All Structured Programs have Small Tree-Width and Good Register Allocation

Mikkel Thorup*
University of Copenhagen

Abstract

The register allocation problem for an imperative program is often modelled as the coloring problem of the interference graph of the control-flow graph of the program. The interference graph of a flow graph G is the intersection graph of some connected subgraphs of G. These connected subgraphs represent the lives, or life times, of variables, so the coloring problem models that two variables with overlapping life times should be in different registers. For general programs with unrestricted gotos, the interference graph can be any graph, and hence we cannot in general color within a factor $O(n^\varepsilon)$ from optimality unless NP=P.

It is shown that if a graph has tree-width k, we can efficiently color any intersection graph of connected subgraphs within a factor $(\lfloor k/2 \rfloor + 1)$ from optimality. Moreover, it is shown that structured ($\equiv$ goto-free) programs, including, for example, short circuit evaluations and multiple exits from loops, have tree-width at most 6. Thus, for every structured program, we can do register allocation efficiently within a factor 4 from optimality, regardless of how many registers are needed.

The bounded tree-width of control-flow graphs of structured programs implies that the many techniques for bounded tree-width may now be applied in control-flow analysis, solving problems in linear time that are NP-hard, or even P-space hard, for general graphs.

1 Introduction

The *register allocation problem* for an imperative program P is usually modelled as the coloring problem of the interference graph I of the control-flow graph G of P [2, 15, 16, 20]. The *control-flow graph* G is a digraph representing the flow of control between program points in the execution of the program P (see Figure 2.2, page 7). The orientation is, however, unimportant in our context, and we will hence perceive G as undirected. The *life time of a variable* is a connected subgraph of G, and the *interference graph* I is the intersection graph of the set X of all life times of variables in the program P. Thus adjacency in I denotes intersecting life times of variables. Consequently, our problem of coloring the interference graph I models that two variables with overlapping life times should be in different registers.

It should be noted that even with a good coloring, we may still run short of physical registers, in which case we have an additional *spilling problem* of simulating desired registers with the physical registers by copying to and from memory locations. The coloring then limits the amount of memory locations needed. The spilling problem is not addressed in this paper.

For general programs with unrestricted gotos, the control-flow graph can be any graph, and so can the interference graph. Hence for some fixed $\varepsilon > 0$, we cannot in polynomial time color

*Department of Computer Science, University of Copenhagen, Universitetsparken 1, 2100 Kbh. Ø, Denmark. (*mthorup@diku.dk, http://www.diku.dk/~mthorup*).

within a factor $O(n^{\varepsilon})$ from optimality unless NP=P [29]. The current best approximation factor is $O(n(\log\log n)/(\log n)^3)$ [25]. However, in this paper we show that for structured ($\equiv$ goto-free) programs[1], including, for example, short circuit evaluations and multiple exits from loops, we can do register allocation in polynomial time within a factor 4 from optimality.

Recently Kannan and Proebsting [26] showed that if the control-flow graph of a program is series-parallel, we can color the interference graph within a factor 2 from optimality. The relevance of series-parallelism follows from the well-known fact (see e.g. [2]) that many of the structured language constructs, such as if-then-else, and while-loops, allows programs to be recursively sub-divided into basic blocks with a single entry and a single exit point. Such a recursive sub-division immediately gives a series-parallel decomposition of the flow-graph (see e.g. [32]). However, even within structured languages, there are well-known exceptions to the sub-division into basic-blocks/series-parallelism. For example [26] points to short circuit evaluation where the evaluation of a boolean expression is terminated as soon as the correct answer is found, e.g. if $e = x_1 \vee \cdots \vee x_n$ is true, then e is only evaluated till the first true x_i is found. Other exceptions include loops with multiple exits/breaks and programs/functions with multiple stop-/return-statements. In [26] the problem of dealing with these exceptions is suggested. So far, however, there has been no approach suggesting that the control-flow graphs of general structured programs should be any simpler than general graphs. The concept of reducibility [2, pp. 606–607] of control-flow graphs is associated with structured programs, but reducibility only refers to the orientation of the edges (any acyclic orientation is reducible). In our case, we are only interested in the underlying undirected graph, so the requirement of reducibility does not limit the class of graphs considered.

Our key to dealing with general structured programs is to introduce the following measure:

Definition 1 *Given a graph G, a $(\leq k)$-*complex listing *is a listing of the vertices of G such that for every vertex $v \in V$, there is a set S_v of at most k of the vertices preceding v in the listing, whose deletion from G separates v in G from all the vertices preceding v in the listing. In this case, we say that G is $(\leq k)$-*complex*. The set S_v is referred to as the* separator *of v in the listing.*

Note above that for a given listing and a vertex v, there is a unique minimal choice of the separator S_v; namely as the set of preceding vertices that can be reached by a path from v with no interior vertices preceding v in the listing.

In [26], Kannan and Proebsting implicitly show that series-parallel graphs are 2-complex. In fact this is the essential property of series-parallel graphs that they use in their algorithms. In this paper, we generalize their techniques to show

Theorem 2 *Given a k-complex listing of the vertices of a graph G, we can efficiently color the intersection graph I of any set X of connected subgraphs of G within a factor $(\lfloor k/2 \rfloor + 1)$ from optimality. If n is the number of nodes in G and ω is the maximal number of subgraphs from X intersecting a single vertex in G, the coloring is done in time $O(k\omega n + \omega^{2.5} n)$. Also, we can color I within a factor $(k+1)$ from optimality in time $O(k\omega n)$.*

Note that for $k = 2$, we get the factor 2 from [26]. Also note that the 1-complex graphs are the forests for which we get a factor 1. Hence follows the colorability of chordal graphs [22, 23].

We show the significance of Theorem 2 by showing

Theorem 3 *Assuming short circuit evaluation,*

- *Goto-free Algol [30] and Pascal [37] programs have (≤ 3)-complex control-flow graphs.*
- *All Modula-2 [38] programs have (≤ 5)-complex control-flow graphs.*

[1]Structured programs is not an well-defined agreed-upon term (see e.g. [18, 19, 28, 30, 31]). In this paper, we are referring specifically to the aspect of being goto-free.

- *Goto-free C [27] programs have (≤ 6)-complex control-flow graphs.*

Without short circuit evaluation, each of the above complexities drops by one[2]*. Control-flow graphs with listings of the above complexities are derived directly (linear time, small constants) from the parse trees of the programs.*

The reason for the gap between Algol/Pascal and Modula-2 is that Modula-2 has loops with multiple exits and multiple returns from functions. The further gap to C is due to C's continue-statement jumping to the beginning of a loop. Combining Theorems 2 and 3, we get

Corollary 4 *In time $O(\omega n + \omega^{2.5} n)$, the register allocation problem can be solved within the following factors from optimality:*

- *2 (2) for Algol/Pascal,*
- *3 (3) for Modula-2, and*
- *4 (3) for C.*

The parenthesized numbers are without short circuit evaluation. If we only want to spend time $O(\omega n)$, we can get the factors: 4 (3) for Algol/Pascal, 6 (5) for Modula-2, and 7 (6) for C.

Finally, we show that k-complexity is a short new definition of tree-width k defined in [34].

Theorem 5 *A graph is k-complex if and only if it has tree-width k. Moreover there are linear transformations between k-complex listings and tree-decompositions of width k.*

It should be noted that our work on the register allocation problem does not follow the usual pattern of deriving fast algorithms for graphs of bounded tree-width. First of all we are studying intersection graphs of connected subgraphs rather than the graph itself. Second, the coloring problem we consider is NP-complete for any $k > 1$. The NP-completeness follows from the fact that a cycle has tree-width 2, and for a cycle the coloring problem for the intersection graph is known to be NP-complete [21].

The bounded tree-width of structured programs implies that the many techniques for bounded tree-width may now be applied in control-flow analysis, solving problems in linear time that are NP-hard [4, 5, 11, 17], or even P-space hard [9], for general graphs.

The paper is divided as follows. Section 2 addresses Theorem 3. Section 3 shows how simple listings may be preserved under the standard optimizations from [2]. In Section 4, we present an efficient algorithm computing the minimum separators of a listing. Section 5 proves Theorem 2. Section 6 presents the proof of Theorem 5. In Appendix A, we discuss a linear time heuristic for finding a good tree-decomposition directly from the three-address code generated, as in [2], from a structured program. This algorithm makes it easier to integrate our approach with compilers where the structural statements of a program have been replaced by `goto`s by the time of register allocation. Here, by *structural statements*, we mean statements that affect the flow of control, e.g. conditional statements, loops, and exits. For basic definitions for programs, grammars, and control-flow graphs, the reader is referred to [2].

[2]Standard Pascal and Algol does not have short circuit evaluation while standard Modula-2 and C does have short circuit evaluation.

2 Simplicity of structured programs

In this section, we will address Theorem 3. Our first tool, is the following simple lemma:

Lemma 6 *From a ($\leq k$)-complex listing L of a graph G, we can derive a ($\leq k$)-complex listing of G with an edge $\{v, w\}$ contracted.*

Proof: By symmetry, we may assume v comes before w. Then, we contract the edge $\{v, w\}$ identifying both end-points with v, and deleting w from L. ■

We will argue the correctness of Theorem 3 by first studying a Modula-2 [38] inspired toy-language STRUCTURED, illustrating all the essential problems of finding good listing of control-flow graphs for structured programs. Afterwards we will outline how the complexity of STRUCTURED carries over to `goto`-free Pascal, Modula-2, and `goto`-free C.

A program in STRUCTURED starts with the key word `program`, then comes the statement of the program, and finally comes the key word `margorp`. Of statements, we have sequences of statements separated by `;`, the conditional statements: `if-then-else-fi` and `if-then-else-fi`, a general loop structure `loop-pool`, an `exit`-statement terminating the nearest surrounding loop, and a `stop`-statement ending the execution of a program. If an `exit`-statement is not surrounded by a loop, we view it as a `stop`-statement. Finally, we have "atomic" statements consisting of single lower-case letters `s`, `t`,... These should be thought of as representing statements such as assignments that do not affect the flow of control of the program, hence which are irrelevant to finding a good listing. We are assuming the standard that control-flow graphs are generated separately for the main program and for procedures, i.e. we are not doing inter-procedural analysis. Hence our atomic statements may also represent procedure calls.

For boolean expressions, we have the connectives `or`, and `and`, both of which are evaluated with short circuit evaluation. Also, we have "atomic" boolean expressions represented by single lower-case letters `a`, `b`,... These should be thought of as representing constants or program variables. In Figure 2.1 are given some examples of control-flow graphs for fragments of programs written in STRUCTURED with an 'in' and an 'out' node showing the entry and exit points. In (a1) and (b1) we have been very liberal in the use of vertices, letting every single word in the program constitute its own vertex. The indices describe a 3-complex listings of the vertices. In (a2) and (b2) we have been more conservative in the use of vertices, contracting some of the edges from (a1) and (b1). However, using Lemma 6, we have inherited a 3-complex listings of the vertices. Thus, because of Lemma 6, without loss of generality, we can restrict our attention to control-flow graphs with one vertex for each word in the program.

We will now describe the order in which to visit the words in a STRUCTURED program so that it corresponds to a (≤ 5)-complex listing. Generally we just follow the structure of the program in a top-down fashion, visiting all key words of a composite structure before we descend into its different parts. Thus, for a program `program` S `margorp`, we first visit the two key words `program` and `margorp`, and then recursively, we visit the contents of the statement S.

For a composite statement of the form

S`;`S', we first visit `;`, and then we visit S and S' recursively.

`loop` S `pool`, we first visit `loop` and `pool`, and then we visit S recursively.

`if` B `then` S `else` S' `fi`, we first visit `if`, `fi`, `then`, and `else`, and then we visit B, S, and S' recursively.

`if` B `then` S `fi`, we first visit `if`, `fi`, and `then`, and then we visit B and S recursively.

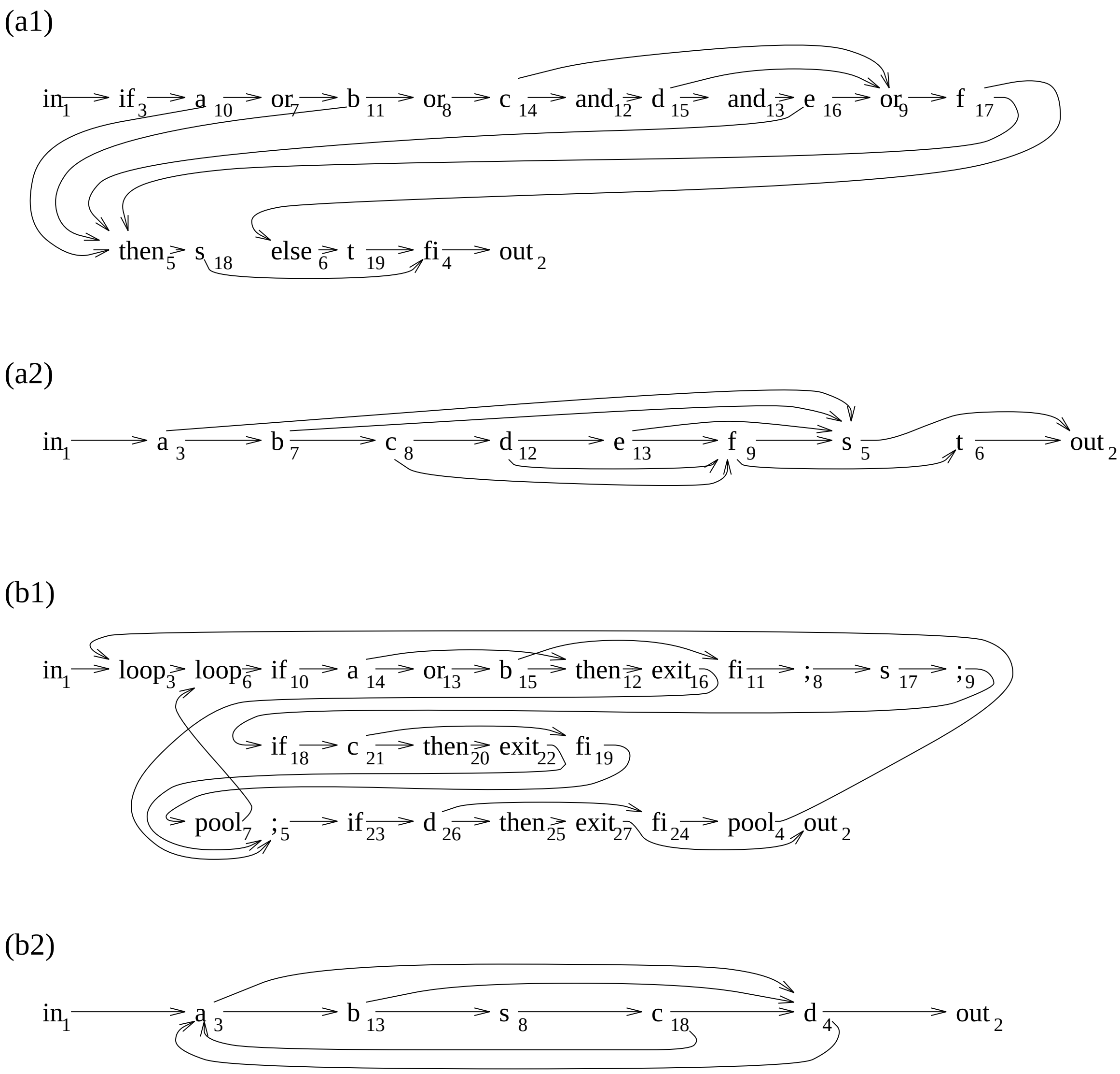

Figure 2.1: Some control-flow graphs

For a composite boolean expression of the form

$B\,\texttt{or}\,B'$, we first visit `or`, and then we visit B, and B' recursively.

$B\,\texttt{and}\,B'$, we first visit `and`, and then we visit B, and B' recursively.

An example of the described visit sequence is given in Figure 2.2.

Theorem 7 *All control-flow graphs derived from STRUCTURED are* (≤ 5)*-complex.*

Proof: We prove the theorem by showing that the above described visit sequence always corresponds to a (≤ 5)-complex listing.

Note that for a statements S, the sub-graph of the control flow graph corresponding to S has only the following potential neighbors:

in the word preceeding S.

out the word succeding S.

$exit$ the word succeding the nearest surrounding loop, or `margorp` if there is no surrounding loop.

$stop$ the end of a program `margorp`.

Similarly for a boolean expression B, the potential neighbors of the sub-graph corresponding to B are

in the word preceeding B.

$true$ the word we jump to if B evaluates to true.

$false$ the word we jump to if B evaluates to false.

Following the recursive structure of our visiting sequence, we will now show both (1) that the all the neighbors of a statement S or boolean expression B are visited before we start visiting words in S or B, and (2) that all words get a separator of size at most 5.

Given a program $\texttt{program}\,S\,\texttt{margorp}$, we first visit `program` and `margorp` that get separators $\emptyset$ and $\{\texttt{margorp}\}$, respectively. The neighbors of S are then $in = \texttt{program}$ and $out = exit = stop = \texttt{margorp}$, all of which have been visited before we start visiting S.

For a statement S of the form $S'\,;\,S''$, we first visit ;. Since none of the words in S' or S'' have been visited, ; gets separator $\{in, out, exit, stop\}$. Now ; separates S' from out and S'' from in. Hence the neighbors of S' and S'' are the same as those of S except that $out' = in'' =$;. Since all the neighbors of S where visited before we started visiting S, it follows that the neighbors of S' and S'' are visited before we start visiting S' and S''.

Now consider the case of a loop statement $S = \texttt{loop}\,S'\,\texttt{pool}$, where first we visit `loop` and `pool`. Then $\{in, out, exit, stop\}$ is a separator for `loop` and $\{\texttt{loop}, out, exit, stop\}$ is a separator for `pool`. For S' we then get the neighbors $in' = \texttt{loop}$, $out' = \texttt{pool}$, $exit' = out$, and $stop' = stop$.

Next consider a conditional statement $S = \texttt{if}\,B'\,\texttt{then}\,S''\,\texttt{else}\,S'''\,\texttt{fi}$. Recall that we first visit `if`, `fi`, `then`, and `else`, and then we visit B', S'', and S''' recursively. Thus `if` gets the separator $\{in, out, exit, stop\}$, `fi` gets the separator $\{\texttt{if}, out, exit, stop\}$, `then` gets the separator $\{\texttt{if}, \texttt{fi}, exit, stop\}$, and `else` gets the separator $\{\texttt{if}, \texttt{fi}, \texttt{then}, exit, stop\}$. For B' we get the neighbors $in' = \texttt{fi}$, $true' = \texttt{then}$, and $false' = \texttt{else}$. For S'', we get the neighbors $in'' = \texttt{then}$, $out'' = \texttt{fi}$, $exit'' = exit$, and $stop'' = stop$. The neighbors of S''' are the same as those of S'' except that $in''' = \texttt{else}$. The `if-then-fi` version is done similarly.

Concerning statements consisting of a single word, the separator of `exit` is $\{in, exit\}$, the separator of `stop` is $\{in, stop\}$, and the separator of atomic statements `t`, `s`, .. is $\{in, out\}$.

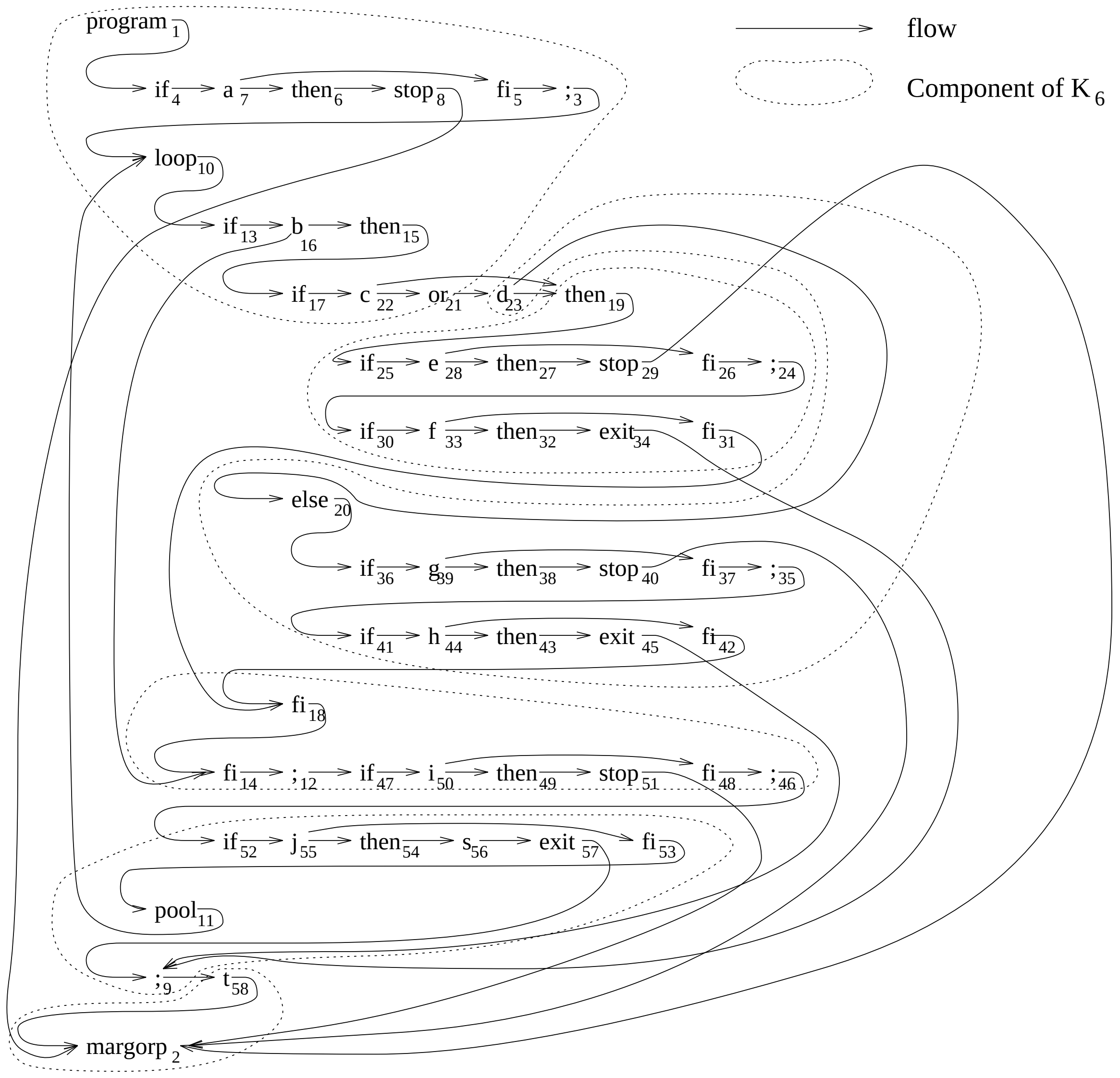

Figure 2.2: The 5-complex control-flow graph of a full program produced by FLOW. It is shown how the edges can be contracted so as to derive a clique of size 6, which is trivially 5-complex. Hence the original control-flow graph is ($\geq$ 5)-complex. The ($\leq$ 5)-complex listing produced by FLOW demonstrates that it is exactly 5-complex.

For a boolean expression $B = B' \,\texttt{or}\, B''$, we first visit `or`, which gets separator $\{in, true, false\}$. The neighbors of B' and B'' are like those of B except that $false' = in'' = \texttt{or}$. The case of $B = B' \,\texttt{and}\, B''$ is similar except that we get $true' = in'' = \texttt{and}$. The separator of an atomic boolean `a`, `b`, .. is $\{in, true, false\}$. ■

Note in the above proof that only word that gets a separator of size 5 is `else`. Unfortunately, a separator of size 5 cannot be avoided in general. The example in figure 2.2 shows that there are are programs written in STRUCTURED that do not have (≤ 4)-complex listings, hence that the bound in Theorem 7 cannot be improved.

We will now show how to adapt the proof of Theorem 7 so as to prove the claims in Theorem 3. First, in Theorem 3 it was claimed that the complexity decreases by one if we do not have short circuit evaluation. Recall from the proof of Theorem 7 that only the word `else` gets a separators of size 5. Suppose we do not have short circuit evaluation. The immediate consequence is that for a boolean expression B, the neighbors $true$ and $false$ both become identified with the word succeeding B in the program. Now consider a statement $S = \texttt{if}\, B' \,\texttt{then}\, S'' \,\texttt{else}\, S''' \,\texttt{fi}$. Let B' be of the form Cb meaning that b is the last word in B'. Without short circuit evaluation, we know that the evaluation of B' pass through b. Now change the visiting sequence so that we first visit `if`, `fi`, b, `then`, and `else`, and then visit C, S'', and S''' recursively. The separators for `if` and `fi` are unchanged, but we get the separator $\{\texttt{if}, \texttt{fi}, exit, stop\}$ for b and $\{b, \texttt{fi}, exit, stop\}$ for both `then` and `else`, that is, no word gets a separator of size > 4.

We will now argue that the programming language Modula-2 has the same complexity as STRUCTURED. First note that Modula-2 does have all the structural (flow-affecting) statements from STRUCTURED. The `stop`-statement from STRUCTURED is not directly available, but in Modula-2 functions we can have multiple return-statements and these have the effect of the stop-statement on the control-flow graphs for functions. Thus, the complexity of Modula-2 is no less than that of STRUCTURED. We now need to show that the structural statements of Modula-2 that are not in STRUCTURED do not increase the complexity. Of these we have `while`-statements, `repeat`-statements and `case`-statements. The `while`-statements and `repeat`-statements are just special cases of our general loop. If a `case`-statement is evaluated by visiting the cases one by one until the right one is found, the control-flow of a `case`-statement is equivalent to that of a a combination of `if-then-else-fi`-statements. However, based on the value of the case selector, we may be able to jump directly to the relevant case, giving a flow like the one illustrated in Figure 2.3. The visiting sequence of the key-words of the `case`-statement is indicated by the indices in the figure. The important point is that we visit `of` and `esac` before we visit any words in the individual cases. Following the terminology of the proof of Theorem 7, the separators for the key-words become $\{in, out, exit, stop\}$ for `case`, $\{\texttt{case}, out, exit, stop\}$ for `of`, $\{\texttt{of}, out, exit, stop\}$ for `esac`, and $\{\texttt{of}, \texttt{esac}, exit, stop\}$ for each `:`, that is, all key-words get a separator of size 4. Thus, even with a direct jump to the relevant case, a `case`-statement does not affect our complexity of 5 negatively. The reader may feel that the jump to the right case takes place directly from `x`. We can achieve this flow simply by contracting the edge from `x` to `of` and removing `of`. By Lemma 6, such a contraction not affect the complexity of the listing. This completes our argument that Modula-2 has the same complexity at STRUCTURED.

Next we turn our attention to Pascal and C. First note that these programming languages do not use key-words like `fi` explicitly terminating an `if-then-else`-statement. This difference is, however, inconsequential, for we can always remove a `fi` from the control-flow graph by contracting the out-going edge to the succeeding word. By Lemma 6, such a contraction cannot increase the complexity.

We will now argue that the complexity of `goto`-free Pascal is two less than that of Modula-2. The difference is due to the Pascal's lack of a general `loop`-statement with multiple `exit`s, and lack of multiple `stop/return`-statements. Referring to the proof of Theorem 7, this removes both

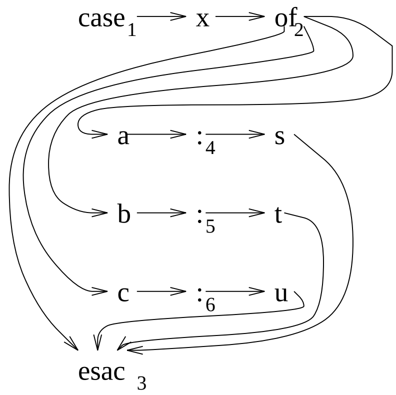

Figure 2.3: Flow of a `case`-statement

the *exit* and the *stop* from the potential neighborhood of statements. Consequently the complexity is reduced by two except that we now have to deal directly with `while`-loops and `repeat`-loops. In the discussion of Modula-2, these loops where just considered special cases of the general loop, but now we will show that they are strictly simpler as they have only one pre-determined exit. Consider a loop of the form $S = \texttt{while}\, B' \,\texttt{do}\, S'' \,\texttt{elihw}$, where the neighborhood of S is in and out. Following the usual pattern, we first visit `while`, `do`, and `elihw`, and then we visit B' and S'' recursively. Thereby we get the separators $\{in, out\}$ for `while`, $\{\texttt{while}, out\}$ for `do`, and $\{\texttt{do}, \texttt{while}\}$ for `elihw`. Thus, all the key-words of a `while`-loop get separators of size 2. For B' the neighborhood becomes $in' = \texttt{while}$, $true' = \texttt{do}$, and $false' = out$. For S'' the neighborhood becomes $in'' = \texttt{do}$ and $out'' = \texttt{while}$. A `repeat`-loop may be treated identically, and hence we conclude that the complexity of Pascal is reduced by two relative to that of STRUCTURED.

In Theorem 3, we claimed that the complexity of `goto`-free C is one higher than that of Modula-2. The difference is that for loops in C, besides a `break`-statement corresponding to the above `exit`-statement, we have a `continue`-statement bringing the control back to the beginning of the loop. Consequently, the neighborhood of statements is augmented with a *continue* being the beginning of the nearest surrounding loop. The separators then have to be increased accordingly. This completes our outline of a proof of Theorem 3.

3 Robustness with respect other optimizations

In this section, we will briefly discuss how our simple listings from the previous section may be preserved under the standard optimizations mentioned in [2], hence that Theorem 3 holds despite these optimizations. For most optimizations, we just note that they can be done preserving the structural statements of a program. This holds for: redundant-instruction elimination, algebraic simplifications, use of machine idioms, elimination of global common subexpression, code motion, copy propagation, and elimination of induction variables. The only optimization from [2] that has to be done after the translation into object code, such as three-address code, is flow-of-control optimization. However, it turns out that standard flow-of-control optimization does not increase the complexity of the control-flow-graph. This is essentially because flow-of-control optimization corresponds to contraction of edges in the control-flow graph, as dealt with in Lemma 6. Take, for example, the following type of optimization from [2, p. 556] (a less contrived example is coming in Figure 3.4):

```
L0: goto L2            L0: goto L2
```

```
L1: goto L3              L1: goto L3
L2: goto L4  --->
L3: S                    L3: S
L4: T                    L2: T
```

The label numbers are assumed to be the numbers in the listing. The above program corresponds to a control-flow graph with edges $(0,2),(1,3),(2,4),(3,4)$. The optimization corresponds to contracting the edge $(2,4)$, giving the program point contracted to, the smaller number (2). This contraction implies that all old jumps to 4 now jumps to 2. Another example is,

```
L0: goto L2              L0: goto L2             L0: goto L2
L1: S                    L1: S; goto L2          L1: S
L2: goto L4  --->                          --->  Ln: goto L2
L3: T                    L3: T                   L3: T
L4: U                    L2: U                   L2: U
```

Above, the first transformation is the same as before, but it introduces an extra statement that needs to have its own program point. This is achieved by the second transformation. Above n is understood to be the last free program point. We need to argue that this choice does not increase the complexity of our listing, but that follows from

Lemma 8 *Let L be a $(\leq k)$-complex listing L of a graph G. Suppose an edge (v,w) is split by a new vertex u (replacing (v,w) with (v,u) and (u,w)). We then get a new $(\leq \max\{k,2\})$-complex listing if we inserting u anywhere after v and w in the listing.*

Proof: The separator of u is $\{v,w\}$, and since u is last in the listing, it does not affect the separators of any of the other vertices. ■

Lemmas 6 and 8 suffice for any flow-of-control optimization mentioned in [2]. A more extensive example of flow-of-control optimization is presented is presented in Figure 3.4.

4 Finding the separators

In this section, we present an efficient algorithm for finding the minimal separators of a listing of the vertices in a graph. Such an algorithm is useful in connection with transformations like those presented in Section 3 because it allows us to forget about the separators until the we have a final listing. Also, it will be useful for Appendix A, where we will present a heuristic for finding good listings but where the separators are yet to be computed.

Consider a graph $G=(V,E)$ and a listing $L=v_1,\ldots,v_n$ of V. By a *separator path* from v_i to v_h, $i>h$, we mean a path $v_i v_{i_1}\cdots, v_{i_k} v_h$ such that $i_1,\ldots,i_k \geq i$. If $k=0$, $v_i v_h$ is a separator path of length 1. Then

$$S(v)=\{u|\text{there is a separator path from } v \text{ to } u\}$$

is a minimum separator for v, contained in any other separator for v.

For technical reasons, below, we allow separator paths to be self-intersecting.

Algorithm A: Input $G=(V,E)$ and a listing $L=v_1,\ldots,v_n$ of V. Outputs the minimum separator $S(v)$ for each $v\in V$ in time $O(\sum_v S(v))$. Below $S^{-1}(w)=\{v|w\in S(v)\}$ is up-dated implicitly together with $S(\cdot)$.

A.1. For $i:=n,\ldots,1$:

A.1.1. $\quad S(v_i):=\{v_h|(v_i,v_h)\in E, h<i\}$.

```
PROGRAM                                   THREE-ADDRESS CODE          SEPARATOR

loop (1)                                  L1: skip                    {0}
   S (4)                                  L4: S                       {1,3}
   ; (3)                                  L3: skip                    {0,1,2}
   if (5)                                 L5: skip                    {0,2,3}
      A (9)                               L9: if not A then goto L4   {5,6,8}
        and (8)                           L8: skip                    {5,6,7}
            B (10)                        L10:if not B then goto L4   {6,7,8}
              then (7)                    L7: skip                    {0,5,6}
      if (11)                             L11:skip                    {0,6,7}
         C (15)                           L15:if not C then goto L14  {11,13,14}
           then (13)                      L13:skip                    {0,11,12}
         T (16)                           L16:T; goto L12             {6,13}
      else (14)                           L14:skip                    {0,11,13}
         exit (17)                        L17:goto L0                 {0,14}
      fi (12)                             L12:skip                    {0,6,11}
   fi (6)                                 L6: skip                    {0,2,5}
pool (2)                                  L2: goto L1                 {0,1}
; (0)                                     L0:                         {}
```

```
L1: S                        {0}          L1: S                       {0}
L8: if not A then goto L2    {1,2,7}      L8: if not A then goto L1   {1,7}
L7: if not B then goto L2    {0,1,2}      L7: if not B then goto L1   {0,1}
L13:if not C then goto L14   {0,2,7}      L13:if not C then goto L14  {0,1,7}
L16:T; goto L2               {2,13}       L17:T                       {13,16}
L14:goto L0                  {0,13}       L16:goto L1                 {1,13}
L2: goto L1                  {0,1}        L14:goto L0                 {0,13}
L0:                          {}           L0:                         {}

Contract (1,4,3,5),(9,8),(10,7,11),       Insert 17 between 13 and 17,
(15,13),(14,17) and (12,6,2).             Contract (2,1).
```

```
L1: S                        {0}
L8: if not A then goto L1    {1,7}
L7: if not B then goto L1    {0,1}
L13:if not C then goto L0    {0,1,7}
L17:T                        {13,14}
L16:goto L1                  {1,13}
L0:                          {}

Contract (14,0)
```

Figure 3.4: Flow-of-control optimization applied to a simple program segment.

A.1.2. For all $w \in S^{-1}(v_i)$,
A.1.2.1. If $w \notin D$, for all $v_h \in S(w)$, $h < i$:
A.1.2.1.1. $S(v_i) := S(v_i) \cup \{v_h\}$.
A.1.2.1.2. $D := D \cup \{v_h\}$.

Correctness: We will prove that the following invariant is satisfied before Step A.1.1.

- For all $j > i$, $S(v_j)$ is the minimum separator for j.

Trivially, this is the case for $i = n$. Now, consider some value of $i < n$. We want to prove that v_h is inserted in $S(v_i)$ if and only if there is a "separator" path from v_i to v_h with all interior vertices in $\{v_j | j > i\}$. Step A.1.1 deals with the case where there is a separator path of length 1, hence with no interior vertices. Thus, it suffices to show that v_h is added to $S(v_i)$ in Step A.1.2.1.1 if and only if there is a separator path $v_i v_{\alpha_1} \cdots, v_{\alpha_k} v_h$ with $k \geq 1$.

$\Leftarrow$ We know we have separator paths $wv_{\alpha_1} \cdots, v_{\alpha_k} v_i$ and $wv_{\beta_1} \cdots, v_{\beta_l} v_h$. Since $w = v_j$ for some $j > i$, it follows that $v_i v_{\alpha_k} \cdots, v_{\alpha_1} w v_{\beta_1} \cdots, v_{\beta_l} v_h$ is a separator path (recall that separator paths may be self-intersecting), hence that the addition of v_h to $S(v_i)$ is correct.

$\Rightarrow$ Suppose there is a separator path of length > 1 from v_i to v_h, and let $v_i v_{\alpha_1} \cdots v_{\alpha_k} v_h$ be such a separator path where the minimal index $\mu = \min_{1 \leq \gamma \leq k}\{\alpha_\gamma\}$ of an interior vertex is as small as possible. Suppose $\alpha_\gamma = \mu$. Then $v_{\alpha_\gamma} \cdots v_{\alpha_1} v_i$ and $v_{\alpha_\gamma} \cdots v_{\alpha_k} v_h$ are separator paths, so $v_{\alpha_\gamma} \in S^{-1}(v_i)$ and $v_h \in S(v_{\alpha_\gamma})$. Hence v_h is correctly added to $S(v_i)$ in Step A.1.2.1.1 unless $v_{\alpha_\gamma} \in D$. However, if $v_{\alpha_\gamma} \in D$, it is because $v_{\alpha_\gamma} \in S^{-1}(v_j)$ for some $j > i$. Then we have a separator path $v_{\alpha_\gamma} v_{\beta_1} \cdots v_{\beta_l} v_j$. But then

$$v_i v_{\alpha_1} \cdots v_{\alpha_\gamma} v_{\beta_1} \cdots v_{\beta_l} v_j v_{\beta_l} \cdots v_{\beta_1} v_{\alpha_\gamma} \cdots v_{\alpha_k} v_h$$

is a separator path and $j < \alpha_\gamma = \mu$, contradicting the minimality of μ.

For the complexity of the algorithm, note that the loop of step A.1.2 is repeated exactly once for every $(v_i, w), v_i \in S(v)$. Also, due to step A.1.2.1.2, the condition $w \notin D$, is satisfied at most once for every w. Hence the loop of step A.1.2.1 is repeated at most once for every $(v_h, w), v_h \in S(w)$. In conclusion, the running time is $O(\sum_v S(v))$. ∎

5 Coloring the intersection graphs of k-complex graphs

Let $G = (V, E)$ be a graph and $v_1, \ldots, v_n$ a k-complex listing of its vertices. Moreover, let X be a set of connected sub-graphs of G, called *variables*, and let the *interference graph* I be the intersection graph over X. That is, two variables $v, w \in X$ are adjacent in I if and only if they they intersect in G. In this section, we study the problem of coloring I given that k is a constant. Our colors will be numbers $1, 2, \ldots$, and our aim is to minimize the maximum color used.

We use $\chi(I)$ to denote the chromatic number of I, i.e. the minimal number of colors needed to color I. For each v_i, set $P_{v_i} = \{v_1, \ldots, v_{i-1}\}$. Moreover, let S_{v_i} denote the set of $w \in P_{v_i}$ that can be reached by a path from v_i with no interior vertices in P_{v_i}. By Definition 1, $|S_{v_i}| \leq k$. Let X_{v_i} be the set of variables $x \in X$ containing v. Note that X_{v_i} is a clique in I. Set $\omega = \max |X_v| \leq \chi(I)$. Finally, let $X^*_{v_i}$ denote $X_{v_i} \cup \bigcup_{w \in S_{v_i}} X_w$.

Lemma 9 *If $x \in X_v$ does not intersect P_v but intersects a variable y intersecting P_v, then y intersects S_v.*

Proof: There exists a path in $x \cup y$ from v to a vertex in P_v. The first vertex u in this path which is in P_v is in S_v. Since x does not intersect P_v, $u \in y$. ∎

The following generalizes an algorithm from [26] for series-parallel graphs ($k \leq 2$):

Algorithm B: Colors I with at most $(k+1)\omega \le (k+1)\chi(I)$ colors.

B.1. For $i := 1, \ldots, n$,

B.1.1. Color the uncolored variables $x \in X_{v_i}$ with the smallest colors not used by variables intersecting S_{v_i}.

Correctness: From Lemma 9 it follows that the algorithm produces a proper coloring of I, i.e. that no two intersecting variables get the same color. When coloring the uncolored variables in X_{v_i}, the largest color used is at most the total number of colors used in $X^*_{v_i}$. Moreover $X^*_{v_i}$ is the union of at most $k+1$ optimally colored sets; namely the cliques $X_w, w \in S_{v_i} \cup \{v_i\}$. Thus the largest color used is $\le (k+1)\omega$. ∎

Note that we color I without constructing I. The algorithm is trivially implemented to run in time $O(n\omega k)$. This matches the time bound of the second algorithm from Theorem 2.

A *biclique* is a graph whose vertex set is partitioned into two cliques. Then

Lemma 10 ([26]) *Let $G = (V_1 \cup V_2, E)$ be a biclique on n vertices with the induced subgraphs on V_1 and V_2 being cliques. Then there is an $O(n^{2.5})$ algorithm that optimally colors G.* ∎

Using the same idea as in [26], we get an improved algorithm using at most $k\chi(I)$ colors if we modify step B.1.1 as follows. If $S_{v_i} \neq \emptyset$, choose any $p(v_i) \in S_{v_i}$ and color the biclique $I|(X_{v_i} \cup X_{p(v_i)})$ using Lemma 10. Rename the colors so that the coloring of $X_{p(v_i)}$ is not changed, and such that the new colors for X_{v_i} are the smallest not used in any $X_w, w \in S_{v_i}$. If $S_{v_i} = \emptyset$ then $X^*_{v_i} = X_{v_i}$ is one optimally colored set, and if $S_{v_i} \neq \emptyset$, $X^*_{v_i}$ is the union of at most k optimally colored sets; namely the biclique $X_{v_i} \cup X_{p(v_i)}$ and the cliques $X_w, w \in S_{v_i} \setminus \{p(v_i)\}$. Thus the modified algorithm uses at most $k\chi(I)$ colors. In [26], $k = 2$, so the change brings their approximation factor down from 3 to 2, which is their main result.

We will now get further down to $(\lfloor k/2 \rfloor + 1)\chi(I)$ colors by carefully choosing the $p(v_i) \in S_{v_i}$. Let $p(v_i) = \perp$ denote that $p(v_i)$ is undefined. Note that any graph F with edges $\{v, p(v)\}$, where $p(v) \in S_v$, is acyclic since $v_h \in S_{v_i}$ implies $h < i$. Hence F is a forest.

Algorithm C: Colors I with at most $(\lfloor k/2 \rfloor + 1)\chi(I)$ colors.

C.1. For $i := 1, \ldots, n$,

C.1.1. Let M_i be a maximal matching in the forest on S_{v_i} with edges $\{w, p(w)\} \in S^2_{v_i}$.

C.1.2. If M_i is perfect ($\bigcup M_i = S_{v_i}$) then

C.1.2.1. $p(v_i) := \perp$.

C.1.2.2. Color the uncolored variables $x \in X_{v_i}$ with the smallest colors not used by variables intersecting S_{v_i}.

C.1.3. If M_i is imperfect then

C.1.3.1. Choose $p(v_i)$ from $S_{v_i} \setminus \bigcup_i M_i$.

C.1.3.2. Color the biclique $I|(X_{v_i} \cup X_{p(v_i)})$ using Lemma 10. Rename the colors so that the coloring of $X_{p(v_i)}$ is not changed, and such that the new colors for X_{v_i} are the smallest not used in any $X_w, w \in S_{v_i}$.

Correctness: First note the loop invariant that $X_v \cup X_{p(v)}$ is an optimally colored biclique for all $p(v) \neq \perp$. For any $W \subseteq V$, let $\#(W)$ denote $|W| - |M|$ where M is a maximal matching in the forest on W with edges $\{w, p(w)\} \in W^2$. Then $\bigcup_{w \in W} X_w$ is the union of at most $\#(W)$ optimally colored sets; namely the $|M|$ bicliques $X_w \cup X_{p(w)}, \{w, p(w)\} \in M$, and the $|W| - 2|M|$ cliques $X_w, w \in W \setminus \bigcup M$. By induction on i we will show $\#(S_{v_i} \cup \{v_i\}) \le \lfloor k/2 \rfloor + 1$.

If M_i is perfect, $\#(S_{v_i}) = |S_{v_i}|/2$, so $\#(S_{v_i} \cup \{v_i\}) \leq \lfloor k/2 \rfloor + 1$. Note that $S_{v_1} = \emptyset$, so this covers the base case of our induction.

For the case where M_i is imperfect, let h is the largest index such that $v_h \in S_{v_i}$. Then $S_{v_i} \subseteq S_{v_h} \cup \{v_h\}$. Hence $\#(S_{v_i}) \leq \#(S_{v_h} \cup \{v_h\})$. By induction, $\#(S_{v_h} \cup \{v_h\}) \leq \lfloor k/2 \rfloor + 1$. Moreover, $M_i \cup \{\{v_i, p(v_i)\}\}$ is a matching in $S_{v_i} \cup \{v_i\}$, so $\#(S_{v_i} \cup \{v_i\}) = \#(S_{v_i})$. That is,

$$\#(S_{v_i} \cup \{v_i\}) = \#(S_{v_i}) \leq \#(S_{v_h} \cup \{v_h\}) \leq \lfloor k/2 \rfloor + 1.$$

This completes the induction, thus proving that the algorithm uses at most $(\lfloor k/2 \rfloor + 1)\chi(I)$ colors. ■

Proof of Theorem 2: First note that it only takes linear time to find a maximal matching M in a forest F, as in step C.1.1. Greedily pick for M any leaf incident edge $\{v, w\}$, and recurse on $F \setminus \{v, w\}$. The leaves are found by keeping track of the degrees. Then Theorem 2 follows from Lemma 10 and the correctness of Algorithms B and C. ■

It should be noted that our improvement from k to $\lfloor k/2 \rfloor + 1$ is not an improvement for the case $k = 2$ studied in [26], but it is an improvement for any $k > 2$.

6 The complexity and tree-width

In this section, we prove the statement of Theorem 5 that graph has tree-width k if and only if it is k-complex. The notion of tree-width was introduced by Robertson and Seymour [34].

Definition 11 *A* tree-decomposition *of a graph $G = (V, E)$ is defined by a tree $T = (I, F)$ together with a family $\{W_i\}_{i \in I}$ of subsets of V such that*

(i) $\bigcup_{i \in I} W_i = V$

(ii) *for all edges $(v, w) \in E$, there exists an $i \in I$ such that $\{v, w\} \subseteq W_i$.*

(iii) *for all $i, j, k \in I$, if j is on the path from i to k then $W_i \cap W_k \subseteq W_j$.*

The width *of the decomposition is* $\max_{i \in I} |W_i| - 1$ *and the* tree-width *of G is the minimal width over all tree-decompositions of G.*

Lemma 12 *Given a k-complex listing $v_1, \ldots, v_n$ of the vertices in G, including the separators S_{v_i}, in linear time, we can construct a tree-decomposition of G of width k.*

Proof: The tree T will have the nodes $1, \ldots, n$. We build up T starting from T_1 consisting of the root 1, and setting $W_1 = \{v_1\}$. Now, for $i = 2, \ldots, n$, let h the the largest index such that $v_h \in S_{v_i}$. Set $T_i = T_{i-1} \cup \{(h, i)\}$ and $W_i = S_{v_i} \cup \{v_i\}$. Return $T = T_n$ and $\{W_i\}_{i \in I}$.

Clearly (i) is satisfied. Also (ii) is satisfied, for consider $\{v_h, v_i\} \in E$ with $h < i$. Then $v_h \in S_{v_i}$, so $\{v_h, v_i\} \in W_i$.

To prove (iii) we use induction on i. Trivially (iii) is satisfied for T_1. Let $T_i = T_{i-1} \cup \{(h, i)\}$ and suppose (iii) is satisfied for T_{i-1}. Consider any path p in T_i not in T_{i-1}. Then i is one of the end-vertices. Let k be the other end-vertex, and let j be any vertex between them. Trivially (iii) is satisfied if $j = i$, so we may assume that $j \neq i$, but then j is on the path in T_{i-1} between h and k. Thus, by induction, $W_j \supseteq W_h \cap W_k$. Since h is the largest index of a vertex in S_{v_i}, $S_{v_i} \subseteq S_{v_h} \cup \{v_h\} = W_h$. However, $W_i = S_{v_i} \cup \{v_i\}$ and $v_i \notin W_k$, so $W_i \cap W_k \subseteq W_h \cap W_k \subseteq W_j$. ■

Lemma 13 *Given a tree-decomposition $(T, \{W_j\}_{j \in I})$ of G of width k, in linear time, we can construct a k-complex listing $v_1, \ldots, v_n$ of the vertices in G including the separators S_{v_i}.*

Proof: Let $T = (I, F)$. Choose any rooting of T, and identify I with $\{1, \ldots, |I|\}$ such that $1, \ldots, |I|$ is a pre-ordering of T. Let $p(j)$ denote the parent of j. Thus $\forall j \in I, p(j) < j$. Also, 1 is the root of T.

Set $n_1 = |W_1|$. Let $\{v_1, \ldots, v_{n_1}\} = W_1$ and $S_{v_i} = \{v_1, \ldots, v_{i-1}\}$ for $i \leq n_1$. Since $|W_1| \leq k+1$, $|S_{v_i}| \leq k$. Let j run from 2 to $|I|$. Set $S_j = W_j \cap W_{p(j)}$, $U_j = W_j \setminus S_j$, and $n_j = n_{j-1} + |U_j|$. Finally, let $\{v_{n_{j-1}+1}, \ldots, v_{n_j}\} = U_j$, and $S_{v_i} = S_j \cup \{v_{n_{j-1}+1}, \ldots, v_{i-1}\}$ for $i = n_{j-1}+1, \ldots, n_j$.

To see that the above produces a valid listing, note for $j > 1$ that (ii) and (iii) implies that S_j separates U_j from $\bigcup_{k<j} W_k$ in G. Hence $U_j = W_j \setminus \bigcup_{k<j} W_k$. Together with (i) this implies that $\{U_j\}_{j \in I}$ is a partitioning of V, hence that all vertices get listed. Also, for $i = n_{j-1}+1, \ldots, n_j$, it implies that, indeed, $S_{v_i} = S_j \cup \{v_{n_{j-1}+1}, \ldots, v_{i-1}\}$ is a separator for v_i in the produced listing. Finally, $|S_{v_i}| \leq k$ since $S_{v_i} \subset W_j$ and $|W_j| \leq k+1$. ∎

Proof of Theorem 5: The theorem is the direct composition of Lemmas 12 and 13. ∎

Definitions 1 and 11 are thus equivalent[3]. Having established the link between structured programs and bounded tree-width, we are now ready to apply the many techniques for bounded tree-width to problems in control-flow graph analysis. Several powerful techniques are already known for constructing linear time algorithms for problems that are generally NP hard, or even P-space hard [4, 5, 9, 11, 17].

7 Concluding remarks

So far, we have only addressed structured programs of imperative languages. However, one could imagine that even goto-users have structured thoughts [28], hence that also the control-flow-graphs of their programs have simple listings/bounded tree-width. For variable k, the problem of deciding the tree-width is NP-hard [3]. For fixed k, however, there are linear time algorithms [8]. Also, for variable k, there has been work done on polynomial approximating algorithms [10]. Finally, the heuristic presented in Appendix A may be of some help, as it deals directly with three-address code that may contain any number of programmer supplied gotos. Our derivation of simple listings from syntax, as described in Section 2, is, however, much simpler than the general approaches to tree-width, so the general advice following from this paper is: you help not only yourself and your fellow humans [19, 31], but also the optimizer, by not using goto's, thus making the structure explicit from the syntax.

Concerning functional programming languages, the ML Kit with Regions [6, 36, Tofte *pc*] compiles Standard ML into goto-free blocks with structured statements of the form treated in this paper. Thus the techniques of this paper are not limited to imperative languages.

Detailed comparison with previous register allocators:

- The classic approach [15, 16] to register allocation via graph coloring uses the scheme: let x be a variable/vertex in the interference graph I. Color $I \setminus \{x\}$ recursively. Color x with the least color not used by any neighboring variable in I. Typically x is chosen to be of low degree, but this does not in itself lead to any guarantees for the quality of the produced coloring.

Consider the order of which our $(k+1)$-approximation algorithm (Algorithm B, Section 5) colors the vertices. The correctness proof implies that if we choose the x in the classic scheme following

[3] It should be noted that alternatively, we could have proved that our k-complex listings were equivalent to partial k-trees which is are known to be equivalent to tree-decompositions of width k. It is trivial to see that if a graph is a partial k-tree, it has a k-complex listing. However, the converse is not quite as trivial. Also, there are some algorithmic advantages to a constructive translation to tree-decompositions, where some W_i may be of smaller than the others.

this order reversely, the largest degree encountered becomes at most $(k+1)\omega - 1$. Here ω is the maximal number of variables live at any single point of the control-flow graph. Hence, given our ordering of the variables, the classic coloring scheme will use at most $(k+1)\omega$ colors, and since at least ω colors are needed, this is at most a factor $(k+1)$ from optimality. Thus our new register allocation algorithms can be seen as clever ways of running the classic register allocation, giving a good worst-case performance. In practice, we may, of course, often get a lot closer to optimality.

- The interference graph I may be of size quadratic in the number of variables and its construction is considered a main obstacle for coloring based register allocation. Hence, for space reasons, many heuristics aim at only having parts of the I constructed at any time [14, 24, 33]. For our $(\lfloor k/2 \rfloor + 1)$-approximation algorithm (Algorithm C, Section 5) the biggest sub-graphs considered are of size $O(\omega^2)$. From the k-complex listing, for each variable, our $(k+1)$-approximation algorithm identifies a small set of potential colored neighbors, but it never checks if they are actual neighbors, that is, it never checks for any two variables whether they actually interfere. Thus our $(k+1)$-approximation algorithm does not produce any part of the interference graph!
- In [24] they color straight line code optimally. By definition the control-flow graph of straight line code is a single path which is 1-complex and is hence also optimally colored by our $(\lfloor k/2 \rfloor + 1)$-approximation algorithm. In [24] they try to use this for the coloring of the straight line code in an innermost loop - which is assumed to be executed most frequently. Good coloring of the innermost loops is also the concern in [14]. However, as observed in [14], if there is more than one innermost loop, the coloring of one may negatively effect the possibility of coloring the other. The variables of different innermost loops may interfere non-trivially, so we cannot just address them independently. Now, suppose that all the most critical parts, like innermost loops, have been marked. Our approximation algorithms can then be used to first find a good coloring of the variables in the critical parts, and afterwards color the rest of the variables with different colors.
- Our algorithms are generalizations of those in [26] for series parallel control-flow graphs. If the control-flow graph is not series parallel, [26] suggest heuristics for removing a minimal set of edges so that the graph becomes a series-parallel. The removed "exception" edges requires special treatment. If the program execution goes through an exception edge, all register values may have to be reassigned. Note that with our approach we have no exceptions: the tree-width may vary, but this only affects the quality of the coloring; no special action needs to be taken.

In [26] they mention short circuit evaluation as a prime example of an obstruction to series parallelism. For example, goto-free Pascal without short circuit evaluation has series parallel control-flow graphs, but if we allow short circuit evaluation we may get exceptions to series parallelism. In fact, short circuit evaluation alone may give rise to arbitrarily many exceptions to series parallelism. However, the tree-width only grows from 2 to 3. As stated in Corollary 4, for our $(\lfloor k/2 \rfloor + 1)$-approximation algorithm, this change does not give a worse approximation factor!

Implications

- With reference to Theorem 3 and 5, it seems that tree-width of control-flow graphs offers a well-defined mathematical measure for how structured programs, or programming languages, are. Tree-width is an established measure for the computational complexity of graphs, and now it turns out to capture aspects of structured programming [18].
- Our result suggests banning gotos for the sake of optimization. Gotos have long been considered harmful to the readability of programs for humans [19, 31], and further gotos may obstruct bounded tree-width. Wirth's move from Pascal [37] to Modula-2 [38] is an excellent example of what can be done. In Modula-2 there are no gotos, but to reconcile the programmers, the language have been enriched with some extra exit structures - multiple exits from loops and multiple returns from functions. As a consequence, we get a tree-decomposition of width at most 5 as a free side-effect of the parsing of any Modula-2 program.

- A substantial theory of tree-width has developed, and we contribute to this theory by showing that not only are graphs of bounded tree-width computationally simple, but so are their intersection graphs. Concretely we color intersection graphs of graphs with tree-width k within a factor $O(k)$ from optimality, while for some ε, coloring within a factor $O(n^{\varepsilon})$ from optimality is NP-hard for general graphs [29]. This is the first concrete result demonstrating the computational simplicity of intersection graphs of graphs of bounded tree-width.
- For bounded tree-width, many linear time algorithms are known for problems that are otherwise NP-complete [4, 5, 11, 17] or PSPACE-complete [9]. As a consequence of Theorem 3 together with Theorem 5, this understanding may now be applied in control-flow graph analysis. The constants bounding the tree-width are truly small (≤ 6), allowing us to develop algorithms working well in both theory and practice. A concrete application of bounded tree-width within control-flow graph analysis is presented in [1].

References

[1] S. Alstrup, P.W. Lauridsen, and M. Thorup, Generalized dominators for structured programs. In *Proceedings of the 3rd Static Analysis Symposium, LNCS 1145*, pages 42–51, 1996.

[2] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass., 1986.

[3] S. Arnborg, D.G. Corneil, and A. Proskorowski, Complexity of Finding Embeddings in a k-Tree, *SIAM J. Alg. Disc. Meth.* **8** (1987) 277–284.

[4] S. Arnborg, J. Lagergren, and D. Seese, Easy problems for tree-decomposable graphs, *J. Algorithms* **12** (1991) 308–340.

[5] S. Arnborg and A. Proskorowski, Linear time algorithms for NP-hard problems restricted to partial k-trees, *SIAM J. Alg. Disc. Meth.* **23** (1989) 11–24.

[6] L. Birkdal, M. Tofte, and M. Vejlstrup, From region inference to von Neyman Machines via region representation inference. *in* "Proc. POPL'96," pp. 171–183, 1996.

[7] H.L. Bodlaender, A Tourist Guide Through Treewidth, *Acta Cybernetica* **11** (1993) 1–23.

[8] H.L. Bodlaender, A Linear Time Algorithm for Finding Tree-Decompositions of Small Treewidth, *in* "Proc. 25th STOC," pp. 226–234, 1993.

[9] H.L. Bodlaender, Complexity of Path Forming Games, *Theor. Comp. Sc.* **110** (1993) 215–245.

[10] H.L. Bodlaender, J.R. Gilbert, H. Hafsteinsson, and T. Kloks, Approximating Treewidth, Pathwidth, Frontsize, and Shortest Elimination Tree, *J. Algorithms* **18**, 2 (1995) 221–237.

[11] R.B. Borie, R.G. Parker, and C.A. Tovey, Automatic Generation of Linear-Time Algorithms from Predicate Calculus Descriptions of Problems on Recursively Constructed Graph Families, *Algorithmica* **7** (1992) 555–581.

[12] P. Briggs, Register allocation via graph coloring, PhD Thesis, Rice University, 1992.

[13] P. Briggs, K.D. Cooper, K. Kennedy, and L. Torczon, Coloring heuristics for register allocation, *in* "Proc. SIGPLAN'89 Conf. Programming Language Design and Implementation," pp. 275–284, 1989.

[14] D. Callahan and B. Koblenz, Register allocation via hierarchical graph coloring, *in* "Proc. SIGPLAN'91 Conf. Programming Language Design and Implementation," pp. 192–203, 1991.

[15] G.J. Chaitin, Register Allocation and Spilling via Graph Coloring, *in* "Proc. SIGPLAN'82 Symp. Compiler Construction," pp. 98–105, 1982.

[16] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hoplins, and P.W. Markstein, Register allocation via graph coloring, *Computer Languages* **6** (1981) 47–57.

[17] B. Courcelle and M. Mosbah, Monadic second-order evaluations on tree-decomposable graphs, **6** (1993) 49–82.

[18] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, *Structured Programming*, Academic Press, London, 1972.

[19] E.W. Dijkstra, Go To Statement Considered Harmful, *Comm. ACM* **11**, 3 (1968) 147–148.

[20] A.P. Ershov, Reduction of the problem of memory allocation in programming to the problem of colouring the vertices of a graph, *Doklady Academii Nauk SSSR* **142**, 4 (1962) 785–787. English version in *Soviet Mathematics* **3** (1962) 163–165.

[21] M.R. Garey, D.S. Johnson, G.L. Miller, and C.H. Papadimitriou, The Complexity of Coloring Circular Arcs and Chords, *SIAM J. Alg. Discr. Meth.* **1**, 2 (1980) 216–227.

[22] F. Gavril, Algorithms for Minimum Coloring, Maximum Clique, Minimum Covers by Cliques, and Maximum Independent Set of Chordal Graphs, *SIAM J. Comp.* **1**, 2 (1972) 180–187.

[23] F. Gavril, The Intersection Graph of Subtrees in Trees Are Exactly the Chordal Graphs, *J. Comb. Th. Ser. B* **16** (1974) 47–56.

[24] R. Gupta, M.L. Soffa, and T. Steele, Register allocation via clique separators, *in* "Proc. SIGPLAN'89 Conf. Programming Language Design and Implementation," pp. 264–274, 1989.

[25] M. M. Halldórsson, A Still Better Performance Guarantee for Approximate Graph Coloring, *Inf. Proc. Lett.* **45** (1993) 19–23.

[26] S. Kannan and T. Proebsting, Register Allocation in Structured Programs, *in* "Proc. 6th SODA," pp. 360–368, 1995.

[27] B.R. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, New Jersey, 1978.

[28] D.E. Knuth, Structured Programming with Go To Statements, *ACM Computing Surveys* **6**, 4 (1974) 261–301.

[29] C. Lund and M. Yannakakis, On the Hardness of Approximating Minimization Problems, *J. ACM* **41** (1994) 960–981.

[30] P. Naur, Revised Report on the Algorithmic Language Algol 60, *Comm. ACM* **6**, 1 (1963) 1–17.

[31] P. Naur, Go To Statements and Good Algol Style, *BIT* **3**, 3 (1963) 204–208.

[32] T. Nishizeki, K. Takamizawa, and N. Saito, Algorithms for detecting series-parallel graphs and D-charts, *Trans. Inst. Elect. Commun. Eng. Japan* **59**, 3 (1976) 259–260.

[33] C. Norris and L.L. Pollock, Register Allocation over the Program Dependence Graph, *in* "Proc. SIGPLAN'94 Conf. Programming Language Design and Implementation," pp. 266–277, 1994.

[34] N. Robertson and P.D. Seymour, Graph Minors I: Excluding a Forest, *J. Comb. Th. Ser. B* **35** (1983) 39–61.

[35] N. Robertson and P.D. Seymour, Graph Minors XIII: The Disjoint Paths Problem. *J. Comb. Th. Ser. B* **63** (1995) 65–110.

[36] M. Tofte and J-P. Talpin, Implementing the call-by-value lambda-calculus using a stack of regions. *in* "Proc. POPL'94," pp. 188–201, 1994.

[37] N. Wirth, The Programming Language PASCAL, *Acta Informatica* **1** (1971), 35–63.

[38] N. Wirth, *Programming in Modula-2 (3rd corr. ed.)*, Springer-Verlag, Berlin, New York, 1985.

A Simple listings from three address code

In this appendix, we present a heuristic for finding a simple listing when the input is in three-address code without structural statements. This allows us to integrate our register allocation with compilers where the program has been reduced to three-address code before register allocation is started. Our heuristic is much simpler and faster than the general known algorithms for finding tree-decompositions of graphs, and it will work well on three-address code produced as in [2] from structured programs. The heuristic has the advantage of producing a listing of any three-address code. In particular, it is expected to find good listings even for programs with a limited or structured use of general gotos. Similar heuristics should work for other types of intermediate code. Especially, we can take advantage of intermediate code containing more structural information than three-address code.

Intuitively, good listings have the following two characteristics: (1) program points that we exit to from many places, say from short-circuit evaluation or loops, should be listed early. (2) The entries of loops and conditional blocks should be listed before points in the body. Our heuristic will be designed with these basic goals in mind, and yet we will try to keep it very simple, not trying to identify the exact original structure of the program. With regards to (1), we essentially just try to list the statements in backwards order. For an if-then-else statement, this has the positive side-effect that we complete listing the else-part before we start listing the then-part. To satisfy (2), we introduce a general way of identifying the most important entries of various structures.

Consider a set I of pairs $(i,j) \in \{1,\ldots,n\}^2$. An *$I$-chain* from i to j, $i<j$, is a sequence of pairs $(i_1,j_1),\ldots,(i_l,j_l) \in I$ such that for all $k<l$, $i_k < i_{k+1} < j_k < j_{k+1}$. An I-chain from i to j is *maximal* if there is neither an $i' < i$ with an I-chain from i' to j, nor a $j' > j$ with an I-chain from i to j'. We shall return to the computation of maximal I-chains in Algorithm E

We are given a list $s_1 \cdots s_n$ of statements, where some contain a jump to another. Let J be the set of pairs (i,j) such that s_i contains a jump to j. Intuitively maximal J-chains are used to bring us from the beginning to the end of a conditional structure. Let S be the symmetric closure of J, i.e. $(i,j) \in S$ iff either $(i,j) \in J$, or $(j,i) \in J$. Intuitively maximal S-chains are used to bring us from loop or conditional structures to their exit points. The above intuition is very simplistic, but nevertheless, we suggest the following heuristic for generated low-complexity listings of three-address code:

Algorithm D: Finding a good listing:

D.1. $i := 0$;

D.2. For $j := n$ downto 1,

D.2.1. If s_j is not marked, mark s_j by i; $i := i+1$;

D.2.2. If there is a maximal S-chain from k to j and s_k is not marked, mark s_k by i; $i := i+1$;

D.2.3. If there is a maximal J-chain from k to j and s_k is not marked, mark s_k by i; $i := i+1$;

Although it is very techical, it can be shown that the above heurstics will give (≤ 5)-complex listings if applied to the three address code generated as in [2] from a program with structural statements from STRUCTURED. In fact, it even seems to work well after the standard optimizations discussed in 3. The working of the heuristic is illustrated in figure A.5. First we have the three address code obtained from the program from figure 2.2 using the flow-of-control optimization from Section 3. As in Section 3, we have used the labels to indicate the listing. Also, for each three-address code statement, we have the corresponding separator. To the right is the new listing generated by Algorithm D with the corresponding new separators. All separators were found using Algorithm A. What we see is the the maximal separator is of size 5 for both the old and the new listing. We will now describe how to find the end-points of maximal chains.

Three address code	Separators	New listing	New Separators
L1: if a then goto L2	{}	2	{1}
L10: if not b then goto L12	{1,2,9}	4	{1,2,3}
L17: if c then goto L19	{2,9,10,12}	11	{1,3,4,8,10}
L21: if not d then goto L20	{17,19,20}	15	{10,11,14}
L19: if e then goto L2	{2,9,12,17}	14	{1,10,11,13}
L30: if f then goto L9	{9,12,19}	13	{1,3,10,11,12}
L31: goto L12	{12,30}	12	{1,3,8,10,11}
L20: if g then goto L2	{2,9,12,17,19}	10	{1,3,4,8,9}
L35: if h then goto L9	{9,12,20}	9	{1,3,4,8}
L12: if i then goto L2	{2,9,10,11}	8	{1,3,4,7}
L46: if not j then goto L10	{2,11,12}	7	{1,3,4,6}
L54: s	{11,46}	6	{1,3,4,5}
L11: goto 10	{2,9,10}	5	{1,3,4}
L9: t	{1,2}	3	{1,2}
L2:	{1}	1	{}

Figure A.5: Applying the heuristic

Algorithm E: Given I, finds the set M of pairs (i, j) with a maximal I-chain from i to j.

E.1. $M := \emptyset$;

E.2. $s := 0$; $(i_0, j_0) := (0, n+1)$;

E.3. For $i := 1$ to n,

E.3.1. If $\exists j > i : (i, j) \in I$:

E.3.1.1. Let $j := \max\{j | (i, j) \in I\}$.

E.3.1.2. While $j_s \leq i$, $M := M \cup \{(i_s, j_s)\}$; $s := s - 1$;

E.3.1.3. While $j \geq j_s > i$, $i := i_s$; $s := s - 1$;

E.3.1.4. $s := s + 1$; $(i_s, j_s) := (i, j)$;

Correctness: The essential point is to note the following invariant for our stack $(i_0, j_0) \cdots (i_s, j_s)$:

$$i_0 < i_1 < \cdots < i_s < j_s < j_{s-1} < \cdots < j_0.$$

■

The computation "$j := \max\{j | (i, j) \in I\}$" takes $O(|I|)$ total time over all i. The rest of the algorithm runs in $O(n)$ total time. Since each statement can have at most one jump, $|J| \leq n$ and $|S| \leq 2n$. Hence the total running time of Algorithm D and E is $O(n)$. Thus our heuristic works in linear total time.