEFAULT ': ' statement
;

; compound_statement
;   '{' '}

APPENDIX D. CODE

```c
| '{ state_list '}
| '{ declaration_list '}
| '{ declaration_list statement_list '}

declaration_list
| declaration
| declaration_list declaration

statement_list
| statement
| statement_list statement

expression_statement
| ';
| expression ';

selection_statement
| IF '{ expression ' statement
| IF '{ expression ' statement ELSE statement
| SWITCH '{ expression ' statement

iteration_statement
| WHILE '{ expression ' statement
| DO statement WHILE '{ expression ' ';'
| FOR '{ expression_statement
| FOR '{ expression_statement expression_statement ');
| FOR '{ expression_statement expression_statement expression_statement ');

jump_statement
| GOTO IDENTIFIER ';
| CONTINUE ';
| BREAK ';
| RETURN ';
| RETURN expression ';

preprocessor_directive
| DEFINE
| INCLUDE
| PRAGMA
| PPIF
| ENDF
| IFDEF
| IFNDEF

preprocessor_statement
| preprocessor_statement expression
| preprocessor_statement statement
| preprocessor_directive

translation_unit
| external_declaration
| translation_unit external_declaration

external_declaration
| function_definition
| declaration

function_definition
| declaration Specifier declarator declaration_list compound_statement
| declarator declaration_list compound_statement
| declarator compound_statement

% void yyerror(char *msg)
{
    printf("\n\nError parsing source file!\n\n");
    exit(0);
}

void iris_bison_func(char *name)
{
D.3 File: iris_extractor.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "iris_params.h"
#include "iris_function_data.h"
#include "iris_graph.h"
#include "iris_hashtable.h"
#include "iris_stack.h"
#include "iris_extractor.h"

// Extern Declarations
extern void iris_bison_set_extractor(iris_extractor *ext);
extern int yyparse();
extern FILE *yyin;
extern int yyget_lineno (void);

// Private Functions

void iris_extractor_new_group(iris_extractor *ext, char *name)
{
    iris_function_group *grp;
    grp = iris_function_group_create(ext->data);
    grp->name = name;
    ext->current_group = grp;
    ext->group_funcs_count++;
    ext->group_funcs[ext->group_funcs_count - 1].funcs = NULL;
    ext->group_funcs[ext->group_funcs_count - 1].count = 0;
    ext->current_group.num = (ext->group_funcs_count - 1);
}

void iris_extractor_resolve.refs(iris_extractor *ext)
{
    int i, v;
    for (i = 0; i < ext->ref_count; i++)
    {
        v = iris_hashtable_get(ext->funct, ext->refs[i].callee);
        if (v != HASHTABLE_NOT_FOUND)
        {
            iris_graph_add_edge(ext->graph, ext->refs[i].caller, v);
            ext->graph->vertex[v].function->ref_count++;
        }
    }
}
```

```
void iris_extractor_add.func(iris_extractor *ext, char *lexeme, int line)
{
    iris_function_info *info;
    int v, vtx;
    ```
iris_extractor_grp_func *grpfunc;

info = iris_function_info_create(ext->data);
info->group = ext->current_group;
info->name = strdup(lexeme);
info->line = line;

if (strcmp(EXTRACTOR_ENTRY_FUNC, lexeme) == 0)
    iris_function_set_entry_point(ext->data, info);

v = iris_graph_add_vertex(ext->graph, info);

vtx = iris_hashtable_get(ext->funcht, lexeme);
if (vtx != HASHTABLE_NOT_FOUND)
{
    info = iris_extractor_add_func(ext, func->lexeme, func->line);
    info->group = ext->data->groups[func->group];
}
else
{
    name = ext->data->groups[func->group]->name;
    len = strlen(func->lexeme) + strlen(name) + 3;
    ch = (char*)malloc(sizeof(char) * (len + 1));
    sprintf(ch, "%s: %s\n", name, func->lexeme);
    iris_stack_push(ext->unused, ch);
}
free(func->lexeme);
free(func);
}

// Public Functions

iris_extractor *iris_extractor_create(iris_graph *graph,
    iris_function_data *data)
{
    iris_extractor *ext;

    ext->unused = iris_stack_create();

    while (!iris_stack_is_empty(ext->func_stack))
    {
        func = (iris_extractor_func*)iris_stack_pop(ext->func_stack);
        v = iris_hashtable_get(ext->funcht, func->lexeme);
        if (v != HASHTABLE_NOT_FOUND)
        {
            info = iris_extractor_add_func(ext, func->lexeme, func->line);
            info->group = ext->data->groups[func->group];
        }
        else
        {
            name = ext->data->groups[func->group]->name;
            len = strlen(func->lexeme) + strlen(name) + 3;
            ch = (char*)malloc(sizeof(char) * (len + 1));
            sprintf(ch, "%s: %s\n", name, func->lexeme);
            iris_stack_push(ext->unused, ch);
        }
        free(func->lexeme);
        free(func);
    }
ext =
    (iris_extractor *) malloc(sizeof(iris_extractor));
if (ext->links != NULL) free(ext->links);
    iris_hashtable_destroy(ext->funcht);
    iris_stack_destroy(ext->func_stack);
    iris_stack_destroy(ext->unused);
    free(ext);
}

void iris_extractor_new_func(iris_extractor *ext, char
    *lexeme)
{
    iris_extractor_func *func;
    func =
        (iris_extractor_func *) malloc(sizeof(iris_extractor_func));
    func->lexeme = strdup(lexeme);
    func->group = ext->current_group_num;
    func->line = yyget_lineno();
    iris_stack_push(ext->func_stack, func);
    ext->current_func = -1;
    ext->counter++;
}

void iris_extractor_add_call(iris_extractor *ext, char
    *lexeme)
{
    int v;
    iris_extractor_func *func;
    if (ext->current_func == -1)
    {
        func =
            (iris_extractor_func *) iris_stack_pop(ext->func_stack);
    }
    v = iris_hashtable_get(ext->funcht, lexeme);
    if (v >= 0)
D.4 File: iris_extractor.h

```c
int iris_extractor_process(iris_extractor *ext, char *file)
{
    iris_extractor_new_group(ext, file);
    ext->counter = 0;
    iris_bison_set_extractor(ext);
    yyin = fopen(file, "r");
    if (yyin != NULL)
    {
        yyparse();
        return ext->counter;
    }
    return -1;
}

void iris_extractor_resolve(iris_extractor *ext)
{
    iris_extractor_resolve_funcs(ext);
    iris_extractor_resolve.refs(ext);
}
```

---

### Definitions

```c
#define EXTRACTOR_HT_SIZE 100
#define EXTRACTOR_ENTRY_FUNC "main"
```

### Structures

```c
typedef struct _iris_extractor_link
{
    int includer;
    enum {WEAK, STRONG} type;
    union
    {
        int includee;
        char *name;
    };
} iris_extractor_link;
```

```c
typedef struct _iris_extractor_ref
{
    int caller;
    int caller_group;
    char *callee;
} iris_extractor_ref;
```
typedef struct _iris_extractor_func {
    char *lexeme;
    int group;
    int line;
} iris_extractor_func;

typedef struct _iris_extractor_grp_func {
    int *funcs;
    int count;
} iris_extractor_grp_func;

typedef struct _iris_extractor {
    iris_graph *graph;
    iris_function_data *data;
    iris_extractor_grp_func *group_funcs;
    int group_funcs_count;
    iris_extractor_ref *refs;
    int ref_count;
    iris_extractor_link *links;
    int link_count;
    iris_hashable *funcht;
    iris_function_group *current_group;
    int current_group_num;
} iris_extractor;

int current_func;
int counter;
iris_stack *func_stack;
iris_stack *unused;

} iris_extractor;

// Functions

iris_extractor *iris_extractor_create(iris_graph *graph,
                                      iris_function_data *data);
void iris_extractor_destroy(iris_extractor *ext);
void iris_extractor_new_func(iris_extractor *ext, char *lexeme);
void iris_extractor_add_call(iris_extractor *ext, char *lexeme);
int iris_extractor_process(iris_extractor *ext, char *file);
void iris_extractor_resolve(iris_extractor *ext);

D.5 File: iris_fudger.c

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include "iris.h"
#include "iris.function_data.h"
#include "iris.graph.h"
#include "iris.rbtree.h"
#include "iris.fudger.h"

// Private Functions
iris_rbtree_tree *iris_fudger_make_tree(iris_graph *graph, int dim)
{
    int i, error;
    iris_rbtree_tree *tree;
    iris_rbtree_node *node;

    tree = iris_rbtree_create();
    error = 0;

    for (i = 0; i < graph->vertex_count; i++)
    {
        node = iris_rbtree_insert_count(tree,
                                        graph->vertex[i].trans[dim], i,
                                        &graph->vertex[i]);

        if (!error && node->count > 0) error = 1;
    }

    if (!error)
    {
        iris_rbtree_destroy(tree);
        tree = NULL;
    }

    return tree;
}

void iris_fudger_order(iris_graph *graph,
                        iris_rbtree_node *node, iris_graph_vertex **vtxs, int *counter)
{
    int i;

    if (node->left != NULL) iris_fudger_order(graph, node->left, vtxs, counter);
    vtxs[(*counter)] = &graph->vertex[node->data];

    for (i = 0, (*counter)++; i < node->count; i++, (*counter)++)
    {
        vtxs[(*counter)] = node->ptrs[i];
    }

    if (node->right != NULL) iris_fudger_order(graph, node->right, vtxs, counter);
}

void iris_fudger_fudge(iris_graph_vertex **vtxs, int len, int dim, int begin, int end)
{
    float min, max, delta;
    int i;

    min = (begin > 0) ? vtxs[begin - 1]->trans[dim] + FLT_MIN : vtxs[begin]->trans[dim];
    max = (end < len - 1) ? vtxs[end + 1]->trans[dim] - FLT_MIN : vtxs[len - 1]->trans[dim] + 0.001f;

    delta = ABS((max - min) / (end - begin));

    if (begin > 0)
    {
        vtxs[begin]->trans[dim] = vtxs[begin - 1]->trans[dim] + delta;
    }
    else
    {
        vtxs[begin]->trans[dim] = delta * (end - begin);
    }

    for (i = begin + 1; i <= end; i++)
    {
        vtxs[i]->trans[dim] = vtxs[i - 1]->trans[dim] + delta;
    }

    void iris_fudger_scan(int len, iris_graph_vertex **vtxs, int dim)
    {
        int i, begin, end;
        float val;

        begin = -1;

val = -1;

for (i = 0; i <= len; i++)
{
    if (begin < 0 && i < len)
    {
        begin = i;
        val = vtxs[i]->trans[dim];
    }

    else if ((i == len && begin > -1) || (i < len && val != vtxs[i]->trans[dim]))
    {
        end = i - 1;

        if (end > begin)
        {
            iris_fudger_fudge(vtxs, len, dim, begin, end);
        }

        if (i < len)
        {
            begin = i;
            val = vtxs[i]->trans[dim];
        }
    }
}

int iris_fudger_error_check(iris_graph *graph, int dim)
{
    int d, i;
    d = 0;

    for (i = 0; i < graph->vertex_count - 1; i++)
    {
        if (graph->vertex[i].trans[dim] ==
            graph->vertex[i + 1].trans[dim])
        {
            return 1;
        }
    }

    return 0;
}

// Public Functions

int iris_fudger_process(iris_graph *graph, int dim)
{
    iris_rbtree_tree *tree;
    iris_graph_vertex **vtxs;
    int counter;

    tree = iris_fudger_make_tree(graph, dim);

    if (tree != NULL)
    {
        vtxs = (iris_graph_vertex**)malloc(sizeof(iris_graph_vertex*) *
            graph->vertex_count);

        counter = 0;
        iris_fudger_order(graph, tree->root, vtxs, &counter);

        iris_fudger_scan(graph->vertex_count, vtxs, dim);

        free(vtxs);
        iris_rbtree_destroy(tree);

        if (iris_fudger_error_check(graph, dim))
        {
            return 0;
        }
    }

    return 1;
}

D.6 File: iris_fudger.h

// Defines
#ifndef ABS
#define ABS(x) ((x < 0.0f) ? (x * -1) : x)
#endif

// Functions

int iris_fudge_process(iris_graph *graph, int dim);

D.7 File: iris_function_data.c

#include <stdlib.h>
#include "iris_function_data.h"

// Public Functions

iris_function_data *iris_function_data_init()
{
    iris_function_data *data;

data = (iris_function_data*)malloc(sizeof(iris_function_data));
    data->functions = NULL;
    data->function_count = 0;
    data->entry_point = NULL;
    data->groups = NULL;
    data->group_count = 0;

    return data;
}

iris_function_group *
iris_function_group_create(iris_function_data *data)
{
    iris_function_group *group;

group = (iris_function_group*)malloc(sizeof(iris_function_group));
    group->name = NULL;

    data->groups = (iris_function_group**)realloc(data->groups,
        sizeof(iris_function_group*) * data->group_count);
    data->groups[data->group_count - 1] = group;

    return group;
}

iris_function_info *
iris_function_info_create(iris_function_data *data)
{
    iris_function_info *info;

    info = (iris_function_info*)malloc(sizeof(iris_function_info));
    info->name = NULL;
    info->ref_count = 1;
    info->group = NULL;
    info->line = -1;

    data->functions = (iris_function_info**)realloc(data->functions,
        sizeof(iris_function_info*) * data->function_count);
    data->functions[data->function_count - 1] = info;

    return info;
}

void iris_function_set_entry_point(iris_function_data *
data, iris_function_info *entry)
{
    data->entry_point = entry;
}

void iris_function_destroy(iris_function_data *data)
{
int i;
for (i = 0; i < data->function_count; i++)
{
    free(data->functions[i]->name);
    free(data->functions[i]);
}
free(data->functions);
for (i = 0; i < data->group_count; i++)
{
    free(data->groups[i]->name);
    free(data->groups[i]);
}
free(data->groups);
free(data);

D.8 File: iris_function_data.h

// Structures

typedef struct _iris_function_data iris_function_data;
typedef struct _iris_function_group
{
    char *name;
} iris_function_group;
typedef struct _iris_function_info
{
    char *name;
    int ref_count;
    int public;
    int line;
    iris_function_group *group;
    struct _iris_function_data
    {
        iris_function_info **functions;
        int function_count;
        iris_function_info *entry_point;
        iris_function_group **groups;
        int group_count;
    };
    // Functions
    iris_function_data *iris_function_data_init();
    iris_function_group *iris_function_group_create(iris_function_data *data);
    iris_function_info *iris_function_info_create(iris_function_data *data);
    void iris_function_set_entry_point(iris_function_data *data, iris_function_info *entry);
    void iris_function_destroy(iris_function_data *data);

D.9 File: iris_graph.c

#include <string.h>
#include <stdlib.h>
#include "iris_function_data.h"
#include "iris_graph.h"
APPENDIX D. CODE

```c
// Public Functions

// IrisGraph init

int irisgraph_init(int vertex)
{
    IrisGraph* graph = (IrisGraph*)malloc(sizeof(IrisGraph));
    graph->vertex = vertex;
    graph->edges = NULL;
    graph->vertex_count = 0;
    //graph->edges_size = vertex;
    graph->edges = (IrisGraphSparse*)malloc(sizeof(IrisGraphSparse) * vertex);
    for (int i = 0; i < vertex; i++)
    {
        graph->edges[i].els = (IrisGraphSparseElement*)malloc(sizeof(IrisGraphSparseElement) * GRAPH_SPARSE_ROW_SIZE);
        graph->edges[i].els_count = -1;
        graph->edges[i].els_size = graph->edges_size = 0;
    }
    return graph;
}

// IrisGraph add vertex

int irisgraph_add_vertex(IrisGraph* graph, IrisFunction* func)
{
    int vindex = graph->vertex_count;
    graph->vertex_count++;
    graph->vertex = (IrisGraphVertex*)realloc(graph->vertex, sizeof(IrisGraphVertex) * graph->vertex_count);
    graph->vertex[vindex].trans[0] = 0;
    graph->vertex[vindex].trans[1] = 0;
    graph->vertex[vindex].trans[2] = 0;
    graph->vertex[vindex].function = func;
    graph->vertex[vindex].edges = NULL;
    graph->vertex[vindex].edge_count = 0;
    graph->edges_row_count++;
    if (graph->edges_size < graph->edges_row_count * GRAPH_SPARSE_ROW_SIZE)
    {
        graph->edges = (IrisGraphSparse*)realloc(graph->edges, sizeof(IrisGraphSparse) * graph->vertex_count);
        row = graph->edges_row_count - 1;
        for (int i = 0; i < vertex; i++)
        {
            graph->edges[row].els[i].els_count = 0;
        }
    }
    return vindex;
}

// IrisGraph get vertex

int irisgraph_get_vertex(IrisGraph* graph, int i)
{
    return graph->vertex[i];
}
```

void iris_graph_add_edge(iris_graph *graph, int v_caller, int v_callee)
{
    int tmp;
    if (v_caller < graph->vertex_count && v_callee < graph->vertex_count)
    {
        tmp = graph->vertex[v_caller].edge_count;
        graph->vertex[v_caller].edge_count++;
        graph->vertex[v_caller].edges =
            (int*)realloc(graph->vertex[v_caller].edges,
                          sizeof(int) * graph->vertex[v_caller].edge_count);
        tmp = graph->vertex[v_callee].edge_count;
        graph->vertex[v_callee].edge_count++;
        graph->vertex[v_callee].edges =
            (int*)realloc(graph->vertex[v_callee].edges,
                          sizeof(int) * graph->vertex[v_callee].edge_count);
        graph->vertex[v_caller].edges[tmp] = v_callee;
        graph->sedges->rows[v_caller].els[tmp].col = v_callee;
        graph->sedges->rows[v_caller].els++;
    }
}

void iris_graph_destroy(iris_graph *graph)
{
    free(graph->vertex);
    free(graph);
}

int iris_graph_edge(iris_graph *graph, int caller, int callee)
{
    int i;
    for (i = 0; i < graph->sedges->rows[caller].els_count; i++)
    {
        if (graph->sedges->rows[caller].els[i].col == callee)
        {
            return 1;
        }
    }
    return 0;
}

D.10 File: iris_graph.h

#define GRAPH_SPARSE_ROW_SIZE 3

typedef struct _iris_graph_sparse_element
{
    int col;
} // int val;
APPENDIX D. CODE

typedef struct _iris_graph_sparse_element {
    iris_graph_sparse_element *els;
    int els_count;
    int els_size;
} iris_graph_sparse_element;

typedef struct _iris_graph_sparse_row {
    iris_graph_sparse_element *els;
    int els_count;
    int els_size;
} iris_graph_sparse_row;

typedef struct _iris_graph_sparse {
    iris_graph_sparse_row *rows;
    int row_count;
    int row_size;
} iris_graph_sparse;

typedef struct _iris_graph_vertex {
    float trans[3];
    iris_function_info *function;
    int *edges;
    int edge_count;
} iris_graph_vertex;

typedef struct _iris_graph {
    iris_graph_vertex *vertex;
    unsigned int vertex_count;
    iris_graph_sparse *edges;
} iris_graph;

// Functions

iris_graph *iris_graph_init(int vertex);
int iris_graph_add_vertex(iris_graph *graph, iris_function_info *func);
iris_graph_vertex iris_graph_get_vertex(iris_graph *graph, int i);
void iris_graph_add_edge(iris_graph *graph, int v_caller, int v callee);
void iris_graph_destroy(iris_graph *graph);
int iris_graph_edge(iris_graph *graph, int caller, int callee);

D.11 File: iris_hashable.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "iris_hashable.h"

// Private Functions

int iris_hashable_hash_helper(char *c) {
    return ((c & 0x1F) + ((c++) != '\0') ? (iris_hashable_hash_helper(c) << 4) : 0));
}

int iris_hashable_hash(char *c, int size) {
    char *key = c;
    return (iris_hashable_hash_helper(key) % size);
}
# APPENDIX D. CODE

```c
void iris_hashable_bucket_destroy(iris_hashable_bucket *htb)
{
    iris_hashable_bucket *next;
    next = htb->next;
    free(htb);
    if (next != NULL)
        iris_hashable_bucket_destroy(next);
}

iris_hashable_bucket *
iris_hashable_get_bucket(iris_hashable *ht, char *lexeme)
{
    int bucket;
    iris_hashable_bucket *htb;
    bucket = iris_hashable_hash(lexeme, ht->count);
    htb = ht->bucket[bucket];
    while (htb != NULL)
    {
        if (strcmp(htb->lexeme, lexeme) == 0)
        {
            return htb;
        }
    }
    htb = htb->next;
}

return NULL;

// Public Functions

iris_hashable *iris_hashable_create(int size)
{
    iris_hashable *ht;
    int i;
    ht = (iris_hashable*)malloc(sizeof(iris_hashable));
    ht->count = size;
    ht->bucket = (iris_hashable_bucket**)malloc(sizeof(iris_hashable_bucket*)
* size);
    //bzero(ht->bucket,
        sizeof(iris_hashable_buckets) * size);
    for (i = 0; i < size; i++) ht->bucket[i] = NULL;
    return ht;
}

void iris_hashable_destroy(iris_hashable *ht)
{
    int i;
    for (i = 0; i < ht->count; i++)
    {
        if (ht->bucket[i] != NULL)
        {
            iris_hashable_bucket_destroy(ht->bucket[i]);
        }
    }
    free(ht);
}

void iris_hashable_insert(iris_hashable *ht, char *lexeme, int data)
{
    int bucket;
    iris_hashable_bucket *htb;
    htb = (iris_hashable_bucket*)malloc(sizeof(iris_hashable_bucket));
    htb->lexeme = strdup(lexeme);
    htb->data = data;
    bucket = iris_hashable_hash(lexeme, ht->count);
    htb->next = ht->bucket[bucket];
    ht->bucket[bucket] = htb;
}```
**APPENDIX D. CODE**

```c
int iris_hashable_get(iris_hashable *ht, char *lexeme) {
    iris_hashable_bucket *b;
    b = iris_hashable_get_bucket(ht, lexeme);
    if (b != NULL) return b->data;
    return HASHTABLE_NOT_FOUND;
}

typedef struct _iris_hashtable {
    struct _iris_hashtable_bucket **bucket;
    int count;
} iris_hashable;

void iris_hashable_update(iris_hashable *ht, char *lexeme, int data) {
    iris_hashable_bucket *b;
    b = iris_hashable_get_bucket(ht, lexeme);
    if (b != NULL) b->data = data;
}

D.12 File: iris_hashtable.h

// Defines
#define HASHTABLE_NOT_FOUND -2

// Structures

typedef struct _iris_hashable_bucket {
    char *lexeme;
    int data;
    struct _iris_hashtable_bucket *next;
} iris_hashable_bucket;

D.13 File: iris_hde.c

#include "stdlib.h"
#include "string.h"
#include "math.h"
#include "stdio.h"
#include "time.h"
#include "iris_params.h"
#include "iris_heap.h"
#include "iris_function_data.h"
#include "iris_graph.h"
#include "iris_hde.h"

// Implementation of Eigenvector Projection Optimised in HDE Space
// See paper: Drawing Graphs by Eigenvectors: Theory and Practice (Y. Koren)

// matrix mult:
```
```c
void print_m(iris_hde_matrix *mat)
{
    int i, j;
    for (i = 0; i < mat->m; i++)
    {
        for (j = 0; j < mat->n; j++)
        {
            printf("%f\n", mat->v[i][j]);
        }
        printf("\n");
    }
}

void print_mc(float *X, int m)
{
    int i;
    for (i = 0; i < m; i++)
    {
        printf("%f\n", X[i]);
    }
}

void print_mci(int *X, int m)
{
    int i;
    for (i = 0; i < m; i++)
    {
        printf("%d\n", X[i]);
    }
}

void print_msz(iris_hde_matrix *m)
{
    printf("%d x %d\n", m->m, m->n);
}

// Private Functions

// Matrix Utility Functions ---

iris_hde_matrix *iris_hde_matrix_create(int m, int n)
{
    int i;
    iris_hde_matrix *matrix;
    matrix = (iris_hde_matrix*)malloc(sizeof(iris_hde_matrix));
    matrix->m = m;
    matrix->n = n;
    matrix->col = -1;
    matrix->v = (float**)malloc(sizeof(float*) * m);
    for (i = 0; i < m; i++)
    {
        matrix->v[i] = (float*)malloc(sizeof(float) * n);
    }
    return matrix;
}

void iris_hde_matrix_destroy(iris_hde_matrix *m)
{
    int i;
    for (i = 0; i < m->m; i++) free(m->v[i]);
    free(m->v);
    free(m);
}

// Dijkstra's Algorithm ---

float *iris_hde_dijkstra(iris_graph *graph, int s)
{
    int v, u;
    float *d;
    float w;
    int t;
    iris_heap *heap;
```
```
iris_heap_node **nodes;

heap = iris_heap_create();

nodes = (iris_heap_node **) malloc(sizeof(iris_heap_node) * graph->vertex_count);

d = (float *) malloc(sizeof(float) * graph->vertex_count);

for (v = 0; v < graph->vertex_count; v++)
    { 
d[v] = (v == s) ? 0 : INFINITY;
    nodes[v] = iris_heap_insert(heap, d[v], v);
    }

while(heap->nodes)
{ 
u = heap->extract_min(heap);
    for (v = 0; v < graph->vertex_count; v++)
    { 
w = 1;
t = graph->vertex[u].edges[v];
    if (d[u] + w < d[t])
    { 
d[t] = d[u] + w;
    iris_heap_decrease_key(heap, nodes[t], d[t]);
    }
    }
}
free(heap);
return d;

// High Dimension Embedding ———
```

```
int iris_hde_max(int *d, int len)
{
    int v;
    int u;
    u = 0;
    for (v = 0; v < len; v++)
    { 
        if (d[v] > d[u])
        { 
            u = v;
        }
    }
    return u;
}

iris_hde_matrix *iris_hde_embed(iris_graph *graph, int m)
{
    int pivot;
    int *d;
    float *dp;
    iris_hde_matrix *X;
    int i, j;
    d = (int *) malloc(sizeof(int) * graph->vertex_count);
    X = iris_hde_matrix_create(m,
                   graph->vertex_count);
    for (i = 0; i < graph->vertex_count; i++)
    { 
        d[i] = graph->vertex_count + 1;
    }
    pivot = rand() % graph->vertex_count;
    for (i = 0; i < m; i++)
    { 
        dp = iris_hde_dijkstra(graph, pivot);
        for (j = 0; j < graph->vertex_count; j++)
        { 
            X->v[i][j] = dp[j];
        }
}
```
d[j] = (int)MIN(d[j],
X->v[i][j]);

pivot = iris_hde_max(d,
graph->vertex_count);
free(dp);

pivot = iris_hde_max(d,
graph->vertex_count);
free(d);
return X;

// Mean shifting -------
float iris_hde_mean(iris_hde_matrix *X, int row)
{
  float tmp;
  int i;
  tmp = 0;
  for (i = 0; i < X->n; i++)
    tmp += X->v[row][i];
  return tmp / X->n;
}

void iris_hde_mean_shift(iris_hde_matrix *X)
{
  int i, j;
  float mi;

  for (i = 0; i < X->m; i++)
  {
    mi = iris_hde_mean(X, i);
    for (j = 0; j < X->n; j++)
      X->v[i][j] -= mi;
  }

  // Matrix Ops -------
iris_hde_matrix *iris_hde_matrix_mult(iris_hde_matrix *X1, iris_hde_matrix *X2)
{
  int i, j, k;
  iris_hde_matrix *R;
  R = iris_hde_matrix_create(X1->m, X2->n);
  for (i = 0; i < X1->m; i++)
  {
    for (j = 0; j < X2->n; j++)
      for (k = 0; k < X2->m; k++)
        R->v[i][j] += X1->v[i][k] * X2->v[k][j];
  }
  return R;
}

iris_hde_matrix *iris_hde_matrix_multv(iris_hde_matrix *X1, iris_hde_matrix *X2)
{
  int i, j, k;
  iris_hde_matrix *R;
  int cols;
  cols = (X2->col > -1) ? 1 : X2->n;
  R = iris_hde_matrix_create(X1->m, cols);
  for (i = 0; i < X1->m; i++)
  {
    for (j = 0; j < cols; j++)
    {
R->v[i][j] = 0;
for (k = 0; k < X2->m; k++)
{
    R->v[i][j] += X1->v[i][k] * X2->v[k][j];
}
return R;
}

void iris_hde_matrix_smult(float s, iris_hde_matrix *m)
{
    int i, j;
    for (i = 0; i < m->n; i++)
    {
        for (j = 0; j < m->n; j++)
        {
            m->v[i][j] *= s;
        }
    }
}

iris_hde_matrix
iris_hde_matrix_transpose(iris_hde_matrix *X)
{
    int i, j, ii;
    iris_hde_matrix *R;
    R = iris_hde_matrix_create(((X->col < 0) ? X->n : 1), X->m);
    for (i = 0, ii = ((X->col < 0) ? 0 : X->col); i < R->m; i++, ii++)
    {
        for (j = 0; j < R->n; j++)
        {
            R->v[i][j] = X->v[j][ii];
        }
    }
    return R;
}

float iris_hde_matrix_dot(iris_hde_matrix *X1, iris_hde_matrix *X2)
{
    int i, c;
    float dot = 0;
    c = (X2->col < 0) ? 0 : X2->col;
    for (i = 0; i < X1->m; i++)
    {
        dot += X1->v[i][0] * X2->v[i][c];
    }
    return sqrt(dot);
}

float iris_hde_matrix_dotv(iris_hde_matrix *X1, iris_hde_matrix *X2)
{
    int i;
    float dot = 0;
    for (i = 0; i < X1->m; i++)
    {
        dot += X1->v[0][i] * X2->v[0][i];
    }
    return sqrt(dot);
}

void iris_hde_matrix_normalise(iris_hde_matrix *X)
{
    float dot;
    int i, c;
    dot = iris_hde_matrix_dot(X, X);
    c = (X->col < 0) ? 0 : X->col;
for (i = 0; i < X->m; i++)
    { X->v[i][c] /= dot; }
}

void iris_hde_matrix_copy(iris_hde_matrix *S, iris_hde_matrix **T)
{
    int i, j, c;
    iris_hde_matrix *X;
    if (*T != NULL)
        { iris_hde_matrix_destroy(*T);
          *T = NULL;
        }
    X = iris_hde_matrix_create(S->m, (S->col < 0) ? S->n : 1);
    for (i = 0; i < X->m; i++)
        { for (j = 0; j < X->n; j++)
            { c = (S->col < 0) ? j : S->col;
              X->v[i][j] = S->v[i][c];
            }
          *T = X;
        }

    // Power Iteration (to find eigenvectors) ———

void iris_hde_eigenvector_set(iris_hde_matrix *eigenvectors, iris_hde_matrix *ev, int k)
{
    int i;
    for (i = 0; i < ev->m; i++)
        { eigenvectors->v[i][k] = ev->v[i][0];
        }
}

iris_hde_matrix *iris_hde_eigenvector_get(iris_hde_matrix *eigenvectors, int k)
{
    int i;
    iris_hde_matrix *ev;
    ev = iris_hde_matrix_create(eigenvectors->m, 1);
    for (i = 0; i < ev->m; i++)
        { ev->v[i][0] = eigenvectors->v[i][k];
        }
    return ev;
}

iris_hde_matrix *iris_hde_power_iteration(iris_hde_matrix *S, int k)
{
    int i, j, l;
    iris_hde_matrix *u;
    iris_hde_matrix *uj;
    iris_hde_matrix *uhat;
    iris_hde_matrix *eigenvectors;
    iris_hde_matrix *tmp;
    float dot;
    u = NULL;
    uhat = NULL;
    tmp = NULL;
    dot = 0;
    eigenvectors = iris_hde_matrix_create(S->m, k);
    for (i = 0; i < k; i++)
        { uhat = iris_hde_matrix_create(S->m, 1);
          for (j = 0; j < uhat->m; j++)
              { uhat->v[j][0] = rand();
              }
        }
}
iris_hde_matrix_normalize(uhat);

do {
    iris_hde_matrix_copy(uhat, &u);
    for (j = 0; j < i - 1; j++)
    {
        uj = iris_hde_eigenvector_get(evectors, i);  // Projection

        dot = iris_hde_matrix_dot(u, uj);

        for (l = 0; l < uj->m; l++)
        {
            u->v[l][0] = u->v[l][0] - (dot * uj->v[l][0]);
        }

        iris_hde_matrix_destroy(uj);
    }
    tmp = iris_hde_matrix_mult(S, u);
    iris_hde_matrix_copy(tmp, &uhat);  // iris_hde_matrix_destroy(uhat);
    iris_hde_matrix_destroy(tmp);
    iris_hde_matrix_normalise(uhat);  // iris_hde_matrix_destroy(uhat);
    dot = iris_hde_matrix_dot(uhat, u);
}

while (dot < 1.0f - HDE_POWER_IT_EPSILON);

iris_hde_eigenvector_set(evectors, uhat, i);
coords = iris_hde_matrix_mult_vector(Xt, ev);
for (j = 0; j < graph->vertex_count; j++)
{
    graph->vertex[j].trans[i] =
    coords->v[j][0] *
    HDE_SCALE_FACTOR;
}
iris_hde_matrix_destroy(ev);
iris_hde_matrix_destroy(coords);
}

// Computing Degree Matrix ---
iris_hde_matrix *iris_hde_compute_degree(iris_graph *
graph)
{
    int i;
    iris_hde_matrix *D;

    D = iris_hde_matrix_create(graph->vertex_count,
    1); // graph->vertex_count;
    for (i = 0; i < D->m; i++)
    {
        // D->v[i][i] = graph->vertex[i].edge_count;
        D->v[i][0] = graph->vertex[i].edge_count;
    }
    return D;
}

// Laplacian ---
iris_hde_matrix *iris_hde_laplacian(iris_graph *
graph, iris_hde_matrix *D)
{
    int i, j, col;
    iris_hde_matrix *L;

    L = iris_hde_matrix_create(D->m, D->n);
    for (i = 0; i < graph->sedges->row_count; i++)
    {
        bzero(L->v[i], sizeof(float) * L->n);
        L->v[i][i] = D->v[i][i];
        for (j = 0; j < graph->sedges->rows[i].els_count;
            j++)
        {
            col =
            graph->sedges->rows[i].els[j].col;
            if (col != i)
            L->v[i][col] -=
            graph->edges[i][col];
            if (col != i)
            L->v[i][col] =
            iris_graph_edge(graph, i,
            col); // graph->edges[i][col];
        }
    }
    return L;
}

// Spectral Drawing ---
iris_hde_matrix *iris_hde_matrix_mult_degree(iris_hde_matrix *D,
iris_hde_matrix *u)
{
    iris_hde_matrix *R;
    int i;

    R = iris_hde_matrix_create(D->m, 1);
    for (i = 0; i < R->m; i++)
    {
        // R->v[i][0] = D->v[i][i] * u->v[i][0];
        R->v[i][0] = D->v[i][0] * u->v[i][0];
    }
    return R;
}

float iris_hde_matrix_vector_mult(iris_hde_matrix *X,
iris_hde_matrix *Y)
```c
{  iris_hde_matrix *R, *T;
  float ret;
  if (Y->col > -1) {
    T = NULL;
    iris_hde_matrix_copy(Y, &T);
    R = iris_hde_matrix_mult(X, T);
    iris_hde_matrix_destroy(T);
  } else {
    R = iris_hde_matrix_mult(X, Y);
  }
  ret = R->v[0][0];
  iris_hde_matrix_destroy(R);
  return ret;
}

iris_hde_matrix *iris_hde_spectral_drawing(iris_graph *graph, iris_hde_matrix *D, int p)
{
  int k, i, l;
  float tmpl, tmp2, dot;
  eigs = iris_hde_matrix_create(D->m, p);
  u0 = iris_hde_matrix_create(D->m, 1);
  for (i = 0; i < u0->m; i++) u0->v[i][0] = 1;
  srand(time(NULL));
  for (k = 1; k < p + 1; k++) {
    uhat = iris_hde_matrix_create(D->m, 1);
    for (i = 0; i < uhat->m; i++)
      uhat->v[i][0] = ((float) rand() / ((float) RAND_MAX) + 1.0f);
    iris_hde_matrix_normalise(uhat);
    u = NULL;
    do {
      if (u != NULL)
        iris_hde_matrix_destroy(u);
      u =
        iris_hde_matrix_create(uhat->m, uhat->n);
      for (i = 0; i < uhat->m; i++)
        u->v[i][0] = uhat->v[i][0];
      for (l = 0; l < k - 1; l++)
        ut =
          iris_hde_matrix_transpose(u);
      ul = (l == 0) ? u0 :
        iris_hde_eigen_vector_get(eigs, l - 1);
      ul->col = (l == 0) ? 0 :
        l - 1;
      ult =
        iris_hde_matrix_transpose(ul);
      dul =
        iris_hde_matrix_multi_degree(D, ul);
      tmpl =
        iris_hde_matrix_vector_mult(ut, dul);
      tmp2 =
        iris_hde_matrix_vector_mult(ult, dul);
      tmpl = tmpl / tmp2;
      for (i = 0; i < u->m; i++)
        u->v[i][0] =
          tmpl->v[i][0] / tmp2->v[i][0];
    } while (u != NULL);
  }
  iris_hde_matrix_destroy(eigs);
  return eigs;
}
```
{  
  u[v[i][0]] = u[v[i][0]] -  
  (tmp1 *  
    ul[v[i][0]]);  
  }  
  
  // Orthonormalisation  
  // if (l != 0)  
  iris_hde_matrix *iris_hde_orthonormalise(iris_hde_matrix  
    ul);  
  iris_hde_matrix_destroy(ul);  
  
  for (i = 0; i < D; i++)  
  {  
    tmp1 = 0;  
    for (l = 0; l < graph->vertex[i].edge_count; l++)  
    {  
      tmp1 += u[v[graph->vertex[i].edges[l]][0]];  
    }  
    tmp1 /= graph->vertex[i].edge_count;  
    uhat[v[i][0]] = 0.5f *  
    (u[v[i][0]] + tmp1);  
  }  
  iris_hde_matrix_normalise(uhat);  
  dot = iris_hde_matrix_dot(uhat,  
    u);  
  }  
  while (dot < 1 - HDEORTH_EPSILON);  
  iris_hde_eigenvector_set(eigs, uhat, k - 1);  
  iris_hde_matrix_destroy(uhat);
tmp = uituj->v[0][0];
iris_hde_matrix_destroy(uituj);
for (l = 0; l < m-1; l++)
{
    ui->v[l][0] = ui->v[1][0] - (tmp * u->v[1][j]);
}
iris_hde_matrix_destroy(uj);
if (iris_hde_matrix_dot(ui, ui) < HDE_ORTH_EPSILON)
{
    zero[0] = 1;
} else {
    iris_hde_matrix_normalise(ui);
    iris_hde_eigenvector_set(u, ui, i);
    zero[i] = 0;
    nonzero++;
}
iris_hde_matrix_destroy(ui);
ret = iris_hde_matrix_create(m>m, nonzero);
for (i = 1, j = 0; i < u->n; i++)
{
    if (!zero[i])
    {
        uj = iris_hde_eigenvector_get(u, i);
        iris_hde_eigenvector_set(ret, uj, j);
        iris_hde_matrix_destroy(uj);
        j++;
    }
}
iris_hde_matrix_destroy(u);
return ret;

// Gershgorin bound (max eigenvalue calculation) ———
float iris_hde_gershgorin_bound(iris_hde_matrix *m)
{
    float val, tmp;
    int i, j;
    val = INFINITY;
    for (i = 0; i < m-1; i++)
    {
        tmp = m->v[i][i];
        for (j = 0; j < m-1; j++)
        {
            if (i != j) tmp += m->v[i][j];
        }
        if (tmp < val || val == INFINITY) val = tmp;
    }
    return val;
}

// Inverts order of eigenvectors for a matrix.
void iris_hde_matrix_invert_eigenvectors(iris_hde_matrix *m)
{
    float gb;
    int i, j;
    gb = iris_hde_gershgorin_bound(m);
    for (i = 0; i < m-1; i++)
    {
for (j = 0; j < m-n; j++)
    {
        m->v[i][j] = ((i == j) ? gb : 0) - m->v[i][j];
    }
}

// Compute XtLX ———
iris_hde_matrix *iris_hde_xtlx(iris_graph *graph,
    iris_hde_matrix *X, iris_hde_matrix *Xt,
    iris_hde_matrix *L)
{
    iris_hde_matrix *LX, *XtLX;
    LX = iris_hde_matrix_mult(L, X);
    XtLX = iris_hde_matrix_mult(Xt, LX);
    iris_hde_matrix_destroy(LX);
    return XtLX;
}

// Public Functions

typedef struct _iris_hde_matrix
{
    float **v;
    int m;
    int n;
    int col;
} iris_hde_matrix;

void iris_hde_process(iris_graph *graph, int dim)
{
    X = iris_hde_embed(graph, HDE_MDIM);
    iris_hde_mean_shift(X);
    if (iris_params_get() ->orth_mode)
    {
        Xo = iris_hde_orthonormalise(X);
    }
    else
    {
        Xo = NULL;
    }
    Xt = iris_hde_matrix_transpose((Xo == NULL) ? X : Xo);
    D = iris_hde_compute_degree(graph);
    u = iris_hde_spectral_drawing(graph, D, dim);
    iris_hde_project(graph, Xt, u, dim);
    iris_hde_matrix_destroy(D);
    iris_hde_matrix_destroy(X);
    iris_hde_matrix_destroy(Xt);
    iris_hde_matrix_destroy(u);
    if (iris_params_get() ->orth_mode)
    {
        iris_hde_matrix_destroy(Xo);
    }
}

D.14 File: iris_hde.h

#include "float.h"

// Structures

#endif

#ifndef MIN
#define MIN(x, y) ( (x < y) ? x : y )
#endif
```c
#include "iris_heap.h"

// Fibonacci Heap
// Based on the GNU CC implementation.
// http://gentoo.osuosl.org/distfiles/gcc-3.4.4.tar.gz/gcc-3.4.4/include/

// Forward Declarations
void iris_heap_cut(iris_heap *heap, iris_heap_node *node, iris_heap_node *parent);
void iris_heap_cascading_cut(iris_heap *heap, iris_heap_node *y);
iris_heap_node *iris_heap_remove(iris_heap_node *node);

// Private Functions
iris_heap_node *iris_heap_node_create();

void iris_hde_process(iris_graph *graph, int dim);

D.15 File: iris_heap.c

int iris_heap_compare(iris_heap_node *a, iris_heap_node *b)
{
    if (a->key < b->key) return -1;
    if (a->key > b->key) return 1;
    return 0;
}

int iris_heap_comp_data(float key, int data, iris_heap_node *b)
{
    iris_heap_node a;
    a.key = key;
    a.data = data;
    return iris_heap_compare(&a, b);
}

iris_heap *iris_heap_union(iris_heap *a, iris_heap *b)
{
    iris_heap_node *aroot, *broot, *temp;
    if ((aroot = a->root) == NULL)
    {
        free(a);
        return b;
    }
    if ((broot = b->root) == NULL)
    {
        free(b);
    }
    return a;
}
```

```c
#define HDE_MDIM 50
#define HDE_POWER_IT_EPSILON 0.001f
#define HDE_SCALE_FACTOR 1.0f
#define HDE_ORTH_EPSILON 0.001f
#define HDE_SPEC_DRAW_TOLERANCE 0.0000001f
```

```c
#ifndef INFINITY
#define INFINITY FLT_MAX
#endif
```

```c
#define HDE_NETWORK
#define HDE_MDIM 50
#define HDE_POWER_IT_EPSILON 0.001f
#define HDE_SCALE_FACTOR 1.0f
#define HDE_ORTH_EPSILON 0.001f
#define HDE_SPEC_DRAW_TOLERANCE 0.0000001f
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```c
void iris_hde_process(iris_graph *graph, int dim);
```
return a;
}

aroot->left->right = broot;
broot->left->right = aroot;
temp = aroot->left;
aroot->left = broot->left;
broot->left = temp;

a->nodes += b->nodes;

if (iris_heap_compare(b->min, a->min) < 0)
{
    a->min = b->min;
}

free(b);

return a;
}

int iris_heap_replace_key_data(iris_heap *heap, iris_heap_node *node, float key, int data)
{
    int odata;
    int okey;
    iris_heap_node *y;

    if (iris_heap_compare(key, data, node) > 0)
    {
        return -1;
    }

    odata = node->data;
    okey = node->key;
    node->data = data;
    node->key = key;
    y = node->parent;

    if (okey == key) return odata;

    if (y != NULL && iris_heap_compare(node, y) <= 0)
    {
        iris_heap_cut(heap, node, y);
        iris_heap_cascading_cut(heap, y);
    }

    if (iris_heap_compare(node, heap->min) <= 0)
    {
        heap->min = node;
    }

    return odata;
}

void iris_heap_insert_after(iris_heap_node *a, iris_heap_node *b)
{
    if (a == a->right)
    {
        a->right = b;
        a->left = b;
        b->right = a;
        b->left = a;
    }
    else
    {
        b->right = a->right;
        a->right->left = b;
        a->right = b;
        b->left = a;
    }
}

void iris_heap_ins_root(iris_heap *heap, iris_heap_node *node)
{
    if (heap->root == NULL)
    {
        heap->root = node;
        node->left = node;
        node->right = node;
        return;
    }

    iris_heap_insert_after(heap->root, node);
}

void iris_heap_rem_root(iris_heap *heap, iris_heap_node *node)
if (node→left == node)
{
    heap→root = NULL;
}
else
{
    heap→root = iris_heap_remove(node);
}

void iris_heap_link(iris_heap *heap, iris_heap_node *node, iris_heap_node *parent)
{
    if (parent→child == NULL)
    {
        parent→child = node;
    }
    else
    {
        iris_heap_insert_before(parent→child, node);
    }
    node→parent = parent;
    parent→degree++;
    node→mark = 0;
}

iris_heap_node *iris_heap_remove(iris_heap_node *node)
{
    iris_heap_node *ret;
    if (node == node→left)
    {
        ret = NULL;
    }
    else
    {
        ret = node→left;
    }
    if (node→parent != NULL && node→parent→child == node)
    {
        node→parent→child = ret;
    }
    node→right→left = node→left;
    node→left→right = node→right;
    node→parent = NULL;
    node→left = node;
    node→right = node;
    return ret;
}

void iris_heap_cut(iris_heap *heap, iris_heap_node *node, iris_heap_node *parent)
{
    iris_heap_remove(node);
    parent→degree--;
    iris_heap_ins_root(heap, node);
    node→parent = NULL;
    node→mark = 0;
}

void iris_heap_cascading_cut(iris_heap *heap, iris_heap_node *y)
{
    iris_heap_node *z;
    while ((z = y→parent) != NULL)
    {
        if (y→mark == 0)
        {
            y→mark = 1;
            return;
        }
        else
        {
            iris_heap_cut(heap, y, z);
            y = z;
        }
    }
}

void iris_heap_consolidate(iris_heap *heap)
{
iris_heap_node *a[1 + 8 * sizeof(long)];
iris_heap_node *w;
iris_heap_node *y;
iris_heap_node *x;
int i;
int d;
int D;

D = 1 + 8 * sizeof(long);
memset(a, 0, sizeof(iris_heap_node*) * D);

while ((w = heap->root) != NULL)
{
    x = w;
    iris_heap_rem_root(heap, w);
    d = x->degree;

    while (a[d] != NULL)
    {
        y = a[d];
        if (iris_heap_compare(x, y) > 0)
        {
            iris_heap_node *temp;
            temp = x;
            x = y;
            y = temp;
        }
        iris_heap_link(heap, y, x);
        a[d] = NULL;
        d++;
    }
    a[d] = x;
}
heap->min = NULL;

for (i = 0; i < D; i++)
{
    if (a[i] != NULL)
    {
        iris_heap_ins_root(heap, a[i]);
        if (heap->min == NULL || iris_heap_compare(a[i], heap->min) < 0)
        {
            heap->min = a[i];
        }
    }
}

iris_heap_node *iris_heap_extract_min_node(iris_heap *heap)
{
    iris_heap_node *ret, *x, *y, *orig;
    ret = heap->min;
    for (x = ret->child, orig = NULL; x != orig && x != NULL; x = y)
    {
        if (orig == NULL)
        {
            orig = x;
        }
        y = x->right;
        x->parent = NULL;
        iris_heap_ins_root(heap, x);
    }
    iris_heap_rem_root(heap, ret);
    heap->nodes--;
    if (heap->nodes == 0)
    {
        heap->min = NULL;
    }
    else
    {
        heap->min = ret->right;
        iris_heap_consolidate(heap);
APPENDIX D. CODE

// Public Functions

iris_heap *iris_heap_create()
{
    iris_heap *heap;
    heap = (iris_heap*)malloc(sizeof(iris_heap));
    bzero(heap, sizeof(iris_heap));
    return heap;
}

iris_heap_node *iris_heap_insert(iris_heap *heap, float key, int data)
{
    iris_heap_node *node;
    node = iris_heap_node_create();
    node->key = key;
    node->data = data;
    iris_heap_ins_root(heap, node);
    if (heap->min == NULL || node->key < heap->min->key)
    {
        heap->min = node;
    }
    heap->nodes++;
    return node;
}

int iris_heap_extract_min(iris_heap *heap)
{
    iris_heap_node *z;
    int val = -1;
    if (heap->min != NULL)
    {
        z = iris_heap_extract_min_node(heap);
        val = z->data;
        free(z);
    }
    return val;
}

void iris_heap_decrease_key(iris_heap *heap, iris_heap_node *node, float key)
{
    iris_heap_replace_key_data(heap, node, key, node->data);
}

D.16  File: iris_heap.h

// Defines

#define iris_heap_insert_before(a, b)  
 iris_heap_insert_after(a->left, b)

// Structures

typedef struct _iris_heap_node
{
    float key;
    int data;
    struct _iris_heap_node *parent;
    struct _iris_heap_node *child;
    struct _iris_heap_node *left;
};


```c
struct iris_heap_node *right;
int degree;
int mark;
}

typedef struct iris_heap
{
    int nodes;
    struct iris_heap_node *root;
    struct iris_heap_node *min;
} iris_heap;

// Functions

iris_heap *iris_heap_create();

iris_heap_node *iris_heap_insert(iris_heap *heap, float key, int data);

int iris_heap_extract_min(iris_heap *heap);

void iris_heap_decrease_key(iris_heap *heap, iris_heap_node *node, float key);

D.17 File: iris_list.c

#include <stdlib.h>
#include <string.h>
#include "iris_list.h"

// Public Functions

iris_list *iris_list_create(int len)
{
    int i;

    iris_list *lst;
    lst = (iris_list *)malloc(sizeof(iris_list));
    lst->vals = (int*)malloc(sizeof(int) * len);
    lst->len = len;
    for (i = 0; i < len; i++) lst->vals[i] = i;
    return lst;
}

void iris_list_remove(iris_list *lst, int index)
{
    int len;
    int *tmp;
    int i;

    if (index < lst->len - 1)
    {
        //memmove(lst + index, lst + index + 1,
        //        (lst->len - index) * sizeof(int));
        //dst = lst + index;
        src = dst;
        for (i = 0; i < lst->len - 1 - index; i++)
        {
            *dst++ = *(++src);
        }
        len = lst->len - index - 1;
        tmp = (int*)malloc(sizeof(int) * len);
        memcpy(tmp, lst->vals + index + 1, len * sizeof(int));
        memcpy(lst->vals + index, tmp, len * sizeof(int));
        free(tmp);
        //printf("dst: %d\tsrc: %d\tlen: %d\n",
        //        *dst, *src, len);
        //memcpy(dst, src, len);
    }
```
D.18 File: iris_list.h

// Structures

typedef struct iris_list
{
    int *vals;
    int len;
} iris_list;

// Functions

iris_list *iris_list_create(int len);
void iris_list_remove(iris_list *lst, int index);
void iris_list_destroy(iris_list *lst);

D.19 File: iris_max_queue.c

#include <stdlib.h>
#include <stdio.h>
#include "iris_max_queue.h"

int main(void)
{
    return 0;
}
// Public Functions

iris_max_queue *iris_max_queue_add(iris_max_queue *q, void *data, float priority)
{
    iris_max_queue *queue;

    queue = (iris_max_queue*)malloc(sizeof(iris_max_queue));
    queue->priority = priority;
    queue->data = data;
    queue->children = NULL;
    queue->next = NULL;

    return iris_max_queue_compare_link(q, queue);
}

iris_max_queue *iris_max_queue_remove_max(iris_max_queue *q)
{
    iris_max_queue *ret, *tmp2, *c, *tmp;

    ret = NULL;

    if (q->children != NULL)
    {
        if (q->children->next == NULL)
        {
            ret = q->children;
        }
        else
        {
            c = q->children;

            while (c->next != NULL)
            {
                tmp = c->next->next;
                tmp2 = c;
                c =
                iris_max_queue_compare_link(c, c->next);
                c->next = tmp;
                tmp2->next = NULL;

                ret = c;
            }
        }
    }

    free(q);
    return ret;
}

iris_max_queue *iris_max_queue_merge(iris_max_queue *q1, iris_max_queue *q2)
{
    return iris_max_queue_compare_link(q1, q2);
}

void iris_max_queue_destroy(iris_max_queue *q)
{
}

D.20 File: iris_max_queue.h

// Structures

typedef struct _iris_max_queue
{
    float priority;
    void *data;
    struct _iris_max_queue *children;
    struct _iris_max_queue *next;
} iris_max_queue;

// Functions
iris_max_queue *iris_max_queue_add(iris_max_queue *q,  
    void *data, float priority);

iris_max_queue *iris_max_queue_remove_max(iris_max_queue  
    *q);

iris_max_queue *iris_max_queue_merge(iris_max_queue *q1,  
    iris_max_queue *q2);

void iris_max_queue_destroy(iris_max_queue *q);

## D.21 File: `iris_params.c`

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "iris_params.h"

// Private Variables

iris_params *params;

// Private Functions

int iris_params_is_switch(char *str) {  
    return (str[0] == '-');
}

void iris_params_help() {  
    printf("\nUsage.: iris [options] [source files] ...\n");
    printf("Options:\n");
    printf("-w Show visualisation in window.\n");
    printf("-d Disable group colouring.\n");
    printf("-v Show verbose output.\n");
    printf("-s Show summary output.\n");
    printf("-o Apply orthonormalisation to graph.\n");
}

void iris_params_init(int argc, char **argv) {  
    int i, tmp, invalid_switch;
    float tmpf;
    invalid_switch = 0;
```
params = (iris_params*) malloc(sizeof(iris_params));

params->window_mode = 0;
params->colour_mode = 1;
params->verbose_mode = 0;
params->summary_mode = 1;
params->orth_mode = 0;
params->dims = 3;
params->refine = 0.5f;
params->check_graph = 1;
params->visualise = 1;
params->files = NULL;
params->file_count = 0;
params->func_count = INITIAL_GRAPH_SIZE;
params->perf = 0;
params->func_max_count = 50;
params->layout = GRAPHAUTO;
params->debug = 0;
params->editor = strdup("vi\%d\%s");

for (i = 1; i < argc; i++)
{
    if (strcmp(argv[i], "-w") == 0)
    {
        params->window_mode = 1;
    }
    else if (strcmp(argv[i], "-c") == 0)
    {
        params->colour_mode = 0;
    }
    else if (strcmp(argv[i], "-v") == 0)
    {
        params->verbose_mode = 1;
    }
    else if (strcmp(argv[i], "-s") == 0)
    {
        params->summary_mode = 0;
    }
    else if (strcmp(argv[i], "-o") == 0)
    {
        params->orth_mode = 1;
    }
    else if (strcmp(argv[i], "-d") == 0)
    {
        if (i + 1 < argc) tmp = atoi(argv[i + 1]);
        if (tmp > 0 & tmp < 4)
        {
            params->dims = tmp;
            i++;
        }
    }
    else if (strcmp(argv[i], "-r") == 0)
    {
        if (i + 1 < argc)
        {
            if (sscanf(argv[i + 1], "%f", &tmpf))
            {
                params->refine = tmpf;
                i++;
            }
        }
    }
    else if (strcmp(argv[i], "-nocheck") == 0)
    {
        params->check_graph = 0;
    }
    else if (strcmp(argv[i], "-novis") == 0)
    {
        params->visualise = 0;
    }
    else if (strcmp(argv[i], "-perf") == 0)
    {
        params->perf = 1;
    }
    else if (strcmp(argv[i], "--help") == 0 || strcmp(argv[i], "-?") == 0)
    {
        iris_params_help();
    }
    else if (strcmp(argv[i], "-f") == 0)
    {
        if (i + 1 < argc) tmp = atoi(argv[i + 1]);
        if (tmp > 0)
APPENDIX D. CODE

```c
params->func_count = tmp;
i++;
}
else if (strcmp(argv[i], "-m") == 0)
{
    if (i + 1 < argc) tmp = atoi(argv[i + 1]);
    if (tmp > 0)
    {
        params->func_max_count = tmp;
i++;
    }
}
else if (strcmp(argv[i], "-a") == 0)
{
    if (i + 1 < argc &&
        strcmp("spring", argv[i + 1]) == 0)
    {
        params->layout = SPRING;
i++;
    }
    else if (i + 1 < argc &&
        strcmp("hde", argv[i + 1]) == 0)
    {
        params->layout = HDE;
i++;
    }
}
else if (strcmp(argv[i], "--debug") == 0)
{
    params->debug = 1;
}
else
{
    if (iris_params_is_switch(argv[i]))
    {
        printf("Invalid switch: \\
               \%s", argv[i]);
        invalid_switch = 1;
    }
    else
    {
        iris_params_add_file(argv[i]);
    }
    if (invalid_switch)
    {
        printf("\n\nTry --help for \
help.\n\n" for \
help.\n\nfaction.
```
int summary_mode;
int orth_mode;
int dims;
float refine;
int check_graph;
int visualise;
char **files;
int file_count;
int func_count;
int perf;
int func_max_count;
enum {HDE, SPRING, GRPHAUTO} layout;
int debug;
char *editor;
} iris_params;

// Functions

void iris_params_init(int argc, char **argv);
iris_params *iris_params_get();
void iris_params_destroy();

D.23 File: iris_queue.c

#include <stdlib.h>
#include <string.h>

#include "iris_function_data.h"
#include "iris_graph.h"
#include "iris_heap.h"
#include "iris_max_queue.h"
#include "iris_refine.h"
#include "iris_queue.h"

// Public Functions

iris_queue *iris_queue_create(int size)
{
    iris_queue *queue;
    queue = (iris_queue*)malloc(sizeof(iris_queue));
    queue->count = 0;
    queue->size = size;
    queue->items = (int*)malloc(sizeof(int) * size);
    queue->data = (iris_refine_event**)malloc(sizeof(iris_refine_event*) * size);
    return queue;
}

void iris_queue_destroy(iris_queue *queue)
{
    if (queue->count > 0)
    {
        free(queue->items);
        free(queue->data);
    }
    free(queue);
}

void iris_queue_add(iris_queue *queue, int val)
{
    queue->count++;
    if (queue->size < queue->count)
    {
        queue->items = (int*)realloc(queue->items, 

typedef struct _iris_queue
{
    int *items;
    iris_refine_event **data;
    int count;
    int size;
} iris_queue;

// Functions

iris_queue *iris_queue_create(int size);
void iris_queue_destroy(iris_queue *queue);
void iris_queue_add(iris_queue *queue, int val);
void iris_queue_add_data(iris_queue *queue, iris_refine_event *data);
void iris_queue_remove(iris_queue *queue, int val);

D.24 File: iris_queue.h

D.25 File: iris_rbtree.c
void iris_rbtree_insert1(iris_rbtree_tree* tree, iris_rbtree_node* node);
void iris_rbtree_remove1(iris_rbtree_tree* tree, iris_rbtree_node* node);

// Private Functions

iris_rbtree_node* iris_rbtree_create_node() {
    iris_rbtree_node* node;
    node = (iris_rbtree_node*) malloc(sizeof(iris_rbtree_node));
    node->parent = NULL;
    node->left = NULL;
    node->right = NULL;
    node->count = 0;
    node->ptrs = NULL;
    node->ptr = NULL;
    return node;
}

iris_rbtree_node* iris_rbtree_insert_internal(iris_rbtree_tree* tree, iris_rbtree_node* node, float value, int data, void* ptr) {
    if (node == NULL) {
        tree->root = iris_rbtree_create_node();
        tree->root->value = value;
        tree->root->colour = RED;
        tree->root->data = data;
        tree->root->ptr = ptr;
        return tree->root;
    } else if (value == node->value && tree->singleton) {
        return NULL;
    } else if (value == node->value) {
        node->count++;
        if (ptr != NULL) {
            node->ptrs = (void**) realloc(node->ptrs, sizeof(void*) * node->count);
            node->ptrs[node->count - 1] = ptr;
        }
        return node;
    } else if (value < node->value && node->left == NULL) {
        node->left = iris_rbtree_create_node();
        node->left->value = value;
        node->left->parent = node;
        node->left->colour = RED;
        node->left->data = data;
        node->left->ptr = ptr;
        return node->left;
    } else if (value < node->value) {
        return iris_rbtree_insert_internal(tree, node->left, value, data, ptr);
    } else if (value > node->value && node->right == NULL) {
        node->right = iris_rbtree_create_node();
        node->right->value = value;
        node->right->parent = node;
        node->right->colour = RED;
        node->right->data = data;
        node->right->ptr = ptr;
        return node->right;
    } else {
        return iris_rbtree_insert_internal(tree, node->right, value, data, ptr);
    }
}
```c
iris_rbtree_node *iris_rbtree_grandparent(iris_rbtree_node *node)
{
    return node->parent->parent;
}

iris_rbtree_node *iris_rbtree_uncle(iris_rbtree_node *node)
{
    if (node->parent == iris_rbtree_grandparent(node)->left)
    {
        return iris_rbtree_grandparent(node)->right;
    }
    else
    {
        return iris_rbtree_grandparent(node)->left;
    }
}

iris_rbtree_node *iris_rbtree_sibling(iris_rbtree_node *node)
{
    if (node == node->parent->left)
    {
        return node->parent->right;
    }
    else
    {
        return node->parent->left;
    }
}

void iris_rbtree_rotate_right(iris_rbtree_tree *tree, iris_rbtree_node *node)
{
    iris_rbtree_node *X, *Y;
    X = node;
    Y = X->left;
    X->left = Y->right;
    if (Y->right != NULL)
    {
        Y->right->parent = X;
    }
    Y->parent = X->parent;
    if (X->parent == NULL)
    {
        tree->root = Y;
    }
    else if (X == X->parent->left)
    {
        X->parent->left = Y;
    }
    else
    {
        X->parent->right = Y;
    }
    Y->right = X;
    X->parent = Y;
}

void iris_rbtree_rotate_left(iris_rbtree_tree *tree, iris_rbtree_node *node)
{
    iris_rbtree_node *X, *Y;
    X = node;
    Y = X->right;
    X->right = Y->left;
    if (Y->left != NULL)
    {
        Y->left->parent = X;
    }
    Y->parent = X->parent;
    if (X->parent == NULL)
    {
        tree->root = Y;
    }
    else if (X == X->parent->left)
    {
        X->parent->left = Y;
    }
    else
    {
        X->parent->right = Y;
    }
    Y->left = X;
    X->parent = Y;
}
```

---

**APPENDIX D. CODE**

108
else if (X == X->parent->left)
    {
        X->parent->left = Y;
    }
else
    {
        X->parent->right = Y;
    }
Y->left = X;
X->parent = Y;
}

void __irbtree_insert5(__irbtree_tree* tree, __irbtree_node *node)
{
    node->parent->colour = BLACK;
    __irbtree_grandparent(node)->colour = RED;
    if (node == node->parent->left && node->parent ==
        __irbtree_grandparent(node)->left)
        {
            __irbtree_rotate_right(tree, 
                __irbtree_grandparent(node));
    }
else
    {
        __irbtree_rotate_left(tree, 
            __irbtree_grandparent(node));
    }
}

void __irbtree_insert4(__irbtree_tree* tree, __irbtree_node *node)
{
    if (node == node->parent->right && node->parent ==
        __irbtree_grandparent(node)->left)
        {
            __irbtree_rotate_left(tree, 
                node->parent); 
        node = node->left;
    }
else if (node == node->parent->left &&
            node->parent ==
    __irbtree_grandparent(node)->right)
        {
            __irbtree_rotate_right(tree, 
                node->parent); 
        node = node->right;
    }
__irbtree_insert5(tree, node);

void __irbtree_insert3(__irbtree_tree* tree, 
                        __irbtree_node *node)
{
    if (!(__irbtree_unsell(node) != NULL &&
        __irbtree_unsell(node)->colour == RED))
        {
            node->parent->colour = BLACK;
            __irbtree_unsell(node)->colour = BLACK;
            __irbtree_grandparent(node)->colour =
                RED;
            __irbtree_insert1(tree,
                __irbtree_grandparent(node));
        }
else
    {
        __irbtree_insert4(tree, node);
    }
}

void __irbtree_insert2(__irbtree_tree* tree, 
                        __irbtree_node *node)
{
    if (node->parent->colour == BLACK)
        {
            return;
        }
else
    {
        __irbtree_insert3(tree, node);
    }
}

void __irbtree_insert1(__irbtree_tree* tree, 
                        __irbtree_node *node)
{
if (node->parent == NULL)
{
    node->colour = BLACK;
}
else
{
    iris_rbtree_insert2(tree, node);
}
}

void iris_rbtree_remove6(iris_rbtree_tree* tree, iris_rbtree_node* node)
{
    iris_rbtree_sibling(node)->colour =
    node->parent->colour;
    node->parent->colour = BLACK;
    if (node == node->parent->left)
    {
        iris_rbtree_sibling(node)->right->colour =
        BLACK;
        iris_rbtree_rotate_right(tree, node->parent);
    }
    else
    {
        iris_rbtree_sibling(node)->left->colour =
        BLACK;
        iris_rbtree_rotate_left(tree, node->parent);
    }
}

void iris_rbtree_remove5(iris_rbtree_tree* tree, iris_rbtree_node* node)
{
    if (node == node->parent->left &&
        iris_rbtree_sibling(node)->colour ==
        BLACK &&
        iris_rbtree_sibling(node)->left->colour ==
        RED &&
        iris_rbtree_sibling(node)->right->colour ==
        BLACK)
    {
        iris_rbtree_remove6(tree, node);
    }
}

void iris_rbtree_remove4(iris_rbtree_tree* tree, iris_rbtree_node* node)
{
    if (node->parent->colour == RED &&
        iris_rbtree_sibling(node)->colour ==
        BLACK &&
        iris_rbtree_sibling(node)->left->colour ==
        BLACK &&
        iris_rbtree_sibling(node)->right->colour ==
        BLACK)
    {
        iris_rbtree_remove5(tree, node);
    }
    else
    {
        iris_rbtree_remove5(tree, node);
    }
}
void iris_rbtree_remove3(iris_rbtree_tree* tree, iris_rbtree_node* node) {
    if (node->parent->colour == BLACK &&
        iris_rbtree_sibling(node)->colour == BLACK &&
        iris_rbtree_sibling(node)->left->colour == BLACK &&
        iris_rbtree_sibling(node)->right->colour == BLACK) {
        iris_rbtree_sibling(node)->colour = RED;
        iris_rbtree_remove1(tree, node->parent);
    } else {
        iris_rbtree_remove4(tree, node);
    }
}

void iris_rbtree_remove2(iris_rbtree_tree* tree, iris_rbtree_node* node) {
    if (iris_rbtree_sibling(node)->colour == RED) {
        node->parent->colour = RED;
        iris_rbtree_remove1(tree, node->parent);
    } else if (prev == node->parent->left) {
        if (node == node->parent->left) {
            iris_rbtree_rotate_left(tree, node->parent);
        } else {
            iris_rbtree_rotate_right(tree, node->parent);
        }
    } else if (prev == node->parent->right) {
        if (node == node->parent->right) {
            iris_rbtree_rotate_right(tree, node->parent);
        } else if (prev == node->parent->right) {
            if (current->value == value) return current;
            next = (current->right != NULL) ?
                current->right :
                current->parent;
            if (next == Nnode->count++;ULL) {
                if (current->value == value) return current;
                next = (current->right != NULL) ?
                    current->right :
                    current->parent;
            }
        } else if (prev == node->parent->left) {
            if (current->value == value) return current;
        }
    }
    iris_rbtree_remove3(tree, node);
}

void iris_rbtree_remove1(iris_rbtree_tree* tree, iris_rbtree_node* node) {
    if (node->parent == NULL) {
        return;
    } else {
        iris_rbtree_remove2(tree, node);
    }
}

iris_rbtree_node* iris_rbtree_search_internal(float value, iris_rbtree_node* node) {
    /*iris_rbtree_node* prev;   iris_rbtree_node* current;   iris_rbtree_node* next;*/
    prev = NULL;
    current = node;
    while (current != NULL) {
        if (prev == current->parent) {
            next = current->left;
            if (next == Nnode->count++;ULL) {
                if (current->value == value) return current;
            } else if (prev == current->left) {
                if (current->value == value) return current;
            }
        } else if (prev == current->right) {
            if (current->value == value) return current;
            next = (current->right != NULL) ?
                current->right :
                current->parent;
            if (next == Nnode->count++;ULL) {
                if (current->value == value) return current;
                next = (current->right != NULL) ?
                    current->right :
                    current->parent;
            }
        }
    }
    return NULL;
}


```c
next = (current->right != NULL) ?
    current->right :
    current->parent;
}
else {
    next = current->parent;
}
prev = current;
current = next;
}
```

```c
return NULL; /*
if (node == NULL)
{
    return NULL;
}
else if (node->value == value)
{
    return node;
}
else if (value < node->value)
{
    return iris_rbtree_search_internal(value,
        node->left);
}
else
{
    return iris_rbtree_search_internal(value,
        node->right);
}
}
```

```c
void iris_rbtree_destroy_node(iris_rbtree_node *node)
{
    if (node->left != NULL)
        iris_rbtree_destroy_node(node->left);
    if (node->right != NULL)
        iris_rbtree_destroy_node(node->right);
    free(node);
}
```

```c
iris_rbtree_tree *iris_rbtree_create()
{
    iris_rbtree_tree *tree;
    tree = (iris_rbtree_tree*)malloc(sizeof(iris_rbtree_tree));
    tree->root = NULL;
    tree->node_count = 0;
    tree->singleton = 0;
    return tree;
}
```

```c
iris_rbtree_tree *iris_rbtree_create_singleton()
{
    iris_rbtree_tree *tree;
    tree = iris_rbtree_create();
    tree->singleton = 1;
    return tree;
}
```

```c
void iris_rbtree_destroy(iris_rbtree_tree *tree)
{
    iris_rbtree_destroy_node(tree->root);
    free(tree);
}
```

```c
iris_rbtree_node *iris_rbtree_insert_count(iris_rbtree_tree *tree,
    float value, int data, void *ptr)
{
    iris_rbtree_node *node;
    node = iris_rbtree_insert_internal(tree,
        tree->root, value, data, ptr);
    if (node != NULL)
      return node;
    return NULL;
}
```

```c
iris_rbtree_node *iris_rbtree_balance(iris_rbtree_node *node)
{
    if (node == NULL)
      return NULL;
    else if (node->left == NULL)
      return node;
    else if (node->right == NULL)
      return node;
    else
    {
      iris_rbtree_node *left = node->left;
      iris_rbtree_node *right = node->right;
      if (left->value < right->value)
        return iris_rbtree_balance(left);
      else
        return iris_rbtree_balance(right);
    }
}
```

```c
void iris_rbtree_balance_internal(iris_rbtree_node *node,
    int balance)
{
    if (balance != 0)
    {
      if (balance < 0)
        node = iris_rbtree_rotate_right(node);
      else
        node = iris_rbtree_rotate_left(node);
    }
    if (node->left != NULL)
      iris_rbtree_balance_internal(node->left,
          node->left->balance);
    if (node->right != NULL)
      iris_rbtree_balance_internal(node->right,
          node->right->balance);
}
```
void iris_rbtree_remove(iris_rbtree_tree *tree, iris_rbtree_node *node)
{
    iris_rbtree_node *child;
    child = (node->right == NULL) ? node->left : node->right;
    if (child != NULL) child->parent = node->parent;
    if (node->parent != NULL && node->parent->right == node)
    {
        node->parent->right = child;
    } else if (node->parent != NULL)
    {
        node->parent->left = child;
    } else if (node->parent == NULL && node->left == NULL && node->right == NULL)
    {
        tree->root = NULL;
    }
    if (node->colour == BLACK)
    {
        if (child != NULL && child->colour == RED)
        {
            child->colour = BLACK;
        } else if (child != NULL)
        {
            iris_rbtree_remove1(tree, child);
        }
    }
    free(node);
    tree->node_count--;
}

int iris_rbtree_next_left(iris_rbtree_tree *tree, float value)
{
    iris_rbtree_node *node =
    iris_rbtree_search_internal(value, tree->root);
    if (node != NULL)
    {
        iris_rbtree_remove(tree, node);
    }
}
node = iris_rbtree_search_internal(value, tree->root);

if (node->left != NULL)
{
    node = node->left;
    while (node->right != NULL) node = node->right;
    return node->data;
}
else if (node->parent != NULL && node == node->parent->right)
{
    return node->parent->data;
}
{
    return node->parent->parent->data;
}
else return -1;

void iris_rbtree_traverse(iris_rbtree_node *node)
{
    if (node->left != NULL)
    iris_rbtree_traverse(node->left);
    printf("%d\n", node->data);
    if (node->right != NULL)
    iris_rbtree_traverse(node->right);
}

iris_rbtree_node *iris_rbtree_search(iris_rbtree_tree *tree, float value)
{
    return iris_rbtree_search_internal(value, tree->root);
}

D.26 File: iris_rbtree.h

// Defines

// Structures

typedef struct iris_rbtree_node
{
    float value;
    int data;
    enum {BLACK, RED} colour;
    int count;
    void **ptrs;
}
void *ptr;
struct _iris_rbtree_node *parent;
struct _iris_rbtree_node *left;
struct _iris_rbtree_node *right;
} iris_rbtree_node;

typedef struct _iris_rbtree_tree
{
    iris_rbtree_node *root;
    int node_count;
    int singleton;
} iris_rbtree_tree;

// Functions
______________
iris_rbtree_tree *iris_rbtree_create();
iris_rbtree_tree *iris_rbtree_create_singleton();
void iris_rbtree_destroy(iris_rbtree_tree *tree);
iris_rbtree_node *iris_rbtree_insert(iris_rbtree_tree *tree, float value, int data);
iris_rbtree_node
   *iris_rbtree_insert_count(iris_rbtree_tree *tree, float value, int data, void *ptr);
void iris_rbtree_remove(iris_rbtree_tree *tree, iris_rbtree_node *node);
void iris_rbtree_remove_value(iris_rbtree_tree *tree, float value);
int iris_rbtree_next_left(iris_rbtree_tree *tree, float value);

int iris_rbtree_next_right(iris_rbtree_tree *tree, float value);
iris_rbtree_node *iris_rbtree_search(iris_rbtree_tree *tree, float value);

D.27 File: iris_refine.c
______________
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <stdio.h>
#include "iris.h"
#include "iris_params.h"
#include "iris_heap.h"
#include "iris_function_data.h"
#include "iris_graph.h"
#include "iris_visualiser.h"
#include "iris_rbtree.h"
#include "iris_max_queue.h"
#include "iris_refine.h"
#include "iris_queue.h"

#include "iris_max_queue.h"
#include "iris_refine.h"
#include "iris_queue.h"

#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <stdio.h>
#include "iris.h"
#include "iris_params.h"
#include "iris_heap.h"
#include "iris_function_data.h"
#include "iris_graph.h"
#include "iris_visualiser.h"
#include "iris_rbtree.h"
#include "iris_max_queue.h"
#include "iris_refine.h"
#include "iris_queue.h"

#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <stdio.h>
#include "iris.h"
#include "iris_params.h"
#include "iris_heap.h"
#include "iris_function_data.h"
#include "iris_graph.h"
#include "iris_visualiser.h"
#include "iris_rbtree.h"
#include "iris_max_queue.h"
#include "iris_refine.h"
#include "iris_queue.h"

// Implementation of algorithm found in paper:
// "Fast Node Overlap Removal in Graph Layout Adjustment"
// Private Variables
_____
iris_refine_block **block, **blockptr;
float *offset;
int timeCtr;
int nodeCount;

#define ABS( x ) ( ( x > 0) ? x : (−1 * x) )

// Private Functions
_____

// Calculate overlaps ———
float iris_refine_olap_x(iris_graph* graph, int u, int v)
{
    return (REFINE_MIN_GAP -
        ABS(graph->vertex[u].trans[0] -
            graph->vertex[v].trans[0]));
}

float iris_refine_olap_y(iris_graph* graph, int u, int v)
{
    return (REFINE_MIN_GAP -
        ABS(graph->vertex[u].trans[1] -
            graph->vertex[v].trans[1]));
}

float iris_refine_olap_z(iris_graph* graph, int u, int v)
{
    return (REFINE_MIN_GAP -
        ABS(graph->vertex[u].trans[2] -
            graph->vertex[v].trans[2]));
}

// Get all neighbours to the left/right with overlap
_________
iris_queue *iris_refine_get_left_nbours(iris_graph* graph, iris_rbtree_tree *scanline, int v, int dim)
{
    int u;
    iris_queue *queue;
    queue = iris_queue_create(1);
    if (scanline == NULL) return queue;
    u = iris_rbtree_next_left(scanline, graph->vertex[v].trans[dim]);
    while (u != -1)
    {
        if (dim == 0)
        {
            if (iris_refine_olap_x(graph, u, v) <= 0)
            {
                iris_queue_add(queue, u);
            }
        }
        else if (dim == 1)
        {
            if (iris_refine_olap_y(graph, u, v) <= 0)
            {
                iris_queue_add(queue, u);
                return queue;
            }
            else if (iris_refine_olap_z(graph, u, v) <= 0)
            {
                iris_queue_add(queue, u);
            }
        }
        else
        {
            if (iris_refine_olap_z(graph, u, v) <= 0)
            {
                return queue;
            }
        }
        u = iris_rbtree_next_left(scanline, graph->vertex[u].trans[dim]);
    }
    return queue;
}
iris_queue *iris_refine_get_right_neighbours(iris_graph *graph, iris_rbtree_tree *scanline, int v, int dim)
{
    int u;
    iris_queue *queue;
    u = iris_rbtree_next_right(scanline, graph->vertex[v].trans[dim]);
    queue = iris_queue_create(1);

    while (u != -1)
    {
        if (dim == 0)
        {
            if (iris_refine_olap_x(graph, u, v) <= 0)
            {
                iris_queue_add(queue, u);
            }
            return queue;
        }

        if (iris_refine_olap_x(graph, u, v) <=
            iris_refine_olap_y(graph, u, v))
        {
            iris_queue_add(queue, u);
        }
        else if (dim == 1)
        {
            if (iris_refine_olap_y(graph, u, v) <= 0)
            {
                iris_queue_add(queue, u);
            }
            return queue;
        }

        if (iris_refine_olap_y(graph, u, v) <=
            iris_refine_olap_z(graph, u, v))
        {
            iris_queue_add(queue, u);
        }
        else
        {
            iris_queue_add(queue, u);
        }

        return queue;
    }

    // Sort event queue ———
    int iris_refine_event_partition(iris_queue *events, int left, int right, int pivot)
    {
        float pivotVal;
        iris_refine_event *tmp;
        int storeIndex, i;

        pivotVal = events->data[pivot]->posn;
        tmp = events->data[pivot];
        events->data[pivot] = events->data[right];
        events->data[right] = tmp;

        storeIndex = left;

        for (i = left; i < right; i++)
        {
            if (events->data[i]->posn <= pivotVal)
            {
                tmp = events->data[storeIndex];
                events->data[storeIndex] = events->data[i];
                storeIndex = i;
            }
        }
    }
}
APPENDIX D. CODE

```c
APPENDIX D. CODE

// Code for handling events and data manipulation

// Handle data events
void handle_data_events(events *data, storeIndex)
{
    // Process data events
    for (i = 0; i < storeIndex; i++)
    {
        // Store data
        storeIndex++;
    }
    // Clear storeIndex
    tmp = events->data[storeIndex];
    events->data[storeIndex] = events->data[right];
    events->data[right] = tmp;
}

// Refine events
void refine_events_events(event, int left, int right)
{
    int pivot;
    if (left < right)
    {
        pivot = left + ((right - left) / 2);
        pivot = refine_event_partition(event, left, right, pivot);
        refine_event_qsort(event, left, pivot - 1);
        refine_event_qsort(event, pivot + 1, right);
    }
}

// Sort events
void refine_event_sort(event)
{
    refine_event_qsort(event, 0, event->count - 1);
}

// Create event queues
iris_queue *iris_refine_event_queue(iris_graph *graph, int type)
{
    int i;
    iris_queue *queue = NULL;
    iris_refine_event *event;
    // Iterate over vertices
    for (i = 0; i < graph->vertex_count; i++)
    {
        // Allocate event
        event = (iris_refine_event *)malloc(sizeof(iris_refine_event));
        event->vertex = i;
        event->type = OPEN;
        event->posn = graph->vertex[i].trans[type] - VISUALISER_SPHERE_SIZE;
        iris_queue_add_data(queue, event);
    }
    // Sort events
    refine_event_sort(queue);
    return queue;
}

// Create overlap constraints
iris_refine_constraint *
refine_generate_contraints(iris_graph *graph, int dim)
{
    // Allocate memory
    iris_queue *events, *leftv, *rightv, **left, **right;
    iris_rbtree_tree *scanline;
    iris_refine_event *event;
    int v, u, i, j, vtx;
```
iris_refine_constraint *constraints, *ctail, *ctmp;
iris_rbtree_node *node;
int fail;

constraints = ctail = NULL;

events = iris_refine_event_queue(graph, dim);
scanline = iris_rbtree_create();

left = (iris_queue**) malloc(sizeof(iris_queue*) * graph->vertex_count);
right = (iris_queue**) malloc(sizeof(iris_queue*) * graph->vertex_count);

bzero(left, sizeof(iris_queue*) * graph->vertex_count);
bzero(right, sizeof(iris_queue*) * graph->vertex_count);

fail = 0;

for (vtx = 0; vtx < events->count && fail == 0; vtx++)
{
    event = events->data[vtx];
    v = event->vertex;
    if (event->type == OPEN)
    {
        node =
            iris_rbtree_insert(scanline, graph->vertex[v].trans[dim], event->vertex);
        if (node->count > 0)
        {
            fail = 1;
            break;
        }
    }
    leftv =
        iris_refine_get_left_nbours(graph, scanline, v, dim);
    rightv =
        iris_refine_get_right_nbours(graph, scanline, v, dim);
    left[v] = leftv;
    for (i = 0; i < leftv->count; i++)
    {
        u = leftv->items[i];
        if (right[u] == NULL)
        {
            right[u] = iris_queue_create(rightv->count);
        }
        iris_queue_add(right[u], v);
        for (j = 0; j < rightv->count; j++)
        {
            iris_queue_remove(right[u], rightv->items[j]);
        }
    }
    right[v] = rightv;
    for (i = 0; i < rightv->count; i++)
    {
        u = rightv->items[i];
        if (left[u] == NULL)
        {
            left[u] = iris_queue_create(leftv->count);
        }
        iris_queue_add(left[u], v);
    }
}
for (j = 0; j < leftv->count; j++)
{
    iris_queue_remove(left[u],
    leftv->items[j]);
}
}
else
{
    for (i = 0; i < left[v]->count; i++)
    {
        u = left[v]->items[i];
        ctmp =
            (iris_refine_constraint*)malloc(sizeof(iris_refine_constraint));
        ctmp->left = u;
        ctmp->right = v;
        ctmp->gap = REFINE_MIN_GAP;
        ctmp->next = NULL;
        if (constraints == NULL)
        {
            constraints =
            ctail = ctmp;
        }
        else
        {
            ctail->next =
            ctmp;
            ctail = ctmp;
        }
    }
    iris_queue_remove(right[u],
    v);
}
for (i = 0; i < right[v]->count; i++)
{
    u = right[v]->items[i];
    ctmp =
        (iris_refine_constraint*)malloc(sizeof(iris_refine_constraint));
    ctmp->left = v;
    ctmp->right = u;
    ctmp->gap = REFINE_MIN_GAP;
    ctmp->next = NULL;
    if (constraints == NULL)
    {
        constraints =
        ctail = ctmp;
    }
    else
    {
        ctail->next =
        ctmp;
        ctail = ctmp;
    }
    iris_queue_remove(left[u],
    v);
    iristrbtree_remove_value(scanline,
    graph->vertex[v].trans[dim]);
}
free(scanline);
free(events);
for (i = 0; i < graph->vertex.count; i++)
{
    if (left[i] != NULL)
        iris_queue_destroy(left[i]);
    if (right[i] != NULL)
        iris_queue_destroy(right[i]);
}
free(left);
free(right);
return (fail) ? (iris_refine_constraint*)1 :
    constraints;
}

// Destroy constraints ———
void iris_refine_constraint_destroy(iris_refine_constraint *C)
{
    iris_refine_constraint *c;
    if (C == NULL) return;
    c = C->next;
    free(C);
    if (c != NULL)
    {
        iris_refine_constraint_destroy(c);
    }
}

// left/right order constraints ———
void iris_refine_constraint_order (iris_refine_constraint *C, iris_rbtree_tree *leftOrder, iris_rbtree_tree *rightOrder)
{
    iris_refine_constraint *c;
    c = C;
    while (c != NULL)
    {
        iris_rbtree_insert_count(leftOrder, c->left, c->left, c);
        iris_rbtree_insert_count(rightOrder, c->right, c->right, c);
        c = c->next;
    }
}

// total order ———
void iris_refine_explore(int v, iris_refine_constraint *C, iris_rbtree_tree *leftOrder, int *visited, int *ordering, int *opos)
{
    iris_refine_constraint *c;
    iris_rbtree_node *node;
    int i;

    visited[v] = 1;
    ordering[*opos] = v;
    *opos += 1;
    node = iris_rbtree_search(leftOrder, v);
    if (node != NULL)
    {
        for (i = -1; i < node->count; i++)
        {
            if (i < 0)
            {
                c = ((iris_refine_constraint*)node->ptr);
            }
            else
            {
                c = ((iris_refine_constraint*)node->ptrs[node->count - 1 - i]);
            }
            if (!visited[c->right])
            {
                iris_refine_explore(c->right, C, leftOrder, visited, ordering, opos);
            }
        }
    }
}

int *iris_refine_total_order(iris_graph *graph, iris_refine_constraint *C, int dim, iris_rbtree_tree
`leftOrder`)
{
    int i, *visited, *ordering, opos;
    visited = (int*)malloc(sizeof(int) *
        graph->vertex_count);
    bzero(visited, sizeof(int) *
        graph->vertex_count);
    ordering = (int*)malloc(sizeof(int) *
        graph->vertex_count);
    opos = 0;
    for (i = 0; i < graph->vertex_count; i++)
    {
        if (!visited[i]) iris_refine_explore(i,
            C, leftOrder, visited, ordering,
            &opos);
    }
    return ordering;
}

// Create a new block ———
iris_refine_block *iris_refine_block_create(int v,
    iris_graph *graph, int dim, iris_refine_constraint
    *C, iris_rbtree_tree *rightOrder)
{
    iris_refine_block *b;
    iris_refine_constraint *c, *ctmp;
    float priority;
    iris_rbtree_node *node;
    int i;
    b = (iris_refine_block*)malloc(sizeof(iris_refine_block));
    b->vars = (int*)malloc(sizeof(int));
    b->vars[0] = v;
    b->nvars = 1;
    b->posn = graph->vertex[v].trans[dim];
    b->weight = 1;
    b->wposn = b->weight * b->posn;
    b->active = NULL;
    b->active_end = NULL;
    b->time = ++timeCtr;
    b->in = NULL;
    node = iris_rbtree_search(rightOrder, v);
    if (node != NULL)
    {
        for (i = -1; i < node->count; i++)
        {
            if (i < 0)
            {
                c = (iris_refine_constraint*)node->ptr;
            }
            else
            {
                c = (iris_refine_constraint*)node->ptrs[i];
            }
            ctmp = (iris_refine_constraint*)malloc(sizeof(iris_refine_constraint));
            ctmp->left = c->left;
            ctmp->right = c->right;
            ctmp->gap = c->gap;
            ctmp->next = NULL;
            ctmp->time = b->time;
            priority =
                graph->vertex[c->left].trans[dim];
            priority += c->gap;
            priority -=
                graph->vertex[c->right].trans[dim];
            b->in = iris_max_queue_add(b->in,
                ctmp, priority);
        }
    }
    block[v] = b;
    blockptr[v] = b;
    offset[v] = 0;
    return b;
APPENDIX D. CODE

```c
// Deletes a block

void iris_refine_block_destroy(iris_refine_block *b) {
    iris_refine_constraint_destroy(b->active);
    free(b->vars);
    iris_max_queue_destroy(b->in);
    free(b);
}

// violation

float iris_refine_violation(iris_refine_constraint *c) {
    float viol;
    if (c == NULL) return 0;
    if ((block[c->left] == NULL || block[c->right] == NULL)) return 0;
    viol = block[c->left]->posn + offset[c->left];
    viol += c->gap;
    viol -= block[c->right]->posn + offset[c->right];
    return viol;
}

// top

iris_refine_constraint **iris_refine_top(iris_max_queue **q) {
    iris_refine_constraint **outOfDate, *c;
    int oodCnt, i;
    iris_max_queue *queue;
    iris_refine_block *l, *r;
    float priority;
    if (*q == NULL) return NULL;
    outOfDate = (iris_refine_constraint **)malloc(sizeof(iris_refine_constraint *) * oodCnt);
    queue = (iris_max_queue *)malloc(sizeof(iris_max_queue));
    if (queue == NULL) return NULL;
    *q = queue;
    for (i = 0; i < oodCnt; i++) {
        c = outOfDate[i];
        c->time = timeCtr;
        priority = iris_refine_violation(c);
        queue = iris_max_queue_add(queue, c, priority);
    }
    if (*q != NULL) *q = queue;
    if (queue == NULL) return NULL;
    return (iris_refine_constraint *)queue->data;
}
```

void iris_refine_merge_block(iris_refine_block *p,
   iris_refine_constraint *c, iris_refine_block *b,
   float distptob, int blnum)
{

   int i;
   p->wposn = p->wposn + b->wposn - (distptob * b->weight);
   p->weight = p->weight + b->weight;
   p->wposn = p->wposn / p->weight;
   if (p->active == NULL)
      { p->active = b->active;
        p->active_end = b->active_end;
      }
   else
     {
      if (b->active != NULL)
        {
        p->active_end->next = b->active;
        p->active_end = b->active_end;
        }
     }
   if (p->active != NULL)
      {
         p->active_end->next = c;
         p->active_end = c;
      }
   else
     {
        p->active = c;
        p->active_end = p->active;
     }
   for (i = 0; i < b->nvars; i++)
   {
      block[b->vars[i]] = p;
      offset[b->vars[i]] += distptob;
   }

   p->vars = (int*)realloc(p->vars, sizeof(int) * (p->nvars + b->nvars));
   for (i = 0; i < b->nvars; i++)
   {
      p->vars[i + p->nvars] = b->vars[i];
   }
   p->nvars += b->nvars;
   timeCtr++;
   iris_refine_top(&p->in);
   iris_refine_top(&b->in);
   p->in = iris_max_queue_merge(p->in, b->in);
   b->time = timeCtr;
}

void iris_refine_merge_left(iris_refine_block *b)
{
   iris_refine_constraint *top, *c;
   iris_refine_block *bl;
   float distbltob;
   top = iris_refine_top(&b->in);
   while (iris_refine_violation(top) > 0)
   {
      c = top;
      b->in = iris_max_queue_remove_max(b->in);
      bl = block[c->left];
      distbltob = offset[c->left] + c->gap -
                  offset[c->right];
      if (b->nvars > bl->nvars)
         {
            iris_refine_merge_block(b, c, bl,
                                     -distbltob, c->left);
         }
   else
void iris_refine_satisfy(iris_graph *graph, iris_refine_constraint *C, int dim)
{
    int i, *v;
    iris_refine_block *b;
    iris_rbtree_tree *leftOrder, *rightOrder;
    if (C != NULL)
    {
        bzero(block, sizeof(iris_refine_block*) * graph->vertex_count);
        bzero(offset, sizeof(float) * graph->vertex_count);
        leftOrder = iris_rbtree_create();
        rightOrder = iris_rbtree_create();
        iris_refine_constraint_order(C, leftOrder, rightOrder);
        timeCtr = 0;
        nodeCount = graph->vertex_count;
        v = iris_refine_total_order(graph, C, dim, leftOrder);
        for (i = 0; i < graph->vertex_count; i++)
        {
            b =
                iris_refine_block_create(v[i], graph, dim, C, rightOrder);
            iris_refine_merge_left(b);
        }
        iris_refine_constraint_destroy(C);
        for (i = 0; i < graph->vertex_count; i++)
        {
            if (block[i] != NULL)
            {
                graph->vertex[i].trans[dim]
                    = block[i]->posn + offset[i];
            }
        }
        for (i = 0; i < graph->vertex_count; i++)
        {
            iris_refine_block_destroy(blockptr[i]);
            block[i] = NULL;
        }
        iris_rbtree_destroy(leftOrder);
        iris_rbtree_destroy(rightOrder);
    }
}

// Public Functions

int iris_refine_process(iris_graph *graph)
{
    iris_refine_constraint *constraints;
    int dim;
    offset = (float*)malloc(sizeof(float) * graph->vertex_count);
    block = (iris_refine_block**)malloc(sizeof(iris_refine_block*) * graph->vertex_count);
    blockptr = (iris_refine_block**)malloc(sizeof(iris_refine_block*) * graph->vertex_count);
    if (!iris_params_get()->orth_mode)
    {
        for (dim = iris_params_get()->dims - 1;
            dim >= 0; dim --)
        {
            // Your code here
        }
    }
}


```c
{  
  
  constraints =  
  iris_refine_generate_constraints(graph,  
  dim);

  if (constraints ==  
      (iris_refine_constraint *))  
    return 0;

  iris_refine_satisfy(graph,  
                      constraints, dim);
}

else  
{
  for (dim = 0; dim <  
       iris_params_get() -> dims; dim++)
  {
    
    constraints =  
    iris_refine_generate_constraints(graph,  
    dim);

    if (constraints ==  
        (iris_refine_constraint *))  
      return 0;

    iris_refine_satisfy(graph,  
                        constraints, dim);
  }
}

free(offset);  
free(block);  
return 1;  
}

D.28 File: iris_refine.h

// Defines
```

```c
#define REFINE_MIN_GAP  
iris_params_get() -> refine
#define REFINE_EVENT_QUEUE_X 0  
#define REFINE_EVENT_QUEUE_Y 1  
#define REFINE_EVENT_QUEUE_Z 2  
// Structures
typedef struct _iris_refine_event  
{  
  int vertex;  
  float posn;  
  enum {OPEN, CLOSE} type;
} iris_refine_event;

typedef struct _iris_refine_constraint  
{  
  int left;  
  int right;  
  float gap;  
  struct _iris_refine_constraint *next;  
  int time;
} iris_refine_constraint;

typedef struct _iris_refine_block  
{  
  int *vars;  
  int nvars;  
  float posn;  
  float wposn;  
  float weight;
}  
```
iris_refine_constraint *active;
iris_refine_constraint *active_end;
iris_max_queue *in;
int time;
} iris_refine_block;

// Functions

int iris_refine_process(iris_graph *graph);

D.29 File: iris_spring.c

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "iris_function_data.h"
#include "iris_graph.h"
#include "iris_max_queue.h"
#include "iris_refine.h"
#include "iris_queue.h"
#include "iris_rbtree.h"
#include "iris_spring.h"

#define MIN(x, y) (((x) < (y)) ? (x) : (y))
#define MAX(x, y) (((x) > (y)) ? (x) : (y))
#define ABS(x) (((x) < 0) ? (-1 * (x)) : (x))

// Private Functions

// Force Calculations

float iris_spring_force_attract(float d)
{
    return (2.0f * log(d));
}

float iris_spring_force_repel(float d)
{
    return (1.0f / sqrt(d));
}

void iris_spring_apply_force(iris_graph *graph, int dim, int v, int u, float dist, float force)
{
    int i;
    float x;
    for (i = 0; i < dim; i++)
    {
        x = graph->vertex[v].trans[i] -
            graph->vertex[u].trans[i];
        graph->vertex[v].trans[i] += 0.01f * (x /
            dist) * force;
    }
}

// Distance

float iris_spring_distance(iris_graph *graph, int dim, int v, int u)
{
    int i;
    float temp, dist;
    temp = dist = 1;
    for (i = 0; i < dim; i++)
    {
        temp = graph->vertex[u].trans[i] -
            graph->vertex[v].trans[i];
        dist += temp * temp;
    }
    return sqrt(dist);
}

// Voxel Funcs

iris_spring_voxel *iris_spring_voxel_create(int id)
{
    iris_spring_voxel *vox;
}
```c
int gridsz, hgridsz;
float x, y;

vox = (iris_spring_voxel*)malloc(sizeof(iris_spring_voxel));
vox->voxelId = id;
vox->verts = iri_queue_create(5);

gridsz = 0xFFFFFFFF / SPRING_VOXEL_SZ;
hgridsz = gridsz / 2;

y = ((id - (id % gridsz)) - 1) * SPRING_VOXEL_SZ;
y = (y < hgridsz) ? (hgridsz - y) : (y - hgridsz);
y = (y * (float)SPRING_VOXEL_SZ +
     ((float)SPRING_VOXEL_SZ / 2.0f));
x = (id % gridsz);
x = (x < hgridsz) ? (hgridsz - x) : (x - hgridsz);
x = (x * (float)SPRING_VOXEL_SZ +
     ((float)SPRING_VOXEL_SZ / 2.0f));

vox->center[0] = x;
vox->center[1] = y;
return vox;
}

void iri_spring_voxel_destroy(iris_spring_voxel *vox)
{
    iri_queue_destroy(vox->verts);
    free(vox);
}

int iri_spring_get_voxel(iris_graph *graph, int v)
{
    int gridsz, hgridsz, col, row;
    gridsz = 0xFFFFFFFF / SPRING_VOXEL_SZ;
hgridsz = gridsz / 2;
    col = (int)ceil(graph->vertex[v].trans[0] / SPRING_VOXEL_SZ);
    row = (int)ceil(graph->vertex[v].trans[1] / SPRING_VOXEL_SZ);
    return ((row * gridsz) + col);
}

void iri_spring_update_voxels(iris_graph *graph, int v,
                              iris_rbtree_tree *voxels, int old_voxel)
{
    iris_spring_voxel *vox;
    iris_rbtree_node *node;
    int new_voxel;
    node = iri_rbtree_search(voxels, old_voxel);
    if (node != NULL)
    {
        vox = (iris_spring_voxel*)node->ptr;
        iri_queue_remove(vox->verts, v);
    }
    new_voxel = iri_spring_get_voxel(graph, v);
    node = iri_rbtree_search(voxels, new_voxel);
    if (node == NULL)
    {
        vox = (iris_spring_voxel*)node->ptr;
        iri_queue_add(vox->verts, v);
    }
    
    return vox;
}
```
```c
iris_queue *iris_spring_voxel_el(Node, int, int, int, int, int, int)
{
    int id;
    iris_rbtree_node *node;
    id = iris_spring_get_voxel(graph, v);
    node = iris_rbtree_search(voxels, id);
    if (node != NULL)
    {
        return ((iris_spring_voxel*)node->ptr)->verts;
    }
    return NULL;
}

int *iris_spring_voxel_surrounds(int id)
{
    int gridsz, *nbours;
    nbours = (int*)malloc(sizeof(int) * 8);
    gridsz = 0xFFFFFFFF / SPRING_VOXEL_SZ;
    if (id < gridsz)
    {
    } else
    {
        nbours[0] = id - 1 - gridsz;
        nbours[1] = id - gridsz;
        nbours[2] = id + 1 - gridsz;
    }
    if (id < gridsz + (gridsz - 1))
    {
        nbours[5] = id - 1 - gridsz;
        nbours[6] = id - gridsz;
        nbours[7] = id + 1 - gridsz;
    } else
    {
    }
    if (id % gridsz == 0)
    {
    } else
    {
        nbours[4] = id + 1;
    }
    if (id % gridsz == gridsz - 1)
    {
    } else
    {
        nbours[3] = id - 1;
    }
    return nbours;
}

void iris_spring_voxel_repulsion(iris_graph *graph, int v, int dim, iris_rbtree_tree *voxels)
{
    float dist, temp, force, x;
    int i, j, *nbours;
    iris_spring_voxel *vox;
    iris_rbtree_node *node;
    nbours = iris_spring_voxel_surrounds(iris_spring_get_voxel(graph, v));
    for (i = 0; i < 8; i++)
    {
        if (nbours[i] > -1)
        {
            node = iris_rbtree_search(voxels, nbours[i]);
            if (node != NULL)
```
```c
void iris_spring_init_layout(iris_graph *graph, int dim)
{
    int i, j;
    for (i = 0; i < graph->vertex_count; i++)
    {
        for (j = 0; j < dim; j++)
        {
            graph->vertex[i].trans[j] = rand() % (graph->vertex_count / 2);
        }
    }
}

// Center Graph ---
void iris_spring_center(iris_graph *graph, int dim)
{
    int i, j;
    float avg[3];

    for (i = 0; i < graph->vertex_count; i++)
    {
        for (j = 0; j < dim; j++)
        {
            graph->vertex[i].trans[j] /= graph->vertex_count;
        }
    }
}
```

```c
void iris_spring_init_layout(iris_graph *graph, int dim)
{
    int i, j;
    for (i = 0; i < graph->vertex_count; i++)
    {
        for (j = 0; j < dim; j++)
        {
            graph->vertex[i].trans[j] = rand() % (graph->vertex_count / 2);
        }
    }
}

// Center Graph ---
void iris_spring_center(iris_graph *graph, int dim)
{
    int i, j;
    float avg[3];

    for (i = 0; i < graph->vertex_count; i++)
    {
        for (j = 0; j < dim; j++)
        {
            graph->vertex[i].trans[j] /= graph->vertex_count;
        }
    }
}
```
void iris_spring_iter_attract(iris_graph *graph, int dim) {
    int v, u, e;
    float dist, force;
    for (v = 0; v < graph->sedges->row_count; v++)
    {
        for (e = 0; e < graph->sedges->rows[v].els_count; e++)
        {
            u = graph->sedges->rows[v].els[e].col;
            dist = iris_spring_distance(graph, dim, v, u);
            force = iris_spring_force_attract(dist);
            iris_spring_apply_force(graph, dim, v, u, dist, force);
            iris_spring_apply_force(graph, dim, u, v, dist, -force);
        }
    }
}

void iris_spring_iter_repel(iris_graph *graph, int dim) {
    int v, u, id, i;
    float dist, force;
    iris_rbtree_tree *voxels;
    iris_queue *queue;
    /*for (v = 0; v < graph->vertex_count; v++)
    {
        for (u = 0; u < graph->vertex_count; u++)
        {
            if (v != u)
            {
                dist = iris_spring_distance(graph, dim, v, u);
                force = iris_spring_force_repel(dist);
                /*iris_spring_apply_force(graph, dim, v, u, dist, force);*/
                /*iris_spring_apply_force(graph, dim, u, v, dist, -force);*/
            }
        }
    }
    voxels = iris_rbtree_create();
    for (v = 0; v < graph->vertex_count; v++)
    {
        id = iris_spring_get_voxel(graph, v);
        queue = iris_spring_voxel_els(graph, v, voxels);
        if (queue != NULL)
        {
            for (i = 0; i < queue->count; i++)
            {
                u = queue->items[i];
                if (v != u)
                {
                    dist = iris_spring_distance(graph, dim, v, u);
                    force = iris_spring_force_repel(dist);
                    /*iris_spring_apply_force(graph, dim, v, u, dist, force);*/
                    /*iris_spring_apply_force(graph, dim, u, v, dist, -force);*/
                }
            }
        }
    }
}
#define SPRING_ITER_MAX 100
#define SPRING_VOXEL_SZ 5

// Structures

typedef struct _iris_spring_voxel
{
  int voxelid;
  float center[3];
  iris_queue *verts;
} iris_spring_voxel;

// Functions

void iris_spring_process(iris_graph *graph, int dim);

D.30 File: iris_spring.h

APPENDIX D. CODE

D.31 File: iris_stack.c

// Structures

typedef struct _iris_stack_item
{
  void *data;
  struct _iris_stack_item *next;
} iris_stack_item;

typedef struct _iris_stack
{
  struct _iris_stack_item *top;
}
int count;
} iris_stack;

// Functions

iris_stack *iris_stack_create();
void iris_stack_destroy(iris_stack *stack);
void iris_stack_push(iris_stack *stack, void *data);
void *iris_stack_pop(iris_stack *stack);
int iris_stack_is_empty(iris_stack *stack);

D.32 File: iris_stack.h

// Structures

typedef struct _iris_stack_item {
    void *data;
    struct _iris_stack_item *next;
} iris_stack_item;
typedef struct _iris_stack {
    struct _iris_stack_item *top;
    int count;
} iris_stack;

// Functions

iris_stack *iris_stack_create();

void iris_stack_destroy(iris_stack *stack);
void iris_stack_push(iris_stack *stack, void *data);
void *iris_stack_pop(iris_stack *stack);
int iris_stack_is_empty(iris_stack *stack);

D.33 File: iris_util.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "iris_params.h"
#include "iris_util.h"

// Private Functions

cap_s *itoa(int val, int base)
{
    char buf[32] = {0};
    int i = 30;
    for (; val &amp; i ; —i, val /= base)
        buf[i] = "0123456789abcdef"[val % base];
    return &buf[i+1];
}

// Public Functions

int iris_util_edit(char *file, int line)
{
    char *editor, *args, *tmp;
    int len;
    if (system(NULL))
    {
        ...
    }
}
editor = iris_params_get()->editor;
tmp = itoa(line, 10);
len = strlen(editor) + strlen(file) +
    strlen(tmp) + 1;
args = (char*)malloc(sizeof(char) * len);
if (strstr("%s", editor) < strstr("%d",
    editor))
    {  
        snprintf(args, len, editor, file,
            line);
    }
else
    {  
        snprintf(args, len, editor, file,
            line);
    }
system(args);
    return 1;
else
    {
        printf("Cannot use system command!\n");
    }
    return 0;
}

D.34 File: iris_util.h

// Functions
int iris_util_edit(char *file, int line);

D.35 File: iris_visualiser.c

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <stdarg.h>
#include "iris.h"
#include "iris_params.h"
#include "iris_function_data.h"
#include "iris_graph.h"
#include "iris_visualiser.h"

#ifndef ABS
#define ABS(x) ( ((x) < 0.0f) ? (-1 * (x)) : (x) )
#endif

// Private Variables
iris_visualiser *iVIS;

// Forward Declarations
void iris_visualiser_generate_graph(iris_visualiser *
    vis);
int iris_visualiser_selection(int select);
void iris_visualiser_reset_selection();

// Private Methods

// Colouring

float iris_visualiser_hue_2_rgb(float v1, float v2, float
    h)
    {
        if(h < 0.0f) h += 1.0f;
        if(h > 1.0f) h -= 1.0f;
        if(6.0f * h < 1.0f) return (v1 + (v2 - v1) * 6.0f * h);
        if(2.0f * h < 1.0f) return v2;
        if(3.0f * h < 2.0f) return (v1 + (v2 - v1) * ((2.0f / 3.0f) - h) * 6.0f);
return v1;
}

float *iris_visualiser_hsl_2_rgb(float h, float s, float l)
{
    float v1, v2;
    float *rgb;
    v2 = (l < 0.5f) ? (l * (1.0f + s)) : (1 + s - (s * l));
    v1 = 2.0f * 1 - v2;
    rgb = (float*)malloc(sizeof(float) * 3);
    rgb[0] = iris_visualiser_hue_2_rgb(v1, v2, h + (1.0f / 3.0f));
    rgb[1] = iris_visualiser_hue_2_rgb(v1, v2, h);
    rgb[2] = iris_visualiser_hue_2_rgb(v1, v2, h - (1.0f / 3.0f));
    return rgb;
}

float *iris_visualiser_get_colour(iris_graph_vertex vertex)
{
    int found;
    iris_function_info *info;
    float saturation, group_index;
    float *rgb;
    float maxcnt;
    if (!iris_params_get()->colour_mode)
    {
        rgb = (float*)malloc(sizeof(float) * 3);
        rgb[0] = 0.3f;
        rgb[1] = 0.3f;
        rgb[2] = 0.3f;
        return rgb;
    }
    info = vertex.function;

    if (info->group != NULL)
    {
        found = 0;
        for (group_index = 0; group_index < _ivis->data->group_count;
             group_index++)
        {
            if (_ivis->data->groups[(int)group_index] == info->group)
            {
                found = 1;
                group_index++; // += VISUALISER_BASE_CLR;
                break;
            }
        }
        if (!found) group_index++;
        if (group_index > 360) group_index = VISUALISER_BASE_CLR;
        if (_ivis->data->group_count < 361)
        {
            group_index = ((361 - _ivis->data->group_count) / _ivis->data->group_count);
        }
    }
    else
    {
        group_index = VISUALISER_DEFAULT_CLR;
    }

    maxcnt = (float)iris_params_get()->func_max_count;
    saturation = VISUALISER_MIN_SATURATION + ((info->ref_count / maxcnt) * (1.0f - VISUALISER_MIN_SATURATION));
    return iris_visualiser_hsl_2_rgb((group_index / 360.0), saturation, saturation * 0.5);
void iris_visualiser_move_node(int x, int y, int z) {
    float xpos, ypos, zpos, xt, yt, zt, dx, dy, dz;
    float m[16];
    int i, node;
    
    glLoadIdentity();
    glLookAt(0, 0, VISUALISER_ZOOM_MAX - (VISUALISER_ZOOM_MULT * _vis->zoom), 0, 0, -1, 0, 1, 0);
    glTranslatef(_vis->panx, _vis->pany, _vis->zpos);
    glRotatef(_vis->angle_x, 0.0f, 1.0f, 0.0f);
    glRotatef(_vis->angle_y, 1.0f, 0.0f, 0.0f);
    glGetFloatv(GL_MODELVIEW_MATRIX, m);
    
    dx = VISUALISER_NODE_MOVE_AMOUNT * (float)x;
    dy = VISUALISER_NODE_MOVE_AMOUNT * (float)y;
    dz = VISUALISER_NODE_MOVE_AMOUNT * (float)z;
    
    xzpos = (float*)malloc(sizeof(int) * _vis->vertex_selected_count);
    ypos = (float*)malloc(sizeof(int) * _vis->vertex_selected_count);
    zpos = (float*)malloc(sizeof(int) * _vis->vertex_selected_count);

    for (i = 0; i < _vis->vertex_selected_count; i++) {
        node = _vis->vertex_selected[i];
        if (node < _vis->graph->vertex_count) {
            xt = xzpos[i];
            yt = ypos[i];
            zt = zpos[i];
            for (j = 0; j < _vis->graph->vertex_count; j++) {
                node = _vis->graph->vertex[j].node;
                if (node < _vis->graph->vertex_count) {
                    xz = _vis->graph->vertex[node].trans[0];
                    yz = _vis->graph->vertex[node].trans[1];
                    zt = _vis->graph->vertex[node].trans[2];
                }
            }
        }
    }
}
APPENDIX D. CODE  

137

{ free(xpos);  
free(ypos);  
free(zpos);  
}  

for (i = 0; i < _iivis->vertex_selected_count;  
i++)  
{  
    node = _iivis->vertex_selected[i];  
    if (node < _iivis->graph->vertex_count)  
    {  
        _iivis->graph->vertex[node].trans[0]  
        += VISUALISER_NODE_MOVE_AMOUNT * (float)x;  
        _iivis->graph->vertex[node].trans[1]  
        += VISUALISER_NODE_MOVE_AMOUNT * (float)y;  
        _iivis->graph->vertex[node].trans[2]  
        += VISUALISER_NODE_MOVE_AMOUNT * (float)z;  
    }  
    _iivis->selection_changed = 1;  
    _iivis->dirty = 1;  
}  

// Event Processing ------  
void iris_visualiser_zoom(int zoomin)  
{  
    float step = VISUALISER_ZOOM_STEP * abs(zoomin);  
    if (zoomin > 0 && _iivis->zoom < 1.0f)  
    {  
        _iivis->zoom += step;  
    }  
    else if (zoomin < 0 && _iivis->zoom > 0.0f)  
    {  
        _iivis->zoom -= step;  
    }  
}  

// Event Processing ------  
void iris_visualiser_exit()  
{  
    if (_iivis->window != -1)  
        glutDestroyWindow(_iivis->window);  
    free(_iivis);  
    iris_destroy();  
}  

void iris_visualiser_panx(int panleft, int fast)  
{  
    float step = VISUALISER_PAN_STEP;  
    if (fast) step *= fast * (1 - _iivis->zoom);  
    if (panleft)  
    {  
        _iivis->panx -= step;  
    }  
    else  
    {  
        _iivis->panx += step;  
    }  
}  

void iris_visualiser_pany(int panleft, int fast)  
{  
    float step = VISUALISER_PAN_STEP;  
    if (fast) step *= 2 * (1 - _iivis->zoom);  
    if (panleft)  
    {  
        _iivis->pany -= step;  
    }  
    else  
    {  
        _iivis->pany += step;  
    }  
}
void iris_visualiser_pos(int forward) {
    if (forward)
        _ivis->zpos += VISUALISER_ZPOS_STEP;
    else
        _ivis->zpos -= VISUALISER_ZPOS_STEP;
}

void iris_visualiser_reset() {
    _ivis->zoom = VISUALISER_ZOOM_DEFAULT;
    _ivis->panx = VISUALISER_PAN_DEFAULT;
    _ivis->pany = VISUALISER_PAN_DEFAULT;
    _ivis->zpos = 0.0f;
    _ivis->angle_x = 0.0f;
    _ivis->angle_y = 0.0f;
}

void iris_visualiser_update_origin() {
    if (_ivis->vertex_hover > -1 && _ivis->mouse_left)
    {
        _ivis->origin_x = _ivis->graph->vertex[_ivis->vertex_hover].trans[0];
        _ivis->origin_y = _ivis->graph->vertex[_ivis->vertex_hover].trans[1];
        _ivis->origin_z = _ivis->graph->vertex[_ivis->vertex_hover].trans[2];
        _ivis->zpos = _ivis->origin_z;
    }
    else if (_ivis->mouse_left)
    {
        _ivis->origin_x = 0.0f;
        _ivis->origin_y = 0.0f;
        _ivis->origin_z = 0.0f;
    }
    iris_visualiser_reshow(_ivis);
    glutPostRedisplay();
}

// Registered Event Handlers ———

void ViewOrtho(int x, int y) {
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    glOrtho(0, x, y, 0, -1, 1);
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();
}

void ViewPerspective(void) {
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();
}

void iris_visualiser_draw_overlay() {
    ViewOrtho(1, 1);
    glColor4f(1.0f, 1.0f, 1.0f, 0.5f);
    glRectf(_ivis->mouse_x_begin, _ivis->mouse_y_begin, _ivis->mouse_x_end, _ivis->mouse_y_end);
    ViewPerspective();
    ViewOrtho(_ivis->width, _ivis->height);
    glColor4f(1.0f, 1.0f, 1.0f, 0.5f);
    glRectf(_ivis->mouse_x_begin, _ivis->mouse_y_begin, _ivis->mouse_x_end, _ivis->mouse_y_end);
    ViewPerspective();
}

}
void iris_visualiser_draw_gl_scene()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(0, 0, VISUALISER_ZOOM_MAX - (VISUALISER_ZOOM_MULT * _ivis->zoom), 0, 0, -1, 0, 1, 0);
    glTranslatef(_ivis->panx, _ivis->pany, _ivis->zpos);
    glRotatef(_ivis->angle_x, 0.0f, 1.0f, 0.0f);
    glRotatef(_ivis->angle_y, 1.0f, 0.0f, 0.0f);
    if (_ivis->region_draw && _ivis->vertex_selected_count > 0)
    {
        if (_ivis->selection_changed)
        {
            if (_ivis->dl_selection_graph != -1)
            {
                glDeleteLists(_ivis->dl_selection_graph, 1);
            }
            _ivis->dl_selection_graph = glGenLists(1);
            glNewList(_ivis->dl_selection_graph, GL_COMPILEANDEXECUTE);
            iris_visualiser_generate_graph(_ivis);
            glEndList();
            _ivis->selection_changed = 0;
        }
        else
        {
            glCallList(_ivis->dl_selection_graph);
        }
    }
    else if (_ivis->sub_graph < 0)
    {
        if (_ivis->dirty)
        {
            glDeleteLists(_ivis->dl_graph, 1);
            _ivis->dl_graph = glGenLists(1);
            glNewList(_ivis->dl_graph, GL_COMPILE);
            iris_visualiser_generate_graph(_ivis);
            glEndList();
            _ivis->dirty = 0;
        }
        else if (_ivis->sub_graph == _ivis->sub_graph_last)
        {
            glCallList(_ivis->dl_sub_graph);
        }
        else
        {
            if (_ivis->has_sub_graph)
            {
                glDeleteLists(_ivis->dl_sub_graph, 1);
                _ivis->dl_sub_graph = glGenLists(1);
                glNewList(_ivis->dl_sub_graph, GL_COMPILEANDEXECUTE);
                iris_visualiser_generate_graph(_ivis);
                glEndList();
                _ivis->sub_graph_last = _ivis->sub_graph;
                _ivis->has_sub_graph = 1;
            }
            //iris_visualiser_draw_overlay();
            if (_ivis->mouse_left)
                iris_visualiser_draw_selector();
            glutSwapBuffers();
    }
}
```c
void iris_visualiser_resize_gl_scene(int Width, int Height)
{
    if (Height==0)
        Height=1;
    glViewport(0, 0, Width, Height);
    _ivis->width = Width;
    _ivis->height = Height;
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0f, (GLfloat)Width/(GLfloat)Height ,0.1f, 100.0f);
    glMatrixMode(GL_MODELVIEW);
}

void iris_visualiser_key_press(unsigned char key, int x, int y)
{
    int changed = 1;
    usleep(100);
    if (key >= 65 && key <= 90) key = ASCII_CAPS_TO_UPPER;
    changed = 1;
    switch(key)
    {
        case KEY_ESCAPE:
            iris_visualiser_exit(); break;
        case KEY_PLUS:
        case KEY_MINUS:
            iris_visualiser_zoom((key == KEY_PLUS) ? 1 : -1)); break;
        case KEY_A:
        case KEY_D:
            iris_visualiser_panx((key == KEY_A), 0); break;
        case KEY_S:
        case KEY_W:
            iris_visualiser_pos((key == KEY_W)); break;
        case KEY_R:
            iris_visualiser_reset(); break;
        case KEY_F:
            if (! _ivis->fullscreen)
                glutLeaveGameMode();
            iris_params_get()->window_mode = _ivis->fullscreen;
            _ivis->fullscreen = !_ivis->fullscreen;
            iris_visualiser_resize(_ivis);
            changed = 0; break;
        case KEY_L:
            _ivis->labels = !_ivis->labels;
            iris_visualiser_resize(_ivis);
            changed = 0; break;
        default:
            printf("KEY: %d\n", key);
            changed = 0; break;
    }
    _ivis->modifiers = glutGetModifiers(); _ivis->modifiers = glutGetModifiers();
    if (changed)
    {
        glutPostRedisplay();
    }
}

void iris_visualiser_handle_left_click(int button, int state, int x, int y)
{
    int *selected, count, i, ctrl, deselect;
    _ivis->mouse_left = (state != GLUT_UP);
    if (_ivis->mouse_left && _ivis->moving_nodes)
    {
        _ivis->moving_nodes = 0;
        _ivis->mouse_x_end = -1;
    }
```
if (_vis->moving_nodes) return;
if (((glutGetModifiers() & GLUT_ACTIVE_ALT) == GLUT_ACTIVE_ALT)
{  
    iris_visualiser_update_origin();
    return;
}

if (_vis->mouse_left)
{
    _vis->mouse_x_begin = _vis->mouse_x;
    _vis->mouse_y_begin = _vis->mouse_y;
    ctrl = (glutGetModifiers() &
           GLUT_ACTIVE_CTRL) ==
           GLUT_ACTIVE_CTRL;
    if (glutGet(GLUT_ELAPSED_TIME) -
        _vis->last_left_click <
        VISUALISER_DBL_CLK)
    {
        if (!ctrl)
        {
            _vis->sub_graph = _vis->vertex_hover;
            _vis->region_draw = 0;
            iris_visualiser_reset_selection();
        }
        glutPostRedisplay();
    }
    else
    {
        count = 0;
        selected = NULL;
    }
}

if (ctrl & _vis->region_draw &&
    _vis->vertex_selected_count > 0)
{
    selected = (int*)malloc(sizeof(int) *
       _vis->vertex_selected_count);
    count = _vis->vertex_selected_count;
    for (i = 0; i < count; i++)
    {
        selected[i] = _vis->vertex_selected[i];
    }
}
    iris_visualiser_selection(0);
    iris_visualiser_reset_selection();
    if (_vis->vertex_hover != -1)
    {
        _vis->vertex_selected = (int*)malloc(sizeof(int));
        _vis->vertex_selected[0] = _vis->vertex_hover;
        _vis->vertex_selected_count = 1;
        if (ctrl)
        {
            _vis->region_draw = 1;
            _vis->selection_changed = 1;
        }
    }
    else
    {
        if (ctrl && count > 0)
        {
            _vis->vertex_selected = (int*)realloc(_vis->vertex_selected,
```c
int sizeof(int) * (count + 1);

ivis->vertex_selected_count = count + 1;

deselect = 0;

for (i = 0; i < count; i++)
{
    if (_ivis->vertex_selected[0] != selected[i])
    {
        _ivis->vertex_selected[i + 1] = selected[i];
    }
    else
    {
        _ivis->vertex_selected[i]
            _ivis->vertex_selected[i + 1] = selected[i];
    }
}

if (deselect)
{
    _ivis->vertex_selected[0] = -1;
    _ivis->region_draw = 1;
}

_ivis->mouse_x_end = x;
_ivis->mouse_y_end = y;
_ivis->region_select = 1;

if (_ivis->mouse_y_end != _ivis->mouse_y_begin ||
    _ivis->mouse_x_end != _ivis->mouse_x_begin)
{
    _ivis->visualiser_selection(1);
    _ivis->region_draw = 1;
    _ivis->selection_changed = 1;
    glutPostRedisplay();
}

_ivis->mouse_x_end = -1;
_ivis->mouse_y_end = -1;

}

void iris_visualiser(mouse(int button, int state, int x, int y)
{
    _ivis->modifiers = glutGetModifiers();
    switch (button)
    {
    case GLUT_LEFT_BUTTON:
        _ivis_visualiser_handle_left_click(button, state, x, y);
        break;
    case GLUT_MIDDLE_BUTTON:
        _ivis->mouse_middle = (state != GLUT_UP);
        break;
    case GLUT_RIGHT_BUTTON:
        _ivis->mouse_right = (state != GLUT_UP);
        break;
    case GLUT_WHEEL_UP:
        //iris_visualiser_zoom(5);
        break;
    case GLUT_WHEEL_DOWN:
        //iris_visualiser_pos(1);
        break;
    }
}
```
iris_visualiser_zoom(1);
glutPostRedisplay();
break;
case GLUTWHEELDOWN:
    //iris_visualiser_zoom(-5);
    //iris_visualiser_pos(0);
    iris_visualiser_zoom(-1);
    glutPostRedisplay();
    break;

    _ivis->mouse_x = -1;
    _ivis->mouse_y = -1;
    glutPostRedisplay();
}

void iris_visualiser_motion(int x, int y)
{
    float dx, dy;
int dxi, dyi;

    if (_ivis->mouse_x >= 0 && _ivis->mouse_y >= 0)
    {
        dx = _ivis->mouse_x - x;
        dy = _ivis->mouse_y - y;

        if (_ivis->angle_right)
        {
            _ivis->angle_x += dx * 1.0f;
            _ivis->angle_y += dy * 1.0f;
            glutPostRedisplay();
        }
        else if (_ivis->mouse_middle)
        {
            if (abs(dx) > abs(dy))
            {
                iris_visualiser_panx((dx > 0), abs(dx));
            }
            else
            {
                iris_visualiser_pany((dy < 0), abs(dy) * VISUALISER_PAN_V_MULTI);
            }
            glutPostRedisplay();
        }
        else
        {
            if (_ivis->modifier & GLUT_ACTIVE_CTRL) != 0 &
                (_ivis->modifier & GLUT_ACTIVE_SHIFT) != 0)// &
                    _ivis->vertex_selected > 0)
            {
                _ivis->moving_nodes = 1;
                dxi = (_ivis->mouse_x - x) % 10;
                dyi = (_ivis->mouse_y - y) % 10;
                iris_visualiser_move_node(dxi, dyi, 0);
            }
            else
            {
                _ivis->mouse_x_end = -1;
                _ivis->mouse_y_end = -1;
            }
        }
    }

    _ivis->mouse_x = x;
    _ivis->mouse_y = y;
    glutPostRedisplay();
}

void iris_visualiser_motion_passive(int x, int y)
{
    int hits;
}
if (_iris->mouse_x >= 0 & _iris->mouse_y >= 0) {
    hits = iris_visualiser_selection(0);
    glutSetCursor(((hits) ? GLUT_CURSOR_INFO : GLUT_CURSOR_INHERIT));
}

_iiris->mouse_x = x;
_iiris->mouse_y = y;

// Selection handling ———
int iris_visualiser_is_selected(int v)
{
    int i;
    for (i = 0; i < _iris->vertex_selected_count; i++) {
        if (_iris->vertex_selected[i] == v)
            return 1;
    }
    return 0;
}

void iris_visualiser_reset_selection()
{
    if (_iris->vertex_selected != NULL) {
        free(_iris->vertex_selected);
        _iris->vertex_selected = NULL;
        _iris->vertex_selected_count = 0;
        _iris->region_draw = 0;
    }
}

int iris_visualiser_selection(int select)
{
    GLuint buffer[VISUALISER_SELECTION_BUFFER];
    GLint hits, i;
    GLint object, sethover, zpos;
    GLint viewport[4];
    GLfloat ratio;
    float beginx, beginy, endx, endy, width, height;
    if (_iris->moving_nodes) select = 0;

    glGetIntegerv(GL_VIEWPORT, viewport);
    glSelectBuffer(VISUALISER_SELECTION_BUFFER, buffer);
    glRenderMode(GL_SELECT);
    glInitNames();
    glPushName(0);
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    ratio = (viewport[2]+0.0) / viewport[3];
    beginx = _iris->mouse_x;
    beginy = _iris->mouse_y;
    width = height = 1;
    if (_iris->region_select)
    {
        beginx = (endx < beginx) ? endx : beginx;
        beginy = (endy < beginy) ? endy : beginy;
        beginx += width / 2;
        beginy += width / 2;
    }
gluPickMatrix(beginx, viewport[3]−beginy, width, height, viewport);
gluPerspective(45, ratio, 0.1, 0xFFFFFFFF);
gMatrixMode(GL_MODELVIEW);
iris_visualiser_draw_gl_scene();
gMatrixMode(GL_PROJECTION);
glPopMatrix();
gMatrixMode(GL_MODELVIEW);
flush();
hits = glRenderMode(GL_RENDER);
if (hits > 0)
{
    object = −1;
    if (select)
        iris_visualiser_reset_selection();
    sethover = 0;
    _vis->vertex_hover = −1;
    zpos = 0xFFFFFFFF;
    for (i = 0; i < hits; i++)
    {
        object = buffer[i * 4 + 3];
        if (object < _vis->graph->vertex_count &&
            buffer[i * 4 + 1] < zpos)
        {
            _vis->vertex_hover = buffer[i * 4 + 3];
            zpos = buffer[i * 4 + 1];
            sethover = 1;
        }
    }
    if (object >= 0 && select &&
        object < _vis->graph->vertex_count)
    {
        _vis->vertex_selected_count++; 
        _vis->vertex_selected =
            (int*)realloc(_vis->vertex_selected, 
            sizeof(int) *
            _vis->vertex_selected_count);
        _vis->vertex_selected[_vis->vertex_selected_count - 1] = object;
    }
    if (_vis->region_select)
        if (_vis->vertex_selected_count == 0) _vis->region_draw = 0;
    if (!sethover) hits = 0;
}
if (_vis->region_select)
    if (_vis->vertex_selected_count == 0)
    _vis->region_select = 0;
if (hits <= 0)
{
    if (select)
    {
        iris_visualiser_reset_selection();
        _vis->vertex_selected_count = 0;
        _vis->vertex_hover = −1;
    }
    return hits;
}
// Drawing ———
void iris_visualiser_print(float x, float y, float z, char *str)
{
    char *c;
    glRasterPos3f(x, y, z);
for (c = str; *c != '\0'; c++)
{
    glutBitmapCharacter(GLUT_BITMAP_HELVETICA, 10, *c);
}

void iris_visualiser_normalise(float *x, float *y, float *z)
{
    float len;
    len = sqrt((*x * *x) + (*y * *y) + (*z * *z));
    *x = *x / len;
    *y = *y / len;
    *z = *z / len;
}

void iris_visualiser_cross_prod(float x1, float y1, float z1, float x2, float y2, float z2, float *rx, float *ry, float *rz)
{
    *rx = (y1 * z2) - (z1 * y2);
    *ry = (z1 * x2) - (x1 * z2);
    *rz = (x1 * y2) - (y1 * x2);
}

float iris_visualiser_node_vector(float x1, float y1, float z1, float x2, float y2, float z2)
{
    float dx, dy, dz, len;
    float forward_x, forward_y, forward_z;
    float right_x, right_y, right_z;
    float up_x, up_y, up_z;
    float matrix[16];
    dx = x2 - x1;
    dy = y2 - y1;
    dz = z2 - z1;
    len = sqrt((dx * dx) + (dy * dy) + (dz * dz));
    up_x = sin(0);
    up_y = -cos(0);
    up_z = 0;
    forward_x = x2 - x1;
    forward_y = y2 - y1;
    forward_z = z2 - z1;
    iris_visualiser_normalise(&forward_x, &forward_y, &forward_z);
    iris_visualiser_cross_prod(up_x, up_y, up_z, forward_x, forward_y, forward_z, &right_x, &right_y, &right_z);
    //iris_visualiser_normalise(&right_x, &right_y, &right_z);
    iris_visualiser_cross_prod(right_x, right_y, right_z, forward_x, forward_y, forward_z, &up_x, &up_y, &up_z);
    //iris_visualiser_normalise(&up_x, &up_y, &up_z);
    iris_visualiser_cross_prod(right_x, right_y, right_z, forward_x, forward_y, forward_z, &up_x, &up_y, &up_z);
    //iris_visualiser_normalise(&up_x, &up_y, &up_z);
    matrix[0] = right_x; matrix[1] = right_y;
    matrix[8] = forward_x; matrix[9] = forward_y;
    matrix[15] = 1;
    glMultMatrixf(matrix);
    return len;
}

void iris_visualiser_generate_graph(iris_visualiser *vis)
{
    int v, callee;
    iris_graph *graph;
    iris_graph_vertex vertex, v callee;
    float len;
    float alpha;
    GLUquadricObj *quadratic;
    int connected;
    float *clr;
    quadratic = gluNewQuadric();
gluQuadricNormals(quadratic, GLU_SMOOTH);

graph = vis->graph;

glInitNames();

alpha = 1.0f;
connected = 0;

for (v = 0; v < graph->vertex_count; v++)
{
    vertex = graph->vertex[v];
    glPushMatrix();
    if (vis->sub_graph > -1)
    {
        connected = (vis->graph->edges[v][vis->sub_graph] ||
                      vis->graph->edges[vis->sub_graph][v]);
        connected = (iris_graph_edge(vis->graph, v, vis->sub_graph) ||
                     iris_graph_edge(vis->graph, vis->sub_graph, v));
        
        alpha = (vis->sub_graph < 0 || connected || v == vis->sub_graph) ? 1.0f : VISUALISER_FADED_ALPHA;
    }
    else
    {
        if (_vis->region_draw) // vis->vertex_selected_count > 0
        {
            connected = 0;
            alpha = (iris_visualiser_is_selected(v)) ? 1.0f : VISUALISER_FADED_ALPHA;
        }

        if (vis->vertex_hover == v) // iris_visualiser_is_selected(v))
        {
            glColor4f(1.0f, 1.0f, 1.0f, alpha);
        }
        else
        {
            if (vis->data->entry_point != vertex.function)
            {
                clr = iris_visualiser_get_colour(vertex);
                glColor4f(clr[0], clr[1], clr[2], alpha);
                free(clr);
            }
            else
            {
                glColor4f(1.0f, 1.0f, 1.0f, alpha);
            }
        }
    }
    glPushName(v);
    if (vis->data->entry_point == vertex.function)
    {
        glColor4f(1.0f, 0.0f, 0.0f, alpha);
        glCallList(_vis->dl_entry_point);
    }
    else
    {
        glCallList(_vis->dl_function);
    }
    glPopName();

    if (_vis->labels && (_vis->sub_graph < 0 || connected || v == vis->sub_graph) &&
APPENDIX D. CODE

```c
vis->vertex_selected_count == 0 | |
(vis->vertex_selected_count > 0 &
iris_visualiser_is_selected(v)))
{
    glColor3f(1.0f, 1.0f, 1.0f);
    iris_visualiser_print(1, 0, 0,
vertex.function->name);
}

glPopMatrix();

for (callee = 0; callee <
graph->vertex_count; callee++)
{
    // if (graph->edges[v][callee])
    if (iris_graph_edge(graph, v,
callee))
    {
        v callee =
        graph->vertex[callee];

glPushMatrix();

    glTranslatef( vertex.trans[0] -
vis->origin_x,
vertex.trans[1] -
vis->origin_y,
vertex.trans[2] -
vis->origin_z);

len =
    // Preparation ———
    iris_visualiser_node_vector{vertex.trans[0] -
vis->origin_x,
vertex.trans[1] -
vis->origin_y,
vertex.trans[2] -
vis->origin_z,
vcallee.trans[0] -
vis->origin_x,
vcallee.trans[1] -
vis->origin_y,
vcallee.trans[2] -
vis->origin_z};

alpha = (vis->sub_graph <
0 || v ==
vis->sub_graph ||
callee ==
vis->sub_graph) ?
1.0f :
VISUALISER_FADED_ALPHA;

if (_vis->region_draw)
{
    alpha =
    (iris_visualiser_is_selected(v))
? alpha :
VISUALISER_FADED_ALPHA;
}

    glColor4f(VISUALISER_LINE_COLOUR,
alpha);
    glPushName(-1);
    gluCylinder(quadratic,
0.05f, 0.05f, len,
VISUALISER_LINE_DETAIL, 1);
    glPopMatrix();
}

void iris_visualiser_generate_lists(iris_visualiser *vis)
{
    int sub_graph, v_select_count, region_draw;

    v_select_count = vis->vertex_selected_count;
    vis->vertex_selected_count = 0;
    region_draw = vis->region_draw;
    vis->region_draw = 0;
    sub_graph = vis->sub_graph;
    vis->sub_graph = -1;
```
vis->dl_function = glGenLists(1);
glNewList(vis->dl_function , GL_COMPILE);
glBegin(GL_POLYGON);
glutSolidSphere (VISUALISER_SPHERE_SIZE , VISUALISER_SPHERE_DETAIL , VISUALISER_SPHERE_DETAIL);
glEnd ( ) ;
glEndList ( ) ;
vis->dl_entry_point = glGenLists(1);
glNewList(vis->dl_entry_point , GL_COMPILE);
glutSolidCube (VISUALISER_ENTRY_PT_SZ);
glEndList ( ) ;
vis->dl_graph = glGenLists(1);
glNewList(vis->dl_graph , GL_COMPILE);
iris_visualiser_generate_graph (vis);
glEndList ( ) ;
vis->sub_graph = sub_graph;
vis->sub_graph_last = 1;
vis->vertex_selected_count = v_select_count;
vis->selection_changed = 1;
vis->region_draw = region_draw;
}

void iris_visualiser_prepare_glut (iris_visualiser *vis) {
    char modestr[10];
    int width , height;

    width = glutGet(GLUT_SCREEN_WIDTH);
    height = glutGet(GLUT_SCREEN_HEIGHT);

    vis->width = width;
    vis->height = height;

    glutInitDisplayMode (GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    if (!iris_params_get ()->window_mode &&
        glutGameModeGet (GLUT_GAME_MODE_POSSIBLE))
    {
        bzero(modestr, 10);
        sprintf(modestr, "%dx%d", width , height);
        glutGameModeString (modestr);
        glutEnterGameMode ( ) ;
    }
    else
    {
        if (!iris_params_get ()->window_mode)
        {
            printf("Fullscreen mode not available!");
        }
        if (_vis->window != -1)
            glutDestroyWindow (_vis->window); 
        glutInitWindowSize (width , height);
        _vis->window = glutCreateWindow ("I.R.I.S");

    }

    iris_visualiser_generate_lists (vis);
    glHint(GL_LINE_SMOOTH_HINT , GL_NICEST);
    glEnable (GL_LINE_SMOOTH);
    glEnable (GL_BLEND);
    glBlendFunc (GL_SRC_ALPHA ,
                 GL_ONE_MINUS_SRC_ALPHA);
    glClearColor (0.0f , 0.0f , 0.0f , 0.0f);
    glClearDepth (1.0);
```c
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glext.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

struct iris_visualiser
{
    float zoom = VISUALISER_ZOOM_DEFAULT;
    float panx = VISUALISER_PAN_DEFAULT;
    float pany = VISUALISER_PAN_DEFAULT;
    int window = -1;
    float zpos = 0.0f;
    int mouse_left = 0;
    int mouse_middle = 0;
    int mouse_right = 0;
    int region_select = 0;
    int region_draw = 0;
    float angle_x = 0.0f;
    float angle_y = 0.0f;
    unsigned char vertice_hover = -1;
    unsigned char vertex_selected = NULL;
    unsigned char vertex_selected_count = 0;
    unsigned char selection_changed = 1;
    unsigned char d1_selection_graph = -1;
    unsigned char sub_graph = -1;
    unsigned char has_sub_graph = 0;
    unsigned char sub_graph_last = -1;
    unsigned char labels = 0;
    unsigned char fullscreen = !iris_params_get()->window_mode;
    float origin_x = 0.0f;
    float origin_y = 0.0f;
    float origin_z = 0.0f;
    int mouse_x_end = -1;
    int mouse_y_end = -1;
    int moving_nodes = 0;
    int selection_changed = 1;
    int dl_selection_graph = -1;
    int sub_graph = -1;
    int has_sub_graph = 0;
    int sub_graph_last = -1;
    int labels = 0;
    int fullscreen = !iris_params_get()->window_mode;
    float origin_x = 0.0f;
    float origin_y = 0.0f;
    float origin_z = 0.0f;
    int mouse_x_end = -1;
    int mouse_y_end = -1;
    int moving_nodes = 0;
};

iris_visualiser* iris_visualiser_init()
{
    iris_visualiser* vis;
    vis = (iris_visualiser*)malloc(sizeof(iris_visualiser));
    return vis;
}

void iris_visualiser_show(iris_visualiser* vis)
{
    vis = vis;
}
```

The code snippet includes the setup for OpenGL rendering, including the use of `glDepthFunc`, `glEnable`, `glMatrixMode`, and various function calls to initialize the visualizer. It also defines a structure for the visualizer and includes methods for initialization and display.
iris_visualiser_prepare(vis);
   glutMainLoop();
}

void iris_visualiser_reshow(iris_visualiser *vis)
{
   _vis = vis;
   iris_visualiser_prepare_glut(vis);
   glutMainLoop();
}

void iris_visualiser_set_graph(iris_visualiser *vis, iris_graph *graph)
{
   vis->graph = graph;
}

void iris_visualiser_set_data(iris_visualiser *vis, iris_function_data *data)
{
   vis->data = data;
}

D.36 File: iris_visualiser.h

#include <GL/glut.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glx.h>
#include <unistd.h>

// Defines

#define PI 3.14159265
#define VISUALISER_SPHERE_SIZE 0.3f
#define VISUALISER_SPHEREDETAIL 10
#define VISUALISER_LINEDETAIL 3
#define VISUALISER_ENTRY_PT_SZ 0.5f
#define VISUALISER_LINE_COLOUR 0.7f, 0.7f, 0.7f
#define VISUALISER_ZOOMMULT 48.0f
#define VISUALISER_ZOOMDEFAULT /s 0.5f/s 0.3f;
#define VISUALISER_ZOOMMAX 50
#define VISUALISER_ZOOMSTEP 0.005f
#define VISUALISER_PANDEFAULT 0.0f
#define VISUALISER_PAN_STEP 0.2f
#define VISUALISER_PANMULT 50
#define VISUALISER_ZPOS_STEP 0.2f
#define KEY_ESCAPE 27
#define KEY_PLUS 43
#define KEY_MINUS 95
#define ASCII_CAPS_TO_UPPER 32
#define KEY_A 97
#define KEY_D 100
#define KEY_W 119
#define KEY_S 115
#define KEY_R 114
#define KEY_F 102
#define KEY_L 108
\#define VISUALISER_SELECTION_BUFFER 2048
\#define VISUALISER_DBL_CLK 300
\#define VISUALISER_FADED_ALPHA 0.05f
\#define VISUALISER_MAX_REF_CNT 50
\#define VISUALISER_MIN_SATURATION 0.4f
\#define VISUALISER_BASE_CLR 10
\#define VISUALISER_DEFAULT_CLR 50
\#define VISUALISER_NODE_MOVE_AMOUNT 0.1f

// Structures

typedef struct iris_visualiser
{
    int window;
    float zoom;
    float panx;
    float pany;
    int mouse_left;
    int mouse_middle;
    int mouse_right;
    int mouse_x;
    int mouse_y;
    int mouse_x_begin;
    int mouse_y_begin;
    int mouse_x_end;
    int mouse_y_end;
    int region_select;
    int region_draw;
    float zpos;
    float angle_x;
    float angle_y;
} iris_visualiser;

// Functions

GLuint dl_sub_graph;
GLuint dl_entry_point;
GLuint dl_selection_graph;
iris_graph *graph;
iris_function_data *data;
int vertex_hover;
int *vertex_selected;
int vertex_selected_count;
int selection_changed;
int sub_graph;
int sub_graph_last;
int has_sub_graph;
int last_left_click;
int fullscreen;
int labels;
int width;
int height;
int menu;
float origin_x;
float origin_y;
float origin_z;
int modifiers;
int moving_nodes;
int dirty;

iris_visualiser *iris_visualiser_init();
void iris_visualiser_show();
void iris_visualiser_reshow(iris_visualiser *vis);
void iris_visualiser_set_graph(iris_visualiser *vis, iris_graph *graph);
void iris_visualiser_set_data(iris_visualiser *vis, iris_function_data *data);

D.37 File: main.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "iris.h"
#include "iris_params.h"
#include "iris_function_data.h"
#include "iris_graph.h"
#include "iris_visualiser.h"
#include "iris_heap.h"
#include "iris_max_queue.h"
#include "iris_refine.h"
#include "iris_hashtable.h"
#include "iris_stack.h"
#include "iris_extractor.h"
#include "iris_rbtree.h"
#include "iris_fudger.h"
#include "iris_queue.h"
#include "iris_spring.h"

extern int yyparse();

// Private Variables

iris_function_data *data;
iris_graph *graph;

// Private Functions

void iris_welcome()
{
    if (iris_params_get()->summary_mode ||
        iris_params_get()->verbose_mode)
    {
        printf("\n-----------------------------------------\n");
        printf("I.R.I.S. Interactive Realisation of Inscrutable Source Code\n");
        printf("-----------------------------------------\n");
    }
}

void iris_load_fake_data(int cnt)
{
    iris_function_group *grp, *grp2;
    iris_function_info *info;
    iris_function_info *info2;

    int i, vindex;

    graph = iris_graph_init(cnt);
    data = iris_function_data_init();
    info2 = NULL;

    grp = iris_function_group_create(data);
    grp2 = iris_function_group_create(data);

    for (i = 0; i < cnt; i++)
    {
        info = iris_function_info_create(data);
        info->group = (i < cnt/2) ? grp : grp2;
        info->ref_count = rand() % iris_params_get()->func_max_count;
        info->name = strdup("MyFunction");

        if (i == 0) info2 = info;

        vindex = iris_graph_add_vertex(graph, info);

        if (i > 0) iris_graph_add_edge(graph, vindex, 0);
    }
}
APPENDIX D. CODE

```c
void iris_load_data_debug()
{
    iris_load_fake_data(DEBUGGRAPH);
    printf("DEBUG MODE: Loaded debugging graph.\n\n" );
}

void iris_load_data_perf()
{
    iris_load_fake_data(PERF_TEST_GRAPH);
}

void iris_load_makefile()
{
    FILE *file;
    file = fopen("Makefile", "r");
    if (file == NULL)
    {
        printf("Makefile expected but not found!\n" );
        printf("See help for details.\n\n" );
        exit(0);
    }
    else
    {
        printf("Cannot parse makefiles yet!\n\n" );
        fclose(file);
    }
}

void iris_load_data()
{
    int i, file_count, verbose, count;
    char *file, *ch;
    iris_extractor *ext;

    file_count = iris_params_get()->file_count;
    verbose = iris_params_get()->verbose_mode;

    if (iris_params_get()->file_count == 0)
    {
        iris_load_makefile();
    }
    else if (verbose)
    {
        printf("-----\nProcessing Files ...\n\n" );
    }

    graph = iris_graph_init(iris_params_get()->func_count);
    data = iris_function_data_init();
    ext = iris_extractor_create(graph, data);
    for (i = 0; i < file_count; i++)
    {
        file = iris_params_get()->files[i];
        if (verbose)
        {
            printf("%s...", file);
        }
        count = iris_extractor_process(ext, file);
        if (verbose && count >= 0)
        {
            printf("%d\n", count);
        } else if (verbose)
        {
            printf("NOT FOUND!\n" );
        }
    }
```

APPENDIX D. CODE

void iris_summary()
{
    if (iris_params_get()->summary_mode || iris_params_get()->verbose_mode)
    {
        printf("Summary:\n");
        printf("\tFunctions: %d\n", data->function_count);
        printf("\tGroups: %d\n", data->group_count);
        printf("\n");
    }
    printf("\n");
    if (verbose)
    {
        printf("\t\t\n");
    }
    if (!iris_stack_is_empty(ext->unused) && verbose)
    {
        printf("\nFound Isolated Functions (%d): \n", ext->unused->count);
        while (!iris_stack_is_empty(ext->unused))
        {
            ch = iris_stack_pop(ext->unused);
            printf("\%s\n", ch);
            free(ch);
        }
        printf("\n");
    }
    else
    {
        printf("\n");
    }
    printf("\n");
    iriextractor_destroy(ext);
    if (verbose)
    {
        printf("\n");
    }
    else
    {
        printf("No files found to process!\n");
        printf("See help for details. \n\n");
    }
}

int iris_check_graph()
{
    int i;
    void iris_error(char *error)
    {
        int i;
        printf("\nError: \s\n", error);
        if (!iris_params_get()->verbose_mode)
        {
            printf("Enable verbose mode (-v) for details. \n\n");
        }
        else
        {
            printf("\nPositions: \n\n");
            for (i = 0; i < graph->vertex_count; i++)
            {
                printf("X: %f\tY: %f\tZ: %f\n",
                       graph->vertex[i].trans[0],
                       graph->vertex[i].trans[1],
                       graph->vertex[i].trans[2]);
            }
        }
        printf("\n\n");
        iridestroy();
    }
if (iris_params.get() -> verbose_mode)
{
    printf("Checking Graph...");
}

for (i = 0; i < graph -> vertex_count; i++)
{
    if (isnan(graph -> vertex[i].trans[0]) ||
        isnan(graph -> vertex[i].trans[1]) ||
        isnan(graph -> vertex[i].trans[2]))
    {
        printf("failed!\n\n");
        return 0;
    }
}

if (iris_params.get() -> verbose_mode)
{
    printf("done\n\n");
}
else if (iris_params.get() -> check_graph)
{
    check = iris_check_graph();
    if (!check && layout == GRPHAUTO)
    {
        printf("Regenerating Graph...");
        fflush(stdout);
        iris_spring_process(graph, dims);
        printf("done\n\n");
        refine = 0;
    }
    else if (!check)
    {
        iris_error("Unable to generate graph!");
    }
}
if (refine > 0.0f)
{
    if (iris_params.get() -> verbose_mode)
    {
        printf("Refining Graph [Min Gap: %.2f]...", refine);
        fflush(stdout);
    }
    success = 1;
for (i = 0; i < iris_params.get()->dims; i++) {
    if (!iris_fudger_process(graph, i)) {
        success = 0;
        break;
    }
}
if (success) {
    success = iris_refine_process(graph);
}
if (success && iris_params.get()->verbose_mode) {
    printf("done\n\n");
} else if (!success && iris_params.get()->verbose_mode) {
    printf("failed!\n\n");
    printf("\nCould not complete node overlap removal, view anyway? (y/n) : ");
    ch = getchar();
    if (ch != 'y' & & ch != 'Y') {
        printf("\nViewing aborted by user.\n\n");
exit(0);
    }
} else if (!success) {
    printf("Failed to refine graph, use verbose mode for details.\n\n");
exit(0);
}
}
}
// Public Functions

void iris_destroy() {
    iris_params_destroy();
    iris_function_destroy(data);
    iris_graph_destroy(graph);
exit(0);
}
// Main

int main(int argc, char **argv) {
    iris_visualiser *vis;
glutInit(&argc, argv);
    iris_params_init(argc, argv);
    iris_welcome();
    if (iris_params.get()->debug) {
        iris_load_data_debug();
    }
return 0;
}
else if (iris_params_get() -> perf)
{
    iris_load_data_perf();
}
else
{
    iris_load_data();
}
iris_summary();
if (graph -> vertex_count > 0)
{
    iris_draw_graph();
    if (iris_params_get() -> verbose_mode)
    {
        printf("Displaying...
Graph...\n\n\n");
    }
}
if (iris_params_get() -> visualise &
   !iris_params_get() -> perf)
{
    vis = iris_visualiser_init();
    iris_visualiser_set_graph(vis, graph);
    iris_visualiser_set_data(vis, data);
    iris_visualiser_show();
}
else if (iris_params_get() -> verbose_mode)
{
    printf("No functions found, aborting...
visualisation!\n\n\n");
}
return 0;