

TR-CS-00-01

**Java Programs do not have Bounded
Treewidth**

**Jens Gustedt, Ole. A. Maehle, and Jan Arne
Telle**

February 2000

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical.Reports@cs.anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

<http://cs.anu.edu.au/techreports/>

Recent reports in this series:

- TR-CS-99-02 Samuel Taylor. *A distributed visualisation tool for digital terrain models*. 1999.
- TR-CS-99-01 Peter E. Strazdins. *A dense complex symmetric indefinite solver for the Fujitsu AP3000*. May 1999.
- TR-CS-98-14 Michael Stewart. *A completely rank revealing quotient uv decomposition*. December 1998.
- TR-CS-98-13 Michael Stewart. *Finding near rank deficiency in matrix products*. December 1998.
- TR-CS-98-12 Vadim Olshevsky and Michael Stewart. *Stable factorization of Hankel and Hankel-like matrices*. December 1998.
- TR-CS-98-11 Michael Stewart. *An error analysis of a unitary Hessenberg QR algorithm*. December 1998.

Java Programs do not have Bounded Treewidth

Jens Gustedt¹, Ole A. Mæhle², and Jan Arne Telle^{2*}

¹ LORIA & INRIA Lorraine, campus scientifique, BP 239,
54506 Vandœuvre lès Nancy, France. Email: `Jens.Gustedt@loria.fr`

² University of Bergen, Department of Informatics, HIB, 5020 Bergen, Norway.
Email: `{olem|telle}@ii.uib.no`

Abstract. We show that the control-flow graphs of Java programs, due to the labelled break and continue statements, have no upper bound on their treewidth. A single Java method containing k labels and a loop nesting depth of $k+1$ can give a control-flow-graph with treewidth $2k+1$.

1 Background

Most structured language constructs such as while-loops, for-loops and if-then-else allow programs to be recursively decomposed into basic blocks with a single entry and exit point [1]. Such a decomposition corresponds to a series-parallel decomposition of the control-flow-graph of the program [7] and can ease static optimization tasks like register allocation [4]. On the other hand, with constructs such as the infamous goto, and also short-circuit evaluation of boolean expressions and multiple exit/break/continue/return-statements, this nice decomposition structure is ruined [4].

However, Thorup has shown in a recent article 'All Structured Programs have Small Tree-Width and Good Register Allocation', see [9], that except for the goto, the other constructs listed above do allow for a related decomposition of the control-flow-graph of the program. For each of those language constructs, it was basically shown that regardless of how often they are used, they will only increase the treewidth of the control-flow graph by one. Since a series-parallel graph has treewidth 2, this means that the control-flow-graphs of goto-free Algol and Pascal programs have treewidth ≤ 3 (add one for short-circuit evaluation), whereas goto-free C programs

* Research completed while a Visiting Fellow at Australian National University.

have treewidth ≤ 6 (add also for multiple exits and continues from loops and multiple returns from functions). Moreover, the related tree-decomposition is easily found while parsing the program, and this structural information can then, as with series-parallel graphs, be used to improve on the quality of the compiler optimization [9, 2].

With unrestricted use of `gotos` it is easy to write a program whose control-flow graph has treewidth greater than any given integer k . Since the treewidth parameter is a measure of the 'treeness' of a graph [8] and a constant bound on this parameter allows many otherwise NP-hard problems to be solved in linear time [5], these results seem to imply that `gotos` are harmful for static analysis tasks. `Gotos` were originally considered harmful for readability and understanding of programs [3], and languages like Modula-2 and Java have indeed banned their use. Modula-2 instead provides the programmer with multiple exits from loops and multiple returns from functions with the pleasant consequence that all control-flow-graphs of Modula-2 programs have treewidth ≤ 5 [9].

As compensation for the lack of a `goto`, the designers of Java decided to add the *labelled* break and continue statements. This allows labelling of loops and subsequent jumping out to any prelabelled level of a nested loop. In the original 'Go To Statement Considered Harmful'-article [3], what was in fact specifically objected to was the proliferation of labels that indicate the target of `gotos`, rather than the `gotos` themselves. The accuracy of this observation is confirmed in the next section, where we show that, using only k labels, we can construct a Java program whose control-flow graph has treewidth $\geq 2k + 1$. Not only is the treewidth high, it is also clear that the programming technique of the example can be misused to construct Java code which is arbitrarily challenging to understand.

2 The main result

We will view the edges of the control-flow-graph as being undirected. Contracting an edge uv of a graph simply means deleting the endpoints u and v from the graph and introducing a new node whose neighbors are the union of the neighbors of u and v . A graph containing a subgraph that can be contracted to a complete graph on k

nodes is said to have a clique minor of size k , and is well-known to have treewidth at least $k - 1$ [8].

The labelled break and continue statements in Java allows the programmer to label a loop and then make a jump from a loop nested inside the labelled loop. In the case of a continue the jump is made to the beginning of the labelled loop, and in the case of a break the jump is made to the statement following the labelled loop. In the right-hand side of Figure 1 we show a listing of part of a Java program, with labels l1, l2 and l3, whose control-flow-graph can be contracted to a clique on 8 nodes.

```

continue1    l1:while (maybe) {
continue2    l2:  while (maybe) {break l1;
continue3    l3:    while (maybe) {break l1; break l2; continue l1;
innerloop                while (maybe) {break l1; break l2; break l3;
innerloop                                continue l1; continue l2; }
remainder3                break l1; break l2; break l3; continue l1; continue l2;}
break3                break l1; break l2; continue l1;}
break2                break l1;}
break1

```

Fig. 1. Skeleton of a Java program whose control-flow graph has treewidth ≥ 7 . Break and continue statements should be conditional, but for the sake of simplicity this has been left out. The left column, in bold font, gives the names of contracted nodes of the control-flow-graph.

For simplicity we have chosen this code fragment that is obviously not real-life code, though it could easily be augmented to become more natural. For example, breaks and continues could be case statements of a switch, such as

```

l3:    while (maybe) {
        switch(num) {
        case 1: break l1;
        case 2: break l2;
        case 3: continue l1;}

```

Each of the 8 contracted nodes will naturally correspond to some lines of the corresponding Java program. Each of the 3 first lines of

the listed code correspond to a node called, respectively, *continue1*, *continue2* and *continue3*, since they form the targets of the respective continue statements labelled l1, l2 and l3. The 4th and 5th lines of the code together form a node that we call *innerloop*, whereas the 6th line we call *remainder3* as it forms the remainder of the loop labelled l3. Lines 7 and 8 of the listing correspond to nodes that we call *break3* and *break2*, respectively, as they form the target of the break statements with labels l3 and l2. The target of the break labelled l1 is whatever statement that follows the listed code and it will be called *break1*, forming the eighth node. See Figure 2 for a drawing of the (directed) control-flow graph. It should be clear that each of

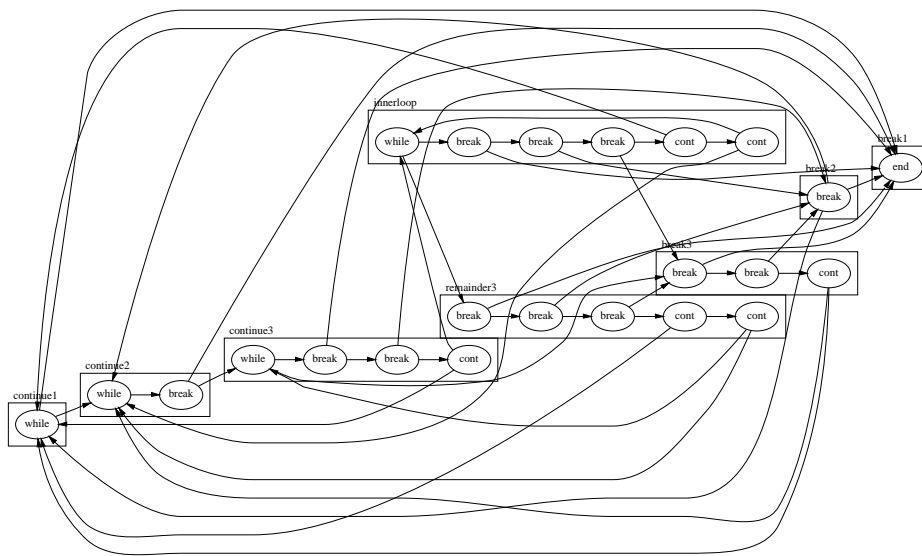


Fig. 2. The control flow graph of the example

these 8 nodes are obtained by contracting a connected subgraph of the control-flow-graph of the program. We now show that they form a clique after contraction, by looking at them in the order *innerloop*, *remainder3*, *continue3*, *break3*, *continue2*, *break2*, *continue1*, *break1* and arguing that each of them is connected to all the ones following it in the given order. Firstly, the node *innerloop* is connected to all

the other nodes, as the control flows from it into *remainder3* when its loop entry condition evaluates to false, control flows naturally into *innerloop* from *continue3* and for each of the other 5 nodes *innerloop* contains the labelled break or continue statement targeting that node. Next, *remainder3* is connected to *continue3* as this is the natural flow of control, and *remainder3* contains the labelled break or continue statement targeting each of the other 5 nodes following it in the given order. The argument for the remaining nodes follows a similar line of reasoning. Moreover, in the same style a larger code example can be made consisting of a method with k labels, a loop nesting depth of $k + 1$ and a clique minor of size $2k + 2$.

Theorem 1 *For any value of $k \geq 0$ there exists a Java method with k labels and nesting depth $k + 1$ whose control-flow-graph has treewidth $\geq 2k + 1$.*

Proof. For a proof consider the following extract of a 'symbolic' Java program using k labels with nesting depth $k + 1$, where the first line with ... should expand to $k - 4$ labelled loops and the second line with ... should expand to $k - 4$ lines forming the continuation of their respectively nested labelled loops.

```

l1: while (maybe) {
l2:   while (maybe) {break l1;
l3:     while (maybe) {break l1; break l2; continue l1;
    ...
lk:     while (maybe) {break l1; ... break lk-1; continue l1; ... continue lk-2;
        while (maybe) {break l1; ... break lk; continue l1; ... continue lk-1; }
        break l1;...;break lk; continue l1;... continue lk-1;}
        break l1; ... break lk-1; continue l1; ... continue lk-2; }
    ...
    break l1; break l2; continue l1;}
break l1;}

```

Here 'symbolic labels' like lk is used to ease the induction in the proof, but in a real Java program of course k must be a constant. Moreover, all the breaks and continues on a single line should be the cases of a switch statement as explained earlier.

Each line of the code should be contracted to a single node in the control-flow graph. With the additional node for the statement following the listed code this gives $2k + 2$ nodes total. The proof that the contracted graph on $2k + 2$ nodes forms a clique follows the same line of reasoning as in the example given earlier. A graph with a clique minor of size $2k + 2$ has treewidth $\geq 2k + 1$.

3 Conclusion

Originally Java was designed to be precompiled to bytecode for the *Java Virtual Machine*, so compiler optimization tasks were then not a main issue. Nevertheless, `gotos` were considered particularly harmful for the conceptual clarity of a program and so they were completely banned from the specification of Java, and a labelled `break` and `continue` were instead added. Nowadays, to speed up applications written in Java there is a strong demand for compiled *and* optimized Java, and so Java-to-native-machine-code compilers are emerging. In this paper we have shown that such compilers must have certain limits that are already inherent in the language itself.

In an ongoing study [6] the treewidth of actual Java programs will be empirically tested. For Java programs not having labelled breaks or continues the results of Thorup [9] will hold and we can find the exact value of the treewidth. For programs containing labelled breaks and continues we can easily give an upper bound on the treewidth, since the examples given here are the worst case for few labels, but we would like to also compute the exact treewidth of these programs and to further study the practical import of the observations in this paper.

References

1. A.V.Aho, R.Sethi and J.D.Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
2. H. Bodlaender, J. Gustedt and J.A. Telle, Linear-time register allocation for a fixed number of registers and no stack variables, *Proceedings 9th ACM-SIAM Symposium on Discrete Algorithms (SODA'98)*, pp. 574-583.
3. E.W.Dijkstra, Go To statement considered harmful, *Comm. ACM* 11, 3 (1968) pp.147-148.
4. S.Kannan and T.Proebsting, Register allocation in structured programs, in *Proceedings 6th ACM-SIAM Symposium on Discrete Algorithms (SODA'95)*, pp. 360-368.
5. J.van Leeuwen, Graph Algorithms - Classes of graphs, in *Handbook of Theoretical Computer Science vol. A*, Elsevier, Amsterdam, 545-551, 1990.
6. O.A.Mæhle, forthcoming cand.scient thesis at Department of Informatics, University of Bergen.
7. T.Nishizeki, K.Takamizawa and N.Saito, Algorithms for detecting series-parallel graphs and D-charts, *Trans. Inst. Elect. Commun. Japan* 59, 3 (a976) pp.259-260.
8. N. Robertson and P. Seymour, Graph minors. II. Algorithmic aspects of tree-width, *Journal of Algorithms*, 7(3):309-322, September 1986.
9. M. Thorup, All Structured Programs have Small Tree-Width and Good Register Allocation, *Information and Computation*, Volume 142, Number 2, May 1, 1998.