Improvements to instruction identification for custom instruction set design

Joseph Reddington

Thesis submitted to the University of London

for the degree of Doctor of Philosophy

July 30, 2009

Department of Computer Science

Egham, Surrey TW20 0EX, England

Declaration

These doctoral studies were conducted under the supervision of Adrian Johnstone and Elizabeth Scott. The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Department of Computer Science as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Joseph Reddington

September 2008

Abstract

High performance embedded systems are required to satisfy tight power, unit cost, and performance criteria. An embedded system will only run one application in its lifetime, so in principle there are opportunities to 'tune' the embedded system to the target application. In today's highly competitive marketplace such tuning must be achievable without large-scale manual intervention.

An obvious way of tuning embedded processor architectures is to remove unused registers and instructions thus saving hardware real estate. More interestingly, common sequences of instructions could be implemented as special custom instructions. Such instructions have the potential to reduce code size and instruction fetch time along with overall execution time.

This thesis describes new algorithms for the generation of candidate custom instructions; providing an analysis of weaknesses in current approaches and giving experimental evidence and proofs of correctness and time complexity.

Contents

List of Figures

List of Tables

List of Algorithms

List of Examples

Chapter 1

Introduction

This thesis presents techniques for use within an application specific processor design toolchain to efficiently create a library of candidate instructions that may be included in the instruction set of a custom processor.

1.1 Background

Modern embedded processors may be required to decode videos, process financial transactions, and process extremely large amounts of other data: this requires the highest performance whilst also meeting demanding cost and power constraints.

For a particular application, a new processor can be designed that is particularly optimised for that application. Successful application specific processors (ASPs) must deliver lower cost solutions than a general purpose embedded processor.

The non-recurring engineering costs for designing a hand-crafted ASP are approaching one hundred million US dollars [IL06], which can only be justified for very high volume products. The development of *customisable processors* over the last decade has been an attempt to bridge the gap between standard processors and hand-crafted ASPs by allowing a set of changes to a processor without the heavy costs of designing from scratch.

Modern embedded systems typically use Field Programmable Gate Arrays (FPGA) — a fixed array of gates and flip-flops with programmable interconnectivity. Early examples of such systems used standard processor chips supported by FPGAs containing application specific interfaces and hardware. As FPGA capacities increased it became possible to implement the processor directly within the FPGA. Customisable processors exploit the potential to extend the architecture directly by adding functional units to the data path and appropriate structures to control them.

A number of such customisable processors have been marketed. These include the ARM Optimo-DE [S:A], the MIPS Pro Series [S:M08], the Tensilica Xtensa [Gon00], the Molen [VWG+04], the Altera Nios II, the MIPS Core Extend, the STMicroelectronics ST 200 [FBF+00] and the ARC Arctangent, to name just a few.

1.1.1 Example

Customisable architectures are extensions to already available architectures. In this we shall use the simplescalar architecture [ALE02], a paper architecture that was developed for academic use, for our examples. The simplescalar code in Figure 1.1 is part of a heavily executed loop in a security benchmark. If a modified simplescalar processor were available that had additional custom functional units to perform instructions such as **laa** (load and add one) and **xas** (xor and store), then this modified processor could run the code shown in Figure 1.2 and would require only 8 clock cycles, rather than 11. This is a 27% performance improvement.

1.1.2 Automatic customisation of instruction sets

By extending the instruction set of a customisable processor in this way, a designer can have many of the advantages of a hand-crafted ASP for a fraction

```
4195264:   lw $5,0($30)
4195272:   addu $6,$0,$5
4195280:   sll $5,$6,0x2
4195288:   addu $4,$4,$5
4195296:   lw $5,0($4)
4195304:   xor $3,$3,$5
4195312:   sw $3,0($2)
4195320:   lw $3,0($30)
4195328:   addiu $2,$3,1
4195336:   addu $3,$0,$2
4195344:   sw $3,0($30)
4195352:   j 4002d8 <sha_transform__FP8SHA_INFO+0xe8>
```

Figure 1.1 Motivation example for customisation of instruction sets

```
4195264:   lw $5,0($30)
4195272:   addu $6,$0,$5
4195280:   sll $5,$6,0x2
4195288:   addu $4,$4,$5
4195296:   lw $5,0($4)
4195304:   xas 0($2),$3,$5
4195320:   laa $3,0($30)
4195344:   sw $3,0($30)
4195352:   j 4002d8 <sha_transform__FP8SHA_INFO+0xe8>
```

Figure 1.2 Code in Figure 1.1 modified using custom instructions

of the cost. However, adjusting an instruction set by hand is difficult and time consuming. In Chapter 3 we shall show that there may be millions of different potential custom instructions. Some instructions may even make other custom instructions completely or partially redundant. A designer must consider all of these issues and the way in which a new instruction set affects a target application that could be many thousands of instructions long. For a designer to choose several new instructions that provide some improvement is simple — however, extracting the maximum amount of improvement from an instruction set is extremely difficult.

There has been much recent research into developing tools to perform this *instruction set customisation* automatically.

Most methods for performing automatic customisation of an instruction set split the problem into the subproblems of (a) identifying all potential candidate instructions and of (b) selecting a set of candidate instructions that give the largest possible improvement, measured by some fitness criteria provided by the designer, to the original processor.

1.2 Modifying compilers to support instruction set customisation

Figure 1.3 shows the internal structure of a high performance compiler that has been augmented by the elements needed to support instruction set customisation, including candidate enumeration (the main topic of this thesis).

We use the existing compiler front end to parse and analyse the application source code yielding a Control Flow Graph (CFG, a representation of all potential control flow paths), a set of Data Dependence Graphs (DDGs, in which edges represent the flow of data between instructions), and a symbol table. We modify the compiler back end to extend instruction selection to the selected custom instructions.

Figure 1.3 Toolchain for instruction set customisation

1.2.1 Instruction selection

This thesis is mainly about instruction enumeration; however, we note here that there is an extensive literature on *instruction selection* — the second of the two subproblems.

The instruction selection process is passed a set of candidate instructions, a set of existing instructions, a set of fitness criteria, and a target application. The instruction selection process must find a set of candidate instructions that maximise the fitness criteria.

Candidate instructions cannot be considered separately because they may overlap, and hence make each other redundant. Each combination of instructions must be considered individually. For most real examples an exhaustive approach is impractical so a number of heuristics have been developed to attack the problem.

The problem of selecting the best subset of a set of candidates has been restated in different ways so that approaches from other branches of computer science can be used. However, each redefinition requires the abstraction of some detail of the design process. Early approaches have used simulated annealing [HD94], heuristics based on solutions to the subset-sub problem [CKY+99], or heuristics based on backtracking [GP03].

Several approaches have attempted to use linear integer programming — originally in [SIH+91] and later [ADO05] (extended in [ADM+07]). The approach of [ADO05] identifies candidates by first using linear integer programming, orders the candidates by use of a merit function and then chooses the first n instructions regardless of any interaction between them. The 'pseudo optimal' approach in [PAI06] takes account of interaction between instructions and provides a useful theoretical framework. However, its methods have a large computational overhead and it is impractical for examples of reasonable size. The same paper also proposes use of genetic algorithms with various seed sets.

1.2.2 Code generation

The code generator examines the CFG, the DDGs, and the customised instruction set. From these it generates an assembly language program that takes advantage of the newly introduced instructions.

The code generator must be automatically reconfigured, or generated, so that it can make use of the availability of new instructions.

A number of different approaches to automatic code generator generation have been presented. The majority of such methods parse syntax trees to produce assembler code.

Early work by Graham and Glanville [GG78] used LR parsing techniques for code generation. The work was extended by Ganapathi and Fischer in 1985 to include attribute grammars [GF85].

Twig [Tji85] is a tool for code-generator generation that combines tree parsing with dynamic programming and has been used to create code generators for the Vax architecture that use only 112 rules [App87].

In contrast to approaches like G&G and Twig, [HO82] showed that, by constructing an automaton to precompute the possible choices made by the compiler, they could shift the computation to compiler construction time providing there was space to store the precomputed data, the tables of which could be exponential in the number of sub-patterns and the arity of the operators. Later methods include **Burg** [PFH92] and **iburg** [FHP92].

Although the area is dominated by tree-parsing methods, there are several methods targeted at Directed Acyclic Graphs (DAGs). The methods published in [LB00] and [Ert99] work directly with DAGs to solve the problems inherent with tree parsing such as the inability of tree parsers to utilise instructions performing parallel computations.

1.3 Thesis overview

Chapter 2 shows that enumerating candidate instructions for a target application is equivalent to searching a collection of Directed Acyclic Graphs (DAGs) for vertex sets that satisfy certain conditions. The motivation for each of these conditions is discussed and the notation used in this thesis is given. Chapter 3 gives theoretical bounds on the number of candidate instructions and compares these bounds to the number of candidates in commercial benchmarks.

A review of previous work in the area is included in Chapter 4. Special attention is paid to the review of the **union** algorithm in Chapter 7. Implementations of some of these algorithms are used in experiments throughout this thesis to compare with our own work.

Chapters 5 and 6 present algorithms that represent a substantial step forward for candidate instruction enumeration.

Chapter 5 presents several algorithms for enumerating candidate instructions, and Chapter 6 presents several algorithms for the case in which register reads and writes are constrained.

Concluding remarks and a short discussion of future work are included in Chapter 8.

Each chapter includes the relevant experimental work and results, and details of the experimental setup used in this thesis are in the appendix, as are descriptions of each test case.

1.4 Summary

This chapter has introduced the instruction set customisation problem as part of a compiler toolchain. It is clear that libraries of candidate instructions are needed for instruction customisation, and must be efficiently generated.

Chapter 2

Notation and problem formulation

Given a set of basic instructions we wish to consider a set of useful custom instructions, obtained by combining basic instructions. This is achieved by forming convex sets of the vertices in a data-dependency graph (a vertex set A is convex if any vertex that is on a path between two vertices in A is also in A). In practice, a given application may force additional constraints on the candidate instructions and this chapter will discuss three examples of such constraints: forbidden operations; connectivity; and Input/Output constraints.

2.1 Notation and definitions

This chapter provides most of the terminology and notation used in this thesis along with several basic results.

2.1.1 Vertices and edges

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.

We say there is an $a - b$ edge in D if $(a, b) \in E(D)$.

A directed graph D' is a subgraph of D if $V(D') \subseteq V(D)$ and $E(D') \subseteq E(D)$

If $A \subseteq V(D)$, then D_A denotes the induced subgraph of A in D. That is, $V(D_A) = A, E(D_A) = \{(a,b)|(a,b) \in E(D) \text{ and } a,b \in A\}$.

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. The set of source vertices in B is denoted by source(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. The set of sink vertices in B is denoted by sink(B).*

2.1.2 Walks and paths

DEFINITION 2 *A* walk *is a sequence $n_0, n_1, \ldots, n_k$ such that: for $1 \leq i \leq k$, $(n_{i-1}, n_i) \in E(D)$.*

We say that a walk $n_0, n_1, \ldots, n_k$ contains the vertices $n_i, 0 \leq i \leq k$, and the edges (n_{i-1}, n_i), $1 \leq i \leq k$. For a walk W, $E(W)$ denotes the set of edges contained in W. If $k = 0$ then W is the empty walk from n_1 to itself.

DEFINITION 3 *A* path *is a walk in which each vertex in the walk is distinct.*

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

DEFINITION 4 *A* cycle *is a walk $n_0, n_1, \ldots, n_k$ such that $k \geq 2$, the vertices $n_0, \ldots, n_{k-1}$ are distinct, and $n_0 = n_k$.*

DEFINITION 5 *A directed graph is* acyclic *if it contains no cycles.*

The following well known result, proved in, for example, [BJG00], will be required.

Lemma 1 *Given a directed graph D with vertices $a, b \in V(D)$, if there is an $a - b$ walk, W, in D, then there must also be an $a - b$ path P in D, such that $E(P) \subseteq E(W)$.*

For any set of paths, $\mathcal{P}$, let $E(\mathcal{P}) = \bigcup\{E(P) : P \in \mathcal{P}\}$.

DEFINITION 6 *A set of vertices V_a is* convex *in D if every path in D between two vertices in V_a is also a path in the induced subgraph D_{V_a}.*

We let $\mathcal{S}(D)$ denote the set of subsets of $V(D)$ that are convex in D and note that $\emptyset \in \mathcal{S}(D)$.

Note, if $A \subseteq B \subseteq V(D)$ and A is convex in D then A is convex in D_B. The converse in not true in general, but is true if B is also convex in D. That is, if $A \subseteq B \subseteq V(D)$, B is convex in D, and A is convex in D_B, then A is convex in D.

DEFINITION 7 *The* convex closure *of a set of vertices A is the set of all vertices of $V(D)$ that lie on a walk between elements of A.*

DEFINITION 8 *Given a set A, let D'_A be a directed graph such that $V(D'_A) = V(D_A) = A$ and $(a,b) \in E(D'_A)$ if $(a,b) \in E(D_A)$ or $(b,a) \in E(D_A)$. Then A is* connected *if, given any two vertices $v, u \in A$, there is a walk from v to u in D'.*

We let $\mathcal{S}_c(D)$ denote the set of subsets of $V(D)$ that are connected and convex in D.

DEFINITION 9 *A set $A \subseteq B \subseteq V(D)$ is a* connected component *of B if A is connected in D and there is no set C such that $C \subseteq B$ is connected in D and $A \subset C$.*

Given a set $B \subseteq V(D)$ whose connected components are $C_1, \ldots, C_k$, we note that B is the disjoint union of $C_1, \ldots, C_k$. That is $C_i \cap C_j = \emptyset$ if $i \neq j$, and $B = C_1 \cup \ldots \cup C_k$. Moreover, any C_i is convex in D_B, and if B is convex in D, then C_i is convex in D.

It is then easy to check that if $A \in \mathcal{S}(D)$ and D has connected components $C_1, \ldots, C_k$, then A is the disjoint union of $A \cap C_1, \ldots, A \cap C_k$. Then there are

some $A_1, \ldots, A_k$ such that $A_i \in \mathcal{S}(D_{C_i}), A = A_1 \cup \ldots \cup A_k$, and that if we have $A_1, \ldots, A_k$ such that $A_i \in \mathcal{S}(D_{C_i})$, then $(A_1 \cup \ldots \cup A_k) \in \mathcal{S}(D)$. It follows that $\mathcal{S}_c(D) = \mathcal{S}_c(D_{C_1}) \cup \ldots \cup \mathcal{S}_c(D_{C_k})$.

2.1.3 Topological ordering

If D is acyclic then let $V(D) = \{v_1, \ldots, v_{n-1}, v_n\}$ such that $(v_a, v_b) \in E(D)$ implies that $a < b$. For any pair of vertices v_a, v_b, if $a < b$, then we say that v_a is topologically before v_b. If $a > b$, then we say that v_a is topologically after v_b. For a set $A \subseteq V(D)$, if there is no vertex in A that is topologically before vertex $a \in A$, then we say that a is the topologically first vertex in A. If there is no vertex in A that is topologically after a, then we say that a is the topologically last vertex in A.

2.1.4 Reachability functions

For a vertex $v \in V(D)$, let $dom(v, D)$ denote the set of vertices m such that $v \neq m$ and there is a $m - v$ path in D. Similarly, let $dom(A, D)$ denote the set of vertices $m \notin A$ such that there is an $m - a$ path in D for some $a \in A$.

For a vertex $v \in V(D)$, let $domBy(v, D)$ denote the set of vertices m such that $v \neq m$ and there is a $v - m$ path in D. Similarly, let $domBy(A, D)$ denote the set of vertices $m \notin A$ such that there is an $a - m$ path in D for some $a \in A$.

2.1.5 Complexity

In this thesis we compare the performance of a number of different algorithms. We are interested in both the theoretical worst-case bounds and the typical performance as measured on some set of examples.

From a theoretical perspective we are interested in the changes to the running time and memory requirements of an algorithm as the size of the input increases. The *time complexity* of an algorithm is a measure of the time re-

quired by an algorithm to process an input of given size. We will often write complexity instead of time complexity.

In this thesis, we use the standard 'big-O' notation when discussing complexity, which is defined by, for example [JS02], as,

$$O(f(n)) = \{g| \text{ for some } c, m, g(i) \leq cf(i), \text{ for all } i \geq m\}.$$

More informally, an algorithm has time complexity $O(g(n))$ for some $g(n)$ if its running time on inputs of size n is always less than a constant multiple of $g(n)$ when n is greater than some constant m.

2.1.6 Flow on networks

A network N is a directed graph associated with the function $c : V(N) \times V(N) \to \mathbb{R}$ such that $\forall(a, b)$, if $(a, b) \notin E(N)$, then $c(a, b) = 0$. We may write c_{ab} instead of $c(a, b)$.

A flow on a network is a function $f : V(N) \times V(N) \to \mathbb{R}$ such that $\forall(a, b)$, if $(a, b) \notin E(N)$, then $f(a, b) = 0$. We may write f_{ab} instead of $f(a, b)$.

A flow, f, is *feasible* if $0 \leq f_{ab} \leq c_{ab}$ for all $a, b \in V(N)$.

We define the function $balance(v, f)$ as

$$balance(v, f) = \left(\sum_{a \in V(N)} f_{va} \right) - \left(\sum_{a \in V(N)} f_{av} \right)$$

.

Let $s, t \in V(N)$. A feasible flow, f, on N is a *flow from s to t* if

1. $balance(s, f) = -balance(t, f) > 0$,
2. $balance(v, f) = 0$ for $v \neq s, t$.

We may call a flow from s to t, an $s - t$ flow.

The *value* of an $s - t$ flow is $balance(s, f)$.

For a given $s, t \in V(N)$, a *maximal* $s-t$ *flow* on N is a flow f from s to t such that there is no flow f' from s to t with a larger value. Example 1 depicts a network N and its associated flows by labelling the edges f_{ab}/c_{ab}, $(a,b) \in E(N)$ to show a maximal $s-t$ flow f on a network, such that f has value 4.

In this thesis we shall construct networks in such a way that there is a single sink vertex for the network and a single source vertex; we shall only be interested in $source-sink$ flows, and we may call a maximal $source-sink$ flow, a maximal flow of the network.

Example 1: A flow on a network

2.2 The target program as a sequence of directed graphs

Directed graphs can be used to represent control flow and data dependencies within application programs.

DEFINITION 10 *A sequence of instructions forms a* basic block *if the instruction in each position dominates, i.e always executes before, all those in later positions, and no other instruction executes between two instructions in the sequence [S:w08].*

DEFINITION 11 *A* control-flow graph *is a representation of the possible paths through a program. Each node in the control graph corresponds to a basic block. Directed edges are used to represent jumps in the control flow.*

DEFINITION 12 *A* data-dependency graph *(DDG) is obtained from a basic block by forming a vertex for each operation and adding an edge to each vertex u from each operation that computes an input operand of u.*

DDGs form directed acyclic graphs (DAGs) because they represent the computation performed by a basic block. For details of the process of generating DDG from basic blocks see Appendix B. An example data dependency graph is shown in Example 2.

The first explicit use of DDGs in candidate enumeration appears in [HD94] in 1994 and it has since become a de facto standard.

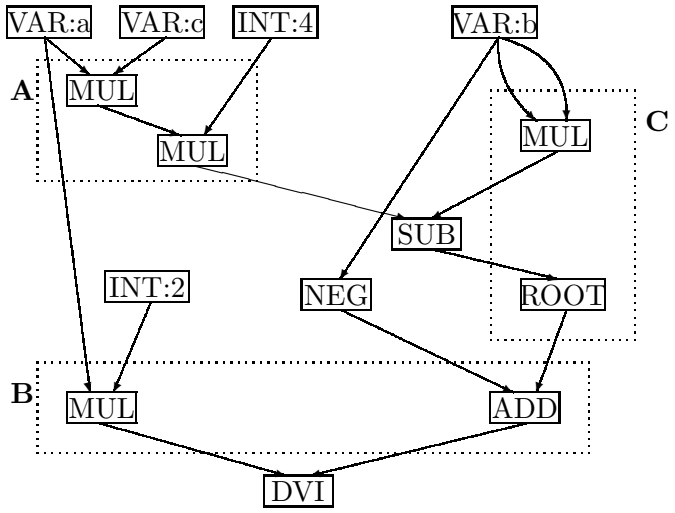
2.2.1 Candidate instructions as vertex sets in a DDG

A candidate instruction will perform a set of basic instructions as a monolithic computation. This candidate instruction corresponds to a subset, C say, of the vertices in D. Then the subset C uniquely identifies a candidate instruction applicable to D. It is these subsets that are sought when enumerating candidate instructions.

Example 2 shows a DDG with several highlighted sections. Section **A** of Example 2 represents a custom instruction to multiply three operands a, c, and 4. Similarly, an instruction to perform two unrelated computations in parallel, such as the **ADD** and **MULT** instructions towards the bottom of Example 2, could be identified in the DDG as the highlighted section **B**.

2.2.2 Interactions between compiler phases

The process of generating candidate instructions, from basic instructions, by examining DDGs is not independent of other 'code optimisations' performed by compilers because each basic block is examined individually. The interactions between such optimisations and custom instruction set selection are currently an area for future research. It is possible that some optimisations reduce the



Example 2: Data dependency graph for $\frac{-b+\sqrt{b^2-4ac}}{2a}$

opportunity for custom instructions, and that reliance on custom instructions prevents the use of equally efficient (and cheaper) code optimisations.

In particular, if loop unrolling is performed on a basic block, then the set of candidate custom instructions can be radically different [BP06], although not necessarily better. We anticipate that customisation of the instruction set will occur after all other optimisations have occurred. Then, one can assume that the quality of the target application is as high as possible before any improvements are achieved by instruction set customisation.

Ideally, the customisation of an instruction set would be one of a range of code improvements that interact together to find the best combination of transformations, rewritings and custom instructions. However, to date code improvement theory has not advanced to the stage that code improvements can reliably interact with each other, and certainly not with different phases of the process.

2.2.3 Other representations

Graph based approaches for candidate enumeration have been popular since they were introduced in [HD94] (the algorithms in [PAI06, CMS07, YM08] are recent examples). Earlier methods, such as [KLST04, GP03, CPH03] examine assembly code and attempt to fuse adjacent instructions, or use existing scheduling techniques to label parallel computations as a single instruction. These representations may prevent algorithms from enumerating complex candidate instructions.

2.3 Convexity

It is not the case that all sets of vertices in a DDG represent useful custom instructions. The highlighted section **C** of Example 2 shows an example of a subset that does not correspond to a useful custom instruction.

If section **C** were to be implemented as an instruction, then it would take as input the result of **SUB**. However, one of the inputs to the vertex **SUB** is one of the outputs of the proposed custom instruction and thus the execution of the instruction depends on an instruction that itself depends on the execution of the first instruction. If we restrict ourselves to convex vertex sets such a situation cannot arise.

Recall from Definition 6 that a set of vertices V_a is convex in D if every path in D between two vertices in V_a is also a path in the induced subgraph D_{V_a}.

Although one can imagine specialised datapath layouts that allowed information to pass in and out of an instruction, it is widely accepted that the hardware and scheduling difficulties outweigh any benefit. Thus in this thesis only candidate instructions that are represented by convex vertex sets of a DDG are considered valid candidate instructions.

2.4 Additional constraints on candidate instructions

Certain architectural features may prevent candidate instructions from being cost effective. These constraints can be defined in terms of vertex sets of a DDG and specific algorithms can be designed that return only candidate instructions that match certain criteria.

The most commonly used constraints are (i) forbidden vertices, (ii) connectivity, and (iii) restrictions on the number of inputs and outputs to the instruction (I/O constraints). An important advantage of taking these constraints into consideration is that it can reduce the number of candidate instructions that the instruction selection phase must consider. It will be shown in Chapter 3 that the number of convex vertex sets (and hence candidate instructions) in a DDG can be exponential in the size of the DDG. However, the number of convex vertex sets that satisfy certain I/O constraints, is at most polynomial in the size of the DDG.

Although these constraints can be used to identify, and hence remove, unwanted candidate instructions early, they carry a risk of removing candidate instructions that would justify the extra cost of supporting the breach in constraints. For example, equipping a processor to support the relaxation of a forbidden vertex constraint may allow candidate instructions that would double the throughput of the processor.

The trade-off between the overall improvement made by a custom instruction set and the restrictions placed on candidate instructions would be an interesting research area in its own right, however is outside the scope of this work.

We now consider each of the three types of constraint mentioned above.

2.4.1 Forbidden vertices

Given a DDG D, let $F(D)$ be the set of forbidden vertices, specified by a designer, such that $F(D) \subseteq V(D)$. If the forbidden vertex constraint is used, then no valid convex set contains any vertex that is a member of $F(D)$. The particular set of operations that are forbidden is dependent on features of the target architecture and the decisions made by the designers.

There are two main situations in which a vertex in the DDG can be made forbidden without adversely affecting the quality of the set of identified candidate instructions. Firstly, the vertex may not represent an operation — it may instead be an extra vertex added to the DDG to represent some computation external to the basic block. Secondly, the vertex may represent an operation not suitable for including in a custom instruction. Memory operations are often excluded because they cannot reliably meet tight timing constraints. Other types of operation that might be considered unsuitable for inclusion are floating-point operations and operations that require more than one clock cycle to execute, such as the multiply operation on early MIPS processors. Some approaches, such as [GPY+06], do not consider any vertices to be forbidden.

2.4.2 Connectivity

Recall from Definition 8 that for a directed graph D, a vertex set $A \subseteq V(D)$ is *connected* if, given any two vertices $v, u \in A$, there is an undirected walk from v to u in D_A^2.

A vertex set that is not connected corresponds to two or more unrelated basic or custom instructions that may be executed simultaneously.

There is potentially a tension between the role of the code scheduler and the design of custom instructions. A scheduler may be able to efficiently schedule instructions to exploit any available parallelism in the datapath. Then custom instructions that also exploit parallelism are either made redundant or they un-

dermine the scheduler by forcing unrelated operations to execute simultaneously when there may be a more useful ordering available.

If the compiler code generator has been created by a tree-parsing tool such as **iburg** [FHP92] or **twig** [AGT89], then only connected candidate instructions are useful because disconnected candidate instructions cannot be effectively utilised by the code generator [LB00].

The connectivity constraint does not ultimately restrict instruction choice because the set of all candidate instructions can be constructed from the set of connected candidate instructions. This approach of generating all candidate instructions from the set of connected candidate instructions is taken both by [YM07], and the partial solution approaches in Chapter 5.

Some algorithms are able to incorporate the restriction to connected sets with minimal adjustments. The most notable algorithm to enumerate only connected sets was presented in [YM04], and methods that work efficiently both with and without the connectivity constraint were presented in [BP07, CMS07].

Results published in [PAI06] suggest that using only connected convex vertex sets reduces the effectiveness of the final instruction set. However, the circumstances of the test somewhat favour unconnected convex vertex sets. The experiment in [PAI06] chose a small number of custom instructions and made no attempt to recombine connected convex vertex sets into unconnected ones. In addition, the benchmark chosen for the test naturally favoured operations executed in parallel. Overall, outcomes of the experiment should be viewed in context and interpreted with care.

2.4.3 I/O constraints

Algorithms that filter by I/O constraints only produce candidate instructions whose register reads and writes (generally known as inputs and outputs) are limited by specified bounds. This thesis denotes the designer specified input and output constraints as $inConstraint$ and $outConstraint$ respectively.

DEFINITION 13 *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 $IN(A, D)$.*

DEFINITION 14 *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 $(a, v) \in E(D)$ such that $v \notin A$. The set of output vertices of a set A in D is denoted by $OUT(A, D)$.*

Given a DDG D, a convex vertex set $C \subseteq V(D)$, and specified I/O constraints of $inConstraint$ and $outConstraint$, if we have either $|OUT(C, D)| > outConstraint$ or $|IN(C, D)| > inConstraint$, then C does not satisfy I/O constraints and will not be included in the final set of candidate instructions.

We let $\mathcal{S}_io(D)$ denote the set of subsets of $V(D)$ that are convex and satisfy I/O constraints.

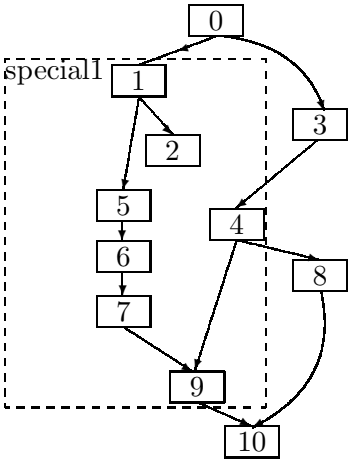
Chapters 3 and 6 show that this restriction can make a dramatic difference to both the number of candidate instructions and to the asymptotic complexities of algorithms that enumerate candidate instructions.

In principle, I/O constraints can significantly reduce the number of valid convex sets found. This is because most customisable processors have register banks that can support a maximum of only two to five register reads and one to three register writes per clock cycle.

However, some architectures allow the number of ports of the register bank to be parameterised, allowing designers to add extra ports to the register bank if there are useful custom instructions that perform too many register accesses. So in principle a designer may want to consider all candidate instructions before making a final decision on the number of ports for the register bank.

Approaches demonstrated in [PPIM03, Poz01, CFHZ04] identify candidates with only one output and [BGJ+02] limits itself to two. Approaches such as [PAI06, YM04, CMS07] do not fundamentally require I/O restrictions for correctness but they are more efficient under these constraints.

A variety of methods have been presented [JLM06, CFH+05, CHZ05, PI05, ADM+07], which allow custom instructions to read and write more values than the number of ports on the register file would normally allow. As a result, the bounds used in I/O constraints can be increased. At the present time, methods for candidate identification are yet to catch up and the algorithms in Chapter 5 make a contribution to narrowing this gap.



Example 3: Showing that restricting by I/O constraints can be suboptimal

There are circumstances in which designers may choose candidate instructions that breach I/O constraints. Consider the instruction *special*1 represented by the vertices $\{1, 2, 4, 5, 6, 7, 9\}$ in Example 3. The instruction *special*1 would ordinarily write values to two registers, which would be used by the vertices $\{8\}$ and $\{10\}$. If special instructions are restricted to only writing one value to a register, then *special*1 would be discarded. However, the best solution for Example 3 may be that *special*1 is implemented to only write the value required by $\{10\}$. The pseudocode to execute this fragment of DDG would be:

```
$3$
$special1$
$4$
$8$
```

Although the operation of vertex {4} will have been executed twice, this may still be the best solution, especially if the modified version of *special*1 is applicable elsewhere in the program. Thus general methods that do not employ I/O constraints may still be preferred.

I/O constraints and forbidden vertices

Inputs and outputs to a basic block are represented by additional forbidden vertices in the DDG. For this reason, algorithms that enumerate valid convex sets under I/O constraints usually filter by a forbidden vertex constraint. Moreover, this allows us to assume that every vertex in $V(D)$ has a directed path to a vertex in $F(D)$ and from a vertex in $F(D)$.

2.4.4 Heuristics for enumerating candidate instructions

Not all approaches define the candidate enumeration problem as above. Although approaches like [PAI06, YM04, BP07, CMS07] deliver the entire set of possible candidate instructions (the *exhaustive* methods)[1], other approaches like [ADO05, CFHZ04, CZM03, SRRJ02, GPY+06, BBD+05, CPH03] return some number of 'good' candidate instructions (the *heuristic* methods).

Heuristic methods attempt only to select a number of 'good' candidate instructions to pass to the instruction selection phase. The 'goodness' measure can take a variety of forms and must be based on local information because each basic block is considered individually. An example 'goodness' measure is to compare the number of operations in a candidate instruction (*totalOp*) with the number of operations in the longest directed path in the candidate instruction (*criticalOp*). The motivation for this heuristic is that *totalOp* is the number of cycles that the instruction will take to execute in software, and *criticalOp* is the number of cycles that the instruction will take to execute as a custom piece of hardware.

[1] subject to any other restrictions imposed by the user

Heuristic methods include approaches such as the use of a linear integer program solver [ADM+07] and combining only adjacent instructions. They may also use simplifying restrictions such as prohibiting the overlap of any candidate instructions [ADO05].

Recent approaches, such as [CZM03], search a section of the DDG and attempt to intelligently 'grow' the search in a fruitful direction, and [SRRJ02] compares discovered candidates with the current best selection, discarding any that are 'inferior'.

We distinguish between methods that are exhaustive with regard to some set of constraints and heuristic approaches. While neither will return all possible candidate instructions, an exhaustive method will guarantee to return all candidates that match the restriction(s), whereas a heuristic method cannot make such a guarantee. The rest of this thesis concerns only exhaustive algorithms.

2.5 Summary

This chapter has described valid candidate instructions as sets of vertices in a directed graph. It has gone on to define the properties that may be required of vertex sets to be candidate instructions. For the remainder of this thesis, we shall only discuss convex vertex sets of a DAG, and not candidate instructions of a DDG.

Chapter 3

The computational complexity of candidate enumeration

This chapter discusses upper and lower bounds for the number of convex vertex sets in DAGs under a variety of conditions. It also introduces a number of 'synthetic' DAGs that will be used to test the efficiency of the algorithms that are presented later in this thesis.

3.1 Provable bounds

In general, it is not possible to exactly calculate the number of convex sets in a DAG without enumerating them explicitly. However, it is possible to show upper and lower bounds for the number of convex vertex sets under certain conditions. Furthermore, for any given number of vertices, we can construct DAGs that display these bounds, and gain some insight into their relative frequency in commercial examples.

3.1.1 The lower bounds of $|\mathcal{S}(D)|$ and $|\mathcal{S}_c(D)|$

Recall that, if D is a DAG, there is an ordering $v_1, \ldots, v_n$ of the elements of $V(D)$ such that if $(v_i, v_j) \in E(D)$, then $i < j$. Given such an ordering, define a new graph D_{low} such that there is an edge from each v_i to every v_j such that

$i < j$. So

$$E(D_{low}) \;=\; \{(v_i, v_j) \in V \times V \mid i < j\}$$

It is easy to see that D_{low} is also a DAG, and by construction $E(D) \subseteq E(D_{low})$. We shall show by Lemma 2, that D has at least as many convex vertex sets as D_{low} and by Lemma 3, D_{low} must have $\frac{n(n+1)}{2} + 1$ convex vertex sets.

Lemma 2 *Suppose that G and H are DAGs where $V(H) = V(G)$ and $E(H) \subseteq E(G)$. If $X \subseteq G$ is convex in G, then it is also convex in H.*

Proof Consider the vertex set of G whose vertices are $v_1, \ldots, v_m$ say. If this vertex set is not convex in H, then there is a path from v_i to v_j, $0 < i < j \leq m$ that includes some vertex $u \notin \{v_1, \ldots, v_m\}$. Because $E(H) \subseteq E(G)$, this path also lies in G. Thus the vertex set $\{v_1, \ldots, v_m\}$ is also not convex in G. $\diamond$

Lemma 3 *Let $V(D_{low}) = \{v_1, \ldots, v_n\}$ and let $E(D_{low}) = \{(v_i, v_j) \mid 1 \leq i < j \leq n\}$. The DAG D_{low} has*

$$\frac{n(n+1)}{2} + 1$$

convex vertex sets.

Proof All the vertex sets that contain a single vertex are automatically convex. There are n such sets.

Of the vertex sets with two vertices $\{v_i, v_j\}, i < j$ there is a path $v_i, v_{i+1}, \ldots, v_j$ so the only convex vertex sets are those corresponding to the sets $\{v_i, v_{i+1}\}$, and these sets are convex. There are $n - 1$ such sets.

In general, for a set $\{v_{i_1}, v_{i_2}, \ldots, v_{i_q}\}$, where $i_1 < i_2 < \ldots < i_q$, if for some j we have $i_{j+1} \neq i_j + 1$, then the path $v_{i_j}, v_{i_j+1}, \ldots, v_{i_{j+1}}$ shows that the vertex set is not convex. Thus the only non-empty sets of vertices that are convex in D_{low} are of the form $V_{i,j} = \{v_i, v_{i+1}, \ldots, v_j\}$. There are

$$n + (n-1) + \ldots + 1 = \frac{n(n+1)}{2}$$

such sets.

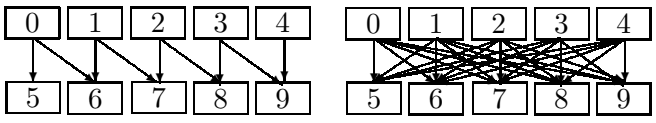
For any pair of vertices v_i, v_j if there is a path $v_i, v_{i_1}, \ldots, v_{i_q}, v_j$, then by definition of D_{low} we must have $i < i_1 < \ldots < i_q < j$ and so $v_{i_k} \in V_{i,j}$ for $1 \leq k \leq q$. Thus the subgraph induced by $V_{i,j}$ is convex, and as $\emptyset$ is convex with respect to D_{low} we have

$$\frac{n(n+1)}{2} + 1$$

convex sets. ◇

3.1.2 The upper bound of $|\mathcal{S}(D)|$

There are $2^{|V(D)|}$ possible vertex sets of a DAG D if convexity is disregarded; it is also possible to construct a DAG in such a way that every possible vertex set is convex. Example 4a is an example of a DAG in which every possible vertex set is convex. Interestingly, Example 4a will also generate the minimum number of connected convex vertex sets for a DAG with 10 vertices. However, Example 4(b) generates maximum numbers of both convex vertex sets and connected convex vertex sets for a DAG with 10 vertices.



(a) (b)

Example 4: DAGs showing upper bounds for the number of both unconnected (a and b) and connected (b) convex vertex sets in a DAG

3.1.3 Upper bound of $|\mathcal{S}_c(D)|$

Example 4b shows a DAG that generates the greatest number of connected convex vertex sets for any DAG with 10 vertices. DAGs of a similar construction

to Example 4b will generate $2^n - 2^{(n/2)+1} + 1$ convex connected vertex sets where there are n vertices in the DAG. A proof that this is an upper bound for $|\mathcal{S}_c(D)|$ is given in [YG]. Section 3.3.2 gives a general construction of such a DAG.

3.1.4 Upper and lower bounds of the number of convex vertex sets under I/O constraints

Example 5: DDG showing lower bound for the number of convex sets under an input constraint of 3 and an output constraint of 2

It has been shown in [BP07] and [CMS07] that when restricting convex vertex sets on the basis of I/O constraints, so a valid convex set can have no more than In inputs and Out outputs, the largest possible number of valid convex vertex sets is bound by $|V(D)|^{Out+In}$. Lemma 4 gives a similar proof. The lower bound on the number of convex sets under I/O constraints is zero and Example 5 (in which forbbiden vertices are represented by f vertices) shows a DAG that generates no valid convex sets ($In = 3$ and $Out = 2$).

Lemma 4 *Given a DAG D, and the sets $I, V_{Out} \subset V(D)$, there is at most one convex set C with $IN(C, D) = I$ and $OUT(C, D) = V_{Out}$ in a DAG D.*

Proof Let A and B be convex sets in D such that $IN(A, D) = IN(B, D) = I$ and $OUT(A, D) = OUT(B, D) = V_{Out}$.

Let $b \in B$. There must be a path P from b to a member of V_{Out}, and all vertices on this path must be within B as B is convex.

If $b \notin A$, then because $V_{out} \subset A$ some vertex on P must be in $|IN(A, D)|$. However, there can be no such vertex because $IN(A, D) = IN(B, D)$ and $IN(B, D) \cap B = \emptyset$ so we have a contradiction and $b \in A$.

By the same reasoning, if $a \in A$ then $a \in B$. Thus $A = B$. ◇

Lemma 5 *For a DAG D, there can be no more than $|V(D)|^{OUT+IN}$ convex sets with at most OUT output vertices and at most IN input vertices in a DAG D.*

Proof There can be at most $|V(D)|^{OUT}$ sets of output vertices of size no greater than OUT. Similarly there can be at most $|V(D)|^{IN}$ sets of input vertices of size no greater than IN. There are $|V(D)|^{OUT+IN}$ possible combinations of an input set with an output set and by Lemma 4, each combination can produce, at most, one convex set. Therefore, there can be no more than $|V(D)|^{OUT+IN}$ convex sets that have no more than OUT output vertices and IN input vertices in a DAG D. ◇

3.2 Real world examples

From an engineering perspective, we are interested in the average number of convex vertex sets over some set of examples. This section examines the number of convex vertex sets in some of the test cases that are used in this thesis. The number of convex sets that these test cases generate under certain conditions are evaluated.

Studies have shown [HP92] that typically, the number of operations in unoptimised, human generated basic blocks (and hence vertices in their attached DDGs) is around six. If all basic blocks were this small, then the enumeration of convex vertex sets would be trivial. However, code improvement techniques, such as loop unrolling and if-conversion, can produce extremely large basic blocks — even as large as 1,000-2,000 operations in length.

Input	Vertices	Connected convex sets	Convex sets
Dijkstra1	15	211	848
Dijkstra2Con	16	245	960
BF1Con	16	278	9,728
sha	16	434	4,026
qsort1Con	17	3,912	65,120
Dijkstra3Con	19	2,475	135,040
qsort3Con	29	24,780	6,069,360
Patricia4Con	29	691,294	53,412,360
qsort2Con	31	8,240	4,139,576
BF4Con	31	21,500	803,734
BF2	32	68,876	1,225,422
sha2	38	12,597	1,052,848
susan4Con	38	4,607,411	75,457,856
cjpeg1Con	39	485,718	8,338,040
susan3	40	5,304,066	125,639,352
sha3	46	225,998	1,661,374

Table 3.1 Sample numbers of convex vertex sets

Table 3.1 and Table 3.2 show the number of convex vertex sets in an arbitrary selection of examples of DDG from the RHUL dataset. Table 3.1 gives the numbers of convex vertex sets with no constraints and the numbers of connected convex vertex sets. Table 3.2 gives the numbers of convex vertex sets with various different I/O constraints[1].

The results show that not only can there be a large number of convex vertex sets (the example 'susan3' produces 125 million convex sets) but also that examples of similar sizes can have large differences in the number of convex sets (the example 'sha3' has more vertices than 'susan3' but has only one million convex sets).

With such large DDGs and sets of convex vertex sets, it is clear that brute force algorithms are insufficient for enumerating $|\mathcal{S}(D)|$.

[1]Appendix B details the construction of the examples in the RHUL dataset and Apprendix A details the conditions that these experiments were run under.

Input	Vertices (forbidden)	In	Out	Convex Sets
bf	467(134)	3	2	7,831
		5	2	40,714
		7	2	161,234
		4	3	105,599
		6	3	570,197
		8	3	2,155,103
cjpeg	152(34)	2	1	406
		4	1	544
		6	1	550
		3	2	41,363
		5	2	113,611
		7	2	140,335
		4	3	2,201,568
rijndael	1237(391)	2	1	1241
		4	1	4,787
		6	1	15,236
		3	2	75,241
		5	2	648,748
sha	1811(351)	2	1	1,546
		4	1	4,372
		6	1	10,152
		3	2	78,132
		5	2	293,259

Table 3.2 Sample numbers of convex vertex sets under I/O constraints

3.3 Synthetic DAGs

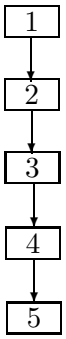
In this thesis we use four classes of *synthetic DAG* to examine the performance of algorithms over families of similar examples.

3.3.1 Sequential

For a sequential DAG

$$V(D) = \{v_0, \ldots, v_n\} \text{ and } E(D) = \{(v_i, v_{i+1}) \mid 0 \leq i < n\}.$$

Sequential graphs consist of 'chains' of operations and, as shown in Lemma 3, if a sequential DAG of n vertices has x convex sets, then there is no DAG with n vertices that has fewer convex sets. Example 6 shows a sequential DAG.



Example 6: Synthetic sequential DAG

3.3.2 Maximal

It is useful to be able to generate a synthetic DAG that produces the largest possible number of convex sets for any given number of vertices. Any DAG that contains no paths of length greater than one will produce the maximum number of convex sets because there can be no combination of vertices that breaks the convexity constraint. There are a number of different ways to construct such a DAG: our construction also generates the maximum number of connected convex sets. Our maximal form synthetic DAGs of size n are given by the

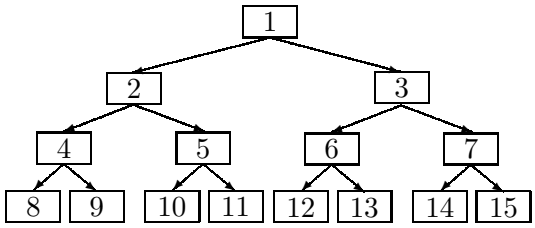
directed complete bipartite DAG $K_{n/2,n/2}$ if n is even or $K_{n+1/2,n/2}$ if n is odd. An example of a maximal DAG was shown in Example 4b.

3.3.3 Tree

For a synthetic tree DAG we have

$$V(D) = \{v_0, \ldots, v_n\} \text{ and } E(D) = \{(v_i, v_{i*2}), (v_i, v_{i*2+1}) \mid 0 \leq i < t\}$$

where $t = n/2$ if n is even or $t = (n-1)/2$ if n is odd. Example 7 shows a synthetic tree DAG.



Example 7: Synthetic tree DAG

3.3.4 Lattice

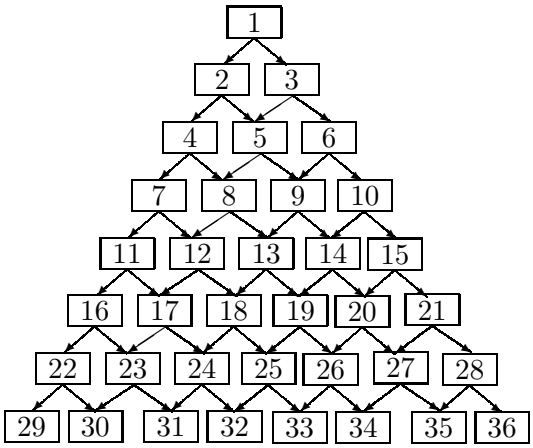
For a synthetic lattice DAG we have $V(D) = \{v_0, \ldots, v_n\}$ and

$$E(D) = \{(v_i, v_j), (v_i, v_{j+1}) \mid j = i + \lfloor \frac{1}{2} + \sqrt{2i} \rfloor, 1 \leq j < n\}.$$

An example of a lattice DAG is shown in Example 8.

3.3.5 Comparison of test cases to synthetic DAGs

In this section we show that, when examining DAGs obtained from commercial benchmarks, the majority of test cases contain fewer convex sets than a tree synthetic graph of the same size and more convex sets than a lattice synthetic graph of the same size.



Example 8: Synthetic lattice DAG

Figure 3.1 plots the relationship between the number of connected convex sets and the number of vertices in the DAG for examples from the RHUL dataset alongside results for the synthetic DAGs Maximal, Tree, and Lattice.

The graph shows that it is unusual for any of our commercial examples to approach the maximum number of possible convex sets, although several examples approach the minimum bound.

More interestingly, the number of connected convex sets in the majority of our test cases lie between the number of connected convex sets from tree-shaped DAGs of equivalent size and the number of connected convex sets from lattice-based ones of equivalent size.

There are two notable test cases that fall outside this classification, but both contain extremely long strings of operations and are similar in construction to the sequential synthetic DAG. This is a highly unusual situation, which is a result of the way that the sha algorithm produces its secure message digests. Both cases are shown in Example 9.

3.4 Summary

In this chapter, upper and lower bounds on the number of convex sets in a DAG have been shown for a variety of constraints.

Figure 3.1 Graph showing the number of convex connected sets in test cases compared to synthetic DAGs

(a) (b)

Example 9: The sha2 and sha3 benchmarks

A selection of 'synthetic DAGs' that can be use to test specific properties of enumeration algorithms have been introduced.

We have shown that the majority of test cases contain fewer convex sets than a synthetic tree graph of the same size and more convex sets than a synthetic lattice graph of the same size. For this reason the 'tree' and 'lattice' synthetic DAGs will later be used as performance indicators for the algorithms presented in this thesis.

Chapter 4

Previous work in candidate instruction enumeration

The chapter focuses on previously published algorithms for exhaustively enumerating convex vertex sets, rather than the heuristic approaches discussed in Chapter 2. This is because the quality of the instruction library generated by such approaches can only be inferior to exhaustive enumeration. We present these algorithms in unified form so as to bring out the similarities and differences between them.

4.1 Brute force

One can attempt to enumerate all possible instruction candidates of a DAG of size n by 'brute force' as shown in Algorithm 1. A bitstring of length n is maintained, where a '1' at position i in the bitstring indicates that the ith element of the DAG is present in the vertex set currently being examined. Each new vertex set is tested for convexity (and any other required constraints) and if the test is successful it is added to the candidate instruction library. The bitstring is then incremented to produce a new vertex set. This algorithm is slow, having to examine each of 2^n vertex sets for convexity (an $O(n^2)$ operation on its own) and any other constraints. Algorithm 2 performs brute force recursively.

Algorithm 1 *brute*(D): enumerating convex vertex sets by brute force

```
brute(D)
{
    Let R be the power set of V(D)
    S = R \ {∅}
    ∀X ∈ S do
        if X is valid
        {
            store X
        }
}
```

4.2 The exhaustive algorithm

The **exhaustive** algorithm (shown in Algorithm 3) was created by Atasu et al [API03] and extended in [PAI06]. It finds all convex vertex sets in a DAG by recursively building convex sets by adding vertices in topological order.

Computational overhead is reduced by pruning recursive calls if their recursive descendents cannot contain any valid convex vertex sets. It was the first algorithm to exhaustively enumerate all convex vertex sets in a given DAG without having to examine every possible subset of vertices of the DAG.

The fundamental insight of the **exhaustive** algorithm is that if a subset X is not convex in a DAG D, and $u \in V(D) \setminus X$ is topologically later than all vertices in X then $X \cup \{u\}$ is also not convex in D.

We use Example 10 to illustrate this. Figure 4.1 shows the call tree formed by *brute*2() (Algorithm 2) when processing Example 10; recursive calls that considered non-convex sets are marked with crosses. We note that any left child vertex of a vertex marked with a cross also contains a non-convex vertex set. By preventing such recursion, the **exhaustive** algorithm can reduce the overall number of recursive calls and hence, computational overhead. We see the reduction in recursive calls made by the **exhaustive** algorithm in Figure 4.2.

If the vertex set represented by a node in the search tree is not convex in D, then it can be shown that no sets represented by descendants of that node

Figure 4.1 Call tree for Example 10

Figure 4.2 Call tree for Example 10 pruned by the **exhaustive** algorithm

Algorithm 2 $brute2(D)$: recursively enumerating convex vertex sets by brute force

```
brute2(D)
{
    bruteRec(∅, 0, D)
}

bruteRec(X, i, D)
{
    if i < |V(D)|
    {
        if X ∪ {i + 1} is valid
        {
            store X ∪ {i + 1}
        }
        bruteRec(X ∪ {i + 1}, i + 1, D)
        bruteRec(X, i + 1, D)
    }
}
```

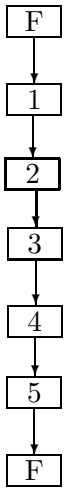
in the search tree will be convex, and the search tree rooted at that vertex can be pruned. Such pruning allows **exhaustive** to avoid performing a large proportion of the unnecessary computations that a brute force algorithm would have to.

4.2.1 I/O constraints

Algorithm 4 shows the **exhaustive** algorithm with I/O constraints.

If the vertices are examined in reverse topological order, then vertices that are currently outputs of the set X will always be outputs of the set X because no vertices can be added below them. Then if at any point there are too many output vertices, the recursive branch can be pruned.

The **exhaustive** algorithm for restricting by input constraint is less efficient than the output restriction. In [API03], there was a simple input check for each convex vertex set before it was stored. However, when the method was revised in [PAI06] an improved input checking algorithm was proposed that was able



Example 10: Motivational example for the **exhaustive** algorithm

Algorithm 3 $exhaustive(D)$: enumerating convex vertex sets, by pruning with regard to convexity

```
exhaustive(D)
{
    exhaustiveRecursive(∅, 0, D)
}

exhaustiveRecursion(X, i, D)
{
    if i < |V(D)|
    {
        if X ∪ {i + 1} is convex
        {
            store X ∪ {i + 1}
            exhaustiveRecursion(X ∪ {i + 1}, i + 1, D)
        }
        exhaustiveRecursion(X, i + 1, D)
    }
}
```

to cull more of the unnecessary branches. The improved algorithm maintainsa 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 *permanent* inputs than allowed by the input constraint, then the recursive branch could be pruned.

Algorithm 4 $exhaustiveIO(D)$: enumerating convex vertex by pruning with regard to convexity and I/O constraints

```
exhaustiveIO(D)
{
    exhaustiveIORecursion(∅, |V(D)|, D)
}

exhaustiveIORecursion(X, i, D)
{
    if i > 0
    {
        if (X ∪ {i − 1} is convex) & (|OUT(X, D)| ≤ outConstraint)
        &(externalInputs ≤ inConstraint)
        {
            if (|IN(X, D)| ≤ inConstraint)
            {
                store X ∪ {i − 1}
            }
            exhaustiveIORecursion(X ∪ {i − 1}, i − 1, D)
        }
        exhaustiveIORecursion(X, i − 1, D)
    }
}
```

4.2.2 Connectivity constraint

The **exhaustive** algorithm was not originally intended to limit its search to only connected convex vertex sets. However, because it is a simple and efficient algorithm, it is an excellent benchmark for other methods of generating connected convex vertex sets when it has been modified to only identify connected convex vertex sets. This modification to the **exhaustive** algorithm is shown in

Input	Vertices	$\lvert\mathcal{S}_c\rvert$	*exhaustiveCon*() time	*exhaustiveCon*() sets per second
bf2	32	4,136	0.43	9,618
bitcnts3	48	1,500	0.08	18,750
cjpeg3	29	105	0.02	5,250
qsort2	33	88	0.31	283

Table 4.1 Efficiency of the **exhaustive** algorithm when enumerating connected convex vertex sets

Algorithm 5.

Each convex vertex set, X, identified by the algorithm is tested for connectivity before it is stored. There are three possible results for the test. If the test finds that there are disconnected parts of X that can never be joined because the necessary vertices have already been removed from consideration, then the convex vertex set is discarded and no consequent recursive calls are made. If the test finds that there are disconnected parts of X that may be joined by a vertex, or set of vertices topologically earlier in the DAG, then the convex vertex set is discarded but the tree branch is not pruned so recursive search continues. Lastly, if the test shows that X is connected, then X is output.

The algorithm is effective and simple to implement but the overheads involved in testing for connectivity prevent the **exhaustive** algorithm from being competitive under a connectivity constraint, so it is not used in the performance tests in later chapters. Tables 4.1 and 4.2 show the reduction in efficiency of the **exhaustive** algorithm when enumerating only connected convex vertex sets[1].

4.3 The split algorithm

The **split** algorithm presented in [CMS07] and shown in Algorithm 6 is a recursive algorithm that uses a grading method to identify the vertex which will allow it to explore the search space with the least computational overhead.

[1]The conditions for this and all experiments in this thesis are detailed in Appendix A and the examples are from the RHUL dataset specified in Appendix B.

Algorithm 5 *exhaustiveCon*(D): enumerating connected convex vertex sets by pruning with regard to convexity and connectivity

```
exhaustiveCon(D)
{
    exhaustiveConRecursion(∅, 0, D)
}
exhaustiveConRecursion(X, i, D)
{
    if i < |V(D)|
    {
        if X ∪ {i+1} is convex
        {
            if connected(X ∪ {i+1}, D)
            {
                store X ∪ {i+1}
                exhaustiveConRecursion(X ∪ {i+1}, i+1, D)
            }
            else if canBeMadeConnected(X ∪ {i+1}, D)
            {
                exhaustiveConRecursion(X ∪ {i+1}, i+1, D)
            }
        }
        exhaustiveConRecursion(X, i+1, D)
    }
}
```

Input	Vertices	$\lvert\mathcal{S}\rvert$	*exhaustive*() time	*exhaustive*() sets per second
bf2	32	43,442	0.32	135,756
bitcnts3	48	5,479	0.06	91,316
cjpeg3	29	4,255	0.01	425,500
qsort2	33	60,415	0.20	302,075

Table 4.2 Efficiency of the **exhaustive** algorithm when enumerating all convex vertex sets

The estimated computation for adding a vertex u to the selection X is given by $estimate(u, X)$ in Algorithm 8. The vertex with lowest value of $estimate()$ is chosen and added to the selection. A number of heuristics may be employed to remove some vertices from consideration and to direct the search in such a way that involves the minimum amount of testing for input and output constraints. If there are no possible candidates, then the selection X is saved as a solution and the recursive call terminates.

When a vertex, v say, is chosen, two recursive calls (Algorithm 6(a) and (b)) are made in which v is made a forbidden vertex. In one of these calls (Algorithm 6(a)) all vertices that have a path to v are made forbidden vertices, and in the other call (Algorithm 6(b)) all vertices that have a path from v are made forbidden vertices. This operation is safe because if a vertex set contains a vertex with a path to v, a vertex with a path from v, and does not contain v, then it is not convex. If either of (Algorithm 6(a) and (b)) would cause a member of X to be made forbidden then that recursive call would not be made.

Finally, v is added to the selection X, along with all vertices necessary to satisfy convexity (Algorithm 6(c)) and a recursive call is made with this new version of X Algorithm 6(d).

4.3.1 Modification to enumerate connected convex vertex sets

Algorithm 7 shows a version of the **split** algorithm that is restricted to only enumerating connected convex vertex sets. This is achieved by only choosing vertices in $dom(X, D) \cup domBy(X, D)$ to add to the convex vertex set being constructed (Algorithm 7(b)). Moreover as the recursive calls Algorithm 6(a) and (b) cannot both be applicable when enumerating connected convex vertex sets we use an **if** cause to only use one(Algorithm 7(a)).

Algorithm 6 *split*(D): enumerating convex sets by recursive construction

```
split(D)
{
    ∀a ∈ V(D)(in topological order)
    {
        splitRecursive({a}, F(D), D)
        delete a from D
    }
}

splitRecursive(X, F, D)
{
    if X ∪ F = V(D)
    {
        store X
        return
    }
    v ← chooseV(X, F, D)
    if v ∉ domBy(X, D)
(a)     splitRecursive(X, F ∪ dom(v, D), D)
    if v ∉ dom(X, D)
(b)     splitRecursive(X, F ∪ domBy(v, D), D)
    X ← X ∪ {v}
(c) X ← X ∪ (dom(X, D) ∩ domBy(X, D))
(d) splitRecursive(X, F, D)
}

chooseV(X, F, D)
{
    return v such that v ∈ V(D) \ F, and estimate(v, X, D) is minimum.
}
```

Algorithm 7 *splitCon*(D): enumeration of connected convex sets by recursive construction

```
splitConRecursive(X, F, D)
{
    if (dom(X, D) \ F) ∪ (domBy(X, D) \ F) = ∅
    {
        store X
        return
    }
    v ← chooseConV(X, F, D)
(a) if v ∈ dom(X, D)
        splitConRecursive(X, F ∪ dom(v, D), D)
    else
        splitConRecursive(X, F ∪ domBy(v, D), D)
    X ← X ∪ {v}
    X ← X ∪ (dom(X, D) ∩ domBy(X, D))
    splitConRecursive(X, F, D)
}

chooseVCon(X, F, D)
{
(b) return v such that v ∈ (dom(X, D) ∪ domBy(X, D)) \ F,
    and estimate(v, X, D) is minimum.
}
```

Algorithm 8 $estimate(u, X, D)$: estimating the number of I/O checks performed by **split** if u is added to X

```
estimate(u, X, D)
{
    if  u ∈ dom(X, D)
    {
        Let s ← |dom(X, D) ∩ domBy(u, D)|
        Let p ← |dom(u, D)|
    }
    else if  u ∈ domBy(X, D)
    {
        Let p ← |domBy(X, D) ∩ dom(u, D)|
        Let s ← |domBy(u, D)|
    }
    else
    {
        Let p ← |dom(u, D)|
        Let s ← |domBy(u, D)|
    }
    return grade1(p, s)
}
grade1(p, s)
{
    \\we require 0 ≤ p, s < 2^16
    Let a ← max(p, s)
    Let b ← min(p, s)
    if b = 0
    {
        return 0
    }
    return(b << 16) + a
}
```

4.3.2 Summary

Results shown in [CMS07] and confirmed in Chapters 5 and 6 show that the **split** algorithm is the fastest previously published approach for enumerating convex sets under a variety of constraints. Furthermore, it can efficiently enumerate convex vertex sets without regard to I/O constraints. In Chapters 5 and 6 we present new algorithms that perform faster than the **split** algorithm on a variety of examples.

4.4 The poly_enum algorithm

The **poly_enum** approach, presented in [BP07], has polynomial time complexity in the size of the I/O constraints, and is the first algorithm for enumeration of convex vertex sets with such a bound. It makes use of *generalised dominators* (introduced in [Gup92]) to enumerate the sets. However, this use of generalised dominators enforces an extra constraint on the convex vertex sets that can be enumerated and so the generated candidate library may be incomplete.

DEFINITION 15 *In a rooted DAG*[2]*, a* generalised dominator *of a vertex set Q is a set A such that each path from the source vertex to Q passes through a vertex in A, and for each vertex in $a \in A$ there is a path from the source vertex to Q that contains only a and no other members of A.*

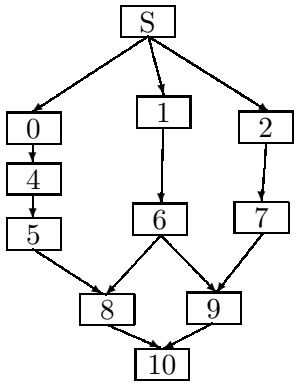
4.4.1 Generalised dominators and input sets

The insight made by the authors of [BP07] is that if A is a generalised dominator of B in D, then there is a convex vertex set S in D with $IN(S, D) = A$ and $B \subseteq OUT(S, D)$.

Example 11 shows this relationship between input sets and generalised dominators. For this DAG consider $F = \{0, 1, 2, 10\}$. Table 4.3 and Table 4.4 show

[2]If necessary any of our target DAGs can be made rooted by creating an additional forbidden vertex (the source) and connecting it to each vertex that has no dominators.

a clear correspondence between generalised dominators and inputs of convex vertex sets for the convex vertex sets with $OUT() = \{8, 9\}$.



Example 11: BP07 Example

Convex sets	Input sets
{8,9}	{5,6,7}
{5,8,9}	{4,6,7}
{4,5,8,9}	{0,6,7}
{6,8,9}	{5,1,7}
{5,6,8,9}	{4,1,7}
{4,5,6,8,9}	{0,1,7}
{7,8,9}	{5,6,2}
{5,7,8,9}	{4,6,2}
{4,5,8,9}	{0,6,2}
{6,7,8,9}	{5,1,2}
{5,6,7,8,9}	{4,1,2}
{4,5,6,7,8,9}	{0,1,2}

Table 4.3 Convex and input sets for Example 11 with $\{8, 9\}$ as outputs

Generalised dominators
{5,6,7}
{4,6,7}
{0,6,7}
{5,1,7}
{4,1,7}
{0,1,7}
{5,6,2}
{4,6,2}
{0,6,2}
{5,1,2}
{4,1,2}
{0,1,2}

Table 4.4 Generalised dominators of the set $\{8, 9\}$ in Example 11

Recall from Lemma 4 that, given an input set V_{in} and output set V_{out}, there is at most one convex vertex set C with $IN(C) = V_{in}$ and $OUT(C) = V_{out}$ in a DAG D. The algorithm presented in [BP07] attempts to enumerate convex vertex sets by pairing output sets with their generalised dominators. The generalised dominators are generated using a known polynomial algorithm from [DTM04]. For each pairing the **poly_enum** algorithm deduces the possible convex vertex set, and if it is suitable (it may be that such a set has extra

outputs or contains a forbidden vertex) stores it.

This approach has polynomial time complexity because the generalised dominators of a vertex set can be found in polynomial time and there can only be a polynomial number of such pairings. A high-level outline of this algorithm is shown in Algorithm 9.

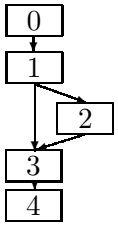
Algorithm 9 *poly_enum*(D): enumerating convex sets by finding generalised dominators of output sets

```
poly_enum(D)
{
    poly_enumRecursive(∅, ∅, ∅, D)
}

poly_enumRecursive(I, Q, S, D)
{
    ∀o ∈ V(D)|{o} ∪ Q is a valid output vertex set
    {
        Let newQ ← Q ∪ {o}
        Let G ← the set of generalised dominators of o
        ∀G ∈ Gsuch that |G ∪ I| ≤ inConstraint
        {
            Let newS ← S ∪ (dom({o}, D) ∩ domBy(G, D))
            if OUT(newS, D) = newQ
            {
                store newS
            }
            if |newQ| < outConstraint
            {
                poly_enumRecursive(G ∪ I, newQ, newS, D)
            }
        }
    }
}
```

4.4.2 Missing convex vertex sets

Although we have shown that if A is a generalised dominator of B in D, then there is a convex vertex set S in D with $IN(S, D) = A$ and $B \subseteq OUT(S, D)$,

the converse is not true.



Example 12: An instance where BP07 will miss a convex vertex set

Consider the convex vertex set $\{3\}$ in Example 4.4.2. $OUT(\{3\}) = \{3\}$ and $IN(\{3\}) = \{1, 2\}$. However, $\{1, 2\}$ is not a generalised dominator of $OUT(\{3\})$ and so **poly_enum** will not find the convex vertex set $\{3\}$ in this example.

It is noted in [BP07] that the algorithm does not find all convex vertex sets and a constraint is offered such that the **poly_enum** algorithm will find all convex vertex sets that satisfy the constraint:

> '... we add another condition for the validity of the convex cut. For each input $w \in I(S)$, there is a vertex $v \in S$, such that at least one path from the root of G to vertex v contains w but not any other input of S. This condition excludes from consideration a few valid cuts, namely those where an input w only has other inputs as predecessors...'.

Although the **poly_enum** algorithm will find all sets subject to this extra constraint, the convex vertex sets that are excluded are not limited to those that have other inputs as predecessors. We refer to this condition as *condition 1.*

Table 4.5 shows the number of convex vertex sets that **poly_enum** finds in comparison with both the total number of convex vertex sets and the number of convex vertex sets that satisfy 'condition 1'. These tests compare the number of convex vertex sets that match the various conditions for test cases in the RHUL dataset.

Input	Vertices (forbidden)	In	Out	Convex Sets	Condition 1	BP07
tree	36(19)	3	2	146	142	117
		4	3	752	711	470
		5	4	2,741	2,510	1,256
lattice	28(8)	3	2	147	141	117
		4	3	685	645	472
		5	4	2,449	2,287	1,367
bitcnts2+	74(23)	3	2	3,428	3,428	2,835
		4	3	63,611	63,611	40,413
bf1+	29(17)	3	2	93	90	89
		4	3	296	274	267
		5	4	623	543	524
fft2+	72(39)	3	2	1,081	1,078	1,047
		4	3	12,870	11,532	11,435
gsm1+	37(19)	3	2	228	228	218
		4	3	754	754	694
		5	4	1684	1684	1587
patricia2+	66(26)	3	2	1,294	956	705
		4	3	13,662	7,403	5,369

Table 4.5 Total convex vertex sets found by the BP07 Algorithm

There is a substantial variation in the effectiveness of the **poly_enum** algorithm when run on different test cases. With some test cases it will find all of the convex vertex sets, and with others it will find fewer than half the sets. Furthermore, there is often a significant difference between 'condition 1' and the total number of convex vertex sets that are found by the **poly_enum** algorithms.

4.4.3 Summary

The **poly_enum** algorithm, although theoretically interesting, is neither fully exhaustive nor consistent enough for industrial implementation. In Chapter 6, several polynomial-time algorithms are presented, which are not only fully exhaustive, but also have lower polynomial time complexity.

4.5 PKP08

A paper by Pothineni, Kumar, and Paul [PKP08] describes an algorithm that is almost identical to the **exhaustive** algorithm.

The PKP08 algorithm works in a very similar way to the **exhaustive** algorithm discussed in section 4.2, but instead of making two recursive calls at every stage, PKP08 makes a recursive call for every remaining vertex. A version of the PKP08 algorithm is shown in Algorithm 10. PKP08 can use virtually the same methods for pruning the search graph for convexity and I/O constraints as **exhaustive**.

The experimental evidence provided in [PKP08] shows PKP08 being only practical on very small input graphs and does not compare results with approaches like the **split** algorithm. We shall not consider this method further.

Algorithm 10 $PKP08(X, i, D)$: enumerating convex vertex sets by pruning with regard to convexity

```
PKP08recursive(D)
{
    PKP08(∅, 0, D)
}

PKP08recursive(X, i, D)
{
    ∀j, v_j ∈ V(D), j > i, X ∪ {v_j} is convex
    {
        store X ∪ {k}
        PKP08recursive(X ∪ {v_j}, j, D)
    }
}
```

4.6 Using cones to enumerate convex vertex sets

Several algorithms [YM04, YM08, GPY+06] make use of *cones* in the DAG.

DEFINITION 16 *A convex connected vertex set D_{upcone} of D is an* upward cone *of a vertex $u \in D_{upcone}$ if for all $n \in D_{upcone}$ there is a path from n to u.*

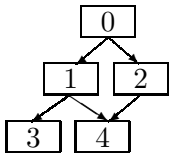
A convex connected vertex set $D_{downcone}$ of D is a downward cone *of a vertex $u \in D_{downcone}$ if for all $n \in D_{downcone}$ there is a path from u to n.*

These cones can be used on their own or combined to reveal convex vertex sets within a DAG. This section introduces cones and shows how they may be generated. The following section gives an overview of the MISO algorithm, which was the earliest use of these cones. Cones feature significantly in the **union** algorithm, which is discussed in detail in Chapter 7.

We note that all single output, connected convex vertex sets are upward cones, but the converse is not necessarily true. In Example 14 the set {2, 3, 5} is both a cone and a single output connected convex vertex set. However, the set {1, 2, 3, 5} is a cone with $OUT(\{1, 2,3, 5\}) = \{1, 5\}$.

4.6.1 Finding and labelling cones

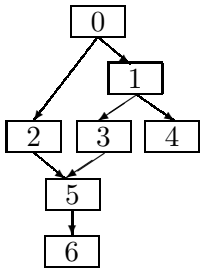
The algorithm $addCones(D, F)$, shown in Algorithm 11, takes a DAG and labels each vertex with its upward cones, using the stub function $addUpwardSets(i, \mathcal{A})$ where i is a vertex in the graph and $\mathcal{A}$ is a set of vertex sets to be associated with i. A similar algorithm can be used to label each vertex with its downward cones. When applied to tree shaped input, the cone finding algorithm will find each cone exactly once. Although it is broadly efficient for other input, the cone finding algorithm may find the same cone several times.



Example 13: A case where the cone labelling algorithm will find the same cone multiple times

Example 13 shows that the labelling algorithm will attach cones {2},{2, 0} to [2] and {1},{1,0} to [1]. However, when these two groups of cones are combined to create the cones for [4], the cone {4, 2, 1, 0} will be located three

times: once by combining {2} with {1, 0}; once by combining {2, 0} with {1}; and once by combining {2, 0} with {1, 0}.



Example 14: The difference between a cone and a single output convex vertex set

4.6.2 MISO

The earliest of the cone-based algorithms is MISO(Algorithm 12), standing for Multiple Input-Single Output [PPIM03, Poz01]. The algorithm finds connected convex vertex sets that have a single output vertex. If we can guarantee that the input is a tree, then we can also guarantee that all convex vertex sets will have a single output vertex and thus MISO is an exhaustive algorithm for tree-like structures. In all other cases MISO must be classed as a heuristic algorithm.

A MISO-like algorithm appears in [CFHZ04]. One of the original intentions of the method was to be able to extract the candidates called MaxiMISOs, which would be the largest possible single output convex vertex sets in a given DAG. The heuristic [GPY+06] examines the potential of combining these MaxiMISOs.

The MISO algorithm can be implemented by running the cone labelling algorithm in Section 11 on a DAG, then discarding all those cones that have more than one output (Algorithm 12). The sets that are output will be the single output convex vertex sets of D. As a result, MISO is of limited applicability.

Algorithm 11 $addCones(D, F)$: labelling vertices with their upward cones

```
addCones(D, F))
{
  ∀i ∈ V(D) \ F
  {
     addUpwardSets(i, {{i}})
     ∀a ∈ V(D)|(a, i) ∈ E(D)
     {
        A = combinePatterns(returnPatternsUp(a), returnPatternsUp(i))
        addUpwardSets(i, A)
     }
  }

combinePatterns(B, C)
{
  Let R = ∅
  ∀B ∈ B
     ∀C ∈ C
        if convex(B ∪ C)
        {
           add (B ∪ C) to R
        }
  return R
}
```

Algorithm 12 $MISO(D, F)$: enumerating convex vertex sets that have a single output

```
MISO(D, F))
{
  ∀i ∈ V(D) \ F
  {
    addUpwardSets(i, {{i}})
    ∀a ∈ V(D)|(a, i) ∈ E(D)
    {
      A = combineMISO(returnPatternsUp(a), returnPatternsUp(i))
      addUpwardSets(i, A)
    }
  }

combineMISO(B, C)
{
  Let R = ∅
  ∀B ∈ B
    ∀C ∈ C
      if convex(B ∪ C)
      {
        add (B ∪ C) to R
        if OUT(B ∪ C) = 1
        {
          output B ∪ C
        }
      }
  return R
}
```

4.7 Summary

This chapter has surveyed algorithms for exhaustively enumerating convex vertex sets. The significant algorithms in the area are the **exhaustive** and **split** algorithms. The following two chapters present new algorithms for exhaustive enumeration of convex vertex sets and demonstrate their superiority.

Chapter 5

Exhaustive search on a directed acyclic graph

This chapter presents the Φ algorithm for enumerating all convex vertex sets, and the Ψ algorithm for enumerating connected convex vertex sets. Modifications to these algorithms that enable them to deal with constraints such as forbidden vertices are also shown. Some elements of this work have been published in [GJR+07] and extended in [GJR+08a].

The chapter concludes by presenting a two-stage process for enumerating all convex vertex sets. This process uses the Ψ algorithm to create a 'partial solution', and we compare several different ways of expanding this partial solution. The performance of each algorithm is tested under a range of different conditions and the results are compared to the performance of the **split** and **exhaustive** algorithms reviewed in the previous chapter.

5.1 The Φ algorithm: all convex vertex sets

The input to the Φ algorithm is a DAG D, and it outputs all convex vertex sets of D exactly once. We begin with an overview of the algorithm (Algorithm 13) and an example of its use. This is followed by a discussion of the asymptotic complexity of the algorithm.

The core insight of the Φ algorithm is that if X is a convex vertex set of a DAG D and x is either a source or a sink vertex of D_X, then $X \setminus \{x\}$ is also convex. Then every convex vertex set can be reached by the removal of a series of source or sink vertices from the original DAG. By controlling the removal of source or sink vertices from a DAG D, the Φ algorithm can enumerate all possible convex vertex sets of D.

Algorithm 13 $\Phi(D)$: enumerating convex vertex sets by removal of source and sink vertices

```
Φ(D)
{
    Φrec(V(D), V(D), D)
}

Φrec(X, Y, D)
{
    store X
    ∀v ∈ (sink(X) ∩ Y) ∪ (source(X) ∩ Y)
    {
        Y ← Y \ {v}
        Φrec(X \ {v}, Y, D)
    }
}
```

5.1.1 Overview

Algorithm 13 is a high-level description of the Φ algorithm. The recursive function at the heart of the Φ algorithm, $\Phi rec(X, Y, D)$, will find all convex vertex sets S such that $S \subseteq X$, $S \cap Y \neq \emptyset$.

In each call, the algorithm stores the current value of X and iterates over the set of vertices that are either source or sink vertices of X, and are also members of Y. For every vertex v in Y that is a member of either $source(X)$ or $sink(X)$, a recursive call $\Phi rec(X \setminus \{v\}, Y', D)$ is made, where $Y' \subset Y$. This recursive call will store all convex vertex sets S such that $S \subseteq X \setminus \{v\}$, $S \cap Y' \neq \emptyset$. This also

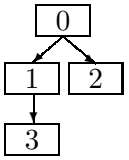
Index	Set
1	{}
2	{3}
3	{2}
4	{1,3}
5	{1}
6	{0,1,2,3}
7	{0,1,3}
8	{0,1,2}
9	{0,1}
10	{0,2}
11	{0}
12	{1,2}
13	{2,3}
14	{1,2,3}

Table 5.1 Convex vertex sets obtained from Example 15

ensures that all convex sets found by further recursive calls of $\Phi rec()$ contain v and ensures that the same convex vertex set cannot be found more than once by $\Phi rec()$.

When there are no remaining vertices that are both in Y and are a source or sink of X, the algorithm stores X and terminates.

5.1.2 Example



Example 15: DAG used for examples

Example 15 will be used for all the examples in this chapter. It generates the convex vertex sets shown in Table 5.1.

The call graph produced by executing the Φ algorithm on Example 15 is shown in Figure 5.1. When $\Phi rec(\{0,1,2,3\},\{0,1,2,3\})$ is called, it corresponds to vertex (A) of the call graph.

Figure 5.1 Recursive calls made by Φ on Example 15

Execution trace of $\Phi rec()$

We present an execution trace of $\Phi rec()$ as it process the DDG in Example 15. This trace corresponds to the call graph in Figure 5.1 and the nesting of recursive calls is represented by indentation.

(A1) The convex vertex set {0, 1, 2, 3} is stored.

(A2) Vertices 0, 2, and 3 are suitable vertices for removal, 0 is chosen for removal first and the recursive call $\Phi rec(\{1,2,3\},\{1,2,3\})$ is made.

(B1) The convex vertex set {1, 2, 3} is stored.

(B2) Vertices 1, 2, and 3 are suitable vertices for removal, 1 is chosen for removal first and the recursive call $\Phi rec(\{2,3\},\{2,3\})$ is made.

(C1) The convex vertex set {2, 3} is stored.

(C2) Vertices 2 and 3 are suitable vertices for removal, 2 is chosen for removal first and the recursive call $\Phi rec(\{3\},\{3\})$ is made.

(D1) The convex vertex set {3} is stored.

(D2) Vertex 3 is the only suitable vertex for removal and the recursive call $\Phi rec(\emptyset,\emptyset)$ is made.

(E1) The convex vertex set {} is stored.

(C3) Vertex 3 is chosen for removal and the recursive call $\Phi rec(\{2\}, \emptyset)$ is made.

(F1) The convex vertex set $\{2\}$ is stored.

(B3) Vertex 2 is chosen for removal and the recursive call $\Phi rec(\{1,3\}, \{3\})$ is made.

(G1) The convex vertex set $\{1,3\}$ is stored.

(G2) Vertex 3 is the only suitable vertex for removal and the recursive call $\Phi rec(\{1\}, \emptyset)$ is made.

(H1) The convex vertex set $\{1\}$ is stored.

(B3) Vertex 3 is chosen for removal and the recursive call $\Phi rec(\{1,2\}, \emptyset)$ is made.

(I1) The convex vertex set $\{1,2\}$ is stored.

(A3) Vertex 2 is chosen for removal and the recursive call $\Phi rec(\{0,1,3\}, \{1,3\})$ is made.

(J1) The convex vertex set $\{0,1,3\}$ is stored.

(J2) Vertex 3 is the only suitable vertex for removal so we make the recursive call $\Phi rec(\{0,1\}, \{1\})$.

(K1) The convex vertex set $\{0,1\}$ is stored.

(K2) Vertex 1 is the only suitable vertex for removal so we make the recursive call $\Phi rec(\{0\}, \emptyset)$.

(L1) The convex vertex set $\{0\}$ is stored.

(A4) Vertex 3 is chosen for removal and the recursive call $\Phi rec(\{0,1,2\}, \{1\})$ is made.

(M1) The convex vertex set $\{0,1,2\}$ is stored.

(M2) Vertex 1 is the only suitable vertex for removal so we make the recursive call $\Phi rec(\{0,2\}, \emptyset)$.

(N1) The convex vertex set $\{0,2\}$ is stored.

5.1.3 Time complexity

The Φ algorithm has an overall time complexity of $O(\sum_{S\in\mathcal{S}(D)}|S|)$ (Lemma 6). Algorithm 14 shows the Φ algorithm in greater detail. In particular, it shows how the *source* and *sink* sets can be maintained by keeping count of the number of in- and out-neighbours that each vertex has within X.

Lemma 6 *Given a DAG D, which includes adjacency lists for each vertex, an adjacency matrix, and the number of in-neighbours and out-neighbours for each vertex*[1]*, the time complexity of $\Phi(D)$ is $O(\sum_{S\in\mathcal{S}(D)}|S|)$.*

Proof Initialising the arrays $in[]$ and $out[]$ (Algorithm 14(a)) can be done in $O(|V(D)|^2)$ time. Given the arrays $in[]$ and $out[]$, which are of size $|V(D)|$, the sets *sink* and *source* can be initialised in $O(|V(D)|)$ time. Thus the initialisation of the Φ algorithm requires $O(|V(D)|^2)$ time.

Each call of $\Phi rec()$ stores exactly one convex vertex set. There can be only $\mathcal{S}(D)$ calls because only convex vertex sets are stored, and each set is stored only once. We must show that each call to $\Phi rec(X,Y,D)$, without recursive calls, has complexity $O(|X|)$.

$\Phi rec(X,Y,D)$ iterates over a linked list of source and sink nodes of $(sink\cap Y)\subseteq X$ (Algorithm 14(b)), which is an $O(|X|)$ operation. For each element v in the set $(source\cup sink)\setminus Y$, a call to $\Phi rec()$ is made (Algorithm 14(c)).

The values of $in[]$ or $out[]$ for neighbours of v that are in X must be updated before the call is made (Algorithm 14(d)). There are at most $|X|-1$ such neighbours so the operation requires $O(|X|)$ time. However, the cost can then be 'charged' to the recursive call that is about to be made. Updating the number of out or in neighbours will also update the *source* or *sink* sets. The process is reversed to restore the values of *in* or *out* as soon as the recursive call terminates (Algorithm 14(e)).

[1]If the in- and out-neighbours are not given, then they can be computed in $O(|V(D)|^2)$ time

Algorithm 14 $\Phi(D)$: enumerating convex vertex sets by removal of source and sink vertices (detailed view)

```
Φ(D)
{
(a)   Let in[] be an array initialised such that in[i] = |{j|(j,i) ∈ E(D)}|
      Let out[] be an array initialised such that out[i] = |{j|(i,j) ∈ E(D)}|
      sink ← all v_i ∈ V(D) such that out[i] = 0
      source ← all v_i ∈ V(D) such that in[i] = 0
      Φrec(V(D), V(D), D, source, sink, in[], out[])
}

Φrec(X, Y, D, source, sink, in, out)
{
      store X
      ∀v ∈ (sink ∩ Y)
      {
          ∀w ∈ X
          {
              if w is a direct predecessor of v
              {
                  decrement out[w]
                  if out[w] = 0
                      sink = sink ∪ {w}
              }
          }
          Y ← Y \ {v}
          Φrec(X \ {v}, Y, D, source, sink, in, out)
          restore values of out[] and sink to before the loop
      }
(b)   ∀v ∈ (source ∩ Y)
      {
(d)       ∀w ∈ X
          {
              if w is a direct sucessor of v
              {
                  decrement in[w]
                  if in[w] = 0
                      source = source ∪ {w}
              }
          }
          Y ← Y \ {v}
(c)       Φrec(X \ {v}, Y, D, source, sink, in, out)
(e)       restore values of in[] and source to before the loop
      }
}
```

Deleting and reinserting vertices requires constant time because we are not concerned with the order of elements in the linked list, so $\Phi rec(X, Y, D)$ has complexity $O(|X|)$.

In summary, we require $O(|V(D)|^2)$ time to initialise the algorithm and each call of $\Phi rec()$ requires $O(|X|)$ time to store a convex set of size $O(|X|)$.

Then the total running time is $\Omega(|V(D)|^2) + O(\sum_{S \in \mathcal{S}(D)} |S|)$. By Lemma 3 $O(\sum_{S \in \mathcal{S}(D)} |S|) = \Omega(|V(D)|^2)$ so the overall running time of the Φ algorithm is $O(\sum_{S \in \mathcal{S}(D)} |S|)$. ◇

5.2 The Ψ algorithm: connected convex vertex sets

The Ψ algorithm (first presented in [GJR+07]) enumerates connected convex vertex sets of a DAG. A high-level description is shown in Algorithm 15.

Whilst the Φ algorithm starts from a full DAG and shrinks the selection down to find convex vertex sets, the Ψ algorithm starts from a single 'seed' vertex and grows the selection by adding vertices from the set Y.

The Ψ algorithm can ensure that the convex vertex set being grown is always connected by ensuring that only vertices that dominate or are dominated by X are chosen to extend by. However, there may be more vertices that need to be added to satisfy convexity because the vertices being added are not necessarily adjacent to the current selection—checking for these extra vertices is an overhead that Φ avoids.

5.2.1 Overview

$\Psi rec(\{x\}, V(D), D)$ will store all connected convex vertex sets in D, for which $\{x\}$ is a member. All convex vertex sets for D can be enumerated by executing $\Psi rec(\{x\}, V(D), D)$ and then deleting vertex $\{x\}$ from D and repeating the process.

If x were on a path between two other vertices in D, then the next recursive

Algorithm 15 $\Psi(D)$: enumerating connected convex vertex sets, by selection and rejection of vertices

```
Ψ(D)
{
    ∀a ∈ V(D)(in topological order)
    {
        Ψrec({a}, V \ {a}, D)
        delete a from D
    }
}

Ψrec(X, Y, D)
{
    if (dom(X, D) ∩ Y) ∪ (domBy(X, D) ∩ Y) = ∅
    {
        store X
        return
    }
    v ← chooseV(X, Y, D)
    Ψrec(X, Y \ {v}, D)
    X ← X ∪ {v}
    X ← X ∪ (dom(X, D) ∩ domBy(X, D))
    Ψrec(X, Y \ X, D)
}

chooseV(X, Y, D)
{
    if (domBy(X, D) ∩ Y) ≠ ∅
    {
        find the vertex v_i ∈ (domBy(X, D) ∩ Y) with maximum i.
    }
    else
    {
        find the vertex v_i ∈ (dom(X, D) ∩ Y) with minimum i.
    }
    return v_i
}
```

Index	X
1	{3}
2	{2}
3	{1,3}
4	{1}
5	{0,1,2,3}
6	{0,1,3}
7	{0,1,2}
8	{0,1}
9	{0,2}
10	{0}

Table 5.2 Connected convex vertex sets obtained from Example 15

call may generate some vertex sets that are not convex in D. To prevent this, vertices are removed from D in topological order.

A significant difference between Φrec and Ψrec is that when Ψrec makes an addition to the X set it may need to add more vertices to X to maintain convexity. The operations $X \leftarrow X \cup \{v\}$ and $X \leftarrow X \cup (dom(X, D) \cap domBy(X, D))$ in Algorithm 15 perform this function for the Ψ algorithms.

v must be a vertex that is not in the current X set and which has not yet been removed from Y. Furthermore, it must either dominate or be dominated by members of X. In the implementation used for the experiments later in this chapter, the topologically first vertex in $dom(X, D) \cap Y$ is chosen for preference, followed by the topologically last vertex in $domBy(X, D) \cap Y$, this ensures that Ψrec does not need to make any extra vertices forbidden when rejecting v from the selection.

5.2.2 Example

Example 15 is used to show the operation of the Ψ algorithm. It generates the connected convex vertex sets shown in Table 5.2 in the order in which they are shown.

When processing the DAG shown in Example 15, the Ψ algorithm will call

the recursive function $\Psi rec()$ once for each of the vertices in the graph. These calls correspond to the four call graphs in Figure 5.2.

Firstly, $\Psi rec()$ is called with $X = \{0\}$, $Y = \{1, 2, 3\}$ corresponding to vertex (A) of the call graph. Once this call has successfully terminated, $\Psi rec()$ is called with: $X = \{1\}$, $Y = \{2, 3\}$; $X = \{2\}$, $Y = \{3\}$; $X = \{3\}$, $Y = \{\}$, which correspond to the call graph vertices $(L), (O)$, and (P) respectively.

Execution trace of Ψrec

(A) Vertices 2 and 3 would both be suitable v vertices, 3 is chosen because it is topologically later. The v vertex is removed from Y and a recursive call (E) is made. v is added to X and because vertex 1 is on a path between 3 and X 1 is added to X and a second recursive call is made to (B).

(B) Vertex 2 is the only suitable v vertex, so it is removed from Y and a recursive call is made (D), v is then added to X and a second recursive call is made (C).

(C) There is no suitable v vertex so the set $\{0, 1, 2, 3\}$ is stored.

(D) There is no suitable v vertex so the set $\{0, 1, 3\}$ is stored.

(E) Vertices 1 and 2 would both be suitable v vertices, 2 is chosen because it is topologically later. v is removed from Y and a recursive call (I) is made. v is added to X and a second recursive call is made to (F).

(F) Vertex 1 is the only suitable v vertex, so it is removed from Y and a recursive call is made (H), v is then added to X and a second recursive call is made (G).

(G) There is no suitable v vertex so the set $\{0, 1, 2\}$ is stored

(H) There is no suitable v vertex so the set $\{0, 2\}$ is stored

(I) Vertex 1 is the only suitable v vertex, so it is removed from Y and a recursive call is made (K), v is then added to X and a second recursive call is made (J).

(J) There is no suitable v vertex so the set $\{0, 1\}$ is stored.

(K) There is no suitable v vertex so the set $\{0\}$ is stored

(L) Vertex 3 is the only suitable v vertex, so it is removed from Y and a recursive call is made (N), v is then added to X and a second recursive call is made (M).

Figure 5.2 Recursive calls made by Ψ on Example 15

(M) There is no suitable v vertex so the set $\{1, 3\}$ is stored

(N) There is no suitable v vertex so the set $\{1\}$ is stored

(O) There is no suitable v vertex so the set $\{2\}$ is stored

(P) There is no suitable v vertex so the set $\{3\}$ is stored

5.2.3 Comparisons with the split algorithm

The Ψ algorithm and **split** [CMS07] have a number of algorithmic similarities, even though they were developed independently. Both algorithms grow a convex vertex set from a single 'seed' vertex and, where necessary, add vertices to the set to maintain convexity. The major difference is that the Ψ algorithm makes use of the first extension point found, whereas **split** finds all possible extension vertices and ranks them in order to decide which would be the best vertex to use as an extension point.

5.2.4 Time complexity

The time complexity of the Ψ algorithm is $O(|V(D)| \cdot |\mathcal{S}_c(D)|)$ (Lemma 7).

Lemma 7 *The time complexity of the Ψ algorithm is* $O(|V(D)| \cdot |\mathcal{S}_c(D)|)$.

Proof By Lemma 8 a reachability matrix can be computed in $O(|V(D|) \cdot |\mathcal{S}_c(D)|)$ time.

Consider the recursive function $\Psi rec(X, Y, D)$, which, with appropriate data structures and a pre-computed reachability matrix, has a worst-case complexity of $O(|Y|)$, $Y \subseteq V(D)$. Then $\Psi rec(X, Y, D)$ is of order $O(|V(D)|)$.

Consider that the recursive calls made by $\Psi rec(X, Y, D)$ form a forest of binary trees. Each leaf vertex in this forest will store exactly one connected convex vertex set. Thus there must be exactly $\mathcal{S}_c(D)$ leaf vertices and the total number of vertices (and hence calls to the function $\Psi rec(X, Y, D)$) in the recursive trees must be of $O(|\mathcal{S}_c(D)|)$. Because the total number of calls of the recursive function is at most $O(|\mathcal{S}_c(D)|)$, and each call has a complexity of $O(|V(D)|)$, then the worst case order of the algorithm is $O(|V(D)| \cdot |\mathcal{S}_c(D)|)$.◇

Lemma 8 *A reachability matrix for a DAG D can be found in $O(|V(D)| \cdot |\mathcal{S}_c(D)|)$ time.*

Proof Pre-computing a reachability matrix for a DAG is equivalent to computing the transitive closure which requires $O(|V(D)|^3)$ time [BJG00].

By Lemma 3 in Chapter 3, $|\mathcal{S}_c(D)|$ is of minimum order $O(|V(D)|^2)$. Then $|V(D)| \cdot |\mathcal{S}_c(D)|$ is of minimum order $O(|V(D)|^3)$. Then a path table for DAG D can be precomputed in $O(|V(D)| \cdot |\mathcal{S}_c(D)|)$ time. ◇

5.2.5 Using Ψ to enumerate all convex vertex sets

It is not hard to modify the Ψ algorithm such that the new algorithm will generate all convex vertex sets of a DAG D in time $O(|V(D)| \cdot |\mathcal{S}(D)|)$, including non-connected sets. The Ψ algorithm will generate all convex vertex sets if we replace $chooseV(X, Y, D)$ with the one shown in Algorithm 16 and relax the constraint for terminating the recursion. The version of the Φ algorithm that includes this modification is referred to as the $\Psi_{allsets}$ algorithm.

Algorithm 16 $\Psi_{allsets}$: enumerating all convex vertex sets

```
Ψ_allsets(D)
{
    ∀a ∈ V(D)(in topological order)
    {
        Ψ_allsets rec({a}, V \ {a}, D)
        delete a from D
    }
}

Ψ_allsets rec(X, Y, D)
{
    if Y = ∅
    {
        store X
        return
    }
    v ← chooseV_allsets(X, Y, D)
    Ψ_allsets rec(X, Y \ {v}, D)
    X ← X ∪ {v}
    X ← X ∪ (dom(X, D) ∩ domBy(X, D))
    Ψ_allsets rec(X, Y \ X, D)
}

chooseV_allsets(X, Y, D)
{
    return any v ∈ (source(Y) ∩ sink(Y)) \ X
}
```

There is some overhead involved in maintaining the convexity of X at each recursive call because the $\Psi_{allsets}$ algorithm adds vertices to a convex vertex set being constructed rather than taking them away from a larger set. Section 5.6.1 shows that the relative speed of the two algorithms reflect this overhead. Moreover, the $\Psi_{allsets}$ algorithm is asymptotically inferior to the Φ algorithm as shown by Lemmas 7 and 6.

5.3 Adding I/O constraints to the Φ and Ψ algorithms

It is simple to modify both the Φ and Ψ algorithms in such a way that they will store only those convex vertex sets that are valid under I/O constraints. However, faster algorithms for enumeration of convex vertex sets under I/O constraints are presented in Chapter 6.

5.4 Forbidden vertices

In Chapter 2 we noted that convex vertex sets are invalid if they contain vertices that represent forbidden operations. If there is a large number of such operations, then they can make a considerable difference to the overall efficiency of the algorithms. In this section, modifications to the Φ and Ψ algorithms are presented that enable them to efficiently process input DAGs that contain forbidden vertices.

We must distinguish between internal and external forbidden vertices. A vertex $f \in F(D)$ is an *internal forbidden vertex* if there is at least one edge (a, f) and one edge (f, b) in $E(D)$ for some $a, b \in V(D)$. All other vertices in $F(D)$ are denoted *external forbidden vertices*.

5.4.1 External forbidden vertices

If all of the forbidden vertices within the target DAG are external forbidden vertices, then simple changes can be made to both Φ and Ψ to incorporate this

restriction.

The calling functions of both algorithms can be modified by the introduction of a set U that contains only vertices that have not been forbidden. The Ψ algorithm uses U, instead of $V(D)$, as the set of possible vertices to extend by. The Φ algorithm uses U as the initial set of vertices before any have been removed. The alterations to the calling functions of both these algorithms are shown in Algorithm 17.

Algorithm 17 Calling functions for the Φ and Ψ algorithms with support for forbidden vertices

```
Φ(D)
{
    Let U ← V(D) \ F(D)
    Φrec(U, U, D)
}

Ψ(D)
{
    Let U ← V(D) \ F(D)
    ∀a ∈ U
    {
        Ψrec({a}, U \ {a}, D)
        delete a from V(D) and U
    }
}
```

5.4.2 Internal forbidden vertices

The Φ algorithm is modified by adding a forbidden vertex check before each convex vertex set is stored. This approach is not efficient and it will cause significant delays if there is a high proportion of forbidden vertices in the DAG.

Algorithm 18 shows further modifications to the Ψ algorithm to incorporate this restriction. In addition to the changes discussed in Section 5.4.1, the $chooseV()$ function is altered so that v may not be a forbidden vertex. Furthermore, $\Psi rec()$ is altered so that if the inclusion of an extension point causes a forbidden vertex to enter X, then no recursive call is made with that value of

X. Lastly, the condition for making recursive calls in $\Psi rec()$ is changed so that if the only possible choices for v are forbidden, then X is stored as a connected convex vertex set and no recursive calls are made.

5.5 Partial solutions using the Ψ algorithm

A *partial solution* is an intermediate stage in the process of enumerating all convex vertex sets in D. We show how a partial solution can be constructed and propose several different methods of expanding the partial solution to enumerate all convex vertex sets of D.

DEFINITION 17 *Given $A \subseteq V(D)$ for some DAG D, let a_i be the topologically last vertex in A. Let V_i be the set of vertices $a_{i+1}, \ldots, a_{|V(D)|}$.*

The augmentation *of A is the set $B = V_i \setminus (dom(A, D) \cup domBy(A, D)$. By Lemma 9 B is convex in D.*

The partial solution, ρ, of D is a set of pairs (A, B), such that $A \in \mathcal{S}_c(D)$, B is the argumentation of A, and $|\rho| = |\mathcal{S}_c(D)|$.

Lemma 9 *Given a DAG D, let $A \subseteq V(D)$. If $B \subseteq V(D)$ is the augmentation of A, then B is convex.*

Proof Suppose for contradiction that B is not convex, then there is a path $P = p_1, \ldots, v, \ldots p_k$ such that $p_1, p_k \in B$ and $v \notin B$.

Because $v \notin B$, then there is a path Q, which is either a $v - A$ path or a $A - v$ path.

In the former case, there is also a $p_1 - A$ path, which contradicts that $p_0 \in B$ and in the latter case there is a $A - p_k$ path, which contradicts that $p_k \in B$. Then we have a contradiction, as required. ◇

Lemma 10 *Given a DAG D, let A be in $\mathcal{S}_c(D)$ and let B be the augmentation of A. If $C \in \mathcal{S}(D_B)$, then $(A \cup C) \in \mathcal{S}(D)$.*

Algorithm 18 $\Psi(D)$: enumerating connected convex vertex sets with support for internal and external forbidden vertices

```
Ψ(D)
{
    Let U ← V(D) \ F(D)
    ∀a ∈ U
    {
        Ψrec({a}, U \ {a}, F(D))
        delete a from V(D) and U
    }
}

Ψrec(X, Y, F)
{
    if ((dom(X, D) ∩ Y) ∪ (domBy(X, D) ∩ Y)) \ F = ∅
    {
        store X
        return
    }
    v ← chooseV(X, Y, F)
    Ψrec(X, Y \ {v}, F)
    X ← X ∪ {v}
    X ← X ∪ (dom(X, D) ∩ domBy(X, D))
    if X ∩ F ≠ ∅
        Ψrec(X, Y \ X, F)
}

chooseV(X, Y, F)
{
    if ((domBy(X, D) ∩ Y) \ F) ≠ ∅
    {
        return v_i ∈ (domBy(X, D) ∩ Y) \ F with maximum i.
    }
    else
    {
        return v_i ∈ (dom(X, D) ∩ Y) \ F with minimum i.
    }
}
```

Proof C is convex in D because any path in D_B is a path in D. Suppose for contradiction that $(A \cup C)$ is not convex in D and that there is a path P from $A \cup C$ to $A \cup C$ that contains a vertex v in $V(D) \setminus (A \cup C)$. Because both A and C are convex in D, P must be either from A to C, or C to A. Then this is a contradiction by Definition 17 because $C \subseteq B$ and B is the augmentation of A. ◇

Lemma 11 *Given a DAG D, let X be in $\mathcal{S}(D) \setminus \mathcal{S}_c(D)$. If ρ is the partial solution of D, then there is precisely one A and C such that $(A, B) \in \rho$, $C \in \mathcal{S}(D_B)$, $A \cup C = X$.*

Proof Let x be the topologically last vertex in X. Let X_1 be the connected component of X such that $x \in X_1$. Because X_1 is a convex connected vertex set, there is precisely one $(A, B) \in \rho$ such that $A = X_1$.

Let X_2 be $X \setminus X_1$. X_2 is clearly convex because there are no $X_1 - X_2$ paths by convexity of X. We now show that $X_2 \subseteq B$. All elements in X_2 have no path to X_1 or from X_1. Moreover, no element of X_2 is topologically later than x. Then all elements of X_2 are in B and $X_2 \in \mathcal{S}(D_B)$. Then there exists an A, C such that $A \cup C = X$ for all X.

For contradiction, we now assume that there is also A', C' such that

$$(A', B') \in \rho, C' \in \mathcal{S}(D_{B'}), A' \cup C' = X, A' \neq A, B' \neq B.$$

By definition of an augmentation, A' must contain x because B' may not contain a vertex that is topologically later than any in A'. Then $A' = X_1$ and we have a contradiction as required. ◇

By Lemmas 11 and 10, the set $\mathcal{S}(D)$ can be constructed from the partial solution of D, by considering the convex vertex sets in the augmentation of the elements of $\mathcal{S}_c(D)$.

Index	X	Y
1	{3}	∅
2	{2}	{3}
3	{1,3}	{2}
4	{1}	{2}
5	{0,1,2,3}	∅
6	{0,1,3}	∅
7	{0,1,2}	∅
8	{0,1}	∅
9	{0,2}	∅
10	{0}	∅

Table 5.3 Partial solution for Example 15

5.5.1 Algorithms incorporating a partial solution

Algorithms based on partial solutions enable the user of an associated CAD tool to direct the search efficiently, save storage space and fine-tune the constraints placed on valid convex sets when more information is available about the generated instruction set.

Direction of search

A partial solution to enumerating all convex vertex sets would allow a user of an associated CAD tool the opportunity to control the enumeration process. By directing the algorithm to only explore certain parts of the partial solution, the user could concentrate the efforts of the toolchain on areas of the DAG that are likely to yield beneficial candidate instructions.

In this way the algorithm could work in tandem with a user, and take advantage of their knowledge of likely useful candidate instructions. If used in this way then our algorithm becomes a heuristic approach for candidate generation; however, the advantages in terms of the use of human intelligence and reduction of search space may well compensate for the possibility of missing a global optimal. Investigations into the effectiveness of this process compared to other heuristic methods are beyond the scope of this work and the possibilities

for this approach are discussed further in Chapter 8.

Storage

It can be noted that the number of connected convex vertex sets in D is much smaller than the total number of convex vertex sets (see Table 5.4 for some examples). This difference in size of solutions means that a partial solution will almost inevitably use less storage space than a fully enumerated set of candidate instructions. Efficient storage of candidate instructions can be of the utmost importance because a complete candidate instruction library may well contain millions of instructions. The only occasions that a partial solution will not use less storage space are extreme cases when the total number of connected convex vertex sets is close to the total number of convex vertex sets. However, this is only likely to occur when the DAG contains a Hamilton path.

Flexible restrictions

The intermediate stage allowed by a partial solution can allow a more dynamic flow of information between the stages of the instruction set customisation toolchain. For example, an instruction selection method may be only able to consider a limited set of 10,000 candidate instructions. This flexibility makes it simple to efficiently deliver, for example, all of the connected convex vertex sets and then a selection of the disconnected ones. Furthermore, it is also simple to deliver all convex vertex sets that consist of n connected components. This flexibility of a partial solution may be useful for particularly esoteric architectures or unusual situations.

5.5.2 Creating a partial solution

A partial solution for D can be generated from the set $\mathcal{S}_c(D)$ with a small amount of processing. In the Ψ algorithm (Algorithm 15) when a connected convex vertex set, X, is stored, the corresponding Y set is the augmentation

of X. By modifying the Ψ algorithm so that these sets are stored together, we have an efficient method for creating a partial solution.

Input	Vertices	Connected convex vertex sets	Convex sets
BF1Con	16	278	9,728
BF2	32	68,876	1,225,422
BF4Con	31	21,500	803,734
dijskstra1	15	211	848
dijskstra2Con	16	245	960
dijskstra3Con	19	2,475	135,040
patricia4Con	29	691,294	53,412,360
qsort1Con	17	3,912	65,120
qsort2Con	31	8,240	4,139,576
qsort3Con	29	24,780	6,069,360
sha	16	434	4,026
sha2	38	12,597	1,052,848
sha3	46	225,998	1,661,374
susan3	40	5,304,066	125,639,352
susan4Cxon	38	4,607,411	75,457,856
cjpeg1Con	39	485,718	8,338,040

Table 5.4 Comparison between numbers of convex vertex sets and connected convex vertex sets with no vertices forbidden

5.5.3 Reclaiming the full set of convex vertex sets from a partial solution

A range of methods have been developed to produce the full set of convex vertex sets from a partial solution. Each of these methods is passed a partial solution as an argument, and returns the set of convex vertex sets in D by iterating over the partial solution. Each element is completely processed before moving onto the next.

Brute force

Brute force is a naïve algorithm that finds the convex vertex sets of each Y set by comparing it with each convex vertex set already found.

For each (A, B) pair in the partial solution ρ, its B set is examined and compared to the A set of every other convex vertex set in the library. If the examined A set is a subset of the examined B set, then the union of the two A sets forms the A set of a new element of S which is added to the end of the array with the empty set as its augmentation. This approach is obviously inefficient but has the advantage that the original DDG need not be known to compute the full set of convex vertex sets. The $partialSolutionBrute(\rho)$ algorithm shown in Algorithm 19 returns all convex vertex sets as A values in T.

Algorithm 19 $partialSolutionBrute(\rho)$: processing a partial solution by brute force

```
partialSolutionBrute(ρ)
{
    Let T be a copy of ρ
    ∀(A, B) ∈ ρ, B ≠ ∅ do
        ∀(A′, B′) ∈ T do
            if A′ ⊆ B
            {
                T ← T ∪ {(A ∪ A′, ∅)}
            }
    return T
}
```

Assisted

The assisted method uses the **exhaustive** algorithm [PAI06] to find all the convex vertex sets of D_B for a given $(A, B) \in \rho$. Once a non-empty B set has been found, a modified version of the exhaustive algorithm in [PAI06] is executed to find all convex vertex sets of D_B. For each convex vertex set C of D_B, $(A \cup C) \in \mathcal{S}(D)$. The algorithm in [PAI06] was chosen for simple implementation, any other algorithm for enumerating convex subsets would have been equally suitable. This method is shown in Algorithm 20.

Algorithm 20 *partialSolutionAssisted*(ρ): processing a partial solution using a convex vertex set enumerating algorithm

```
partialSolutionAssisted(ρ)
{
    D ← ∅
    ∀(A, B) ∈ ρ, B ≠ ∅ do
        S(D_B) ← exhaustive(D_B)
        ∀C ∈ S(D_B) do
            D ← D ∪ (A ∪ C)
    return D
}
```

Search tree

The search tree method requires that call trees corresponding to the execution trace of the Ψ algorithm are available. These trees are used to build a search tree that allows efficient lookup of convex subsets.

The search tree (shown in Figure 5.3) and created from Example 15 is constructed from the call trees shown in Figure 5.2. These call trees are combined to form the larger search tree by adding vertices that connect different subtrees together. Internal vertices are labelled with a splitting vertex and leaf vertices are labelled with a convex vertex set.

Search trees created in this way have the property that if a search tree vertex t is labelled with a DAG vertex n, then all the convex vertex sets in the left subtree will contain the vertex n and all the convex vertex sets in the right subtree will not contain n. This property may be used to find all convex subsets of a given set in linear time complexity proportional to the number of subsets found.

A sample algorithm for outputting the convex subsets of a given vertex set is shown in Algorithm 21. The function $label(t)$ returns the pivot vertex attached to a search tree vertex t, whereas the $nextLeft(t)$ and $nextRight(t)$ return the left and right descendants of t respectively. Finally, the function $set(t)$ returns the convex set attached to the search tree vertex t. The overall search tree

Figure 5.3 An example of a completed Search Tree formed by combining the components of Figure 5.2

algorithm is shown in Algorithm 22.

Algorithms such as Algorithm 21 can be used to quickly find all of the convex subsets of a B set. Hence a full solution can be quickly built from a partial solution.

Each new convex vertex set found by this method must be immediately added to the search tree so that convex vertex sets of three or more connected components are found. Furthermore, because new convex vertex sets are added to the search tree when they are found, we must ensure that all convex vertex sets required for the expansion of an (A, B) pair already exist in the search tree. This is achieved by processing vertex sets in (A, B) in reverse order of creation.

5.6 Performance of algorithms

We present experimental data on the performance of the algorithms in this chapter. Appendix A describes the conditions of the tests and Appendix B describes the provenance of the data.

Algorithm 21 $convexSubsets(S, t)$: returning all convex subsets of a set S given a search tree rooted at t

```
convexSubsets(S, t)
{
    if (nextLeft(t) = nextRight(t) = ∅)
    {
        output set(t)
        return
    }
    if (label(t) ∈ S))
    {
        convexSubsets(S, nextLeft(t))
    }
    convexSubsets(S, nextRight(t))
}
```

Algorithm 22 $partialSolutionST(\rho, t)$: processing a partial solution using program search tree rooted at t

```
partialSolutionST(ρ, t)
{
    D ← ∅
    ∀(A, B) ∈ ρ, B ≠ ∅ do
        U ← convexSubsets(B, t)
        ∀U ∈ U do
        {
            D ← D ∪ (A ∪ U)
            addToSearchTree(A) ∪ U)
        }
    return D
}
```

Input	Vertices	Convex sets	Time **split**	Time Φ	Time $\Psi_{allsets}$	Time **exhaustive**
bf1	18	38,912	0.07	0.02	0.02	0.03
bf2	32	1,225,422	1.97	0.66	0.68	3.70
bf4Con	31	803,734	1.22	0.41	0.44	1.90
cjpeg3Con	24	564,000	0.93	0.26	0.28	0.81
qsort1	21	1,041,920	1.38	0.48	0.50	1.09

Table 5.5 Timings all sets, no forbidden vertices

Input	Vertices	Convex sets	Calls **split**	Calls Φ	Calls $\Psi_{allsets}$	Calls **exhaustive**
bf1	18	38,912	60,500	39,912	77,804	196,487
bf2	32	1,225,422	1,851,525	1,225,422	2,450,810	10,303,079
bf4Con	31	803,734	1,217,199	803,734	1,607,435	5,281,741
cjpeg3Con	24	564,000	870,014	564,000	1,127,974	3,254,935
qsort1	21	1,041,920	1,634,484	1,041,920	2,083,819	4,960,831

Table 5.6 Recursive calls all sets, no forbidden vertices

5.6.1 Enumerating all convex vertex sets

Table 5.5 and Table 5.6 show the performance of the four algorithms for enumerating all convex sets:Φ; $\Psi_{allsets}$;**split** and **exhaustive**. Figure 5.5 and Figure 5.4 show the performance of the algorithms on tree and lattice based DAGs of various sizes.

The performance gains by both the Φ and $\Psi_{allsets}$ algorithms over the **split** and **exhaustive** algorithms are substantial. The Φ algorithm performs slightly better than the $\Psi_{allsets}$ algorithm.

Enumerating all convex vertex sets subject to external forbidden vertices

This section gives results on the ability of the Φ and $\Psi_{allsets}$ algorithms to process test cases that include external forbidden vertices.

Table 5.7 and Table 5.8 show the performance of the four algorithms for enumerating all convex sets:Φ; $\Psi_{allsets}$; **split** and **exhaustive**. s The performance gains by both the Φ and $\Psi_{allsets}$ algorithms over the **split** and **exhaustive**

Figure 5.4 Performance of our algorithms against **split** and **exhaustive** on DAGs of lattice format

Figure 5.5 Performance of our algorithms against **split** and **exhaustive** on DAGs of tree format

algorithms are substantial. The Φ algorithm performs slightly better than the $\Psi_{allsets}$ algorithm.

Enumerating all convex vertex sets subject to external and internal forbidden vertices

Table 5.9 and Table 5.10 show the performance of the four algorithms for enumerating all convex sets:Φ; $\Psi_{allsets}$; **split** and **exhaustive**. In this experiment both external and internal vertices can be forbidden.

It is notable that the Φ algorithm quickly becomes unable to cope with the introduction of internal forbidden vertices and is unable to complete many of the test cases. However, the $\Psi_{allsets}$ algorithm remains efficient and continues to assert its dominance over the **split** and **exhaustive** algorithms.

5.6.2 Enumerating connected convex vertex sets

Tables 5.11 and 5.12 show the performance of the Ψ and **split** algorithms when enumerating connected convex sets. Figure 5.6 shows the performance of the algorithms on lattice-based DAGs of various sizes, whereas Figure 5.7 shows the performance of the algorithms on tree-based ones. It can be noted that the Ψ correctly enumerates all connected convex vertex sets in around half the time taken by the **split** algorithm.

The algorithms are also compared when enumerating connected convex vertex sets with external forbidden vertices; their relative performance in this case can be seen in Table 5.13 and Table 5.14.

5.6.3 Feasibility of partial solutions

The time taken by the SearchGraph and Assisted methods to generate a full set of convex vertex sets are compared to time taken by $Psi_{allsets}$ when applied to the synthetic tree graphs. In Figure 5.9 the experiment is performed on

Input	Vertices (forbidden)	Convex sets	Time **split**	Time Φ	Time $\Psi_{allsets}$	Time **exhaustive**
bitcnts1+	68(22)	319,865	0.57	0.24	0.42	1.43
bitcnts3+	55(19)	5,703	0.01	0.00	0.00	0.03
cjpeg3+	39(23)	11,967	0.02	0.00	0.02	0.02
gsm8+	41(13)	84,727	0.13	0.06	0.07	0.26
patricia4+	52(29)	3,194,983	5.79	2.60	4.07	5.82
susan3+	53(27)	489,215	0.77	0.37	0.60	1.58
sha3+	52(10)	334,388	0.56	0.26	0.30	0.90
rijndeal4+	52(15)	112,379	0.18	0.08	0.10	0.69

Table 5.7 Timings all sets, external forbidden vertices

Input	Vertices (forbidden)	Convex sets	Calls **split**	Calls Φ	Calls $\Psi_{allsets}$	Calls **exhaustive**
bitcnts1+	68(22)	319,865	490,237	319,865	639,684	4,339,400
bitcnts3+	55(19)	5,703	10,327	5,703	11,370	96,882
cjpeg3+	39(23)	11,967	31,654	11,967	23,918‘	88,795
gsm8+	41(13)	84,727	132,153	84,727	169,426	922,361
patricia4+	52(29)	3,194,983	7,319,026	3,194,983	6,684,853	23,063,249
susan3+	53(27)	489,215	838,187	489,215	978,404	4,756,387
sha3+	52(10)	334,388	511,010	334,388	668,734	2,502,745
rijndeal4+	52(15)	112,379	175,884	112,379	224,721	1,779,723

Table 5.8 Recursive calls all sets, external forbidden vertices

Input	Vertices (forbidden)	Convex sets	Time **split**	Time Φ	Time $\Psi_{allsets}$	Time **exhaustive**
bf4+	45(24)	8,527	0.01	0.17	0.01	0.04
bitcnts1+	68(40)	3,328	0.01	DNF	0.00	0.01
bitcnts2+	74(43)	3,454,311	2.18	DNF	1.42	7.62
cjpeg1+	65(39)	66,943	0.13	DNF	0.07	0.39
cjpeg4+	62(40)	17,695	0.03	DNF	0.02	0.05
fft2+	72(39)	7,340,063	4.78	DNF	1.44	16.89
patricia3+	73(44)	1,576,960	2.98	0.41	0.89	6.08

Table 5.9 Timings all sets, external and internal forbidden vertices

Input	Vertices (forbidden)	Convex sets	Calls **split**	Calls Φ	Calls $\Psi_{allsets}$	Calls **exhaustive**
bf4+	45(24)	8,527	20,451	240,943	86,301	114,406
bitcnts1+	68(40)	3,328	6,642	DNF	6,628	71,183
bitcnts2+	74(43)	3,454,311	5,689,187	DNF	7,225,205	55,131,601
cjpeg1+	65(39)	66,943	129,053	DNF	306,660	1,548,680
cjpeg4+	62(40)	17,695	133,159	DNF	212,058	198,103
fft2+	72(39)	7,340,063	11,941,593	DNF	20,709,403	85,661,935
patricia3+	73(44)	1,576,960	3,119,458	5,230,239	5,144,225	23,926,510

Table 5.10 Recursive calls all sets, external and internal forbidden vertices

Figure 5.6 Performance of Ψ against **split** on DAGs of lattice format for connected convex vertex sets

Figure 5.7 Performance of Ψ against **split** on DAGs of tree format for connected convex vertex sets

Input	Vertices	Convex sets	Time **split**	Time Ψ
FFT1	34	1,358,876	2.56	0.57
bf2	32	68,876	0.11	0.05
bf4Con	31	21,500	0.03	0.02
patricia4Con	29	691,294	1.16	0.47
qsort2Con	31	8,240	0.01	0.01
qsort3Con	29	24,780	0.03	0.02
sha2	38	12,597	0.02	0.01
sha3	46	225,998	0.34	0.19
susan3	40	5,304,066	8.38	4.04
susan4Con	38	4,607,411	8.28	3.47
cjpeg1Con	39	485,718	0.71	0.37
cjpeg3Con	24	15,734	0.03	0

Table 5.11 Timings for enumerating connected sets with no forbidden vertices

Input	Vertices	Convex sets	Calls **split**	Calls Ψ
FFT1	34	1,358,876	2,717,787	2,717,718
bf2	32	68,876	137,785	137,720
bf4Con	31	21,500	43,032	42,969
patricia4Con	29	691,294	1,382,618	1,382,559
qsort2Con	31	8,240	16,512	16,449
qsort3Con	29	24,780	49,590	49,531
sha2	38	12,597	25,233	25,156
sha3	46	225,998	452,043	451,950
susan3	40	5,304,066	10,608,173	10,608,092
susan4Con	38	4,607,411	9,214,861	9,214,784
cjpeg1Con	39	485,718	971,476	971,397
cjpeg3Con	24	15,734	31,493	31,444

Table 5.12 Recursive calls made when enumerating connected sets with no forbidden vertices

Input	Vertices (forbidden)	Convex sets	Time **split**	Time Ψ
bitcnts1Con+	60(15)	69,417	0.14	0.07
bitcnts2Con+	72(23)	4,658,237	8.97	4.58
bitcnts4+	54(18)	36,976	0.08	0.04
cjpeg1Con+	51(16)	210,911	0.37	0.19
cjpeg4Con+	57(19)	378,696	0.75	0.37
fft2Con+	66(27)	944,902	1.63	0.89
patricia3+	73(38)	207,856	0.46	0.2
sha3+	52(10)	125,806	0.21	0.11

Table 5.13 Timings for enumerating connected sets with external forbidden vertices

Input	Vertices (forbidden)	Convex sets	Calls **split**	Calls Ψ
bitcnts1Con+	60(15)	69,417	138,910	138,789
bitcnts2Con+	72(23)	4,658,237	9,316,570	9,316,425
bitcnts4+	54(18)	36,976	74,025	73,916
cjpeg1Con+	51(16)	210,911	421,890	421,787
cjpeg4Con+	57(19)	378,696	757,469	757,354
fft2Con+	66(27)	944,902	1,889,898	1,889,765
patricia3+	73(38)	207,856	415,824	415,677
sha3+	52(10)	125,806	251,675	251,570

Table 5.14 Recursive calls made when enumerating connected sets with external forbidden vertices

synthetic lattice graphs, and in Figure 5.8 the experiment is performed on synthetic tree graphs.

It is apparent from these results that the time taken to create and use a partial solution to enumerate all convex vertex sets is competitive with the time taken by the other algorithms presented in this chapter. This is an encouraging result and validates these methods as being practical.

5.7 Summary

An algorithm has been presented for fast and flexible enumeration of connected convex vertex sets of a DAG that performs favourably with the current fastest

Figure 5.8 Performance of methods for processing a partial solution on graphs of lattice format

Figure 5.9 Performance of methods for processing a partial solution on graphs of tree format

algorithms. Furthermore, several algorithms for enumeration of all convex vertex sets of a DAG have been presented. These include a fast, but inflexible algorithm and a slightly slower, but more flexible, approach.

The concept of a partial solution has been explored and methods have been demonstrated that can extract a full solution from a partial one in reasonable time.

It is clear that the choice of enumeration algorithm depends greatly on the requirements of the user and there are circumstances in which each of these algorithms is attractive.

Chapter 6

Exhaustive search under I/O constraints

This chapter presents an algorithm, *outAlgorithm*, for enumerating convex vertex sets subject to output constraints. It then presents the Ω family of three algorithms: Ω_{count}; Ω_{paths}; and $\Omega_{splitting}$, which enumerate convex vertex sets that correspond to candidate instructions whose inputs and outputs are limited by specified bounds.

The *outAlgorithm* algorithm forms the foundation for the Ω family of algorithms — each member of the Ω family modifies *outAlgorithm* so that only convex vertex sets that satisfy input constraints are stored. We give the worst case and average case complexity of each algorithm. Elements of the work in this chapter have previously appeared in [GJR+08b].

6.1 Overview

In Chapter 2 we noted that real processor implementations limit the number of simultaneous register reads and writes. As a result, a user may place an upper limit on the number of values that a valid candidate instruction may read or write during execution. Recall from Lemma 5 in Chapter 3 that when convex vertex sets are bounded by specific I/O constraints, the maximum number of convex vertex sets is polynomial in the size of the input DAG. As a result, algorithms designed to enumerate such convex vertex sets can achieve a lower

time complexity than the exhaustive algorithms shown previously.

In practice this allows significantly larger DAGs to be processed. For example, we could filter the convex vertex sets found by the Φ and Ψ algorithms in Chapter 5 so that only convex vertex sets that satisfied the I/O constraints were stored. However, our experiments showed that neither Φ or Ψ could process DAGs that contained more than 80 valid vertices. By contrast, the Ω family of algorithms can make use of the asymptotic advantage given by the constraint, and in Section 6.6, we shall report that all members of the Ω family process the benchmark 'sha' from the PY04 dataset, which has 1,811 vertices, in under 20 seconds for I/O constraints as large as 5/2.

6.2 Output sets

DEFINITION 18 *An* output set *is a set $V_{Out} \subseteq V(D)$, such that there is at least one convex vertex set $C \in \mathcal{S}(D)$ where $OUT(C, D) = V_{Out}$.*

The algorithms presented in this chapter first find all output sets in D that have cardinality less than or equal to $outConstraint$. Then for each output set V_{Out}, all the valid convex vertex sets C for which $OUT(C, D) = V_{Out}$ are found.

The number of output sets in D that have cardinality no greater than $outConstraint$ is much smaller than the total number of subsets with cardinality no greater than $outConstraint$. In Table 6.1, the relationship between the total number of sets and the number of output sets is shown on a variety of test cases for $outConstraint = 2$. It is clear that searching naïvely for output sets would be inefficient.

To efficiently enumerate output sets we utilise three properties of output sets to prune the search space. If a set X satisfies thee following three conditions, and Y is the convex closure of X, then Y is a convex set such that $OUT(Y, D) = X$.

Input	Vertices (valid)	Total sets	Output sets
BF2	32(23)	253	172
BF3	338(328)	53,628	5,743
BF4	35(27)	351	260
cjpeg1	43(31)	465	352
cjpeg2	161(148)	10,878	8,912
gsm2	328(276)	37,950	22,221
gsm3	446(401)	80,200	60,272
susan1	112(101)	5,050	3,039
susan2	207(202)	20,301	11,647

Table 6.1 Comparing numbers of output sets of no more than two elements with total number of sets with no more than two elements

Condition 1

If A is an output set, then $A \cap F(D) = \emptyset$. If $A \cap F(D) \neq \emptyset$ and $OUT(C, D) = A$, then $C \cap F(D) \neq \emptyset$ and C is not a valid convex vertex set.

Condition 2

If A is an output set, there is a path from each member of A to some vertex $f \in F(D)$, such that the path does not contain any other member of A[1].

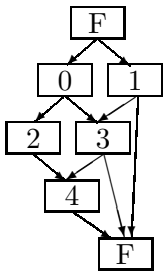
If this condition is not satisfied, then there is a vertex $a \in A$ such that all paths in D from a to any vertex $f \in F(D)$ contain a member of the set $A \setminus \{a\}$ and A is not an output set. A proof of this is given in Lemma 12.

Example 16 shows that if the output set is chosen to be {4, 3, 2}, then there would be no corresponding convex vertex sets. All convex vertex sets that include {4, 3, 2} have output set {4, 3}; the vertex 2 cannot be an output in this case because all paths from 2 to forbidden vertices go through {3, 4}.

Lemma 12 *If all paths in a DAG D from a vertex $a \in A$ to any vertex $f \in F(D)$ contain a member of the set $A \setminus \{a\}$, then there exists no convex vertex set C for which $OUT(C, D) = A$.*

[1]Recall from Chapter 2 that there is a path from a forbidden vertex to each vertex in the DAG, and at least one path from every vertex to a forbidden vertex.

Proof Let C be a convex vertex set of D such that $OUT(C, D) = A$. Because a is an output vertex it has a child $b \notin C$. There is a path from b to a vertex in $F(D)$ that, by definition, includes a vertex $c \in A \subseteq C$ so C cannot be convex. $\diamond$



Example 16: Two cases where a set is not a valid output set

Condition 3

If A is an output set, then A does not require the addition of another output vertex to satisfy convexity (Lemma 13). For example, if $A = \{1, 4\}$ in Example 16, then A is not an output set because $\{3\}$ must be included to maintain convexity in any convex vertex set produced. However, if $\{3\}$ is included, then $\{3\}$ will be an output of the convex vertex sets produced.

Lemma 13 *If there is a path in a DAG D from a vertex b to an element of $F(D)$ and that path does not contain at least one member of A, then:*

a) there is no convex vertex set C such that $b \in C$ and $OUT(C, D) = A$.

b) if b is also on a path between two elements of V_{Out}, there is no convex vertex set C such that, $b \in C$ and $OUT(C, D) = A$.

Proof a) Let P be a path in D such that $P = b, p_1, p_2, \ldots, p_i, \ldots, f$ where $f \in F(D)$ and $P \cap A = \emptyset$. Without loss of generality let $p_i \in C$ and $p_{i+1} \notin C$. So p_i is an output of C, but $p_i \notin A$, thus $OUT(C, D) \neq A$.

b) If b is on a path between two element of A then any set C such that

$OUT(C, D) = A$ must contain b due to convexity, however by (a) this is not possible. ◇

6.2.1 The $validOutputs()$ function

Algorithm 24 shows an efficient method for enumerating output sets of cardinality $\leq outConstraint$. The function is called with $validOutputs(\emptyset, |V(D)|, D)$ and stores all output sets of cardinality $\leq outConstraint$.

The $validOutputs()$ function is passed an output set V_{Out} and identifies vertices in $V(D)$ that could be added to V_{Out} to make a new output set. If a vertex s is suitable, then $V_{Out} \cup \{s\}$ is an output set, which is stored before a recursive call is made with $V_{Out} \cup \{s\}$ as the new output set. The variable $lastAdded$ is used to ensure that the same output set is not stored more than once, and the function will not make any recursive calls if $|V_{Out}| = outConstraint$.

The function $L(A, D)$(Algorithm 23) returns all vertices $v \in V(D)$ such that there is a directed path P from v to a member of $F(D)$ and P does not contain any element of A. The $L(A, D)$ function is used to ensure that only output sets are stored by $validOutputs()$. If V_{Out} is the current output set then the vertices contained in $L(V_{Out}, D)$ are exactly those vertices that can be added to V_{Out} by Lemma 12. Then $validOutputs()$ satisfies condition 2 by only choosing vertices in $L(V_{Out}, D)$ to add to the current output set.

The $L(A, D)$ function is also used by $validOutputs()$ to satisfy condition 3. If V_{Out} is the current output set and a vertex $s \in L(V_{Out}, D)$ dominates any of the vertices in V_{Out}, then additional checks are required. This is because the inclusion of s may force the inclusion of extra vertices to retain convexity. If any of these extra vertices are members of $L(V_{Out}, D)$, then condition 3 would be violated. To avoid this, $validOutputs()$ will only add a vertex that dominates V_{Out} if all of its direct descendants that also dominate V_{Out} are not members of $L(V_{Out}, D)$.

Algorithm 23 $L(V_{Out}, D)$: filtering vertices with regard to an output set

$L(V_{Out}, D)$
{
    $L \leftarrow \emptyset$
    $\forall (a, b) \in E(D)$ (in reverse lexicographic order)
        if $((b \in L, a \notin V_{Out}) \vee (a \in F(D)))$
        {
            $L \leftarrow L \cup \{a\}$
        }
    return L
}

Algorithm 24 $validOutputs(V_{Out}, lastAdded, D)$: enumerating output sets

$validOutputs(V_{Out}, lastAdded, D)$
{
    if $|V_{Out}| \leq outConstraint$
    {
        $temp \leftarrow L(V_{Out}, D) \cap dom(V_{Out}, D)$
        $temp2 \leftarrow temp \setminus X$, where $x \in X$ if there is an $x - temp$ path in D, of length > 1
        $valid \leftarrow L(V_{Out}, D) \setminus temp2$.
        $\forall n_i \in valid, i < lastAdded$
        {
            store $V_{Out} \cup \{n_i\}$
            $validOutputs(V_{Out} \cup \{n_i\}, i, D)$
        }
    }
}

Lemma 14 *a) The $L(V_{Out}, D)$ function (Algorithm 23) has time complexity $O(|E(D)|)$.*

b) The $validOutputs(V_{Out}, lastadded, D)$ function (Algorithm 24) has time complexity $O(|E(D)|\cdot|V(D)|^{outConstraint})$ when called as $validOutputs(\emptyset, |V(D)|, D)$

.

Proof a) The function $L(V_{Out}, D)$ requires that the set of edges are iterated over (an $O(|E(D)|)$ operation), during which up to $|E(D)|$ vertices may be added to the set L. Thus the function $L(V_{Out}, D)$ has complexity $O(|E(D)|)$.

b) The function $validOutputs()$ performs the following operations. The set $valid$ is created in $O(|E(D)|)$ time and has size at most $|E(D)|$. For each v in $valid$, the sets $newF$ and X can be formed in $O(|E(D)|)$ time and this computation can be charged to the recursive call $recurseF\Omega_{out}(X \cup \{v\}, newF, D)$. Thus a single call to $validOutputs()$ is $O(|E(D)|)$. Because the $validOutputs$ algorithm finds each output set exactly once, it can make no more than $|V(D)|^{outConstraints}$ recursive calls, when enumerating output sets with no more than $outConstraint$ elements. Thus $validOutputs(\emptyset, |V(D)|, D)$ function has complexity $O(|E(D)| \cdot |V(D)|^{outConstraint}$. ◇

6.2.2 Enumerating convex vertex sets subject to output constraints

Algorithm 25 shows the *outAlgorithm* algorithm, which is an algorithm for enumerating the convex vertex sets of a DAG that satisfy output constraints. It iterates over the set of output sets that have been stored by $validOutput()$. For each output set V_{Out}, *outAlgorithm* will store all convex vertex sets C, such that $OUT(C, D) = V_{Out}$.

The *outAlgorithm* algorithm has the same approach and basic functionality as the Ω family of algorithms. Later sections will show the different ways that the Ω family influence the structure of this algorithm.

The key insight of the *outAlgorithm* algorithm is that if $OUT(C, D) = V_{Out}$, then all vertices in $L(V_{Out}, D)$ can safely be made forbidden by Lemma 15. This reduces the size of the search space considerably, particularly when there are a small number of outputs some distance apart.

The *outAlgorithm* algorithm forms the set $newF = L(V_{Out}, D)$ and the set X is initialised as the convex closure of V_{Out}. Then $X \subseteq C$ and $newF \cap C = \emptyset$ for any C such that $OUT(C, D) = V_{Out}$.

The *outAlgorithm* algorithm passes the X and $newF$ sets to a recursive function $recurseF\Omega_{out}()$, which uses a similar divide and conquer strategy to the Ψ algorithm that was presented in Section 5.2. During each call, $recurseF\Omega_{out}()$ chooses a vertex that is not a member of X or $newF$, and makes one recursive call in which the vertex is added to X, and one recursive call in which the vertex (along with all vertices that have a path to the splitting vertex) is added to $newF$. This vertex is known as the *splitting vertex*. If there are no available splitting vertices, then $recurseF\Omega_{out}()$ stores the current X set and terminates.

Lemma 15 *Let $C \subseteq V(D)$ be convex in D. Iff $OUT(C, D) = V_{Out}$, then $C \cap L(V_{Out}, D) = \emptyset$.*

Proof First assume that $OUT(C, D) = V_{Out}$. For contradiction let s be the topologically last vertex in $C \cap L(V_{Out}, D)$. Because $s \in L(V_{Out}, D)$ there is a path, P say, in $V(D) \setminus V_{Out}$ from s to a forbidden vertex. Let $P = \{s, p_1, \ldots, f\}$ and note that $p_1 \notin C$. Because s is the topologically last vertex in $C \cap L(V_{Out}, D)$, and it has an edge to p_1, then s is an output vertex of C. However, because $OUT(C, D) = V_{Out}$ and $V_{Out} \cap L(V_{Out}, D)$ is empty by definition, then we have a contradiction, as required.

Now assume $C \cap L(V_{Out}, D) = \emptyset$. For contradiction let $a \in OUT(C, D) \setminus V_{Out}$. Because $a \in OUT(C, D)$ it has a path to a forbidden vertex not containing an element in V_{Out}(Lemma 12). Then $a \in L(V_{Out}, D)$ by definition of $L()$ so

we have a contradiction, as required. ◇

Algorithm 25 *outAlgorithm* : enumerating convex vertex sets under output constraints

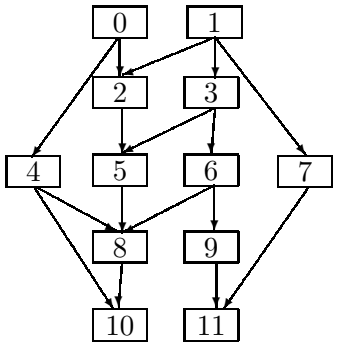
```
Ω_Out(D)
{
  ∀V_Out stored by validOutputs(∅, |V(D)|, D)
  {
    newF ← L(V_Out, D)
    X ← V_Out ∪ (dom(V_Out) ∩ domBy(V_Out))
    recurseFΩ_out(X, newF, D)
  }
}

recurseFΩ_out(X, newF, D)
{
  if (dom(X, D) \ newF) = ∅
  {
    store X
    return
  }
  Let v be the vertex v_i ∈ (dom(X, D) \ newF) such that i is maximum.
  recurseFΩ_out(X ∪ {v}, newF, D)
  recurseFΩ_out(X, newF ∪ {v} ∪ dom(v, D), D)
}
```

Example

Example 17 is used to show the operation of the *outAlgorithm* algorithm given the output set $V_{Out} = \{8, 9\}$, and $F(D) = \{0, 1, 10, 11\}$. A trace of the calls made by $recurseF\Omega_{out}(\{8, 9\}, \{0,\ 1,\ 4,\ 7,\ 10,\ 11\}, D)$ is shown in Figure 6.1. It successfully finds all convex vertex sets whose outputs set is {8, 9}.

Firstly, the vertices $\{4, 7\}$ are added to $newF$ because they are in the set returned by $L(\{8, 9\}, D)$. The set X is $\{8, 9\}$ because there are no paths between members of V_{Out}, and the set $newF$ is {0, 1, 4, 7, 10, 11}.

To illustrate the potential savings associated with pruning the search space by use of the $L()$ function, we consider the situation in which V_{Out} in Example 17 is changed to {5, 9}. Then 2 is the only possible choice of splitting vertex



Example 17: DAG to demonstrate execution of Ω_{count} and *outAlgorithm* with $V_{Out} = \{8, 9\}$ and $F(D) = \{0, 1, 10, 11\}$

because $L(\{5,\ 9\}, D) = \{0,\ 1,\ 3,\ 4,\ 6,\ 7,\ 8,\ 10,\ 11\}$. Then only two recursive calls are made and the convex vertex sets $\{2,\ 5,\ 9\}$ and $\{5,\ 9\}$ are stored.

6.2.3 Restricting by input constraints with the Ω family

We require the *outAlgorithm* algorithm to be modified so that it only stores convex vertex sets that obey a given input constraint. It would be possible to add an input test just before each convex vertex set is stored so that only sets that passed the test were stored. However, such an approach is inefficient because it still requires the algorithm to visit every set that satisfies the output constraint.

The following sections present more efficient ways of limiting the search space. Each member of the Ω family gives a different modification to the basic *outAlgorithm* algorithm. Each of these modifications significantly reduce the number of recursive calls needed for exhaustive enumeration of valid convex vertex sets under both input and output constraints.

6.3 The Ω_{count} algorithm

This section presents the Ω_{count} algorithm, which generates all convex vertex sets that obey both input and output constraints. The *outAlgorithm* algorithm shown earlier is modified to only store convex vertex sets that satisfy an input

Figure 6.1 Call tree for $recurseF\Omega_{out}(\{8, 9\}, \{0, 1, 4, 7, 10, 11\}, D)$ when called on Example 17

constraint. This modification prunes recursive calls based on the number of permanent inputs to a convex vertex set. Examples and asymptotic proofs follow.

6.3.1 Adding input constraints

During the execution of $recurseF\Omega_{out}$ in Algorithm 25, vertices can be added to X and $newF$, but never removed. So no vertex in $newF$ can be removed from $newF \cap IN(X, D)$ and hence $newF \cap IN(X, D)$ can only increase in size. If $c = |newF \cap IN(X, D)|$, then all convex vertex sets found by further recursive calls have at least c inputs. If $c > inConstraint$, then the recursive branch in question can be pruned because it will produce no useful convex vertex sets.

Adding this pruning criterion to the *outAlgorithm* algorithm gives the Ω_{count} algorithm(Algorithm 26) for enumerating convex vertex sets under output and input constraints.

The splitting vertex always has an edge to a vertex in X. This ensures that $|newF \cap IN(X, D)|$ increases in the recursive call in which the splitting vertex is made forbidden: this is necessary to satisfy proof of complexity (Lemma 18).

Example

We use Example 18 to show the advantages of the input pruning performed by the Ω_{count} algorithm.

Consider the case where $V_{Out} = \{19\}$ and $inConstraint = 2$. In this situation, the *outAlgorithm* algorithm would make 49 calls to enumerate 25 convex vertex sets, but only one of those convex vertex sets would satisfy the input constraint. By contrast, the Ω_{count} algorithm would only perform the 9 calls shown in Figure 6.2 because of its use of the pruning criterion.

Example 18: Advantage of pruning by input constraints $V_{Out} = X = \{19\}$, $F(D) = \{0, 1, 2, 3, 4, 5, 6, 7, 20\}$

Figure 6.2 Call tree for $recurseF\Omega_{count}(\{19\}, \{0,\ldots,8,20\}, D)$ when called on Example 18 with $inConstraint = 2$

Algorithm 26 Ω_{count} : enumerating convex vertex sets under I/O constraints

```
Ω_count(D)
{
  ∀V_Out stored by validOutputs(∅, |V(D)|, D)
  {
    Let newF ← L(V_Out, D)
    X ← V_Out ∪ (dom(V_Out) ∩ domBy(V_Out))
    recurseFΩ_count(X, newF, D)
  }
}

recurseFΩ_count(X, newF, D)
{
  if |IN(X, D) ∩ newF| > inConstraint
    return
  if (dom(X, D) \ newF) = ∅
  {
    store X
    return
  }
  Let v be the vertex v_i ∈ (dom(X, D) \ newF) such that i is maximum.
  recurseFΩ_count(X ∪ {v}, newF, D)
  recurseFΩ_count(X, newF ∪ {v} ∪ dom(v, D), D)
}
```

6.3.2 Time complexity

This section gives an upper bound for the time complexity of Ω_{count} . Lemma 18 shows that Ω_{count} has time complexity $O(|V(D)|^{outConstraint+inConstraint+1})$.

Lemma 16 *For any X, D, and $newF$, with $inConstraint = I$, the number of calls made by $recurseF\Omega_{count}(X, newF, D)$ will be $O(n^{I+1})$ where $n = |V(D)|$.*

Proof The calls made by $recurseF\Omega_{count}()$ form a binary call tree. For each internal vertex, one child represents the addition of some vertex a and the other child represents the rejection of the vertex a. If the vertex a is rejected, then for all further recursive calls in the subtree a will be an external input. Each vertex in the tree will be reachable from the root vertex by a unique sequence of 'select' and 'reject' edges.

It follows that the depth of the call tree can be no more that n and that there is no vertex in the call tree that is reachable from the root by a path that contains more than I 'reject' edges because the external input test would have prevented the recursive call. Then by Lemma 17, there are no more than

$$\sum_{b=1}^{I+1} \binom{n}{b}$$

vertices in the call tree.

Because each $\binom{n}{b}$ term in the summation has complexity $O(n^{I+1})$ and there are I such terms, then $\sum_{b=1}^{I+1} \binom{n}{b}$ has complexity $O(n^{I+1})$.

Because the number of vertices in the recursive call tree is of $O(n^{I+1})$, the number of recursive calls made by $recurseF\Omega_{count}()$ is $O(n^{I+1})$ for any X,D and $newF$. ◇

Lemma 17 *Given a fully populated binary tree T of depth n in which child vertices can be reached from parent vertices by either following a left link or a right link, the number of vertices in T that can be reached from the root of T by taking no more than L left links 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 links used to reach it from the root. The vertices in T are in bijection with all sequences of left and right links of length at most $n-1$.

We are interested in those sequences that contain no more than L left links, and we note that $\binom{a}{b}$ will return the number of sequences of length a that contain exactly b left links. It follows that $\binom{a}{b}$ is also 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 links is

$$\sum_{b=0}^{L} \left(\sum_{a=0}^{n-1} \binom{a}{b} \right)$$

.

The 'Christmas Stocking Theorem' [Wei] gives $\sum_{a=0}^{n-1} \binom{a}{b} = \binom{n}{b+1}$, so the number of vertices in T that can be reached from the root of T by taking no more than L left links and unlimited right links is

$$\sum_{b=1}^{L+1} \binom{n}{b}$$

$\diamond$

Lemma 18 *The overall time complexity of Ω_{count} is $O(n^{U+I+1})$, where U is the output constraint, I is the input constraint, and $n = |V(D)|$.*

Proof The number of recursive calls made by $validOutputs()$ is limited by n^U. The complexity of each call is dominated by the call made to $\Omega_{count}(X, newF, D)$,

which has linear time complexity but may make of $O(n^{I+1})$ recursive calls by Lemma 16. Because there are n^U calls to a function of complexity $O(n^{I+1})$, then the complexity of the overall algorithm is $O(n^{U+I+1})$. ◇

Worst-case bounds verses the real world

When engineering implementations of algorithms it is occasionally the case that achieving a theoretical bound compromises the performance on average case examples.

Consider the choice of splitting vertex in Ω_{count}. From the point of view of correctness all vertices in $V(D) \setminus (newF \cup X)$, are equivalent, but there may be a performance advantage to choosing either the first such vertex or the last.

Consider Example 17 with $inConstraint = 2$. If the splitting vertex is the topologically last vertex not in $newF$, then Ω_{count} will make the 14 calls shown in Figure 6.3, and find no valid convex vertex sets. However, if the splitting vertex were to be the topologically *first* such vertex, then Ω_{count} would make the nine calls shown in Figure 6.4[2].

This difference in the number of calls is that if the topologically first vertex is added to X, then all of its direct ancestors (which are forbidden) are inputs. This results in $IN(X, D) \cap newF$ growing rapidly when vertices are added to X. However, because the direct ancestors of the splitting vertex may already be inputs to X, then it is possible that $IN(X, D) \cap newF$ does not change.

Choosing the first available vertex to be the splitting vertex will make $IN(X, D) \cap newF$ grow rapidly in most cases. This can reduce the number of recursive calls made and hence reduce the time taken by the algorithm.

By contrast, when the topologically last vertex is added to X, it is unlikely that there are any direct ancestors that are forbidden and $IN(X, D) \cap newF$ grows slowly when vertices are added to X. However, when the splitting vertex

[2]Note that if the topologically first vertex is choosen, then the algorithm may have to include additional vertices to maintain convexity; however it will not have to ever add more than one vertex to $newF$

is made forbidden, it is guaranteed that $IN(X) \cap newF$ will grow by exactly one each time. This guarantee allows the proof of polynomial time complexity in Lemma 18 to hold.

This thesis uses the polynomial formulation of the Ω_{count} algorithm because one of the goals of this thesis is to build a more rigorous theoretical foundation in this area.

6.3.3 Limitations of the Ω_{count} algorithm

Although it can be guaranteed that no valid convex vertex sets are lost by the pruning criterion, it is not strong enough to prune unnecessary recursive branches early enough. There are even cases where the pruning criterion removes no recursive calls at all.

For example, if the $recurseF\Omega_{count}$ algorithm is called on Example 17 such that $inConstraint = 3$, then it will make 14 recursive calls to enumerate 8 convex vertex sets. These recursive calls are identical to the ones shown in Figure 6.1. However, if $recurseF\Omega_{count}$ is called on the same example with $inConstraint = 2$, then the algorithm will still make 14 recursive calls but it will not find any valid convex vertex sets. The recursive behaviour in this case is shown in Figure 6.3.

We now present the Ω_{paths} algorithm which recognises when a combination of $newF$ and X can lead to no valid convex vertex sets and stops the recursive call immediately.

6.4 The Ω_{paths} algorithm: using flow-based pruning to reduce total recursive calls

The Ω_{count} algorithm presented in the previous section is efficient, but its pruning criterion is not optimal — this can result in the algorithm chasing up 'blind alleys'. This section presents a different method for pruning the recursive calls

Figure 6.3 Call tree for $recurseF\Omega_{count}(\{8,9\},\{0,1,4,7,10,11\},D)$ when called on Example 17 with $inConstraint = 2$

Figure 6.4 Call tree for $recurseF\Omega_{count}(\{8,9\},\{0,1,4,7,10,11\},D)$ when called on Example 17 with $inConstraint = 2$ and choosing first available splitting vertex

of the *outAlgorithm* algorithm by examining paths from forbidden vertices to members of X. We introduce the use of such paths to prune the search space of *outAlgorithm*, with several examples to motivate the work and to highlight limitations that must be overcome. Later sections will apply techniques from involving flows in networks to the problem. Finally, the intuition behind the flow approach is distilled into the implementation-friendly Ω_{paths} algorithm that is based on path finding in an augmented graph. Proofs are given for the Ω_{paths} algorithm.

Lemma 19 *Let $X \subseteq V(D) \setminus F(D)$. Let P be a path $\{p_0, p_1, \ldots, p_i\}$ where $p_0 \in F(D)$ and $p_i \in X$. If $C \subseteq V(D)$ is convex in D such that $X \subseteq C$ then $(P \cap IN(C, D)) \neq \emptyset$.*

Proof Let j be the smallest j such that $p_j \in C$. Because P is a path and $p_0 \notin C$, then $p_{j-1} \notin C$ and has an edge to a member of C. Thus $p_{j-1} \in IN(C, D)$ and $P \cap IN(C, D) \neq \emptyset$. ◇

6.4.1 Motivation

Consider Example 17 with $V_{Out} = X = \{8, 9\}$, $L(V_{Out}, D) = \{0, 1, 4, 7, 10, 11\}$ and $inConstraint = 2$. It was shown in Section 6.3.3 that Ω_{count} requires 14 recursive calls to determine that there are no valid convex vertex sets containing X for an $inConstraint$ of two. Now consider the following paths: $0, 2, 5, 8$; $1, 3, 6, 9$; and $4, 8$. By Lemma 19 any convex vertex set that contains X will have at least as many input vertices as there are paths. If the Ω_{count} algorithm had been able to recognise these paths, then it would have terminated the recursion after a single call. This use of paths from members of $newF$ to members of X forms the basis for the Ω_{paths} algorithm that is presented in this section.

In general, if there are n paths from a forbidden vertex to a member of X in D such that vertices in $V(D) \setminus X$ may only appear in one path, then all

convex vertex sets that contain X will have at least n inputs. By finding all such paths, some branches can be pruned.

It is quite possible to modify the *outAlgorithm* algorithm in such a way that it performs a greedy search for these paths at the beginning of each recursive call. Such an approach is practical and provides an improvement to the pruning criterion of the Ω_{count} algorithm. However, there are some areas of inefficiency that are explored in the following section. Refinements to the approach that eliminate these inefficiencies are then presented.

6.4.2 Inefficiencies of a greedy approach

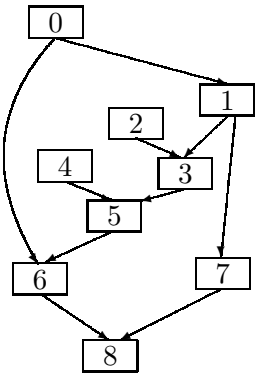
Pruning the recursive calls made by the *outAlgorithm* algorithm by greedily finding directed paths from members of $newF$ to members of X is insufficient to prune all unnecessary recursive calls for two main reasons.

Firstly, the use of directed $newF - X$ paths does not take into account the need to add vertices to satisfy convexity. Consider Example 19 with $V_{Out} = X = \{6, 7\}$ and $F(D) = \{0, 2, 4\}$. There are a maximum of 2 paths from forbidden vertices to vertices in X[3] (the paths $\{0, 1, 7\}$ and $\{4, 5, 6\}$ are used in this example). However, there is no convex vertex set C with $X \subseteq C$ such that C has only 2 inputs because if the vertex $\{1\}$ is included in C, then the vertex set $\{3, 5\}$ is also included. If $\{3, 5\}$ is included, then the vertices $\{2, 4\}$ become inputs.

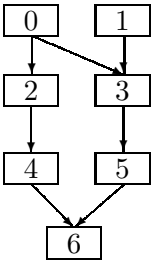
Secondly, finding a maximum number of paths is not trivial. Consider Example 20 with $V_{Out} = X = \{4, 5\}$ and $F(D) = \{0, 1, 6\}$. There is at most two paths ($0, 2, 4$ and $1, 3, 5$) in the DAG, but if the path $0, 3, 5$ is found first, then no more paths can be found. This may result in unnecessary recursive calls being made.

The following sections present a way of using flows in networks to improve the pruning of recursive calls by overcoming both of these problems.

[3]Subject to our condition that each vertex in $V(D) \setminus X$ can appear in only one path



Example 19: DAG demonstrating the need for path-based methods to account for included vertices, $V_{Out} = X = \{6, 7\}$ and $F(D) = \{0, 2, 4\}$



Example 20: Difficulty in finding maximum number of paths, with $V_{Out} = X = \{4, 5\}$ and $F(D) = \{0, 1, 6\}$

6.4.3 Modelling the problem using networks

Flows on networks [Len90] (See Section 2.1.6) can be used to find a set of $newF - X$ paths such that any set $C \supset X$ has at least one input vertex for each path in the set. This will improve the pruning of recursive calls by overcoming the problems associated with DAGs such as Examples 19 and 20.

This section shows how finding the maximum value of flow in a network can be used to develop efficient pruning criteria for our algorithms. We give some examples of the use of network flow, before going on to create a new, path-based, algorithm based on the flow principles. We later prove the correctness of the path-based version.

Building the network

Given a DAG D, a selection X, output set V_{Out} and forbidden set $newF$ such that $V_{out} \subseteq X \subseteq (V(D) \backslash newF)$, $newF \subseteq L(V_{Out}, D)$, and $V_{Out} = OUT(X, D)$, one can construct a network N_D in such a way that the flow of data through a DDG is modelled. Each path that data may take into a custom instruction will add flow in the network.

The network N_D is constructed as follows.

For each vertex $n_i \in V(D)$, two vertices, $n_i a$ and $n_i b$, are created in N_D and an edge $(n_i a, n_i b)$ with capacity 1 is added. Note that such edges are the only limited capacity edges in the network.

For each edge $(n_x, n_y) \in E(D)$, the edges $(n_x b, n_y a)$ and $(n_y a, n_x a)$ are added to N_D both with infinite capacity, *unless* $n_x \in V_{Out}$.

The vertices s and t are added to the network. For each vertex $n_i \in newF$ the edge $(s, n_i a)$ is added with infinite capacity. For each vertex $n_i \in X$ the edge $(n_i a, t)$ is added with infinite capacity.

The network N has the property that the value of a maximal $s - t$ flow in N is equal to the minimum number of inputs required by a convex vertex set C such that $OUT(C, D) = V_{Out}, X \subseteq C$.

Finding a maximal flow in a network

The well known Ford-Fulkerson algorithm [FF56, FF57] is used to find the maximum value of $s - t$ flow. In general, the Ford-Fulkerson algorithm is not guaranteed to terminate, but under the conditions used here[4] the runtime of Ford-Fulkerson is bounded by $O(|E(N_D)| \cdot inConstraint)$.

The operation of the Ford-Fulkerson algorithm is simple: while there is a path from s to t in a network N such that there is available capacity on all edges in the path, a unit of flow is added to each edge on the path. We note

[4]The capacities on edges are all integers (edges with infinate capacity can be safely replaced by edges with any integer capacity that is greater than the value of *inConstaint*) and the process can be stopped when there is a value of flow greater than the value of *inConstraint*.

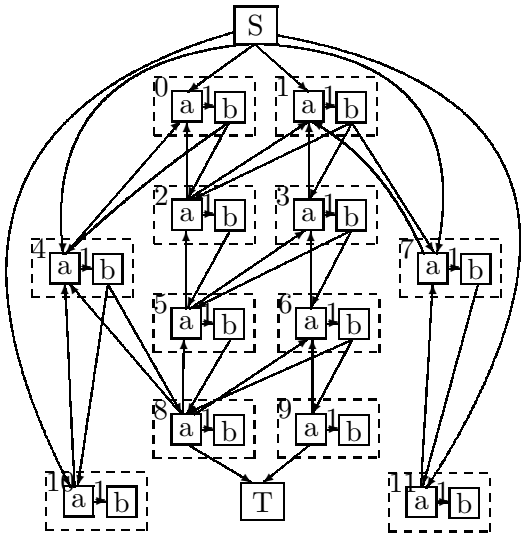


Figure 6.5 Network for Example 17

that if an edge $f_{ab} = k$, then we may consider that c_{ba} has increased by k when searching for paths.

When the Ford-Fulkerson algorithm has found a maximal $s - t$ flow, the flow can be converted back into a set of paths that each contain an input. We then have an efficient way of finding a lower bound on the number of inputs required for any convex vertex set containing X.

Examples

Example 17 is used to show the improvement on the pruning criteria by this use of networks. We let $\{8, 9\}$ be the output set for the following.

The network N_D is constructed and shown in Figure 6.5. The maximum value of flow in the network is 3. An example of such a flow is (S, $0a$, $0b$, $2a$, $2b$, $5a$, $5b$, $8a$, T; S, $4a$, $4b$, $8a$, T; S, $1a$, $1b$, $3a$, $3b$, $6a$, $6b$, $9a$, T). If the input constraint were 2, then there can be no valid convex vertex sets for these values of X and $newF$. This is a significant improvement on the seven recursive calls that Ω_{count} required.

Using flows in networks provides a solution to the problems demonstrated by Examples 19 and 20.

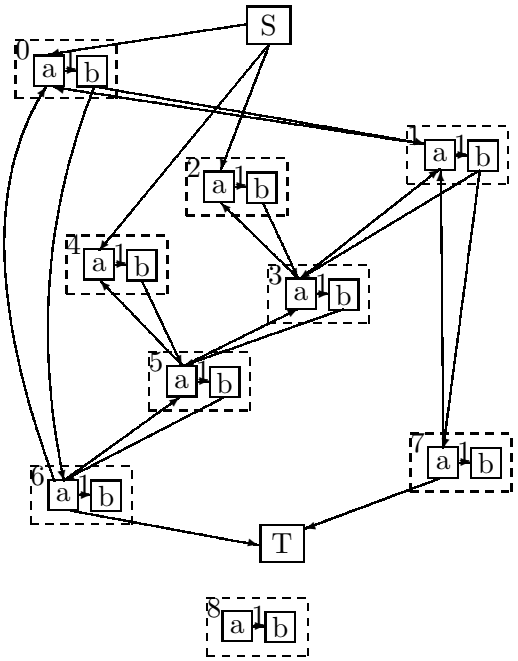


Figure 6.6 Network for Example 19

Now consider Example 20. For the output and X set $\{4, 5\}$ and set vertices $\{0, 1\}$, the network N_D is shown in Figure 6.7. The Ford-Fulkerson algorithm begins by finding a $s - t$ path with available capacity on each edge. The first such path found is S, $0a$, $0b$, $3a$, $3b$, $5a$, T. This is equivalent to the 0, 3, 5 path found earlier; however because a flow of 1 on the edge $(a, b) \in N_D$ is equivalent to a capacity of 1 on an edge $(b, a) \in N_D$, the path S, $1a$, $1b$, $3a$, $1b$, $2a$, $2b$, $4a$, T is now available. When the flow in N_D is converted back into path edges, we get the two paths $\{0, 2, 4\}$ and $\{1, 3, 5\}$ — because the Ford-Fulkerson algorithm automatically adapts paths there is no possibility of them being chosen in the wrong order.

Now consider Example 19. For the output and X set $\{6, 7\}$ and forbidden set $\{0, 2, 4\}$ the network N_D is shown in Figure 6.6.

The Ford-Fulkerson algorithm finds the paths (S, $0a$, $0b$, $6a$,T; S, $4a$, $4b$, $5a$, $5b$, $6a$, T; $2a$, $2b$, $3a$, $1a$, $1b$, $7a$, T) before terminating. This flow is equivalent to the path edges shown in Figure 6.8 (edges used in paths are solid lines, unused

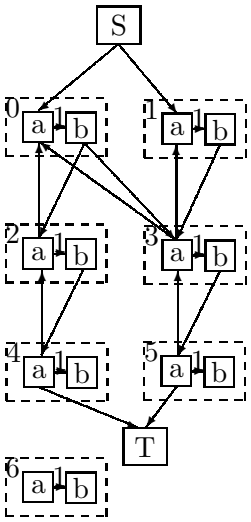


Figure 6.7 Network for Example 20

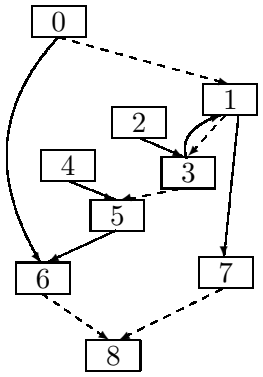


Figure 6.8 Paths in Example 19 found by use of a maximal flow

edges are dashed).

The paths $0, 6$ and $4, 5, 6$ are both visible, but we also have the undirected path $2, 3, 1, 7$, which allows for vertices that must be included for convexity by following edges in reverse. This use of undirected paths allows the correct number of paths to be found.

The method of pruning recursive calls by total flow in the network allows a tighter pruning criterion to be built into an algorithm.

Drawbacks of use of flows in networks

Although the Ford-Fulkerson algorithm exhibits a low time complexity, the constants of proportionality are large and the creation of each network is computationally intensive. However, the operation of the Ford-Fulkerson algorithm in this special case and the intuition behind the use of flows can be reduced down to a simple path-finding algorithm. The following section gives the path-finding algorithm in detail, and presents the Ω_{paths} algorithm for enumeration of convex vertex sets under I/O constraints by use of the path-finding principles. Because the flow based algorithms have been presented as an intermediate stage in the development of the Ω_{paths} algorithm, we do not give proofs of correctness for the network approach.

6.4.4 Input path sets

In this section we define an *input path set*, which can be used by the Ω_{paths} algorithm to prune unnecessary recursive calls. The Ω_{paths} algorithm is a modification of the *outAlgorithm* algorithm, so we have the following sets:

V_{Out} : V_{Out} is the output set for any valid convex vertex sets that will be created,

X : X is a convex set, such that $V_{Out} \subseteq X$ and $OUT(X, D) = V_{Out}$,

$newF$: $newF$ is a set of vertices that may appear in no valid convex vertex set. $L(V_{Out}, D) \subseteq newF$.

Let $E_{A,B}$ be a set of edges such that $(a, b) \in E_{A,B}$ if $(a, b) \in E(D), a \notin A, b \notin B$. Let $E^{-1}_{A,B}$ be the set of edges such that $(b, a) \in E^{-1}_{A,B}$ if $(a, b) \in E_{A,B}$. It is easy to check that if D is a directed acyclic graph $E_{A,B}$ and $E^{-1}_{A,B}$ are disjoint. If there is an $a - b$ edge (path) using only edges from $E_{A,B}$ we say that there is a *forward* edge (path) $a - b$. If there is an $a - b$ edge (path) using only edges from $E^{-1}_{A,B}$ we say that there is an *inverse* edge (path) $a - b$.

Given a DAG D, and two sets A and B, $D'_{A,B}$ is the directed graph with $V(D'_{A,B}) = V(D)$, $E(D'_{A,B}) = E_{A,B} \cup E^{-1}_{A,B}$.

DEFINITION 19 *Given a DAG D and the disjoint sets X and F, we call a set of paths $\mathcal{P}$ in $D'_{X,F}$ from F to X an input path set if:*

(1) $\mathcal{P} \neq \emptyset$.

(2) *no forward edge may be contained in two distinct paths in $\mathcal{P}$.*

(3) *no pair of distinct forward edges in $E(\mathcal{P})$ have the same initial vertex.*

The Ω_{paths} algorithm create an input path set $\mathcal{P}$ from $newF$ to X in each recursive call. If $|\mathcal{P}| > inConstraint$, then that recursive call can safely be terminated. This modification allows the Ω_{paths} algorithm to tell if a recursive call is worth pursuing instantly. Lemma 20 shows that this pruning is safe.

The following sections discuss how an input path set is formed and the computational complexities that its use involves.

Lemma 20 *Given a DAG D and the disjoint sets A and B such that $A, B \subset V(D)$, let $\mathcal{P}$ be an input path set from A to B. If S is a set such that $B \subseteq S \subseteq V(D) \setminus A$ and $OUT(S, D) = OUT(B, D)$, then $|IN(S, D)| \geq |\mathcal{P}|$.*

Proof Any path from A to $B \subseteq S$ in $D'_{A,B}$ contains at least one forward edge of the form $(i, s) \in E(D'_{AB})$ such that $s \in S$ and $i \notin S$. Then either (i, s) is a forward edge and $i \in IN(S, D)$ or (i, s) is an inverse edge and so (s, i) being a forward edge implies that $s \in OUT(S, D)$.

In the latter case, $s \in OUT(S, D) = OUT(B, D)$ is contrary to the construction of the inverse edges. Thus $i \in IN(S, D)$.

Let $\mathcal{P} = \{P_1, \ldots, P_k\}$ s.t $k = |\mathcal{P}|$, and let $P_j = \ldots, i_j, s_j, \ldots, i_j \notin S, s_j \in S, 1 \leq j \leq k$.

Note that for any i_a, i_b such that $1 \leq a \leq b \leq n$, if $i_a = i_b$, then $(i_a, s_a) = (i_b, s_b)$ by Definition 19(3). Thus $a = b$ by Definition 19(2). Then $i_1, \ldots, i_k$ are distinct. Because they are all in $IN(S, D)$, $|IN(S, D)| \geq |\mathcal{P}|$. $\diamond$

6.4.5 Constructing an input path set

The Ω_{paths} algorithm requires that it can quickly construct an input path set from $newF$ to X, and that there is no larger input path set from $newF$ to X.

We require the following functions. Let $\mathcal{P} = \{W_1, \ldots, W_k\}$ be a set of walks. Let $M \subset \mathbb{N} \times \mathbb{N}$.

For any walk $W_i = w_{i1}, \ldots, w_{ij}$, we define the functions

$$B((t,u), W_i, M) = |\{s | t = w_{is}, u = w_{i(s+1)}, (i,s) \notin M\}|.$$

(Note that $B((t,u), W_i, \emptyset)$ is the number of instances of the edge (t,u) on W_i.)

$$f^+(t, W_i, M) = \sum_{u \in V(D)} B((t,u), W_i, M)$$

$$f^-(t, W_i, M) = \sum_{u \in V(D)} B((u,t), W_i, M)$$

$$ff(t, W_i, M) = f^+(t, W_i, M) - f^-(t, W_i, M)$$

(Note that $ff(t, W_i, \emptyset)$ is the number of edges leaving t on W_i less the number of edges entering t on W_i.)

We generalise these functions over sets of walks as follows.

$$ff(t, \mathcal{P}, M) = \sum_{0 < i \leq k} ff(t, W_i, M),$$

$$f^+(t, \mathcal{P}, M) = \sum_{0 < i \leq k} f^+(t, W_i, M),$$

$$f^-(t, \mathcal{P}, M) = \sum_{0<i\leq k} f^-(t, W_i, M),$$

and

$$B((t,u), \mathcal{P}, M) = \sum_{0<i\leq k} B((t,u), W_i, M).$$

Clearly, if W_i is an $a-b$ walk and $t \in V(D) \setminus \{a,b\}$, then $f(t, W_i, \emptyset) = 0$.

Let $\bar{A} = V(D) \setminus A$. Suppose that W_i is an $A-B$ walk and A, B are disjoint subsets of $V(D)$. Because W_i starts in A and finishes in $\bar{A}$ there is exactly one more $A-\bar{A}$ edge contained in W_i than there are $\bar{A}-A$ edges contained in W_i. So we have

$$\sum_{t\in A} ff(t, W_i, \emptyset) = 1$$

and similarly

$$\sum_{t\in B} ff(t, W_i, \emptyset) = -1.$$

So

$$\sum_{a\in A} ff(a, \mathcal{P}, \emptyset) = |\mathcal{P}| = -\sum_{b\in B} ff(b, \mathcal{P}, \emptyset).$$

Finally, for any walk or walk set we may write $ff(z, P)$ instead of $ff(z, P, \emptyset)$ and $B((a,b), P)$ instead of $B((a,b), P, \emptyset)$.

To show how the Ω_{paths} algorithm constructs an input path set, $\mathcal{P}$, we define a $\mathcal{P}$-compliant walk.

DEFINITION 20 *Given a directed acyclic graph D and disjoint sets $A, B \subset V(D)$, let $\mathcal{P}$ be an input path set from A to B, in D'_B.*

Let W be a walk in D'_B, $W \notin \mathcal{P}$. We say that W is $\mathcal{P}$-compliant if:

(1) there is no forward edge in both $E(W)$ and $E(\mathcal{P})$,

(2) if (a,b) and (a,c) are distinct forward edges in $E(W)$, then (b,a) or (c,a) is also in $E(W)$,

(3) for each forward edge $(x,y) \in E(W)$ we have at least one of:

(a) x has no forward edge out of it in $E(\mathcal{P})$;

(b) the inverse edge $(y,x) \in E(\mathcal{P})$;

(c) the preceding edge (w,x) on W is an inverse edge and (x,w) is a forward edge in $E(\mathcal{P})$.

Lemma 21 *Let $\mathcal{P}$ be a walk set, and $M \subset \mathbb{N} \times \mathbb{N}$.*

There is some N such that $M \subseteq N \subset \mathbb{N} \times \mathbb{N}$, $ff(t,\mathcal{P},M) = ff(t,\mathcal{P},N)$ for all $t \in V(D)$, and for any $a,b \in V(D)$, if $B((a,b),\mathcal{P},N) > 0$, then $B((b,a),\mathcal{P},N) = 0$, Moreover, if $B((a,b),\mathcal{P}) > 0$ and $B((b,a),\mathcal{P}) > 0$, then $B((a,b),\mathcal{P},N) < B((a,b),\mathcal{P})$, and $B((b,a),\mathcal{P},N) < B((b,a),\mathcal{P})$.

Proof If there is no s,t such that $B((s,t),\mathcal{P},N) > 0$ and $B((t,s),\mathcal{P},N) > 0$, then we take $N = M$, as required.

We then suppose that there is some s,t such that $B((s,t),\mathcal{P},N) > 0$ and $B((t,s),\mathcal{P},N) > 0$. Then there are the walks W_a and W_b say such that $w_{bc} = w_{a(d+1)} = s$,$w_{b(c+1)} = w_{ad} = t$, and $(a,c),(b,d) \notin M$.

We form $M' = M \cup \{(c,a),(d,b)\}$. Note that $ff(t,\mathcal{P},M) = ff(t,\mathcal{P},M')$ for all $t \in V(D)$ because exactly one instance of an edge leaving a (b) is in $M' \setminus M$ and exactly one instance of an edge entering a (b) is in $M' \setminus M$.

We repeat the process of finding such (s,t) and (t,s) pairs until we have some set $O, M' \subseteq O$, such that $ff(t,\mathcal{P},M) = ff(t,\mathcal{P},O)$, and if $B((a,b),\mathcal{P},O) > 0$, then $B((b,a),\mathcal{P},O) = 0$ for any $a,b \in V(D)$.

Thus $O = N$, as required. $\diamond$

Lemma 22 *Let $\mathcal{Q}$ be a set of $A-B$ walks where A,B are disjoint subsets of $V(D)$. Let M be a set such that if*

(a) for all $t \in V(D)$*:*

1. *if* $ff(t, \mathcal{Q}) > 0$*, then* $ff(t, \mathcal{Q}, M) \geq 0$*,*
2. *if* $ff(t, \mathcal{Q}) < 0$*, then* $ff(t, \mathcal{Q}, M) \leq 0$*,*
3. *if* $ff(t, \mathcal{Q}) = 0$*, then* $ff(t, \mathcal{Q}, M) = 0$*.*

(b) $\sum_{t \in A} ff(t, \mathcal{Q}, M) > 0$*,*

then there is an $A - B$ *path* P *s.t. for any* (s, t)*,* $B((s, t), P) \leq B((s, t), \mathcal{Q}, M)$*. Furthermore, if* P *is an* $a - b$ *path, then* $ff(a, \mathcal{Q}, M) > 0$ *and* $ff(b, \mathcal{Q}, M) < 0$*.*

Proof Because $\sum_{t \in A} ff(t, \mathcal{Q}, M) > 0$, there is some vertex $a \in A$ such that $ff(a, \mathcal{Q}, M) > 0$. Choose such an a and let $T = a, \ldots, z$ say, be a longest walk such that $B((s, t), T) \leq B((s, t), \mathcal{Q}, M)$ for any (s, t) on T. So for any s, $B((s, z), T) \leq B((s, z), \mathcal{Q}, M)$, we have $f^-(z, T) \leq f^-(z, \mathcal{Q}, M)$. Note that $f^-(z, T) \geq f^+(z, T)$.

First suppose that $ff(z, \mathcal{Q}, M) = 0$ and hence $a \neq z$. Then $f^-(z, T) = f^+(z, T) + 1$. Because $ff(z, \mathcal{Q}, M) = 0$, we have $f^+(z, \mathcal{Q}, M) = f^-(z, \mathcal{Q}, M)$. Hence

$$f^+(z, \mathcal{Q}, M) = f^-(z, \mathcal{Q}, M) \geq f^-(z, T) = f^+(z, T) + 1 > f^+(z, T).$$

So

$$f^+(z, \mathcal{Q}, M) > f^+(z, T).$$

Then there is some u such that $B((z, u), T) < B((z, u), \mathcal{Q}, M)$. Then $T' = a, \ldots, z, u$ is a walk in D'_B such that for any (s, t), $B((s, t), T') \leq B((s, t), \mathcal{Q}, M)$. T' is longer than T contrary to the choice of T. Then $ff(z, \mathcal{Q}, M) \neq 0$.

Now suppose that $ff(z, \mathcal{Q}, M) \geq 1$. So $f^+(z, \mathcal{Q}, M) > f^-(z, \mathcal{Q}, M)$. Because $f^-(z, T) \geq f^+(z, T)$ we have

$$f^+(z, \mathcal{Q}, M) > f^-(z, \mathcal{Q}, M) \geq f^-(z, T) \geq f^+(z, T).$$

So

$$f^+(z, \mathcal{Q}, M) > f^+(z, T).$$

Then there is some u such that $B((z,u),T) < B((z,u),\mathcal{Q},M)$. Then $T' = a,\ldots,z,u$ is a walk in D'_B such that for any (s,t), $B((s,t),T') \leq B((s,t),\mathcal{Q},M)$ and T' is longer than T contrary to the construction of T. Then $ff(z,\mathcal{Q},M) \not> 0$.

Then $ff(z,\mathcal{Q},M) < 0$. $ff(z,\mathcal{Q},M) < 0$ can only hold if z is the final vertex on a walk in $\mathcal{Q}$. All walks in $\mathcal{Q}$ terminate in B so $z \in B$, and T is an $A-B$ walk.

Let P be an $A-B$ path such that $E(P) \subseteq E(T)$. Then P is an $A-B$ path such that for any (s,t), $B((s,t),P) \leq B((s,t),\mathcal{Q},M)$. $\diamond$

Lemma 23 *Given a DAG D and the disjoint sets A and B such that $A, B \subset V(D)$, let $\mathcal{P}$ be an input path set from A to B. Let W be a $\mathcal{P}$-compliant walk in D'_B from A to B. If $|\mathcal{P}| = k$, then there exists an input path set $\mathcal{S}$ from A to B such that $|\mathcal{S}| = k+1$.*

Proof Let $\mathcal{Q} = \mathcal{P} \cup \{W\}$. Then $|\mathcal{Q}| = k+1$, because $W \notin \mathcal{P}$ by Definition 20. By Lemma 21, we may assume that there is a set M such that, $ff(t,\mathcal{Q}) = ff(t,\mathcal{Q},M)$ for all $t \in V(D)$, and for any $a,b \in V(D)$, if $B((a,b),\mathcal{Q},M) > 0$, then $B((b,a),\mathcal{Q},M) = 0$, and if $B((a,b),\mathcal{Q}) > 0$ and $B((b,a),\mathcal{Q}) > 0$, then $B((a,b),\mathcal{Q},M) < B((a,b),\mathcal{Q})$, and $B((b,a),\mathcal{Q},M) < B((b,a),\mathcal{Q})$.

We note that by definitions 19 and 20, if (s,t) is a forward edge in D'_B, then $B((s,t),\mathcal{Q}) \leq 1$. Furthermore, if $B((t,s),\mathcal{Q}) > 0$, then, by Lemma 21, $B((s,t),\mathcal{Q},M) = 0$.

We inductively define $M_0 \subset \ldots \subset M_k \subset \mathbb{N}\times\mathbb{N}$, and the corresponding paths $P_1,\ldots,P_{k+1}$, which have the property that if (s,t) is a forward edge, then (s,t) appears in at most one P_l as follows.

Let $M_0 = M$ so M_0 satisfies (a) and (b) of Lemma 22.

Suppose $0 \leq i \leq k$ and we have defined $M_0 \subset \ldots \subset M_i$, which satisfy (a) and (b) of Lemma 22 and $P_1, \ldots, P_i$, which have no pairwise common forward edge. By Lemma 22 there exists an $a-b$ path, P say, $a \in A$, $b \in B$, such that for any (s,t), $B((s,t),P) \leq B((s,t),\mathcal{Q},M_i)$, $ff(a,\mathcal{Q},M_i) > 0$ and $ff(b,\mathcal{Q},M_i) < 0$. Let $P_{i+1} = P$.

Furthermore, let $M_i' \subset \mathbb{N} \times \mathbb{N}$ be a smallest set such that for each edge (s,t), contained in $E(P_{i+1})$, there exists an $(l,m) \in M_i'$ where $W_l \in \mathcal{Q}$, $W_l = \ldots, w_{lm}, w_{l(m+1)}, \ldots$, such that $w_{lm} = s, w_{l(m+1)} = t$ and $(l,m) \notin M_i$. Let $M_{i+1} = M_i \cup M_i'$.

We show no forward edge in $E(P_{i+1})$ is in $E(P_h), h \leq i$. If (s,t) were also in $E(P_h)$, then $(x,y) \in M_h$ where $W_l \in \mathcal{Q}$, $W_x = \ldots, w_{xy}, w_{x(y+1)}, \ldots$, such that $w_{xy} = s, w_{x(y+1)} = t$. If (s,t) were a forward edge then by definitions 19 and 20, $W_x = W_l$, so $x = l$ and $y = m$. Then $(l,m) \in M_h$ and because $M_h \subset M_i$, $(l,m) \in M_i$, which contradicts the choice of l and m.

Clearly $M_i \subset M_{i+1}$, we now show that M_{i+1} satisfies Lemma 22(a) and (b).

It is easy to check that $ff(a,\mathcal{Q},M_i) = ff(a,\mathcal{Q},M_{i+1}) - 1$, $ff(b,\mathcal{Q},M_i) = ff(b,\mathcal{Q},M_{i+1}) + 1$, and $ff(t,\mathcal{Q},M_i) = ff(t,\mathcal{Q},M_{i+1})$ for all $t \in V(D) \setminus \{a,b\}$. So (a) of Lemma 22 hold for M_{i+1}.

We further note that

$$\sum_{t \in V(D)} ff(t,\mathcal{Q},M_{i+1}) = \sum_{t \in V(D)} ff(t,\mathcal{Q},M_i) \; - \sum_{t \in V(D)} ff(t,P_i)$$

$$= \left(\sum_{t \in V(D)} ff(t,\mathcal{Q},M_i) \right) - 1$$

and so

$$\sum_{t \in V(D)} ff(t,\mathcal{Q},M_{i+1}) = \left(\sum_{t \in V(D)} ff(t,\mathcal{Q},M_0) \right) - i$$

.

Then

$$\sum_{t \in V(D)} ff(t, \mathcal{Q}, M_{i+1}) = k + 1 - i.$$

Thus M_{i+1} satisfies (a) and (b) of Lemma 22 and we have $M_0 \subset \ldots \subset M_k$ that satisfy Lemma 22 and corresponding paths $P_1, \ldots, P_{k+1}$, as required.

Because a forward edge may only appear in one of $P_1, \ldots, P_{k+1}$ and each $A-B$ path in D'_B contains at least one forward edge, then each of $P_1, \ldots, P_{k+1}$ is distinct and $|\{P_1, \ldots, P_{k+1}\}| = k + 1$. We denote the set $\{P_1, \ldots, P_{k+1}\}$ as $\mathcal{R}$.

We now show that $\mathcal{R}$ is an input path set. Clearly, $\mathcal{R} \neq \emptyset$, and no forward edge may be contained in two distinct paths in $\mathcal{R}$ so Definition 19(1) and (2) are satisfied.

We now show that $\mathcal{R}$ satisfies Definition 19(3). Suppose (u, v) and (u, w) are distinct forward edges in $E(\mathcal{R})$. Any edge in $E(\mathcal{R})$ is also in either $E(\mathcal{P})$ or $E(W)$. Because $\mathcal{P}$ is an input path set, both cannot be in $E(\mathcal{P})$.

Suppose both edges are in $E(W)$, then by Definition 20 either (v, u) or (w, u) is in $E(W)$. Without loss of generality let (v, u) be in $E(W)$, so $B((w, u), \mathcal{Q}) > 0$ Because W is a $\mathcal{P}$-compliant walk, $B((u, w), \mathcal{Q}) = 1$. But by choice of M, if $B(w, u), \mathcal{Q}, M) \geq 0$, then $B((u, w), \mathcal{Q}, M) = 0$, so (u, w) could not appear in any path in $\mathcal{R}$ and we have a contradiction.

Then both edges cannot be in $E(W)$. Then without loss of generality we assume that $(u, v) \in E(\mathcal{P})$ and $(u, w) \in E(W)$.

Then (u, w) is a forward edge in $E(W)$ that satisfies Definition 20(3b) or (3c).

Suppose that Definition 20(3b) holds for (u, w). Then (w, u) is in $E(\mathcal{P})$ so $B((w, u), \mathcal{Q}) > 0$ and $B((u, w), \mathcal{Q}) = 1$. But by choice of M, $B(w, u), \mathcal{Q}, M) \geq 0$ and $B((u, w), \mathcal{Q}, M) = 0$, so (u, w) could not appear in any path in $\mathcal{R}$ and we have a contradiction.

Now suppose that Definition 20(3c) holds for (u, w). Then (v, u) is in $E(\mathcal{P})$ and $B((v, u), \mathcal{Q}) > 0$ and that $B((u, v), \mathcal{Q}) = 1$. But by construction of M, $B((v, u), \mathcal{Q}, M) \geq 0$ and $B((u, v), \mathcal{Q}, M) = 0$, so (u, v) could not appear in any path in $\mathcal{R}$ and we have a contradiction.

Thus there can be no such (u, v) and (u, w) edges and $\mathcal{R}$ is an input path set of size $k + 1$ and we take $\mathcal{R} = \mathcal{S}$. $\diamond$

If $\mathcal{P}$ is an input path set from A to B and W is a $\mathcal{P}$-compliant walk from A to B, then by Lemma 23, the Ω_{paths} algorithm can construct a larger input path set, by first constructing a compliant walk set. By repeatedly finding such paths, input path sets of the largest possible size can be efficiently built.

DEFINITION 21 *An input path set $\mathcal{P}$ from A to B is a* maximal input path set *if there is no $\mathcal{P}$-compliant walk from A to B.*

6.4.6 The Ω_{paths} algorithm

Algorithm 27 shows the Ω_{paths} algorithm. The algorithm is largely similar to Ω_{count} although the external input checking has been replaced with the finding of $newF$-X paths.

6.4.7 Time complexity of the Ω_{paths} algorithm

By Lemma 24, finding a $\mathcal{P}$-compliant walk for an input path set $\mathcal{P}$ requires $|V(D)| + |E(D)|$ time. Each recursive call of the $recurseF\Omega_{paths}(X, newF, D)$ may search for a $\mathcal{P}$-compliant walk up to $inConstraint$ times. This searching dominates the remaining operations in $recurseF\Omega_{paths}(X, newF, D)$ so it requires $|V(D)| + |E(D)|$ time, not including recursive calls. It is clear that the Ω_{count} algorithm would never prune a recursive subtree that the Ω_{paths} algorithm would not, because every external input to X identified by the Ω_{count} algorithm would be in an input path set used by the Ω_{paths} algorithm. Moreover, because the Ω_{count} and Ω_{paths} algorithms have the same criteria for selecting

Algorithm 27 Ω_{paths} : enumerating convex vertex sets under I/O constraints using $newF - X$ paths

```
Ω_paths(D)
{
  For all V_Out stored by validOutputs(∅, |V(D)|, D)
  {
    Let newF ← L(V_Out, D)
    X ← V_Out ∪ (dom(V_Out) ∩ domBy(V_Out))
    recurseFΩ_out(X, newF, D)
  }
}

recurseFΩ_paths(X, newF, D)
{
  if |P(X, newF, D)| > inConstraint
    return
  if (dom(X, D) \ newF) = ∅
  {
    store X
    return
  }
  Let v be the vertex v_i ∈ (dom(X, D) \ newF) such that i is maximum.
  recurseFΩ_paths(X ∪ {v}, newF, D)
  recurseFΩ_paths(X, newF ∪ {v} ∪ dom(v, D), D)
}
```

a splitting vertex, the Ω_{paths} algorithm cannot make more recursive calls than the Ω_{count} algorithm would.

By Lemma 16, the upper bound on the number of recursive calls made by the Ω_{paths} algorithm is $|V(D)|^{inConstraint+outConstraint+1}$. Thus the time complexity of Ω_{paths} is $O((|V(D)| + |E(D)|)^{inConstraint+outConstraint+1}$.

Although the Ω_{paths} algorithm has a worse time complexity than the Ω_{count} algorithm, the improved pruning criterion that make a dramatic difference to the number of recursive calls made on real-world examples.

Lemma 24 *Let $\mathcal{P}$ be an input path set from A to B. In time $O(|V(D)| + |E(D)|)$ we can find a $\mathcal{P}$-compliant walk or determine that it does not exist. If it does not exist, we can also find the set S and T for $\mathcal{P}$ in time $O(|V(D)| + |E(D)|)$.*

Proof We will define a directed graph D' as follows. Let R contain all vertices in D that have a forward edge out of them in $E(\mathcal{P})$. Let the vertex set of D' be $V(D') = V(D) \cup \{r' \mid r \in R\}$ (that is we duplicate all vertices in R). For all edges $(u, v) \in E(D)$ add the following edges to D'.

(R1): If (u, v) is a forward edge and $(v, u) \notin E(\mathcal{P})$ and $u \in R$, then add (u', v) to D'.

(R2): If (u, v) is a forward edge and $(v, u) \in E(\mathcal{P})$ or $u \notin R$, then add (u, v) to D'.

(B1): If (u, v) is an inverse edge and $(v, u) \in E(\mathcal{P})$, then add (u, v) and (u, v') to D'.

(B2): If (u, v) is an inverse edge and $(v, u) \notin E(\mathcal{P})$, then add (u, v) to D'.

We use depth first search to find a (A, B)-path in D' if it exists. First assume that such a path P' exists. Replacing vertices of the form u' with u, we obtain a walk W in D, which we will show is a $\mathcal{P}$-compliant walk. Because we

are in exactly one of the cases (R1), (R2), (B1) or (B2) above, we note that W does not contain the same edge (u, v) twice (as P' does not contain the same vertex twice).

Assume that (u, v) is a forward edge on the walk W. If the corresponding edge on P' was (u', v) (implying it was created in (R1), then the edge into u' on P' is of the form $(w, u')'$, as (B1) is the only case where we add edges into a vertex of the form u'. Therefore (w, u) is an inverse edge and $(u, w) \in E(\mathcal{P})$, so the forward edge (u, v) satisfies the properties for a $\mathcal{P}$-compliant walk. Now assume that the edge on P' was (u, v) (implying we used (R2), when considering (u, v)), then $(v, u) \in E(\mathcal{P})$ or $u \notin R$, so again (u, v) satisfies the properties for a $\mathcal{P}$-compliant walk. Therefore W is a $\mathcal{P}$-compliant walk in D.

Now assume that some $\mathcal{P}$-compliant walk, W', in D exists. Let $W' = w_1 w_2 w_3 \cdots w_l$. If $w_i w_{i+1}$ is a forward edge and $w_i \in R$ and $w_{i+1} w_i \notin E(\mathcal{P})$ then let $p_i = w_i'$ otherwise let $p_i = w_i$ for all $i = 1, 2, \ldots, l-1$. We will show that $P = p_1 p_2 \cdots p_l$ is a walk from A to B in D'. If (p_i, p_{i+1}) has the form (w_i', w_{i+1}) then (p_i, p_{i+1}) was indeed generated in (R1).

If (p_i, p_{i+1}) has the form (w_i, w_{i+1}') then (p_i, p_{i+1}) was generated in (B1), because W is a $\mathcal{P}$-compliant walk, we have that $(w_{i+1}, w_i) \in E(\mathcal{P})$ is a forward edge.

If (p_i, p_{i+1}) has the form (w_i, w_{i+1}), then (p_i, p_{i+1}) was generated in (B1) or (B2) if (w_i, w_{i+1}) is an inverse edge. Otherwise it was generated in (R2).

Because W is a $\mathcal{P}$-compliant walk in D, (p_i, p_{i+1}) cannot have the form (w_i', w_{i+1}'). Because we have $p_1 = w_1$(not $p_1 = w_1'$), we note that P is indeed a walk from A to B in D', we take P' to be the internal path of P.

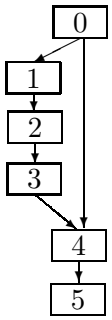
We have now shown that there exists a $\mathcal{P}$-compliant walk from A to B in D if and only if there exists a (A, B)-path in D'. This gives us the claimed time complexity because $|V(D')| \leq 2|V(D)|$ and $|E(D')| \leq 2|E(D)|$.

If there is no (A, B)-path in D', then it is not difficult to find the set of vertices S', such that there is an (A, s')-path in D' if and only if $s' \in S'$. Note

that if $u' \in S'$ for some $u \in V(D)$, then $u \in S'$, as if the edge (w, u') exists in D' then (w, u) also exists. Note that $S = S' \cap V(D)$ is the desired S from the statement of this lemma. Analogously we can find T. $\diamond$

6.5 The $\Omega_{splitting}$ algorithm: intelligent vertex selection for asymptotic superiority

Although an improved pruning criterion has been presented, there are still cases where a large number of avoidable recursive calls are made.



Example 21: Motivational example for improving choice of splitting vertex, $X = V_{Out} = \{4\}$, $F(D) = \{0, 5\}$

Consider Example 21. Given an input constraint of 1, and an output set of $\{4\}$, there is only one convex vertex set that matches the constraints ($\{1, 2, 3, 4\}$). However, the Ω_{paths} algorithm requires seven recursive calls to find this convex vertex set (as shown in Figure 6.9).

If $\{1\}$ has been chosen to be the splitting vertex in call a, then there would have been only three calls to $recurseF\Omega_{paths}$. Moreover, if the algorithm had recognised that there was only one convex vertex set, then that set could have been stored immediately.

This section shows how sophisticated criteria can be used to direct the choice of splitting vertices. The resulting algorithm guarantees that, if there is only one valid set that can be produced, it will be stored with no further recursive calls. If there is more than one remaining valid set, the splitting vertex will be

Figure 6.9 Call tree for $recursiveF\Omega_{paths}(\{4\}, \{0, 5\}, D)$ when called on Example 21 with $inConstraint = 1$

chosen in such a way that there is at least one convex vertex set that contains the splitting vertex and one that does not. This ensures that for the $\Omega_{splitting}$ algorithm, if there are n valid sets that have V_{Out} as their output set, then at most $2n$ recursive calls will be required to enumerate them.

6.5.1 S and T sets

The S and T sets for a particular maximal input path set, $\mathcal{P}$, from $newF$ to X, have a number of useful properties.

DEFINITION 22 *Given a directed graph D and a maximal input path set $\mathcal{P}$ from A to B, define the T set of $\mathcal{P}$ such that if there is a $\mathcal{P}$-compliant walk from a vertex t to B, then $t \in T$.*

DEFINITION 23 *Given a directed graph D and a maximal input path set $\mathcal{P}$ from A to B, define the S set of $\mathcal{P}$ such that if there is a $\mathcal{P}$-compliant walk in D from a member of A to a vertex s, then $s \in S$. Note that $A \subseteq S$ because every vertex has a $\mathcal{P}$-compliant walk to itself.*

Firstly, we note that T is convex in D (Lemma 27) and has the same output set as X (Lemma 29). Moreover $|IN(T, D)| = |\mathcal{P}|$ (Lemma 30). So if $|\mathcal{P}| \leq inConstraint$, then T is a valid convex set.

Secondly, we note that the vertex set $V(D) \setminus S$ is convex in D (Lemma 28) and has the same output set as X (Lemma 29). Moreover $|IN(V(D) \setminus S, D)| = |\mathcal{P}|$ (Lemma 31). So if $|\mathcal{P}| \leq inConstraint$, then $V(D) \setminus S$ is a valid convex set.

This set of properties for T and S will allow the $\Omega_{splitting}$ algorithm to precisely direct its choice of splitting vertex.

Lemma 25 *Consider a DAG D, two sets X and $newF$ such that $X \subset V(D)$, $F(D) \subseteq newF \subseteq V(D)$, and $newF \cup X = \emptyset$. If $\mathcal{P}$ is a maximal input path set with associated S and T sets then $newF \cap T = \emptyset$.*

Proof Without loss of generality let a be in $newF \cap T$. There is a $\mathcal{P}$-compliant walk P from a to a member of X. P is a $\mathcal{P}$-compliant walk from a member of $newF$ to a member of X. The presence of a $\mathcal{P}$-compliant walk contradicts that $\mathcal{P}$ is a maximal input path set, and there is a contradiction. ◇

Lemma 26 *Let $X \subseteq V(D) \setminus newF$ be a convex set such that $OUT(X, D) = V_{Out}$. Let $\mathcal{P}$ be a maximal input path set from $newF$ to X, with associated T set. Let $p, b \in V(D) \setminus L(V_{Out}, D)$. If p is a direct ancestor of b and $p \in T$ then $b \in T$.*

Proof Given that $X \subseteq T$, first suppose $p \in X$. Then $b \in T$ by convexity of X, as required.

Now suppose $p \notin X$. Because $p \in T$, there exists a $\mathcal{P}$-compliant walk, $W = p, \ldots, x$, $x \in X$. There is a back edge $(b, p) \in E(D'_{XF})$ because $p \notin X$. Then there is a $\mathcal{P}$-compliant walk, $W = b, p, \ldots, x$, $x \in X$. ◇

Lemma 27 *Let $X \subseteq V(D) \setminus newF$ be a convex set such that $OUT(X, D) = V_{Out}$, $L(V_{Out}, D) \subseteq newF$. If $\mathcal{P}$ is a maximal input path set from $newF$ to X then its T set is convex in D.*

Proof For contradiction, suppose that T is not convex and there is a vertex $w \notin T$, a path P_1 in D from a vertex in T to w, and a path P_2 in D from w to a vertex in T.

The vertex w cannot be in $L(V_{Out}, D)$ because then all vertices on P_1 would be in $L(V_{Out}, D)$, which contradicts $T \cap L(V_{Out}, D) = \emptyset$ (Lemma 25).

However, if $w \notin L(V_{Out}, D)$, then all vertices on P_1 are also in T by Lemma 26, so we have a contradiction, as required. $\diamond$

Lemma 28 *Let $X \subseteq V(D) \setminus newF$ be a convex set such that $OUT(X, D) = V_{Out}$, $L(V_{Out}, D) \subseteq newF$. If $\mathcal{P}$ is a maximal input path set from $newF$ to X and it has the associated set S, then $V(D) \setminus S$ is convex in D.*

Proof

For contradiction, suppose that $V(D) \setminus S$ is not convex and there is a vertex w such that $w \in S$, and w is a direct descendent of a vertex x say, $x \notin S$, and a path P in D from w to a vertex in $V(D) \setminus S$.

There is a $\mathcal{P}$-compliant walk, $W = w_1, \ldots, w, \ldots$ say, from $newF$ to w because $w \in S$. Then $W_1, \ldots, w, x$ is a $\mathcal{P}$-compliant walk in D' from a member of $newF$ to x. Then x is in S, a contradiction. $\diamond$

Lemma 29 *Consider a DAG D, two sets X and $newF$ such that $X \subset V(D)$, $F(D) \subseteq newF \subseteq V(D)$, and $newF \cup X = \emptyset$, and a maximal input path set $\mathcal{P}$ from $newF$ to X, with associated T and S sets. If $L(V_{Out}, D) \subseteq newF$ then a) $OUT(T, D) = V_{Out}$ b) $OUT(V(D) \setminus S, D) = V_{Out}$.*

Proof a) By Lemma 25 we have that $T \cap newF = \emptyset$ and by Lemma 27 we have that T is convex. $X \subseteq T$ because every vertex has a $\mathcal{P}$-compliant path to itself. Then by Lemma 15 $OUT(T, D) = V_{Out}$.

b) Because every vertex in $newF$ has a path to itself $newF \subseteq S$ and because $L(V_{Out}, D) \subseteq newF$ we have $(V(D) \setminus S) \cap L(V_{Out}, D) = \emptyset$. Then by Lemma 15 $OUT(V(D) \setminus S(\mathcal{P}D, newF, X), D) = V_{Out}$. $\diamond$

Lemma 30 *Let $X \subseteq V(D) \setminus newF$ be a convex set such that $OUT(X, D) = V_{Out}$, $L(V_{Out}, D) \subseteq newF$. If $\mathcal{P}$ is a maximal input path set from $newF$ to X with associated T set, then $|IN(T, D)| = |\mathcal{P}|$.*

Proof Any path from $newF$ to $X \subseteq T$ in $D'_{newF,X}$ contains at least one forward edge of the form (i, s) such that $s \in T$ and $i \notin T$. Then either (i, s) is a forward edge and $i \in IN(T, D)$ or (i, s) is an inverse edge and so (s, i) being a forward edge implies that $s \in OUT(T, D)$.

In the latter case, $s \in OUT(T, D) = OUT(X, D)$ is contrary to the construction of the inverse edges. Thus $i \in IN(T, D)$.

Let $\mathcal{P} = \{P_1, \ldots, P_k\}$ such that $k = |\mathcal{P}|$, and let $P_j = \ldots, i_j, s_j, \ldots$, such that $i_j \notin T, s_j \in T, 1 \leq j \leq n$.

Note that for any i_a, i_b such that $1 \leq a \leq b \leq n$, if $i_a = i_b$, then $(i_a, s_a) = (i_b, s_b)$ by Definition 19(3). Thus $a = b$ by Definition 19(4). Then $\{i_1, \ldots, i_k\}$ are distinct. Because they are all in $IN(T, D)$, we have that $|IN(T, D)| \geq |\mathcal{P}|$.

Let I be the set of all i_j. For contradiction, let $i \in IN(T, D) \setminus I$. Because i is an input vertex, it has an edge to a vertex $t \in T$ in D. If there is no forward edge out of i in $E(\mathcal{P})$, then $\{i, t\}$ is a $\mathcal{P}$-compliant walk and $i \in T$, a contradiction.

However, if there is a forward edge out of i in $E(\mathcal{P})$, then i is on a path to a member of X. Let j be next vertex on the path from i to a member of X. The vertex j is not in T because then i would be a member of I. However, j, i, t is a $\mathcal{P}$-compliant walk so $j \in T$ and we have a contradiction. Thus $IN(T) \setminus I = \emptyset$.

Given that $IN(T, X) \setminus I = \emptyset$ and $I \subseteq IN(T, D)$, we have that $|IN(T, D)| = |\mathcal{P}|$ as required. $\diamond$

Lemma 31 *Let $X \subseteq V(D) \setminus newF$ be a convex set such that $OUT(X, D) = V_{Out}$, $L(V_{Out}, D) \subseteq newF$. If $\mathcal{P}$ is a maximal input path set from $newF$ to X with associated S set, then $|IN(V(D) \setminus S, D)| = |\mathcal{P}|$.*

Proof Any path from $newF$ to $X \subseteq V(D) \setminus S$ in $D'_{newF,X}$ contains at least one forward edge of the form (i, s) such that $s \in V(D) \setminus S$ and $i \in S$. Then either (i, s) is a forward edge and $i \in IN(V(D) \setminus S, D)$ or (i, s) is an inverse edge and so (s, i) being a forward edge implies that $s \in OUT(V(D) \setminus S, D)$.

In the latter case, $s \in OUT(V(D) \setminus S, D) = OUT(X, D)$ is contrary to the construction of the inverse edges. Thus $i \in IN(V(D) \setminus S, D)$.

Let $\mathcal{P} = \{P_1, \ldots, P_k\}$ s.t $k = |\mathcal{P}|$, and let $P_j = \ldots, i_j, s_j, \ldots, i_j \in S, s_j \in V(D) \setminus S, 1 \leq j \leq n$.

Note that for any i_a, i_b such that $1 \leq a \leq b \leq n$, if $i_a = i_b$, then $(i_a, s_a) = (i_b, s_b)$ by Definition 19(3). Then $a = b$ by Definition 19(4). Thus $\{i_1, \ldots, i_k\}$ are distinct and they are all in $IN(V(D) \setminus S, D)$, so $|IN(V(D) \setminus S, D)| \geq |\mathcal{P}|$.

Let I be the set of all i_j. For contradiction, let $i \in IN(T, D) \setminus I$. i has an edge to a vertex $t \in V(D) \setminus S$ in D. If there is no forward edge out of i in $E(\mathcal{P})$, then $\{i, t\}$ is a $\mathcal{P}$-compliant walk and $i \in T$, which is a contradiction because $i \in S$.

However, if there is a forward edge out of i in $E(\mathcal{P})$, then i is on a path to a member of X. Let j be next vertex on the path from i to a member of X. The vertex j is not in $V(D) \setminus S$ because then i would be a member of I. However, j, i, t is a $\mathcal{P}$-compliant walk so $j \in T$ and we have a contradiction because $j \in S$. Then $IN(V(D) \setminus S) \setminus I = \emptyset$.

Given that $IN(T, X) \setminus I = \emptyset$ and $I \subseteq IN(T, D)$, we have $|IN(V(D) \setminus S, D)| = |\mathcal{P}|$ as required. $\diamond$

Directing the choice of splitting vertex

We wish to choose a splitting vertex in such a way that there is at least one valid convex vertex set that contains the splitting vertex and one that does not.

Consider the case where $S \cup T \neq V(D)$. $V(D) \setminus S$ and T are valid convex vertex sets so it is simple to choose any vertex $v \in V(D) \setminus (S \cup T)$ as the splitting

vertex.

For all further cases it can be assumed that $S \cup T = V(D)$.

Consider the case where the size of the input path set is equal to $inConstraint$. Then no more paths may be added in further recursive calls and so S and T cannot change. In this situation, the only valid convex vertex set for the given X and $newF$ is T. So T is stored and no recursive calls are made.

For all further cases it can be assumed that $S \cup T = V(D)$ and that $n < inConstraint$.

If $T \neq X$, then let a be the topologically first vertex in $T \setminus X$. $T \setminus \{a\}$ is convex if T is, because $a \in source(T)$ so no extra vertices will be added to maintain convexity. Then a is the only new vertex added to $newF$ so there can be only one more input path. $|\mathcal{P}(D, newF \cup \{a\}, X)| \leq |\mathcal{P}(D, newF, X)| + 1$ so the input constraint will still hold. Then $T(\mathcal{P}(X, newF, D)) \setminus \{a\}$, and $T(\mathcal{P}(X, newF, D))$ are valid convex vertex sets, so a is chosen to be the splitting vertex.

For all further cases it can be assumed that $T = X$, $S \cup T = V(D)$, and that $n < inConstraint$.

We cannot choose any element of T to be the splitting vertex because $T = X$. If there are any more convex vertex sets that contain X, they also contain at least one element of $IN(T, D)$. In this case a maximal input path set $\mathcal{P}_i$ from $newF \cup \{i\}$ to X is calculated for all $i \in IN(T, D)$. If $|\mathcal{P}_i| \leq inConstraint$ for some i, then that value of i is used as the splitting vertex.

If there is no i such that $|\mathcal{O}_i| \leq inConstraint$ then X is the only valid convex vertex set so we store it and return

We now have an algorithm that identifies when there is only one remaining valid convex vertex set for a given X, $newF$ and D, and will choose the splitting vertex in such a way that there is always at least one valid convex vertex set in every branch. The algorithm $\Omega_{splitting}$ incorporates these advances. Note that if $recurseF\Omega_{splitting}()$ makes any recursive calls, then each leaf in the recursive

call tree will represent the storage of a valid convex vertex set.

6.5.2 Time complexity

The *validOutputs*() function will make a maximum of $n^{outConstraint}$ recursive calls when enumerating output sets that have no more than *outConstraint* elements and, by Lemma 32, we also have that $recurseF\Omega_{splitting}()$ will be executed no more than $n^{outConstraint} + 2|\mathcal{S}_{io}(D)|$ times. By Lemma 14 and Lemma 24 the complexities of *validOutputs*() and $recurseF\Omega_{splitting}()$ are $O(|V(D| + |E(D)|)$. Then the overall time complexity of the $\Omega_{splitting}$ algorithm is $O(|V(D)|^{outConstraint} + |\mathcal{S}_{io}(D)| \cdot |E(D)|)$.

Lemma 32 *For a DAG D of size n, which has m valid convex vertex sets under I/O constraints when the input constraint is inConstraint, and the output constraint is outConstraint, $recurseF\Omega_{splitting}()$ will be executed at most $n^{outConstraint} + 2m$ times.*

Proof Consider that the calls to $recurseF\Omega_{splitting}()$ can be viewed as a forest of binary trees. Each tree is either a single vertex, which represents a situation where there were no valid convex vertex sets for a particular output set (there can clearly be at most $n^{outConstraint}$ of these), or they are binary trees, which have the property that each leaf of the tree corresponds to a valid convex vertex set. At least half the vertices in a binary tree are leaf vertices so if there are a total of m leaf vertices in the forest then there can be no more than $2m$ such vertices. Thus there can be no more than $n^{outConstraint} + 2m$ calls to the $recurseF\Omega_{splitting}()$ function. ◇

6.6 Performance of algorithms

This section compares the performance of the Ω family of algorithms with that of the **split** and **exhaustive** algorithms. Appendix A gives details of the ex-

Algorithm 28 $\Omega_{splitting}$: enumerating convex vertex sets under I/O constraints using intelligent selection of splitting vertex

```
Ω_splitting (D)
{
    For all V_Out stored by validOutputs(∅, |V(D)|, D)
    {
        Let newF ← L(V_Out, D)
        X ← V_Out ∪ (dom(V_Out) ∩ domBy(V_Out))
        recurseFΩ_splitting(X, newF, D)
    }
}

recurseFΩ_splitting(X, newF, D)
{
    numberOfPaths ← |P(X, newF, D)|
    if numberOfPaths > inConstraint
        return
    S ← S(P(X, newF, D))
    T ← T(P(X, newF, D)
    if V(D) ≠ S ∪ T
        v ← v_i ∈ V(D) \ (S ∪ T) such that i is maximum.
    else
    {
        if numberOfPaths = inConstraint
            store X
        else
            if X = T
                v ← v_i ∈ S \ newF such that i is maximum.
            else
                if ∃i ∈ IN(X, D) s,t inConstraint ≥ |P(X, newF ∪ {i}, D)|
                    v ← i
    }
    recurseFΩ_splitting(X ∪ {v}i ∪ (domBy(v, D) \ newF), newF, D)
    recurseFΩ_splitting(X, newF ∪ {v}, D)
}
```

perimental setup.

6.6.1 Performance on test cases

Results are shown in Tables 6.2 and 6.3 for the PY04 dataset and Tables 6.6 and 6.7 for the CMS07 dataset.

It is clear that the Ω_{count} algorithm is faster on all test cases than the **split** and **exhaustive** algorithms. The improvement made by the Ω_{count} algorithm varies greatly from case to case, ranging from 30% to over an order of magnitude on test cases in the PY dataset.

The Ω_{paths} algorithm performs well, although it is inferior to **split** on some of the test cases. Its performance is generally better than **exhaustive** and it is clearly a practical algorithm for tackling the problem.

6.6.2 Performance on synthetic DAGs

Figures 6.10 and 6.11 show the relative performance of the five algorithms on the tree and lattice synthetic graphs with an input constraint of 3 and an output constraint of 2. There were 20 test cases for each type of syntectic graph. Figure 6.12 shows the relative performance on tree synthetic graphs from close to the x-axis. It is clear that the algorithms presented in this chapter consistently perform best. The **split** algorithm suffers greatly on tree based input and is inferior to the **exhaustive** algorithm in this situation.

6.6.3 Analysis of algorithmic factors influencing performance

In this section the relative performance of each algorithm is examined in detail and the cases in which performance suffers are isolated.

After close examination of the results it becomes clear that (unlike algorithms such as **split** and **exhaustive**) the internal structure of a test DAG has little effect on the time-per-convex-set results of the Ω family of algorithms.

Figure 6.10 Performance on the synthetic lattice DAG under I/O constraints

Figure 6.11 Performance on the synthetic tree DAG under I/O constraints

Figure 6.12 Performance on the synthetic tree DAG under I/O constraints (Viewed from close to x-axis)

Figure 6.13 The relationship between time-per-convex-set and the ratio of inputs to outputs for Ω_{count}

However, other factors can influence the efficient running of these algorithms. Figure 6.13 shows the relation between time-per-convex-set and the ratio of inputs to outputs for all test cases involving Ω_{count}. There is an obvious correlation between these factors—Ω_{count} is most efficient for I/O constraints such as 4/1 and 7/2, but suffers greatly when give I/O constraints such as 4/3 and 5/4. This inefficiency stems from the fact that Ω_{count} enumerates all outputs sets even if they are unlikely to contain any valid convex vertex sets. In the case where the I/O constraints are 5/4, there are a very small number of valid convex vertex sets that have 4 outputs and only 5 inputs in real examples.

As the I/O ratio decreases the output enumeration phase of the algorithm tends to dominate its running time because all output sets are enumerated. Table 6.8 shows some results of running the Ω family of algorithms against **split** and **exhaustive** in the highly unusual case where there is a tighter input constraint than output constraint. The results in Table 6.8 show that all of the Ω family suffer greatly under these constraints, particularly Ω_{count}. But, it is difficult to think of an architecture that allows more register writes than reads, and it is just as challenging to think of an application that would require such constraints.

6.7 Summary

This chapter has presented four algorithms that use varying degrees of sophistication to enumerate convex vertex sets efficiently under I/O constraints. An algorithm for enumerating convex vertex sets subject to output constraints, *outAlgorithm*, was presented. In addition to its individual usefulness, it formed the basis for the rest of the algorithms in the chapter.

A fast algorithm, Ω_{count}, that used an external input check to prune the search space and achieve polynomial complexity was presented. Experiments

Input and vertices (forbidden)	I/O	Convex sets	Time **split**	Time **exhaustive**	Time Ω_{count}	Time Ω_{paths}	Time $\Omega_{splitting}$
bf 467(134)	2/1	482	0.04	2.78	0.01	0.02	0.01
	4/1	1,920	0.07	15.71	0.01	0.03	0.04
	6/1	7,669	0.11	34.61	0.03	0.16	0.17
	3/2	7,831	0.35	91.51	0.09	0.15	0.15
	5/2	40,714	0.79	352.95	0.32	0.92	0.92
	7/2	161,234	1.70	790.80	0.81	4.38	4.40
	4/3	105,599	2.31	DNF	1.28	2.76	2.81
	6/3	570,197	7.02	DNF	4.08	14.56	14.57
cjpeg 152(34)	2/1	406	0.02	0.10	0.00	0.00	0.00
	4/1	544	0.02	0.10	0.00	0.00	0.00
	6/1	550	0.01	0.11	0.00	0.00	0.00
	3/2	41,363	0.61	13.86	0.08	0.30	0.28
	5/2	113,611	0.82	19.30	0.14	0.59	1.02
	7/2	140,335	0.94	20.15	0.18	0.89	1.53
	4/3	2,201,568	20.50	DNF	7.24	22.22	18.78
rijndael 1,237(391)	2/1	1,241	2.79	51.43	0.04	0.04	0.05
	4/1	4,787	3.51	253.30	0.05	0.16	0.17
	6/1	15,236	4.09	DNF	0.10	0.61	0.59
	3/2	75,241	83.96	DNF	3.15	4.36	3.98
	5/2	648,748	201.41	DNF	7.95	26.54	23.88
sha 1,811(351)	2/1	1,546	3.76	DNF	0.11	0.10	0.13
	4/1	4,372	4.23	DNF	0.13	0.28	0.24
	6/1	10,152	5.30	DNF	0.17	0.53	0.49
	3/2	78,132	85.14	DNF	5.06	7.15	6.75
	5/2	293,259	164.97	DNF	6.28	15.82	15.37
md5 1,170(353)	4/1	2,304	1.66	DNF	0.04	0.06	0.08
	6/1	3,546	1.70	DNF	0.04	0.12	0.12
	3/2	54,476	51.45	DNF	2.41	3.89	3.24

Table 6.2 Results for timings on PY04 dataset

Input and vertices (forbidden)	I/O	Convex sets	Calls **split**	Calls **exhaustive**	Calls Ω_{count}	Calls Ω_{paths}	Calls $\Omega_{splitting}$
bf 467(134)	2/1	482	24,009	796,775	4,741	1,279	631
	4/1	1,920	34,084	4,467,923	21,697	5,155	3,507
	6/1	7,669	55,374	9816,778	43,851	23,325	15,005
	3/2	7,831	176,631	25,169,197	103,202	23,588	12,302
	5/2	40,714	383,570	101,091,122	437,948	109,530	75,376
	7/2	161,234	814,515	216,584,931	997,284	455,042	316,386
	4/3	105,599	1,122,520	DNF	1,437,265	330,171	189,037
	6/3	570,197	3,342,391	DNF	5,188,167	1,613,829	1,085,505
cjpeg 152(34)	2/1	406	21,907	61,832	886	854	694
	4/1	544	22,003	70,216	982	982	970
	6/1	550	22,003	70,216	982	982	982
	3/2	41,363	677,813	9,880,064	178,877	158,617	76,907
	5/2	113611	875,155	13,460,590	259,413	254,191	220,999
	7/2	140,335	896,688	13,721,462	284,323	283,435	274,377
	4/3	2,201,568	18,454,621	DNF	15,984,520	11,367,292	4,238,390
rijndael 1,237(391)	2/1	1,241	697,778	5,473,096	10,818	3,358	1,636
	4/1	4,787	786,732	27,471,175	33,224	13,598	8,728
	6/1	15,236	878,083	DNF	74,310	44,712	29,626
	3/2	75,241	11,575,641	DNF	1,769,119	309,567	145,477
	5/2	64,8748	31,777,459	DNF	6,971,175	2,098,957	1,207,733
sha 1,811(351)	2/1	1,546	300,752	DNF	10,996	3,710	1,632
	4/1	4,372	345,994	DNF	25,052	1,0670	7,284
	6/1	10,152	432,350	DNF	45,334	26,918	18,844
	3/2	78,129	6,450,724	DNF	742,919	227,341	117,159
	5/2	293,259	12,652,418	DNF	1,809,397	743,945	494,507
md5 1,170(353)	3/2	54,476	6,109,809	DNF	377,805	169,387	102,389
	5/2	223,820	19,633,660	DNF	679,985	489,809	360,717
	7/2	377,943	20,557,975	DNF	927,173	772,545	668,729

Table 6.3 Results for recursive calls on PY04 dataset

Input and vertices (forbidden)	I/0	Convex sets	Time **split**	Time **exhaustive**	Time Ω_{count}	Time Ω_{paths}	Time $\Omega_{splitting}$
cjpeg2+ 183(31)	3/2	53,175	1.40	26.71	0.09	0.38	0.34
	5/2	116,784	1.63	27.98	0.14	0.53	0.83
	7/2	119,477	1.71	27.10	0.14	0.52	0.87
	4/3	3,210,234	42.30	DNF	7.03	35.84	25.71
cjpeg5+ 191(39)	4/1	514	0.04	0.18	0.00	0.00	0.00
	6/1	514	0.03	0.18	0.00	0.00	0.01
	3/2	50,938	1.45	27.69	0.10	0.38	0.33
	5/2	113,874	1.69	28.02	0.13	0.52	0.82
	7/2	116,576	1.70	28.82	0.14	0.54	0.86
	4/3	2,971,403	42.90	DNF	6.95	34.16	24.46
rijndael1+ 1,299(86)	2/1	2,887	4.51	DNF	0.05	0.10	0.09
	6/1	46,436	6.72	DNF	0.22	1.38	1.40
susan1+ 139(35)	2/1	165	0.01	0.19	0.00	0.01	0.00
	4/1	386	0.01	1.29	0.00	0.01	0.00
	3/2	7,046	0.18	9.03	0.01	0.06	0.05
	5/2	17,871	0.27	53.95	0.05	0.20	0.29
	7/2	64,673	0.41	271.60	0.22	0.83	1.31
	4/3	192,628	2.44	212.22	0.44	1.19	1.33
	6/3	476,504	4.10	1034.98	1.23	4.13	6.14
	8/3	1,642,247	7.79	DNF	5.09	16.73	28.3
susan2+ 234(31)	2/1	333	0.06	2.91	0.00	0.01	0.01
	4/1	968	0.08	39.48	0.01	0.04	0.07
	6/1	5,894	0.14	375.16	0.06	0.21	0.42
	3/2	26,303	2.36	DNF	0.17	0.43	0.52
	5/2	96,784	4.94	DNF	1.17	2.53	4.34
	7/2	732,530	11.05	DNF	7.81	14.66	29.32
	4/3	1,380,507	67.20	DNF	10.78	16.47	19.21
gsm2+ 336(57)	2/1	1,143	0.19	2.96	0.01	0.24	0.16
	4/1	8,792	0.36	24.59	0.06	1.76	1.42
	6/1	64,829	2.10	184.35	0.55	11.98	10.29
	3/2	77,569	10.32	200.57	0.65	10.14	7.62
	5/2	658,818	30.83	DNF	4.41	82.09	68.91
gsm3+ 475(72)	2/1	908	0.70	33.54	0.01	0.02	0.01
	4/1	6,365	0.86	575.49	0.06	0.13	0.13
	6/1	68,324	1.88	DNF	0.47	1.37	1.38
	3/2	215,817	78.59	DNF	1.97	4.22	3.29
gsm4+ 117(35)	6/1	196	0.01	0.03	0.00	0.00	0.00
	3/2	7,705	0.13	0.76	0.01	0.03	0.03
	5/2	9,248	0.13	0.79	0.01	0.03	0.04
	7/2	10,455	0.14	0.81	0.01	0.04	0.06
	4/3	275,531	2.66	18.92	0.30	0.96	1.31
	6/3	324,857	2.77	18.99	0.34	1.18	1.64
	8/3	357,348	2.82	19.65	0.36	1.37	2.02
gsm5+ 429(41)	2/1	2,021	0.30	7.58	0.00	0.02	0.02
	4/1	7,923	0.32	12.52	0.02	0.07	0.09
	6/1	7,923	0.30	12.98	0.02	0.08	0.11

Input and vertices (forbidden)	I/0	Convex sets	Calls **split**	Calls **exhaustive**	Calls Ω_{count}	Calls Ω_{paths}	Calls $\Omega_{splitting}$
cjpeg2+ 183(31)	3/2	53,169	1,763,875	12,673,782	148,483	184,222	97,860
	5/2	116,784	1,997,568	13,040,760	226,846	229,500	224,114
	7/2	119,477	1,999,888	13,040,760	229,500	229,500	229,500
	4/3	3,210,234	44,299,122	DNF	12,928,052	15,134,145	6,157,303
cjpeg5+ 191(39)	4/1	514	47,986	83,764	876	876	876
	6/1	514	47,986	83,764	876	876	876
	3/2	50,938	1,799,277	13,305,021	143,321	178,612	93,424
	5/2	113,874	2,042,102	13,720,107	220,992	223,646	218,260
	7/2	116,567	2,044,422	13,720,107	223,646	223,646	223,646
	4/3	2,971,403	44,500,213	DNF	12,169,650	14,208,292	5,683,704
rijndael1+ 1,299(86)	4/1	9,557	1,021,462	DNF	46,292	24,541	17,901
	6/1	46,436	1,190,680	DNF	139,196	114,733	91,659
susan1+ 139(35)	2/1	165	16,994	124,304	555	290	226
	4/1	386	18,347	856,933	2,208	892	668
	3/2	7,046	254,135	5,924,234	25,865	13,783	11,457
	5/2	17,871	328,750	34,678,943	98,204	40,165	32,585
	7/2	64,673	445,761	166,259,927	390,607	152,065	126,151
	4/3	192,628	2,855,889	134,689,437	708,890	390,431	345,401
	6/3	476,504	4,306,382	643,018,370	2,446,025	1,030,261	897,609
	8/3	1,642,247	6,924,579	DNF	9,240,739	3,636,485	3,227,717
susan2+ 234(31)	2/1	333	51,576	1,203,143	2,086	573	463
	4/1	968	61,022	16,083,406	14,446	2,283	1,733
	6/1	5,894	87,365	151,318,756	88,949	14,187	11,585
	3/2	26,303	1,666,852	DNF	245,182	50,306	42,854
	5/2	96,784	2,730,585	DNF	1,948,384	221,910	182,188
	7/2	732,530	5,473,593	DNF	12,115,784	1,628,700	1,453,606
	4/3	1,380,507	38,976,354	DNF	16,435,288	2,725,819	2,467,005
gsm2+ 336(57)	2/1	1,143	94,401	996,324	7,526	2,957	2,007
	4/1	8,792	155,327	8,303,029	54,853	23,347	17,305
	6/1	64,829	744,073	61,285,858	405,121	173,269	129,379
	3/2	77,569	4201,325	64,460,499	567,671	208,008	140,888
	5/2	658,818	9,917,960	DNF	4,185,076	1,752,838	1,299,904
gsm3+ 475(72)	2/1	908	405,293	6,482,306	4,278	2,035	1,413
	4/1	6,365	484,260	109,583,305	58,610	22,593	12,327
	6/1	68,324	918,285	DNF	485,600	226,243	136,245
	3/2	215,817	39,178,348	DNF	2,071,958	663,947	382,023
gsm4+ 117(35)	6/1	196	13,054	28,321	331	330	310
	3/2	7,705	252,571	745,155	14,108	13,524	12,856
	5/2	9,248	255,480	780,160	16,863	16,412	15,872
	7/2	10,455	257,343	801,833	18,990	1,8646	18,248
	4/3	275,531	3,863,747	17,078,742	537,552	516,631	500,037
	6/3	324,857	3,943,215	17,820,664	621,799	606,953	596,661
	8/3	357,348	3,992,122	18,262,773	676,168	666,275	660,659
gsm5+ 429(41)	2/1	2,021	242,704	2,013,750	7,576	7,532	3,654
	4/1	7,923	250,665	3,282,024	15,458	15,458	15,458
	6/1	7,923	250,665	3,282,024	15,458	15,458	15,458

Input and vertices (forbidden)	I/O	Convex sets	Time **split**	Time **exhaustive**	Time Ω_{count}	Time Ω_{paths}	Time $\Omega_{splitting}$
aes1 35(9)	3/2	483	0.00	0.01	0.00	0.00	0.00
	4/3	1,895	0.00	0.02	0.01	0.01	0.00
	6/3	5,981	0.01	0.04	0.01	0.02	0.04
	8/3	15,477	0.02	0.07	0.01	0.05	0.12
	5/4	5,141	0.01	0.04	0.01	0.02	0.03
	7/4	13,089	0.03	0.07	0.02	0.04	0.09
	9/4	19,425	0.03	0.07	0.02	0.07	0.14
	6/5	9,899	0.01	0.06	0.01	0.03	0.05
	8/5	21,705	0.04	0.07	0.02	0.07	0.15
	10/5	21,705	0.04	DNF	0.02	0.07	0.15
bf_enc0 137(42)	2/1	170	0.00	0.02	0.00	0.00	0.00
	4/1	170	0.01	0.02	0.00	0.00	0.00
	6/1	170	0.00	0.02	0.00	0.00	0.00
	3/2	9,728	0.18	0.81	0.01	0.03	0.04
	5/2	9,728	0.18	0.78	0.01	0.03	0.04
	7/2	9,728	0.19	0.79	0.02	0.03	0.04
	4/3	325,892	3.52	20.21	0.39	1.17	1.44
	6/3	325,892	3.46	20.03	0.39	1.17	1.46
	8/3	325,892	3.45	19.66	0.39	1.18	1.46
aesSort 885(102)	2/1	2,817	0.86	102.17	0.04	0.06	0.05
	4/1	30,729	1.30	299.26	0.13	0.44	0.53
	6/1	56,709	1.46	312.66	0.16	0.84	1.06
	3/2	696,889	37.20	DNF	8.66	10.72	10.40
sha0 1,049(18)	2/1	1,776	0.13	57.11	0.04	0.07	0.06
	4/1	4,443	0.15	95.49	0.05	0.11	0.10
	6/1	8,644	0.17	93.44	0.06	0.16	0.16
	3/2	28,255	1.08	DNF	0.43	0.82	0.69
	5/2	94,373	1.73	DNF	0.66	1.96	1.79
	7/2	203,711	2.57	DNF	0.98	3.53	3.73
	4/3	246,533	4.59	DNF	2.93	6.82	5.37
	6/3	881,147	11.17	DNF	5.37	19.99	16.98

Table 6.6 Results for timings on CMS07 dataset

Input and vertices (forbidden)	I/O	Convex sets	Calls **split**	Calls **exhaustive**	Calls Ω_{count}	Calls Ω_{paths}	Calls $\Omega_{splitting}$
aes1 35(9)	3/2	483	1,864	25,462	2,776	1,036	832
	4/3	1,895	4,735	66,064	9,597	4,067	3,425
	6/3	5,981	10,845	131,396	22,653	12,725	11,597
	8/3	15,477	24,719	173,780	30,589	30,589	30,589
	5/4	5,141	9,936	123,226	20,490	10,670	9,602
	7/4	13,089	21,452	195,478	38,170	26,554	25,498
	9/4	19,425	30,798	209,734	38,170	38,170	38,170
	6/5	9,899	16,875	177,780	33,561	20,303	18,845
	8/5	21,705	34,071	227,904	42,457	42,457	42,457
	10/5	21,705	34,071	DNF	42,457	42,457	42,457
bf_enc0 137(42)	2/1	170	14,487	19,848	245	245	245
	4/1	170	14,487	19,848	245	245	245
	6/1	170	14,487	19,848	245	245	245
	3/2	9,728	326,718	747,585	15,980	15,980	15,980
	5/2	9,728	326,718	747,585	15,980	15,980	15,980
	7/2	9,728	326,718	747,585	15,980	15,980	15,980
	4/3	325,892	53,376,20	18,400,185	575,631	575,631	575,631
	6/3	325,892	5,337,620	18,400,185	575,631	575,631	575,631
	8/3	325,892	53,376,20	18,400,185	575,631	575,631	575,631
aesSort 885(102)	2/1	2,817	259,820	15,400,057	35,761	5,139	4,851
	4/1	30,729	385,700	47,901,517	109,539	61,057	60,675
	6/1	56,709	438,456	48,308,541	112,635	112,635	112,635
	3/2	696,889	10,288,513	DNF	11,499,177	1,566,061	1,332,655
sha0 1049(18)	2/1	1,776	43,785	5,771,899	7,667	3,847	2,521
	4/1	4,443	52,815	9,330,259	14,773	10,759	7,855
	6/1	8,644	60,122	9,455,447	18,309	17,715	16,257
	3/2	28,255	290,713	DNF	160,972	80,484	46,736
	5/2	94,373	506,763	DNF	345,424	245,470	175,214
	7/2	203,711	739,287	DNF	50,2984	455,982	393,734
	4/3	246,533	1,430,343	DNF	1,442,694	761,928	438,068
	6/3	881,147	3,372,142	DNF	3,331,460	2,395,740	1,688,330

Table 6.7 Results for recursive calls on CMS07 dataset

Input and vertices (forbidden)	I/O	Convex sets	Time **split**	Time **exhaustive**	Time Ω_{count}	Time Ω_{paths}	Time $\Omega_{splitting}$
bf_enc0 137(42)	1/3	832	0.01	0.94	0.21	0.25	0.26
	1/4	1,216	0.02	2.38	3.30	4.24	4.27
	2/5	235,004	1.16	DNF	49.48	56.63	58.46
aesSort 885(102)	1/2	4,605	1.48	DNF	1.43	1.23	1.15
	1/3	12,925	2.40	DNF	50.31	43.78	43.46
sha0 1,049(18)	1/2	1,545	0.20	484.24	0.31	0.40	0.38
	1/3	2,211	0.16	1438.44	1.74	2.18	2.13

Table 6.8 Results for unusual I/O constraints on CMS07 dataset

showed that Ω_{count} takes considerably less time to enumerate convex vertex sets under I/O constraints.

To solve the problem of the Ω_{count} algorithm chasing down 'blind alleys', the Ω_{paths} algorithm, based on the concept of flows in networks, was presented, which traded greater computational overhead for a more efficient pruning criterion.

This flow based approach was further refined to find $\Omega_{splitting}$, which used S and T sets to precisely direct the search. The $\Omega_{splitting}$ algorithm has the property that each leaf in its call trees will be a valid convex vertex set.

The experiments performed for this chapter indicated that Ω_{count} was the fastest performing algorithm in the general case, although both the Ω_{paths}, and $\Omega_{splitting}$ algorithms achieved significantly lower numbers of recursive calls.

Chapter 7

Detailed analysis of the union algorithm

The **union** algorithm [YM04] was published in 2004 as an alternative approach to the method introduced by [API03]. It is notable for being the first method to enumerate only connected convex vertex sets.

This chapter examines the model, completeness, and efficiency of the **union** algorithm [YM04, YM08, YM07]. An overview of the algorithm is given, followed by a discussion of the correctness of the algorithm as originally published. The chapter concludes by examining the experimental setup used in [YM04, YM08, YM07]. For this part of our work we must give credit to one of the authors of [YM04] who was kind enough to send us the original input DAGs used in [YM04]. Without his co-operation, much of this work would not have been possible.

7.1 Overview of the algorithm

The **union** algorithm uses the notion of upward and downward cones (see Section 4.6) to search for all the connected convex vertex sets in a DAG, and it focuses heavily on satisfying I/O conditions. The **union** algorithm uses these cheaply computable cones by combining them to produced all connected convex vertex sets in a DAG. It is claimed to be exhaustive on connected convex sets and the published results show large improvements over the **exhaustive**

algorithm.

The approach has been updated by [YM08, YM07] and extended to also enumerate disconnected convex vertex sets by combining existing sets. This chapter examines only the enumeration of connected convex vertex sets.

Although it has been cited in many publications, to our knowledge the **union** algorithm has only been implemented by one other research group for comparison purposes. In [CMS07] the **union** algorithm was tested against the **exhaustive** and **split** algorithms [PAI06, CMS07]. In that publication it was shown that, contrary to the results in [YM04], the **union** algorithm did not perform as well on commercially representative benchmarks as **exhaustive**.

7.1.1 Operation of the union algorithm

The **union** algorithm proceeds in two phases. In the first phase, all convex upward and downward cones of vertices in D are calculated. In the second phase these cones are combined together to form the set $\mathcal{S}_c(D)$. The second phase is the most complex and will be examined in detail.

Proofs given in [YM07] show that every convex connected vertex set can be formed from one or more cones present in D.

The connected convex vertex sets are enumerated by the recursive function $unionRecursive()$, which is shown in detail in Algorithm 29. Algorithm 30 shows the **union** algorithm as it was first presented. The original formation is given to ensure that no detail is lost in translation.

The $unionRecursive()$ function is passed a connected convex vertex set, A, and a set $\mathcal{A} \subseteq \mathcal{S}_c(D_A)$ that returns the set of connected convex vertex sets that can be formed by the union of an element of $\mathcal{A}$ and an element of $\mathcal{S}(D_{V(D)-A})$.

If a is the topologically first vertex in D, then $unionRecursive(\{a\}, \{\{a\}\}, D, down)$ will return the set of connected convex sets in D that contain a. By deleting a and repeating the process all connected convex vertex sets in D can be found.

7.1.2 Functions used by the union algorithm

For a vertex s in the DAG D, the **union** algorithm defines the *maximum upward cone* of s as the set formed from the vertices that are present in at least one upward cone of s. Similarly, the *maximum downward cone* is the set formed from the vertices that are present in at least one downward cone of s. Recall from Chapter 2 that this is equivalent to the definitions of $dom(s, D)$ and $domBy(s, D)$, however in this chapter we shall use the functions $MAX\_UP(s, D)$ and $MAX\_DOWN(s, D)$.

The function $upCones(a, D)$, returns the set of upward cones rooted at the vertex a in D. Likewise the function $downCones(a, D)$ returns the set of downward cones rooted at the vertex a in D.

7.1.3 The operation of $unionRecursive()$ function

The $unionRecursive()$ function operates in either upwards or downwards mode, its functionality in downwards mode is described here, the upwards mode is analogous.

Firstly, the recursive function finds all *extension points* of the set A. A downwards extension point e of a set A is a vertex in A that has an edge to a vertex $f \notin A$.

Given a set of extension points the $unionRecursive()$ recursive function forms the set $\mathcal{B}$, which contains all combinations of such extension points.

For each combination of extension points B_i, the $union()$ recursive function forms the set $\mathcal{A}_i$ from all the convex vertex sets in $\mathcal{A}$ that include exactly those extension points.

If there are k combinations of extension points then the sets $\mathcal{A}_1, \ldots, \mathcal{A}_k$ form a partition of $\mathcal{A}$.

The $union()$ recursive function then creates the set A_i^+, such that $A_i^+ = A \cup dom(B_i, D)$, and the set $\mathcal{D}_i$, that contains all the connected convex sets that

Sets
{8}
{8,5}
{8,5,7}
{8,5,7,4}
{8,5,7,4,6}
{8,5,7,4,6,2}
{8,5,7,4,6,2,3}
{8,5,7,4,2}
{8,5,7,4,2,3}

Table 7.1 Connected convex vertex sets that contain vertex 8, obtained from Example 22

can be formed from a union of an element of $\mathcal{A}_i$ and an element of $\mathcal{S}(A_i^+ \setminus \mathcal{A}_i$. The $union()$ function then makes the recursive call $union(A_i^+, \mathcal{D}_i, D, !direction)$ and repeats the process for each combination of extension points.

I/O constraints

Whenever a set $\mathcal{D}_i$ is created, each set added to it must be first checked for convexity and I/O constraints, if any check is failed, then the set is pruned. [YM04] provides a proof that this pruning does not prevent the **union** algorithm for enumerating all valid connected convex sets.

7.1.4 Example

The following section walks through the operation of the **union** algorithm on Example 22. For conciseness we shown only the first iteration of the algorithm: the process of finding all connected convex vertex sets that include the vertex 8. For simplicity we also ignore I/O constraints. The connected convex vertex sets that include the vertex 8 are shown in Table 7.1.

Walkthrough

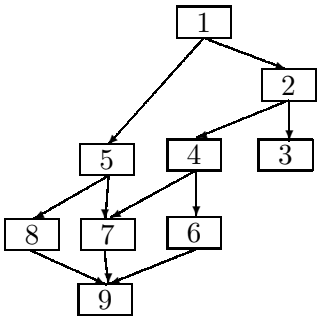
We enter $unionRecursive(\{5, 8\}, \{\{8\}, \{5, 8\}\}, D)$.

Algorithm 29 *union*(D): enumeating connected convex vertex sets by combining cones

```
union(D)
{
  ∀a ∈ V(D)(in topological order)
  {
    store unionRecurisve({a}, {{a}}, D, down)
    delete a from D
  }
}

unionRecursive(region, currentSets, D, direction)
{
  extensionPoints ← {a|a ∈ region, ((v, a) ∈ E(D)||(a, v) ∈ E(D)), v ∉ region}
  Let B be the set of all combinations of extensionPoints
  tempSets ← currentSets
  ∀B ∈ B
  {
    A ← {A|A ∈ currentSets, A ∩ extensionPoints = B}
    Let B ← {b_1 ... b_i}
    if direction = down
    {
      Let D ← (A × downCones(b_1) × ... × downCones(b_i)) \ currentSets
      Let E ← region ∪ maxDownCone(b_1) ∪ ... ∪ maxDownCone(b_i)
    }
    else
    {
      Let D ← (A × upCones(b_1) × ... × upCones(b_i)) \ currentSets
      Let E ← region ∪ maxUpCone(b_1) ∪ ... ∪ maxUpCone(b_i)
    }
    tempSets ← tempsets ∪ unionRecursive(E, D, D, !direction)
  }
  return tempsets
}
```

Algorithm 30 *algorithm*(): the **union** algorithm as it was originally presented in [YM04]

```
algorithm()
begin
    for all vertices v of R in reverse topological order do
        PATTERNS(v) := UC_SET(v);
        ext:= OUT(MAX_UC(v,R));
        if ext ≠ ∅ then
            PATTERNS(v) := UNION(PATTERNS(v),ext,down);
        remove v from R;
    end
end

UNION(core,ext,direction)
begin
    new_core:=core;
    Let ext = {v_1, ..., v_k};
    for i = 1 to k do
        for j = 1 to ( k choose i ) do
            Let V = {v_{x_1}, ..., v_{x_i}} be the j^th combination of extension points;
            P := {p|p ∈ core ⋀ p ⊆ ⋀ p ∩ (ext − V) = ∅};
            if direction = down then
                tmp := DC_SET(v_{x_1}) × ... × DC_SET(v_{x_i}) × P;
            else
                tmp := UC_SET(v_{x_1}) × ... × UC_SET(v_{x_i}) × P;
            end
            temp_core := ∅
            for each t ∈ tmp do
                Let t = {pat_1, ..., pat_{i+1}};
                pat := pat_1 ∪ ... ∪ pat_{i+1};
                if (direction = down) ⋀ CONVEX(pat) ⋀ OUT_CHECK(pat) then
                    tmp_core := temp_core ∪ {pat};
                if (direction = up) ⋀ CONVEX(pat) ⋀ IN_CHECK(pat) then
                    tmp_core := temp_core ∪ {pat};
            end
            if direction = down then
                tmp_ext := ⋃_{v_{x_i}} IN(MAX_UC(v_{x_i}, R)
            else
                tmp_ext := ⋃_{v_{x_i}} OUT(MAX_DC(v_{x_i}, R)
            end
            tmp_ext := REMOVE_EXT(tmp_ext ∩ {vertices present in new_core});
            if tmp_ext ≠ ∅then
                new_core := new_core ∪ UNION(tmp_core, tmp_ext, !direction);
            else
                new_core := new_core ∪ tmp_core;
            end
        end
    end
    new_core := {p|p ∈ new_core ⋀ IN_CHECK(p) ⋀ OUT_CHECK(p)};
    return new_core;
end
```



Example 22: Showing the functionality of the **union** algorithm

b1 The direction is downwards and the only extension point is 5, so there is only one combination.

b2 Let $B_1 = \{5\}$, and $A_1 = \{\{8, 5\}\}$. Then $A_1^+ = \{8, 5, 7\}$ and $\mathcal{D}_1 = \{8, 5, 7\}$.

b3 We make the recursive call $unionRecursive(\{8, 5, 7\}, \{\{8, 5, 7\}\}, D)$.

c1 The direction is upwards and the only extension point is 7 so there is again only one combination.

c2 Let $B_1 = \{7\}$, and $A_1 = \{\{8, 5, 7\}\}$. Then $A_1^+ = \{8, 5, 7, 4, 2\}$ and $\mathcal{D}_1 = \{\{8, 5, 7, 4\}, \{8, 5, 7, 4, 2\}\}$.

c3 We make the recursive call $unionRecursive(\{8, 5, 7, 4, 2\}, \{\{8, 5, 7, 4\}, \{8, 5, 7, 4, 2\}\}, D)$

d1 The direction is downwards are there are two extension points $\{2, 4\}$, we consider first the extension point 2.

d2 Let $B_1 = \{2\}$, and $A_1 = \emptyset$. $\mathcal{A}_1$ is empty, so there is no recursion.

d3 Let $B_2 = \{4\}$, and $A_2 = \{\{8, 5, 7, 4\}\}$. Then $A_2^+ = \{8, 7, 5, 4, 2, 6\}$ and $\mathcal{D}_2 = \{\{8, 5, 7, 4, 6\}\}$.

d4 We make the recursive call $unionRecursive(\{8, 7, 5, 4, 2, 6\}, \{\{8, 5, 7, 4, 6\}\}, D)$.

e1 There are no new extension points so terminate and return $\{\{8, 5, 7, 4, 6\}\}$.

d5 *tempSets* becomes {{8, 5, 7, 4}, {8, 5, 7, 4, 2}, {8, 5, 7, 4, 6}}.

d6 Let $B_3 = \{2, 4\}$, and $A_3 = \{\{8, 5, 7, 4, 2\}\}$. Then $A_3^+ = \{8, 5, 7, 4, 2, 6, 3\}$ and $\mathcal{D}_3 = \{\{8, 5, 7, 4, 2, 6\}, \{8, 5, 7, 4, 2, 3\}, \{8, 5, 7, 4, 2, 6, 3\}\}$.

d7 We make the recursive call *unionRecursive*({8, 5, 7, 4, 2, 6, 3}, {{8, 5, 7, 4, 2, 6}, {8, 5, 7, 4, 2, 3}, {8, 5, 7, 4, 2, 6, 3}}, D).

f1 There are no new extension points so terminate and return {{8, 5, 7, 4, 2, 6}, {8, 5, 7, 4, 2, 3}, {8, 5, 7, 4, 2, 6, 3}}.

d5 *tempSets* becomes {{8, 5, 7, 4}, {8, 5, 7, 4, 2}, {8, 5, 7, 4, 6}, {8, 5, 7, 4, 2, 6}, {8, 5, 7, 4, 2, 3}, {8, 5, 7, 4, 2, 6, 3}}

d6 Terminate and return {{8, 5, 7, 4}, {8, 5, 7, 4, 2}, {8, 5, 7, 4, 6}, {8, 5, 7, 4, 2, 6}, {8, 5, 7, 4, 2, 3}, {8, 5, 7, 4, 2, 6, 3}}.

c4 Terminate and return {{8, 5, 7}, {8, 5, 7, 4}, {8, 5, 7, 4, 2}, {8, 5, 7, 4, 6}, {8, 5, 7, 4, 2, 6}, {8, 5, 7, 4, 2, 3}, {8, 5, 7, 4, 2, 6, 3}}.

b4 Terminate and return {{8}, {8, 5}, {8, 5, 7}, {8, 5, 7, 4}, {8, 5, 7, 4, 2}, {8, 5, 7, 4, 6}, {8, 5, 7, 4, 2, 6}, {8, 5, 7, 4, 2, 3}, {8, 5, 7, 4, 2, 6, 3}}.

When all instances of the recursive function *unionRecursive*() have terminated, we have the sets {8}, {8, 5}, {8, 5, 7}, {{8, 5, 7, 4}, {8, 5, 7, 4, 2}, {8, 5, 7, 4, 6}, {8, 5, 7, 4, 2, 6}, {8, 5, 7, 4, 2, 3}, {8, 5, 7, 4, 2, 6, 3}}, which are all of the connected convex sets involving vertex 8.

7.2 Issues with the union algorithm

This section presents a number of issues with the **union** algorithm, these include problems with the correctness, the model used for I/O constraints, and the experimental setup.

7.2.1 Typographical

A minor issue, included here for completeness, is that the **union** algorithm in Algorithm 30 (taken from [YM04]) appears to contain a typographical error at:

if direction = down **then**

$tmp_extit := \bigcup_{v_{x_i}} IN(MAX\_UC(v_{x_i}, R)$

else

$tmp_extit := \bigcup_{v_{x_i}} OUT(MAX\_DC(v_{x_i}, R)$

end

It is the understanding of the author that this should be:

if direction = down **then**

$tmp_extit := \bigcup_{v_{x_i}} OUT(MAX\_UC(v_{x_i}, R)$

else

$tmp_extit := \bigcup_{v_{x_i}} IN(MAX\_DC(v_{x_i}, R)$

end

because, by definition, upward cones have no inward edges and downward cones have no outward edges. This error has been corrected in [YM08].

7.2.2 Correctness

The number of recursive calls made by the **union** algorithm can grow quickly because the recursive function of the **union** algorithm must make a recursive call for each possible combination of extension points. Moreover, it can result in the same set being created repeatedly. For the **union** algorithm to be competitive over reasonably sized examples, the number of duplicated sets that are constructed must be reduced.

The **union** algorithm attempts to avoid this computational explosion by eliminating extension points that will not result in the identification of any new connected convex vertex sets. The published algorithm defines a function REMOVE_EXT() that, when given a set of extension points, will eliminate extension points that match any of the following criteria.

- ◇ If at least one of the outgoing edges of an extension point e does not lead to a vertex with a valid set of cones (i.e a vertex in the DAG, rather than a forbidden vertex), then e can be eliminated.
- ◇ Given two downward extension points u and v, if $MAX-DOWN(u, D) \subseteq MAX - DOWN(v, D)$, then u can be eliminated. The reverse is true for upward extension points.
- ◇ If an extension point has already been considered by a recursive ancestor, then it can be eliminated.

If extension points are eliminated, then the number of combinations of extension points is greatly reduced. This modification allows a large speedup of the **union** algorithm. However, there is a weakness with the REMOVE_EXT function that causes some valid connected convex sets to be missed by the enumeration process.

The authors of [YM04] provided a proof that their algorithm generates all connected convex vertex sets by proving the algorithm in the trivial case, and then proving that pruning convex vertex sets based on convexity, input and output constraints is safe.

However, the proof does not show that the actions of the REMOVE_EXT function are safe, possibly because it is the result of optimising work performed after the algorithm was proved correct. The author believes that the algorithm is correct if this function is omitted. We now go on to discuss the correctness of this function by means of an example.

REMOVE_EXT example

Consider Example 22, which was used to shown the operation of the **union** algorithm in Section 7.1.4. This section performs the walkthrough again, under the same conditions, but this time eliminating the extension points using a fully operational REMOVE_EXT function.

Walkthrough with REMOVE_EXT

We call $unionRecursive(\{5, 8\}, \{\{8\}, \{5, 8\}\}, D)$.

b1 The direction is downwards and the only extension point is 5, so there is only one combination.

b2 Let $B_1 = \{5\}$, and $A_1 = \{\{8, 5\}\}$. Then $A_1^+ = \{8, 5, 7\}$ and $\mathcal{D}_1 = \{8, 5, 7\}$.

b3 We make the recursive call $unionRecursive(\{8, 5, 7\}, \{\{8, 5, 7\}\}, D)$.

c1 The direction is upwards and the only extension point is 7 so there is again only one combination.

c2 Let $B_1 = \{7\}$, and $A_1 = \{\{8, 5, 7\}\}$. Then $A_1^+ = \{8, 5, 7, 4, 2\}$ and $\mathcal{D}_1 = \{\{8, 5, 7, 4\}, \{8, 5, 7, 4, 2\}\}$.

c3 We make the recursive call $unionRecursive(\{8, 5, 7, 4, 2\}, \{\{8, 5, 7, 4\}, \{8, 5, 7, 4, 2\}\}, D)$

d1 The direction is downwards are there are two extension points $\{2, 4\}$, The REMOVE_EXT eliminates extension point 4 by criteria 2.

d2 Let $B_1 = \{2\}$, and $A_1 = \{\{8, 5, 7, 4, 2\}\}$. Then $A_1^+ = \{8, 5, 7, 4, 2, 6, 3\}$ and $\mathcal{D}_1 = \{\{8, 5, 7, 4, 2, 6\}, \{8, 5, 7, 4, 2, 3\}, \{8, 5, 7, 4, 2, 6, 3\}\}$.

d7 We make the recursive call $unionRecursive(\{8, 5, 7, 4, 2, 6, 3\}, \{\{8, 5, 7, 4, 2, 6\}, \{8, 5, 7, 4, 2, 3\}, \{8, 5, 7, 4, 2, 6, 3\}\}, D)$.

f1 There are no new extension points so terminate and return $\{\{8, 5, 7, 4, 2, 6\}, \{8, 5, 7, 4, 2, 3\}, \{8, 5, 7, 4, 2, 6, 3\}\}$.

d5 *tempSets* becomes $\{\{8, 5, 7, 4\}, \{8, 5, 7, 4, 2\}, \{8, 5, 7, 4, 2, 6\}, \{8, 5, 7, 4, 2, 3\}, \{8, 5, 7, 4, 2, 6, 3\}\}$.

d6 Terminate and return $\{\{8, 5, 7, 4\}, \{8, 5, 7, 4, 2\}, \{8, 5, 7, 4, 2, 6\}, \{8, 5, 7, 4, 2, 3\}, \{8, 5, 7, 4, 2, 6, 3\}\}$.

c4 Terminate and return $\{8, 5, 7\}, \{\{8, 5, 7, 4\}, \{8, 5, 7, 4, 2\}, \{\{8, 5, 7, 4, 2, 6\}, \{8, 5, 7, 4, 2, 3\}, \{8, 5, 7, 4, 2, 6, 3\}\}$.

b4 Terminate and return $\{8\}, \{8, 5\}, \{8, 5, 7\}, \{\{8, 5, 7, 4\}, \{8, 5, 7, 4, 2\}, \{\{8, 5, 7, 4, 2, 6\}, \{8, 5, 7, 4, 2, 3\}, \{8, 5, 7, 4, 2, 6, 3\}\}$.

When all instances of the recursive function $unionRecursive()$ have terminated, we have the sets $\{8\}, \{8, 5\}, \{8, 5, 7\}, \{\{8, 5, 7, 4\}, \{8, 5, 7, 4, 2\}, \{\{8, 5, 7, 4, 2, 6\}, \{8, 5, 7, 4, 2, 3\}, \{8, 5, 7, 4, 2, 6, 3\}$. The use of the REMOVE_EXT function has caused the **union** algorithm to ignore the valid set $\{8, 5, 7, 4, 6\}$.

The operation of the two walkthoughs are the same until d1. At that point, there are two extension points 2 and 4. Extension point 4 is eliminated because the maximum downward cone of 4 is a subset of the maximum downward cone of 2. This prevents the connected convex set $\{8, 5, 7, 4, 6\}$ from being found.

Fix for REMOVE_EXT

We discussed this observation with the authors of [YM04]. This excerpt [YM06] from the correspondence is significant.

> The fault part is in the description of the ELIMI_EXT function, the second point of the three: "Given two extension points u and v, if MAX DC(u, R) $\subset MAXDC(v, R)$, then u can be eliminated

from further consideration.". It should be "for downward extension points u and v, if $u \in predecessors(v)$, an extension vertex combination is redundant if it contains u but not v." It is actually the extension vertices combination that UNION would further recursively dive into that has been eliminated, but not the extension vertex itself as described wrongly in the paper.

This correspondence predates the updated version of the algorithm that was given in [YM08]. In [YM08] the extension point criterion has been changed accordingly.

7.2.3 Input constraints

The **union** algorithm uses an unusual formulation for its I/O criteria. Instead of counting the number of register reads and writes that a custom instruction requires, the **union** algorithm counts the number of operands to the instruction. For example, consider a candidate instruction C that is equivalent to the following assembler code.

```
addiu $2,$3,8
addu $5,$3,$5
sll $5,$3,0x2
```

A normal method for checking I/O constraints would find that C required three inputs — in this example they are the registers $\$2, \$3, \$5$. However, the formulation used by the **union** is that *operands* to the instruction are counted as inputs, so C would require five inputs — the same registers as before and the values 8 and 0x2. However, the values 8 and 0x2 are not fetched from the register file.

This approach makes a significant difference to the total number of connected convex sets found. To illustrate this, one of the input DAGs provided

In	Out	Sets in [YM04]	Generated sets
3	1	159	555
4	1	298	624
5	1	379	633
3	2	233	1,513
4	2	458	4,372
5	2	726	9,479
3	3	233	1,885
4	3	578	6,350
5	3	1,003	19,920

Table 7.2 Sets found by processing the cjpeg benchmark

by the authors of the **union** algorithm has been processed using 'normal' I/O constraints. The input DAG used is the cjpeg benchmark from the PY04 dateset, which has 152 vertices. The experiments in [YM04, YM08] have been replicated to the best of our ability — in particular we have ensured that the set of forbidden vertices is the same as used in [YM04]. In Table 7.2, the number of connected convex sets found by using normal I/O constraints is compared with the number of connected convex sets reported in [YM04]. There is clearly a large difference between the number of connected convex vertex sets in the two experiments.

All information about this test, including the input files and a list of the convex sets found in each combination of constraints are available at [Red].

Although this form of I/O checking does not effect the core **union** algorithm, it is an element of concern. In particular, the *exhaustive* algorithm[1] is at a comparative disadvantage when using this formulation because inputs that come from immediate operands can never be part of a large candidate instruction and so the **exhaustive** algorithm wastes a disproportionate number of recursive calls chasing down 'blind alleys'.

[1]The 2003 version as was used in [YM04]

Inconsistency in results

It is noticeable that for the same input files[2] the **union** algorithm produces slightly different numbers of valid convex sets in [YM04] and [YM08].

For example, in [YM04] there are 438 valid sets for the 'Rinjdael' benchmark with a 3/1 I/O constraint and 159 valid sets for the 'cjpeg' benchmark with the same constraint, but, in [YM08] the numbers of sets are 437 and 166 respectively.

This is surprising because the **union** algorithm is exhaustive and should always find the same number of connected convex sets with each execution.

7.2.4 Treatment of exhaustive algorithm

In [YM04] there is a situation where the **exhaustive** (which is included for comparison purposes) algorithm processes a DAG formed from the benchmark 'blowfish'. It is reported that when the **exhaustive** algorithm has I/O constraints of 3/1, it makes 350,120 recursive calls. However, when it has I/O constraints of 3/2 it makes only 339,058—this is extremely surprising, especially considering the normal operation of the **exhaustive** algorithm.

7.3 Summary

The basic **union** algorithm presented in [YM04] is a provably correct exhaustive enumeration algorithm for finding all connected convex vertex sets in a given DAG. Attempts to improve the efficiency of the algorithm with the REMOVE_EXT function have been successful at the expense of being exhaustive. The author has been unable to produce an implementation of **union** that approaches the speedups reported in [YM04]. The results published in [CMS07] suggest that the performance of **union** does not compare favourably with other methods in the area.

[2]We have confirmed that the input files are the same with the authors of the publication.

Chapter 8

Concluding remarks

In this final chapter, the work and results presented in the preceding chapters are summarised, and some possible future research directions are examined.

8.1 Conclusions

This thesis began by discussing some of the issues facing processor designers, and showing how automatically customised processors could provide many of the advantages of an ASP at a fraction of the cost. A number of novel contributions in the area of candidate instruction enumeration have been presented. For example,

In Chapter 4, existing algorithms for creating a library of candidate instructions were reviewed and given a detailed analysis.

Several algorithms were presented in Chapter 5. These included algorithms for enumerating all of the convex vertex sets and all of the connected convex vertex sets of a DAG. Modifications that allowed these algorithms to function efficiently under forbidden vertex constraints were then presented. These algorithms possessed improved asymptotic complexities and are relatively simple to implement.

The same chapter introduced the idea of a partial solution to the problem of enumerating candidate instructions, potentially allowing designers to direct

the search for candidate instructions.

The chapter concluded with a set of experiments that tested the effectiveness of these algorithms against the current fasted algorithms. The experiments used both real-world benchmarks and synthetic examples as their test cases.

In Chapter 6, a family of algorithms that enumerate convex sets under I/O constraints were presented. These algorithms use varying degrees of sophistication to enumerate convex sets efficiently under I/O constraints. The algorithms show a clear progression, trading greater computational overhead for a more efficient pruning criteria. All algorithms in the Ω family have polynomial time complexity, and perform extremely well in our experiments. Later in the Chapter, we examine a set of conditions that cause the Ω to perform less well. The relevance and possible frequency of these conditions are analysed.

Finally, in Chapter 7, the **union** algorithm was examined in detail. An overview of its use was given, followed by an example. Then a number of problems with the contribution of the **union** algorithm were considered and supported with some numerical data.

8.2 Contributions of this thesis

Notwithstanding the acknowledgement of intellectual property in Appendix C, this section lists those elements of this thesis that the author believes are key contributions.

- ◇ The Ω family of algorithms are a novel, efficient, and elegant new approach to enumerating convex sets under I/O constraints.
- ◇ The Ψ and Φ algorithms for enumerating convex sets without I/O constraints are a new and important step forwards in the research area.
- ◇ The use of the 'partial solution' approach to enumerating all convex sets gives a great degree of flexibility to users and has the potential to be heavily exploited by later phases of the processor customisation process.

- ◇ The detailed review of the **union** algorithm has identified several problems that require attention and has provided support to the performance data in [CMS07].

8.3 Directions for future research

Although this thesis has made a number of novel contributions, it has also uncovered several interesting opportunities for further study.

Extension of algorithms to consider more constraints

One of the most obvious directions for future work is to examine the possibility of adding support for other, less commonly used, constraints to the suite of algorithms that have been presented in this thesis. It would be useful to have algorithms that could efficiently enumerate convex vertex sets, not just by connectivity and I/O constraints, but also by total area and a single cycle constraint.

Candidate equivalence

One of the most important areas for future research, is that of *candidate equivalence*. It is likely that two or more of the convex sets found in the DAG will correspond to the same candidate instruction of the DDG. This is because each vertex represents an operation that may occur many times in a target application,

Moreover, there are a multitude of ways that two instructions can be shown to always compute the same output values for the same set of input values.

It is clear that there is a subset of the set of convex sets in which each candidate instruction features only once. This subset would be extremely valuable because it would cut down greatly on the work that the instruction selection phase needs to perform. Methods are required that can reliably and efficiently

remove superfluous instructions from the library.

Instruction set selection

There is little agreement in the published literature for instruction selection and there remains a strong sense that an ad-hoc method is independently generated for each instance of the problem. It would be useful to have a complete and detailed review of the advantages and disadvantages of such methods and present a framework into which they fit.

However, it is clear that instruction set selection is a small part of the overall design of a customisable chip in the same way that register allocation is a small part of data-flow analysis. The interaction between the different phases of custom processor design is a fertile research area.

Relationship with optimisation and back-end code generation

As discussed in the introduction chapter, the development of automatic generation of instruction sets has only been made possible by progress in the area of automatically generated compiler back-ends and rule-based optimisers.

Although this progress has been enough to enable functioning toolchains, the relationships that the compiler and optimiser have with an automatic instruction selection mechanism are complex. Even though the instructions for a target application must be optimised before candidate instructions are found, a post-selection optimisation is required. This post-selection optimisation phase allows basic blocks to be rewritten in such a way that custom instructions intended for one part of the target application can be used in others.

There is clearly a cyclic dependency here—the knowledge that the target application may be rewritten with reference to custom instructions may influence the selection of the target instructions.

Partial solution with human guidance

In Chapter 5 the concept of a partial solution was introduced. As discussed, a designer may selectively expand elements of the partial solution as they choose. Allowing a human element to select the direction of search will mean that the resulting library will no longer be exhaustive, but it is possible that the quality of instructions in the library will be high.

This trade off between reducing the number of instructions against the possible risk of losing good candidates and their effect on the effectiveness of the resulting instruction set requires detailed investigation and is a fertile research area.

Appendix A

Experimental setup

This Appendix details the experimental setup used to produce the results in this thesis. 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) and timings were taken with reference to the machine's internal time clock.

If experiments did not terminate within two hours, they were considered not to finish in a reasonable amount of time.

A.1 Algorithm implementation

All algorithms featured in this work were implemented in C++ by the author. The only exception is the **split** algorithm, which was implemented in C++ by its creators. The author remains indebted to them for their extremely helpful attitude.

All algorithms in this work that were implemented by the author have been made available online, along with sample candidate instruction lists. Convex sets are stored as arrays of boolean values and the DAGs are represented internally by edge lists, adjacency matrices, and reachability matrices.

A.1.1 Special data structures

Some of the algorithms presented in this thesis require unusual data structures to meet their asymptotic bounds.

In particular, some of our algorithms require an *indexed list* structure that allows constant time lookup, insertion and deletion for a list of size v, while also allowing iteration over the list in $O(v)$ time.

To implement an indexed list L, we maintain arrays of integers L_{NEXT} and L_{PREV} indexed from 0 to n (Recall that n is the size of the input DAG D). If element $e \notin L$ then $L_{NEXT}(e) \neq e$ and $L_{PREV}(e) \neq e$. Locations $L_{NEXT}(0)$ and $L_{PREV}(0)$ point to the start and end of the list respectively. We can iterate over the elements in the list by following the chain of links from $L_{NEXT}(0)$ or $L_{PREV}(0)$ and constant time insertion and deletion can be achieved using functions similar to those found in a linked list.

A.1.2 The split algorithm

The C++ source code of the implementation of the **split** algorithm was made available for our use by its creators. The implementation received allowed for enumeration of connected and disconnected convex sets under I/O constraints. The author made some modifications to the source so that it may run with all I/O checking code removed, this way **split** is not disadvantaged when enumerating convex sets without I/O constraints such as in Chapter 5.

A.2 Correctness of results

Extensive checks were carried out to confirm that exactly the right sets are being enumerated, for small examples the relevant sets are worked out by hand, and for medium sized ones the sets are compared with the sets generated by a brute force method. When dealing with test cases that generate extremely large numbers of sets the number of sets of each size were compared for several

different methods. One of the algorithms used (the **split**) was implemented separately by an entirely unrelated research group, so if all of the methods generate the same number of sets of each size, then it is possible to assume with some confidence that the convex sets found are correct.

A.3 Choice of test cases for each test

For each experiment all practical test cases were tested—results are presented for only a subset of them. The other tests are not included because all algorithms either execute too quickly to measure, or all fail to terminate due to the size of the search space.

A.4 Test cases

Three datasets are used in our experiments, one of which was generated by the author as detailed in Appendix B while the other two have been provided by other research groups.

The combination of these datasets is useful for a number of reasons — firstly because it allows direct comparison with the results published in the relevant source (especially as there are cases where two publications directly contradict each other), secondly it allows us to use test cases where other data-dependency graphs have been constructed from assembler source on different processors and under slightly different conditions.

A.4.1 PY04 dataset

This dataset was provided by the authors of [YM04] and was used in the experiments included in that work. It consists of five large labelled DAGs that are mainly used for experiments involving I/O in Chapter 6.

Like our own test cases they were generated from MiBench benchmarks compiled for the simplescalar architecture.

A.4.2 CMS07 dataset

This dataset was supplied by the authors of [CMS07]. They contain several mibench benchmarks and, like the PY04 dataset, they have been unrolled to some extent to increase the potential for efficiency improvements. The CMS07 dataset was created from code compiled for a MIPS architecture.

A.4.3 RHUL dataset

This dataset has been created in-house by the author to provide a larger range of test cases than is provided by the PY04 and CMS07 datasets. It consists of a large number of test cases ranging dramatically in size. The precise construction of these test cases is detailed in Appendix B and all of the test cases within it are available online at [Red].

A.5 Experimental comments

For some of the experiments in this thesis, the requirement that an algorithm must store all of the convex sets in memory was removed. This was partially because our experimental machine lacked the physical space to store all of the convex sets but also to ensure a fair test, given that the **split** algorithm uses a different storage mechanism to the other algorithms. Unless noted in the text this requirement was only removed on those experiments processing synthetic DAGs.

Forbidden vertices

When it is necessary to forbid some operations of a DDG, the set of operations that are forbidden are: all memory operations, all multiply operations, and all floating point operations.

Appendix B

RHUL dataset

In this appendix a listing is given of the test cases included in the RHUL dataset, and an overview of the creation process. The source code for these test cases is from the Mibench benchmarking suite [GRE+01] and, unless otherwise stated, they have been profiled using the 'runme_large' scripts provided with each benchmark. The information given in this chapter will allow a reader to reconstruct the test cases.

Some of the DDG produced are from library functions rather than parts of the benchmark source code—they are often heavily executed and are a legitimate target for this work. However, care is taken that the same test case does not appear in more than one benchmark.

Instructions for downloading benchmarks from the mibench suite can be found in [GRE+01] and source code for the functions used to create the DAGs can be found at [Red] along with the source code for all algorithms except **split**.

B.1 Test cases

In this section, a description of each test case in the RHUL dataset is given, including the function each one was taken from. In general if *bench1* is the first test case from benchmark 'bench' then *bench1Con* will be the largest connected region of bench1 (assuming bench1 is not a connected DAG) Similarly, *bench1+*

is the DAG formed by augmenting bench1 with external inputs and outputs.

B.1.1 FFT

FFT is a benchmark that performs a set of fast Fourier transformations on an input array.

ID	Vertices	Edges	External Forbidden Vertices	All Forbidden Vertices	Original function
FFT2	53	57	-	-	vprint
FFT2Con	49	53	-	-	vprint
FFT2+	72	75	33	39	vprint
FFT2Con+	66	71	27	33	vprint
FFT3	64	74	-	-	fft_floats
FFT3+	82	94	33	39	fft_floats

B.1.2 Patricia

The Patricia benchmark implements a radix-tree based method for storing IP traffic data.

ID	Vertices	Edges	External Forbidden Vertices	All Forbidden Vertices	Original function
patricia2	52	68	-	-	n/a mpn_divmod
patricia2+	66	82	26	35	mpn_divmod
patricia3	54	68	-	-	n/a log_D
patricia3+	73	77	38	44	log_D
patricia4	34	32	-	-	log_D
patricia4Con	29	28	-	-	log_D
patricia4+	52	50	29	38	log_D
patricia4Con+	47	46	26	34	log_D

B.1.3 Qsort

The qsort benchmark is an implementation of the well-known sorting algorithm.

ID	Vertices	Edges	External Forbidden Vertices	All Forbidden Vertices	Original function
qsort1	21	18	-	-	word_cpy
qsort1Con	17	15	-	-	word_cpy
qsort1+	22	20	13	21	word_cpy
qsort1Con+	20	19	11	19	word_cpy
qsort2	33	33	-	-	qsort_mem
qsort2Con	31	32	-	-	qsort_mem
qsort2+	40	42	19	28	qsort_mem
qsort2Con+	38	41	18	26	qsort_mem
qsort3	33	33	-	-	qsort_mem
qsort3Con	29	31	-	-	qsort_mem
qsort3+	40	42	20	25	qsort_mem
qsort3Con+	36	39	22	25	qsort_mem

B.1.4 Dijkstra

The Dijkstra benchmark calculates the shortest path between every pair of vertices in a (large) graph using Dijkstra's algorithm.

ID	Vertices	Edges	External Forbidden Vertices	All Forbidden Vertices	Original function
dijkstra2	19	20	-	-	dijkstra
dijkstra2Con	15	17	-	-	dijkstra
dijkstra2+	25	26	12	17	dijkstra
dijkstra2Con+	22	24	10	15	dijkstra
dijkstra3	34	25	-	-	dijkstra
dijkstra3Con	18	17	-	-	dijkstra
dijkstra3+	49	40	39	44	dijkstra
dijkstra3Con+	27	26	18	22	dijkstra

B.1.5 Cjpeg

Implementation of the standard compression algorithm. It was profiled using the file 'imageTest.gif', which can be found with the rest of the project source.

ID	Vertices	Edges	External Forbidden Vertices	All Forbidden Vertices	Original function
cjpeg1	43	47	-	-	get_interlaced_row
cjpeg1Con	39	44	-	-	get_interlaced_row
cjpeg1+	65	69	27	39	get_interlaced_row
cjpeg1Con+	51	56	16	28	get_interlaced_row
cjpeg2	161	233	-	-	jpeg_fdct_islow
cjpeg2+	183	255	31	43	jpeg_fdct_islow
cjpeg3	29	26	-	-	start_input_gif
cjpeg3Con	24	22	-	-	start_input_gif
cjpeg3+	39	36	23	32	start_input_gif
cjpeg3Con+	31	30	25	32	start_input_gif
cjpeg4	51	49	-	-	forward_DCT
cjpeg4Con	48	47	-	-	forward_DCT
cjpeg4+	62	59	23	40	forward_DCT
cjpeg4Con+	57	55	19	36	forward_DCT
cjpeg5	168	237	-	-	jpeg_fdct_islow
cjpeg5+	191	260	39	52	jpeg_fdct_islow

B.1.6 Blowfish

This benchmark implements the standard block-cipher algorithm. This benchmark is unusual in that all of the test cases come from the same function.

ID	Vertices	Edges	External Forbidden Vertices	All Forbidden Vertices	Original function
bf1	18	14	-	-	BF_cfb64_encrypt
bf1Con	15	13	-	-	BF_cfb64_encrypt
bf1+	29	25	17	19	BF_cfb64_encrypt
bf1Con+	25	23	9	15	BF_cfb64_encrypt
bf2	32	36	-	-	BF_cfb64_encrypt
bf2+	40	45	17	22	BF_cfb64_encrypt
bf3	337	468	-	-	BF_cfb64_encrypt
bf3+	346	477	14	92	BF_cfb64_encrypt
bf4	34	38	-	-	BF_cfb64_encrypt
bf4Con	31	35	-	-	BF_cfb64_encrypt
bf4+	45	48	14	24	BF_cfb64_encrypt
bf4Con+	36	41	8	18	BF_cfb64_encrypt

B.1.7 Sha

The Sha benchmark implements the SHA secure hash algorithms for producing message digests.

ID	Vertices	Edges	External Forbidden Vertices	All Forbidden Vertices	Original function
sha1	16	15	-	-	sha_transform__FP8SHA_INFO
sha1+	23	22	13	17	sha_transform__FP8SHA_INFO
sha2	38	44	-	-	sha_transform__FP8SHA_INFO
sha2+	44	50	12	28	sha_transform__FP8SHA_INFO
sha3	46	55	-	-	sha_transform__FP8SHA_INFO
sha3+	52	61	10	31	sha_transform__FP8SHA_INFO

B.1.8 Rijndael

Rijndael has been selected as the Advanced Encryption Standard(AES). This benchmark implements the encryption algorithm.

ID	Vertices	Edges	External Forbidden Vertices	All Forbidden Vertices	Original function
rijndael1	1280	1717	-	-	encrypt
rijndael1+	1299	1736	86	387	encrypt
rijndael2	89	107	-	-	set_key
rijndael2+	104	122	29	57	set_key
rijndael3	43	51	-	-	fseek
rijndael3Con	39	49	-	-	fseek
rijndael3+	52	60	16	22	fseek
rijndael3Con+	46	56	11	17	fseek
rijndael4	44	52	-	-	encfile
rijndael4Con	39	49	-	-	encfile
rijndael4+	52	60	15	26	encfile
rijndael4Con+	45	55	10	16	encfile

B.1.9 Susan

Susan is an image recognition package It was developed for use in MRI machines for mapping the brain.

ID	Vertices	Edges	External Forbidden Vertices	All Forbidden Vertices	Original function
susan1	112	150	-	-	susan_corners
susan1+	139	177	35	79	susan_corners
susan2	207	287	-	-	susan_edges
susan2+	234	314	31	123	susan_edges
susan3	40	51	-	-	enlarge
susan3+	53	64	27	35	enlarge
susan4	45	51	-	-	median
susan4Con	38	45	-	-	median
susan4+	57	63	29	42	median
susan4Con+	48	55	24	37	median

B.1.10 GSM

The Global Standard for Mobile (GSM) communications is used for encoding voice transmissions in Europe. This benchmark encodes data streams taken from large speech samples.

ID	Vertices	Edges	External Forbidden Vertices	All Forbidden Vertices	Original function
gsm1	25	23	-	-	Gsm_Short_Term_Analysis_Filter
gsm1Con	23	22	-	-	Gsm_Short_Term_Analysis_Filter
gsm1+	37	35	19	23	Gsm_Short_Term_Analysis_Filter
gsm1Con+	34	33	17	21	Gsm_Short_Term_Analysis_Filter
gsm2	328	442	-	-	Autocorrelation
gsm2+	336	450	57	235	Autocorrelation
gsm3	446	641	-	-	Calculation_of_the_LTP_parameters
gsm3+	475	670	73	236	Calculation_of_the_LTP_parameters
gsm4	98	108	-	-	RPE_grid_selection
gsm4+	117	127	35	62	RPE_grid_selection
gsm5	423	502	-	-	gsm_encode
gsm5+	429	493	41	142	gsm_encode
gsm6	200	219	-	-	RPE_grid_selection
gsm6+	223	242	60	114	RPE_grid_selection
gsm7	82	112	-	-	Weighting_filter
gsm7+	90	120	11	20	Weighting_filter
gsm8	33	38	-	-	Gsm_Preprocess
gsm8+	41	46	13	15	Gsm_Preprocess

B.1.11 Bitcnts

This benchmark tests the bit manipulation abilities of a processor by counting the number of bits in an array of integers by several different methods.

ID	Vertices	Edges	External Forbidden Vertices	All Forbidden Vertices	Original function
bitcnts1	61	63	-	-	bitcnts
bitcnts1Con	55	59	-		bitcnts
bitcnts1+	68	69	22	40	bitcnts
bitcnts1Con+	60	63	32	40	bitcnts
bitcnts2	67	72	-	-	ntbl_bitcount
bitcnts2Con	65	71	-	-	ntbl_bitcount
bitcnts2+	74	78	25	43	ntbl_bitcount
bitcnts2Con+	72	77	23	41	ntbl_bitcount
bitcnts3	48	47	-	-	AR_btbl_bitcount
bitcnts3Con	41	43	-	-	AR_btbl_bitcount
bitcnts3+	55	54	19	44	AR_btbl_bitcount
bitcnts3Con+	46	48	12	36	AR_btbl_bitcount
bitcnts4	41	51	-	-	clock
bitcnts4+	54	63	18	26	clock

B.2 Construction of test cases

The RHUL Dataset is constructed using the well known simplescalar toolset, which is a system software infrastructure used in other aspects of processor design.

To generate test cases, source code from the benchmarks in [GRE+01] were compiled for the simplescalar architecture using a simplescalar-targeted version of gcc with -O3 level of optimisation.

Simplescalar's profiling tools were used to generate execution frequencies for our basic blocks and the 'ss2ddg' tool was used to convert the basic blocks from simplescalar assembler to DAGs.

```
addu  $8,$0,$4
lw    $9,0($5)
multu $9,$7
mflo  $3
mfhi  $4
addu  $3,$3,$2
sltu  $2,$3,$2
addu  $2,$4,$2
addiu $6,$6,1
sw    $3,0($8)
addiu $8,$8,4
addiu $5,$5,4
```

Figure B.1 Low-level code to be converted to a DDG

Three to eight basic blocks where taken from each benchmark to use as test cases. Where possible the most heavily executed basic block was included as a test case. The other test cases where selected by inspecting the code to find basic blocks that were executed reasonably often, and that were large enough to be of interest to our algorithms.

In this section, some of the issues in the construction of DDGs are explored and ways that simplify the enumeration process are presented.

B.2.1 Generation of DDGs from low-level code

The code snippet in Figure B.1 is used as an example of the contents of a basic block. A naïvely generated DDG for the code fragment in Figure B.1 would resemble the one shown in Figure B.2. However, this DDG is not sufficient for our purposes.

Operations storing more than one value

Figure B.1 shows the candidate instruction C that consists of the multiply operation in Figure B.2 and its adjacent load operation. Notice that in Figure B.2 there are two data values leaving the multiply instruction using two different registers. Recall that the number of outputs of a convex set is defined as the

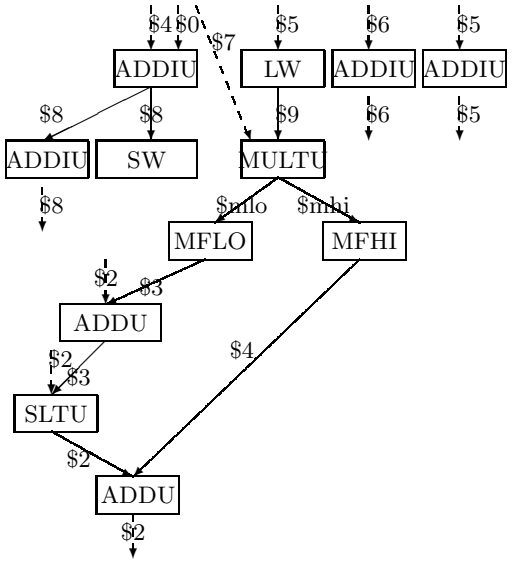


Figure B.2 DDG for example code in Section B.2.1

number of vertices in the convex set that directly depend on vertices outside the set. Under this definition, candidate instruction C will have one output, however the candidate instruction C will clearly store two values to the register file rather than one.

It would be possible to alter our definition of $OUT(C, D)$ in such a way that these extra outputs were counted, but such a change would mean that many of the previously presented algorithms would either no longer be able to enumerate convex sets under I/O constraints or would need heavy modification.

Our solution is that if an operation a stores n values then it must be represented by n vertices, if a was dependent on another vertex in the DDG then each of the new vertices must also be dependent on that vertex. In Figure B.2.1 a DDG is shown for the code in Figure B.1 with this modification taken into account.

External inputs and outputs

The second issue is that of external inputs and outputs. The code segment in Figure B.1 uses the values in \$5,\$6,\$8,\$9,\$0, which are all written to from

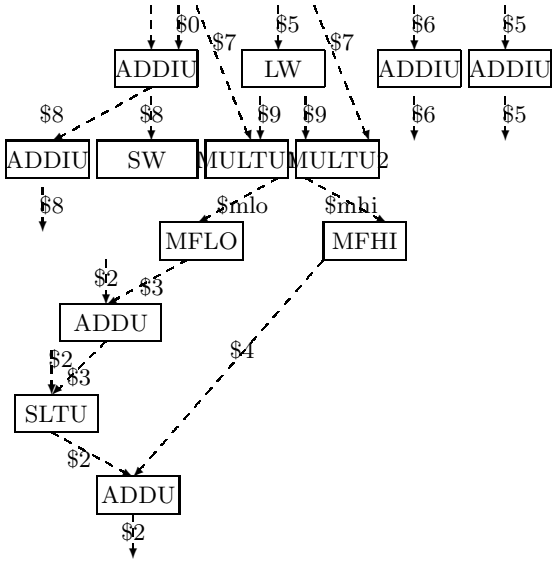


Figure B.3 DDG with support for operations storing more than one value

outside of the basic block. Once again this may cause problems when examining the inputs and outputs of a given candidate instruction so the DDG is augmented with extra vertices that represent the values entering and leaving the basic block. Each register that is written from outside the basic block will be represented by an individual vertex.

External outputs are dealt with in much the same way, except it is sufficient to have a single extra vertex that is connected to all of the vertices that produce values used outside the basic block.

A DDG generated from the code segment in Figure B.1 that incorporates external inputs and outputs is shown in Figure B.4.

Clearly the extra vertices can not be part of a candidate instruction as they represent the inputs or outputs to a basic block so they are made part of the forbidden vertex set, see Section 2.4.1 for more details on forbidden vertices.

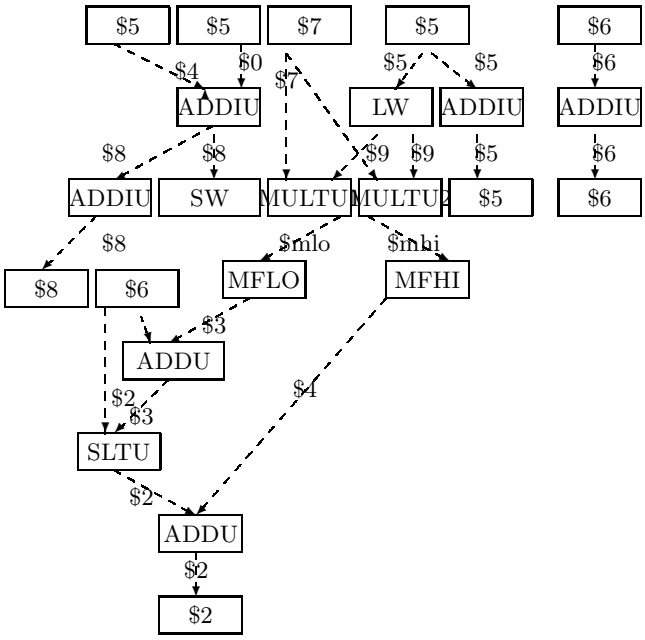


Figure B.4 DDG for example code with support for external inputs and outputs added

B.2.2 Data dependency analysis

It must be noted that when augmenting DAGs with external output vertices, each register that could have been used outside the basic block was connected to an external output. Without doubt, some of these connections made were unnecessary and the register value in question was not used. The use of data-dependency analysis would have been able to pinpoint these cases and allow for a more accurate representation of the information flow within the basic block.

Appendix C

Acknowledgement of intellectual property

Parts of the intellectual property in this work were developed in conjunction with other members of the research group.

C.1 The Φ algorithm

The insight that if X is a convex vertex set of a DAG D, then $X \setminus \{x\}$ is also convex if x is either a source or a sink vertex of D and a sample algorithm exploiting this insight were supplied by Dr S. Gerke and Dr P. Balister. The algorithm was heavily rewritten and modified by the author before implementation.

C.2 The Ψ algorithm

The original algorithm of generation of convex connected sets was provided by Dr A. Yeo. The author added support for forbidden vertices and all work relating to the extension of the algorithm to disconnected convex vertex sets.

C.3 The Ω family

The use of flows in a network was proposed and an almost complete algorithm was presented by Dr A.Yeo. As a result of collaboration with the group a full

algorithm was prototyped. The forward and inverse edge based approach for measuring flow is the work of Dr A.Yeo, however large parts of the algorithm where later rewritten by the author particularly in the area of output set enumeration and the recursive flow of the algorithm. Lemma 24 was originally the work of Dr A.Yeo but has been modified by the author.

C.4 Toolchain

The application for generating DAGs from assembler code was originally implemented by Dr A. Johnstone and heavily extended by the author.

C.5 Artwork

Example 2 was originally drawn by Dr E. Scott.

Bibliography

[ADM+07] Kubilay Atasu, Robert G. Dimond, Oskar Mencer, Wayne Luk, Can Özturan, and Günhan Dündar. Optimizing instruction-set extensible processors under data bandwidth constraints. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 588–593, San Jose, CA, USA, 2007. EDA Consortium.

[ADO05] Kubilay Atasu, Gunhan Dundar, and Can Ozturan. An integer linear programming approach for identifying instruction-set extensions. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 172–177, New York, NY, USA, 2005. ACM Press.

[AGT89] Alfred V Aho, Mahadevan Ganapathi, and Steven W K Tjiang. Code generation using tree matching and dynamic programming. *ACM transactions on Programming Languages and Systems, Vol 11, No 4*, 1989.

[ALE02] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[API03] Kubilay Atasu, Laura Pozzi, and Paolo Ienne. Automatic application-specific instruction-set extensions under microarchitec-

tural constraints. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 256–261, New York, NY, USA, 2003. ACM Press.

[App87] A. Appel. Concise specifications of locally optimal code generators, 1987.

[BBD+05] Partha Biswas, Sudarshan Banerjee, Nikil Dutt, Laura Pozzi, and Paolo Ienne. Isegen: Generation of high-quality instruction set extensions by iterative improvement. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1246–1251, Washington, DC, USA, 2005. IEEE Computer Society.

[BGJ+02] Massimo Baleani, Frank Gennari, Yunjian Jiang, Yatish Patel, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Hw/sw partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 151–156, New York, NY, USA, 2002. ACM Press.

[BJG00] Jorgen Bang-Jensen and Gregory Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, August 2000.

[BP06] Paolo Bonzini and Laura Pozzi. Code transformation strategies for extensible embedded processors. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 242–252, New York, NY, USA, 2006. ACM.

[BP07] Paolo Bonzini and Laura Pozzi. Polynomial-time subgraph enumeration for automated instruction set extension. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1331–1336, New York, NY, USA, 2007. ACM Press.

[CFH+05] Jason Cong, Yiping Fan, Guoling Han, Ashok Jagannathan, Glenn Reinman, and Zhiru Zhang. Instruction set extension with shadow registers for configurable processors. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 99–106, New York, NY, USA, 2005. ACM Press.

[CFHZ04] Jason Cong, Yiping Fan, Guoling Han, and Zhiru Zhang. Application-specific instruction generation for configurable processor architectures. In *FPAG '02*, pages 183–189, 2004.

[CHZ05] J. Cong, Guoling Han, and Zhiru Zhang. Architecture and compilation for data bandwidth improvement in configurable embedded processors. In *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 263–270, Washington, DC, USA, 2005. IEEE Computer Society.

[CKY+99] Hoon Choi, Jong-Sun Kim, Chi-Won Yoon, In-Cheol Park, Seung Ho Hwang, and Chong-Min Kyung. Synthesis of application specific instructions for embedded DSP software. *IEEE Transactions on Computers*, 48(6):603–614, 1999.

[CMS07] X. Chen, D. L. Maskell, and Y. Sun. Fast identification of custom instructions for extensible processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):359–368, 2007.

[CPH03] Newton Cheung, Sri Parameswaran, and Jörg Henkel. Inside: Instruction selection/identification & design exploration for extensible processors. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, page 291, Washington, DC, USA, 2003. IEEE Computer Society.

[CZM03] Nathan Clark, Hongtao Zhong, and Scott Mahlke. Processor acceleration through automated instruction set customization. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 129, Washington, DC, USA, 2003. IEEE Computer Society.

[DTM04] Elena Dubrova, Maxim Teslenko, and Andrés Martinelli. On relation between non-disjoint decomposition and multiple-vertex dominators. In *ISCAS (4)*, pages 493–496, 2004.

[Ert99] M. Anton Ertl. Optimal code selection in dags. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 242–249, New York, NY, USA, 1999. ACM Press.

[FBF+00] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *The 27th Annual International Symposium on Computer architecture 2000*, pages 203–213, New York, NY, USA, 2000. ACM Press.

[FF56] L.R. Ford and D.R. Fulkerson. Maximal flow through a network. *Canadian Journal ofMathematics*, 1956.

[FF57] L.R. Ford and D.R. Fulkerson. A simple algorithm for finding maximal network flows and an application to the hitchcock problem. *Canadian Journal of Mathematics*, 1957.

[FHP92] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst.*, 1(3):213–226, 1992.

[GF85] Mahadevan Ganapathi and Charles N. Fischer. Affix grammar driven code generation. *ACM Trans. Program. Lang. Syst.*, 7(4):560–599, 1985.

[GG78] Susan Graham and R Steven Glanville. A new method for compiler code gneration. *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, 1978.

[GJR+07] Gregory Gutin, Adrian Johnstone, Joseph Reddington, Elizabeth Scott, Arezou Soleimanfallah, and Anders Yeo. An algorithm for finding connected convex subgraphs of an acyclic digraph. 2007.

[GJR+08a] Gregory Gutin, Adrian Johnstone, Joseph Reddington, Elizabeth Scott, Arezou Soleimanfallah, Anders Yeo, Paul Balister, and Stephanie Gerke. An algorithm for finding connected convex subgraphs of an acyclic digraph. *Journal of discrete algorithms (to appear)*, 2008.

[GJR+08b] Gregory Gutin, Adrian Johnstone, Joseph Reddington, Elizabeth Scott, and Anders Yeo. An algorithm for finding input-output constrained convex sets in an acyclic digraph. In *Procedings of the 34th International Workshop on Graph-Theoretic Concepts in Computer Science*, 2008.

[Gon00] Ricardo E. Gonzalez. Xtensa — A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, /2000.

[GP03] David Goodwin and Darin Petkov. Automatic generation of application specific processors. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 137–147, New York, NY, USA, 2003. ACM Press.

[GPY+06] Carlo Galuzzi, Elena Moscu Panainte, Yana Yankova, Koen Bertels, and Stamatis Vassiliadis. Automatic selection of application-specific instruction-set extensions. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 160–165, New York, NY, USA, 2006. ACM Press.

[GRE+01] Matthew R Guthaus, Jeffery S Ringenberg, Dan Ernst, Todd M Auttin, Trevor Mudge, and Richard B Brown. Mibench: A free, commerically representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[Gup92] Rajiv Gupta. Generalized dominators and post-dominators. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 246–257, New York, NY, USA, 1992. ACM.

[HD94] Ing-Jer Huang and Alvin M. Despain. Synthesis of instruction sets for pipelined microprocessors. In *DAC '94: Proceedings of the 31st annual conference on Design automation*, pages 5–11, New York, NY, USA, 1994. ACM.

[HO82] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, 1982.

[HP92] John L. Hennessy and David A. Patterson. *Computer Architecture; A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[IL06] Paolo Ienne and Rainer Leupers. *Customizable Embedded Processors: Design Technologies and Applications (Systems on Silicon).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[JLM06] Ramkumar Jayaseelan, Haibin Liu, and Tulika Mitra. Exploiting forwarding to improve data bandwidth of instruction-set extensions. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 43–48, New York, NY, USA, 2006. ACM Press.

[JS02] Adrian Johnstone and Elizabeth Scott. Cs2490 time and space:designing efficient programs. Technical report, Royal Holloway, University of London, 2002.

[KLST04] Uwe Kastens, Dinh Khoi Le, Adrian Slowik, and Michael Thies. Feedback driven instruction-set extension. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 126–135, New York, NY, USA, 2004. ACM Press.

[LB00] Rainer Leupers and Steven Bashford. Graph-based code selection techniques for embedded processors. *ACM Trans. Des. Autom. Electron. Syst.*, 5(4):794–814, 2000.

[Len90] Thomas Lengauer. *Combinatorial algorithms for integrated circuit layout*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[PAI06] Laura Pozzi, Kubilay Atasu, and Paolo Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(7):1209–1229, 2006.

[PFH92] Todd A Proebsting, Christopher W Fraser, and Robert R Henry. Burg - fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices Vol 27, Issue 4*, 1992.

[PI05] Laura Pozzi and Paolo Ienne. Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In

CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems, pages 2–10, New York, NY, USA, 2005. ACM Press.

[PKP08] Nagaraju Pothineni, Anshul Kumar, and Kolin Paul. Exhaustive enumeration of legal custom instructions for extensible processors. In *VLSID '08: Proceedings of the 21st International Conference on VLSI Design*, pages 261–266, Washington, DC, USA, 2008. IEEE Computer Society.

[Poz01] Laura Pozzi. Automatic topology-based identification of instruction-set extensions for embedded processors, 2001.

[PPIM03] Armita Peymandoust, Laura Pozzi, Paolo Ienne, and Giovanni De Micheli. Automatic instruction set extension and utilization for embedded processors. In *ASAP*, pages 108–. IEEE Computer Society, 2003.

[Red] J Reddington. Implementations and documents relating to thesis: cs.rhul.ac.uk/home/joseph/thesis/.

[S:A] Arm, http://www.arm.com.

[SIH+91] Jun Sato, Masaharu Imai, Tetsuya Hakata, Alauddin Y. Alomary, and Nobuyuki Hikichi. An integrated design environment for application specific integrated processor. In *ICCD '91: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 414–417, Washington, DC, USA, 1991. IEEE Computer Society.

[S:M08] Mips, http://www.mips.com, 2008.

[SRRJ02] Fei Sun, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Synthesis of custom processors based on extensible platforms. In

ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design, pages 641–648, New York, NY, USA, 2002. ACM Press.

[S:w08] Wikipedia entry for 'basic block', 2008.

[Tji85] S. W. K. Tjiang. Twig reference manual. Technical report, AT&T Bell Laboratories, 1985.

[VWG+04] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena Moscu Panainte. The molen polymorphic processor. *IEEE Trans. Comput.*, 53(11):1363–1375, 2004.

[Wei] Eric W Weisstein. 'christmas stocking theorem' from *mathworld*–a wolfram web resource, http://mathworld.wolfram.com/christmasstockingtheorem.html.

[YG] Anders Yeo and Gregory Gutin. On the number of connected convex subgraphs of a connected acyclic graph. (submitted).

[YM04] Pan Yu and Tulika Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 69–78, New York, NY, USA, 2004. ACM Press.

[YM06] Pan Yu and Tulika Mitra. Personal communication, December 2006.

[YM07] Pan Yu and Tulika Mitra. Disjoint pattern enumeration for custom instructions identification. In *Procedings of the International Conference on Field Programmable Logic and Applications, 2007. FPL 2007.*, 2007.

[YM08] Pan Yu and Tulika Mitra. Efficient custom instruction identification with exact enumeration. Technical report, National University of Singapore, 2008.