Visualizing Code Execution

Andrew Davison
Dept. of Computer Eng.
Prince of Songkla Univ.
Hat Yai, Songkhla 90110
Thailand

E-mail: ad@fivedots.coe.psu.ac.th

14th April 2019

Abstract

The CallGraph Java class is described, using the Fibonacci function as an example of how the class can be used to visualize and clarify the execution of a program. There are also brief discussions of applying CallGraph to the n-queens problem and mergesort.

1. Introduction

The CallGraph class offers a small set of methods for augmenting Java code so that various forms of call graphs can be generated utilizing GraphViz (https://graphviz.gitlab.io) .

GraphViz is an open source graph visualization system which translates descriptions of graphs written in a text language into diagrams in multiple formats, such as images and PDF. GraphViz also has numerous options for adjusting the colors, fonts, line styles, and shapes used in a graph. Sadly, GraphViz doesn't offer a Java API, but it's possible to call its tools using Java's Runtime and Process classes. CallGraph.java is derived from code using this approach written by by Laszlo Szathmary, and available at https://github.com/jabbalaci/graphviz-java-api

A typical recursive function that confuses new programmers is the implantation of the Fibonacci equation:

```
public static void main(String args[])
{
  if (args.length != 1) {
    System.out.println("Usage: Fib <n>");
    return;
  }

  int n = 20;
  try {
    n = Integer.parseInt(args[0]);
  }
  catch (NumberFormatException e) {
    System.out.println("Not an integer; using 20");
  }

  System.out.println("Fib(" + n + ") = " + fib(n));
}  // end of main()


private static long fib(int n)
```

```
{
  if ((n == 1) || (n == 2))
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

The difficulties probably stem from the need for the student to construct an image of the execution of the function. For example, students often cannot answer questions such as "how many "n" variables are created during a call to fib()?" or "where does fib() return to when a return line is executed?" A call graph of the function's execution supplies a visualization which makes it much easier to understand and answer these questions.

Unlike some other call graph tools, a programmer using CallGraph must explicitly augment their code with calls to the class' methods. This allows the programmer to decide how much information will be displayed.

2. Using CallGraph with Fibonacci

The main() function must include a few lines that initialize CallGraph, and some code after the call to fib() which completes the graph, runs GraphViz. and saves an image to s file. main() becomes:

```
private static CallGraph cg;


public static void main(String args[])
{
  // code for reading in n, as before

  cg = new CallGraph();

  System.out.println("Fib(" + n + ") = " + fib(n,"main"));

  cg.end();
  // System.out.println(cg.getSource());
  cg.save("fib.png");
}  // end of main()
```

The commented out line that calls CallGraph.getSource() is a way to print the GraphViz text which it will process into a graph. It's a useful debugging tool when the generated graph is incorrect in some way.

fib() must now be passed the name of its calling function – "main" in this case. This name is required by CallGraph methods called in fib().

The simplest call graph is one showing function calls as nodes. It is created using CallGraph.onCall():

```
private static long fib(int n, String parNode)
```

```
// fib with calling graph
{
  String nodeNm = cg.onCall(parNode, "fib", "n:"+n);
  if ((n == 1) || (n == 2))
    return 1;
  else
    return fib(n-1, nodeNm) + fib(n-2, nodeNm);
}
```

The version of onCall() used above requires the name of the caller, the name of this function, and the third argument. a string, will appear inside the node.

When this version of the program is run, a "fib.png" image is created, which is shown in Figure 1.

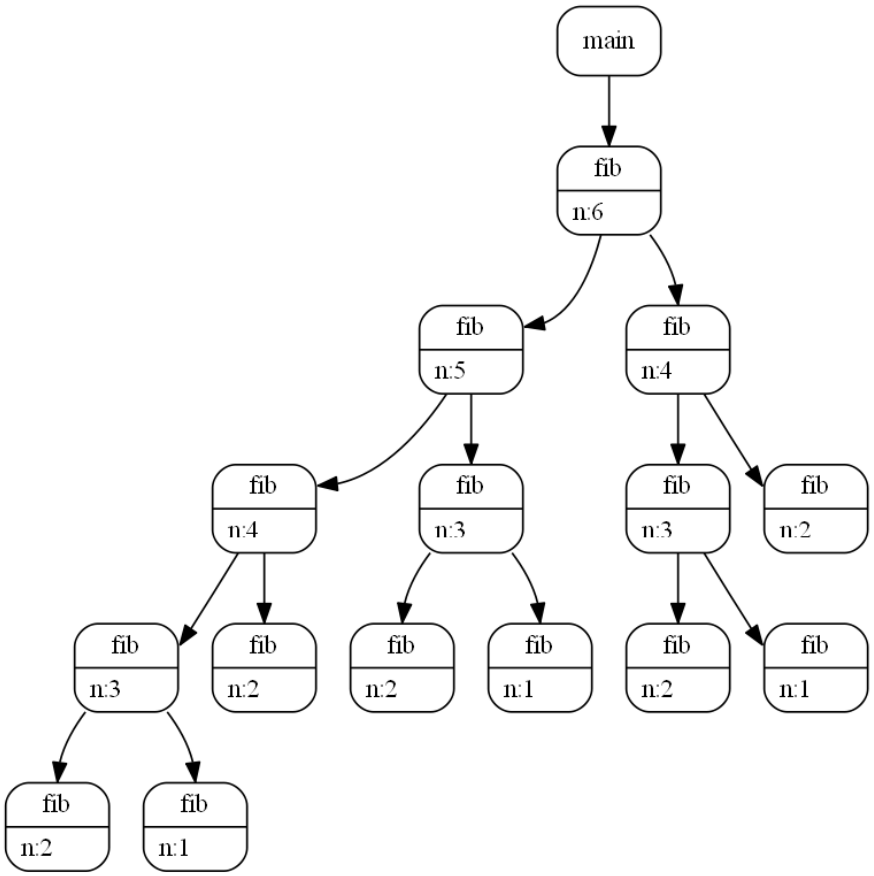

Figure 1. A CallGraph for fib(6).

In response to my earlier question about "n" variables, the graph shows that a considerable number are created – one for each fib() call.

Missing from Figure 1 is a representation of how fib() returns a result. This information can be included by adding calls to CallGraph.onReturn():

```
private static long fib(int n, String parNode)
// fib with calling and return graph
{
```

```
    String nodeNm = cg.onCall(parNode, "fib", "n:"+n);
    if ((n == 1) || (n == 2)) {
      cg.onReturn(nodeNm, "1", parNode);
      return 1;
    }
    else {
      long res = fib(n-1, nodeNm) + fib(n-2, nodeNm);
      cg.onReturn(nodeNm, ""+res, parNode);
      return res;
    }
  }
```

The version of onReturn() used above has three arguments: the function's node's name, a string that will be shown in the return node, and the parent node's name.

When Fibonacci is now run, the call graph includes orange return nodes, as in Figure 2.

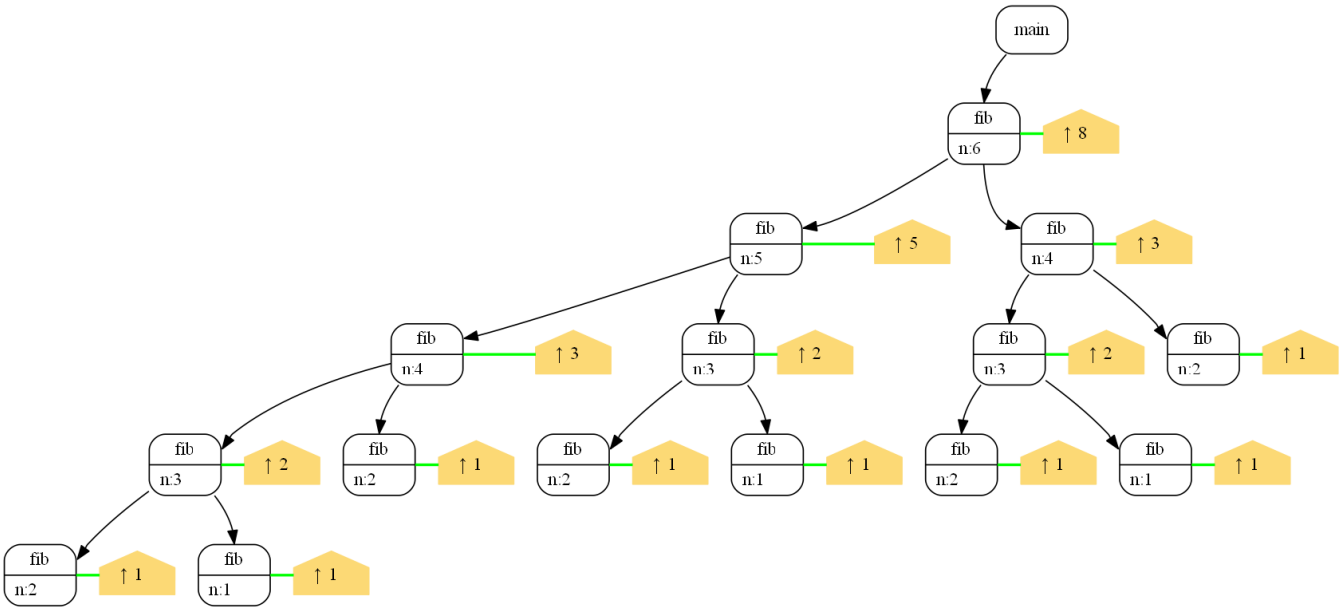

Figure 2. A CallGraph for fib(6). with Return Nodes.

This helps to answer my other question about returns in Fibonacci: each return of a function goes back to its parent, and only the very first fib() call returns to main().

3. Remembering Fibonacci

A typical next step after introducing the recursive version of Fibonacci is to use it as an example of inefficiency due to repeated work. This can be seen in Figure 1 where the entire right-hand branch call to fib(4) is a duplicate of the same sub-tree within the left branch.

The solution is to have the algorithm remember solutions so that if the same calculation reappears, then the stored result can be used instead of carrying out a possibly lengthy recalculation.

One solution involves passing a memo[] array around in the calls to fib() to store results:

```
public static long mfib(int n)
{
  long[] memo = new long[n+1];
  Arrays.fill(memo, -1);
  memo[0] = 0;
  memo[1] = 1;
  memo[2] = 1;
  return mfib(memo, n);
}


private static long mfib(long[] memo, int n)
{
  if ((n == 1) || (n == 2))
    return memo[n];
  else {
  // Compute memo[n] if not computed already
    if (memo[n] == -1)
      memo[n] = mfib(memo, n-1) + mfib(memo, n-2);
    return memo[n];
  }
}
```

It may be far from obvious to a new programmer why this code is more efficient than the plain recursive version. Generating a call graph makes things clearer.

The changes to the code to generate the graph are quite minimal:

```
// inside the first mfib() function
return mfib1(memo, n, "main");


private static long mfib(long[] memo, int n,
                                        String parNode)
{ String nodeNm = cg.onCall(parNode, "mfib", "n:"+n);
  if ((n == 1) || (n == 2))
    return memo[n];
  else {
    // Compute memo[n] if not computed already
    if (memo[n] == -1)
      memo[n] = mfib(memo, n-1, nodeNm) +
                mfib(memo, n-2, nodeNm);
    return memo[n];
  }
}
```

The resulting call graph in Figure 3 is obviously less large than the graph for the original version of fib(6).

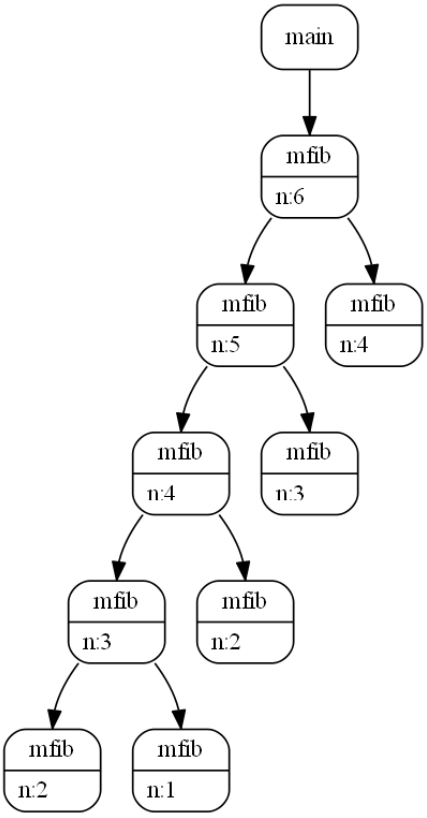

Figure 3. A Call Graph for a Memoing mfib(6).

One weakness of Figure 3 is that it doesn't indicate why the graph is smaller. This can be remedied by the inclusion of "comment" nodes to explain the execution inside the function.

The revised mfib() becomes:

```
private static long mfib(long[] memo, int n,
                                        String parNode)
{ String nodeNm = cg.onCall(parNode, "mfib", "n:"+n);

  if ((n == 1) || (n == 2)) {
    cg.comment(nodeNm, "memo["+n+"] exists",
                                      CallGraph.LIME);
    return memo[n];
  }
  else {
    // Compute memo[n] if not computed already
    if (memo[n] == -1) {
      cg.comment(nodeNm, "no memo["+n+"]",
                                     CallGraph.PINK);
      memo[n] = mfib(memo, n-1, nodeNm) +
                mfib(memo, n-2, nodeNm);
    }
    else
      cg.comment(nodeNm, "memo["+n+"] exists",
                                       CallGraph.LIME);
    return memo[n];
  }
}
```

The three calls to CallGraph.comment() are all similar. There are three arguments – the node name, the comment string, and an optional color argument which specifies the background color of the node. The resulting call graph is shown in Figure 4.

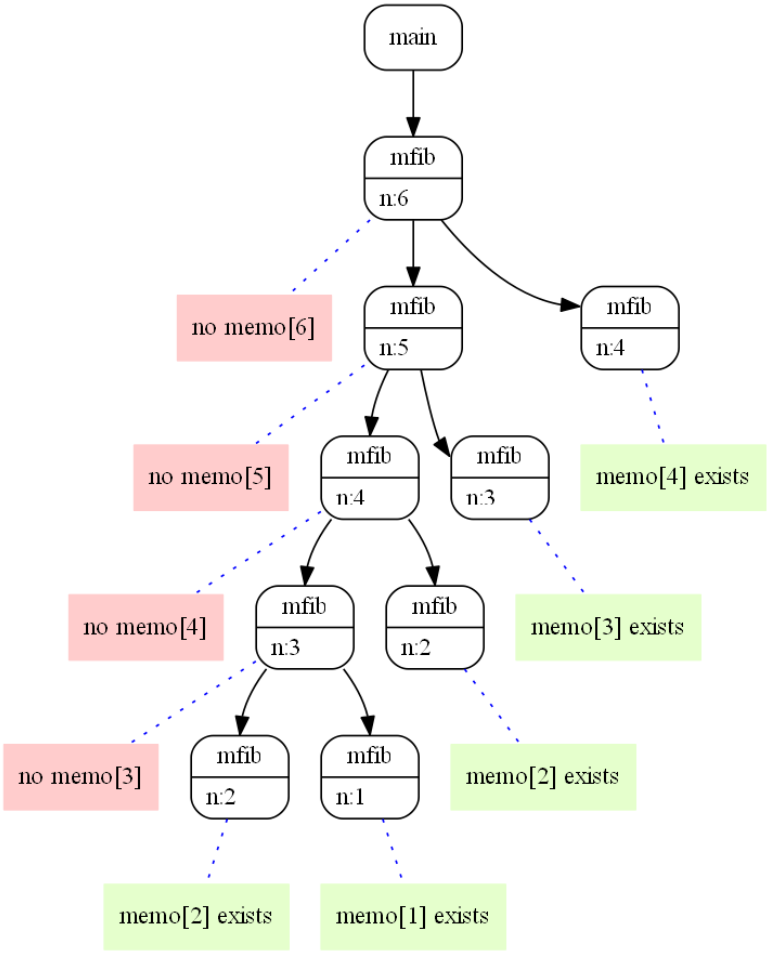

Figure 4. A Call Graph for a Memoing mfib(6) with Comments.

The two lines of comments helps to justify the argument that the running time of the program is now linear, with the memo[] array being filled by the execution along the left hand branch, and used during the evaluation of the right-hand branches to prevent recalculations.

4. The Problem of Call Graph Size

The drawback of this approach is that a generated call graph can easily become so large that it may confuse a novice programmer. As an example, consider the n-queens problem. The code is based on an example from the book (and website) "Computer Science: An Interdisciplinary Approach" by Robert Sedgewick and Kevin Wayne (https://introcs.cs.princeton.edu/java/23recursion/Queens.java).

When the program is run on a 4x4 board, only two non-attacking queen positions are generated:

```
>  java Queens 4
* Q * *
* * * Q
Q * * *
* * Q *

* * Q *
Q * * *
* * * Q
* Q * *
```

The corresponding call graph utilizes onCall(), onReturn() and comment() nodes, and so is quite large. The solution is to focus on just a significant sub-branch of the graph, such as the steps required to reach the first solution. Figure 5 shows roughly a quarter of the call graph generated by a for the 4x4 board.

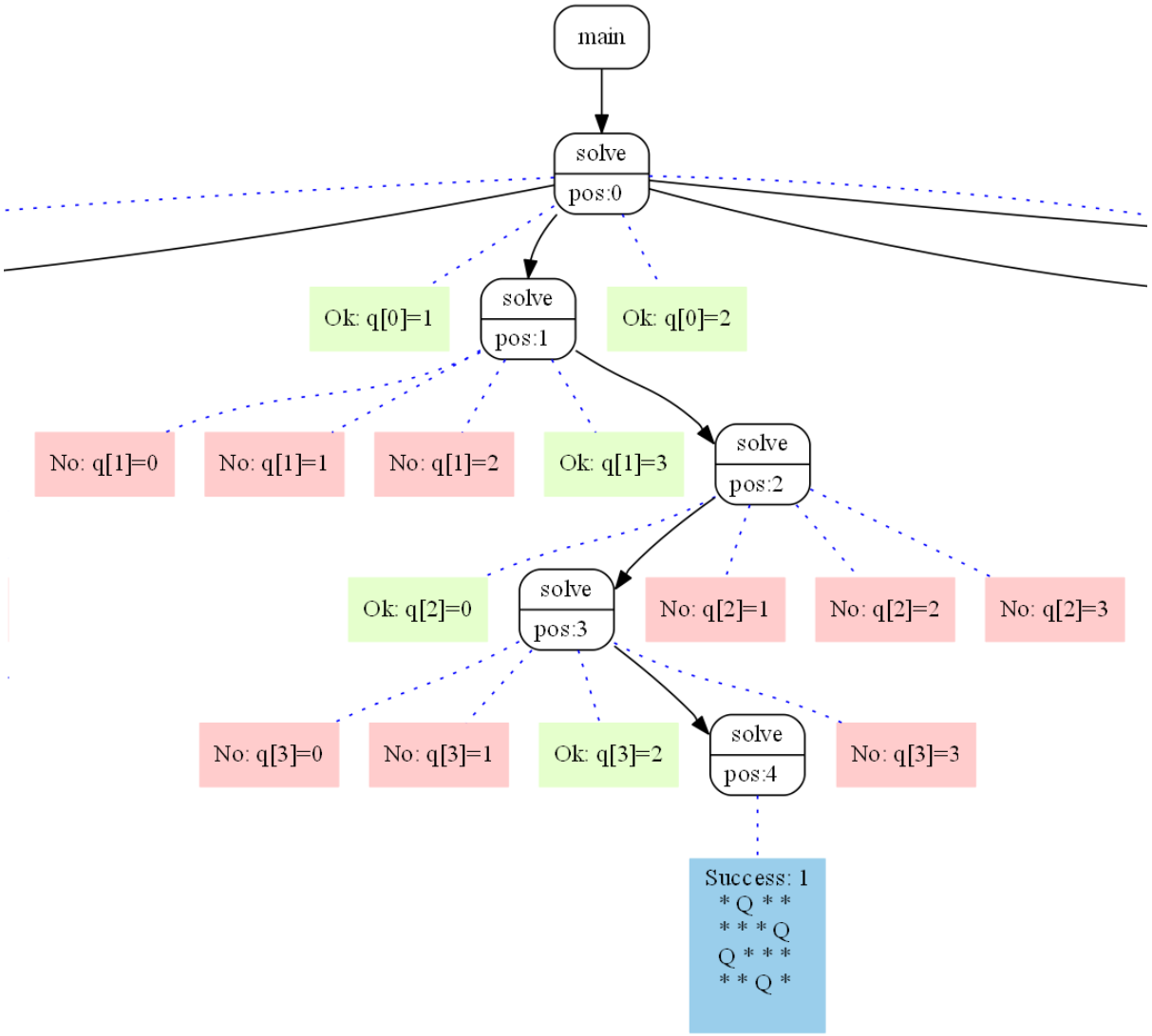

Figure 5. Part of the Call Graph for Queens 4.

The pink comments indicate failed position tests, the green ones are for successes, and the blue comment is used to display the answer. The solve() function that does all of this is:

```
public static void solve(int[] rows, int pos,
                                        String parNode)
{ String nodeNm = cg.onCall(parNode, "solve", "pos:"+pos);

  int n = rows.length;
  if (pos == n) {    // done all rows, so print
    printQueens(rows);
    numSolns++;
    String[] results = new String[]{ "Success: " +
                          numSolns, toString(rows) };
    cg.comment(nodeNm, results);
  }
  else {   // on row number: pos
    for (int i = 0; i < n; i++) {   // consider all cols
      rows[pos] = i;
      if (isConsistent(rows, pos)) {
        // okay with previous rows
        cg.comment(nodeNm, "Ok: q[" + pos + "]=" + i,
                           CallGraph.LIME);
        solve(rows, pos+1, nodeNm);   // onto next row
      }
      else
        cg.comment(nodeNm, "No: q[" + pos + "]=" + i,
                           CallGraph.PINK);
    }
  }
}  // end of solve()
```

The only new CallGraph element used in solve() is the ability to pass an array of strings as comments to CallGraph.comment(). This call also doesn't include a color argument, which causes the node to be drawn in light blue.

5. The Right Information: MergeSort

MergeSort is a tricky algorithm to explain since the incoming array isn't 'divided' into pieces explicitly, but is divided based on left, right, and mid index variables. These variables quickly multiply as mergesort calls itself, and so drawing its call graph is both a tiresome and tricky undertaking. The generation of the graph by code is much less work.

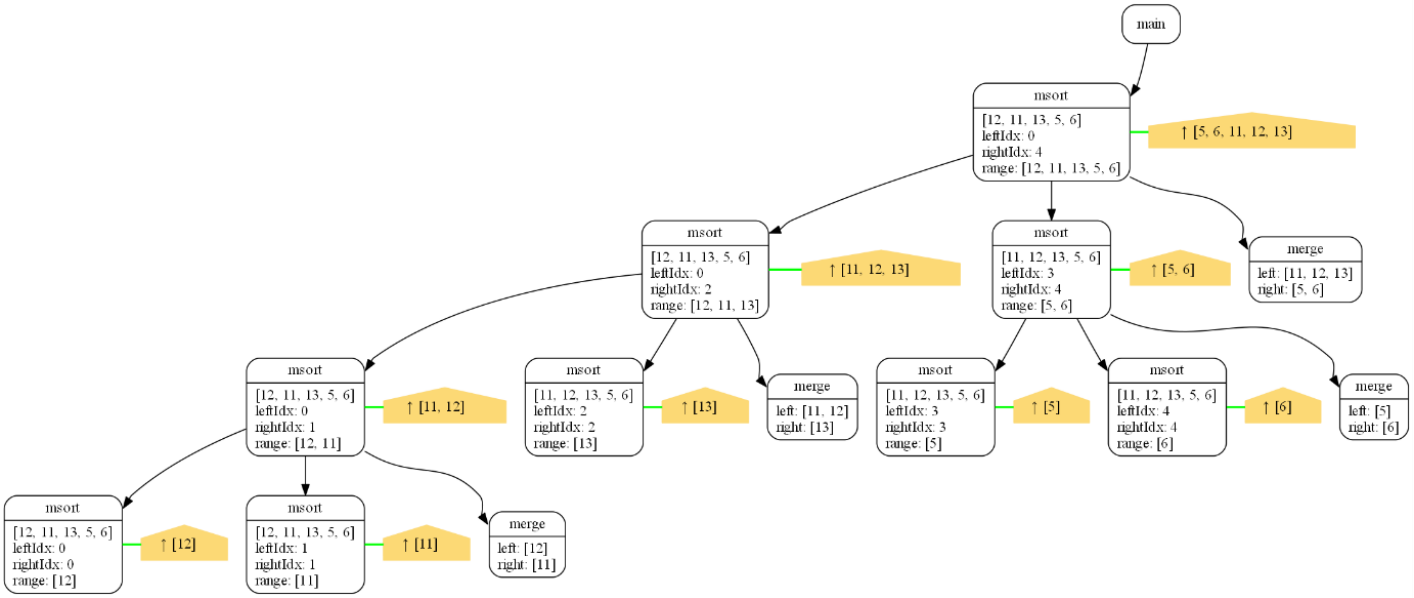

Figure 6. MergeSort of [12, 11, 13, 5, 6].

The graph in Figure 6 is so large because each "msort" call node includes four lines of information: the input array, the left and right indicies, and the range of the array being considered in this call. Figure 7 shows a close up of the nodes at level 2 of the Figure 6 graph.

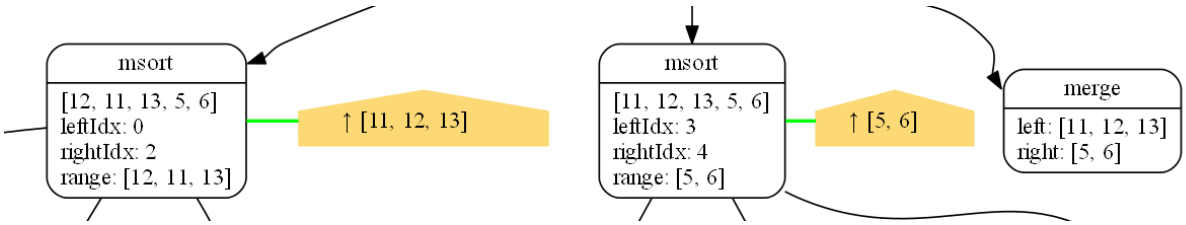

Figure 7. Nodes at Level 2 in Figure 6.

Mergesort splits into two recursive calls (shown in Figure 6), and a call to merge() at the end. The node details show that the input array is unchanged, but the indicies define different sub-ranges in the two calls. The return nodes show how those sub-ranges are sorted, and then passed to the merge() call for merging.

The code for generating the msort() nodes are in the mergeSort() function:

```
  private static void mergeSort(int arr[],
                                int leftIdx, int rightIdx,
                                String parNode)
  {
    int[] subArr = getSubRange(arr, leftIdx, rightIdx+1);
    String[] args = new String[]{ Arrays.toString(arr),
                    "leftIdx: "+leftIdx,
                    "rightIdx: "+rightIdx,
                    "range: " + Arrays.toString(subArr)};
    String nodeNm = cg.onCall(parNode, "msort", args);
```

```
    if (leftIdx < rightIdx) {
      // Find the middle point
      int midIdx = (leftIdx+rightIdx)/2;

      // Sort first and second halves
      mergeSort(arr, leftIdx, midIdx, nodeNm);
      mergeSort(arr , midIdx+1, rightIdx, nodeNm);

      // Merge the sorted halves
      merge(arr, leftIdx, midIdx, rightIdx, nodeNm);

      subArr = getSubRange(arr, leftIdx, rightIdx+1);
      // sorted subarray
    }
    cg.onReturn(nodeNm, Arrays.toString(subArr), parNode);
  } // end of mergeSort()
```

The call to CallGraph.onCall() includes an array of 4 strings, which creates the four lines of data inside the "msort" node.

6. Conclusions

Based purely on anecdotal evidence from using these call graphs during teaching, this way of visualizing complex function calls (often involving recursion) is of great help to new programmers. The ability to fine-tune the types of information shown (calls, returns, and comments), and the amount of information in each node, makes the CallGraph class a useful tool.

CallGraph, along with numerous examples, can be found online at
`http://fivedots.coe.psu.ac.th/~ad/callgraph`