Incl: A Tool to Analyze Include Files

Kiem-Phong Vo, Yih-Farn Chen – AT&T Bell Laboratories

ABSTRACT

Large C and C++ software projects typically share common types, macros, and variables among modules via include files organized into hierarchies. Many of these hierarchies grow so complex that it is hard for programmers to figure out when a file must be included. Since including unused symbols is usually harmless, application code tends to include more files than required. Knowing when files are or are not needed is useful to restructure the code and reduce the time required to build a product. It also helps in reorganizing the include hierarchies – should this be deemed necessary. *Incl* is a tool that analyzes include hierarchies to (1) show the dependencies among include files in graphical or text forms, (2) infer what files are not needed, and (3) provide ways to remove unused include files. The inference and removal of unused include files must be done with care for that may change the meaning of the application programs. We shall describe precise conditions under which include files can be safely ignored for compilation and give a linear time algorithm to compute such files. *Incl* has been used on many projects and experience shows that, in many cases, eliminating unnecessary include files significantly reduces compilation time.

Motivation

C and C++ software systems typically share data types, macros, and declarations of global variables by including common header files. The header files and their interdependencies form include hierarchies. As with any other parts of a software system, an include hierarchy grows with a project as features are added, deleted, or refined, and eventually becomes large and complex. For example, the include hierarchy for the X Window System graphics library and tools on our machine contains over a hundred files in several different directories.

When an include hierarchy is sufficiently complex, it is hard for programmers to find out exactly when a file must be included. Since including a file that does not contain useful information is usually harmless, the tendency is to include enough files so that the code will compile. For large projects, this practice may even be institutionalized by providing global header files that simply include the world. This practice simplifies programming at the extra cost of compilation overhead due to the processing of unneeded include files. For projects that distribute source code, long build time may convey to customers a poor image of quality. Therefore, at some stages of software development, it is useful to find out when an include file is needed or not needed. This information can be used to redo the code and avoid unnecessary include files. It also helps software architects to reengineer the include hierarchies – should that be necessary.

For a given source file, finding what set of include files is needed requires construction and analysis of complex reference relationships among symbols across the nested include files. As an example, Figure 1 shows the include hierarchy and reference graph of a typical X11 application program[1] In this picture, edges between files mean either inclusion or reference relationships:

- A dotted edge means that the tail file includes the head file but does not directly use any information contained in it.
- A dashed edge means that the tail file does not include the head file directly but refers to information contained in it.
- A solid edge means both inclusion and reference.

All include files that are unnecessary for the compilation of `load.c` are shown in ovals. Among the 22 include files, 12 are unnecessary. For example, `X11/Intrinsic.h` includes `X11/Object.h`, but does not use any symbols defined in that file (dotted edge). On the other hand, `X11/Object.h` does not include `X11/Intrinsic.h`, but refers to some symbols defined in it (dashed edge).

The exploration of relationships among include files is an example of software reverse engineering. One attempts to reconstruct high level information about large objects (in this case, files) in a software system from the source code. The C Information Abstractor [2] creates a C program database from C source files that stores, among other things, the reference relationships between all global program objects (types, macros, functions, variables, and files). To analyze relationships among include files, we need all of these reference relationships. In fact, the determination of when to exclude an include file must be done with care so that the meaning of the

[1] To simplify the picture, path prefixes of the form `/usr/include` were replaced with `UINC` and `/usr/local/include/X11` with `X11`.

program does not change. Based on the CIA[2] database, we shall describe precise conditions for an include file to become unnecessary during compilation and give a linear time algorithm to detect such files.

We implemented the include file analysis algorithm and many of its common applications in a tool, *incl*, which can be used to generate textual or graphical representations of relationships among include files. It can also generate scripts usable with other programs such as *ed* , the line editor, or *cpp*, the C preprocessor to exclude unneeded include files.

[2]In this paper, we use *cia* to refer to the tool and CIA to refer to the system and concept.

The resulted tool has been used to reduce the compilation time by a third for many C source files. We shall give examples on the use of *incl* and statistics collected from a few projects.

Determining Unnecessary Include Files

The determination of what include files are needed depends on a number of factors defined by the compilation environment. This section discusses such factors and how they influence when files are needed. To simplify the discussion, for each file, we define the following sets based solely on the include relationships:

- *Closure(F)*: all include files included by F directly or indirectly.
- *First(F)*: include files directly included by F.

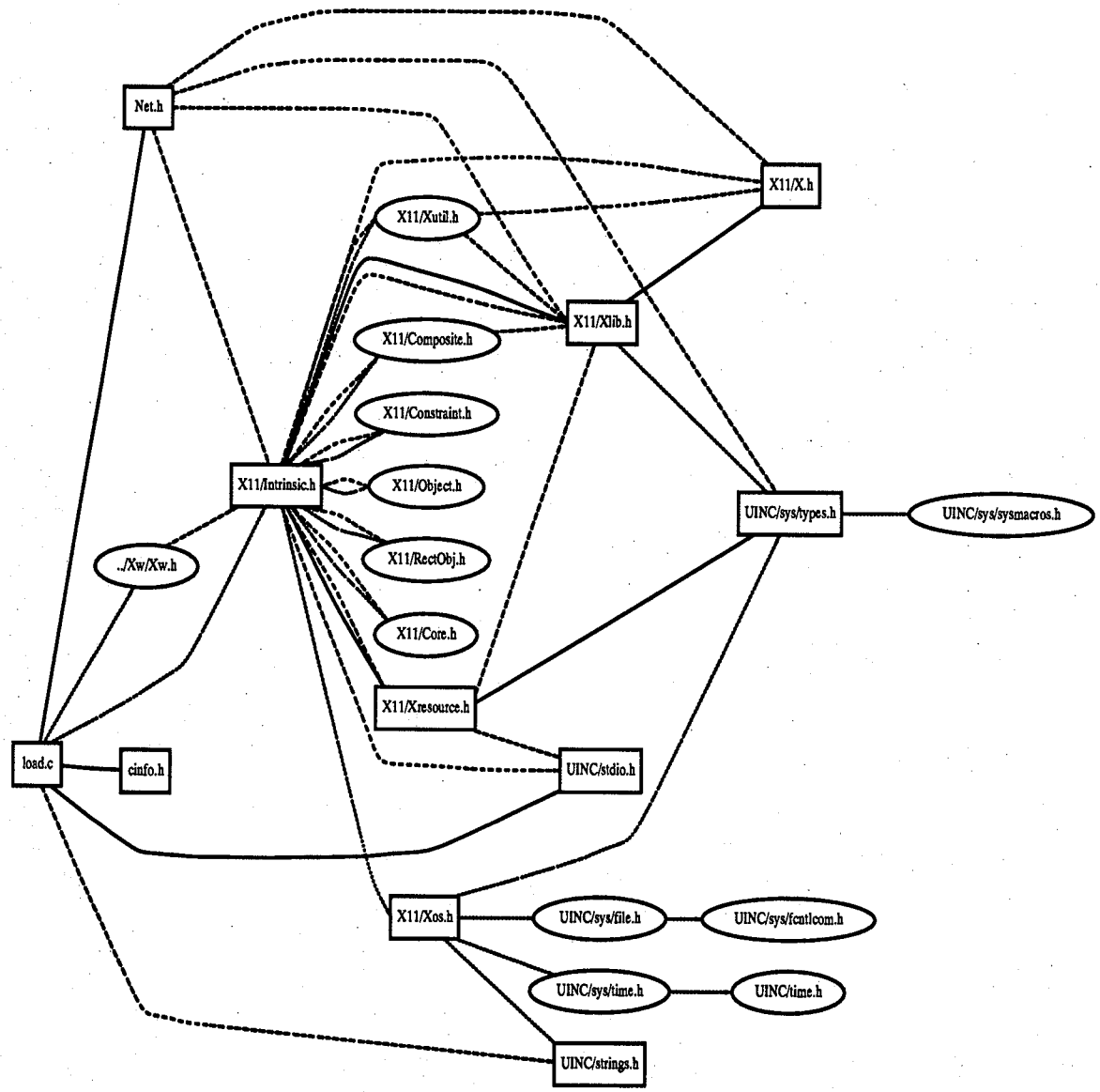

Figure 1: An Include and Reference Graph that Shows Unnecessary Include Files in Ovals

- *Nested(F)*: members in Closure(F) that are not in First(F).

For example, in Figure 1, First(`load.c`) consists of 5 members, Nested(`load.c`) consists of 17 members, and Closure(`load.c`) consists of 22 members.

C program global symbols can be divided into two classes, *definition* and *declaration*. A definition typically requires storage allocation while a declaration only gives type or value information without requiring any space in the generated code. Examples of definitions are functions and globally defined variables. Examples of declarations are macro and type definitions, or `extern` variables and functions. In the compilation of a C source file, all definitions are required. On the other hand, a declaration is required only if it is directly or indirectly referred to by some other definitions. If a declaration is not required, we shall call it *unused*.

The key observation about an unused declaration is that it cannot influence program behaviors. Therefore, reasonable compilation systems routinely ignore unused declarations in the generated code. This is the model of compilation that we shall assume for the rest of the paper. To rephrase, we assume that the generated code will remain the same with or without unused declarations. Given this, it would be nice if such declarations could be ignored completely during compilation. However, this is not always possible without extensive changes in either the source code or the underlying compilation system. Since virtually every C compilation system requires a file as the minimum compilation unit, a good compromise is to detect and ignore include files that consist only of unused declarations. This almost works except for two problems. The first problem is due to the way that most compilers process symbols. When a file is processed, all symbols that it refers to must be fully resolved whether or not they are really needed for code generation. This means that once a file is included, certain other files may be needed to resolve symbol references even though these symbols may not figure into the final code generation. The second problem is due to the way nested include files are processed. If an include file H is ignored, then all files in Closure(H) will also be ignored. This means that if an include file is deemed necessary by the above definition, at least a path of include files from the base file to it must be included. These considerations lead to the following recursive characterization of a necessary include file H:

1. H is necessary if it contains a definition of a variable or a function.
2. H is necessary if it contains a declaration of a symbol referenced by some other necessary file.
3. H may be necessary if it is on a path from the base file to a necessary file. If this path is unique, then H is necessary.

In developing an algorithm to mark files necessary, Conditions 1 and 2 are straightforward to implement. Condition 3 requires only that there is a path consisting of files marked necessary between the base file and any necessary file. Ideally, the number of marked files should be minimized. However, it is easily seen that this is an instance of the Steiner tree problem which is NP-complete [5]. Therefore, a heuristic approach is appropriate. The `path()` algorithm presented in the next section is a linear-time heuristic based on depth-first search.

```
[ hdr/db.h ]
#include "dir.h"
#include "cdb.h"
#include "error.h"
extern DIR *dbdir;
[ hdr/cdb.h ]
#define SYMDB "symbol.db"
typedef char *CDBNAME;
extern CDBNAME *dbs[];
extern DIR *dbdir;
[ hdr/error.h ]
#define ERR_FOPEN 1
[ hdr/dir.h ]
#include <sys/stat.h>
typedef char *DIR;
int CheckDir(/* char *dbdir */);
[ opendb.c ]
#include <stdio.h>
#include <ctype.h>
#include "db.h"
FILE *opendb(f)
char *f;
{
  FILE *fp;
#ifdef CDB
  if (!f) f=SYMDB; else f=dbs[0];
#endif
  if (!(fp=fopen(f, "w")))
    exit(ERR_FOPEN);
  else return(fp);
}
```

Figure 2: A Simple C Program that Includes Unnecessary Include Files

To illustrate the concepts of include and reference relationships, Figure 2 shows a small C source file and associated include files. This example will also be used later to demonstrate various uses of *incl*, the program to analyze include relationships among files. Figure 3 shows all include and

reference relationships of this example, following the conventions used in Figure 1.

Following the definition of necessary files, we see that:

- Both `hdr/error.h` and `hdr/cdb.h` are necessary because there are symbols defined in these files referenced by `opendb.c`. They satisfy condition 2.
- `hdr/db.h` is necessary because it is on the include path from `opendb.c` to `hdr/cdb.h`. It satisfies condition 3.
- `hdr/dir.h` becomes necessary because it is referenced by `hdr/db.h`. It satisfies condition 2.
- `<ctype.h>` and `<sys/stat.h>` do not satisfy any of the three conditions; therefore, these two files are unnecessary for the compilation of `opendb.c`.

The Algorithm

Determining when files are needed for compilation requires knowledge of reference relationships among symbols. From the source code, the CIA system generates databases of global symbols and their reference relationships. We developed an algorithm to detect unnecessary include files based on CIA data and the definition given in the last section. Currently, CIA does not keep exact information about data structures and functions whose definitions span include files. This, in turn, incurs a limitation on the tool. However, one can easily argue that partial definitions of structures and functions in include files constitute bad programming practice. In our experiences, we have not seen code written this way.

To ease the description of the algorithm that detects necessary include files, we shall use a pseudo-C language. Each algorithm will be presented with line numbers which are used for references in subsequent discussions. First, we describe the primitives that retrieve data from a CIA database. These primitives are provided by CIA itself.

- `Ciaobj_t* getciaobj(basefile)`: This function reads a CIA object record pertaining to the file `basefile`. An object record contains at least the following fields:
  - `name`: the name of the object.
  - `file`: the file where it is defined.
  - `sclass`: the storage class of the object. We are interested in whether or not the object is defined or just declared in the associated `file`.
- `Ciaref_t* getciaref(basefile)`: This function reads a CIA reference record pertaining to the file `basefile`. Each reference record contains at least the following fields:
  - `name1, name2`: the names of the referring and referred objects.
  - `file1, file2`: the files where the objects appear.
  - `kind1, kind2`: the types of the objects.

The overall *incl* algorithm to detect necessary files is as follows:

```
 1: incl(basefile)
 2: {   root = makegraph(basefile);
 3:     dfnumber(root);
 4:     root->type = NECESSARY;
 5:     enqueue(root);
 6:     while(notempty())
 7:     {   node = dequeue();
 8:         reference(node);
 9:         path(node);
10:     }
11: }
```

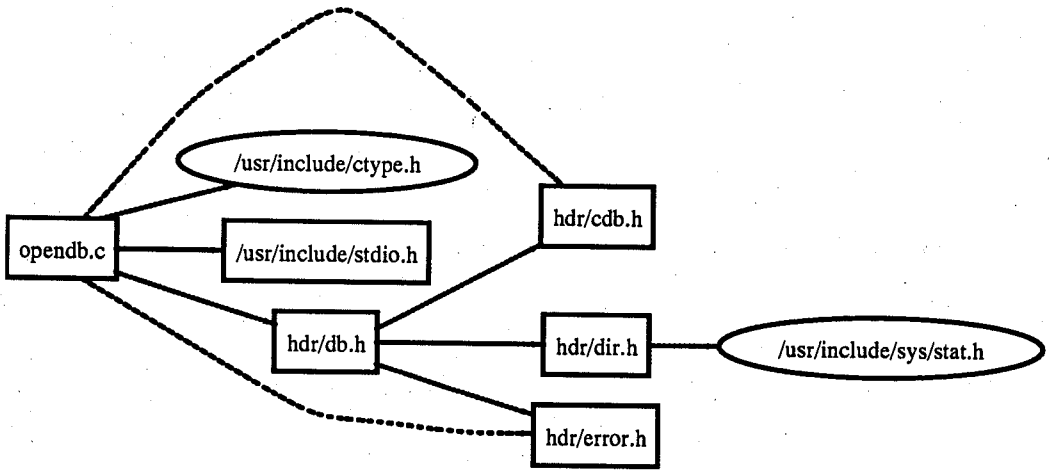

Figure 3: The Include and Reference Relationships in `opendb.c`

Remarks

Line 2 constructs the include/reference graph rooted at the file `basefile`. Line 3 computes a numbering of the nodes based on depth-first search. We shall omit the description of `dfnumber()` as its implementation is straightforward [1]. Lines 4-5 mark the root node as necessary, then insert it into the queue to be processed. We assume as given the operations `enqueue()`, `dequeue()` and `notempty()` which respectively inserts an element to the queue, deletes an element the queue and checks to see if the queue is not empty. Lines 6-10 process each element on the queue by first calling `reference()` to mark other necessary files that contain symbols directly or indirectly referenced from it. Then, `path()` is called to ensure that there is a path of files marked necessary from the root file to the given file.

Code Segment 1 shows the algorithm to construct the include/reference graph from a source file. Lines 2-10 read object descriptions from the CIA database and construct the nodes of the include graph. The function `getnode()` is called on line 3 to either find a node corresponding to the file `obj->file` or construct one if it does not yet exist. A node structure contains at least the following fields:

- `name`: the file name associated with the node.
- `type`: type of node.
- `edges`: edges outgoing from the node.
- `dfn`: the depth-first number of the node.

Lines 4-9 mark `node->type` as `NECESSARY` if the storage class of the object is statically defined (`s`) or globally defined (`g`) and insert the file into the queue of files that must be processed to resolve symbol references. This implements condition 1 of the definition of necessary files. Lines 11-22 read object reference information from the CIA database and construct the edges connecting the involved files. An edge connection is made via the function `getedge()`. Each edge structure contains the following fields:

- `node`: the head of the edge. This is the file being referenced.
- `type`: the type of edge.

Lines 15-21 mark `edge->type` as either `INCLUDE` if both objects are files or as `REFERENCE` otherwise. Note that an edge may be both `INCLUDE` and `REFERENCE`. When an `INCLUDE` edge is constructed, its opposite `INCLUDED` edge is also constructed. This opposite edge is used for an efficient implementation of the `path()` algorithm below. Finally, line 23 returns the root node of the include/reference graph.

Code Segment 2 shows the algorithm to mark files that contain symbols directly or indirectly referenced from a given file.

Line 2 iterates over each edge coming from `node`. Lines 3-4 ensures that only `REFERENCE` edges will be searched and only nodes that have not been searched will be searched. Lines 5-7 mark an

```
 1: makegraph(basefile)
 2: {   while((obj = getciaobj(basefile)) != EOF)
 3:     {   node = getnode(obj->file);
 4:         if(obj->sclass == 's' || obj->sclass == 'g')
 5:         {   if(node->type != NECESSARY)
 6:             {   node->type = NECESSARY;
 7:                 enqueue(node);
 8:             }
 9:         }
10:     }
11:     while((ref = getciaref(basefile)) != EOF)
12:     {   node1 = getnode(ref->file1);
13:         node2 = getnode(ref->file2);
14:         edge  = getedge(node1,node2);
15:         if(ref->kind1 == 'f' && ref->kind2 == 'f')
17:         {   edge->type |= INCLUDE;
18:             e = getedge(node2,node1);
19:             e->type |= INCLUDED;
20:         }
21:         else    edge->type |= REFERENCE;
22:     }
23:     return getnode(basefile);
24: }
```

Code Segment 1: Constructing include/reference graph

unsearched node as NECESSARY, then recurse. `reference()` implements condition 2 in the characterization of necessary files.

Code Segment 3 is the algorithm to mark nodes on paths from the root file to a necessary files.

To understand how `path()` works, we note that `dfnumber()` does two things: (1) it computes a spanning tree rooted at the base file, and (2) it numbers the nodes as they are searched. Now, for any given node H, let *in* (H) be the set of nodes with edges pointing to H. A property of depth-first numbering is that, for all K in *in* (H), if the depth-first number of K is smaller than that of H, then K is an ancestor of H in the spanning tree. Further, such ancestor nodes are numbered so that smaller numbered nodes are ancestors of higher ones in the spanning tree. Given this, it is easy to see that the `for(;;)` loop between lines 3-12 examines and selects some node from the set of ancestors of node to mark necessary. Lines 7-8 stop the algorithm without marking any node if such a node already exists. Lines 9-10 make sure that if a new node is to be marked, the one that is closest to the root will be selected.

Let F be a source file to be compiled and H some include file in Closure(F). We note that each NECESSARY file H is enqueued once and dequeued once. When the file is dequeued, `path()` is called to ensure that there is another NECESSARY file closer to the base file on the spanning tree that directly includes H. Thus, an easy induction based on the distance from the root file shows that:

Theorem 1. If H is an include file of F that is marked NECESSARY, there is an include path from F to H in which all files are marked NECESSARY.

Next consider an include file H in Closure(F) that is not marked NECESSARY by `incl()`. We need to show that H can be safely excluded in the compilation of F. Assume by contradiction that there is some symbol s in H that may affect the compilation of F. If s is a defined symbol, then `makegraph()` would mark H as necessary, a contradiction. This means that s must be a declared symbol. In this case, s can affect the compilation of F only because it is referenced by some other symbol in a file X that is marked necessary. Since X is marked necessary, it must be enqueued at some point in time. Upon dequeuing, it is examined by `reference()`. Now, the check on line 5 of

```
1: reference(node)
2: {   for(edge in node->edges)
3:     {   if(!(edge->type&REFERENCE) || edge->node->type == NECESSARY)
4:             continue;
5:         edge->node->type = NECESSARY;
6:         enqueue(edge->node);
7:         reference(edge->node);
8:     }
9: }
```

Code Segment 2: Marking referenced files

```
 1: path(node)
 2: {   mark = NULL;
 3:     for(edge in node->edges)
 4:     {   if(!(edge->type&INCLUDED))
 5:             continue;
 6:         if(edge->node->dfn < node->dfn)
 7:         {   if(edge->node->type == NECESSARY)
 8:                 return;
 9:             if(mark == NULL || e->node->dfn < mark->dfn)
10:                 mark = edge->node;
11:         }
12:     }
13:     if(mark != NULL)
14:     {   mark->type = NECESSARY;
15:         enqueue(mark);
16:     }
17: }
```

Code Segment 3: Marking nodes on paths

`reference()` would cause H to be marked necessary, a similar contradiction. This proves:

Theorem 2. If H is an include file of F that is not marked necessary by `incl()`, then it is safe to exclude H in the compilation of F.

Theorems 1 and 2 show that `incl()` correctly implements the conditions for necessary include files discussed previously. Therefore, in an appropriate compilation model, the files that `incl()` does not mark as necessary can be safely excluded. It would be nice to have the reverse, i.e., all files marked necessary are required for compilation. However, this is not generally true. Consider an example where a base file `a.c` includes two files `b.h` and `c.h`. In turn, both `b.h` and `c.h` include `d.h`. Finally, `c.h` includes `e.h`. Suppose that neither `b.h` nor `c.h` contains any definitions or declarations required by `a.c` but `d.h` and `e.h` do. It is clear that, for correct compilation of `a.c`, only `c.h`, `d.h`, and `e.h` are required. However, `path()` may also mark `b.h` as necessary if `dfnumber()` searches `b.h` before `c.h`. This shows a limitation of the `path()` heuristic algorithm.

Finally, we need to analyze the time requirement of `incl()`. This can be divided into two parts: the construction of the include/reference graph and the search for necessary include files. In constructing the graph, the time is dominated by the primitives to access the CIA database. However, these primitives are called exactly once for each symbol and reference relationships. The CIA database is arranged so that it takes constant time to perform each call to these primitives. The search for an existing node or edge by the functions `getnode()` and `getedge()` can be implemented in hash tables so that each function call is constant time on average. This means that the graph construction phase can be done in linear time (on average). In the search for necessary include files, first note that the depth-first numbering of the nodes requires linear time. Then, note that each node in the graph can be marked as `NECESSARY` exactly once. Each `REFERENCE` edge is searched at most once when its tail node is examined by `reference()`. Likewise, each `INCLUDED` edge is searched at most once when its head node is examined by `path()`. Thus, the total run time for `reference()` and `path()` is linearly bounded in the number of files and relationships among them. To sum this up, we have:

Theorem 3. The algorithm `incl()` runs in linear time in the number of files, symbols and symbol references.

Analyzing Include Hierarchies

Section 3 shows that finding unnecessary include files is computationally feasible. The program *incl* simplifies the analysis of include hierarchies by providing a variety of ways to extract and present information. Using the C program shown in Figure 2 as a base, this section gives examples on the use of *incl*. We shall use the convention that a user input command line starts with `$`, the shell prompt. Any output from the command will immediately follow this line.

Before *incl* can be used, a C program database file `opendb.A` must be created with the *cia* command using the same options as given to the C compiler:

```
$ cia -c -Ihdr -DCDB opendb.c
```

Note that compiler options like `-DCDB` may influence the file include relationships because of `#ifdef` 's. This, in turn, influences the working of *incl*.

After the database `opendb.A` is created, *incl* can be used to generate the include and reference graph derived from `opendb.c`. The best way to see this information is to generate a picture such as the one in Figure 3 with:

```
$ incl -R7,3.5 opendb.A
$ dag -Tps opendb.d
```

`Here, the option -R7,3.5` directs *incl* to generate a file `opendb.d` that contains a description of the include hierarchy as a directed graph to be drawn in an area that is 7 inches by 3.5 inches. Then, the program *dag* [4] is used to generate a picture specified in the PostScript language (the option `-Tps`).

Below is a quick scan to see what files are not needed. The result shows that two files can be skipped. One of them, (`/usr/include/sys/stat.h`), is included indirectly.

```
$ incl opendb.A
opendb.c:
    /usr/include/ctype.h
    /usr/include/sys/stat.h
```

To see the full hierarchy of include relationships with proper indentation, use the `-l` option. In this textual view, the character ~ tags unnecessary files in Closure(`opendb.c`):

```
$ incl -l opendb.A
opendb.c:
    /usr/include/stdio.h
    /usr/include/ctype.h
    hdr/db.h
        hdr/dir.h
            /usr/include/sys/stat.h~
        hdr/cdb.h
        hdr/error.h
```

After finding out what include files are not needed, a few options are available to eliminate or at least avoid them. The most direct approach is to edit a C source file F and remove any unneeded files

in First(F). For example, the statement `#include <ctype.h>` can be safely deleted from `opendb.c`. However, deleting `#include <sys/stat.h>` from `hdr/dir.h` is dangerous because `hdr/dir.h` may be used by other programs. Let's suppose for now that we have control over `opendb.c` so we can delete any unnecessary `#include` statements from it. We can run *incl* with the option `-e` to generate an *ed* script, then run *ed* to actually delete the unnecessary statements:

```
$ incl -e opendb.A
opendb.c:
            /usr/include/ctype.h
$ cat opendb.e
2d
w
$ ed opendb.c < opendb.e
"opendb.c" 16 lines, 219 characters
#include "db.h"
"opendb.c" 15 lines, 200 characters
```

Though it is not generally safe to delete code from a source file, `incl -e` can help in software reengineering. An example of this is to partition a large module of code into smaller units. Each new unit may start by including all original include files. Then, `incl -e` can be used to modify the new units so that they will include only what is needed.

In contrast to source files, deleting code from include files is inherently dangerous because header files are usually shared. In certain development organizations, for a variety of reasons, programmers may not be allowed to delete code from certain source files. In such a case, we must rely on a smart compilation system to skip unnecessary include files. The C preprocessor developed by Glenn Fowler and distributed with *nmake* [3] takes a special option `-I-I-.u` that reads *file* `.u` to determine what files to skip during C preprocessing. Assuming that this special C preprocessor is available, `incl -u` can be used to generate appropriate `.u` files for compilation:

```
$ incl -u opendb.A
opendb.A:
$ cat opendb.u
"/usr/include/ctype.h"
"/usr/include/sys/stat.h"
$ cc -I-I-.u -Ihdr -DCDB -c opendb.c
```

Figure 4 shows the include graph of the file `graphics.c` from a picture drawing program. Most path prefixes of the filenames and reference-only edges were removed to simplify the picture. Out of the 67 files included directly or indirectly by `graphics.c`, there are 52 unnecessary files (shown in oval nodes). Using a similar compilation approach as in the above example, *cpp* skipped these 52 files, which reduced the total lines that cpp processed from 23155 to 9076 – a saving of 61%. The total compilation time of `graphics.c` went from 10.04 (user+sys) seconds to 6.64 seconds on a SUN Sparcstation I, a saving of 34%. We shall give more detailed statistics on three projects in the next section.

Statistics

To see the effectiveness of using *incl*, we experimented with compiling the source code from two different projects on a Solbourne running SUN OS 4.0. These projects are based on the graphics facilities provided in SunView and X respectively. Figure 5 shows the data from the experiment. For comparative purpose, the third column of Figure 5 shows data from a module in a large project on a different machine architecture. This data was given to us from a user of *incl*.

measure	*Project A*	*Project B*	*Project C*
`NumSrcFiles`	23	19	35
`NumIncFiles`	74	54	163
`NumIncScans`	1158	378	1168
`NumIncSkips`	821	209	479
`PercIncSave`	71%	55%	41%
`OrgCompTime`	152 secs	96 secs	15m58s
`SkipCompTime`	98 secs	85 secs	11m41s
`PercTimeSaved`	36%	12%	26%

Figure 5: Include File Statistics on Three Projects

Here are the measures displayed in the table:

- `NumSrcFiles`: the number of files in the database with suffix `.c`.
- `NumIncFiles`: the number of files with suffix `.h`, including all user and system header files.
- `NumIncScans`: the total number of times that include files are scanned. Note that an include file shared by several source files may be scanned several times.
- `NumIncSkips`: the number of include file scans that can be skipped with `incl -u`.
- `PercIncSave`: this is the ratio of `NumIncSkips` over `NumIncScans`.
- `OrgCompTime`: the time it takes to compile all `.c` files. The time taken is the sum of the user time and the system time obtained by using the UNIX regmark] *time* command (actually, a built-in shell command in our case). It does not include the linking time.
- `SkipCompTime`: the time it takes to compile all `.c` files by ignoring unnecessary include files.
- `PercTimeSaved`: one minus the ratio of `SkipCompTime` over `OrgCompTime`.

As Figure 5 shows, the compile time saving ranges from 12% to 36%. Note that there are two different wastage problems with processing header

files: processing unnecessary files and processing files that are included multiple times. To avoid the latter, a number of the header files in the study are protected by pairs of `#ifndef, #endif`. This helps standard C preprocessors to avoid scanning such files more than once. Our C preprocessor (by Glenn Fowler) is smart enough to, in fact, avoid reopening such header files when applicable. Therefore,

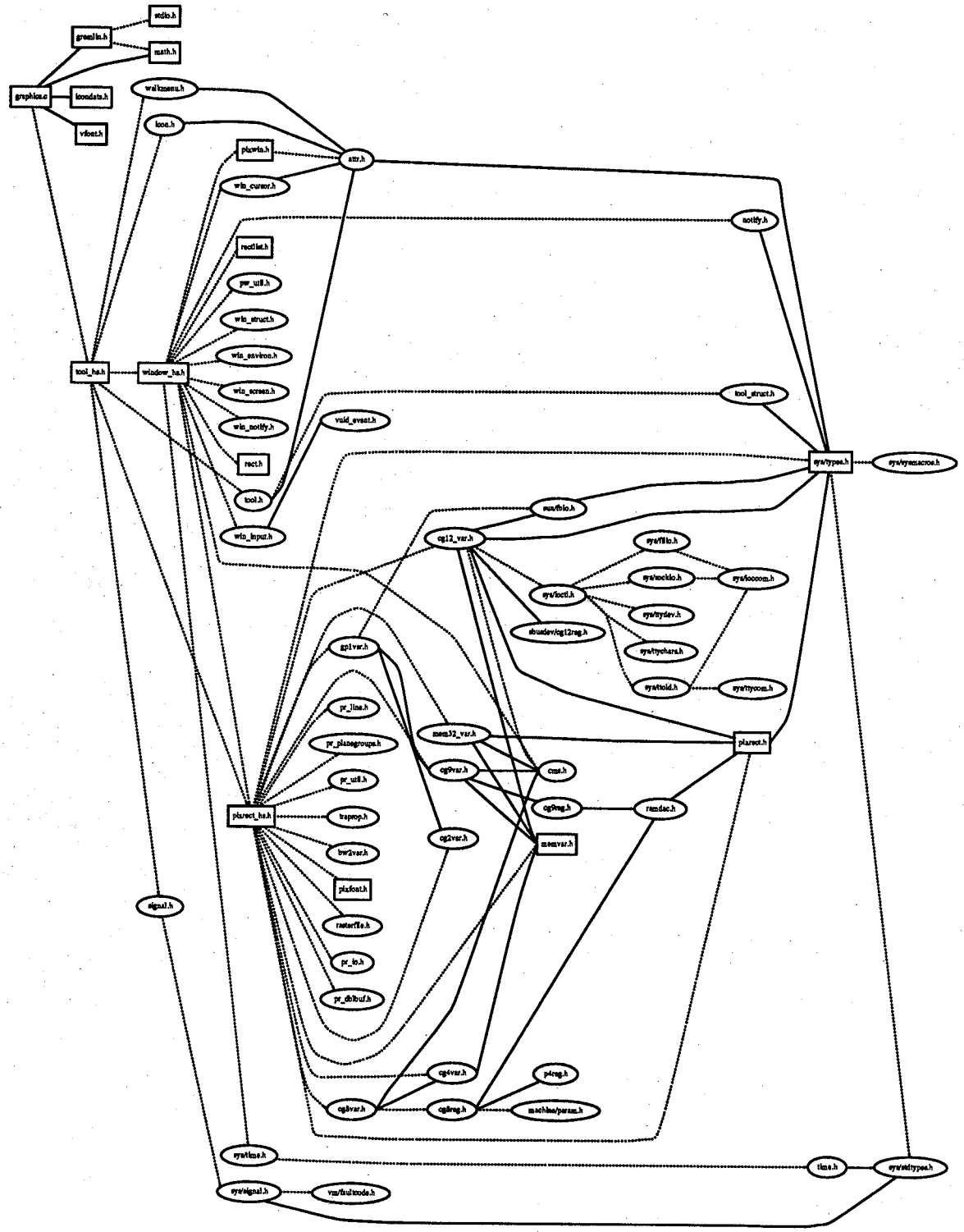

Figure 4: An Include Hierarchy with Many Unnecessary Include Files

the compile time via `incl -u` measures the true saving of skipping files that are unnecessary, and not just the saving of avoiding multiple file openings.

Conclusion

Reverse engineering is the activity of recovering from a system's implementation its high level objects and their interrelationships. This paper describes a tool, *incl*, suitable to analyze the relationships among include files in a C software system. The analysis of such files must be done with care if we want to use the resulting information to reengineer the software. We described a precise set of conditions under which include files could be safely ignored during compilation, implemented a linear time algorithm to compute such files, and proved the algorithm's correctness. Our method is general and could be used to analyze C++ include files, but we have not yet explored this direction.

Incl can be used to generate textual and graphical information on include hierarchies. Such information shows the include structures of large projects and provides a starting point in any effort to reengineer such structures. In a more limited fashion, *Incl* can also be used to generate scripts for automatic deletion of unused include files. We gave examples of how to use the program.

Eliminating unused include files can save significant compile time. *incl* can be used in conjunction with a smart C preprocessor to ignore unused include files during compilation. Note that this approach is different from the standard practice of using `#ifndef _HEADER_FILE` to avoid multiply included files. We presented experimental data showing that up to a third of compile time can be saved by ignoring unused include files.

Finally, *incl* is a part of the repertoire of C software analysis tools provided under the umbrella of the CIA system. It is a small application (650 lines of amply commented C code) written on top of the CIA database. The ease of its implementation shows that the CIA conceptual model and data provide a good basis for developing C analysis tools that deal with non-local C objects. This also shows the power of software tool modularity in which appropriate abstractions are captured and implemented in the right place.

References

[1] Alfred V. Aho and John E. Hopcroft and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley Publishing Company, 1974.

[2] Yih-Farn Chen. The C Program Database and Its Applications. In *USENIX Baltimore 1989 Summer Conference Proceedings*, 1989.

[3] G. S. Fowler. A Case for make. *Software – Practice and Experience*, 20:35-46, June 1990.

[4] E. R. Gansner and S. C. North and K. P. Vo. DAG – A Program that Draws Directed Graphs. *Software: Practice and Experience*, 18(11), November 1988.

[5] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

Author Information

Phong Vo lost his B.A degree somewhere but did manage to retain an M.A and a Ph.D. in Mathematics from the University of California at San Diego in 1977 and 1981 respectively. He joined Bell Labs in 1981 and is currently a Distinguished Member of Technical Staff. His research interests include graph theory, data structures and algorithms, user interface and reusable software tools. Aside from obscure theoretical works, Phong is responsible or partially responsible for a number of popular software tools including the current System V <curses> and malloc libraries, IFS, the Interpretive Frame System, a language for building applications with menu and form interfaces, and DAG, a program to draw directed graphs. He was awarded an AT&T Bell Labs Fellowship this year. Phong can be reached at kpv@ulysses.att.com or Kiem-Phong Vo, AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, New Jersey 07974.

Yih-Farn (Robin) Chen received the B.S. degree in electrical engineering from National Taiwan University, Taiwan, in 1980, the M.S. degree in Computer Science from University of Wisconsin, Madison, in 1983, and the Ph.D. degree in Computer Science from the University of California, Berkeley, in 1987. He is currently a Member of Technical Staff at AT&T Bell Laboratories in Murray Hill, New Jersey. His research interests include the modeling and integration of software databases, programming environments, and network management. Yih-Farn can be reached at chen@ulysses.att.com or Yih-Farn Chen, AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, New Jersey 07974.