Using Slicing to Identify Duplication in Source Code

Raghavan Komondoor
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706 USA
raghavan@cs.wisc.edu

Susan Horwitz
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706 USA
and
IEI del CNR, Pisa Italy
horwitz@cs.wisc.edu

Abstract. Programs often have a lot of duplicated code, which makes both understanding and maintenance more difficult. This problem can be alleviated by detecting duplicated code, extracting it into a separate new procedure, and replacing all the clones (the instances of the duplicated code) by calls to the new procedure. This paper describes the design and initial implementation of a tool that finds clones and displays them to the programmer. The novel aspect of our approach is the use of program dependence graphs (PDGs) and program slicing to find isomorphic PDG subgraphs that represent clones. The key benefits of this approach are that our tool can find non-contiguous clones (clones whose components do not occur as contiguous text in the program), clones in which matching statements have been reordered, and clones that are intertwined with each other. Furthermore, the clones that are found are likely to be meaningful computations, and thus good candidates for extraction.

1 Introduction

Programs undergoing ongoing development and maintenance often have a lot of duplicated code. The results of several studies [1, 12, 13] indicate that 7–23% of the source code for large programs is duplicated code. Duplication results in increased code size and complexity, making program maintenance more difficult. For example, when enhancements or bug fixes are done on one instance of the duplicated code, it may be necessary to search for the other instances in order to perform the corresponding modification. Lague et al [13] studied the development of a large software system over multiple releases and found that in fact, programmers often missed some copies of duplicated code when performing modifications.

A tool that finds clones (instances of duplicated code) can help alleviate these problems: the clones identified by the tool can be extracted into a new procedure, and the clones themselves replaced by calls to that procedure. In that case, there will be only one copy to maintain (the new procedure), and the fact that the procedure can be reused may cut down on future duplication. (Note that for a language like C with a preprocessor, macros can be used instead of procedures if there is a concern that introducing procedures will result in unacceptable performance degradation.)

For an example illustrating clone detection and extraction, see Figure 1. The left column shows four fragments of code from the Unix utility *bison*. The four clones are indicated by the "++" signs. The function of the duplicated code is to grow the buffer pointed to by `p` if needed, append the current character `c` to the buffer and then read the next character. In the right column, the duplicated code has been extracted into a new procedure `next_char`, indicated by the "++" signs, and all four clones replaced by calls to this procedure. The four calls are indicated by "**" signs.

This paper describes the design and initial implementation of a tool for C programs that finds clones suitable for procedure extraction and displays them to the programmer. The novel aspect of the work is the use of *program dependence graphs* (PDGs) [9], and a variation on *program slicing* [19, 16] to find isomorphic subgraphs of the PDG that represent clones. The key benefits of a slicing-based approach, compared with previous approaches to clone detection that were based on comparing text, control-flow graphs, or abstract-syntax trees, is that our tool can find non-contiguous clones (i.e., clones whose statements do not occur as contiguous text in the program, such as in Fragments 1 and 2 in Figure 1), clones in which matching statements have been reordered, and clones that are intertwined with each other. Furthermore, the clones found using this approach are likely to be meaningful computations, and thus good candidates for extraction.

The remainder of this paper is organized as follows: Section 2 describes how our tool uses slicing to find clones, and the benefits of this approach. Section 3 describes an implementation of our tool, and some of the insights obtained from running the tool on real programs. Section 4 discusses related work, and Section 5 summarizes our results.

2 Slicing-Based Clone Detection

2.1 Algorithm Description

To find clones in a program, we represent each procedure using its program dependence graph (PDG) [9]. In the PDG, nodes represent program statements and predicates, and edges represent data and control dependences. The algorithm performs three steps (described in the following subsections):

Step 1: Find pairs of clones.
Step 2: Remove subsumed clones.
Step 3: Combine pairs of clones into larger groups.

Step 1: Find pairs of clones. We start by partitioning all PDG nodes into equivalence classes based on the syntactic structure of the statement/predicate that the node represents, ignoring variable names and literal values; two nodes

Fragment 1:

```
   while (isalpha(c) ||
          c == '_' || c == '-') {
++    if (p == token_buffer + maxtoken)
++        p = grow_token_buffer(p);
      if (c == '-') c = '_';
++    *p++ = c;
++    c = getc(finput);
   }
```

Fragment 2:

```
   while (isdigit(c)) {
++    if (p == token_buffer + maxtoken)
++        p = grow_token_buffer(p);
      numval = numval*20 + c - '0';
++    *p++ = c;
++    c = getc(finput);
   }
```

Fragment 3:

```
   while (c != '>') {
      if (c == EOF) fatal();
      if (c == '\n') {
          warn("unterminated type name");
          ungetc(c, finput);
          break;
      }
++    if (p == token_buffer + maxtoken)
++        p = grow_token_buffer(p);
++    *p++ = c;
++    c = getc(finput);
   }
```

Fragment 4:

```
    while (isalnum(c) ||
           c == '_' || c == '.') {
++    if (p == token_buffer + maxtoken)
++        p = grow_token_buffer(p);
++    *p++ = c;
++    c = getc(finput);
    }
```

Rewritten Fragment 1:

```
   while (isalpha(c) ||
          c == '_' || c == '-') {
      if (c == '-') c = '_';
**    next_char(&p, &c);
   }
```

Rewritten Fragment 2:

```
   while (isdigit(c)) {
      numval = numval*20 + c - '0';
**    next_char(&p, &c);
   }
```

Rewritten Fragment 3:

```
   while (c != '>') {
      if (c == EOF) fatal();
      if (c == '\n') {
          warn("unterminated type name");
          ungetc(c, finput);
          break;
      }
**    next_char(&p, &c);
   }
```

Rewritten Fragment 4:

```
    while (isalnum(c) ||
           c == '_' || c == '.') {
**     next_char(&p, &c);
    }
```

Newly extracted procedure:

```
 void next_char(char **ptr_p, char *ptr_c){
++  if (*ptr_p == token_buffer + maxtoken)
++      *ptr_p = grow_token_buffer(*ptr_p);
++  *(*ptr_p)++ = *ptr_c;
++  *ptr_c = getc(finput);
 }
```

Fig. 1. Duplicated code from *bison*

in the same class are called matching nodes. Next, for each pair of matching nodes $(r1, r2)$, we find two isomorphic subgraphs of the PDGs that contain $r1$ and $r2$.

The heart of the algorithm that finds the isomorphic subgraphs is the use of backward slicing: starting from $r1$ and $r2$ we slice backwards in lock step, adding a predecessor (and the connecting edge) to one slice iff there is a corresponding, matching predecessor in the other PDG (which is added to the other slice). Forward slicing is also used: whenever a pair of matching loop or if-then-else predicates $(p1, p2)$ is added to the pair of slices, we slice forward one step from $p1$ and $p2$, adding their matching control-dependence successors (and the connecting edges) to the two slices. Note that while lock-step backward slicing is done from *every* pair of matching nodes in the two slices, forward slicing is done only from matching predicates. An example to illustrate the need for this kind of limited forward slicing is given in Section 2.2.

When the process described above finishes, it will have identified two isomorphic subgraphs (two matching "partial" slices) that represent a pair of clones. The process is illustrated using Figure 2, which shows the PDGs for the first two code fragments from Figure 1. (Function calls are actually represented in

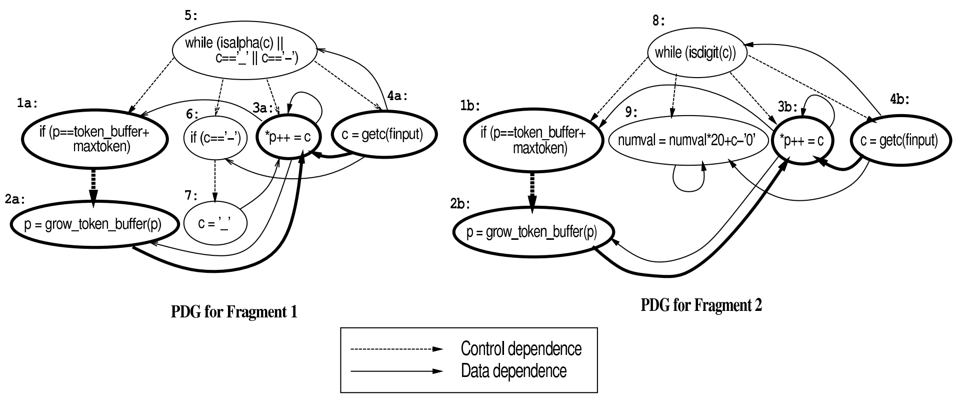

Fig. 2. Matching partial slices starting from `*p++ = c;`. The nodes and edges in the partial slices are shown in bold.

PDGs using multiple nodes: one for each actual parameter, one for the return value, and one for the call itself. For clarity, in this example we have treated function calls as atomic operations.) Nodes $3a$ and $3b$ match, so we can start with those two nodes. Slicing backward from nodes $3a$ and $3b$ along their incoming control-dependence edges we find nodes 5 and 8 (the two `while` nodes). However, these nodes do not match (they have different syntactic structure), so they are not added to the partial slices. Slicing backward from nodes $3a$ and $3b$ along their incoming data-dependence edges we find nodes $2a$, $3a$, $4a$, and 7 in the first PDG, and nodes $2b$, $3b$, and $4b$ in the second PDG. Node $2a$ matches $2b$, and node $4a$ matches $4b$, so those nodes (and the edges just traversed to reach them) are added to the two partial slices. (Nodes $3a$ and $3b$ have already been added, so those nodes are not reconsidered.) Slicing backward from nodes $2a$ and $2b$, we find nodes $1a$ and $1b$, which match, so they (and the traversed edges) are added. Furthermore, nodes $1a$ and $1b$ represent `if` predicates; therefore we slice forward from those two nodes. We find nodes $2a$ and $2b$, which are already in the slices, so they are not reconsidered. Slicing backward from nodes $4a$ and $4b$, we find nodes 5 and 8, which do not match; the same two nodes are found when slicing backward from nodes $1a$ and $1b$.

The partial slices are now complete. The nodes and edges in the two partial slices are shown in Figure 2 using bold font. These two partial slices correspond to the clones of Fragments 1 and 2 shown in Figure 1 using "++" signs.

Step 2: Remove subsumed clones A clone pair $(S1', S2')$ *subsumes* another clone pair $(S1, S2)$ iff $S1 \subseteq S1'$ and $S2 \subseteq S2'$. There is no reason for the tool to report subsumed clone pairs; therefore, this step removes subsumed clone pairs from the set of pairs identified in Step 1.

Step 3: Combine pairs of clones into larger groups This step combines clone pairs into clone groups using a kind of transitive closure. For example, clone pairs $(S1, S2)$, $(S1, S3)$, and $(S2, S3)$ would be combined into the clone group $(S1, S2, S3)$.

2.2 Need for Forward Slicing

Our first implementation of the clone-detection tool did not include any forward slicing. However, when we looked at the clones that it found we saw that they sometimes were subsets of the clones that a programmer would have identified manually. In particular, we observed that conditionals and loops sometimes contain code that a programmer would identify as all being part of one logical operation, but that is not the result of a backward slice from any single node.

One example of this situation is error-handling code, such as the two fragments in Figure 3 from the Unix utility *tail*. The two fragments are identical except for the target of the final `goto`, and are reasonable candidates for extraction; they both check for the same error condition, and if it holds, they both perform the same sequence of actions: calling the `error` procedure, setting the global `error` variable, and freeing variable `tmp`. (The final `goto` should of course not be part of the extracted procedure; instead, that procedure would need to return a boolean value to specify whether or not the `goto` should be executed.) However, the two fragments cannot be identified as clones using only backward slicing, since the backward slice from any statement inside the `if` fails to include any of the other statements in the `if`. It is the forward-slicing step from the pair of matched `if` predicates that allows our tool to identify these two code fragments as clones.

Fragment 1:

```
if (tmp->nbytes == -1)
  {
    error (0, errno, "%s", filename);
    errors = 1;
    free ((char *) tmp);
    goto free_lbuffers;
  }
```

Fragment 2:

```
if (tmp->nbytes == -1)
  {
    error (0, errno, "%s", filename);
    errors = 1;
    free ((char *) tmp);
    goto free_cbuffers;
  }
```

Fig. 3. Error-handling code from `tail` that motivates the use of forward slicing.

Other examples where forward slicing is needed include loops that set the values of two related but distinct variables (e.g., the head and tail pointers of a linked list). In such examples, although the entire loop corresponds to a single logical operation, backward slicing alone is not sufficient to identify the whole loop as a clone.

2.3 Preventing Clones that Cross Loops

Based on experience gained from applying the algorithm to real programs, we found that we needed a heuristic to prevent clones that "cross" loops; i.e., clones that include nodes both inside and outside a loop but not the loop itself. To illustrate this, consider the two code fragments (from *bison*) in Figure 4. The clones

Fragment 1:

```
   fp3 = lookaheadset + tokensetsize;
   for (i = lookaheads(state);
        i < k; i++) {
++     fp1 = LA +
             i * tokensetsize;
++     fp2 = lookaheadset;
++     while (fp2 < fp3)
++         *fp2++ |= *fp1++;
   }
```

Fragment 2:

```
   fp3 = base + tokensetsize;
   ...
   if (rp) {
       while ((j = *rp++) >= 0) {
           ...
++         fp1 = base;
++         fp2 = F +
++               j * tokensetsize;
++         while (fp1 < fp3)
++             *fp1++ |= *fp2++;
       }
    }
```

Fig. 4. Two clones from *bison* that illustrates the heuristic that avoids "crossing" a loop. These clones also illustrate variable renaming and statement reordering.

identified by our tool are shown using "++" signs. Each of these clones modifies a portion of a bit array (`lookaheadset` / `base`) by performing a bit-wise *or* with the contents of another array (`LA` / `F`). The clones are identified by slicing back from the statement that does the bit-wise *or*. Note that the two initial assignments to `fp3` are matching statements that are data-dependence predecessors of matching nodes in the two clones (the nodes that represent the final `while` predicates). Therefore, the algorithm as described above in Section 2.1 would have included the two initial assignments in the clones. It would not, however, have included in the clones the `for` loop in the first fragment and the outer `while` loop in the second fragment because the predicates of those loops do not match.

The resulting clones would therefore contain the statements inside the loops and the assignments outside the loops, but not the loops themselves. This would make it difficult to extract the clones into a separate procedure. To prevent the algorithm from identifying "difficult" clones like these, we use a heuristic during the backward slicing step: when slicing back from two nodes that are inside loops, we add to the partial slices predecessor nodes that are outside the loops only if the loop predicates match (and so will also be added to the partial slices). That is why, as indicated in Figure 4, the initial assignments to `fp3` are not included in the clones identified by the tool.

2.4 Benefits of the Approach

As stated in the Introduction, the major benefits of a slicing-based approach to clone detection are the ability to find non-contiguous, reordered, and intertwined clones, and the likelihood that the clones that are found are good candidates for extraction. These benefits, discussed in more detail below, arise mainly because slicing is based on the PDG, which provides an abstraction that ignores arbitrary sequencing choices made by the programmer, and instead captures the important dependences among program components. In contrast, most previous approaches to clone detection used the program text, its control-flow graph, or its abstract-syntax tree, all of which are more closely tied to the (sometimes irrelevant) lexical structure.

Finding non-contiguous, reordered, and intertwined clones: One example of non-contiguous clones identified by our tool was given in Figure 1. By running a preliminary implementation of the proposed tool on some real programs, we have observed that non-contiguous clones that are good candidates for extraction (like the ones in Figure 1) occur frequently (see Section 3 for further discussion). Therefore, the fact that our approach can find such clones is a significant advantage over most previous approaches to clone detection.

Non-contiguous clones are a kind of *near* duplication. Another kind of near duplication occurs when the ordering of matching nodes is different in the different clones. The two clones shown in Figure 4 illustrate this. The clone in Fragment 2 differs from the one in Fragment 1 in two ways: the variables have been renamed (including renaming `fp1` to `fp2` and vice versa), and the order of the first and second statements (in the clones, not in the fragments) has been reversed. This renaming and reordering does not affect the data or control dependences; therefore, our approach finds the clones as shown in the figure, with the first and second statements in Fragment 1 that are marked with "++" signs matching the second and first statements in Fragment 2 that are marked with "++" signs.

The use of program slicing is also effective in finding intertwined clones. An example from the Unix utility *sort* is given in Figure 5. In this example, one clone is indicated by "++" signs while the other clone is indicated by "xx" signs. The clones take a character pointer (`a`/`b`) and advance the pointer past all blank characters, also setting a temporary variable (`tmpa/tmpb`) to point to the first non-blank character. The final component of each clone is an `if` predicate that uses the temporary. The predicates were the starting points of the slices used to find the two clones (the second one – the second-to-last line of code in the figure – occurs 43 lines further down in the code).

```
++ tmpa = UCHAR(*a),
xx tmpb = UCHAR(*b);
++ while (blanks[tmpa])
++   tmpa = UCHAR(*++a);
xx while (blanks[tmpb])
xx   tmpb = UCHAR(*++b);
++ if (tmpa == '-') {
     tmpa = UCHAR(*++a);
     ...
   }
xx else if (tmpb == '-') {
     if (...UCHAR(*++b)...) ...
```

Fig. 5. An intertwined clone pair from *sort*.

Finding good candidates for extraction: As discussed in the Introduction, the goal of our current research is to design a tool to help find clones to be extracted into new procedures. In this context, a good clone is one that is meaningful as a separate procedure (functionally) and that can be extracted out easily without changing program semantics. The proposed approach to finding clones is likely to satisfy both these criteria as discussed below.

Meaningful clones: In order for a code fragment to be meaningful as a separate procedure, it should perform a single conceptual operation (be highly cohesive [17]). That means it should compute a small number of outputs (outputs include values assigned to global variables and through pointer parameters, the value returned, and output streams written). Furthermore, all the code to be extracted should be relevant to the computation of the outputs (i.e., the backward slices from the statements that assign to the outputs should include the entire clone).

A partial slice obtained using backward slicing has a good chance of being cohesive because we start out from a single node and include only nodes that are relevant to that node's computation. However, in addition to being cohesive, a meaningful procedure should be "complete". In practice, we have found that there are examples (like the one in Figure 3) where backward slicing alone omits some relevant statements. Our use of forward slicing seems to address this omission reasonably well.

Extractable clones: A group of clones cannot be eliminated by procedure extraction if it is not possible to replace the clones by calls to the extracted procedure without changing program semantics. Such clone groups are said to be inextractable. Since semantic equivalence is, in general, undecidable, it is not always possible to determine whether a group of clones is extractable. In [11] we identified sufficient conditions under which a single, non-contiguous clone can be extracted by first moving its statements together (making it contiguous), then creating a new procedure using the contiguous statements, and finally replacing the clone with a call to the new procedure.

In the example in Figure 1, the duplicated code indicated by the "++" signs meets the extractability criteria of [11]. However, in the same example, if we wanted each clone to consist of just the two lines indicated by "++" signs below, we would face problems:

```
++    if (p == token_buffer + maxtoken)
          p = grow_token_buffer(p);
++    *p++ = c;
```

There is no obvious way of extracting out just these two lines because the statement `p = grow_token_buffer(p)` cannot be moved out of the way from in between the above two lines without affecting data and control dependences (and hence without affecting semantics).

Because backward slicing follows dependence edges in the PDG, it is more likely to avoid creating a "dependency gap" (e.g., including the statement `*p++ = c` and its dependence grandparent `if (p == token_buffer + maxtoken)`, but omitting its dependence parent `p = grow_token_buffer(p)`) than a text- or syntax-tree based algorithm that detects non-contiguous clones. The heuristic described in Section 2.3 is another aspect of our approach that helps avoid identifying inextractable clones.

3 Experimental Results

We have implemented a preliminary version of our proposed tool to find clones in C programs using the slicing-based approach described above. Our implementation uses CodeSurfer [10] to process the source code and build the PDGs. CodeSurfer also provides a GUI to display the clone groups identified by the tool using highlighting.

The implementation of Step 1 of the algorithm (finding clone pairs) is done in Scheme, because CodeSurfer provides a Scheme API to the PDGs. The other two steps of the algorithm (eliminating subsumed clone pairs, and combining clone pairs into clone groups) are done in C++.

We have run the tool on three Unix utilities, *bison*, *sort* and *tail*, and on four files from a graph-layout program used in-house by IBM. The results of these experiments are presented in the following subsections.

3.1 Unix Utilities

Figure 6 gives the sizes of the three Unix utilities (in lines of source code and in number of PDG nodes), and the running times for the three steps of the algorithm. Figure 7 presents the results of running the tool on those three programs; for each of eight clone size ranges, three sets of numbers are reported: the number of clone groups identified that contain clones of that size, and the max and mean numbers of clones in those groups (the median number of clones in the groups of each size range was always two). Our experience indicates that clones with fewer than five PDG nodes are too small to be good candidates for extraction, so they are ignored by our tool.

	Program Size		Running Times (elapsed time)		
Program	# of lines of source	# of PDG nodes	find clone pairs (Scheme)	eliminate subsumed clone pairs (C++)	combine pairs into groups(C++)
bison	11,540	28,548	1:33 hours	15 sec.	50 sec.
sort	2,445	5,820	10 min.	5 sec.	2 sec.
tail	1,569	2,580	40 sec.	1 sec.	2 sec.

Fig. 6. Unix program sizes and running times

When run on the Unix utilities, the tool found a number of interesting clones, many of which are non-contiguous and some of which involve reordering and intertwining. These preliminary results seem to validate both the hypothesis that programs often include a significant amount of "near" duplication, and the potential of the proposed approach to find good quality clones.

Some examples of the interesting clones identified by the tool are listed below.

- The four-clone group shown in Figure 1, from *bison*.
- The two clones shown in Figure 4, from *bison*. These were part of a three-clone group. The third clone involved a different renaming of variables, and used the same statement ordering as the clone in Fragment 1.
- The pair of intertwined clones shown in Figure 5, from *sort*.
- A group of seven clones from *bison*, identical except for variable names. Two of the clones are shown in Figure 8. The clones were found by slicing back from the statement `putc(',', ftable)`. This code prints the contents of an array (`check` / `rrhs`), ten entries to a line, separated by commas.

	Clone Size Ranges (# of PDG nodes)							
bison	5–9	10–19	20–29	30–39	40–49	50–59	60–69	70–227
# clone groups	513	164	34	16	9	9	6	49
max # clones in a group	61	26	11	2	2	2	2	4
mean # clones in a group	3.7	2.8	3.3	2	2	2	2	2.1
sort	5–9	10–19	20–29	30–39	40–48			
# clone groups	105	57	30	9	14			
max # clones in a group	17	8	6	3	2			
mean # clones in a group	3.0	2.8	2.4	2.1	2			
tail	5–9	10–19	20–29	30–39	40–49	50–59	60–69	70–85
# clone groups	21	4	0	0	4	1	0	2
max # clones in a group	12	8			3	2		2
mean # clones in a group	3.2	3.5			2.3	2		2

Fig. 7. Results of running the tool

Fragment 1:
```
++  j = 10;
++  for (i=1; i < high; i++) {
++      putc(',', ftable);
++      if (j >= 10) {
++          putc('\n', ftable);
++          j = 1;
++      }
++      else
++          j++;
++      fprintf(ftable, "%6d", check[i]);
++  }
```

Fragment 2:
```
++  j = 10;
++  for (i=1; i < nrules; i++) {
++      putc(',', ftable);
++      if (j >= 10) {
++          putc('\n', ftable);
++          j = 1;
++      }
++      else
++          j++;
++      fprintf(ftable, "%6d", rrhs[i]);
++  }
```

Fig. 8. Seven copies of this clone were found in *bison*.

One limitation of the tool is that it often finds variants of the "ideal" clones (the clones that would be identified by a human) rather than finding exactly the ideal clones themselves. To illustrate this, consider the example in Figure 5. In that example, the ideal clones would not include the final `if` predicates; therefore, the clones found by the tool (which do include those predicates) are variants on the ideal ones. In the same example fragment, the tool also identifies a second pair of clones that is a slightly different variant of the ideal pair: this second pair includes everything in the ideal clones, does not include the `if` predicates, but does include the expressions `UCHAR(*++a)` and `UCHAR(*++b)` that occur in the last and fifth-to-last lines of code (the lines not marked with "++" or "xx" signs).

To further evaluate the tool we performed two studies, described below. The goals of the studies were to understand better:

a. whether the tool is likely to find (variants of) all of the ideal clones;
b. to what extent the tool finds multiple variants of the ideal clones rather than exactly the ideal ones;
c. how many "uninteresting" clones the tool finds (i.e., clones that are not variants of any ideal clone), and how large those clones are;
d. how often non-contiguous clones, intertwined clones, and clones that involve statement reordering and variable renaming occur in practice.

For the first study, we examined one file (*lex.c*) from *bison* by hand, and found four ideal clone groups. We then ran the tool on *lex.c*, and it identified forty-three clone groups. Nineteen of those groups were variants of the ideal clone groups (including several variants for each of the four ideal groups, so no ideal clones were missed by the tool), and the other twenty-four were uninteresting. More than half of the uninteresting clone groups (13 out of 24) had clones with fewer than 7 nodes (which was the size of the smallest ideal clone); the largest uninteresting clone had 9 nodes.

For the second study we examined all 25 clone groups identified by the tool for *bison* in the size range 30-49 (we chose an intermediate clone size in order to test the hypothesis that the uninteresting clones identified by the tool tend to be quite small). All but one of those 25 groups were variants of 9 ideal clone groups (i.e., only one of them was uninteresting).

In the two studies, we encountered a total of 11 ideal clone groups (two groups showed up in both studies) containing a total of 37 individual clones. Of those 37, 10 were non-contiguous. Two of the 11 ideal clone groups involved statement reordering, five involved variable renaming, and none involved intertwined clones.

3.2 IBM Code

The goals of the experiments using the IBM code were:

a. to see whether this code also contained non-contiguous, reordered, and intertwined clones;

b. to gather some quantitative data on the immediate effects of extracting clones.

Due to limitations of CodeSurfer, we were not able to process the entire IBM program. Therefore, we selected four (out of the 70+ files) and ran the tool on each of those files individually. The larger clones found by the tool were then examined manually (about 250 clone groups were examined), and the clones best-suited for extraction were identified. The "ideal" versions of those clones were (manually) extracted into macro definitions, which were placed in the same files, and each instance of a clone was replaced by a macro call. (Macros rather than procedures were used to avoid changing the running time of the program.) A total of 30 clone groups containing 77 "ideal" clones were extracted.

The results of the study are summarized in Figure 9, which gives, for each file:

- the size (in lines of source code and in number of PDG nodes);
- the running time for the tool (in all four cases, Step 1 of the algorithm – finding clone pairs – accounted for at least 90% of the running time);
- the number of clone groups that were extracted;
- the total number of extracted clones;
- the reduction in size of the file (in terms of lines of code);
- the average reduction in size for functions that included at least one extracted clone (in terms of lines of code).

	# of lines of source	# of PDG nodes	running time (elapsed)	# of clone groups extracted	total # of clones extracted	file size reduction	av. fn size reduction
file 1	1677	2235	1:02 min	3	6	1.9%	5.0%
file 2	2621	4006	7:49 min	12	24	4.7%	12.4%
file 3	3343	6761	5:15 min	3	7	2.1%	4.4%
file 4	3419	4845	13:00 min	12	40	4.9%	10.3%

Fig. 9. IBM file sizes and clone-extraction data

Of the 30 clone groups that were extracted, 2 involved reordering of matching statements, 2 involved intertwined clones, and most of them involved renamed variables. Of the 77 extracted clones, 17 were non-contiguous.

3.3 Summary of Experimental Results

The results of our experiments indicate that our approach is capable of finding interesting clones that would be missed by other approaches. Many of these clones are non-contiguous and involve variable renaming; some also involve statement reordering and intertwining. The Unix-code studies also indicate that the tool is not likely to miss any clones that a human would consider ideal, and additionally is not likely to produce too many clones that a human would consider uninteresting (except very small ones). The IBM-code study provides some additional data about the amount of extractable duplicated code that may be found by our tool, and how extracting that code affects file and function sizes. Of course, the more important question is how duplicate code extraction affects the ease of future maintenance; unfortunately, such a study requires resources beyond those available to us (as noted in the Introduction, the work of [13] does provide a first step in that direction).

The two sets of studies also reveal that the current implementation often finds multiple variants of ideal clones rather than just the ideal ones. This may however not be a problem in practice; manually examining the 250 clone groups reported by the tool for the four IBM files and identifying the corresponding 30 ideal clone groups took only about 3 hours. Nevertheless, future work includes devising more heuristics (like the one described in Subsection 2.3) that will reduce the number of variants reported by the tool by finding clones that are closer to ideal.

As for the running time, although the tool is currently very slow, we believe that this is more a question of its implementation than of some fundamental problem with the approach. As indicated in the table in Figure 6, the bottleneck is finding clone pairs; one reason this step is so slow is that it is implemented in Scheme, and we use a Scheme interpreter, not a compiler. Another factor is that our primary concern has been to get an initial implementation running so that we can use the results to validate our approach (rather than trying to implement the algorithm as efficiently as possible). Future engineering efforts may reduce the time significantly. Furthermore, improvements that eliminate the generation of undesirable clones (e.g., variants of ideal clones) should speed up the tool. Finally, it may be possible (and profitable) to generate clone groups directly, rather than generating clone pairs and then combining them into groups (because for each clone group that contains n clones, we currently generate $(n^2 - n)/2$ clone pairs first).

4 Related Work

The long-term goal of our research project is a tool that not only finds clones, but also automatically extracts a user-selected group of clones into a procedure. A first step in that direction was an algorithm for semantics-preserving procedure extraction [11]. However, that algorithm only applies to a single clone; different techniques are needed to determine when and how a *group* of clones can be extracted into a procedure while preserving semantics. Also, while that work was related to the work presented here in terms of our over-all goal, it addressed a very different aspect, namely, how to do procedure extraction; there was no discussion of how to identify the code to be extracted, which is the subject of the current work.

Other related work falls into 3 main categories: work on clone detection, work on converting procedural code to object-oriented code, and work on subgraph isomorphism.

Clone detection: Baker [1, 2] describes an approach that finds all pairs of matching "parameterized" code fragments. A code fragment matches another (with parameterization) if both fragments are contiguous sequences of source lines, and some global substitution of variable names and literal values applied to one of the fragments makes the two fragments identical line by line. Comments are ignored, as is whitespace within lines. Because this approach is text-based and line-based, it is sensitive to lexical aspects like the presence or absence of new lines, and the ordering of matching lines in a clone pair. Our approach does not have these shortcomings. Baker's approach does not find intertwined clones. It also does not (directly) find non-contiguous clones. A postpass can be used to group sets of matching fragments that occur close to each other in the source, but there is no guarantee that such sets belong together logically.

Kontogiannis et al [12] describe a dynamic-programming-based approach that computes and reports for every pair of `begin-end` blocks in the program the distance (i.e., degree of similarity) between the blocks. The hypothesis is that pairs with a small distance are likely to be clones caused by cut and paste activities. The distance between a pair of blocks is defined as the least costly sequence of insert, delete and edit steps required to make one block identical line-by-line to the other. This approach does not find clones in the sense of our approach, or Baker's approach. It only gives similarity measures, leaving it to the user to go through block pairs with high reported similarity and determine whether or not they are clones. Also, since it works only at the block level it can miss clone fragments that are smaller than a block, and it does not effectively deal with variable renamings or with non-contiguous or out-of-order matches.

Two other approaches that involve metrics are reported in [7, 14]. The approach of [7] computes certain features of code blocks and then uses neural networks to find similar blocks based on their features, while [14] uses function level metrics (e.g., number of lines of source, number of function calls contained, number of CFG edges, etc.) to find similar functions.

Baxter et al [4] find exact clones by finding identical abstract-syntax tree subtrees, and inexact clones by finding subtrees that are identical when variable names and literal values are ignored. Non-contiguous and out-of-order matches will not be found. This approach completely ignores variable names when asked to find inexact matches; this is a problem because ignoring variable names results in ignoring all data flows which itself could result in matches that are not meaningful computations worthy of extraction.

Debray et al [8] use the CFG to find clones in assembly-language programs for the purpose of code compression. They find matching clones only when they occur in different basic blocks, no intertwined clones, and only a limited kind of non-contiguous clones.

Converting procedural code to object-oriented code: The primary goal of the work described by Bowdidge and Griswold in [6] is to help convert procedural code to object-oriented code by identifying methods. As part of this process, they do a limited form of clone detection. Given a variable of interest, the tool does forward slicing from all uses of the variable. The slices are subsequently decomposed into a set of (overlapping) paths, with each path stretching from the "root" node (i.e., the node that has the use of the variable) to the end point of the slice. Finally the paths obtained from all slices are overlayed visually on a single diagram (only the operators of the nodes are shown) with common prefixes drawn only once. Each common prefix is a set of isomorphic paths in the PDG and therefore represents a duplicated computation; the user selects the prefixes to be extracted. There are a few significant differences between their approach and ours. They report only isomorphic paths in the PDG, whereas we report isomorphic partial slices. Our observation is that most clones that are interesting and worthy of extraction are not simply paths in the PDG. Their diagram can be very large for large programs, making it tedious for the user to figure out what clones to extract. Finally, they do only forward slicing, which in our experience is not as likely to produce meaningful clones as a combination of backward and forward slicing; for example, of all the clones found by our tool that are illustrated in this paper, only the ones in Figures 3 and 5 correspond to forward slices.

Subgraph isomorphism: A number of people have studied the problem of identifying maximal isomorphic subgraphs [3, 15, 5, 18]. Since this in general is a computationally hard problem, these approaches typically employ heuristics that seem to help especially when the graphs being analyzed are representations of molecules. In our approach we identify isomorphic partial slices, not general isomorphic subgraphs. We do this not only to reduce the computational complexity, but also because clones found this way seem more likely to be meaningful computations that are desirable as separate procedures.

5 Conclusions

We have described the design and implementation of a tool that finds duplicated code fragments in C programs and displays them to the programmer. The most innovative aspect of our work is the use of program-dependence graphs and program slicing, which allows our tool to find non-contiguous clones, intertwined clones, and clones that involve variable renaming and statement reordering.

Our implementation indicates that the approach is a good one; real code does include the kinds of clones that our tool is well-suited to handle (and that most previous approaches to clone detection would not be able to find), and the tool does find the clones that would be identified by a human. However, it currently finds many variants of the ideal clones. Future work includes developing heuristics to cut down on the number of variants identified, as well as to improve the running time of the implementation.

Acknowledgements

This work was supported in part by the National Science Foundation under grants CCR-9970707 and CCR-9987435, and by the IBM Center for Advanced Studies.

References

1. B. Baker. On finding duplication and near-duplication in large software systems. In *Proc. IEEE Working Conf. on Reverse Engineering*, pages 86–95, July 1995.
2. B. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Jrnl. on Computing*, 26(5):1343–1362, Oct. 1997.
3. H. Barrow and R. Burstall. Subgraph isomorphism, matching relational structures and maximal cliques. *Information Processing Letters*, 4(4):83–84, Jan. 1976.
4. I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Int. Conf. on Software Maintenance*, pages 368–378, 1998.
5. D. Bayada, R. Simpson, A. Johnson, and C. Laurenco. An algorithm for the multiple common subgraph problem. *Jrnl. of Chemical Information and Computer Sciences*, 32(6):680–685, Nov.–Dec. 1992.
6. R. Bowdidge and W. Griswold. Supporting the restructuring of data abstractions through manipulation of a program visualization. *ACM Trans. on Software Engineering and Methodology*, 7(2):109–157, Apr. 1998.
7. N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley. The development of a software clone detector. *Int. Jrnl. of Applied Software Technology*, 1(3-4):219–36, 1995.
8. S. Debray, W. Evans, R. Muth, and B. D. Sutter. Compiler techniques for code compaction. *ACM Trans. on Programming Languages and Systems*, 22(2):378–415, Mar. 2000.
9. J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, July 1987.
10. http://www.codesurfer.com.
11. R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 155–169, Jan. 2000.
12. K. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1–2):77–108, 1996.
13. B. Lague, D. Proulx, J. Mayrand, E. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Int. Conf. on Software Maintenance*, pages 314–321, 1997.
14. J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the Int. Conf. on Software Maintenance*, pages 244–254, 1996.
15. J. McGregor. Backtrack search algorithms and maximal common subgraph problem. *Software – Practice and Experience*, 12:23–34, 1982.
16. K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*, pages 177–184, 1984.

17. W. Stevens, G. Myers, and L. Constantine. Structured design. *IBM Systems Jrnl.*, 13(2):115–139, 1974.
18. T. Wang and J. Zhou. Emcss: A new method for maximal common substructure search. *Jrnl. of Chemical Information and Computer Sciences*, 37(5):828–834, Sept.–Oct. 1997.
19. M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, SE-10(4):352–357, July 1984.