Complexity of Computing Convex Subgraphs in Custom Instruction Synthesis

Joseph Reddington and Kubilay Atasu

Abstract—Synthesis of custom instruction processors from high-level application descriptions involves automated evaluation of data-flow subgraphs as custom instruction candidates. A subgraph $S$ of a graph $D$, is convex if no two vertices of $S$ are connected by a path in $D$ that is not also in $S$. An algorithm for enumerating all convex subgraphs of a directed acyclic graph (DAG) under input, output, and forbidden vertex constraints was given in [1]. We show that this algorithm makes no more than $O(|V(D)|^{|N_{in}| + |N_{out}| + 1})$ recursive calls, where $|V(D)|$ is the number of vertices in $D$, and $N_{in}$ and $N_{out}$ are input and output constraints respectively. Therefore, when $N_{in}$ and $N_{out}$ are constants, the algorithm given in [1] is of polynomial complexity.

Furthermore, a convex subgraph $S$ is a maximal convex subgraph if it is not a proper subgraph of some other convex subgraph, assuming that both are valid under forbidden vertex constraints. The largest maximal convex subgraph is called the maximum convex subgraph. There exist popular algorithms that enumerate maximal convex subgraphs [2], [3], [4], [5], all of which have exponential worst-case time complexity. This work shows that although no polynomial-time maximal convex subgraph enumeration algorithm can exist, the related maximum convex subgraph problem can be solved in polynomial time.

Index Terms—Complexity, Subgraph Enumeration

I. INTRODUCTION AND RELATED WORK

Automated synthesis of custom instruction processors from high-level application descriptions has been an active research domain for the last decade. The automated techniques typically rely on compiler infrastructures to extract the source-level data flow graphs. Subgraphs of these graphs are evaluated as custom instruction candidates. There are potentially an exponential number of possible subgraphs, so much work in the area has focused on restricting the size of the set of ‘candidate instructions’ without reducing the potential quality.

The traditional approach in custom instruction synthesis enumerated the subgraphs by restricting the maximum number of input and output operands that custom instructions can have to the available register file ports [1], [6], [7], [8], [9], [10], [11]. Additional heuristic methods that generate custom instructions without exhaustive enumeration exist [12], [13], [14]. In [15], Ahn et al. describe a dynamic-programming method that can compute input/output constrained subgraphs that optimize an additive merit function in polynomial-time.

Recently, a class of subgraphs, called ‘maximal convex subgraphs’ received considerable attention [2], [3], [4], [5]. It is shown in [3] that the subgraph that offers the highest speed-up can be found by enumerating maximal convex subgraphs and applying a resource constrained scheduling on each such subgraph to pipeline the register file accesses. The number of subgraphs that should be enumerated are of exponential order and the scheduling problem is NP-Hard in the general case. Therefore, a heuristic scheduling algorithm is used in [3] to reduce the complexity. Computing the maximum convex subgraphs is motivated by the experimental evidence provided in [4], [5], [16], where experiments on benchmarks with relatively large basic blocks showed that in most cases the subgraphs that offered the highest speed-up were maximum convex subgraphs.

II. NOTATION AND THEORETICAL TERMS

We define a directed graph $D$ to be a finite set of vertices $V(D)$, and a set of directed edges $E(D) \subseteq V(D) \times V(D)$, which represent data dependencies between operations.

A directed graph $D'$ is a subgraph of $D$ if $V(D') \subseteq V(D)$ and $E(D') \subseteq E(D)$. A subgraph $D'$ induced by $A \subseteq V(D)$ has $V(D') = A$, and $E(D')$ is the set of all edges in $E(D)$ whose endpoints are both members of $A$.

DEFINITION 1: For $B \subseteq V(D)$ and $b \in B$, if there is no edge $(a, b) \in E(D)$, $a \in B$, then we say that $b$ is a source vertex of $B$. Likewise, if there is no edge $(b, a) \in E(D), a \in B$, then we say that $b$ is a sink vertex of $B$.

Let $F(D) \subseteq V(D)$ denote some set that includes the source and sink vertices of $V(D)$. We call $F(D)$ the forbidden vertices. Forbidden vertices are those vertices that cannot be
included in custom instructions due to the limitations of the custom processor architecture, due to the limitations of the custom datapath or by the choice of the custom processor designer. Examples include floating point operations, division operations, load/store operations, and branch operations that can be expensive to replicate in custom datapaths.

DEFINITION 2: A path is a sequence \( n_0, n_1, \ldots, n_k \) such that: for \( 1 \leq i \leq k \), \( (n_{i-1}, n_i) \in E(D) \) and each vertex in the sequence is distinct.

We say that \( n_0, n_1, \ldots, n_k \) is a path from \( n_0 \) to \( n_k \) or a \( n_0 \rightarrow n_k \) path. Moreover, for vertex sets \( A, B \), with \( n_0 \in A \) and \( n_k \in B \), we say that the path \( n_0, n_1, \ldots, n_k \) is a path from \( A \) to \( B \), or \( A-B \) path.

For a vertex \( v \in V(D) \), let desc\( (v, D) \) denote the descendants of \( v \), i.e., the set of vertices \( m \) such that there is a \( v-m \) path in \( D \). Similarly, let \( \text{desc}(A, D) \) denote the set of vertices \( m \) such that for some \( a \in A \) there is a \( a-m \) path in \( D \).

DEFINITION 3: A cycle is a sequence \( n_0, n_1, \ldots, n_k \) such that: for \( 1 \leq i \leq k \), \( (n_{i-1}, n_i) \in E(D) \), \( k \geq 2 \), the vertices \( n_0, \ldots, n_{k-1} \) are distinct, and \( n_0 = n_k \).

A directed graph is acyclic if it contains no cycles. We will refer to a directed acyclic graph as a DAG.

DEFINITION 4: A set of vertices \( V_a \) is convex in \( D \) if \( V_a \cap F(D) = \emptyset \) and every path in \( D \) between two vertices in \( V_a \) is in the subgraph induced by \( V_a \).

DEFINITION 5: A set of vertices \( V_a \) is a maximal convex set in \( D \) if it is valid, convex in \( D \), and there is no \( x \in V(D) \setminus V_a \) such that \( V_a \cup \{x\} \) is a valid convex set in \( D \).

DEFINITION 6: Given a set \( A \subset V(D) \), a vertex \( v \) is an input vertex of \( A \) if \( v \notin A \), and there is an edge \((v, a) \in E(D)\) such that \( a \in A \). The set of input vertices of a set \( A \) in \( D \) is denoted by \( \text{IN}(A, D) \).

DEFINITION 7: Given a set \( A \subset V(D) \) a vertex \( a \) is an output vertex of \( A \) if \( a \in A \), and there is an edge \((v, a) \in E(D)\) such that \( v \notin A \). The set of output vertices of a set \( A \) in \( D \) is denoted by \( \text{OUT}(A, D) \).

In Fig. 1, the set \{4, 7, 8\} is convex, whereas the set \{4, 7, 9\} is not because the path 4, 6, 9 is not in the subgraph induced by \{4, 7, 9\}. The set \{4, 5, 7, 8\} is a maximal convex set in Fig. 1, whereas the set \{7, 8, 9\} is not because there is a convex superset (e.g., \{5, 7, 8, 9\}). Finally, \( \text{IN}((4, 7, 8)) \) is \{1, 2, 3, 5\} in Fig. 1, whereas \( \text{OUT}((4, 5, 7, 8)) \) is \{4, 7, 8\}.

Given a DAG \( D \), a convex vertex set \( C \subseteq V(D) \), and I/O constraints of \( N_{in} \) and \( N_{out} \), if we have either \( |\text{OUT}(C, D)| > N_{out} \) or \( |\text{IN}(C, D)| > N_{in} \), then we say that \( C \) does not satisfy I/O constraints.

A convex subgraph of \( D \) is induced by a convex subset of the vertices of \( D \). A maximal convex subgraph of \( D \) is induced by a maximal convex subset of the vertices of \( D \). We will refer to convex subgraphs when discussing results but use convex sets in our proofs to simplify notation.

We say a set \( I \subseteq V(D) \) of vertices is independent in \( D \) if for every pair of vertices in \( I \), there is no edge connecting the two; furthermore \( I \) is a maximal independent set if there is no independent set for which \( I \) is a proper subset.

A vertex \( v \in V(D) \) can be associated with a non-negative integer weight \( W(v) \in \mathbb{Z} \), which can represent the software latency of \( v \), i.e., the time in clock cycles that it takes to execute \( v \) on the pipeline of a reference processor. We say a set \( M \subseteq V(D) \) of vertices is a maximum (weighted) convex set if \( M \) is a maximal convex set in \( D \) and if there is no maximal convex set \( M_L \subseteq V(D) \) for which \( \sum_{v \in V(M_L)} W(v) > \sum_{v \in V(M_M)} W(v) \). A maximum (weighted) convex subgraph of \( D \) is induced by a maximum (weighted) convex subset of the vertices of \( D \). In the rest of the text, we omit the word “weighted” for brevity, and imply maximum weighted convex set/subgraph when we say maximum convex set/subgraph.

III. ENUMERATING CONVEX SUBGRAPHS UNDER I/O CONSTRAINTS

Several algorithms for enumerating all convex subgraphs of a DAG \( D \) under input/output and forbidden vertex constraints have been presented with polynomial time complexity in terms of the number of vertices in \( D \), such algorithms also take the parameters \( N_{in} \) and \( N_{out} \) as constants, which is of course true for any single use of the algorithm. Bonzini and Pozzi [8] produced an algorithm with running time \( O(|V(D)|^{N_{in}+N_{out}+1}) \). However, this algorithm does not generate all valid convex subgraphs; for more discussion, see [17]. Recently, Gutin et al [11], produced an algorithm of running time \( O(|V(D)|^{N_{out}+|S(D)|}) \) where \( |S(D)| \) is the set of all valid subgraphs. Several algorithms with related approaches were given in [17], [18]. We will show that the algorithm of [11], [6] also belongs in this grouping.

The exhaustive algorithm (Algorithm 1) was created by Atasu et al [6] and extended in [1]. It finds all convex subgraphs in a dagraph by recursively building convex sets and effectively pruning a search tree. Computational overhead is reduced by pruning recursive calls if their recursive descendants do not contain any valid convex subgraphs. It was the first algorithm to exhaustively enumerate all convex subgraphs in a given DAG without having to examine all possibilities.

The fundamental insight of the exhaustive algorithm is that if a subgraph induced by the set \( X \) is not convex in a DAG \( D \), and \( u \in V(D) \setminus X \) is topologically later than all vertices in \( X \) then the subgraph induced by \( X \cup \{u\} \) is also not convex in \( D \). If the subgraph represented by a node in the search tree is not convex in \( D \), then no subgraphs represented by descendants of that node in the search tree will be convex, and the search tree rooted at that vertex can be pruned. Furthermore, if the vertices are examined in reverse topological order, then vertices that are currently outputs of the convex subgraph \( Y \) will always be outputs to any recursive descendant of \( Y \) because no vertices can be added ‘below’ them. Thus recursive branches can also be pruned on the grounds of output vertices.

The exhaustive algorithm’s restriction by input constraint is less efficient than the output restriction. In [6], there was
a simple input check for each convex subgraph before it was stored. However, in [1] an improved input checking algorithm was proposed that was able to cull more of the unnecessary branches. This was achieved by maintaining a list of those input vertices that could never be added to the current selection, either because they were forbidden vertices or because the vertex had already been considered for inclusion by a recursive ancestor. If there were more of these inputs than allowed by the input constraint, then the recursive branch could be pruned (see the call to externalInputs() function in Algorithm 1). These pruning criteria allow exhaustive to avoid much of the computation of a brute force algorithm.

Our analysis considers only the improved variant of the exhaustive algorithm presented in [1]. We posit that it is the improved pruning criteria that allowed the algorithm to achieve the polynomial complexity bound.

### A. Recursive calls made by the algorithm

This section shows that exhaustive algorithm will make no more than

$$\sum_{b=1}^{N_{in} + N_{out} + 1} \binom{|V(D)|}{b}$$

recursive calls on an input graph.

Consider that the calls made to the recursive function of the exhaustive algorithm form a binary tree, $T$. Each node $t$ in $T$ represents a call that has selection $X_t$, a forbidden vertex set $F_t$ and a vertex $v$. Unless $t$ is a leaf of $T$, it will have a child $t^F$, which represents the recursive descendant in which $v$ has been forbidden. It may also have a child $t^X$ that represents the recursive descendant in which $v$ joins $X$.

For all nodes $t$ in $T$, if there is a path from the respective $v$ to $X$ in the DAG, then we colour the edge in the tree $T$ from $t$ to $t^F$ red and the edge from $t$ to $t^X$ blue. However, if there is no path from the respective $v$ to $X$ in the DAG, then we colour the edge in $T$ from $t$ to $t^F$ blue and the edge from $t$ to $t^X$ red. Clearly there can be no edges in the Tree coloured both red and blue, and there can be no more than one blue edge or one red edge from each node. It is not difficult to see that every edge in the call tree is now coloured red or blue.

By Lemmas 1 and 2 we can see that if $t$ is a node in $T$ and $t_{red}$ is the node reached by following a red edge from $t$, then $|OUT(X_t, D)| + |IN(X_t, D) \cap F| < |OUT(X_{t_{red}}, D)| + |IN(X_{t_{red}}, D) \cap F|$. Then there cannot be more than $N_{in} + N_{out}$ red edges on the path between the root of $T$ and some node $c$ because then $|OUT(X_c, D)| + |IN(X_c, D)| > N_{in} + N_{out}$ and one of the I/O checks would have prevented this level of recursion.

So every node in $T$ can be reached from the root vertex by using no more than $N_{in} + N_{out}$ red edges. Then, by Lemma 3, there can only be $\sum_{b=1}^{N_{in} + N_{out} + 1} \binom{|V(D)|}{b}$ nodes in the tree, providing an upper bound to the number of recursive calls by the exhaustive algorithm. Thus, the number of recursive calls is of $O(\binom{|V(D)|}{N_{in} + N_{out} + 1})$.

For general DAGs, each instance of the recursive call can be implemented to run in linear time. For graphs with bounded in-degree, such as those extracted from compiler intermediate representations, each instance of the call can be implemented to run in constant time [1], [6]. Therefore, the exhaustive algorithm is of time complexity $O(\binom{|V(D)|}{N_{in} + N_{out} + 1})$ for graphs with bounded in-degree, and thus is theoretically comparable to the algorithms of [11] and [18].

**Lemma 1**: Let $X, F \subseteq V(D)$, such that $F \cap X = \emptyset$ and $X$ is convex in $D$. If $i \in V(D) \setminus (X \cup F)$ is topologically earlier than all vertices in $X$, and $i \notin.desc(X, D)$ then $|OUT(X, D)| + 1 = |OUT(X \cup \{i\}, D)|$.

**Proof**: $i$ is not in $F$ so it cannot be the sink vertex of $D$. Then there is some edge $(i, j) \in E(D)$. Because $i \notin.desc(X, D)$, then $j \notin X$ and so $i$ is an output vertex of $X \cup \{i\}$. All other output vertices of $X \cup \{i\}$ must also be output vertices of $X$ and so $|OUT(X, D)| + 1 = |OUT(X \cup \{i\}, D)|$.

**Lemma 2**: Let $X, F \subseteq V(D)$, such that $F \cap X = \emptyset$ and $X$ is convex in $D$. If $i \in V(D) \setminus (X \cup F)$ is topologically earlier than all vertices in $X$, and $i \notin.desc(X, D)$, and $X \cup \{i\}$ is convex in $D$, then $|IN(X, D) \cap F| + 1 = |IN(X, D) \cap (F \cup \{i\})|$.

**Proof**: $i \in.desc(X, D)$ so $i$ must have at least one descendant that is either in $X$ or is in desc$(X, D)$. Let $j$ be one such descendant. If $j \notin X$, then $X \cup \{i\}$ is not convex in $D$. Then $j \in X$. Then $i \in IN(X, D)$. Since $i \notin F$ we have $|IN(X, D) \cap F| + 1 = |IN(X, D) \cap (F \cup \{i\})|$.

**Lemma 3**: Given a binary tree $T$ of depth $n$, the number of vertices in $T$ that can be reached from the root of $T$ by taking no more than $L$ left edges is $\sum_{b=1}^{L+1} \binom{n}{b}$.

**Proof**: Because $T$ is a tree, every vertex in $T$ can be uniquely identified by the sequence of left and right edges used to reach it from the root. We are interested in those sequences that contain no more than $L$ left links, and we note that the binomial coefficient $\binom{n}{b}$ will return the number of sequences of length $a$ that contain exactly $b$ left links. Thus $\binom{n}{b}$ is the number of vertices at level $a$ in $T$ that are reachable by exactly $b$ left links. The total number of vertices in $T$ reachable by no more than $L$ left edges is $\sum_{b=0}^{L} \binom{n-1}{b+1}$.

The ‘Christmas Stocking Theorem’ gives $\sum_{a=0}^{n-1} \binom{n}{a} = \binom{n+1}{L+1}$, so the number of vertices in $T$ that can be reached from the root of $T$ by taking no more than $L$ left edges and unlimited right edges is $\sum_{b=1}^{L+1} \binom{n}{b}$.
IV. ENUMERATING MAXIMAL CONVEX SUBGRAPHS

The maximal convex subgraph enumeration problem was first targeted by Pothineni et al. [2] who define an incompatibility graph \( D' \) as in Definition 8. Pothineni et al. proposed that enumerating maximal convex subgraphs of \( D \) is equivalent to enumerating maximal independent sets of \( D' \).

**DEFINITION 8:** Given a DAG \( D \), which has forbidden nodes \( F \subset V(D) \), create an undirected graph \( D' \) called the incompatibility graph such that \( V(D') = V(D) \setminus F \), and \( E(D') \) is the set of edges \( (a,b) \) where \( a,b \in V(D') \) and there is a path in \( D \) from \( a \) to \( b \) that includes a \( c \in F \).

Verma et al. [3] formally proved an equivalent result: they define a cluster graph, which happens to be the complement graph of an incompatibility graph for any given DAG and go on to show that enumerating maximal cliques in a cluster graph is equivalent to enumerating maximal convex subgraphs of \( D \). Given that the cluster graph is the complement graph of the incompatibility graph for any given DAG, and that clique enumeration in a graph \( D \) is equivalent to independent set enumeration in \( D' \)'s complement, the two methods are equivalent. Corollary 1 follows from the Lemmas in [3]. For clarity, we will adopt the model of enumerating maximal independent sets of an incompatibility graph as in [2].

**Corollary 1:** a) If \( X \) is a valid maximal convex set in \( D \) then it is also a maximal independent set in \( D' \)'s incompatibility graph \( D' \). b) If \( X \) is a maximal independent set in the incompatibility graph \( D' \) of \( D \), then it is also a valid maximal convex set in \( D \).

**DEFINITION 9:** A directed graph \( D \) is transitive if \((a,b),(b,c) \in E(D)\) implies \((a,c) \in E(D)\).

**DEFINITION 10:** A comparability graph is a graph that has a transitive orientation: each edge of the graph can be assigned a one-way direction such that the adjacency relation of the resulting oriented graph is transitive [19].

**Lemma 4:** For any DAG \( D \), the incompatibility graph \( D' \) of \( D \) is a comparability graph.

**Proof:** We first create a directed graph \( D'' \) that defines an orientation for the edges of \( D' \). Given a DAG \( D \), which has forbidden nodes \( F \subset V(D) \), and the incompatibility graph \( D' \) of \( D \), we create a new graph \( D'' \) such that \( V(D'') = V(D') \), and \( E(D'') \) is the set of directed edges \( (a,b) \) where there is a path in \( D \) from \( a \) to \( b \) that includes a member of \( F \). Note that for every undirected edge \((a,b)\) in \( E(D') \), there exists a directed edge \((a,b)\) in \( E(D'') \).

Given the edges \((a,b),(b,c) \in E(D'')\) for any \( a,b,c \in V(D'') \), there must be an \( a \) to \( b \) and \( b \) to \( c \) path in \( D \), both of which contain at least one forbidden vertex. Then, there is an \( a \) to \( c \) path in \( D \) that contains at least two forbidden vertices and hence an edge \((a,c)\) in \( E(D'') \) exists. So \( D'' \) defines a transitive orientation for the edges of \( D' \). Thus, \( D' \) is a comparability graph.

It is well known that the number of maximal independent sets of a graph with \( n \) nodes is upper bounded by \( 3^{n/3} \) [20]. Note that given an incompatibility graph \( D' \) derived from \( D \), \( n = |V(D) \setminus F| \). Note also that an exhaustive maximal independent set enumeration algorithm has to produce all maximal independent sets. Therefore, the complexity of enumeration has to be at least as high as the number of independent sets.

The time complexity of the Bron-Kerbosch algorithm [21] is \( O(3^{n/3}) \), effectively reaching the ideal bound on complexity.

We are not aware of an existing work that derives the complexity of independent set enumeration on comparability graphs. The closest work we are aware of is given by [4], [5], where Atasu et al. enumerate all maximal convex sets in \( O(2|F|) \) computational steps. Various clustering and constraint propagation techniques that can significantly reduce the exponential enumeration complexity are also described in [4], [5].

To see that no polynomial-time maximal convex subgraph enumeration algorithm can exist, consider the example graph \( D \) (Figure 2), which has \( N \) connected components. Each connected component contains one forbidden vertex, and each forbidden vertex (shown in gray) has exactly one ancestor and exactly one descendant that are not forbidden (shown in white). The vertices \( v \in V(D) \) are associated with positive weights \( W(v) \). Figure 3, shows the incompatibility graph \( D' \) of Pothineni et al. [2] for the same example. In Figure 3, vertex \( i \) is incompatible with vertex \( 2N+i \) for \( i \in \{1..N\} \), and compatible with the rest of the vertices. While enumerating the maximal independent sets of \( D' \), one can choose either vertex \( i \) or node \( 2N+i \) for \( i \in \{1..N\} \), and each such choice results in a distinct maximal independent set. Therefore, the number of maximal independent sets of the graph \( D' \) shown in Figure 3 is \( 2^N \). Since the number of maximal independent sets is exponential in the number of vertices, we conclude that no polynomial-time maximal convex subgraph enumeration algorithm exists for this example and so in the general case.

V. COMPUTING MAXIMUM CONVEX SUBGRAPHS

We now show that the problem of finding the maximum convex subgraph of a DAG is equivalent to the problem of finding the maximum independent set of a comparability graph, which has a well studied solution in polynomial time.

**Theorem 1:** The maximum convex subgraph problem can be solved in polynomial time on any DAG \( D \).

**Proof:** A maximal convex set of \( D \) is also a maximal independent set of the incompatibility graph \( D' \), and vice versa.
Similarly, a maximum convex set of $D$ is also a maximum independent set of the incomparability graph $D'$, and vice versa.

By Lemma 4, $D'$ is a comparability graph. Every comparability graph is a perfect graph (see [19], Chapter 5, pp. 133). Creation of $D'$ from $D$ takes $O(|V(D)|^2)$ time and the maximum independent set problem can be solved in polynomial time on perfect graphs (see for example [22], Chapter 9, pp. 273–303). By definition, each maximum convex set induces a maximum convex subgraph; thus, the maximum convex subgraph problem can be solved in polynomial time on any given DAG $D$.

A polynomial-time method for finding the maximum independent set of a comparability graph $D'$ can be found in [19], Chapter 5, pp. 134–135). The method transforms the transitive orientation $D''$ of $D'$ into a transportation network by adding two new vertices $s$ and $t$ and by adding the edges $(s, x)$ and $(y, t)$ for each source vertex $x$ and sink vertex $y$ of $D''$. A lower capacity of $W(v)$ can be assigned to each vertex $v \in V(D'')$, and a minimum flow algorithm can be called to compute an integer valued flow. It can be shown that the value of the minimum valued flow will be equal to the size of the maximum independent set. Such a minimum flow algorithm can be implemented to run in polynomial time (see [23]).

The maximum convex subgraph problem is not only important from a theoretical point of view, but also has a significant practical value. It is shown in [5] that by iteratively computing maximum convex subgraphs to cover the vertices of an input DAG, high quality custom instructions can be generated. Although an exponential-time enumeration algorithm is used in [5] to compute the maximum convex subgraphs, our work provides a polynomial-time solution and reduces the overall computational complexity of custom instruction generation.

VI. CONCLUSION AND FUTURE WORK

This work has made three contributions to the area of convex set enumeration. Firstly, the exhaustive algorithm of Pozzi, Atasu and Ienne [1] has a time complexity of $O(|V(D)|^{|N_{in}+N_{out}|+1})$ (Section III). No example has been found that requires $O(|V(D)|^{|N_{in}+N_{out}|+1})$ time to process. Thus, future work could either isolate such an example or further reduce the bound. Further analysis could create a ‘taxonomy’ of methods in terms of time and space complexity.

Secondly, we have shown that the maximal convex subgraph enumeration problem has exponential time complexity in the general case (Section IV); and so a polynomial algorithm cannot exist without the addition of further constraints. Clear future work in this area includes the identification of classes of input graphs that do not require exponential time complexity, with a view to producing algorithms for those cases.

Thirdly, we have shown that the closely related maximum convex subgraph problem can be solved in polynomial-time by transforming the problem into a minimum flow problem (Section V). This enables a polynomial-time overall solution for a recent custom instruction generation approach [4], [5].

ACKNOWLEDGMENTS

The authors would like to thank A.-M. Cromlechner, IBM Research - Zurich, and Prof. C. Öztrum from Boğaziçi University, Istanbul for their valuable comments.

REFERENCES


