

---

# An Algorithm for Finding Connected Convex Subgraphs of an Acyclic Digraph

G. GUTIN, A. JOHNSTONE, J. REDDINGTON,  
E. SCOTT, A. SOLEIMANFALLAH, A. YEO

---

ABSTRACT. A subgraph  $H$  of an acyclic digraph  $D$  is convex if there is no directed path between vertices of  $H$  which contains an arc not in  $H$ . A digraph  $D$  is connected if the underlying undirected graph of  $D$  is connected. We construct an algorithm for enumeration of all connected convex subgraphs of a connected acyclic digraph  $D$  of order  $n$ . The time complexity of the algorithm is  $O(n \cdot cc(D))$ , where  $cc(D)$  is the number of connected convex subgraphs in  $D$ . The space complexity is  $O(n^2)$ . Connected convex subgraphs of connected acyclic digraphs are of interest in the area of modern embedded processors technology. Our computational experiments demonstrate that our algorithm is better than the state-of-the-art algorithm of Chen, Maskell and Sun. Moreover, unlike the algorithm of Chen, Maskell and Sun, our algorithm has a provable (almost) optimal worst time complexity.

Using the same approach, we design an algorithm for generating all connected induced subgraphs of a connected undirected graph  $G$ . The complexity of the algorithm is  $O(n \cdot c(G))$ , where  $n$  is the order of  $G$  and  $c(G)$  is the number of connected induced subgraphs of  $G$ . The previously reported algorithm for connected induced subgraph enumeration is of running time  $O(mn \cdot c(G))$ , where  $m$  is the number of edges in  $G$ .

## 1 Introduction

A subgraph  $H$  of an acyclic digraph  $D$  is convex if there is no directed path between vertices of  $H$  which contains an arc not in  $H$ . In Section 3, we consider an algorithm  $\mathcal{A}$  for generating all connected convex subgraphs of a connected acyclic digraph  $D$  of order  $n$ . The running time of  $\mathcal{A}$  is  $O(n \cdot cc(D))$ , where  $cc(D)$  is the number of connected convex subgraphs in  $D$ . Thus, the algorithm is (almost) optimal with respect to its time complexity. Interestingly, to generate only  $k$  cc-sets using  $\mathcal{A}$  we need  $O(n^3 + kn)$  time. In Section 4, we give experimental results demonstrating that the algorithm

is practical on reasonably large data dependency graphs for basic blocks generated from target code produced by the Trimaran's code generator [16]. Our experiments show that  $\mathcal{A}$  is better than the state-of-the-art algorithm of Chen, Maskell and Sun [4]. Moreover, unlike the algorithm in [4], our algorithm has a provable (almost) optimal worst time complexity.

Avis and Fukuda [2] designed an algorithm for generating all connected induced subgraphs in a connected graph  $G$  of order  $n$  and size  $m$  with time complexity  $O(mn \cdot c(G))$  and space complexity  $O(n + m)$ , where  $c(G)$  is the number of connected induced subgraphs in  $G$ . Observe that when  $G$  is bipartite there is an orientation  $D$  of  $G$  such that every connected induced subgraph of  $G$  corresponds to a connected convex subgraph of  $D$  and vice versa. To obtain  $D$  orient every edge of  $G$  from  $X$  to  $Y$ , where  $X$  and  $Y$  are partite sets of  $G$ .

The algorithm of Avis and Fukuda is based on a so-called reverse search. Applying the approach used to design  $\mathcal{A}$  to connected induced subgraph enumeration, in Section 5, we describe an algorithm  $\mathcal{C}$  for generating all connected induced subgraphs in a connected graph  $G$  of order  $n$  with much better time complexity,  $O(n \cdot c(G))$ . This demonstrates that our approach can be applied with success to various subgraph enumeration problems. The space complexity of our algorithm matches that of the algorithm of Avis and Fukuda.

In Section 6, we conjecture that  $\mathcal{A}$  is optimal and point out that  $\mathcal{A}$  can be easily modified to generate all convex subgraphs of an acyclic digraph.

### 1.1 Algorithm Application

There is an immediate application for  $\mathcal{A}$  in the field of so-called *custom computing* in which central processor architectures are parameterized for particular applications.

An embedded or *application specific* computing system only ever executes a single application. Examples include automobile engine management systems, satellite and aerospace control systems and the signal processing parts of mobile cellular phones. Significant improvements in the price-performance ratio of such systems can be achieved if the instruction set of the application specific processor is specifically tuned to the application.

This approach has become practical because many modern integrated circuit implementations are based on Field Programmable Gate Arrays (FPGA). An FPGA comprises an array of logic elements and a programmable routing system, which allows detailed design of logic interconnection to be performed directly by the customer, rather than a complete (and very high cost) custom integrated circuit having to be produced for each application. In extreme cases, the internal logic of the FPGA can even be modified

whilst in operation.

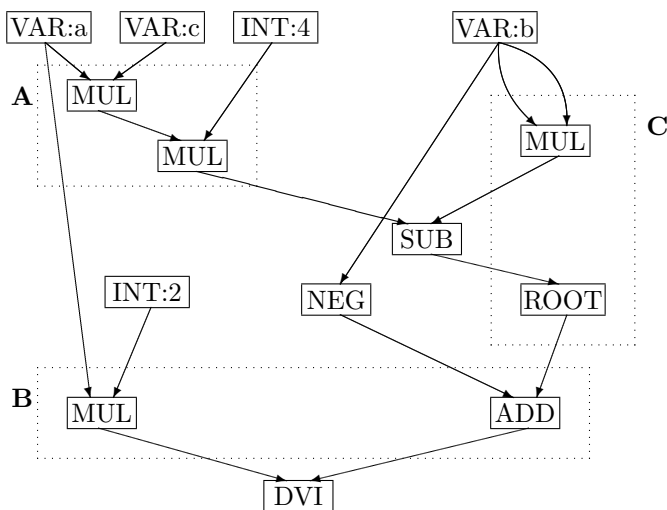
Suppliers of embedded processor architectures are now delivering *extensible* versions of their general purpose processors. Examples include the ARM OptimoDE [1], the MIPS Pro Series [13] and the Tensilica Xtensa [15]. The intention is that these architectures be implemented either as traditional logic with an accompanying FPGA containing the hardware for extension instructions, or be completely implemented within a large FPGA. By this means, hardware development has achieved a new level of flexibility, but sophisticated design tools are required to exploit its potential.

The goal of such tools is the identification of time critical or commonly occurring patterns of computation that could be directly implemented in custom hardware, giving both faster execution and reduced program size, because a sequence of base machine instructions is being replaced by a single custom *extension* instruction. For example, a program solving simultaneous linear equations may find it useful to have a single instruction to perform matrix inversion on a set of values held in registers.

The program analysis proceeds by first locating the *basic blocks* which are regions of sequential computation with no control transfers into them. For each basic block we construct a *data dependency graph* (DDG) which contains vertices for each base (unextended) instruction in the block, along with a vertex for each initial input datum. Figure 1 shows an example of a DDG. There is an arc to the vertex for the instruction  $u$  from each vertex whose instruction computes an input operand of  $u$ . DDG's are acyclic because execution within a basic block is by definition sequential.

Extension instructions are combinations of base machine instructions and are represented by subgraphs of the DDG. In Figure 1, sections A and B are convex subgraphs that represent candidate extension instructions. However, Section B is not connected. If such a region were implemented as a single extension instruction we should have separate independent hardware units within the instruction. Although this presents no special difficulties, present engineering practice is to restrict the search to connected convex components on the grounds that unconnected convex components are composed of connected ones, and that the system's code scheduler will perform better if it is allowed to arrange the independent computations in different ways at different points in the program.

Unlike connectivity however, convexity is not optional. An extension instruction cannot perform computations that depend on instructions external to the extension instruction. This means that there can be no data flows out of and then back into the extension instruction: the DDG subgraph corresponding to an extension instruction must be convex. Thus section C in Figure 1 does not represent a candidate extension instruction since it

Figure 1. Data dependency graph for  $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$ 

breaches the ‘no external computation rule’ because it is non-convex: there is a path *via* the **SUB** node that is not in the subgraph.

Ideally we would like to fully consider all possible candidate instructions and select the combination which results in the most efficient implementation. In practice this is unlikely to be feasible as, in worst case, the number of candidates will be exponential in the number of original program instructions. However, it is useful to have a process which can find all the potential instructions, even if the set of instructions used for final consideration has to be restricted. We then use heuristics based on block size and profiling data from sample runs of the application to select subgraphs for detailed implementation as custom extension instructions.

## 1.2 Related Theoretical Research

Many other algorithms for special subgraph generation have been studied in the literature. Kreher and Stinson [11] describe an algorithm for generating all cliques in a graph  $G$  of order  $n$  with running time  $O(n \cdot cl(G))$ , where  $cl(G)$  is the number of cliques in  $G$ .

Several algorithms were suggested for generation of all spanning trees in a connected graph  $G$  of order  $n$  and size  $m$ . Let  $t$  be the number of spanning trees in  $G$ . The first spanning trees generating algorithms [6, 12, 14] used backtracking which is useful for enumerating various kinds of subgraphs

such as paths and cycles. Using the algorithms from [12, 14], Gabow and Myers [6] suggested an algorithm of time complexity  $O(tn+n+m)$  and space complexity  $O(n+m)$ . If we output all spanning trees by their edges, this algorithm is optimal in terms of time and space complexities. Later another type of algorithms was developed; these algorithms (see, e.g., [10, 17, 18]) find a new spanning tree by exchanging a pair of edges. As a result, the algorithms of Kapoor and Ramesh [10] and Shioura and Tamura [17] requires only  $O(t+n+m)$  time and  $O(nm)$  space. The algorithm of Shioura, Tamura and Uno [18] is of the same optimal running time, but also of optimal space:  $O(n+m)$ .

An out-tree is an orientation of a tree such that all vertices but one are of in-degree 1. Kapoor, Kumar and Ramesh [9] presented an algorithm for enumerating all spanning out-trees of a digraph with  $n$  vertices,  $m$  arcs and  $t$  spanning out-trees. The algorithm takes  $O(\log n)$  time per spanning tree; more precisely, it runs in  $O(t \log n + n^2 \alpha(n, n) + nm)$ , where  $\alpha$  is the Inverse Ackermann function. It first outputs a single spanning out-tree and then a list of arc swaps; each spanning out-tree can be generated from the first spanning out-tree by applying a prefix of this sequence of arc swaps.

## 2 Terminology, Notation and Preliminaries

Let  $D$  be a digraph. If  $xy$  is an arc of  $D$  ( $xy \in A(D)$ ), we say that  $y$  is an *out-neighbor* of  $x$  and  $x$  is an *in-neighbor* of  $y$ . The set of out-neighbors of  $x$  is denoted by  $N_D^+(x)$  and the set of in-neighbors of  $x$  is denoted by  $N_D^-(x)$ . For a set  $X$  of vertices of  $D$ , its *out-neighborhood* (*in-neighborhood*) is  $N_D^+(X) = \cup_{x \in X} N_D^+(x) \setminus X$  ( $N_D^-(X) = \cup_{x \in X} N_D^-(x) \setminus X$ ). A digraph  $D^{TC}$  is called the *transitive closure* of  $D$  if  $V(D^{TC}) = V(D)$  and a vertex  $x$  is an in-neighbor of a vertex  $y$  in  $D^{TC}$  if and only if there is a path from  $x$  to  $y$  in  $D$ . It is easy to see that the cc-sets of  $D$  are precisely the cc-sets of  $D^{TC}$ .

Let  $S$  be a non-empty set of vertices of a digraph  $D$ . The set  $S$  is called *connected* if the subgraph  $D[S]$  of  $D$  induced by  $S$  is *connected*, i.e., its underlying (undirected) graph is connected. A directed path  $P$  of  $D$  is an  $S$ -path if  $P$  has at least three vertices, its initial and terminal vertices are in  $S$  and the rest of the vertices are not in  $S$ . The set  $S$  is *convex* if there is no  $S$ -path in  $D$ . The set  $S$  is a *connected convex set* (*cc-set*) if  $S$  is connected and convex. The subgraph induced by a cc-set is called a *cc-subgraph*. For a digraph  $D$ ,  $cc(D)$  denotes the number of cc-sets in  $D$ . An ordering  $v_1, v_2, \dots, v_n$  of vertices of an acyclic digraph  $D$  is called *acyclic* if for every arc  $v_i v_j$  of  $D$  we have  $i < j$ .

LEMMA 1. *Let  $D$  be a connected acyclic digraph and let  $S$  be a vertex set in  $D$ . Then  $S$  is a cc-set in  $D$  if and only if it is a cc-set in  $D^{TC}$ .*

**Proof.** Let  $S$  be a set of vertices of  $D$ . We will first prove that there is an  $S$ -path in  $D$  if and only if there is an  $S$ -path in  $D^{TC}$ . Since all arcs of  $D$  are in  $D^{TC}$ , every  $S$ -path in  $D$  is an  $S$ -path in  $D^{TC}$ . Let  $Q = x_1x_2 \dots x_q$  be an  $S$ -path in  $D^{TC}$ . Then there are paths  $P_2, P_3, \dots, P_q$  such that  $Q' = x_1P_2x_2P_3x_3 \dots x_{q-1}P_qx_q$  is a path in  $D$  ( $Q'$  must be a path since  $D$  is acyclic). Since  $x_1$  and  $x_q$  belong to  $S$  and  $x_2$  does not belong to  $S$ , there is a subpath of  $Q'$  which is an  $S$ -path.

If  $S$  is connected in  $D$  then it is clearly connected in  $D^{TC}$ , which implies that if  $S$  is a cc-set in  $D$  then it is a cc-set in  $D^{TC}$ . Now let  $S$  be a cc-set in  $D^{TC}$ . Assume that  $D[S]$  is not connected and let  $x$  and  $y$  be vertices in different connected components in  $D[S]$ , but which are connected by an arc in  $D^{TC}$ . Without loss of generality  $xy$  is the arc in  $D^{TC}$  and  $Q$  is a path from  $x$  to  $y$  in  $D$ . However as  $S$  is convex all vertices in  $Q$  also belong to  $S$  and therefore  $x$  and  $y$  belong to the same connected component in  $D[S]$ , a contradiction. ■

REMARK 2. The transitive closure of a digraph of order  $n$  can be found in  $O(n^3)$  time. See [3].

We will need the following two results proved in [7].

**THEOREM 3.** *For every connected acyclic digraph  $D$  of order  $n$ ,  $cc(D) \geq n(n+1)/2$ . If an acyclic digraph  $D$  of order  $n$  has a Hamiltonian path, then  $cc(D) = n(n+1)/2$ .*

**THEOREM 4.** *Let  $f(n) = 2^n + n + 1 - d_n$ , where  $d_n = 2 \cdot 2^{n/2}$  for every even  $n$  and  $d_n = 3 \cdot 2^{(n-1)/2}$  for every odd  $n$ . For every connected acyclic digraph  $D$  of order  $n$ ,  $cc(D) \leq f(n)$ . Let  $\vec{K}_{p,q}$  denote the digraph obtained from the complete bipartite graph  $K_{p,q}$  by orienting every edge from the partite set of cardinality  $p$  to the partite set of cardinality  $q$ . We have  $cc(\vec{K}_{a,n-a}) = f(n)$  provided  $|n - 2a| \leq 1$ .*

### 3 Algorithm for Generating Connected Convex Subgraphs of an Acyclic Digraph

In this section  $D$  denotes a connected acyclic digraph of order  $n$  and size  $m$ . Now we describe the main algorithm of this paper; we denote it by  $\mathcal{A}$ . The input of  $\mathcal{A}$  is  $D$  and  $\mathcal{A}$  outputs all cc-sets of  $D$ . The formal description of  $\mathcal{A}$  is followed by an example and proofs of correctness of  $\mathcal{A}$  and its complexity. Finally, we show that to produce  $k$  cc-sets  $\mathcal{A}$  requires  $O(n^3 + kn)$  time.

**Step 1:** Find the transitive closure of  $D$  and set  $D = D^{TC}$ .

**Step 2:** Find an acyclic ordering  $v_1, v_2, \dots, v_n$  of  $D$ .

**Step 3:** For each  $i = 1, 2, \dots, n$  do the following. Set  $X := \{v_i\}$ ,  $Y := \{v_{i+1}, v_{i+2}, \dots, v_n\}$  and call  $\mathcal{B}(X, Y, D)$ .

**Step 4 subroutine  $\mathcal{B}(X, Y, D)$ :** *Comments:  $\mathcal{B}(X, Y, D)$  finds all cc-sets  $S$  in  $D$  such that  $X \subseteq S \subseteq X \cup Y$ .*

*At each step, if possible we remove an element  $v$  from  $Y$  and add it to  $X$ . If  $X$  has out-neighbors we choose  $v$  to be the largest out-neighbor (line 3), otherwise if  $X$  has in-neighbors we choose  $v$  to be the smallest in-neighbor (line 8). Then we find the other nodes required to maintain convexity (line 4 or line 9). If there are no in- or out-neighbors we output  $X$ , otherwise we find all the cc-sets such that  $X \subseteq S \subseteq X \cup Y$  and  $v \in S$  (line 12) and then all the cc-sets such that  $X \subseteq S \subseteq X \cup Y$  and  $v \notin S$  (line 13).*

$\mathcal{B}(X, Y, D)\{$

1. set  $A = N_{D^{TC}}^+(X) \cap Y$
2. **if**  $A \neq \emptyset$  {
3.     set  $v = v_j$ , where  $j = \max\{i : v_i \in A\}$
4.     set  $R = \{v\} \cup (N_{D^{TC}}^-(v) \cap A)$  }
5. **else** {
6.     set  $B = N_{D^{TC}}^-(X) \cap Y$
7.     **if**  $B \neq \emptyset$  {
8.         set  $v = v_k$ , where  $k = \min\{i : v_i \in B\}$
9.         set  $R = \{v\} \cup (N_{D^{TC}}^+(v) \cap B)$  }
10. **if**  $A = \emptyset$  and  $B = \emptyset$  { output  $X$  }
11. **else** {
12.      $\mathcal{B}(X \cup R, Y \setminus R, D)$
13.      $\mathcal{B}(X, Y \setminus \{v\}, D)$  }

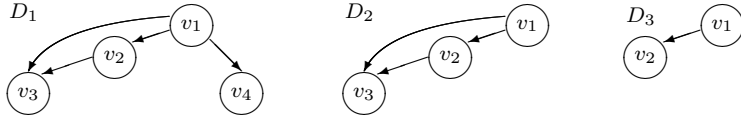
Before proving correctness of  $\mathcal{A}$ , we consider the following example.

EXAMPLE 5. Let  $D$  be the graph on the left below



In Step 1, we find  $A(D^{TC}) = A(D) \cup \{v_1v_3, v_2v_5, v_1v_5\}$  (above right). Observe that  $v_1, v_2, v_3, v_4, v_5$  is an acyclic ordering. We may assume that this is the ordering found in Step 2.

For  $i = 1$  in Step 3, we have  $X = \{v_1\}$  and  $Y = \{v_2, v_3, v_4, v_5\} = N^+(X)$ , and we call  $\mathcal{B}(\{v_1\}, \{v_2, v_3, v_4, v_5\})$ . Then in Step 4, line 1, we compute  $A = \{v_2, v_3, v_4, v_5\}$  and then, lines 3 and 4, obtain  $v = v_5$ ,  $N_{D^{TC}}^-(v) = \{v_1, v_2, v_3, v_4\}$  and  $R = \{v_2, v_3, v_4, v_5\}$ . Then, at line 12, we make a recursive call to  $\mathcal{B}(V(D), \emptyset, D)$ . In this call we have  $A = B = \emptyset$  so, at line 10, the set  $V(D) = \{v_1, \dots, v_5\}$  is output and the recursive call returns, to line 13 of  $\mathcal{B}(\{v_1\}, \{v_2, v_3, v_4, v_5\}, D)$ , where we make a call to  $\mathcal{B}(\{v_1\}, \{v_2, v_3, v_4\}, D)$ . We are now effectively looking at the graph  $D_1$  below.



In Step 4, lines 1-4, we compute  $A = \{v_2, v_3, v_4\}$  and obtain  $v = v_4$ ,  $N_{D^{TC}}^-(v) = \{v_1\}$  and  $R = \{v_4\}$ . At lines 12 and 13 we make a recursive calls to  $\mathcal{B}(\{v_1, v_4\}, \{v_2, v_3\}, D)$  and  $\mathcal{B}(\{v_1\}, \{v_2, v_3\}, D)$  respectively.

In the call to  $\mathcal{B}(\{v_1, v_4\}, \{v_2, v_3\}, D)$ , lines 1-4, we obtain  $v = v_3$  and  $R = \{v_2, v_3\}$ . This in turn generates calls to  $\mathcal{B}(\{v_1, v_4, v_2, v_3\}, \emptyset, D)$ , which just outputs  $\{v_1, v_2, v_3, v_4\}$  and returns, and  $\mathcal{B}(\{v_1, v_4\}, \{v_2\}, D)$ . The latter call generates calls to  $\mathcal{B}(\{v_1, v_4, v_2\}, \emptyset, D)$  and  $\mathcal{B}(\{v_1, v_4\}, \emptyset, D)$ , which output  $\{v_1, v_2, v_4\}$  and  $\{v_1, v_4\}$ , respectively.

In the call to  $\mathcal{B}(\{v_1\}, \{v_2, v_3\}, D)$ , where we are effectively looking at  $D_2$  above, we obtain  $v = v_3$  and  $R = \{v_2, v_3\}$ . This in turn generates calls to  $\mathcal{B}(\{v_1, v_2, v_3\}, \emptyset, D)$ , which just outputs  $\{v_1, v_2, v_3\}$  and returns, and  $\mathcal{B}(\{v_1\}, \{v_2\}, D)$  (graph  $D_3$  above). The latter call generates calls to  $\mathcal{B}(\{v_1, v_2\}, \emptyset, D)$  and  $\mathcal{B}(\{v_1\}, \emptyset, D)$ , which output  $\{v_1, v_2\}$  and  $\{v_1\}$ , respectively. This completes the case  $i = 1$  in Step 3, and all the cc-sets containing  $v_1$  have been output.

Now we perform Step 3 with  $i = 2$ , effectively looking at the graph  $D_4$ .



The call to  $\mathcal{B}(\{v_2\}, \{v_3, v_4, v_5\}, D)$  at generates further recursive calls in the following order

$$\begin{aligned} &\mathcal{B}(\{v_2, v_5, v_3\}, \{v_4\}, D) \\ &\quad \mathcal{B}(\{v_2, v_5, v_3, v_4\}, \emptyset, D), \text{ output } \{v_2, v_3, v_4, v_5\} \end{aligned}$$

$$\begin{aligned} & \mathcal{B}(\{v_2, v_5, v_3\}, \emptyset, D), \text{ output } \{v_2, v_3, v_5\} \\ & \mathcal{B}(\{v_2\}, \{v_3, v_4\}, D) \\ & \quad \mathcal{B}(\{v_2, v_3\}, \{v_4\}, D), \text{ output } \{v_2, v_3\} \\ & \mathcal{B}(\{v_2, v_3\}, \{v_4\}, D), \text{ output } \{v_2\}. \end{aligned}$$

Thus all the cc-sets containing  $v_2$  but not  $v_1$  are output.

Performing Step 3 again with  $i = 3$ , effectively looking at the graph  $D_5$  above, the call to  $\mathcal{B}(\{v_3\}, \{v_4, v_5\}, D)$ , generates the following recursive calls

$$\begin{aligned} & \mathcal{B}(\{v_3, v_5\}, \{v_4\}, D) \\ & \quad \mathcal{B}(\{v_3, v_5, v_4\}, \emptyset, D), \text{ output } \{v_3, v_4, v_5\} \\ & \quad \mathcal{B}(\{v_3, v_5\}, \emptyset, D), \text{ output } \{v_3, v_5\} \\ & \mathcal{B}(\{v_3\}, \{v_4\}, D), \text{ output } \{v_3\} \end{aligned}$$

which output all the cc-sets containing  $v_3$  but not  $v_1$  or  $v_2$ .

For the case  $i = 4$  in Step 3 we get the following calls

$$\begin{aligned} & \mathcal{B}(\{v_4\}, \{v_5\}, D) \\ & \quad \mathcal{B}(\{v_4, v_5\}, \emptyset, D), \text{ output } \{v_4, v_5\} \\ & \quad \mathcal{B}(\{v_4\}, \emptyset, D), \text{ output } \{v_4\} \end{aligned}$$

and for  $i = 5$  we get

$$\mathcal{B}(\{v_5\}, \emptyset, D), \text{ output } \{v_5\}$$

after which  $\mathcal{A}$  terminates.

**LEMMA 6.** *Algorithm  $\mathcal{A}$  correctly outputs all cc-sets of  $D$ .*

**Proof.** Recall, the convex (connected) sets of  $D$  are precisely the convex (connected) sets of  $D^{TC}$ . We prove the result for  $D^{TC}$ .

First we show that all the sets  $X$  output by  $\mathcal{A}$  are cc. Clearly it is sufficient to show that, within  $\mathcal{A}$ , for any call  $\mathcal{B}(X, Y, D)$  we have that  $X \cap Y = \emptyset$ ,  $X \cup Y$  is convex and  $X$  is a cc-set.

These properties hold for the Step 3 calls  $\mathcal{B}(\{v_i\}, \{v_{i+1}, \dots, v_n\}, D)$  as we have chosen an acyclic ordering. Thus we suppose that the properties hold for the sets  $X, Y$  and consider the sets  $X \cup R, Y \setminus R$  and  $X, Y \setminus \{v\}$  constructed in  $\mathcal{B}(X, Y, D)$ . In both cases clearly the intersections are empty, and since  $R \subseteq N_{D^{TC}}^+(X) \cup N_{D^{TC}}^-(X)$ ,  $X \cup R$  is connected.

Now suppose that there is a path  $u, y, w$  where  $u, w \in X \cup R$ . By convexity of  $X \cup Y$  we have  $y \in X \cup Y$ , so we suppose that  $y \in (Y \setminus \{v\})$ . If  $R = \{v\} \cup (N_{D^{TC}}^-(v) \cap N_{D^{TC}}^+(X) \cap Y)$ , since  $R \subseteq N_{D^{TC}}^+(X)$  and  $X$  is convex there cannot be any arcs  $rx \in R \times X$ . Thus, since  $u \in X \cup R$ , we must have  $w \in R$  and, by transitivity,  $y \in N_{D^{TC}}^+(X)$ . Then  $w \in N_{D^{TC}}^-(v)$  or  $w = v$  and in either case  $y \in N_{D^{TC}}^-(v)$ , so  $y \in R$ . Similarly, if  $R = \{v\} \cup (N_{D^{TC}}^+(v) \cap N_{D^{TC}}^-(X) \cap Y)$  then  $y \in R$ . So  $X \cup R$  is convex.

If there is a path  $u, v, w$ , where  $u, w \in (X \cup Y)$ , then, by choice of  $v$  and transitivity of  $D^{TC}$ , we have  $v, w \in N_{D^{TC}}^+(X)$  or  $u, v \in N_{D^{TC}}^-(X)$ . Since

we have an acyclic ordering, these cases contradict the maximal or minimal (respectively) choice of  $v$ . Thus  $X \cup (Y \setminus \{v\})$  is convex.

Secondly we show that if  $X \neq \emptyset$  is cc then  $X$  is output by  $\mathcal{A}$ . If  $S$  is a cc-set and  $j = \min\{i : v_i \in S\}$  then  $\{v_j\} \subseteq S \subseteq \{v_j, v_{j+1}, \dots, v_n\}$ . Thus it is sufficient to show that if  $S$  is cc and  $X \subseteq S \subseteq X \cup Y$  then  $\mathcal{B}(X, Y, D)$  outputs  $S$ . We prove this by induction on  $|Y|$ .

First we note that if  $(N_{DTC}^+(X) \cap Y) = \emptyset = (N_{DTC}^-(X) \cap Y)$  then, since  $S$  is connected,  $S = X$  and  $\mathcal{B}(X, Y, D)$  outputs  $X$  at line 10. This proves the result for  $|Y| = 0$ , and for  $|Y| \geq 1$  we may assume that  $v \in (N_{DTC}^+(X) \cap Y \cap N_{DTC}^-(X))$ .

If  $v \notin S$  then we have  $X \subseteq S \subseteq (X \cup (Y \setminus \{v\}))$  and  $|Y \setminus \{v\}| < |Y|$ , so by induction the call to  $\mathcal{B}(X, Y \setminus \{v\}, D)$  at line 13 outputs  $S$ . If  $r \in (R \setminus \{v\})$ , we have arcs  $rv$  and  $xr$ , for some  $x \in X \subseteq S$ . Thus, if  $v \in S$ , by convexity of  $S$  we have  $R \subseteq S$ . Then, since  $|Y \setminus R| < |Y|$ , the call to  $\mathcal{B}(X \cup R, Y \setminus R, D)$  at line 12 outputs  $S$ . ■

LEMMA 7. *The running time of  $\mathcal{A}$  is  $O(n \cdot cc(D))$ .*

**Proof.** Note that by Theorem 3 and the fact that  $D$  is connected we have  $n \times cc(D) \geq n^2(n+1)/2$ . Therefore the transitive closure of  $D$  can be found in  $O(n \cdot cc(D))$  time, by Remark 2. It is well-known that an acyclic ordering can be found in time  $O(n+m)$ , see, e.g., [3].

We will now show that  $\mathcal{B}(X, Y, D)$  runs in time  $O(|Y| \cdot cc'(X, Y))$ , where  $cc'(X, Y)$  is the number of cc-sets  $S$  such that  $X \subseteq S \subseteq X \cup Y$ . Note that  $\mathcal{B}(X, Y, D)$  returns at line 10 or makes two recursive calls to  $\mathcal{B}$  (lines 12,13). If  $\mathcal{B}(X, Y, D)$  returns at line 10 then we call this a *leaf* call otherwise the function call is an *internal* call. All function calls can be viewed as nodes of a binary tree (every node is a leaf or has two children) whose leaves and internal nodes correspond to calls to  $\mathcal{B}$ . It is easy to see, by induction, that the number of internal nodes equals the number of leaves minus one.

Since we have  $cc'(X, Y)$  leaf subroutine calls, we have  $cc'(X, Y) - 1$  internal subroutine calls. A call  $\mathcal{B}(X, Y, D)$  takes at most  $O(|Y|)$  time (if  $\mathcal{B}(X, Y, D)$  calls  $\mathcal{B}(X', Y', D)$  then  $|Y'| < |Y|$ ) we get the desired time bound  $O(|Y| \cdot cc'(X, Y))$ . By Step 3, we conclude that the total running time is

$$O\left(\sum_{i=1}^n cc'(\{v_i\}, \{v_{i+1}, v_{i+2}, \dots, v_n\}) \cdot (n-i)\right) = O(cc(D) \cdot n).$$

■

THEOREM 8. *Algorithm  $\mathcal{A}$  is correct and its time and space complexities are  $O(n \cdot cc(D))$  and  $O(n^2)$ , respectively.*

**Proof.** The correctness and time complexity follows from the two lemmas above. The space complexity is dominated by the space complexity of Step 1,  $O(n^2)$ . ■

Since  $cc(D)$  may well be exponential, we may wish to generate only a restricted number  $k$  of cc-sets. Theorem 9 can be viewed as a result in fixed-parameter algorithmics [5] with  $k$  being a parameter.

**THEOREM 9.** *To output  $k$  cc-sets the algorithm  $\mathcal{A}$  requires  $O(n^3 + kn)$  time.*

**Proof.** We may assume that  $k$  is at most the number of cc-sets containing vertex  $v_1$  since otherwise the proof is analogous.

We consider the binary tree  $T$  introduced in the proof of Lemma 7 and prove our claim by induction on  $k$ . It takes  $O(n^3)$  time to perform Steps 1,2 and 3. It takes  $O(n)$  internal nodes of  $T$  to reach the first leaf of  $T$  and, thus, for  $k = 1$  we obtain  $O(n^3 + n)$  time. Assume that  $k \geq 2$ . Let  $x$  be the first leaf of  $T$  reached by  $\mathcal{A}$ , let  $y$  be the parent of  $x$  on  $T$ , let  $z$  be another child of  $y$  on  $T$  and let  $u$  be the parent of  $y$ . Observe that after deleting the nodes  $x$  and  $y$  and adding an edge between  $u$  and  $z$ , we obtain a new binary tree  $T'$ . By induction hypothesis, to reach the first  $k - 1$  leaves in  $T'$ , we need  $O(n^3 + (k - 1)n)$  time. To reach the first  $k$  leaves in  $T$ , we need to reach  $x$  and the first  $k - 1$  leaves in  $T'$ . Thus, we need to add to  $O(n^3 + (k - 1)n)$  the time required to visit  $x$  and  $y$  only, which is  $O(n)$ . Thus, we have proved the desired bound  $O(n^3 + kn)$ . ■

## 4 Implementation and Experimental Results

In order to test our algorithm  $\mathcal{A}$  for practicality we have implemented it and run it on several instances of DDG's of basic blocks. We have compared our algorithm with the state-of-the-art algorithm of Chen, Maskell and Sun [4] (the CMS algorithm) using their own implementation, but with the code for I/O constraint checking removed so as to ensure that their algorithm was not disadvantaged. Both algorithms were coded in C++ and all experiments were carried out on a 2 x Dual Core AMD Opteron 265 1.8GHz processor with 4 Gb RAM, running SUSE Linux 10.2 (64 bit).

Our first set of tests is based on C++ programs taken from the benchmark suites of MiBench [8] and Trimaran [16]. We used IMPACT (Trimaran's front end) to compile these programs and perform data flow analysis. From here we examined the control-flow graph for each program to select a basic block within a critical loop of the program (often this block had been unrolled to some degree to increase the potential for efficiency improvements).

ID	A	B	C	D	E
NV	35	42	26	39	45
NE	38	45	28	94	44
NS	139,190	4,484,110	5,891	3,968,036	1,466,961
CT					
$\mathcal{A}$	96	3,246	4	2,710	1,156
CMS	170	5,546	6	4,346	1,750

Table 1. DDG produced by benchmark programs

We considered basic blocks, ranging from 20 to 45 lines of low level, intermediate, code, for which we generated the DDGs. We then selected, from these DGGs, the non-trivial connected components on which to run our algorithms.

We give some preference to benchmarks which suite the intended application of the research taking our test cases from security applications including benchmarks for the Advanced Encryption Standard (B,C) and safety-critical software (A, E). We also include a basic example from the Trimaran benchmark suite: Hyper (D).

The results we have obtained are given in Table 1. In both Tables 1 and 2, NV denotes the number of vertices, ID identifies the benchmark, NS denotes the number of generated sets, NA number of arcs, and CT clock time in  $10^{-3}$  CPU seconds.

We also consider examples with worst-case numbers of cc-sets. Let, as in Theorem 4,  $\vec{K}_{p,q}$  denote the digraph obtained from the complete bipartite graph  $K_{p,q}$  by orienting every edge from the partite set of cardinality  $p$  to the partite set of cardinality  $q$ . By Theorem 4 the digraphs  $\vec{K}_{a,n-a}$  with  $|n-2a| \leq 1$  have the maximum possible number of cc-sets. Our experimental results for digraphs  $\vec{K}_{a,n-a}$  with  $|n-2a| \leq 1$  are given in Table 2.

The above results demonstrate that our algorithm outperforms the CMS algorithm on both the benchmark examples and the extreme cases with large number of cc-sets.

## 5 Connected Induced Subgraphs Generation Algorithm

Let  $G$  be a connected (undirected) graph with vertex set  $V(G) = \{v_1, v_2, \dots, v_n\}$  and let  $G$  have  $m$  edges. For a vertex  $x \in V(G)$  and a set  $X \subseteq V(G)$ , let  $N(x) = \{z \in V(G) : xz \in E(G)\}$  and  $N(X) = \cup_{x \in X} N(x) \setminus X$ . The following is an algorithm,  $\mathcal{C}$ , for generating all connected induced subgraphs of

NV	15	16	17	18
NE	56	64	72	81
NS	32,400	65,041	130,322	261,139
CT				
$\mathcal{A}$	16	23	60	113
CMS	30	56	114	240
NV	19	20	21	22
NE	90	100	110	121
NS	522,772	1,046,549	2,094,102	4,190,231
CT				
$\mathcal{A}$	253	513	1,048	2,156
CMS	540	1,080	2,166	4,086

Table 2. Digraphs with maximum number of cc-sets

$G$ , or equivalently all sets  $Q$  which induce connected subgraphs of  $G$ . Such sets are called *connected*.

**Step 1:** For each  $i = 1, 2, \dots, n$  do the following. Set  $X := \{v_i\}$  and  $Y := \{v_{i+1}, v_{i+2}, \dots, v_n\}$ . Initiate the set  $N_X$  as  $N_X := N(X) \cap Y$ .

**Step 2 (subroutine  $\mathcal{D}$ ):** *Comment:*  $\mathcal{D}$  finds all connected sets  $Q$  in  $D$  such that  $X \subseteq Q \subseteq X \cup Y$ .

**(2a):** If  $N_X = \emptyset$  then return the connected set  $X$  (and stop).

**(2b):** If  $N_X \neq \emptyset$ , then let  $v \in N_X$  be arbitrary.

**(2c):** *Comment:* In this step we will find all connected sets  $S$  such that  $X \subseteq S \subseteq (X \cup Y)$  and  $v \in S$ . Set  $N_{X,0} := N_X$ ,  $X_0 := X$  and  $Y_0 := Y$ . Remove  $v$  from  $Y$  and  $N_X$ , and add it to  $X$ . For every  $u \in Y \setminus N_X$  check whether  $u$  has an edge to  $v$  and if it does then add it to  $N_X$ .

Make a recursive call to subroutine  $\mathcal{D}$ . *Comment:* we consider the new  $X$  and  $Y$ .

Change  $N_X$ ,  $X$ , and  $Y$  back to their original state by setting  $N_X := N_{X,0}$ ,  $X := X_0$ , and  $Y := Y_0$ .

**(2d):** *Comment:* In this step we will find all connected sets  $S$  such that  $X \subseteq S \subseteq (X \cup Y)$  and  $v \notin S$ . Remove  $v$  from  $Y$  and remove  $v$  from  $N_X$ .

Make a recursive call to subroutine  $\mathcal{D}$ .

Change  $Y$  back to its original state by adding  $v$  back to  $Y$ . Also change  $N_X$  back to its original state by adding  $v$  to it.

Similarly to Theorem 8, one can prove the following:

**THEOREM 10.** *Let  $c(G)$  be the number of connected induced subgraph of a connected graph  $G$ . Algorithm  $\mathcal{C}$  is correct and its time and space complexities are  $O(n \cdot c(G))$  and  $O(n + m)$ , respectively.*

## 6 Discussions and Open Problems

Let  $Z$  be the sum of the sizes of all cc-sets of a connected acyclic digraph  $D$ . We conjecture that  $Z = \Theta(n \cdot cc(D))$ . If this is true, then our algorithm  $\mathcal{A}$  is optimal with respect to the running time. If the conjecture is wrong, it would be interesting to obtain an optimal  $O(Z)$ -algorithm.

Our computational experiments show that  $\mathcal{A}$  performs well and is of definite practical interest. We have tried various heuristic approaches to speed up the algorithm in practice, but all approaches were beneficial for some instances and inferior to the original algorithm for some other instances. Moreover, no approach could significantly change the running time. The algorithm was developed independently from the CMS algorithm. However, the two algorithms are closely related, and work continues to isolate the implementation effects that give the performance differences.

It is not hard to modify  $\mathcal{A}$  such that the new algorithm will generate all convex subgraphs of an acyclic digraph  $D$  in time  $O(n \cdot co(D))$ , where  $co(D)$  is the number of convex subgraphs in  $D$ . This and other algorithms will be studied in our future publications. Let  $Z'$  be the sum of the orders of all convex subgraphs of an acyclic digraph  $D$ . We conjecture that  $Z' = \Theta(n \cdot co(D))$ .

**Acknowledgements.** We are grateful to the authors of [4] for helpful discussions and for giving us access to their code allowing us to benchmark our algorithm against theirs. Research of Gregory Gutin and Anders Yeo was supported in part by an EPSRC grant. Research of Gutin was also supported in part by the IST Programme of the European Community, under the PASCAL Network of Excellence, IST-2002-506778.

## BIBLIOGRAPHY

- [1] ARM, [www.arm.com](http://www.arm.com)
- [2] D. Avis and K. Fukuda, Reverse search for enumeration. *Discrete Appl. Math.* **65** (1996), 21-46.
- [3] J. Bang-Jensen and G. Gutin, *Digraphs: Theory, Algorithms and Applications*. Springer-Verlag, London, 2000, 754 pp.

- [4] X. Chen, D.L. Maskell, and Y. Sun, Fast identification of custom instructions for extensible processors. *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.* **26** (2007), 359–368.
- [5] R.G. Downey and M.R. Fellows, *Parameterized Complexity*, Springer–Verlag, New York, 1999.
- [6] H.N. Gabow and E.W. Myers, Finding All Spanning Trees of Directed and Undirected Graphs. *SIAM J. Comput.* **7** (1978), 280–287.
- [7] G. Gutin and A. Yeo, On the number of connected convex subgraphs of a connected acyclic graph. Submitted.
- [8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, MiBench: A free, commercially representative embedded benchmark suite, In *Proceedings of the WWC-4, 2001 IEEE International Workshop on Workload Characterization*, (2001), 3–14.
- [9] S. Kapoor, V. Kumar and H. Ramesh, An Algorithm for numerating All Spanning Trees of a Directed Graph. *Algorithmica* **27** (2000), 120–130.
- [10] S. Kapoor and H. Ramesh, Algorithms for enumerating all spanning trees of undirected and weighted graphs. *SIAM J. Comput.* **24** (1995), 247–265.
- [11] D.L. Kreher and D.R. Stinson, *Combinatorial Algorithms: Generation, Enumeration and Search*, CRC, 1999.
- [12] G.J. Minty, A simple algorithm for listing all trees of a graph. *IEEE Trans. Circuit Theory* **CT-12** (1965), 120.
- [13] MIPS, [www.mips.com](http://www.mips.com)
- [14] R.C Read and R.E Tarjan, Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks* **5** (1975), 237–252.
- [15] Tensilica, [www.tensilica.com](http://www.tensilica.com)
- [16] The Trimaran Compiler Infrastructure, [www.trimaran.org](http://www.trimaran.org)
- [17] A. Shioura and A. Tamura, Efficiently scanning all spanning trees of an undirected graph. *J. Operation Research Society Japan* **38** (1995), 331–344.
- [18] A. Shioura, A. Tamura and T. Uno, An optimal algorithm for scanning all spanning trees of undirected graphs. *SIAM J. Comput.* **26** (1997), 678–692.

G. Gutin, A. Johnstone, J. Reddington,  
 E. Scott, A. Soleimanfallah, and A. Yeo  
 Department of Computer Science  
 Royal Holloway University of London  
 Egham, Surrey TW20 0EX, UK

{G.Gutin,A.Johnstone,J.Reddington,E.Scott,A.Soleimanfallah,A.Yeo}@rhul.ac.uk