TOWARDS EFFICIENT STOCHASTIC OPTIMIZATION OF FUNCTIONS OF CONVEX SETS

BY

CHRISTIAN J. CONVEY

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2013

DOCTOR OF PHILOSOPHY DISSERTATION

OF

CHRISTIAN J. CONVEY

APPROVED:

Dissertation Committee:

Major Professor Dr. Lutz Hamel, Ph.D.

Dr. Lisa DiPippo, Ph.D.

Dr. Joan Peckham, Ph.D.

Dr. Woong Kook, Ph.D.

Dr. Nasser H. Zawia, Ph.D.

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2013

ABSTRACT

Let D be a DAG and let $\mathcal{X}$ be any non-empty subset of D's vertices. $\mathcal{X}$ is a **convex set of** D if D contains no path that originates in $\mathcal{X}$, then visits one or more vertices not in $\mathcal{X}$, and then re-enters $\mathcal{X}$. This work presents basic convexity algorithms for creating, growing, and shrinking convex sets using two different approaches: predecessor and successor sets, and topological sorts. It shows that the algorithms based on predecessor and successor sets typically have higher asymptotic running times than those based on topological sorts. However, when creating a convex set based upon a potentially non-convex set of "seed" vertices, the use of predecessor and successor sets permits the creation of a convex set which is the uniquely smallest superset of the seeds.

This work also considers the problem of stochastically searching for a global minimum over all convex sets of a given DAG, using the basic convexity algorithms described above. This work demonstrates the existence of such an algorithm that, when run for an unbounded finite number of iterations, the probability of it covering the entire search space approaches one.

This work presents one possible mapping which extends this work's optimization of a single parallel task to the optimization of task-parallel programs with multiple tasks. This mapping is studied experimentally. The results demonstrate a significant (32%) speed improvement over a human-crafted parallelization of the same program, suggesting the possible merit in this work's approach to automated parallel-program optimization.

ACKNOWLEDGMENTS

Thanks to Gregory Gutin of the Royal Holloway University of London and Jørgen Bang-Jensen of the University of Southern Denmark, Odense for helpful discussions regarding the convex set enumeration algorithm presented in their excellent book [1].

Thanks to Pat Miller of J.P. Morgan and Jim McGraw of Lawrence Livermore National Laboratory (USA) for considerable help in understanding the background and semantics of the IF1 and IF2 dataflow languages presented in [2, 3].

Thanks to the dedicated staff at the technical library of the Naval Undersea Warfare Centers, Newport (USA) for timely and thorough assistance in obtaining background materials to support this work.

Thanks to Lorraine Berube, secretary of the Computer Science and Statistics department, for her never-ending cheer and helpfulness.

And finally, great thanks to my academic advisor, Lutz Hamel, for his tutelage and wise counsel during my studies of this fascinating subject.

DEDICATION

I dedicate this thesis to my parents, James and Stephanie, my wife Jennifer, her mother Lois, and my children. They have all made tremendous sacrifices to allow me to complete these studies. No words can adequately express my gratitude.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Overview

This work develops a machine-learning technique to automate the work of adapting existing computer codes to take advantage of multi-processor / multi-core computers.

This work focuses on a model of parallel programming called **task-parallelism**, in which a computer source code is divided into sections called **tasks** (see Section 2.10). When the resulting program is run, different tasks can run on different processor cores, assuming that certain interdependencies are satisfied. In this way parallel execution is obtained.

For all but the smallest programs, there are typically many different valid ways to divide a program code into tasks. Generally, different groupings (called **task sets**) result in different running times for the resulting program, due to factors such as cache warmth and thread scheduling overhead. A number of different approaches may be taken to execute a program code that has been divided according to some task set. In this work's proof-of-concept (see Chapter 8), the program and specified task set are converted into a multi-threaded C++ program, which is then compiled to a native executable using any standard C++ compiler. This translation process is described in detail in Section 3.3.

The **full task-set optimization problem** (see Section 3.4) is the optimization problem of finding a task set which leads to a parallel program with an approximately minimal running time.

While the full task-set optimization problem is the motivation for this present research, this work focuses on a restricted form of the problem, called the **single-task optimization problem** (see Section 3.5). The relationship between these two problems is briefly discussed below, and in much more detail in Chapter 3. Because each task is also a **convex set** (see Section 2.9), the single-task optimization problem is also called the single-convex-set optimization problem. This work treats these two terms as interchangeable.

For this work, any program to be optimized is represented as a **dataflow graph (DFG)** (see Section 3.1). When representing program codes as dataflow graphs, we represent an individual task as some subset of the DFG's vertices. Figure 1 (page 19) provides an example of a dataflow graph for a program which computes a simple mathematical expression. [1]

A task set is a set of tasks, and thus is modeled as a set of vertex sets. In this work, the pair of a DFG and a task set is considered to be a complete specification of a task-parallel program. Other details which may affect a task-parallel program's running time, such as how the (DFG, task-set) pair is mapped to C++ source code, are outside the scope of this research and are treated as fixed details during the optimization process. Figure 5 (pages 88-89) provides an example of different task sets defined over the same DFG.

The key complicating factor for the optimization problems considered in this work is that not all task sets are valid. For a given program, there may be some task sets which would lead to the resulting program to deadlock during execution. This problem is partially avoided by requiring that each task to be a convex set. However, as was discovered in the latter stages of this research effort, the convexity of each individual task in a set is not sufficient to ensure the absence of deadlock at runtime. This issue is treated in depth in Chapter 3.

1.2 Three Problem Levels

This thesis develops three broad groups of algorithms: (1) the Full Task-set Optimization Algorithm, (2) the Single Convex-Set Optimization Algorithm, and (3) Basic Convexity

[1]For simplicity, the DFG model used in this work cannot represent branching or looping. However, more expressive DFG languages exist and have been used in other systems.

Algorithms.

These three groups of algorithms, and the relationships between the groups, are described briefly below.

1.2.1 Full Task-set Optimization Algorithm

The full task-set optimization algorithm directly addresses the automatic parallelization problem which motivates this thesis. This algorithm is studied in the proof-of-concept described in Chapter 8.

In the early stages of this research, our goal was to focus on the development and study of an algorithm which directly addressed the full task-set optimization problem. Furthermore, we strove to ensure that given an unbounded number of iterations, the probability of the algorithm finding a globally optimal value would approach one.

Our early approach to this algorithm was to develop another algorithm which mutates a single convex set (see Subsection 1.2.2, below). We designed the full task-set optimization algorithm to execute one instance of the single convex-set mutation algorithm for each task in the evolving task set. The evolved task-set would simply be the collection of convex sets evolved in this manner. We assumed that as the full task-set optimization algorithm ran for an unbounded number of iterations, the probability of it discovering a globally optimal task set would approach one. We expected this to be an emergent property resulting from trivially combining individual convex sets whose mutation algorithm had a similar probability property.

This approach to the research was necessarily changed when we discovered that some collections of convex sets did not form a valid task set. When a task set contains exactly one task, deadlock is avoided precisely when that task is a convex set. However, when a task set contains multiple tasks, deadlock may arise even when each individual task is a convex set. This problem is discussed in detail in Subsection 3.3.1.3.

To address this problem, we modified our full task-set optimization algorithm to alter individual tasks, as necessary, to eliminate any deadlock which might arise when combining those tasks into a single task set. Our approach is described in Appendix E. However, this resulted in a non-trivial mapping from individually mutated convex sets to a full task set. The non-triviality of that mapping raised serious doubts that the resulting full task-set optimization algorithm could be proven to discover globally optimal task sets given an unbounded number of iterations.

Due to the then-limited amount of time to complete this thesis' research, a decision was made to refocus our efforts on proving the above-mentioned probability quality for only our single task-set optimization algorithm. Our hope, and a central assumption of this work, is that by proving that quality for our single convex-set optimization algorithm, we provide a stepping-stone for future attempts to develop a full task-set optimization problem with that probability property.

The proof of concept in Chapter 8 demonstrates that regardless of whether or not our full task-set optimization algorithm would probably discover a globally optimal task set given enough iterations, the algorithm can in practice discover useful task-parallel optimizations.

1.2.2 Single Convex-Set Optimization Algorithm

This algorithm (Algorithm 3 in Chapter 7) discovers a single convex set which approximately minimizes a specified cost function. As the algorithm runs for an unbounded number of iterations, the probability of it discovering a global minimum asymptotically approaches one.

Note that the full-convex-set optimization algorithm described above does not use the entirety of this single-convex-set optimization algorithm. Instead, it uses this algorithm's technique for mutating individual convex sets, which is arguably the most interesting aspect of this algorithm.

This thesis establishes the following qualities for the presented single-convex-set optimization algorithm:

Asymptotically Complete Search/Optimization As discussed above, this is the property that if run for an unbounded number of iterations, the probability of this algorithm examining every convex set of the specified DFG approaches one. This quality is proven in Subsection 7.5.3.

When this algorithm is used in its full form, examining every convex set of the DFG implies finding a global minimum.

As noted earlier, this thesis' implementation of the full-task-set optimization algorithm uses just part of this single-convex-set optimization algorithm. However, even the excerpted portions of this algorithm visit all convex sets of the DFG with a provable asymptotic probability of one.

Per-iteration Efficiency We show that each individual iteration of this algorithm has a running time that is polynomial with respect to the size of the DFG. We establish this (Subsection 7.5.2) to demonstrate the practical usefulness of the algorithm.

Usefulness to the Full Task-set Optimization Problem Because the single-convex-set optimization problem is not identical to the full-task-set optimization algorithm which motivates this work, it remains necessary to show that this algorithm is at least a step towards solving the motivating problem. This thesis' proof-of-concept (Chapter 8) provides empirical evidence of this.

This algorithm draws upon a third group of algorithms, basic convexity algorithms, described below.

1.2.3 Basic Convexity Algorithms

Appendix C presents a collection of algorithms which create a new convex set, or grow or shrink an existing convex set. These algorithms are required by the single-convex-set optimization algorithm, and by implication the full-task-set optimization algorithm, both described above.

Two alternative groups of these algorithms are developed and studied. One group of algorithms is based on predecessor and successor sets (Chapter 5), and the other group is based on topological sorts (Chapter 4). The comparative merits of these two approaches are discussed in Section 7.4.

1.3 Structure of Thesis

The remainder of this work is structured as follows. This chapter presents this work's contribution (Section 1.4) and related work (Section 1.5).

Chapter 2 presents the notation and basic graph theory concepts used in this work.

Chapter 3 describes in detail the a parallel-program optimization problems which motivate this present work.

Chapters 4 and 5 develop two distinct approaches to creating, growing, and shrinking some arbitrary convex set of a DAG. Chapter 4 develops these basic convexity operations based upon a DAG's topological sorts. Chapter 5 presents the same basic convexity operations, computed in terms of the predecessor and successor sets.

Chapter 6 establishes that given some DAG D and any two convex sets of D, $\mathcal{X}$ and $\mathcal{Y}$, one can always evolve $\mathcal{X}$ into $\mathcal{Y}$ using a sequence of single-vertex additions, deletions, or replacements, such that the set resulting from each modification is itself a convex set of D.

Chapter 7 discusses the design of this work's single-convex-set stochastic optimization algorithm, Algorithm 3. Algorithm 3 is implemented in terms of topological sort-based basic convexity algorithms (see Chapter 4 and Appendix D). However, the optimization algorithm may be trivially modified to instead use operations based on predecessor/successor sets (see Chapter 5 and Appendix D). The relative merits of these two approaches is discussed. This chapter also discusses a modification to the optimization algorithm in order to achieve local refinement.

Chapter 8 presents experimental results obtained in the empirical validation of this work's algorithms.

Chapter 9 provides the thesis' conclusions and suggestions for future work.

Appendix A details the primitive operations used by this work's algorithms. For each primitive operation we state an asymptotic running time and, when appropriate, the randomness properties exhibited by the operation.

Appendix B presents utility algorithms which are called by this work's higher-level algorithms, but which are not the focus of this work.

Appendix C presents algorithms for initializing, growing, and shrinking an arbitrary convex set of a DAG. This appendix discusses both the basic algorithms based upon topological sort as well as those based on predecessor and successor sets. A running-time analysis is given for each algorithm in this appendix, to support the running-time analysis of the single-convex-set algorithm, Algorithm 3. Analyses are also given for the randomness properties of the algorithms which grow or shrink convex sets, to support related proofs in Section 7.5.

Appendix D presents algorithms which are not used by the optimization algorithm which is the focus of this work (Algorithm 3), but are instead used in the constructive proofs of Algorithm 3's correctness.

Appendix E presents an algorithm for the deconfliction of task sets, an issue raised in Subsection 3.3.1.3.

Appendix F presents one of the task-parallel C++ codes produced during that same experiment.

1.4 Contribution

In Chapter 4, Theorem 4.2.2 establishes that for any convex set $\mathcal{X}$ of some DAG D, there exists a topological sort of D in which the vertices of $\mathcal{X}$ appear as a contiguous subsequence.

Theorems 4.2.6 and 4.2.7 show that for any convex set $\mathcal{X}$ of some DAG D, one can always generate a superset or subset of $\mathcal{X}$, having an arbitrary order (i.e., number of vertices), that is also convex set of D.

In Chapter 5, Lemma 5.2.11 shows that the formula used in [3, Fig. 4-7] and possibly referenced in [2], called in this present work an *internal path closure*, is sufficient to ensure the convexity of a set.

Corollary 5.3.5 shows that the internal path closure of any set $\mathcal{X}$ is the uniquely smallest improper superset of $\mathcal{X}$ that is also convex.

Theorem 5.4.7 provides a precise formula for identifying which vertices in some convex set $\mathcal{X}$ have the quality that deleting just that one vertex from $\mathcal{X}$ yields another convex set.

Section 7.5 presents an efficient stochastic optimization algorithm (Algorithm 3) whose domain is all convex sets of a DAG. As the algorithm's iterations increase without bound, its probability of exploring the entire search space approaches 1. To the author's knowledge this is the first presentation of an optimization algorithm over this domain having these qualities.

In Appendix D, Algorithm 19 demonstrates that given any DAG D and any two sets $\mathcal{X}$ and $\mathcal{Y}$ that are convex sets of D, one can always evolve $\mathcal{X}$ into $\mathcal{Y}$ using a sequence of single-vertex additions, deletions, and replacements, such that each intermediate set produced by the single-vertex changes is itself a convex set of D.

1.5 Related Work

1.5.1 Iterative / Optimal Compiling

Numerous projects have used machine learning and/or iterative compilation to improve the optimization heuristics employed by compilers[4, 5]. Fursin et al[6] developed a system that repeatedly compiles a target program, searching for the optimal optimization parameters for each individual construct in the target. Cooper et al[7] seek to reduce the time required for iterative compilation by using models rather than actual executions to determine the running times of various optimized compilations of the target program.

1.5.2 Automated Algorithm Selection

The FFTW project[8, 9] provides a library for calculating discrete Fourier transforms (DFTs). It benchmarks the running times of algorithm fragments, and then composes them to handle arbitrarily large inputs. The Spiral[10] system searches through the space of valid employments of interchangeable algorithms in order to minimize program running time. As in FFTW, Olszewski and Voss[11] use benchmark-informed search for optimal algorithm choices, but do so for parallel sorting algorithms. The PetaBricks project[12] goes further by allowing application programmers to specify a set of functionally similar algorithms, which the PetaBricks system seeks to optimally compose.

1.5.3 Task Partitioning and Forking

Extensive research has occurred regarding finding the optimal partitioning of applicative programs into tasks for parallel execution. This research typically involves partitioning a dataflow-graph representation of the target program[13, 14, 3]. Smyk et al[15] describe a genetic algorithm for partitioning dataflow graphs, but with the goal of defining a computational mesh for a distributed system, rather than a set of tasks to be activated as needed on a shared-memory system.

The Mul-T system[16] requires programs to explicitly indicate which function calls may be safely executed in parallel, unlike my proposed work and [17] whose functions are parallel-safe by construction. Our work also can reorganize the original program to have parallel functions that never existed in the original source code, whereas Mul-T cannot.

1.5.4 Conditional Parallelism

The Mul-T system[16] decides at runtime, based on issues such as current system load, whether to spawn a function call as a parallel task or to instead directly call the function from within the thread of the caller. Rus et al[18] use runtime analysis to decide the safety, not the profitability, of executing a program fragment asynchronously. Huelsbergen et al[19] use static analysis to estimate the cost of each function call based on the size of its parameters, and at runtime use that information to decide whether or not the overhead of an asynchronous call is worth incurring.

Duran et al[20] and Prechelt et al[21] use machine learning to decide whether to make a recursive function call synchronously or asynchronously based on the level of recursive nesting and on current system load. Both of these systems use decision functions whose form is fixed by their learning systems.

1.5.5 Parallelization of Sisal Programs

Various projects have parallelized *Sisal* or *IF1* codes. Sarkar[3] uses modeling to estimate the execution cost of each task created by a particular partitioning of a target program's *IF1* representation, ultimately producing tasks whose estimated running time always exceeds some minimum threshold. The *OSC* compiler[22] offers automatic data parallelism of some loops (via *loop slicing*), based on factors including the estimated cost of certain units of work, and on thresholds specified during compilation. Beard[23] developed a *Sisal* compiler back-end to target distributed (e.g., message-passing) computers.

1.5.6 POSC

This work is most directly a continuation of the work done by Sarkar and Cann for the *POSC*[17] extension to the *OSC* compiler. In *POSC*, each function is potentially a different child task; compile-time-specified *dofork flags* determine whether a given function call is performed inline or asynchronously via a `fork` operation. My work goes beyond this by actually formulating new functions to be parallelized.

1.5.7 Convex Sets

Sarkar and Hennesy [2] describe an approach to parallelization in which the vertices of a dataflow graph may be grouped together into one or more parallel tasks. Their paper recognizes the need for convexity to ensure freedom from deadlock at runtime. They present an algorithm which potentially generates non-convex tasks, which are later converted into similar, convex sets using what the author's call the *convex hull* of the non-convex set. The paper provides no obvious definition for that term, but later work ([3]) suggests that their convex hull is the same as this present work's internal path closure concept (see Section 5.3).

The algorithm in [2, 3] uses a deterministic optimization technique to efficiently obtain a collection of convex sets representing parallel tasks. However, the problem which they seek to solve has NP time complexity ([3, section 4.2-4.3]), and so their approach cannot guarantee optimality.

Sanchez and Trystram [24] use a genetic algorithm to obtain a collection of tasks from a dataflow graph. As with [2, 3], all convex sets in a collection are mutually disjoint. However, [24] assigns every vertex to some convex set, whereas [2, 3] permits some vertices in the dataflow graph to remain unassigned. It is not clear from this paper whether or not the genetic algorithm has a non-zero probability of searching the entire problem space when starting with an arbitrary seed and allowed to run for a finite but unbounded number of iterations.

Pecero and Bouvry [25] provide a different genetic algorithm to obtain a collection of convex sets in a dataflow graph. As with [2], a local heuristic is employed to convert a non-convex task into a convex one. However, whereas [2] obtains convexity by adding vertices, Pecero and Bouvry's algorithm obtains convexity by deleting vertices.

The mutation operator of [25] splits some convex set $\mathcal{X}$ into two convex sets: for some $u \in \mathcal{X}$, the sets $\{u\}$ and $(\mathcal{X} \setminus \{u\})$. [25] states that u is drawn from the "top" or "bottom" of $\mathcal{X}$, suggesting a recognition of the validity of Theorem 5.4.5 in Section 5.4 of this present work.

Andersen *et al* [26] consider a problem domain in which groups of operations are permitted to overlap. This is motivated by the fact that in distributed systems, the performance impact of moving data between processors may outweigh the cost of redundantly computing the data at whichever additional processors require it.

Baslister *et al* [27] and Bang-Jensen and Gutin [1, section 17.2.3] present an efficient algorithm for enumerating all convex sets of any directed acyclic graph.

1.5.8 Coverage of Search Space by Stochastic Algorithms

Rudolf and others have studied the problem of proving that certain classes of stochastic optimization algorithms cover the entire search space of a specified problem, given enough iterations. Rudolf uses Markov chains to demonstrate that for a particular conception of genetic algorithms, convergence to a globally optimal solution is entirely contingent on whether or not elitism is used to retain the best result [28]. [29] extends this analysis to a broader class of evolutionary algorithms.

CHAPTER 2

Terminology, Notation, and Preliminaries

The circumflex diacritic appearing in $\hat{P}$ and $\hat{S}$ (see Chapter 5) is used exclusively to denote those two sets and has no broader meaning. Other notational conventions are presented in the appropriate subsections below.

2.1 Sets

This work considers sets of several kinds of objects. Items surrounded by curly braces ($\{\ \}$) always indicate the specification of a set's content, either using set-builder notation $\{x|P(x)\}$, or by explicit enumeration of a set's values, i.e. $\{a,\ b,\ c\}$.

2.2 Sequences

This work uses several different kinds of sequences: sequences of vertices, which sometimes represent paths (see Section 2.3), and sequences of vertex sets. In this subsection we give the sequence-related notation that applies regardless of the kind of element contained within the sequence.

A sequence may be identified by its elements, enclosed in square brackets and with individual elements separated by commas, as in $[a,\ b,\ c]$. Ellipses indicate an region of the sequence having zero or more unspecified elements, for example $[a,\ b,\ \ldots,\ z]$.

Names of paths and other vertex sequences are given as upper-case non-script letters with an overhead arrow, e.g., $\vec{Q}$. An upper-case script letter with an overbar denotes a sequence of sets. For example, $\overline{\mathcal{T}} = [\mathcal{T}_1,\ \mathcal{T}_2,\ \ldots,\ \mathcal{T}_n]$ denotes a sequence of n sets.

The elements of each sequence are numbered 1, 2, etc. As with sets, enclosing a sequence's name within vertical bars (e.g., $|\vec{X}|$) indicates the number of elements in the sequence, called its **length**.

Suppose $\vec{X}$ is some sequence. Then $\vec{X}[\![i]\!]$ denotes the i^{th} element of the sequence. $\vec{X}[\![i \ldots j]\!]$ denotes the subsequence of $\vec{X}$ from $\vec{X}[\![i]\!]$ to $\vec{X}[\![j]\!]$, inclusive.

Suppose $\vec{X}$ and $\vec{Y}$ are two sequences. Then the notation $[\vec{X}\vec{Y}]$ indicates the concatenation of the elements of the $\vec{X}$ and $\vec{Y}$:

$$[\vec{X}\vec{Y}] = [\vec{X}[\![1]\!],\ \vec{X}[\![2]\!],\ \ldots,\ \vec{X}[\![|\vec{X}|]\!],\ \vec{Y}[\![1]\!],\ \vec{Y}[\![2]\!],\ \ldots,\ \vec{Y}[\![|\vec{Y}|]\!]]$$

$[\vec{X}\vec{Y}]$ does *not* indicate a two-element-long sequence of sequences.

When the same letter is used to name both a path and an unordered vertex set, e.g. $\vec{Q}$ and $\mathcal{Q}$, the two objects implicitly contain the same vertex set.

2.3 Directed Graphs, Paths, Cycles

A directed graph $D = (\mathcal{V}, A)$, also called a **digraph**, is defined as a set $\mathcal{V}$ of vertices, and a set A of arcs.

Lower-case letters indicate individual vertices within a graph. Unordered sets of vertices are named with upper-case script letters, e.g. $\mathcal{X}$. The letter D always names an entire digraph, A always indicates the complete set of arcs in a digraph, and $\mathcal{V}$ always indicates the complete vertex set in a digraph.

The number of vertices in a graph is called the graph's **order**, and the number of arcs in a graph is called its **size**. Each arc $a \in A$ is denoted (u, v), where u and v are called the arc's **source** and **destination**, respectively.

In some cases an unnamed digraph may be identified by its vertex set and arc set. For example, $(\mathcal{V}, A)$ rather than D. We treat these two notations as interchangeable.

This work uses a shorthand notation for arcs, in that an arc's source and/or destination may be specified as a set, for example $(\mathcal{X}, y)$. This notation indicates that the arc is an unspecified member of the set of arcs whose source and/or destination vertex is a member of the indicated set. For example, "$(\mathcal{X}, y)$" is shorthand for "(u, y) for some $u \in \mathcal{X}$".

Suppose $D = (\mathcal{V}, A)$ is a digraph, and $\vec{P}$ is a vertex sequence in $\mathcal{V}$. We say that $\vec{P}$ is a **path** of D if and only if $(\vec{P}[\![1]\!], \vec{P}[\![2]\!]) \in A$, $(\vec{P}[\![2]\!], \vec{P}[\![3]\!]) \in A$, $\ldots$, $(\vec{P}[\![|\vec{P} - 1|]\!], \vec{P}[\![\ |\vec{P}|\ ]\!]) \in A$. $\vec{P}[\![1]\!]$ is called the **initial vertex** of $\vec{P}$, $\vec{P}[\![\ |\vec{P}|\ ]\!]$ is called the **terminal vertex** of $\vec{P}$, and all other vertices in $\vec{P}$ are called **internal vertices**.

Let $\vec{P}$ be any path in some digraph D. Using the notation of [1], we say that $\vec{P}$ is an $(\mathbf{x}, \mathbf{y})$**-path** if $\vec{P}[\![1]\!] = x$ and $\vec{P}[\![|\vec{P}|]\!] = y$. We say that $\vec{P}$ is an $(\mathcal{X}, \mathcal{Y})$**-path** if $\vec{P}[\![1]\!] \in \mathcal{X}$ and $\vec{P}[\![|\vec{P}|]\!] \in \mathcal{Y}$. This notation also entails $(x, \mathcal{Y})$ and $(\mathcal{X}, y)$ constructions with the obvious meaning.

A **subpath** is any contiguous subsequence within a given path. For example, in the path $\vec{P}$ with $|\vec{P}| = 3$, the subpaths are $[\vec{P}[\![1]\!]]$, $[\vec{P}[\![2]\!]]$, $[\vec{P}[\![3]\!]]$, $[\vec{P}[\![1]\!],\ \vec{P}[\![2]\!]]$, $[\vec{P}[\![2]\!],\ \vec{P}[\![3]\!]]$, and $[\vec{P}[\![1]\!],\ \vec{P}[\![2]\!],\ \vec{P}[\![3]\!]]$. Note that if $\vec{P}$ is a path in some digraph D, then every subpath of $\vec{P}$ is also a path in D.

A path is **direct** if it has a length of exactly two, and is **indirect** if it has length greater than two.

Consider a digraph $D = (\mathcal{V}, A)$. A vertex $x \in \mathcal{V}$ is a **source** of D if any only if there is no arc in A which has x as the arc's destination. Similarly, $x \in \mathcal{X}$ is a **sink** of D if any only if there is no arc in A which has x as the arc's source.

2.4 Directed Acyclic Graphs (DAG's)

A directed acyclic graph (DAG) is a digraph which contains no cyclic paths. That is a digraph D is a DAG if and only if there exists no (x, x) path in D. In this present work we assume that all DAG's are **simple graphs** (i.e., for any two vertices $x, y \in \mathcal{V}$, the graph contains at most one (x, y) arc.)

Theorem 2.4.1. *Let $D = (\mathcal{V}, A)$ be a DAG. Then $|A| \leq |\mathcal{V}|(|\mathcal{V}| - 1)/2$.*

Proof. A sketch of this proof is as follows. Readers unfamiliar with topological sorts may wish to review Chapter 4 before proceeding.

A DAG is acyclic precisely if there exists at least one topological sort of its vertices. Let $\vec{T}$ be any total ordering of $\mathcal{V}$. We now consider what set of arcs A would permit $\vec{T}$ to be a topological sort of D while maximizing $|A|$.

$\vec{T}$ is a topological sort of D if and only if for every arc $(x, y) \in A$, x appears before y in $\vec{T}$. We assert but do not prove that the maximally large arc set A_{max} meeting this requirement is constructed as follows. For each vertex $\vec{T}[\![i]\!]$ in $\vec{T}$, create an arc from $\vec{T}[\![i]\!]$ to every higher-indexed vertex in $\vec{T}$. That is,

$$A_{max} = \{\, (\vec{T}[\![i]\!], \vec{T}[\![j]\!]) \mid 1 \leq i < j \leq |\mathcal{T}| \,\}$$

Then for any $i \in [1, |\mathcal{T}|]$, A_{max} contains $|\mathcal{T}| - i$ arcs whose source vertex is $\vec{T}[\![i]\!]$. From this we have:

$$|A_{max}| = \sum_{i=1}^{|\mathcal{T}|} (|\mathcal{T}| - i) = |\mathcal{T}|(|\mathcal{T}| - 1)/2$$

Because $\mathcal{T} = \mathcal{V}$, we have $|A_{max}| = |\mathcal{V}|(|\mathcal{V}| - 1)/2$. $\square$

2.5 Contractions of Digraphs

Let $D = (\mathcal{V}, A)$ be a directed graph, and let $\mathcal{X}$ be any non-empty subset of $\mathcal{V}$. Informally, the **contraction of $\mathcal{X}$ in** D (see also [1, sect. 1.3]) is based on D, but replaces all of $\mathcal{X}$ with a single placeholder vertex x'.

More precisely, let $D_C = (\mathcal{V}_C, A'_C)$ be the contraction of the vertex set $\mathcal{X}$ to x' in D. Then D_C is constructed as follows:

$$\begin{array}{rllllllll}
\mathcal{V}_C = & (\mathcal{V} \setminus \mathcal{X}) \cup \{x'\} \\
A_C = & \{(u, v) & \mid & ((u, v) \in A) & \wedge & (u \notin \mathcal{X}) & \wedge & (v \notin \mathcal{X})\} & \cup \\
 & \{(u, x') & \mid & ((u, v) \in A) & \wedge & (u \notin \mathcal{X}) & \wedge & (v \in \mathcal{X})\} & \cup \\
 & \{(x', v) & \mid & ((u, v) \in A) & \wedge & (u \in \mathcal{X}) & \wedge & (v \notin \mathcal{X})\} & \cup \\
 & \{(x', x') & \mid & ((u, v) \in A) & \wedge & (u \in \mathcal{X}) & \wedge & (v \in \mathcal{X})\}
\end{array}$$

Algorithm 5 (Appendix B.1) computes the contraction of a digraph.

Note that the contraction of a DAG might contain a cycle. Suppose $D = (\mathcal{V}, A)$ is a DAG, and $\mathcal{X}$ is the subset of $\mathcal{V}$ to be replaced by the new vertex x' in contracted directed graph. If A contains at least one arc having both endpoints in $\mathcal{X}$, then the contracted graph contains the the arc (x', x').

2.6 Induced Subdigraphs

Let $D = (\mathcal{V}, A)$ be a digraph, and let $\mathcal{X}$ be any non-empty subset of $\mathcal{V}$. The subdigraph $D\!<\!\mathcal{X}\!>= (\mathcal{V}_S, A_S)$, read "the subdigraph of D induced by $\mathcal{X}$", is defined as follows (see also ([1, sect. 1.2]):

$$\begin{aligned} \mathcal{V}_S &= \mathcal{X} \\ A_S &= \{\, (u, v) \,|\, ((u, v) \in A) \,\wedge\, (u \in \mathcal{X}) \,\wedge\, (v \in \mathcal{X}) \,\} \end{aligned}$$

Algorithm 6 (Appendix B.2) computes induced subdigraphs.

2.7 Transitive Closure

Every DAG D has a **transitive closure**, denoted $TC(D)$.

Informally, $TC(D)$ has the same vertices as D, but a superset of the arcs in D. The added arcs indicate the existence of indirect paths between any two vertices in D. Suppose u and v are vertices in D. Then $TC(D)$ contains the arc (u, v) if any only if there exists a (u, v) path in D.

It has been shown that $TC(D)$ can be computed for a given directed graph of order n in time $O(n^{2.376})$ [1, Prop. 2.3.5].

2.8 In/Out-neighbors

For a DAG $D = (\mathcal{V}, A)$, we define the **in-neighbors of a vertex** x as the set of vertices $\{\, w \,|\, (w, x) \in A \,\}$. Similarly, the **out-neighbors of a vertex** x are the vertices

$\{\, y \,|\, (x, y) \in A \,\}$.

The concept of in-neighbors and out-neighbors may be extended to the neighbors of a set rather than of an individual vertex, as follows. The **in-neighbors of a vertex set** $\mathcal{X}$ is the set of vertices $\{\, w \,|\, ((w, \mathcal{X}) \in A) \wedge (w \notin \mathcal{X}) \,\}$. Similarly, the **out-neighbors of a vertex set** $\mathcal{X}$ is the set of vertices $\{\, y \,|\, ((\mathcal{X}, y) \in A) \wedge (y \notin \mathcal{X}) \,\}$. Note that our definitions ensure that a vertex is not both a member and an in/out-neighbor of the same set.

A vertex w is an **strictly direct in-neighbor of a vertex set** $\mathcal{B}$ if and only if w is an in-neighbor of $\mathcal{B}$ and D contains no path of the form $[w,\ \ldots,\ x,\ \ldots,\ \mathcal{B}]$, with $x \notin \mathcal{B}$. That is, there must be an arc from w to $\mathcal{B}$, but there must not be any indirect paths that begin in w, and then pass through other vertices not in $\mathcal{B}$, and then enter $\mathcal{B}$.

Similarly, a vertex y is an **strictly direct out-neighbor of a vertex set** $\mathcal{B}$ if and only if y is an out-neighbor of $\mathcal{B}$ and D no path of the form $[\mathcal{B},\ \ldots,\ x,\ \ldots,\ y]$, with $x \notin \mathcal{B}$.

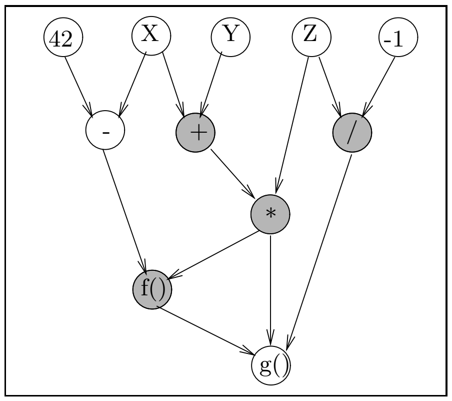
The set of strictly direct in-neighbors of a set $\mathcal{X}$ is denoted $\mathcal{N}^{\ominus}(\mathcal{X})$, and its set of strictly direct out-neighbors is denoted $\mathcal{N}^{\oplus}(\mathcal{X})$. Algorithms 7 and 8 (Appendix B.3) compute $\mathcal{N}^{\ominus}$ and $\mathcal{N}^{\oplus}$, respectively

2.9 Convexity and C-paths

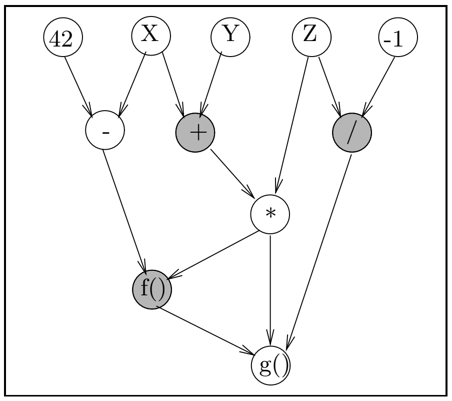
Let $D = (\mathcal{V}, A)$ be a DAG, and $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$. A directed acyclic path $\vec{Q}$ in D is a **C-path of $\mathcal{X}$** [1] if $\vec{Q}$'s initial and terminal vertices are members of $\mathcal{X}$, $\vec{Q}$ has at least one internal vertex, and and all of $\vec{Q}$'s internal vertices are members of $D \setminus \mathcal{X}$.

Let $D = (\mathcal{V}, A)$ be a DAG, and $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$. We say $\mathcal{X}$ is **a convex set of** D if and only if D contains no C-path of $\mathcal{X}$. If D does contain at least one C-path of $\mathcal{X}$ then $\mathcal{X}$ is **a concave set of** D.

[1] Identical to the notion of **S-path** in [27]. Renamed for notational purposes.



(a) The gray vertex set $\{+,\ /,\ *,\ f\}$ is a convex set of this DFG.



(b) The gray vertex set $\{+,\ /,\ f\}$ is not a convex set of this DFG, because there exists a path, $[+,\ *,\ f]$, which originates within the set, and then exits the set, and then re-enters the set.

Figure 1: Dataflow graph for computing the expression $g(\,f(42 - X, (X + Y) \times Z)\,,\,((X + Y) \times Z)\,,\,(Z \div (-1))\,)$.

Examples of convex and concave sets of a DAG are given in Figure 1a and Figure 1b, respectively.

A DAG $D = (\mathcal{V}, A)$ may have as many as $2^{|\mathcal{V}|} - 1$ convex sets. This occurs when $A = \emptyset$, in which case every non-empty subset of $\mathcal{V}$ is also convex set of D.

2.10 Threads vs. Tasks

For this work we assume that the term **thread** has the usual meaning of a mechanism for executing a sequence of program instructions. Examples of thread implementations include POSIX `pthreads`, the `std::thread` class provided by the C++11 standard library, and low-level portions of the Thread Building Blocks library.

This present work does not strongly distinguish between threads and tasks, as both are units of execution which can be created, used to execute a specified subroutine, and then be deleted.

The only difference in this work between threads and tasks is one of connotation, reinforced by the terminology of the Thread Building Blocks library used in this work's proof of concept implementation (Chapter 8). In some parallel programming environments (including the Thread Building Blocks), a thread is a long-lived object which executes zero or more tasks over its lifetime. Within that framework, this work's optimization technique concerns the assignment of program instructions to tasks, not threads.

2.11 Algorithm Pseudocode

This work contains numerous algorithms presented as pseudocode. For readability, common mathematical notation is used when possible. For example, $\mathcal{X} \leftarrow \emptyset$ denotes assigning the vertex set identifier $\mathcal{X}$ to the value empty-set. To clarify the semantics and running time of a given line of pseudocode, supplemental text is often provided on the right-hand side of the algorithm's text.

For statements other than loop-control statements, both the name of the primitive operation or subroutine, and an asymptotic running time may be provided. For example: This indicates that each activation of this line requires running time $O(|\mathcal{V}|)$.

```
1: $\mathcal{X} \leftarrow u$                 vset_init_by_vertex( $\mathcal{X}, u$ )    $O(|\mathcal{V}|)$
```

For pseudocode loop-control lines such as the following: the $O(|\mathcal{V}|)$ indicates the running

```
1: for all $u \in \mathcal{X}$ do          vset_iterate( $\mathcal{X}$ )          $O(|\mathcal{V}|)$
2:     ...
3: end for
```

time required for the loop iterator to obtain all $|\mathcal{X}|$ members of $\mathcal{X}$ for use in the loop. For pseudocode statements appearing within the body of a loop, a specified running time denotes the execution of that statement during a single iteration of the loop.

In some cases, one line of algorithm pseudocode may imply multiple operations, each of which has a running time that must be considered. In such cases, one of the underlying operations is presented on the same line as the pseudocode, and the remaining underlying operations are listed on subsequent lines, as in this example:

```
1: if $\hat{P}(y) \cap \hat{S}(\mathcal{X}) \subseteq \mathcal{X}$ then    tc_vtx_pred( $TC(D), y$ )     $O(|\mathcal{V}|)$
                                          tc_vset_succ( $TC(D), \mathcal{X}$ )  $O(|\mathcal{V}|^2)$
                                          vset_is_subset( $\ldots$ )      $O(|\mathcal{V}|)$
2:     ...
3: end if
```

When a single line of pseudocode indicates multiple operations, in some cases for brevity only the operation with the dominant asymptotic running time is provided.

CHAPTER 3

Motivation and Problem Statement

The primary motivation for this work is the problem of efficiently finding an adaptation of a program for task-parallel execution which approximately minimizes its running time.

Section 3.1 provides a brief overview of the automatic task parallelization problem.

Section 3.2 describes several execution models which motivate this work.

Section 3.3 provides a complete description of the process used by this present research to convert one or more convex sets to a task-parallel computer program with a measured average running-time. Of particular interest is Subsection 3.3.1.2, which discusses the need for parallel tasks to be convex sets. This provides the fundamental connection between this thesis' focus on convex sets and the motivating problem of parallel tasks.

Section 3.4 presents the formalization of an optimization problem for solving automatic task parallelization. In this problem, the solution space is all non-overlapping convex sets of a dataflow graph. This present research is a step towards solving this problem, but does not fully solve this optimization problem.

Section 3.5 presents a special case of the full optimization problem discussed in Section 3.4. In the special case, solutions are restricted to containing just one convex set. A solution to this problem is the focus of this present work.

Section 3.6 discusses one approach that may be used to solve the full task-set optimization problem using an algorithm which solves only the single task-set optimization problem.

3.1 Automatic Task Parallelization

In **task-parallel computing**, fragments of a computer program's code are grouped together into units of work called tasks, which at certain points in the program's execution may be executed in parallel with other running parts of the program.

A parallel program's running time can be significantly affected by the particular mapping of program instructions to tasks. **Automatic task parallelization** is the problem of using machine learning to identify a grouping of program instructions into tasks such that the running time of the resulting program is approximately minimized.

When automating task parallelization, it is convenient to represent the logic of the source program as a **dataflow graph (DFG)** (see Figure 4). A DFG is a directed acyclic graph (Section 2.3) in which each vertex represents either a program input source, a program output destination, or basic program operation. Each arc (u, v) in the DFG indicates that the data provided by vertex u is required by vertex v. Each task is defined as a particular subset of that DFG's vertices. An example of mapping each vertex in a DFG to some task or to no task at all can be seen in Figure 5 (pages 88-88). In that figure, all vertices contained within the same dash-outlined region belong to the same task.

A DFG, paired with a collection of tasks defined over that DFG, may then be translated into a standard parallel executable program.

3.2 Execution Models

For this work we consider two execution models. The first is an intermediate execution model for task-parallel DFG's. This first execution model is used to reason about task-parallel optimization in graph-theoretical terms. The second is a more concrete target execution model, in which DFG's are translated into compiled C++ code for actual execution. This second execution model is used in this work's proof of concept (see Chapter 8).

The theorems and algorithms developed within this thesis are concerned primarily with this intermediate model. The translation of a DFG into this C++ code is not the focus of this thesis. Both models are described below.

3.2.1 Task-parallel DFG Execution Model

3.2.1.1 Overview

In the task-parallel DFG execution model, the program is executed by evaluating each vertex in the given DFG precisely once per activation of that DFG. The inputs to the program are given by the source vertices (see Section 2.3) of the DFG. When a DFG is activated, the source vertices' values have already been set by whichever code or entity activated the DFG. A DFG's outputs are indicated by the values supplied to the DFG's sink vertices (see Section 2.3) during the DFG's execution.

We assume a **strict execution model**, in which a vertex may begin execution any time after all of its input data are available. An input datum is available immediately if it is a program input or a constant value. Otherwise, an input datum is available as soon as the vertex producing it has completed execution. Parallel execution of a DFG is possible precisely when all input data are available for two or more vertices that have not yet completed execution.

When a DFG is executed within a thread, the DFG's vertices are executed in some particular total order, called a **thread-local schedule**. For this work, we consider a thread-local schedule to be valid if and only if it orders the vertices in a sequence compatible with the assumptions of a strict execution model. In other words, a thread-local schedule is valid if any only if it is a topological sort of the DFG (see Chapter 4).

A **multi-threaded schedule** is set of thread-local schedules for all threads in the program. A two-threaded program may use DFG's which admit only one thread-local schedule per thread, and yet have a multiple multi-threaded schedules. This arises because we make no assumption about the relative paces at which two threads execute the vertices of their currently respective DFG's. In the execution model assumed by this work, a task-parallel DFG executes to completion if and only if each individual task executes to completion.

3.2.1.2 Computational Vertices

This model does not limit the kinds of operations that may be performed by a given vertex. However, some assumptions are made about all vertices.

- A vertex's output is assumed to be purely a function of its inputs.
- A vertex can only influence the output of the program through the particular values it produces on its output arcs.
- Once a vertex begins execution, it runs to completion within a finite amount of time.

3.2.1.3 Threading

For this work we assume a two-level hierarchical threading model, with one parent thread and zero or more child threads. Each thread may execute up to one DFG at a time.

The parent thread's DFG may contain `SPAWN(`D`)` and `WAIT(`D`)` vertices, where D is the name of some other DFG to be executed within a child thread. Child DFG's may not contain `SPAWN` and `WAIT` vertices. This limitation is imposed only to constrain the size of the search space treated by this thesis.

The `SPAWN` and `WAIT` vertices provide both synchronization and data transfer in the multi-threaded environment.

A `SPAWN(`D`)` vertex creates a new task in which the DFG D executes to completion. The `SPAWN(`D`)` vertex does not necessarily create a new thread; instead it creates a task which is to be executed at sometime in the future by an otherwise idle thread. For each arc in the parent DFG whose destination is a `SPAWN` vertex, the data carried by that arc are treated as external inputs to DFG D.

A `WAIT(D)` vertex can begin execution at any time, but will not complete its execution until the entire DFG D has completed its execution. For each arc in the parent DFG whose source is a `SPAWN` vertex, the data carried by that arc are treated as external outputs of the DFG D.

For each child DFG D, we assume a one-to-one pairing of `SPAWN(`D`)` and `WAIT(`D`)` vertices in the parent DFG.

3.2.1.4 Task-parallel DFG Specification

We define a **task** as any non-empty subset of vertices in a DFG. For a given DFG D, we define a **task set** $P = \{\mathcal{T}_1 \ldots \mathcal{T}_n\}$ as a set of tasks of D.

For this work's purposes, we assume that within a given task set P, all tasks are pairwise disjoint. That is, $\forall i \neq j, T_i \cap T_j = \emptyset$. [1] One example of a task set may be seen in Figure 5a (page 88). In that figure, the DFG has eleven vertices, and the dashed regions collectively indicate a task set containing six tasks: {{`randomize1`, `quicksort2`}, {`randomize3`, `quicksort4`}, {`randomize6`, `quicksort7`}, {`quicksort9`}, {`mergesort5`, `mergesort10`}, {`mergesort11`}}.

Let D be a DFG, and let P be a task set of D. As described below (Section 3.3), the translation from (D, P) to a system-native executable program is a deterministic function of only D and P. This present work therefore considers the pair (D, P) to fully specify a task-parallel program in its optimization problems.

[1]To accommodate a machine-learning process in which each task is independently evolved, we allow the tasks in P to overlap, and apply an overlap-elimination algorithm (Algorithm 20) as needed.

3.2.2 C++ Program Execution Model

For this work, the target is C++ source code, using the Thread Building Blocks library, compiled and executed on a computer running the Linux operating system.

Each DFG (the parent and each child-task DFG) is represented as a C++ function. Each input arc for a DFG is represented by an input parameter to the function, and each output arc is represented by an output parameter to the function.

Each vertex in a DFG is translated to a pre-defined group of one or more C++ statements. The C++ renditions of a DFG's vertices are ordered within the C++ function in a manner consistent with some topological sort of the DFG.

The C++ program has a predefined `main` function, which calls the C++ function representing the top-level DFG. The top-level DFG uses Thread Building Block library calls to spawn each child DFG's function as a parallel task, and to later block on that task until it completes.

3.3 Translation Chain: From Multiple Convex Sets to Task-Parallel Executables

Translating the pair $(D, P = \{\mathcal{T}_1,\ \mathcal{T}_2,\ \ldots,\ \mathcal{T}_n\})$ into a measured average running time involves several steps, described in detail below.

1. Translate the pair $(D, P = \{\mathcal{T}_1,\ \mathcal{T}_2,\ \ldots,\ \mathcal{T}_n\})$ into an equivalent collection of DFG's, $\{D_{parent},\ D_{child\_1},\ D_{child\_2},\ \ldots,\ D_{child\_m}\}$, with $m \leq n$. (See Subsection 3.3.1.)

2. Translate the DFG's $\{D_{parent},\ D_{child\_1},\ D_{child\_2},\ \ldots,\ D_{child\_m}\}$ into C++ source code, and use a system-supplied C++ compiler to create an executable program. (See Subsection 3.3.2.)

3. The executable program is repeatedly run a fixed number of times to obtain an

average running time. (See Subsection 3.3.3.)

We show in Subsection 3.3.1 that certain limitations exist regarding which task sets can be translated into DFG sets. The existence of these limitations provides the structure of the search space over which the optimization problems of interest (Sections 3.4 and 3.5) search.

3.3.1 From Task-set to DFG Set

The approach studied in this thesis to translate a pair $(D, P = \{\mathcal{T}_1,\ \mathcal{T}_2,\ \ldots,\ \mathcal{T}_n\})$ into an equivalent collection of DFG's $\{D_{parent},\ D_{child\_1},\ D_{child\_2},\ \ldots,\ D_{child\_m}\}$ is given below.

We begin, however, with describing one obvious approach to such translation (Subsection 3.3.1.1), and examine the problems to which that approach leads (Subsections 3.3.1.2 and 3.3.1.3). Those issues motivate a corrected translation algorithm, described in Subsection 3.3.1.4.

3.3.1.1 Naïve Translation Algorithm

One obvious translation of $(D, P = \{\mathcal{T}_1,\ \mathcal{T}_2,\ \ldots,\ \mathcal{T}_n\})$ into an equivalent collection of DFG's $\{D_{parent},\ D_{child\_1},\ D_{child\_2},\ \ldots,\ D_{child\_m}\}$ is as follows:

- Each child task $\mathcal{T}_i$ is formed into a new DFG by simply using computing the subgraph of D induced by $\mathcal{T}_i$ (see Section 2.6). That is, $D_{child\_i} = D{<}\mathcal{T}_i{>}$.
- The parent DFG D_{parent} is formed by replacing each task D_{child_i} with a pair of vertices `SPAWN(`D_{child_i}`)` and `WAIT(`D_{child_i}`)` , and an implicit arc (`SPAWN(`D_{child_i}`)`,`WAIT(`D_{child_i}`)`) as described in Subsection 3.2.1.3.

3.3.1.2 Necessity of Child-task Convexity

Consider the DFG and task set pair $(D, P = \{\mathcal{T}_1,\ \mathcal{T}_2,\ \ldots,\ \mathcal{T}_n\})$. If using the naïve translation technique described above, every task $\mathcal{T}_i$ must be a convex set (see Section 2.9) of the D. The reason is as follows.

Recall that the execution of a DFG in the task-parallel DFG execution model terminates if and only if the DFG is acyclic (Subsection 3.2.1.1).

Suppose D^i_{parent} is the graph which results from replacing, in D, $\mathcal{V}_i$ with the vertices $\texttt{SPAWN(}\mathcal{T}_i\texttt{)}$ and $\texttt{WAIT(}\mathcal{T}_i\texttt{)}$. Because D is a DFG, it is by definition acyclic. However, D^i_{parent} will contain a cycle if and only if $\mathcal{T}_i$ is not a convex set of D, for reasons explained below. If D^i_{parent} contains a cycle, it cannot be executed to completion in our execution model. We therefore require that every task $\mathcal{T}_i$ is a convex set of D.

$\mathcal{T}_i$ being a *concave* set of D implies convexity in D^i_{parent} as follows. By the definition of convex sets, $\mathcal{T}_i$ is concave if and only if there exists at least one vertex $v \notin \mathcal{T}_i$ such that D contains both a $(\mathcal{T}_i, v)$ path and a $(v, \mathcal{T}_i)$ path.

Now consider D^i_{parent}. Because $v \notin \mathcal{T}_i$, v resides in D^i_{parent} rather than in some child-task DFG. However, either directly or indirectly, $\mathcal{T}_i$ requires the output of v as an input to the DFG for the child task $\mathcal{T}_i$. Therefore D^i_{parent} contains a $(v, \texttt{SPAWN(}D_{child\_i}\texttt{)})$ path.

However, due to the concavity of $\mathcal{T}_i$, D also contains a $(\mathcal{T}_i, v)$ path. That is, either directly or indirectly, v requires the output of $\mathcal{T}_i$ as an input to v. Therefore D^i_{parent} contains a $(\texttt{WAIT(}D_{child\_i}\texttt{)}, v)$ path.

Finally, recall that D^i_{parent} contains an implicit arc $(\texttt{SPAWN(}D_{child\_i}\texttt{)}, \texttt{WAIT(}D_{child\_i}\texttt{)})$, to express the fact that $\texttt{SPAWN(}\mathcal{T}_i\texttt{)}$ must complete before $\texttt{WAIT(}\mathcal{T}_i\texttt{)}$.

It follows then that D^i_{parent} contains a cyclic path of the form $[v,\ \ldots,\ \texttt{SPAWN(}D_{child\_i}\texttt{)},\ \texttt{WAIT(}D_{child\_i}\texttt{)},\ \ldots,\ v]$.

3.3.1.3 Insufficiency of Child-task Convexity with Multiple-Task Sets

Consider again the DFG and task-set pair $(D, P = \{\mathcal{T}_1,\ \mathcal{T}_2,\ \ldots,\ \mathcal{T}_n\})$.

When P contains precisely one task (i.e., $n = 1$), and $\mathcal{T}_1$ is a convex set of D, the naïve translation algorithm described above (Subsection 3.3.1.1) always yields a parent DFG D_{parent} which is a convex set of D.

However, when $n > 1$, there are some DFG's D and task sets P such that the naïve algorithm yields a parent graph D_{parent} contain a cycle. This is possible even when every task in P is itself a convex set of D. An example of this is shown in Figure 2 on page 31.

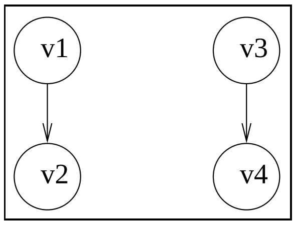
This issue arises because, while all the tasks in P are convex sets within the original DFG D, they are not necessarily convex sets within the DFG's formed as the individual tasks within P are successively replaced with $SPAWN$ and $WAIT$ vertices.

3.3.1.4 Corrected Translation Algorithm

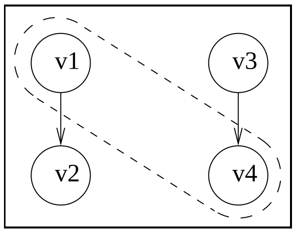
Here we consider improvements to the naïve translation algorithm (Subsection 3.3.1.1) such that the problem of induced cycles described above (Subsection 3.3.1.3) is not a problem. The corrected algorithm is given in Algorithm 1 (page 32). The algorithm's correction comes from the calls on lines 1 and 2 of the functions `eliminate_overlap` and `eliminate_cycle_induction`. See Appendix E for a discussion of how those functions ensure that top-level DFG produced by Algorithm 1 is acyclic.

3.3.2 From DFG Set to Executable Program

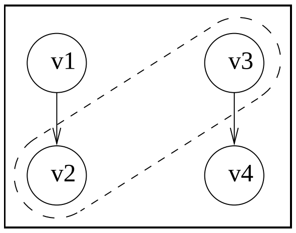
In this thesis' research, a family of dataflow graphs $\{D_{parent},\ D_{child\_1},\ D_{child\_2},\ \ldots,\ D_{child\_m}\}$ is is translated into a multi-threaded imperative computer program. This work produces a C++11 code, using the Thread Building Blocks library to provide the multi-threading functionality.



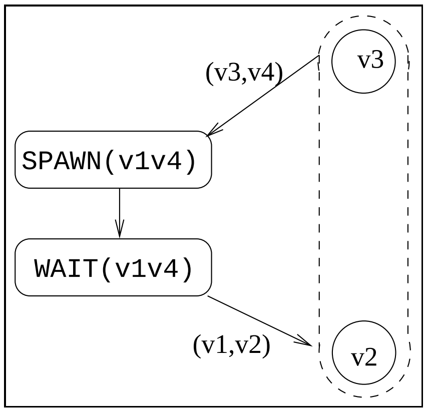
(a) The original DFG D.



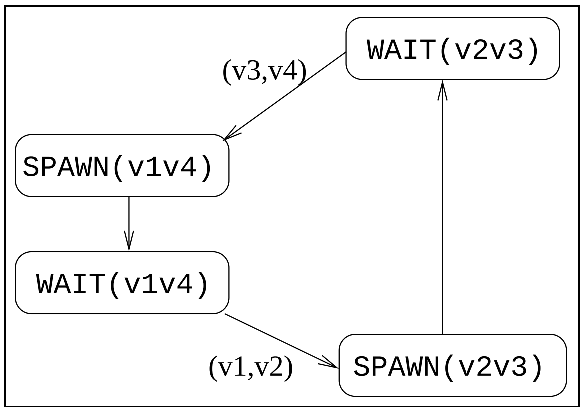
(b) D with the task $\{v1,\ v4\}$ delineated. Note that $\{v1,\ v4\}$ is a convex set of D.



(c) D with the task $\{v2,\ v3\}$ delineated. Note that $\{v2,\ v3\}$ is a convex set of D, and that $\{v1,\ v4\} \cap \{v2,\ v3\} = \emptyset$.



(d) The DFG D^1_{parent}, resulting from the contraction (see Section 2.5) of task $\{v1,\ v4\}$ in D. The task $\{v2,\ v3\}$ has not yet been contracted. Note that task $\{v2,\ v3\}$ is a convex set of D, but is not a convex set of D^1_{parent}. The arc $(v3, \texttt{SPAWN}(v1v4))$ is present because the output of $v3$ is a required input to $v4$. The arc $(\texttt{WAIT}(v1v4), v2)$ is present because the output of $v1$ is a required input to $v2$.



(e) The graph D^2_{parent}, resulting from the contraction of task $\{v2,\ v3\}$ in D^1_{parent}. D^2_{parent} contains a cycle, and is therefore not a DFG.

Figure 2: Example of how the naïve transformation (Subsection 3.3.1.1) can create a cycle in the parent DFG, even when the constituent tasks are mutually disjoint and are all convex sets of the original DFG D.

Algorithm 1 Corrected Task-Set to DFG-Set Translation

```
Function task_set_to_DFG_set_corrected : $(D,\ P) \rightarrow S$
Require:
    (R1)    $D = (\mathcal{V}, A)$ is a DAG.
    (R2)    $P = \{\mathcal{T}_1 \ldots \mathcal{T}_n\}$
    (R3)    Each $\mathcal{T}_i \in P$ is a convex set of $D$
Ensure:
    (E1)    $R = \{D_{parent},\ D_{child\_1} \ldots D_{child\_m}\}$
    (E2)    Each graph $D_i \in R$ is a DFG.
    (E3)    $R$ is functionally equivalent to $D$.

 1: $P' \leftarrow$ call eliminate_overlap( $D, P$ ) (Algorithm 20)
 2: $P'' \leftarrow$ call eliminate_cycle_induction( $D, P'$ ) (Algorithm 21)
 3: $D_{parent} \leftarrow$ clone of $D$
 4: $R \leftarrow \emptyset$
 5: for all $i \in [1, n]$ do
 6:     if $\mathcal{T}_i'' \neq \emptyset$ then
 7:         $D_{child\_i} \leftarrow D_{parent}<\mathcal{T}_i''>$
 8:         $R \leftarrow R \cup \{D_{child_i}\}$
 9:         Add vertex SPAWN($D_{child\_i}$) to $D_{parent}$.
10:         Update each $(u, dummy\_i)$ arc in $D_{parent}$ to be $(u,$ SPAWN($D_{child\_i}$)$)$.
11:         Update each $(dummy\_i, v)$ arc in $D_{parent}$ to be (WAIT($D_{child\_i}$)$, v)$.
12:         Delete $dummy\_i$ from $D_{parent}$.
13:     end if
14: end for
15: $R \leftarrow R \cup \{D_{parent}\}$
16: return $R$
```

For each DFG D_i, an equivalent C++ function is defined. Each arc in the DFG which enters $\mathcal{T}_i$ is an input parameter of the C++ function, and each arc leaving $\mathcal{T}_i$ is an output parameter of the C++ function. If a given task $\mathcal{T}_i = \emptyset$, then the translation proceeds as though $\mathcal{T}_i$ does not exist as a task.

The DFG D_{parent} is translated into a C++ function which directly executes within the C++ program's main thread. SPAWN and WAIT vertices are directly mapped to equivalent Thread Building Blocks constructs.

Within this present work, each DFG vertex maps to an equivalent block of C++ code. Some topological sort of the DFG's vertices is computed, and the C++ renditions of the DFG vertices appear in the C++ function according to that sort.

DFG's sometimes have multiple topological sorts, and it is possible that different orderings of the translated DFG vertices within the C++ function lead to different running times for that function. The freedom to choose among different topological sorts could be a reasonable component of any optimization algorithm which seeks to minimize running time. The careful selection of this topological sort lies outside the scope of this present research.

3.3.3 Obtaining Program Running Time

For this research, we assume that a program's average running time is obtained by repeatedly executing a program some number of times to obtain an average. More sophisticated approaches may be used to optimally estimate the program's mean running-time, assuming some particular underlying distribution and a desired confidence interval for the mean. However, these more sophisticated approaches were not employed, as the focus of this thesis' research lies elsewhere.

3.4 Full Task-set Optimization Problem

One approach to parallelization is to identify one or more groups of tasks in a DFG such that (1) each task can be safely run in parallel with the rest of the program, and (2) program running time is approximately minimized [2, 3, 24, 25, 30].

We define the **full task-set optimization problem** as follows.

Let $D = (\mathcal{V}, A)$ be a dataflow graph representing part or all of a program's logic. Let S be the collection of all valid task sets for D. A task set $P = \{\mathcal{T}_1 \ldots \mathcal{T}_n\}$ is considered a valid task set of D if and only if it has the following qualities:

1. Every task $\mathcal{T} \in P$ is a convex set of D. See Subsection 3.3.1.2 for the origin of this requirement.
2. For any two tasks $\mathcal{T}_i$ and $\mathcal{T}_j$ in P, with $i \neq j$, $\mathcal{T}_i \cap \mathcal{T}_j = \emptyset$. See Subsection 3.2.1.4 for the origin of this requirement.
3. Each of the DFG's in $\{D', f_{\mathcal{T}_1}, f_{\mathcal{T}_2}, \ldots, f_{\mathcal{T}_n}\}$, the contraction[2] of P in D, is acyclic. See Subsection 3.3.1.3 for the origin of this requirement.

Note that the corrected translation algorithm (Algorithm 1) accepts inputs in which only the first of those three requirements, convexity, is guaranteed. The output of Algorithm 1 is a task set which ensures all three of the above-listed requirements are satisfied. By providing this mapping, we allow a greater variety of solution-generating kernels to be used for this optimization problem, as we have a strategy for using output from kernels whose outputs do not necessarily meet the second or third requirements listed above.

Let `cost` $: S \to \mathbb{R}$ be a cost function. A typical definition of `cost` might be the average running time of the program obtained by translating a task set S into an executable program and then averaging its running time from several runs, as described in Subsection 3.3.1.4.

[2]See Section 2.5

The task-set optimization problem is to find a task set $P_{min} \in S$ which approximately minimizes `cost`.

The author is unaware of an efficient deterministic solution for this optimization problem. Solution by full evaluation of the solution space is impractical for non-trivial dataflow graphs, as the size of the solution space can be at least exponential in the number of vertices $|\mathcal{V}|$ depending on the arc set A. Convex-function optimization techniques are problematic as well, as we make no assumption that the cost function is globally convex for any obvious presentation of the search space.

3.5 Single-task Optimization Problem

We define the **single-task optimization problem** as the special form of the task-set optimization in which we impose the additional limitation that the solution space is limited to task sets containing precisely one task. By imposing this constraint on the solution space of the more general full task-set optimization problem, several other constraints in the task-set optimization problem become moot, as they're trivially satisfied for task sets containing just one task.

By permitting the task set to contain only a one task, the search space described for the full task-set optimization problem (Section 3.4) is greatly simplified, leaving only the first requirement that the task in the task set is a convex set of the DFG.

As with the task-set optimization problem, the single-task optimization problem cannot be practically solved via full evaluation of the search space for any but the smallest DFG's, and our lack of assumptions about the structure of the cost function prevents the use of optimization algorithms which assume a globally convex cost function.

3.6 Extending Single-task to Full Task-set Optimization

The research presented in this thesis was motivated by the full task-parallelization problem (Section 3.4), and yet the research focuses on a significantly restricted form of the problem (Section 3.5). We demonstrate here that an algorithm which solves the restricted optimization problem does in fact help to solve the full optimization problem.

Algorithm 3 is a per-iteration-efficient stochastic optimization algorithm for the single-task optimization problem. Consider a variation of Algorithm 3, which we'll call Z. Each iteration of Z yields some convex set of the specified DFG, but may also emit the empty set.

From this we may construct a stochastic algorithm which seeks to solve the full task-set optimization problem. A as sketch of which is given by Algorithm 2. It works as follows.

The algorithm Z evolves each convex set, but can also yield the empty set. Any empty sets which do occur are eliminated, in line 8 of the algorithm, from the current candidate-solution task set P. In this manner Algorithm 2 potentially searches task sets containing anywhere between zero and $|\mathcal{V}|$ tasks. The capacity to examine task sets of any size between 1 and $|\mathcal{V}|$ is clearly a necessity for any algorithm which seeks to solve the full task-set optimization problem.

This thesis does not attempt to show that Algorithm 2 is constructed in a manner that, given enough iterations, has a non-zero probability of searching the entire search space of all valid task sets.

Algorithm 2 is presented merely to demonstrate that Algorithm 3 is a step toward solving the full task-set optimization problem, in that it can be a building block of algorithms such as Algorithm 2 which may in fact solve the full task-set optimization problem.

Algorithm 2 Algorithm which Solves the Full Task-Set Optimization Problem

Function `optimize_full_task_set` : $(D,\ P) \rightarrow P_{best}$

Require:

(R1) $D = (\mathcal{V}, A)$ is a DAG.

(R2) *max_iter* is the maximum number of iterations to perform.

(R3) Z is a single-task-set optimization algorithm like that presented in Algorithm 3. Z is also permitted to return the empty set, which is by definition not a convex set.

Ensure:

(E1) P_{best} is the fastest-performing task set discovered in *max_iter* iterations.

```
 1: Instantiate $|\mathcal{V}|$ copies of the algorithm $Z$. Each algorithm instance $Z_i$ evolves the
    task (or empty set) $\mathcal{T}_i$, for $i \in [1, |\mathcal{V}|]$.
 2: $time\_best \leftarrow \infty$
 3: for $iter = 1, 2, \ldots, max\_iter$ do
 4:     for all $i \in [1, |\mathcal{V}|]$ do
 5:         $\mathcal{T}_i \leftarrow$ set yielded by one iteration of $Z_i$
 6:     end for
 7:     $P \leftarrow [\mathcal{T}_1,\ \mathcal{T}_2,\ \ldots,\ \mathcal{T}_{|\mathcal{V}|}]$
 8:     Delete from $P$ any members which are the empty set.
 9:     $S \leftarrow$ call task_set_to_DFG_set_corrected( $D, P$ ) (Algorithm 1)
10:     Translate $S$ to equivalent C++ code and compile it to a native executable, $E$.
11:     $time \leftarrow$ average running time from 3 executions of $E$.
12:     if $time < time\_best$ then
13:         $P_{best} \leftarrow P$
14:     end if
15: end for
16: return $P_{best}$
```

CHAPTER 4

Topological Sort Theory

4.1 Introduction

Let $D = (\mathcal{V}, A)$ be a directed graph, and let $\vec{Q}$ be a total ordering of the vertices in $\mathcal{V}$. Then $\vec{Q}$ is a **topological sort** of D if and only if for every $(\vec{Q}[\![i]\!], \vec{Q}[\![j]\!]) \in A$, $i < j$. [1]

Every directed *acyclic* graph has at least one topological sort of its vertices [1, Prop. 2.1.3]. As shown in [1, Thm. 2.1.4] and elsewhere, a topological sort for a DAG $D = (\mathcal{V}, A)$ may be generated in time $O(|\mathcal{V}| + |A|)$.

Some directed graphs have multiple topological sorts. For example, consider the directed graph $D = (\mathcal{V} = \{a,\ b,\ c\}, A = \{(a, c)\})$. Any total ordering of $\mathcal{V}$ in which a appears before c is a topological sort of D. The total orderings $[a,\ b,\ c]$ and $[a,\ c,\ b]$ are topological, but $[c,\ a,\ b]$ is not.

4.2 Results

Theorem 4.2.1. *Let $D = (\mathcal{V}, A)$ be a DAG, and let $\vec{Q}$ be any topological sort of D. Let $\vec{X}$ be any contiguous subsequence of $\vec{Q}$, so that the structure of $\vec{Q} = [\ldots,\ \vec{X},\ \ldots]$. Then $\mathcal{X}$, the set of vertices comprising $\vec{X}$, is a convex set of D.* [2]

Proof. Suppose for contradiction that $\vec{X}$ is a contiguous subsequence of $\vec{Q}$, but $\mathcal{X}$ is not a convex set of D.

Because $\mathcal{X}$ is not convex, D must contain at least one C-path of $\mathcal{X}$. Let $\vec{C}$ be such a C-path. By the definition of C-path, $\vec{C}[\![1]\!]$ and $\vec{C}[\![\ |\vec{C}|\ ]\!]$ are members of $\mathcal{X}$, and all

[1]Based on the definition given in [1, Prop. 2.1.2]. The authors of [1] recommend the term **acyclic order** for this concept.

[2]A similar conclusion may be recorded in [3, p. 57], however the author's wording makes this uncertain.

internal vertices of $\vec{C}$ are not members of $\mathcal{C}$. Note that every C-path must have at least three elements.

By the definition of a path of D, each adjacent pair of vertices in $\vec{C}$ is an arc in A. That is, $(\vec{C}[\![1]\!], \vec{C}[\![2]\!]) \in A$, $(\vec{C}[\![2]\!], \vec{C}[\![3]\!]) \in A$, …, $(\vec{C}[\![|\mathcal{C}|-1]\!], \vec{C}[\![\ |\vec{C}|\ ]\!]) \in A$.

From the definition of topological sorts, $(u, v) \in A$ implies that every topological sort of D must have the structure $[\ldots,\ u,\ \ldots,\ v,\ \ldots]$. Applying this to $\vec{Q}$ and $\vec{C}$, it follows that the structure of $\vec{Q}$ is:

$$\vec{Q} = [\ldots,\ \vec{C}[\![1]\!],\ \ldots,\ \vec{C}[\![2]\!],\ \ldots,\ \vec{C}[\![\ |\vec{C}|\ ]\!],\ \ldots]$$

Recall that $\vec{C}[\![2]\!] \notin \mathcal{X}$ and $\vec{C}[\![1]\!], \vec{C}[\![\ |\vec{C}|\ ]\!] \in \mathcal{X}$. It follows then that $\vec{X}$ is not a contiguous subsequence of $\vec{Q}$, violating our assumption to the contrary. □

Theorem 4.2.2. *Let $D = (\mathcal{V}, A)$ be a DAG, and let $\mathcal{X}$ by any convex set of D. Then there exists a total ordering $\vec{X}$ of $\mathcal{X}$, such that $\vec{Q} = [\ldots,\ \vec{X},\ \ldots]$ is a topological sort of D.*

Proof. The existence of Algorithm 9 (see Appendix B.4) demonstrates that this is true. □

Theorem 4.2.3. *Let $D = (\mathcal{V}, A)$ be a DAG, and let $\mathcal{X}$ by any convex set of D. Let $\vec{T_1} = [\vec{W},\ \vec{X}_1,\ \vec{Y}]$ be any topological sort of D such that $\vec{X}_1$ is a total ordering of $\mathcal{X}$. Then for any topological sort $\vec{X}_2$ of D<$\mathcal{X}$>, $\vec{T_2} = [\vec{W},\ \vec{X}_2,\ \vec{Y}]$ is a topological sort of D.*

Proof. Suppose for contradiction that $\vec{T_2}$ were not a topological sort of D. Then there must be some vertices $u, v \in \mathcal{V}$ such that u appears before v in $\vec{T_2}$, but $(v, u) \in A$.

We now consider where u and v might reside within $\vec{T_2}$. Because we assume that u precedes v in $\vec{T_2}$, we have six possible cases. We show that each of these six cases yields a contradiction.

Case 1: $u \in \mathcal{W}, v \in \mathcal{W}$:

Case 1 implies that u precedes v in $\vec{Y}$. Therefore no total ordering of D containing $\vec{Y}$ could be a topological sort of D. This violates our assumption that $\vec{T}_1$ is a topological sort of D.

Case 2: $u \in \mathcal{W}, v \in \mathcal{X}$*:*

Like $\vec{T}_2$, $\vec{T}_1$ has the structure $[\vec{W},\ \vec{X},\ \ldots]$. Therefore we have u preceding v not only in $\vec{T}_2$, but also in $\vec{T}_1$. This violates our assumption that $\vec{T}_1$ is a topological sort of D.

Case 3: $u \in \mathcal{W}, v \in \mathcal{Y}$*:*

The same logic applies here as for Case 2 above.

Case 4: $u \in \mathcal{X}, v \in \mathcal{X}$*:*

By the definition of $D{<}\mathcal{X}{>}$, every $(\mathcal{X}, \mathcal{X})$ arc in D is also an arc in $D{<}\mathcal{X}{>}$.

By our original assumptions the arc (v, u) is an arc in D. By Case 4's assumption, u and v are both members of $\mathcal{X}$. Therefore the arc (v, u) is also an arc in $D{<}\mathcal{X}{>}$.

However, if (v, u) is an arc in $D{<}\mathcal{X}{>}$, and u precedes v in $\vec{X}_2$, it cannot be true that $\vec{X}_2$ is a topological sort of $D{<}\mathcal{X}{>}$. This violates our assumption to the contrary.

Case 5: $u \in \mathcal{X}, v \in \mathcal{Y}$*:*

The same logic applies here as for Case 2 above.

Case 6: $u \in \mathcal{Y}, v \in \mathcal{Y}$*:*

The same logic applies here as for Case 1 above.

□

Lemma 4.2.4. *Let* $D = (\mathcal{V}, A)$ *be a DAG, let* $\emptyset \subset \mathcal{T} \subseteq \mathcal{V}$*, let* $\vec{S}$ *be any topological sort of* D*, and let* i *and* j *be any two integers with* $1 \leq i < j \leq |\mathcal{V}|$*. Then the vertex sequence* $\vec{T} = \vec{S}[\![i \ldots j]\!]$ *is a topological sort of the induced subgraph* $D{<}\mathcal{T}{>}$*.*

Proof. Suppose for contradiction that $\vec{T}$ were not a topological sort of $D{<}\mathcal{T}{>}$. Then there would be two vertices $u, v \in \mathcal{T}$ such that $\vec{T} = [\ldots,\ u,\ \ldots,\ v,\ \ldots]$ and (v, u) is an arc in $D{<}\mathcal{T}{>}$.

$\vec{T}$ is a subsequence of $\vec{S}$. Therefore because u precedes v in $\vec{T}$, u precedes v in $\vec{S}$.

Similarly, every arc in $D{<}\mathcal{T}{>}$ is also an arc in D. Therefore we have $(v, u) \in A$.

Because u precedes v in $\vec{S}$, and $(v, u) \in A$, $\vec{S}$ is not a topological sort of D. This contradicts our assumption to the contrary. $\square$

Theorem 4.2.5. *Let $D = (\mathcal{V}, A)$ be a DAG, and let $\mathcal{X}$ by any convex set of D. Let $v \in \mathcal{V} \setminus \mathcal{X}$ be any vertex such that $\mathcal{X} \cup \{v\}$ is also a convex set of D. Let $\vec{X}$ be any topological sort of $D{<}\mathcal{X}{>}$. Then there exists a topological sort of D having either the form $\vec{T} = [\ldots v, \vec{X} \ldots]$ or $\vec{T} = [\ldots \vec{X}, v \ldots]$.*

Proof. Both of the induced subgraphs $D{<}\mathcal{X}{>} = (\mathcal{X}, A_{\mathcal{X}})$ and $D{<}\mathcal{X} \cup \{v\}{>} = (\mathcal{X} \cup \{v\}, A_{\mathcal{X} \cup \{v\}})$ only have a subset of the arcs in D, and cannot contain cycles. Therefore both $D{<}\mathcal{X}{>}$ and $D{<}\mathcal{X} \cup \{v\}{>}$ are DAG's.

Clearly from the definition of induced subgraphs (Section 2.6) we have $A_{\mathcal{X}} \subseteq A_{\mathcal{X} \cup \{v\}}$. Any arc present in $D{<}\mathcal{X} \cup \{v\}{>}$ but not $D{<}\mathcal{X}{>}$ must have the form $(v, \mathcal{X})$ or $(\mathcal{X}, v)$.

It is impossible that $D{<}\mathcal{X} \cup \{v\}{>}$ has both $(v, \mathcal{X})$ arcs and $(\mathcal{X}, v)$. If both kinds of arcs were present in $D{<}\mathcal{X} \cup \{v\}{>}$, there would exist a vertex $v \in \hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})$. This would be a contradiction, as it would indicate that $\mathcal{X}$ was not in fact convex (Theorem 5.3.3).

We therefore have three possibilities for the structure of the arcs found in $A_{\mathcal{X} \cup \{v\}}$ but not in $A_{\mathcal{X}}$. For each of the three cases we demonstrate how to construct a topological sort $\vec{R}$ of $D{<}\mathcal{X} \cup \{v\}{>}$ such that v appears as either the first or last element of the $\vec{R}$. After showing this for each of the three cases below, we proceed provide the rest of the proof.

Case 1: $A_{\mathcal{X}} = A_{\mathcal{X} \cup \{v\}}$:

Every DAG has at least one topological sort [1, Prop. 2.1.3]. Let $\vec{Q}$ be any topological sort of $D{<}\mathcal{X}{>}$, and let $\vec{R} = [v,\ \vec{Q}]$.

$\vec{R}$ is a topological sort of $D{<}\mathcal{X} \cup \{v\}{>}$ if and only if $\vec{R}$ is a total ordering of $\mathcal{X} \cup \{v\}$, and for each arc $(p, q) \in A_{\mathcal{X}\cup\{v\}}$, p appears before q in $\vec{R}$.

$\vec{R}$ is clearly a total ordering of $\mathcal{X} \cup \{v\}$, because its structure is $\vec{R} = [v,\ \vec{Q}]$ and $\vec{Q}$ is a total ordering of $\mathcal{X}$.

For every (u, v) arc of the form $(\mathcal{X}, \mathcal{X})$, u appears before v in $\vec{Q}$. Because $\vec{R}$ preserves the relative ordering of each arc in $\vec{Q}$, we have that the ordering of $\vec{R}$ is consistent with every arc of the form $(\mathcal{X}, \mathcal{X})$.

In Case 1 we have $A_{\mathcal{X}} = A_{\mathcal{X}\cup\{v\}}$, and therefore we've established that the ordering $\vec{R}$ is consistent with every arc in $A_{\mathcal{X}\cup\{v\}}$. Therefore $\vec{R} = [v,\ \vec{Q}]$ is a topological sort of $D{<}\mathcal{X} \cup \{v\}{>}$ for Case 1.

Note that in Case 1, placing v anywhere in within $\vec{R}$ would have preserved $\vec{R}$'s status as a topological sort of $D{<}\mathcal{X} \cup \{v\}{>}$. However our overall goal is to show that we can always place v either immediately before or after $\mathcal{X}$ in some topological sort of D, and so we place it at the beginning of $\vec{R}$.

Case 2: $A_{\mathcal{X}\cup\{v\}} \setminus A_{\mathcal{X}}$ is a set of arcs of the form $(v, \mathcal{X})$:

The reasoning for Case 2 is essentially the same as for Case 1, except that there are additional arc in $A_{\mathcal{X}\cup\{v\}}$ that must be respected in the ordering of $\vec{R}$.

These additional arcs are all of the form $(v, \mathcal{V})$, and are clearly respected by the order $\vec{R} = [v,\ \vec{Q}]$. Therefore $\vec{R} = [v,\ \vec{Q}]$ is a topological sort of $D{<}\mathcal{X} \cup \{v\}{>}$ for Case 2.

Case 3: $A_{\mathcal{X}\cup\{v\}} \setminus A_{\mathcal{X}}$ is a set of arcs of the form $(\mathcal{X}, v)$:

The reasoning for Case 3 is similar to that of Case 2, except the additional arcs to be respected are all of the form $(\mathcal{X}, v)$, not $(v, \mathcal{X})$.

These additional arcs are clearly satisfied by the ordering $\vec{R} = [\vec{Q},\ v]$. Therefore $\vec{R} = [\vec{Q},\ v]$ is a topological sort of $D{<}\mathcal{X} \cup \{v\}{>}$ for Case 3.

We now complete our proof. Because $\mathcal{X} \cup \{v\}$ is a convex set of D, there exists some topological sort $\vec{S}$ in which the vertices $\mathcal{X} \cup \{v\}$ appear as a contiguous subsequence (Theorem 4.2.2). We therefore have $\vec{S} = \vec{E}\vec{F}\vec{G}$, where $\vec{F}$ is some ordering of $\mathcal{X} \cup \{v\}$.

From Lemma 4.2.4, we have that $\vec{F}$ is a topological sort of $D{<}\mathcal{X} \cup \{v\}{>}$.

We now have that both $\vec{F}$ and $\vec{R}$ are topological sorts of $D{<}\mathcal{X} \cup \{v\}{>}$, and $\vec{S} = [\vec{E},\ \vec{F},\ \vec{G}]$ is a topological sort of D. Applying Theorem 4.2.3 it follows that $\vec{T} = [\vec{E},\ \vec{R},\ \vec{G}]$ is also a topological sort of D. Furthermore, $\vec{R}$ has either the structure $[v,\ \vec{X}]$ or $[\vec{X},\ v]$. ☐

Theorem 4.2.6. *Let $D = (\mathcal{V}, A)$ be a DAG, and let $\mathcal{X} \subset \mathcal{V}$ by any convex set of D. Then for every $1 \leq \delta \leq |\mathcal{V} \setminus \mathcal{X}|$, D contains a convex superset of $\mathcal{X}$ having order $|\mathcal{X}| + \delta$.*

Proof. The existence of Algorithm 15 (see Appendix C.6) proves this. ☐

Theorem 4.2.7. *Let $D = (\mathcal{V}, A)$ be a DAG, and let $\mathcal{X} \subset \mathcal{V}$ by any convex set of D, with $|\mathcal{X}| > 1$. Let $1 \leq \delta < |\mathcal{X}|$. Then D contains a convex subset of $\mathcal{X}$ having order $|\mathcal{X}| - \delta$.*

Proof. The existence of Algorithm 17 (see Appendix C.8) proves this. ☐

CHAPTER 5

Predecessor and Successor Theory

This section develops a number of results regarding predecessor and successor sets, and their relationships to convex sets.

Let $D = (\mathcal{V}, A)$ be a DAG, and let $\mathcal{X}$ be any set with $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$. The **predecessor set** $\hat{P}(\mathcal{X})$ is the set of all vertices $q \in \mathcal{V}$ such that D contains a $(q, \mathcal{X})$-path. Similarly, the **successor set** $\hat{S}(\mathcal{X})$ is the set of all vertices $q \in \mathcal{V}$ such that D contains a $(\mathcal{X}, q)$-path. We define $\hat{P}(\emptyset) = \hat{S}(\emptyset) = \emptyset$.

$\hat{P}$ and $\hat{S}$ are useful concepts because they can be used to efficiently describe which vertices lie on paths contributing to a set's concavity (i.e., C-paths of that set), and provide an indication of which vertices must be added to a concave vertex set in order to achieve convexity (see Theorem 5.3.4).

$\hat{P}$ and $\hat{S}$ are closely related to the notion of a directed graph's transitive closure. A vertex x's in-neighbors in $TC(D)$ are precisely x's predecessor set in D. Similarly, a x's out-neighbors in $TC(D)$ are x's successor set in D.

5.1 Relationship between $\hat{P}/\hat{S}$ and $\mathcal{N}^{\ominus}/\mathcal{N}^{\oplus}$

Here we briefly discuss the relationship between $\hat{P}/\hat{S}$ and $\mathcal{N}^{\ominus}/\mathcal{N}^{\oplus}$. $\mathcal{N}^{\ominus}(\mathcal{X})$ can be calculated using $\hat{P}$ and $\hat{S}$:

$$\mathcal{N}^{\ominus}(\mathcal{X}) = \{v \,|\, ((v, \mathcal{X}) \in A) \wedge (v \notin \mathcal{X}) \wedge (\hat{S}(v) \cap \hat{P}(\mathcal{X}) \subset \mathcal{X})\} \tag{1}$$

As sketch for validity of Equation (1) is as follows. The terms

$$((v, \mathcal{X}) \in A) \wedge (v \notin \mathcal{X})$$

limit $\mathcal{N}^{\ominus}(\mathcal{X})$ to only those vertices which are in-neighbors (strictly direct or otherise) of $\mathcal{X}$. For an in-neighbor of X, vertex v, to be a *strictly direct* in-neighbor, there must not exist any path $[v,\ \ldots,\ u,\ \ldots,\ \mathcal{X}]$ in D, with $u \notin \mathcal{X}$. Any such vertex u is a member of $(\hat{S}(v) \cap \hat{P}(\mathcal{X}))$ but not a member of $\mathcal{X}$. This gives rise to the final clause [1] in our formula:

$$(\hat{S}(v) \cap \hat{P}(\mathcal{X}) \subset \mathcal{X})$$

By similar reasoning we may obtain:

$$\mathcal{N}^{\oplus}(\mathcal{X}) = \{v \,|\, ((\mathcal{X}, v) \in A) \wedge (v \notin \mathcal{X}) \wedge (\hat{P}(v) \cap \hat{S}(\mathcal{X}) \subset \mathcal{X})\} \tag{2}$$

5.2 Basic $\hat{P}/\hat{S}$ and $\mathcal{N}^{\ominus}/\mathcal{N}^{\oplus}$ Results

In this section we present a number of basic results pertaining to $\hat{P}$ and $\hat{S}$.

Lemma 5.2.1. *Let $D = (\mathcal{V}, A)$ by a DAG, and let $\mathcal{X}$ be any set with $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$. If $\mathcal{X}$ is a convex set of D, then $(\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})) \subseteq \mathcal{X}$.*

Proof. Proof by contradiction. Suppose D contains no C-path of $\mathcal{X}$, but $\neg((\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})) \subseteq \mathcal{X})$. By the second premise, there must exist a vertex x such that $x \in (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$ and $x \notin \mathcal{X}$. We'll show that this allows us to construct a C-path in $\mathcal{X}$, thus contradicting our first premise.

Let x be an arbitrary vertex with $x \in (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$ and $x \notin \mathcal{X}$. Because $x \in \hat{S}(\mathcal{X})$, there exists a path in in D of the form $[s_1,\ A,\ x]$, with $s_1 \in \mathcal{X}$ and with A being a subpath of zero or more vertices not in $\mathcal{X}$. Similarly, because $x \in \hat{P}(\mathcal{X})$, there exists a path in in D of the form $[x,\ B,\ s_2]$, with $s_2 \in \mathcal{X}$ and with B being a subpath of zero or more vertices not in $\mathcal{X}$. Therefore D contains a path of the form $[s_1,\ A,\ x,\ B,\ s_2]$, which is by definition a C-path in $\mathcal{X}$. $\square$

[1]Note that in the clause $(\hat{S}(v) \cap \hat{P}(\mathcal{X}) \subset \mathcal{X})$, we could equally well have used $\subseteq$ rather than $\subset$, because it is always the true in any DAG that $\hat{P}(\mathcal{X}) \neq \mathcal{X}$. I.e., $(\hat{P}(\mathcal{X}) \subset \mathcal{X}) \leftrightarrow (\hat{P}(\mathcal{X}) \subseteq \mathcal{X})$.

Lemma 5.2.2. *Let $D = (\mathcal{V}, A)$ be a DAG, let $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$, and let $\emptyset \subset \mathcal{Y} \subseteq \mathcal{V}$. Then $\hat{P}(\mathcal{X} \cup \mathcal{Y}) = \hat{P}(\mathcal{X}) \cup \hat{P}(\mathcal{Y})$.*

Proof. The proof flows directly from the definition of $\hat{P}$. First prove that $\hat{P}(\mathcal{X} \cup \mathcal{Y}) \subseteq (\hat{P}(\mathcal{X}) \cup \hat{P}(\mathcal{Y}))$. Suppose for contradiction that there exists a vertex $v \in \hat{P}(\mathcal{X} \cup \mathcal{Y})$, but $v \notin \hat{P}(\mathcal{X})$ and $v \notin \hat{P}(\mathcal{Y})$. By the first premise, there must be a directed path in D that passes through v and terminates at some vertex in $\mathcal{X}$ and/or$\mathcal{Y}$. But this contradicts the second premise that $v \notin \hat{P}(\mathcal{X})$ and $v \notin \hat{P}(\mathcal{Y})$.

Now prove that $(\hat{P}(\mathcal{X}) \cup \hat{P}(\mathcal{Y})) \subseteq \hat{P}(\mathcal{X} \cup \mathcal{Y})$, again by contradiction. Suppose there exists a vertex $v \in (\hat{P}(\mathcal{X}) \cup \hat{P}(\mathcal{Y}))$, but $v \notin \hat{P}(\mathcal{X} \cup \mathcal{Y})$. By the first premise, v must lie on some directed path in D that terminates in either $\mathcal{X}$ or$\mathcal{Y}$. But the existence of such a path contradicts the second premise. □

Lemma 5.2.3. *Let $D = (\mathcal{V}, A)$ be a DAG, let $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$, and let $\emptyset \subset \mathcal{Y} \subseteq \mathcal{V}$. Then $\hat{S}(\mathcal{X} \cup \mathcal{Y}) = (\hat{S}(\mathcal{X}) \cup \hat{S}(\mathcal{Y}))$.*

Proof. This proof is identical in structure to the proof of Lemma 5.2.2. □

Although $\hat{P}$ and $\hat{S}$ distribute over union (Lemmas 5.2.2 and 5.2.3), they do not distribute over intersection. For this reason the following two lemmas are weaker than the previous two.

Lemma 5.2.4. *Let $D = (\mathcal{V}, A)$ be a DAG, let $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$, and let $\emptyset \subset \mathcal{Y} \subseteq \mathcal{V}$. Then $\hat{S}(\mathcal{X} \cap \mathcal{Y}) \subseteq \hat{S}(\mathcal{X}) \cap \hat{S}(\mathcal{Y})$.*

Proof. Suppose for contradiction that there exists a vertex $v \in \hat{S}(\mathcal{X} \cap \mathcal{Y})$, and yet $v \notin (\hat{S}(\mathcal{X}) \cap \hat{S}(\mathcal{Y}))$.

From the premise that $v \in \hat{S}(\mathcal{X} \cap \mathcal{Y})$, there must be a vertex $u \in (\mathcal{X} \cap \mathcal{Y})$ such that $\vec{Q} = [u, \ldots, v]$ is a path in D. Because $\vec{Q}$ originates at a vertex in $\mathcal{X}$, $v \in \hat{S}(\mathcal{X})$. And because $\vec{Q}$ also originates at a vertex in $\mathcal{Y}$, $v \in \hat{S}(\mathcal{Y})$. Therefore $v \in (\hat{S}(\mathcal{X}) \cap \hat{S}(\mathcal{Y}))$, which contradicts the second premise, $v \notin (\hat{S}(\mathcal{X}) \cap \hat{S}(\mathcal{Y}))$. □

Lemma 5.2.5. *Let $D = (\mathcal{V}, A)$ be a DAG, and let $\emptyset \subseteq \mathcal{X} \subseteq \mathcal{V}$. Then $\hat{P}(\hat{P}(\mathcal{X})) \subseteq \hat{P}(\mathcal{X})$.*

Proof. By the definition of $\hat{P}$, every vertex $e \in \hat{P}(\hat{P}(\mathcal{X}))$ lies on a path of the form $\vec{Q} = [\ldots,\ e,\ \ldots,\ f]$, with $f \in \hat{P}(\mathcal{X})$.

Also by the definition of $\hat{P}$, every vertex $f \in \hat{P}(\mathcal{X})$ lies on a path of the form $\vec{R} = [\ldots,\ f,\ \ldots,\ x]$, with $x \in \mathcal{X}$.

Therefore, for every $e \in \hat{P}(\hat{P}(\mathcal{X}))$, there exists a path in D of the form $\vec{S} = [\ldots,\ e,\ \ldots,\ f,\ \ldots,\ x]$, with $f \in \hat{P}(\mathcal{X})$ and $x \in \mathcal{X}$. Therefore e is also a non-terminal vertex of some path ($\vec{S}$) terminating in $\mathcal{X}$, and thus $e \in \hat{P}(\mathcal{X})$. Because every $e \in \hat{P}(\hat{P}(\mathcal{X}))$ is also a member of $\hat{P}(\mathcal{X})$, we have $\hat{P}(\hat{P}(\mathcal{X})) \subseteq \hat{P}(\mathcal{X})$. □

Lemma 5.2.6. *Let $D = (\mathcal{V}, A)$ be a DAG, and let $\emptyset \subseteq \mathcal{X} \subseteq \mathcal{V}$. Then $\hat{S}(\hat{S}(\mathcal{X})) \subseteq \hat{S}(\mathcal{X})$.*

Proof. This proof is similar in structure to the proof for Lemma 5.2.5. □

Lemma 5.2.7. *Let $D = (\mathcal{V}, A)$ be a DAG, and let $\emptyset \subset \mathcal{X} \subseteq \mathcal{Y} \subseteq \mathcal{V}$. Then $\hat{S}(\mathcal{X}) \subseteq \hat{S}(\mathcal{Y})$.*

Proof. This proof has the same structure as that for Lemma 5.2.10. □

Lemma 5.2.8. *Let $D = (\mathcal{V}, A)$ be a DAG, let $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$, and let $\mathcal{X}$ be a convex set of D. Then $\hat{P}(\mathcal{X} \cup (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))) = \hat{P}(\mathcal{X})$.*

Proof. We derive the right-hand side of the equation from the left-hand side, as follows.

First distribute using Lemma 5.2.2:

$$\hat{P}(\mathcal{X} \cup (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$$

$$= \hat{P}(\mathcal{X}) \cup \hat{P}(\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$$

Using Lemma 5.2.1:

$$= \hat{P}(\mathcal{X}) \cup \hat{P}(\mathcal{Y}), \text{with}\, \mathcal{Y} \subseteq \mathcal{X}$$

By Lemma 5.2.10, if $\mathcal{Y} \subseteq \mathcal{X}$, then $\hat{P}(\mathcal{Y}) \subseteq \hat{P}(\mathcal{X})$. This gives us:

$$= \hat{P}(\mathcal{X}) \cup \mathcal{Z}, \text{with}\, \mathcal{Z} \subseteq \hat{P}(\mathcal{X})$$

which trivially reduces to:

$$= \hat{P}(\mathcal{X})$$

□

Lemma 5.2.9. *Let $D = (\mathcal{V}, A)$ be a DAG, let $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$, and let $\mathcal{X}$ be a convex set of D. Then $\hat{S}(\mathcal{X} \cup (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))) = \hat{S}(\mathcal{X})$.*

Proof. This proof has the same structure as the proof of Lemma 5.2.8. □

Lemma 5.2.10. *Let $D = (\mathcal{V}, A)$ be a DAG, and let $\emptyset \subset \mathcal{X} \subseteq \mathcal{Y} \subseteq \mathcal{V}$. Then $\hat{P}(\mathcal{X}) \subseteq \hat{P}(\mathcal{Y})$.*

Proof. Because $\mathcal{X} \subseteq \mathcal{Y}$, every path ending in $\mathcal{X}$ is also a path ending in $\mathcal{Y}$. Therefore the non-terminal vertices of paths ending $\mathcal{X}$ constitute a subset of the non-terminal vertices of paths ending in $\mathcal{Y}$. □

Lemma 5.2.11. *Let $D = (\mathcal{V}, A)$ by a DAG, and let $\mathcal{X}$ be any set with $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$. If $(\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})) \subseteq \mathcal{X}$, then $\mathcal{X}$ is a convex set of D.*

Proof. Proof by contradiction. Suppose that $(\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})) \subseteq \mathcal{X}$, but D *does* contain a path $\vec{Q}$ which is a C-path of $\mathcal{X}$. Let x be an arbitrary internal vertex of $\vec{Q}$. From Lemma 5.3.1, $x \in ((\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$. Because of our premise that $(\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})) \subseteq \mathcal{X}$, we have $x \in \mathcal{X}$. However, the definition of a C-path of $\mathcal{X}$ indicates that each internal vertex is not in $\mathcal{X}$. Because x cannot be both in $\mathcal{X}$ and not in $\mathcal{X}$, we have a contradiction. □

Lemma 5.2.12. *Let $D = (\mathcal{V}, A)$ be a DAG, and let $\emptyset \subseteq \mathcal{X} \subset \mathcal{V}$ be a convex set of D. Then $\mathcal{N}^{\ominus}(\mathcal{X}) \subseteq \hat{P}(\mathcal{X}) \setminus \hat{S}(\mathcal{X})$.*

Proof. $\mathcal{N}^{\ominus}(\mathcal{X}) \subseteq \hat{P}(\mathcal{X})$ is trivially true, because every vertex that has an strictly direct path to $\mathcal{X}$ also has a path to $\mathcal{X}$.

From Lemma 5.2.1, we have that a vertex v which is external to a convex set $\mathcal{X}$ cannot be in both $\hat{P}(\mathcal{X})$ and $\hat{S}(\mathcal{X})$. □

Lemma 5.2.13. *Let $D = (\mathcal{V}, A)$ be a DAG, and let $\emptyset \subseteq \mathcal{X} \subset \mathcal{V}$ be a convex set of D. Then $\mathcal{N}^{\oplus}(\mathcal{X}) \subseteq \hat{S}(\mathcal{X}) \setminus \hat{P}(\mathcal{X})$.*

Proof. This proof is similar to that of Lemma 5.2.12. □

5.3 Obtaining Convexity via Internal Path Closure

Let $D = (\mathcal{V}, A)$ be a DAG, and let $\mathcal{X}$ be an arbitrary non-empty subset of $\mathcal{V}$. We define the **internal path closure** of $\mathcal{X}$ to be the set $\mathcal{X} \cup (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$ [2].

Theorem 5.3.3 and Theorem 5.3.4 below demonstrate that all internal path closure sets are convex.

Furthermore, Lemma 5.2.1 shows that if a given set is already convex, its internal path closure is identical to the original set. Therefore stochastic algorithms may indiscriminately apply the internal path closure formula to each candidate solution, without the risk of pointlessly modifying an already valid solution.

Finally, Corollary 5.3.5 establishes that the internal path closure of some set $\mathcal{X}$ is the smallest convex set containing $\mathcal{X}$. This result is useful for search algorithms that establish

[2]This approach is used in [3, Fig. 4-7]. However, that work does not appear to offer proof for the validity of the approach.

convexity by adding vertices to a concave set, but strive to modify the original set as little as possible.

Lemma 5.3.1. *Let $D = (\mathcal{V}, A)$ by a DAG, let $\mathcal{X}$ be any set with $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$, and let $\vec{Q}$ be any C-path of $\mathcal{X}$. Then every internal vertex of $\vec{Q}$ is a member of $(\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$.*

Proof. By the definition of C-path of $\mathcal{X}$, $\vec{Q}$'s initial and terminal vertices are in $\mathcal{X}$. Because $\vec{Q}$'s initial vertex is in $\mathcal{X}$, there exists a directed path from $\mathcal{X}$ to each of $\vec{Q}$'s internal vertices. Therefore each of $\vec{Q}$'s internal vertices is a member of $\hat{S}(\mathcal{X})$. Similarly, because $\vec{Q}$'s terminal vertex is in $\mathcal{X}$, there exists a directed path from each internal vertex of $\vec{Q}$ to $\mathcal{X}$. Therefore each of $\vec{Q}$'s internal vertices is a member of $\hat{P}(\mathcal{X})$. □

Theorem 5.3.2. *Let $D = (\mathcal{V}, A)$ be a DAG, let $\emptyset \subseteq \mathcal{X} \subset \mathcal{V}$ be a convex set of D, and let $v \in (\mathcal{V} \setminus \mathcal{X})$, with $v \notin \hat{P}(\mathcal{X})$ and $v \notin \hat{S}(\mathcal{X})$. Then the set $\mathcal{Y} = \mathcal{X} \cup \{v\}$ is a convex set of D.*

Proof. Suppose for contradiction that $\mathcal{Y}$ is not a convex set of D. Then there must be some C-path of D, $\vec{C} = [a, \ \ldots, \ w, \ \ldots, \ z]$, with $a, c \in \mathcal{Y}$ and $w \notin \mathcal{Y}$.

Because $\mathcal{Y} = \mathcal{X} \cup \{v\}$, we consider four possible variations of $\vec{C}$: $a \in \mathcal{X}$ or $a = v$, and $z \in \mathcal{X}$ or $z = v$. We show below that each of the four possible variations leads to a contradiction. Note that these four cases are not necessarily mutually exclusive, however the following proofs do not depend their mutual exclusion.

Case 1: $a \in \mathcal{X}$, $z \in \mathcal{X}$:

Because $a \in \mathcal{X}$, and $[a, \ \ldots, \ w]$ is a path in D, we have $w \in \hat{S}(\mathcal{X})$. Similarly, because $z \in \mathcal{X}$, and $[w, \ \ldots, \ z]$ is a path in D, we have $w \in \hat{P}(\mathcal{X}))$. Therefore $w \in (\hat{P}(X) \cap \hat{S}(X))$. From Lemma 5.2.1, we have $(\hat{P}(X) \cap \hat{S}(X)) \subseteq \mathcal{X}$. From the transitivity of the subset relation, this gives us $w \in \mathcal{X}$, which violates our assumption to the contrary.

Case 2: $a = v$, $z = v$:

Case 2 implies that $[v, \ldots, w, \ldots, v]$ is a path in D, which violates the assumption that D is acyclic.

Case 3: $a \in \mathcal{X}$, $z = v$:

Case 3 implies that $[\mathcal{X}, \ldots, w, \ldots, v]$ is a path in D, which violates our assumption that $v \notin \hat{S}(X)$.

Case 4: $a = v$, $z \in \mathcal{X}$:

Case 4 implies that $[v, \ldots, w, \ldots, \mathcal{X}]$ is a path in D, which violates our assumption that $v \notin \hat{P}(X)$.

□

Theorem 5.3.3. *Let $D = (\mathcal{V}, A)$ be a DAG, and let $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$. Then ($\mathcal{X}$ is a convex set of D) $\iff$ $((\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})) \subseteq \mathcal{X})$.*

Proof. By the definition of convexity, $\mathcal{X}$ is a convex set of D if and only if D contains no C-path of $\mathcal{X}$. For convenience we'll use that fact to restate the theorem to be proven: D contains no C-path of $\mathcal{X}$ if and only if $((\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})) \subseteq \mathcal{X})$.

First, prove that (D contains no C-path of $\mathcal{X}$) $\rightarrow$ $((\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})) \subseteq \mathcal{X})$. This directly follows from Lemma 5.2.1.

Second, prove that $((\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})) \subseteq \mathcal{X}) \rightarrow$ (D contains no C-path of $\mathcal{X}$). This directly follows from Lemma 5.2.11. □

Theorem 5.3.4. *Let $D = (\mathcal{V}, A)$ be a DAG, and let $\mathcal{X}$ be any set with $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$, and let $\mathcal{T} = \hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})$. Then $\mathcal{X} \cup \mathcal{T}$ is a convex set of D.*

Proof. Suppose for contradiction that $\mathcal{X} \cup \mathcal{T}$ is concave in D. Then there exists in D a C-path of $\mathcal{X} \cup \mathcal{T}$ in D. Let $\vec{C} = [x, \ldots, y, \ldots, z]$ be such a path, with with $x, z \in \mathcal{X} \cup \mathcal{T}$ and $y \notin \mathcal{X} \cup \mathcal{T}$.

We'll use case analysis to show that four possible forms exist for $\vec{C}$, and none of those forms can exist.

Case 1: $x \in \mathcal{X}$, $z \in \mathcal{X}$*:*

Because $[x,\ \ldots,\ y]$ is a path in D, $y \in \hat{S}(X)$. Because $[\ldots,\ z]$ is a path in D, $y \in \hat{P}(X)$. Therefore $y \in (\hat{P}(X) \cap \hat{S}(X))$, i.e., $y \in \mathcal{T}$. This contradicts our assumption that $y \notin \mathcal{X} \cup \mathcal{T}$.

Case 2: $x \in \mathcal{X}$, $z \in \mathcal{T}$*:*

Because $[x,\ \ldots,\ y]$ is a path in D, $y \in \hat{S}(X)$.

Because $[y,\ \ldots,\ z]$ is a path in D, $y \in \hat{P}(T)$, i.e., $y \in \hat{P}(\hat{P}(X) \cap \hat{S}(X))$. Because $y \in \hat{P}(\hat{P}(X) \cap \hat{S}(X))$, it trivially follows that $y \in \hat{P}(\hat{P}(X))$. By Lemma 5.2.5, $\hat{P}(\hat{P}(X)) \subseteq \hat{P}(X)$, and therefore $y \in \hat{P}(X)$.

Having established then that $y \in \hat{S}(X)$ and $y \in \hat{P}(X)$, it follows that $y \in (\hat{P}(X) \cap \hat{S}(X))$, i.e., $y \in \mathcal{T}$. This contradicts our assumption that $y \notin \mathcal{T}$.

Case 3: $x \in \mathcal{T}$, $z \in \mathcal{X}$*:*

This case is false for reasoning similar to that used in Case 2. However, this case's logic requires Lemma 5.2.6, not Lemma 5.2.5.

Case 4: $x \in \mathcal{T}$, $z \in \mathcal{T}$*:*

Because $[x,\ \ldots,\ y]$ is a path in D, we have $y \in \hat{S}(\mathcal{T})$, i.e., $y \in \hat{S}(\hat{P}(X) \cap \hat{S}(X))$. Because $y \in \hat{S}(\hat{P}(X) \cap \hat{S}(X))$, it's trivially true that $y \in \hat{S}(\hat{S}(X))$. By Lemma 5.2.6, we have $\hat{S}(\hat{S}(X)) \subseteq \hat{S}(X)$, and so $y \in \hat{S}(X)$.

Similarly, because $[y,\ \ldots,\ z]$ is a path in D, we have $y \in \hat{P}(\mathcal{T})$, i.e., $y \in \hat{P}(\hat{P}(X) \cap \hat{S}(X))$. Because $y \in \hat{P}(\hat{P}(X) \cap \hat{S}(X))$, it's trivially true that $y \in \hat{P}(\hat{P}(X))$. By Lemma 5.2.5, we have $\hat{P}(\hat{S}(X)) \subseteq \hat{P}(X)$, and so $y \in \hat{P}(X)$.

Having established then that $y \in \hat{S}(X)$ and $y \in \hat{P}(X)$, it follows that $y \in (\hat{P}(X) \cap \hat{S}(X))$, i.e., $y \in \mathcal{T}$. This contradicts our assumption that $y \notin \mathcal{T}$.

□

Corollary 5.3.5. *Let $D = (\mathcal{V}, A)$ be a DAG, let $\mathcal{X}$ be any set with $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$, and let $\mathcal{Y} = \mathcal{X} \cup (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$. Then $\mathcal{Y}$ is the uniquely smallest improper superset of X that is also a convex set of D.*

Proof. Suppose for contradiction that there exists another set $\mathcal{Z} \neq \mathcal{Y}$ that is also a convex improper superset of $\mathcal{X}$, and yet is no larger than $\mathcal{Y}$. We'll show that $\mathcal{Z}$ cannot exist.

Because $\mathcal{Z} \neq \mathcal{Y}$ and $|\mathcal{Z}| \leq |\mathcal{Y}|$, there must be some vertex in $v \in \mathcal{Y}$ that is not present in $\mathcal{Z}$.

Because $v \in \mathcal{Y}$, and by the definition of $\mathcal{Y}$, we have $v \in \mathcal{X}$ and/or $v \in (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$. We'll consider those two cases separately, and show that both lead to contradictions.

Suppose for contradiction that $v \in \mathcal{X}$. Then v is a vertex in $\mathcal{X}$ that is absent from $\mathcal{Z}$. This contradicts our assumption that $\mathcal{Z}$ is an improper superset of $\mathcal{X}$.

Suppose for contradiction the second case: $v \in (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$. By Lemmas 5.2.10 and 5.2.7, $(\mathcal{X} \subseteq \mathcal{Z}) \implies \hat{P}(\mathcal{X}) \subseteq \hat{P}(\mathcal{Z})$ and $(\mathcal{X} \subseteq \mathcal{Z}) \implies \hat{S}(\mathcal{X}) \subseteq \hat{S}(\mathcal{Z})$. It follows then that not only is $v \in (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$, as assumed for this case, but also that $v \in (\hat{P}(\mathcal{Z}) \cap \hat{S}(\mathcal{Z}))$.

Because $v \in (\hat{P}(\mathcal{Z}) \cap \hat{S}(\mathcal{Z}))$, and $v \notin \mathcal{Z}$ (by our definition of v), it is *not* the case that $(\hat{P}(\mathcal{Z}) \cap \hat{S}(\mathcal{Z})) \subseteq \mathcal{Z})$. According to Theorem 5.3.3, this implies that $\mathcal{Z}$ is not a convex set of D, which contradicts our definition of $\mathcal{Z}$. □

5.4 Modifying Convex Sets with $\hat{P}/\hat{S}$ and $\mathcal{N}^{\ominus}/\mathcal{N}^{\oplus}$

In this subsection we develop a theoretical basis for algorithms which seek to grow or shrink an existing convex set by adding or deleting vertices.

Theorem 5.4.1. *Let $D = (\mathcal{V}, A)$ be a DAG, let $\emptyset \subseteq \mathcal{X} \subset \mathcal{V}$ be a convex set of D and let $v \in (\mathcal{V} \setminus \mathcal{X})$ with $v \in \hat{P}(\mathcal{X})$. Then $\mathcal{X} \cup \{v\}$ is a convex set of D if and only if $v \in \mathcal{N}^{\ominus}(\mathcal{V})$.*

Note: By assumption $v \notin \mathcal{X}$, and $v \in \hat{P}(\mathcal{X})$. It follows then from Lemma 5.2.1 that $v \notin \hat{S}(\mathcal{X})$.

Informally, this Lemma states the following. Suppose $\mathcal{X}$ is a convex set of D, and $v \in \hat{P}(\mathcal{X})$ is not in $\mathcal{X}$. Then a necessary and sufficient criterion for the convexity in D of the set $\mathcal{Y} = \mathcal{X} \cup \{v\}$ is that the only paths from v to X are direct. That is, $(v, \mathcal{X})$ is an arc in A, but that *indirect* path goes from the v to $\mathcal{X}$. A path from v to $\mathcal{X}$ is indirect if and only if it contains one or more intermediate vertices.

Proof. We begin by showing that if an indirect path from v to $\mathcal{V}$ exists, then $\mathcal{Y}$ is concave. Suppose without loss of generality that $\vec{P} = [v,\ \ldots,\ w,\ \ldots,\ X]$ is a path in D. By the definition of convexity, a set is a convex set of D if and only if no C-path of that set exists in D. Then $\vec{P}$ is a C-path of the set $\mathcal{Y}$, and therefore $\mathcal{Y}$ is not a convex set of D.

We now show that when the only paths from v to $\mathcal{V}$ are direct, $\mathcal{Y}$ is convex. We do this using Lemma 5.2.11, which states that a set $\mathcal{Y}$ is a convex set of D if (and only if) $(\hat{P}(\mathcal{Y}) \cap \hat{S}(\mathcal{Y})) \subseteq \mathcal{Y})$. We proceed by developing the left-hand size of the relation, $(\hat{P}(\mathcal{Y}) \cap \hat{S}(\mathcal{Y}))$ to obtain a result that is trivially a subset of $\mathcal{Y}$.

We begin with the left-hand-side expression $\hat{P}(\mathcal{Y}) \cap \hat{S}(\mathcal{Y})$, and replace $\mathcal{Y}$ with its definition:

$$\hat{P}(\mathcal{Y}) \cap \hat{S}(\mathcal{Y}) \quad = \quad \hat{P}(\mathcal{X} \cup \{v\}) \cap \hat{S}(\mathcal{X} \cup \{v\}) \qquad (3)$$

Distribute $\hat{P}$ over union (Lemma 5.2.2):

$$\hat{P}(\mathcal{Y}) \cap \hat{S}(\mathcal{Y}) \quad = \quad (\hat{P}(\mathcal{X}) \cup \hat{P}(\{v\})) \cap \hat{S}(\mathcal{X} \cup \{v\})) \tag{4}$$

Distribute $\hat{S}$ over union (Lemma 5.2.3):

$$\hat{P}(\mathcal{Y}) \cap \hat{S}(\mathcal{Y}) \quad = \quad (\hat{P}(\mathcal{X}) \cup \hat{P}(\{v\})) \cap (\hat{S}(\mathcal{X}) \cup \hat{S}(\{v\})) \tag{5}$$

Apply the distributive law of sets $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$, with $A = \hat{P}(\mathcal{X}) \cup \hat{P}(\{v\})$, $B = \hat{S}(\mathcal{X})$, and $C = \hat{S}(\{v\})$:

$$\begin{aligned}\hat{P}(\mathcal{Y}) \cap \hat{S}(\mathcal{Y}) \quad = \quad & [\,(\hat{P}(\mathcal{X}) \cup \hat{P}(\{v\})) \;\cap\; \hat{S}(\mathcal{X}) \;\;] \;\cup \\ & [\,(\hat{P}(\mathcal{X}) \cup \hat{P}(\{v\})) \;\cap\; \hat{S}(\{v\}) \;\;]\end{aligned} \tag{6}$$

Again apply the distributive law of sets $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ to the left side of (6), with $A = \hat{S}(\mathcal{X})$, $B = \hat{P}(\mathcal{X})$, and $C = \hat{P}(\{v\})$:

$$\begin{aligned}\hat{P}(\mathcal{Y}) \cap \hat{S}(\mathcal{Y}) \quad = \quad & [\,[\,\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})\,] \;\cup\; [\,\hat{P}(\{v\}) \cap \hat{S}(\mathcal{X})\,] \;\;] \;\cup \\ & [\,[\,\hat{P}(\mathcal{X}) \cup \hat{P}(\{v\})\,] \;\cap\; \hat{S}(\{v\}) \;\;]\end{aligned} \tag{7}$$

Now apply the distributive law of sets $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ to the right side of (7), with $A = \hat{S}(\{v\})$, $B = \hat{P}(\mathcal{X})$, and $C = \hat{P}(\{v\})$:

$$\begin{aligned}\hat{P}(\mathcal{Y}) \cap \hat{S}(\mathcal{Y}) \quad = \quad & [\,[\,\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})\,] \;\cup\; [\,\hat{P}(\{v\}) \cap \hat{S}(\mathcal{X})\,] \;\;] \;\cup \\ & [\,[\,\hat{S}(\{v\}) \cap (\hat{P}(\mathcal{X}))\,] \;\cup\; [\,\hat{S}(\{v\}) \cap \hat{P}(\{v\})\,] \;\;]\end{aligned} \tag{8}$$

We now have an expression of the form $[A \cup B] \cup [C \cup D]$. By the associativity rules of set union, we may eliminate some grouping:

$$\begin{aligned}\hat{P}(\mathcal{Y}) \cap \hat{S}(\mathcal{Y}) = & \\ & \text{(9a)} \quad (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})) \quad \cup \\ & \text{(9b)} \quad (\hat{P}(\{v\}) \cap \hat{S}(\mathcal{X})) \quad \cup \\ & \text{(9c)} \quad (\hat{S}(\{v\}) \cap \hat{P}(\mathcal{X})) \quad \cup \\ & \text{(9d)} \quad (\hat{S}(\{v\}) \cap \hat{P}(\{v\}))\end{aligned} \tag{9}$$

We now show that each of the four major expressions (9a) to (9d) in Eq. (9) is, individually, a subset of $\mathcal{Y}$. When that has been established for each of the four subexpressions, we can conclude that the conjunction of those four expressions is itself a subset of $\mathcal{Y}$, which completes our proof.

Case 9a: $\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})$*:*

Because $\mathcal{X}$ is convex, Lemma 5.2.1 indicates that $\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}) \subseteq \mathcal{X}$. Therefore we have $\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}) \subseteq \mathcal{X}$.

We also have $\mathcal{Y} = \mathcal{X} \cup \{v\}$, and so $\mathcal{X} \subseteq \mathcal{Y}$. Therefore $\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}) \subseteq \mathcal{Y}$.

Case 9b: $\hat{P}(\{v\}) \cap \hat{S}(\mathcal{X})$*:*

This expression must be the empty set. Suppose for contradiction that there exists a vertex $q \in \hat{P}(\{v\}) \cap \hat{S}(\mathcal{X})$. Because $q \in \hat{P}(\{v\}$, there exists a path in D of the form $[q, \ldots, v]$. Because $q \in \hat{S}(\mathcal{X})$, there exists a path in D of the form $[\mathcal{X}, \ldots, q]$. Combining these paths at q, we have that D contains a path of the form $\mathcal{X} \ldots v$. This implies that $v \in \hat{S}(\mathcal{X})$. However, this theorem also has by assumption $v \in \hat{P}(\mathcal{X})$. Therefore we have $v \in (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{V}))$, and yet $v \notin \mathcal{X}$. This implies that v lies on a C-path of $\mathcal{X}$. This violates our assumption that $\mathcal{X}$ is a convex set of D.

Case 9c: $\hat{S}(\{v\}) \cap \hat{P}(\mathcal{X})$*:*

This must be the empty set. Suppose for contradiction that it was not. Then there would be a vertex q in D such that $[v, \ldots, q]$ and $[q, \ldots, \mathcal{V}]$ are both paths in D. Therefore, D would contain a path of the form $[v, \ldots, q, \ldots, \mathcal{V}]$, which violates our assumption that D contains no indirect path from v to $\mathcal{V}$.

Case 9d: $\hat{S}(\{v\}) \cap \hat{P}(\{v\})$*:*

This must be the empty set. If not, D would contain a path of the form $[v \ldots v]$, violating the assumption that D is acyclic.

□

Theorem 5.4.2. *Let $D = (\mathcal{V}, A)$ be a DAG, let $\emptyset \subseteq \mathcal{X} \subset \mathcal{V}$ be a convex set of D, and let $v \in (\mathcal{V} \setminus \mathcal{X})$, with $v \in \hat{S}(\mathcal{X})$. Then $\mathcal{X} \cup \{v\}$ in D if and only if $v \in \mathcal{N}^{\oplus}(\mathcal{V})$.*

Note: By assumption $v \notin \mathcal{X}$, and $v \in \hat{S}(\mathcal{X})$. It follows then from Lemma 5.2.1 that $v \notin \hat{P}(\mathcal{X})$.

Proof. The proof for this is similar to that of Theorem 5.4.1. $\square$

Theorem 5.4.3. *Let $D = (\mathcal{V}, A)$ be a DAG. let $\emptyset \subseteq \mathcal{X} \subset \mathcal{V}$ be a convex set of D. Let $\mathcal{R} = \mathcal{V} \setminus (\mathcal{X} \cup \hat{P}(\mathcal{X}) \cup \hat{S}(\mathcal{X}))$.* [3] *Let $\mathcal{Y} = \mathcal{N}^{\ominus}(\mathcal{X}) \cup \mathcal{N}^{\oplus}(\mathcal{X}) \cup \mathcal{R})$ Then the collection of convex supersets of $\mathcal{X}$ having order $|\mathcal{X}| + 1$ is given precisely by $\{\mathcal{X} \cup \{y\} | y \in \mathcal{Y}\}$.*

Proof. We divide the vertices which might be added to $\mathcal{X}$ into several groups, each of which we consider separately. Note that we limit our consideration to vertices in $\mathcal{V} \setminus \mathcal{X}$, because we're only interested in vertices that are not yet members of $\mathcal{X}$.

We distinguish the vertices in $\mathcal{V} \setminus \mathcal{X}$ based on their memberships in $\hat{P}(\mathcal{X})$ and in $\hat{S}(\mathcal{X})$. Note that the vertices in $\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})$ are already members of $\mathcal{X}$, as proven in Lemma 5.2.1. This leaves us with just three groups of vertices to examine:

Case 1: $v \in \hat{P}(\mathcal{X}), v \notin \hat{S}(\mathcal{X}), v \notin \mathcal{X}$:

By Theorem 5.4.1, we have that for any vertex $w \in \hat{P}(\mathcal{X}) \setminus \hat{S}(\mathcal{X})$, $\mathcal{X} \cup \{w\}$ is a convex set of D if and only if $w \in \mathcal{N}^{\ominus}(\mathcal{X})$.

Furthermore, every vertex in $\mathcal{N}^{\ominus}(\mathcal{X})$ is covered by Case 1 (Lemma 5.2.12).

It follows then that each vertex $w \in \mathcal{N}^{\ominus}(\mathcal{X})$ has the quality that $\mathcal{X} \cup \{w\}$ is a convex set of D.

Furthermore, we conclude that no other vertex in covered by Case 1 (that is, no other single vertex in $\hat{P}(\mathcal{X}) \setminus \hat{S}(\mathcal{X})$) may be added to $\mathcal{X}$ to yield a convex set.

Case 2: $v \notin \hat{P}(\mathcal{X}), v \in \hat{S}(\mathcal{X}), v \notin \mathcal{X}$:

[3]The set $\mathcal{R}$ is defined only for notational convenience and carries no special meaning.

By Theorem 5.4.2, we have that for any vertex $w \in \hat{S}(\mathcal{X}) \setminus \hat{P}(\mathcal{X})$, $\mathcal{X} \cup \{w\}$ is a convex set of D if and only if $w \in \mathcal{N}^{\oplus}(\mathcal{X})$.

Furthermore, every vertex in $\mathcal{N}^{\oplus}(\mathcal{X})$ is covered by Case 2 (Lemma 5.2.13).

It follows then that each vertex $w \in \mathcal{N}^{\oplus}(\mathcal{X})$ has the quality that $\mathcal{X} \cup \{w\}$ is a convex set of D.

Furthermore, we conclude that no other vertex in covered by Case 2 (that is, no other single vertex in $\hat{S}(\mathcal{X}) \setminus \hat{P}(\mathcal{X})$) may be added to $\mathcal{X}$ to yield a convex set.

Case 3: $v \notin \hat{P}(\mathcal{X}), v \notin \hat{S}(\mathcal{X}), v \notin \mathcal{X}$:

Theorem 5.3.2 indicates that any one vertex w in this group has the quality that $\mathcal{X} \cup \{w\}$ is a convex set of D.

We have shown that Groups 1, 2, and 3 collectively cover all of the vertices that are candidates for single-vertex additions to $\mathcal{X}$.

The vertices from Group 1 that may be added to $\mathcal{X}$ are precisely the set $\mathcal{N}^{\ominus}(\mathcal{X})$

The vertices from Group 2 that may be added to $\mathcal{X}$ are precisely the set $\mathcal{N}^{\oplus}(\mathcal{X})$.

The vertices from Group 3 that may be added to $\mathcal{X}$ are precisely all vertices in Group 3. That is, $\mathcal{V} \setminus (\mathcal{X} \cup \hat{P}(\mathcal{X}) \cup \hat{S}(\mathcal{X}))$, or $\mathcal{R}$.

Therefore, the vertex set of which any one may be added to $\mathcal{X}$ to yield a convex set is given by $\mathcal{N}^{\ominus}(\mathcal{X}) \cup \mathcal{N}^{\oplus}(\mathcal{X}) \cup \mathcal{R}$.

□

Corollary 5.4.4. *The formula $\{\mathcal{X} \cup \{y\} \,|\, y \in \mathcal{Y}\}$ in Theorem 5.4.3 always provides at least one superset of $\mathcal{X}$.*

Proof. Theorem 4.2.6 establishes that any convex set $\mathcal{X}$ may always be grown by the addition of exactly one vertex, yielding another convex set.

Theorem 5.4.3 identifies $\{\, \mathcal{X} \cup \{y\} \,|\, y \in \mathcal{Y}\,\}$ as the exact set of such vertices. This set cannot be empty if at least one such superset of $\mathcal{X}$ exists. $\square$

Theorem 5.4.5. *Let $D = (\mathcal{V}, A)$ be a DAG. Let $\mathcal{X} \subseteq \mathcal{V}$ be a convex set of D, with $|\mathcal{X}| \geq 2$. Let $\mathcal{Y} = \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$. Let $\mathcal{Z} \subseteq \mathcal{Y}$. Then $\mathcal{X} \setminus \mathcal{Z}$ is a convex set of D.*

Note that a result similar to Theorem 5.4.5 is presented in [1, Lemma 17.2.4]:

> Let D be an acyclic graph, let X be a convex set of D and let $s \in X$ be a source or sink of D<X>. Then $X \setminus \{s\}$ is a convex set of D.

The sources and sinks of $D < \mathcal{X} >$ are precisely those vertices in $\mathcal{Y}$. Theorem 5.4.5 strengthens [1, Lemma 17.2.4] by establishing that deleting from $\mathcal{X}$ any subset of $\mathcal{Y}$, rather than just a single vertex $s \in \mathcal{Y}$, yields another convex set of D.

Proof. Suppose for contradiction that $\mathcal{X} \setminus \mathcal{Z}$ is not a convex set of D. Then D must contain a C-path of $\mathcal{X} \setminus \mathcal{Z}$, having the structure $\vec{C} = [(\mathcal{X} \setminus \mathcal{Z}),\ \ldots,\ w,\ \ldots,\ (\mathcal{X} \setminus \mathcal{Z})]$, with $w \notin (\mathcal{X} \setminus \mathcal{Z})$.

We consider the three possible regions of $\mathcal{V}$ in which w could reside, and show that w's presence in each of those regions leads to a contradiction. The three possibilities are $w \in \mathcal{V} \setminus \mathcal{X}$, $w \in \mathcal{X} \setminus \mathcal{Z}$, and $w \in Z$.

Case 1: $w \in \mathcal{V} \setminus \mathcal{X}$:

Recall that the presumed structure of $\vec{C}$ is

$$\vec{C} = [(\mathcal{X} \setminus \mathcal{Z}),\ \ldots,\ w,\ \ldots,\ (\mathcal{X} \setminus \mathcal{Z})]$$

Because $(\mathcal{X} \setminus \mathcal{Z}) \subseteq \mathcal{X}$, it is also true that the structure of $\vec{C}$ is

$$\vec{C} = [\mathcal{X},\ \ldots,\ w,\ \ldots,\ \mathcal{X}]$$

In Case 1, we have $w \in \mathcal{V} \setminus \mathcal{X}$, and therefore $w \notin \mathcal{X}$. Therefore the structure of $\vec{C}$ is $\vec{C} = [\mathcal{X}, \ldots, (w \notin \mathcal{X}), \ldots, \mathcal{X}]$. However, such a path is a C-path of $\mathcal{X}$ in D. The existence of such a path in D indicates that $\mathcal{X}$ is not a convex set of D, in contradiction of our assumptions.

Case 2: $w \in \mathcal{X} \setminus \mathcal{Z}$:

Case 2 is not possible by the definition of w.

Case 3: $w \in Z$:

Because $w \in \mathcal{Z}$, and $\mathcal{Z} \subseteq \mathcal{Y}$, we have $w \in \mathcal{Y}$. By the definition of $\mathcal{Y}$, no vertex in $\mathcal{Y}$ is a member of $\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})$. Therefore, $w \notin \hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})$.

From Lemma 5.2.7, we have $\hat{S}(\mathcal{X} \setminus \mathcal{Z}) \subseteq \hat{S}(\mathcal{X})$. From Lemma 5.2.10, we have $\hat{P}(\mathcal{X} \setminus \mathcal{Z}) \subseteq \hat{P}(\mathcal{X})$. Now using the algebra of sets we may conclude that $(\hat{P}(\mathcal{X} \setminus \mathcal{Z}) \cap \hat{S}(\mathcal{X} \setminus \mathcal{Z})) \subseteq (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$.

Therefore, because $w \notin \hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})$, we also have $w \notin (\hat{P}(\mathcal{X} \setminus \mathcal{Z}) \cap \hat{S}(\mathcal{X} \setminus \mathcal{Z}))$.

The structure of $\vec{C}$ is $\vec{C} = [(\mathcal{X} \setminus \mathcal{Z}), \ldots, w, \ldots, (\mathcal{X} \setminus \mathcal{Z})]$. From that structure it follows that $w \in \hat{P}(\mathcal{X} \setminus \mathcal{Z})$, and $w \in \hat{S}(\mathcal{X} \setminus \mathcal{Z})$, and therefore $w \in (\hat{P}(\mathcal{X} \setminus \mathcal{Z}) \cap \hat{S}(\mathcal{X} \setminus \mathcal{Z}))$ This contradicts our determination above that $w \notin (\hat{P}(\mathcal{X} \setminus \mathcal{Z}) \cap \hat{S}(\mathcal{X} \setminus \mathcal{Z}))$.

□

Corollary 5.4.6. *In Theorem 5.4.5, $\mathcal{Y} \neq \emptyset$.*

Proof. Recall that in Theorem 5.4.5, $\mathcal{Y} = \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$.

By the assumptions of Theorem 5.4.5 we have that $\mathcal{X} \neq \emptyset$. Therefore, in order to have $\mathcal{Y} = \emptyset$, it must the case that $\mathcal{X} \subseteq (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$. That is, $\forall x \in \mathcal{X}, (x \in \hat{P}(\mathcal{X}) \wedge x \in \hat{S}(\mathcal{X}))$.

However, if every vertex in $\mathcal{X}$ is a predecessor of some other vertex in $\mathcal{X}$, then $\mathcal{X}$ contains a cycle. This contradicts the assumption of Theorem 5.4.5 that D, the digraph containing $\mathcal{X}$, is acyclic. □

Theorem 5.4.7. *Let $D = (\mathcal{V}, A)$ be a DAG. Let $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$ be a convex set of D, with $|\mathcal{X}| \geq 2$. Let $v \in \mathcal{X}$. Then $\mathcal{X} \setminus \{v\}$ is a convex set of D if and only if $v \in \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$.*

This theorem complements Theorem 5.4.5, which states that given a convex set $\mathcal{X}$, one can remove from $\mathcal{X}$ any combination of vertices in $(\mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})))$ to yield another convex set. However, Theorem 5.4.5 only establishes the *sufficiency* of that condition.

This present theorem (5.4.7) establishes that when removing just a single vertex v from $\mathcal{X}$, it is not merely sufficient, but also necessary, that $v \in \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$ in order for $\mathcal{X} \setminus \{v\}$ to be convex. That is, the set $\mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$ is the precise formula for which individual vertices may be removed from $\mathcal{X}$ to yield a convex subset of $\mathcal{X}$ having order $|\mathcal{X}| - 1$.

Note that one of the this theorem's assumptions is that $|\mathcal{X}| > 1$. This is because the empty set is, by definition, not a convex set of D. Therefore removing a single vertex from $\mathcal{X}$ when $|\mathcal{X}| = 1$ would not yield a convex set of D.

Proof. From Theorem 5.4.5 we have that if $v \in \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$ then $\mathcal{V} \setminus \{v\}$ is a convex set of D.

What remains is to show the opposite implication: if $\mathcal{X} \setminus \{v\}$ is a convex set of D, then $v \in \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$.

Suppose for contradiction that $\mathcal{X} \setminus \{v\}$ is a convex set of D, and yet $v \notin \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$.

$v \notin \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$ is satisfied by $v \notin \mathcal{X}$ or $v \in (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$. However, $v \in \mathcal{X}$ is a requirement of the theorem, and so the only remaining possibility is that $v \in (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$.

From $v \in (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$, it follows that D contains a path of the form $\vec{C} = [\mathcal{X},\ \ldots,\ v,\ \ldots,\ \mathcal{X}]$.

Because D is acyclic, a vertex cannot appear more than once in any path of D. We may therefore refine the structure of $\vec{C}$ to be $\vec{C} = [(\mathcal{X} \setminus \{v\}),\ \ldots,\ v,\ \ldots,\ (\mathcal{X} \setminus \{v\})]$.

However, we see from the structure of $\vec{C}$ that it is a C-path of the set $(\mathcal{X} \setminus \{v\})$. Therefore, we have that $(\mathcal{X} \setminus \{v\})$ is not a convex set of D. This violates our assumption to the contrary. $\square$

Corollary 5.4.8. *Let $D = (\mathcal{V}, A)$ be a DAG. Let $\mathcal{X} \subseteq \mathcal{V}$ be a convex set of D, with $|\mathcal{X}| \geq 2$. Let $\mathcal{Y} = \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$. Let $\mathcal{Z} \subseteq \mathcal{Y}$, with $|\mathcal{Z}| \geq 2$. Then for $\delta \geq 2$, there are some graphs D and convex sets $\mathcal{X}$, such that $\{\mathcal{X} \setminus \mathcal{Z} \,|\, \mathcal{Z} \subseteq \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})), |\mathcal{Z}| = \delta\}$ does not generate all subsets of $\mathcal{X}$ that are convex sets of D and that have order $|\mathcal{X}| - \delta$.*

Theorem 5.4.7 established that when $\delta = 1$, the formula

$$\{\mathcal{X} \setminus \mathcal{Z} \,|\, \mathcal{Z} \subseteq \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})), |\mathcal{Z}| = \delta\}$$

yields exactly those convex sets of D that are subsets of $\mathcal{X}$, and have order $|\mathcal{X}| - \delta$.

This corollary demonstrates that when $\delta > 1$, there are some DAG's D with particular convex sets $\mathcal{X}$, such that the formula $\{\mathcal{X} \setminus \mathcal{Z} \,|\, \mathcal{Z} \subseteq \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})), |\mathcal{Z}| = \delta\}$ generates only *some* of convex sets of D that are subsets of $\mathcal{X}$ and have order $|\mathcal{X}| - \delta$.

Proof. We prove this with a counterexample.

Let $D = (\mathcal{V}, A)$, with $\mathcal{V} = \{a,\ b,\ c\}$ and $A = \{(a, b),\ (b, c)\}$. To keep our example small, let $\mathcal{X} = \mathcal{V}$. Let $\delta = 2$.

From this we have:

$$\begin{aligned}
\hat{P}(\mathcal{X}) &= \hat{P}(\{a\}) \cup \hat{P}(\{b\}) \cup \hat{P}(\{c\}) \\
&= \emptyset \cup \{a\} \cup \{a,\ b\} \\
&= \{a,\ b\} \\
\\
\hat{S}(\mathcal{X}) &= \hat{S}(\{a\}) \cup \hat{S}(\{b\}) \cup \hat{S}(\{c\}) \\
&= \{b,\ c\} \cup \{c\} \cup \emptyset \\
&= \{b,\ c\} \\
\\
\mathcal{Y} &= \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})) \\
&= \{a,\ b,\ c\} \setminus \{b\} \\
&= \{a,\ c\}
\end{aligned}$$

Now consider the collection of subsets generated by the approach under consideration: $\{\mathcal{X} \setminus \mathcal{Z} \,|\, \mathcal{Z} \subseteq \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X})), |\mathcal{Z}| = \delta\}$

Only one value of $\mathcal{Z}$ meets all of the generator's criteria when $\delta = 2$: $\mathcal{Z} = \{a,\ c\}$. Therefore this generator produces only one subset of $\mathcal{X}$: $\mathcal{X} \setminus \mathcal{Z} = \{a,\ b,\ c\} \setminus \{a,\ c\} = \{b\}$.

However, by inspection of D and $\mathcal{X}$, we may easily identify other subsets of $\mathcal{X}$ that are convex sets of D, and have order $|\mathcal{X}| - \delta = 1$. Those sets are $\{a\}$ and $\{c\}$. □

Theorem 5.4.9. *et $D = (\mathcal{V}, A)$ be a DAG. Let $\mathcal{X} \subseteq \mathcal{V}$ be a convex set of D, with $|\mathcal{X}| \geq 2$. Let $\mathcal{Y} = \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$. Let $\mathcal{Z} \subseteq \mathcal{Y}$. Then $\mathcal{X} \setminus \mathcal{Z}$ is a convex set of D.*

Proof. Theorem 4.2.7 establishes that any convex set $\mathcal{X}$ may always be shrunk by the deletion of exactly one vertex, yielding another convex set.

Theorem 5.4.7 identifies $\{\, \mathcal{X} \cup \{y\} \,|\, y \in \mathcal{Y} \,\}$ as the exact set of such vertices. This set cannot be empty if at least one such subset of $\mathcal{X}$ exists. □

CHAPTER 6

Arbitrary Transformation of Convex Sets

In this section we show that given any two convex sets $\mathcal{X}$ and $\mathcal{Y}$ of the same DAG D, it is always possible to transform $\mathcal{X}$ into $\mathcal{Y}$ using a sequence of single-vertex changes, such that each intermediate set produced is also a convex set of D. We establish this by showing the existence of an algorithm (Algorithm 19 in Appendix D.2) which performs such a transformation. This provides the basis for our proof (Theorem 7.5.3 in Subsection 7.5.3) that the probability of Algorithm 3 searching all convex sets of any DAG approaches 1 as the number of algorithm iterations approaches infinity.

This section is structured as follows. In Section 6.1 we give an overview of the approach used by Algorithm 19. In Section 6.2 we walk through an example execution of Algorithm 19 to clarify its approach.

In keeping with the structure of this work, a proof of Algorithm 19's correctness is presented alongside the algorithm itself (see Appendix D.2.1).

The focus of this chapter is to demonstrate that particular evolutionary chains always exist for convex sets. This thesis presents Algorithm 19 solely for the purpose of establishing that fact. Algorithm 19 is not intended to for use in any actual optimization system, and for this reason we do not study its asymptotic running time.

6.1 General Approach

The general approach taken by Algorithm 19 is as follows. Variable and parameter names used below have the meanings given in Algorithm 19 on page 134.

We first identify a sequence $\overline{\mathcal{S}}$ of $|\mathcal{X}| - 1$ single-vertex deletions that transforms $\mathcal{X}$ into a one-vertex convex set $\{u\}$.

Next, we identify a sequence $\overline{\mathcal{T}}$ of $|\mathcal{Y}| - 1$ single-vertex deletions that transforms $\mathcal{Y}$ into a one-vertex convex set $\{v\}$. Importantly, the reversal of this sequence $\overline{\mathcal{T}}$ is a sequence of single-vertex *additions* which transforms $\{v\}$ into $\mathcal{Y}$.

From this we obtain the overall transformation sequence of $\mathcal{X}$ to $\mathcal{Y}$ as follows. We begin with $\mathcal{X}$. Apply, in order, each of the $|\mathcal{X}|-1$ single-vertex deletions indicated by $\overline{\mathcal{S}}$, giving us $\{u\}$.

Now replace $\{u\}$ with $\{v\}$. This swap operation is valid because it only changes one vertex in our evolving set, and every single-vertex set is trivially a convex set.

Finally, grow $\{v\}$ into $\mathcal{Y}$ using the sequence of single-vertex additions indicated by the reversal of $\overline{\mathcal{T}}$.

A concrete example of this follows.

6.2 Example of Algorithm 19

Here we step through an instance of `evolve_between_convex_sets` (Algorithm 19) in order to provide an intuitive understanding of its operation. As shown in Figure 3, we use the DAG:

$$D = (\{a \dots h\}, \{(a,c),\ (b,d),\ (c,e),\ (c,f),\ (d,f),\ (e,g),\ (f,g),\ (f,g)\})$$

an initial convex set $\mathcal{X} = \{c,\ e,\ f,\ g\}$ and a final convex set $\mathcal{Y} = \{a,\ c,\ d\}$.

On line 1 of `evolve_between_convex_sets` we invoke the algorithm `evolve_convex_set_to_vertex` (Algorithm 18) with the arguments D and $\mathcal{X}$.

On line 1 of `evolve_convex_set_to_vertex` we compute some arbitrary topological sort of D in which $\mathcal{X}$ appears as a contiguous subsequence. For this example let that ordering be $\vec{Q} = [a,\ b,\ d,\ c,\ e,\ f,\ g,\ h]$.

On line 2 of `evolve_convex_set_to_vertex` we set $x\_low \leftarrow 4$ and $x\_high \leftarrow 7$.

On lines 3-6 of `evolve_convex_set_to_vertex` we build the sequence:

$$[\{c,\ e,\ f,\ g\},\ \{e,\ f,\ g\},\ \{f,\ g\},\ \{g\}]$$

which is then returned to `evolve_between_convex_sets` where it is named $\overline{\mathcal{Q}}$. Note that as required, each set in $\overline{\mathcal{Q}}$ is itself a convex set of D. This progression is shown in Figures 3a through 3d.

Line 2 of the algorithm `evolve_between_convex_sets` again invokes `evolve_convex_set_to_vertex`, this time with arguments $\mathcal{Y}$ and D. For this example we assume that this invocation of `evolve_convex_set_to_vertex` returns the sequence $\overline{\mathcal{R}} = [\{a,\ c,\ d\},\ \{a,\ d\},\ \{d\}]$. This is then reversed on line 3 of `evolve_between_convex_sets` to give us $\overline{\mathcal{S}} = [\{d\},\ \{a,\ d\},\ \{a,\ c,\ d\}]$, shown in Figures 3e through 3g.

Finally on line 6 of `evolve_between_convex_sets`, $\overline{\mathcal{Q}}$ and $\overline{\mathcal{S}}$ are combined into the final sequence:

$$\overline{\mathcal{T}} = [\{c,\ e,\ f,\ g\},\ \{e,\ f,\ g\},\ \{f,\ g\},\ \{g\},\ \{d\},\ \{a,\ d\},\ \{a,\ c,\ d\}]$$

The progression of modifications made to $\mathcal{X}$ is therefore as follows: delete c, delete e, delete f, replace g with d, add a, add c. Notice that each of these operations makes only a single-vertex change to the current vertex set, and preserves the convexity of the set as it evolves.

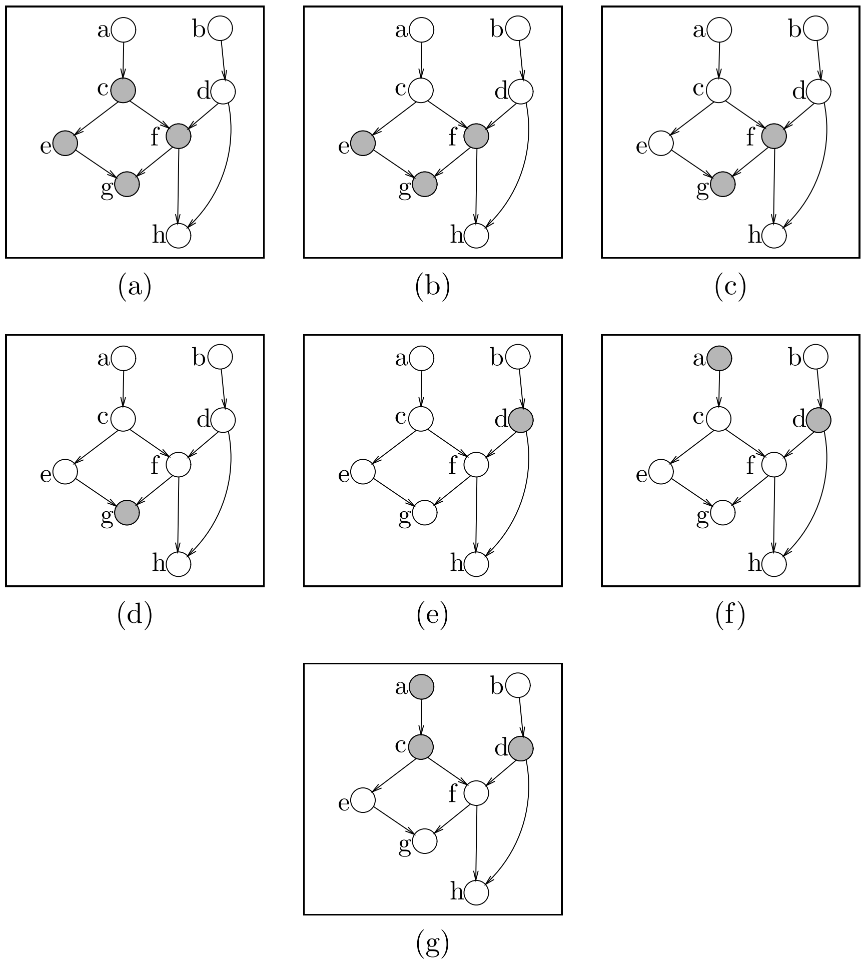


Figure 3: Evolution of the convex set $\mathcal{X} = \{c,\ e,\ f,\ g\}$ to the convex set $\mathcal{Y} = \{a,\ c,\ d\}$ using Algorithm 19. The algorithm ensures that each intermediate set (Figure 3b-3f) is also a convex set of D.

CHAPTER 7
Stochastic Optimization of a Single Convex Set

In this chapter, we consider in detail the optimization problem of finding a global minimum over all convex sets of a DAG, as motivated by the program parallelization problem described in Section 3.1.

Section 7.1 formally states the optimization problem. In Section 7.2 we discuss how simulated annealing may be useful in this kind of optimization algorithm. Section 7.3 discusses options regarding seeding such an optimization algorithm, and Section 7.4 discusses the choice of using choosing predecessor/successor sets versus topological sorts when evolving convex sets. Finally, Section 7.5 presents an algorithm which optimizes a single convex set of a DFG.

7.1 Problem Statement

Our optimization problem is as follows. Given a DAG D and a cost function `cost` : (any convex set of D) $\to \mathbb{R}$, find some convex set $\mathcal{M}$ of D that minimizes `cost(`$\mathcal{M}$`)`. This optimization problem is addressed by Algorithm 3 in Section 7.5.

Note that Algorithm 3 only attempts to evolve one convex set at a time, rather than a collection of convex sets as discussed in Section 3.1.

Algorithm 3 is presented primarily to demonstrate the existence of an evolutionary algorithm which is efficient on a per-iteration basis, and which always retains the possibility to cover the entire search space in a finite number of future iterations. This is in contrast to some genetic algorithms which may become trapped in local minima due to premature convergence [31], or whose designs *a priori* may preclude parts of the search space for certain inputs.

We make no assumption that, as presented, Algorithm 3 is likely to identify approximately an optimal solution within a usefully small number of iterations. Below we discuss several adaptations to Algorithm 3 which may be useful in pursuing such an algorithm. These may be fruitful areas of future investigation.

Note that when proving that Algorithm 3 potentially covers the entire search space, for generality we make no assumption that the function `cost`. It follows then that in order to find a global minimum, Algorithm 3 must potentially visit every convex set in a DAG. In contrast, the algorithm modifications described below are only effective when the cost function contains at least some structure to guide the search.

7.2 Simulated Annealing

Simulated annealing (SA) [32, 33] is an evolutionary optimization method in which initial iterations permit large changes to the trial solutions, but as the algorithm continues, successive iterations permit progressively smaller changes to the trial solutions.

SA requires a distance function $\delta : (\mathcal{X}, \mathcal{Y}) \rightarrow \mathbb{R}$ to be defined over members of the search space. SA further requires a random perturbation operator π which, given a trial solution $\mathcal{X}$ and an upper-limit Δ, returns a new trial solution $\mathcal{Y}$ such that $\delta(\mathcal{X}, \mathcal{Y}) \leq \Delta$. That is, $\delta(\mathcal{X}, \pi(\Delta, \mathcal{X})) \leq \Delta$.

The basic convexity algorithms presented in this work may be well-suited to simulated annealing-type optimizers. If we define δ to be the number of vertices added or deleted from an existing convex set, then δ may be easily defined using the algorithms `grow_convex_set_PS`, `grow_convex_set_TS`, `shrink_convex_set_PS`, and `shrink_convex_set_TS` (Appendix C).

Another potentially useful approach is to let δ be the number of single-vertex changes applied to some trial solution $\mathcal{X}$, and for π to simply be the random application of up to Δ single-vertex changes. Algorithm 3 implicitly provides an example of such perturbation operator, when δ iterations of its main loop are executed.

7.3 Initialization by Seed Set vs. Order

In Appendix C we present to separate two variations of algorithms for initializing a convex set.

`init_convex_set_by_seeds_PS` and `init_convex_set_by_seeds_TS` take a potentially non-convex *seed set* of vertices and return some convex set which is a superset of the seed set. In contrast, `init_convex_set_by_order_PS` and `init_convex_set_by_order_TS` return a convex set having some specified number of vertices, without regard to which particular vertices constitute that set.

There may exist applications in which a seed set carries special significance, such that the optimization algorithm benefits from beginning its search in the neighborhood of the seed set(s). For such applications, it may be preferable to use `init_convex_set_by_seeds_PS` rather than `init_convex_set_by_order_TS`.

`init_convex_set_by_seeds_PS` is guaranteed to return the smallest superset of the seed set that is also a convex set, whereas `init_convex_set_by_order_TS` makes no provision for limiting the size of the resulting set. Therefore `init_convex_set_by_seeds_PS` may initialize the search algorithm with a trial solution that is closer to an acceptable solution, leading to faster convergence than would be obtained if using `init_convex_set_by_order_TS`.

However, if any of the $\hat{P}/\hat{S}$-based basic convexity algorithms is employed by the optimization algorithm, it must first compute the transitive closure of the dataflow graph, which for a DAG of order $|\mathcal{V}|$ requires time $O(|\mathcal{V}|^{2.376})$ (see Section 2.7). Algorithm 3 uses `init_convex_set_by_order_TS` and so limits the running time of its initialization phase to just $O(|\mathcal{V}|^2)$.

7.4 $\hat{P}/\hat{S}$ vs. Topological Sort

Appendix C presents two essentially interchangeable collections of basic convexity algorithms, one based on predecessor and successor sets, and one based on topological sorts. Algorithm 3 could be trivially modified to use either group of basic convexity algorithms, or any combination thereof.

In general the basic convexity algorithms in Appendix C which are based upon $\hat{P}$ and $\hat{S}$ have longer asymptotic running times than those using topological sort. All of the $\hat{P}/\hat{S}$-based algorithms require the initial computation of the DAG's transitive closure, requiring time $O(|\mathcal{V}|^{2.376})$. Furthermore, the algorithms
`init_convex_set_by_order_PS` and `grow_convex_set_PS` require time $O(|\mathcal{V}|^3)$ each time they execute.

From a running-time perspective there appears to be little reason to use the $\hat{P}/\hat{S}$-based algorithms. If reasons exist to initialize the search using a seed set as discussed in Section 7.3, a hybrid approach may be desirable in which
`init_convex_set_by_seeds_PS` is used during initialization of the optimization algorithm, but convex set growth and shrinking are accomplished using the topological sort-based algorithms `grow_convex_set_TS` and `shrink_convex_set_TS`, which have running time $O(|\mathcal{V}|^2)$.

Note that this work assumes the existence of a random algorithm which has a positive probability of producing each topological sort of a given DAG (Appendix A.3).

7.5 Convex Set Stochastic Optimization Algorithm

In this section we present and discuss a stochastic optimization algorithm, Algorithm 3, whose domain is all convex sets of a given DAG. This section is structured as follows.

The pseudocode for Algorithm 3 is presented on page 73.

Subsection 7.5.1 provides a brief overview of the algorithm's approach.

Subsection 7.5.2 demonstrates that each individual iteration of the Algorithm 3 is fairly efficient (time $O|\mathcal{V}^2|$), and Subsection 7.5.3 presents a proof that the when permitted to run indefinitely, the probability of Algorithm 3 searching all convex set of a DAG approaches 1.

7.5.1 Overview of Algorithm 3

Algorithm 3 operates as follows.

A convex set, $\mathcal{X}_{trial}$ is initialized to some arbitrary convex set (line 2). Although line 2 selects a random convex set of order 1, any convex set of D would be acceptable as an initial value of $\mathcal{X}_{trial}$.

Each iteration of the loop evaluates the cost function score of the current value of $\mathcal{X}_{trial}$ (line 4), and if the score is the minimum value so far discovered, both the score and the convex set which yielded it are recorded (lines 5-10).

Line 11 calls `is_done` to provide the user an opportunity to terminate the search for any reason, such as time constraints. As stated in the algorithm's preconditions, we permit `is_done` to retain state between calls so that it may detect convergence over multiple iterations of the optimization algorithm.

Lines 12-29 choose and apply any valid single-vertex change to the current value of $\mathcal{X}_{trial}$.

7.5.2 Running Time of Algorithm 3

We easily see that the algorithm lines other than 4 and 11 have a maximum running time of $O(|\mathcal{V}|^2)$. We make no assumption about the running times of the functions `cost` and `is_done`. Therefore the overall running time of each iteration of this algorithm is $max($ $O(|\mathcal{V}|^2)$, (running time of `cost`), (running time of `is_done`)).

Algorithm 3 Stochastically Optimize over all Convex Sets of a DAG

Function `stochastic_search_TO` : $(D,$ `cost`, `is_done`$) \rightarrow \mathcal{X}_{best}$

Require:

(R1) $D = (\mathcal{D}, A)$ is a DAG.

(R2) `cost` : (all convex sets of D) $\rightarrow \mathbb{R}$

(R3) `is_done` : (vertex set, $\mathbb{R}$) $\rightarrow$ *Boolean* is a subroutine which may retain internal state between invocations, and whose internal state is assumed to be properly initialized prior to the execution of this algorithm.

Ensure:

(E1) Of all sets evaluated by `cost` before `is_done` returns **true**, $\mathcal{X}_{best}$ is one of the sets for which `cost` returned the lowest value.

```
 1: $first\_iter \leftarrow$ true                                          (bind identifier)     $O(1)$
 2: $\mathcal{X}_{trial} \leftarrow$ call vset_random_subset( $\mathcal{V}, 1$ )                        $O(|\mathcal{V}|)$
 3: repeat
 4:   $cost_{trial} \leftarrow$ call cost( $\mathcal{X}_{trial}$ )                              (time of cost)
 5:   $promote \leftarrow (cost_{trial} < cost_{best})$ or $first\_iter$    (scalar op)           $O(1)$
 6:   if $promote$ then                                   (scalar op)           $O(1)$
 7:     $first\_iter \leftarrow$ false                              (bind identifier)     $O(1)$
 8:     $cost_{best} \leftarrow cost_{trial}$                             (bind identifier)     $O(1)$
 9:     $\mathcal{X}_{best} \leftarrow \mathcal{X}_{trial}$                                 (bind identifier)     $O(1)$
10:   end if
11:   $done \leftarrow$ call is_done( $\mathcal{X}_{trial}, cost_{trial}$ )                  (time of is_done)
12:   if not $done$ then                                  (scalar op)           $O(1)$
13:     if $|\mathcal{X}_{trial}| = 1$ then                           (scalar op)           $O(1)$
14:       $valid\_ops \leftarrow$ {SWAP}                        symset_init(...)      $O(1)$
15:     else
16:       $valid\_ops \leftarrow$ {SWAP, SHRINK}                symset_init(...)      $O(1)$
17:     end if
18:     if $|\mathcal{X}_{trial}| < |\mathcal{V}|$ then                       (scalar op)           $O(1)$
19:       $valid\_ops \leftarrow valid\_ops \cup$ {GROW}              symset_add(...)       $O(1)$
20:     end if
21:     $op \leftarrow$ call symset_random_elem( $valid\_ops$ )                       $O(1)$
22:     if $op$ = GROW then
23:       $\mathcal{X}_{trial} \leftarrow$ call grow_convex_set_TS( $\mathcal{X}_{trial}, 1$ )               $O(|\mathcal{V}|^2)$
24:     else if $op$ = SHRINK then
25:       $\mathcal{X}_{trial} \leftarrow$ call shrink_convex_set_TS( $\mathcal{X}_{trial}, 1$ )             $O(|\mathcal{V}|^2)$
26:     else (Comment: $op$ = SWAP)
27:       $\mathcal{X}_{trial} \leftarrow$ call vset_random_subset( $\mathcal{V}, 1$ )                  $O(|\mathcal{V}|)$
28:     end if
29:   end if
30: until $done$
31: return $\mathcal{X}_{best}$
```

This search algorithm is efficient insofar as, aside from the unconstrained running times of `cost` and `is_done`, its per-iteration running time is polynomial with respect to the order of the DAG. The relevance of such efficiency is clearly dictated by the running times of `cost` and `is_done` in any particular application of this algorithm.

7.5.3 Complete Search Coverage of Algorithm 3

In this subsection we show that given enough iterations, `stochastic_search_TO` always discovers some vertex set $\mathcal{M}$ which minimizes `cost`.

Lemma 7.5.1. *Let $D = (\mathcal{V}, A)$ be a DAG, and let `cost` and `is_done` be defined as required by Algorithm 3.*

Let m be the minimal value that `cost` returns for any convex set of D, and let $\mathcal{M}$ be any convex set of D such that `cost`$(\mathcal{M}) = m$.

*Let `stochastic_search_TO` $(D$, `cost`, `is_done`$)$ (Algorithm 3) be mid-execution, with line 3 as the next instruction to be executed, and with the loop spanning lines 3-30 having previously executed $i - 1$ times for some $i \geq 1$. Let `is_done` return **false** during next $2 \times |\mathcal{V}| - 1$ iterations of the algorithm's loop. (I.e., iterations $i \ldots (i + 2 \times |\mathcal{V}| - 1)$). Then there is a finite, positive probability that the algorithm sets $\mathcal{X}_{trial} = \mathcal{M}$ during loop iteration s, with $s \in [1, (i + 2 \times |\mathcal{V}| - 1)]$.*

Informally, Lemma 7.5.1 states that regardless of where Algorithm 3 is in the search process, as encoded by its current values for $\mathcal{X}_{trial}$, $\mathcal{X}_{best}$, $first\_iter$ and $cost\_best$, there is a finite, positive probability that during its next $(2 \times |\mathcal{V}| - 1)$ iterations the algorithm discovers some global minimum $\mathcal{M}$. However, because the function `is_done` could terminate the algorithm after any loop iteration, we require the assumption that `is_done` doesn't terminate the algorithm until the next $(2 \times |\mathcal{V}| - 1)$ iterations have completed.

Proof. Let $\overline{\mathcal{T}} =$ `evolve_between_convex_sets` $(D, \mathcal{X}_{trial}^{(i)}, \mathcal{M})$. (Note that `evolve_between_convex_sets` is used only in our analysis, and is invoked neither directly nor indirectly by `stochastic_search_TO`.)

Let $\mathcal{X}_{trial}^{(i)}$ indicate the value of $\mathcal{X}_{trial}$ immediately prior to the i^{th} iteration of the loop.

We proceed by showing that there is a finite, positive probability that the next $|\overline{\mathcal{T}}| - 1$ iterations of the algorithm's loop evolve $\mathcal{X}_{trial}$ into $\mathcal{M}$ in precisely the sequence indicated by $\overline{\mathcal{T}}$. That is, for all $j \in [1, |\overline{\mathcal{T}}|]$, the algorithm sets $\mathcal{X}_{trial}^{(i+j-1)} = \overline{\mathcal{T}}[\![j]\!]$.

We proceed inductively. Assuming that $\mathcal{X}_{trial}^{(i+j-1)} = \overline{\mathcal{T}}[\![j]\!]$, we demonstrate that there is a finite, positive probability that the next iteration of the algorithm's loop (the $(i+j)^{th}$ iteration overall) evolves the convex set $\mathcal{X}_{trial}^{(i+j-1)}$ into $\mathcal{X}_{trial}^{(i+j)}$ such that $\mathcal{X}_{trial}^{(i+j)} = \overline{\mathcal{T}}[\![j+1]\!]$.

For our inductive base case, we must show that when $j = 1$, there is a finite, positive probability that $\mathcal{X}_{trial}^{(i+j-1)} = \overline{\mathcal{T}}[\![j]\!]$. This is trivially established by the definition of `evolve_between_convex_sets`, which deterministically ensures that $\overline{\mathcal{T}}[\![1]\!] = \mathcal{X}_{trial}^{(i)}$.

Our inductive step covers $1 < j < |\overline{\mathcal{T}}|$, that is, the $(i+1)^{th}$ through the $(i + |\overline{\mathcal{T}}| - 1)^{th}$ loop iterations. By our inductive assumption, prior to each of these loop iterations we have $\mathcal{X}_{trial}^{(i+j-1)} = \overline{\mathcal{T}}[\![j]\!]$. Our goal is to demonstrate that each of these loop iterations has a finite, positive probability of setting $\mathcal{X}_{trial}^{(i+j)} = \overline{\mathcal{T}}[\![j+1]\!]$.

We consider three cases reflecting the different relationships that may exist between $\overline{\mathcal{T}}[\![j]\!]$ and $\overline{\mathcal{T}}[\![j+1]\!]$. We show that for each of these cases, the next (i.e., the $(i+j-1)^{th}$ overall) loop iteration has a finite, positive probability of setting $\mathcal{X}_{trial}^{(i+j)} = \overline{\mathcal{T}}[\![j+1]\!]$.

Case 1: Delete one vertex:

In Case 1, there is some vertex $u \in \overline{\mathcal{T}}[\![j]\!]$, such that $\overline{\mathcal{T}}[\![j+1]\!] = \overline{\mathcal{T}}[\![j]\!] \setminus \{u\}$.

By the definition of `evolve_between_convex_sets`, every element of $\overline{\mathcal{T}}$ is a non-empty set. Therefore $|\overline{\mathcal{T}}[\![j+1]\!]| \geq 1$. Because $\overline{\mathcal{T}}[\![j+1]\!]$ is the result of deleting a vertex from $\overline{\mathcal{T}}[\![j]\!]$, we have $|\overline{\mathcal{T}}[\![j]\!]| > 1$. This ensures that line 16 of the algorithm executes, and so we have `SHRINK` $\in$ *valid_ops*.

Line 21 has a finite, positive probability of selecting any symbol in *valid_ops*, and so after line 21 there's a finite, positive probability that $op =$ `SHRINK`, causing line 25 to execute.

Line 25 performs the operation
$\mathcal{X}_{trial}^{(i+j)} \leftarrow$ `shrink_convex_set_TS(` $\mathcal{X}_{trial}^{(i+j-1)}, 1$ `)`. As stated in postcondition (E4) of Algorithm 17 (see Appendix C.8), this invocation of
`shrink_convex_set_TS` has a finite, positive probability of returning *any* convex subset of $\mathcal{X}_{trial}^{(i+j-1)}$ resulting from the deletion of a single vertex.

By our inductive assumption we have $\mathcal{X}_{trial}^{(i+j-1)} = \overline{\mathcal{T}}[\![j+1]\!]$. Therefore line 25 has a finite, positive probability of returning any convex subset of $\overline{\mathcal{T}}[\![j]\!]$ resulting from the deletion of a single vertex from $\overline{\mathcal{T}}[\![j]\!]$. $\overline{\mathcal{T}}[\![j+1]\!]$ is one such set, and so there is a finite, positive probability that line 25 sets $\mathcal{X}_{trial}^{(i+j)} = \overline{\mathcal{T}}[\![j+1]\!]$.

Case 2: Add one vertex:
In Case 2, there is some vertex $u \notin \overline{\mathcal{T}}[\![j]\!]$, such that $\overline{\mathcal{T}}[\![j+1]\!] = \overline{\mathcal{T}}[\![j]\!] \cup \{u\}$.

The proof for Case 2 is similar in structure to that of Case 1, and for brevity we provide only a sketch.

$|\overline{\mathcal{T}}[\![j]\!]| \leq |\mathcal{V}|$, and so $|\overline{\mathcal{T}}[\![j+1]\!]| < |\mathcal{V}|$. Therefore there is a finite, positive probability that line 23 of the algorithm executes during this loop iteration.

Similarly to `shrink_convex_set_TS`, there is a finite, positive probability that `grow_convex_set_TS` produces any one of the convex sets of D which is a superset of $\overline{\mathcal{T}}[\![j]\!]$ and has order $|\overline{\mathcal{T}}[\![j]\!]| + 1$. $\overline{\mathcal{T}}[\![j+1]\!]$ is one such set, and so there is a finite, positive probability that that line 23 of the algorithm sets $\mathcal{X}_{trial}^{(i+j)} = \overline{\mathcal{T}}[\![j+1]\!]$.

Case 3: Swap vertex:

From the definition of `evolve_between_convex_sets`, Case 3 only arises when we have $|\overline{\mathcal{T}}[\![j+1]\!]| = |\overline{\mathcal{T}}[\![j]\!]| = 1$.

In Case 3 there is a finite, positive probability that line 21 of the algorithm sets $op = \texttt{SWAP}$, leading to the execution of line 27.

`vset_random_subset` $(\mathcal{V}, 1)$ function is defined to have a finite, positive probability of returning any set $\{v\}$ such that $v \in \mathcal{V}$. $\overline{\mathcal{T}}[\![j+1]\!]$ is one such set, and so there's a finite, positive probability that line 27 of the algorithm sets $\mathcal{X}_{trial}^{(i+j)} = \overline{\mathcal{T}}[\![j+1]\!]$.

Through induction we've shown that there is a finite, positive probability that $\mathcal{X}_{trial}^{(i+|\overline{\mathcal{T}}|-1)} = \overline{\mathcal{T}}[\![\ |\overline{\mathcal{T}}|\ ]\!]$. By the definition of `evolve_between_convex_sets` $(D, \mathcal{X}_{trial}^{(i)}, \mathcal{M})$ we have $\overline{\mathcal{T}}[\![\ |\overline{\mathcal{T}}|\ ]\!] = \mathcal{M}$.

We conclude with proving the final claim of this Lemma: that $(2 \times |\mathcal{V}| - 1)$ iterations of the algorithm's loop have a finite, positive probability of yielding $\mathcal{X}_{trial} = \mathcal{M}$, regardless of the initial state of the algorithm.

This follows directly from the maximum length of $\overline{\mathcal{T}}$. `evolve_between_convex_sets` $(D, \mathcal{X}_{trial}^{(i)}, \mathcal{M})$ returns a vertex set sequence with the following structure. The first $|\mathcal{X}_{trial}^{(i)}|$ elements of $\overline{\mathcal{T}}$ are improper subsets $\mathcal{X}_{trial}^{(i)}$. The remaining elements of $\overline{\mathcal{T}}$ are subsets of $\mathcal{M}$, either $|\mathcal{M}|$ or $|\mathcal{M}|-1$ in number. Therefore the maximum length of $\overline{\mathcal{T}}$ is $|\mathcal{X}_{trial}^{(i)}| + |\mathcal{M}|$.

Both $\mathcal{X}_{trial}$ and $\mathcal{M}$ are constrained to be improper subset of $\mathcal{V}$, giving us $|\mathcal{X}_{trial}| \leq |\mathcal{V}|$ and $|\mathcal{M}| \leq |\mathcal{V}|$. From this we have $|\overline{\mathcal{T}}| \leq 2 \times |\mathcal{V}|$. Note however that immediately prior to the i^{th} iteration of the algorithm's loop, we already have $\mathcal{X}_{trial} = \overline{\mathcal{T}}[\![1]\!]$. Therefore we need only $|\overline{\mathcal{T}}| - 1$, or $2 \times |\mathcal{V}| - 1$, loop iterations to obtain $\mathcal{X}_{trial} = \overline{\mathcal{T}}[\![\ |\overline{\mathcal{T}}|\ ]\!]$ using the evolutionary sequence indicated by $\overline{\mathcal{T}}$. $\square$

Lemma 7.5.2. *Let $D = (\mathcal{V}, A)$ be a DAG and let* `cost` *be a cost function as required by Algorithm 3. For any particular activation of Algorithm 3, let P_i be the probability that*

Algorithm 3 finds a global minimum within the first i iterations, with $P_i < 1$. Let P_j be the probability that Algorithm 3 finds a global minimum within the first $i + (2 \times |\mathcal{V}| - 1)$ loop iterations. Then $P_i < P_j \leq 1$.

Informally, Lemma 7.5.2 states that the more iterations executed by the loop in Algorithm 3, the greater the cumulative probability that a global minimum is found and returned.

Note that the theorem does not claim that every single iteration following the i^{th} increases the probability of finding a global minimum. We avoid such a claim because there is no assurance that the convex set indicated by $\mathcal{X}_{trial}$ just prior to the i^{th} iteration of the algorithm's loop can be evolved into some global minimum of `cost` by adding, deleting, or swapping precisely one vertex. However, Lemma 7.5.1 establishes that regardless of the graph topology of D and regardless of the state of the algorithm just prior to the i^{th} loop iteration, there is a finite, positive probability of reaching a global minimum within the subsequent $2 \times |\mathcal{V}| - 1$ iterations.

Proof. From Lemma 7.5.1 we have that there is a finite, positive probability that during iterations i to $i + (2 \times |\mathcal{V}| - 1)$ of the algorithm, there is some finite, positive probability that X_{trial} is set to some global minimum of `cost`. Let P_{ij} by the probability of some particular global minimum $\mathcal{M}$ being found during these $2 \times |\mathcal{V}| - 1$ iterations.

Then the cumulative probability P_j of the algorithm finding $\mathcal{M}$ after $i + (2 \times |\mathcal{V}| - 1)$ loop iterations have been performed is:

$$P_j = P_i + (1 - P_i) \times P_{ij}$$

From Lemma 7.5.1 we have that $P_{ij} > 0$, and so $P_j > P_i$. $\square$

Theorem 7.5.3. *Let $D = (\mathcal{V}, A)$ be a DAG and let `cost` be a cost function as required by Algorithm 3. For any particular activation of Algorithm 3, let $P(i)$ be the probability that Algorithm 3 finds a global minimum within the first i iterations. Then $\lim_{i \to \infty} P(i) = 1$.*

Proof. Lemma 7.5.1 shows that for every $2 \times |\mathcal{V}| - 1$ consecutive iterations of Algorithm 3, there is a finite, non-zero probability that the algorithm discovers some global minimum.

Without loss of generality, let p be the lowest-valued probability with which Algorithm 3 discovers a global minimum of `cost` during any $2 \times |\mathcal{V}| - 1$ consecutive iterations.

Then the probability $P(i)$ of Algorithm 3 discovering some particular global minimum $\mathcal{M}$ during its first $i \times (2 \times |\mathcal{V}| - 1)$ loop iterations is given by:

$$P(i) = p + (1-p)p + (1-p)^2 p + \ldots + (1-p)^{(i-1)} p \tag{10}$$

$$= p \times \sum_{j=0}^{i-1} (1-p)^j \tag{11}$$

$$= p \times \left(\left(\sum_{j=0}^{i} (1-p)^j \right) - (1-p)^i \right) \tag{12}$$

Equation (10) gives our basic formula. For any j, the j^{th} term in the series provides the the probability that loop iterations $1 \ldots (j-1)$ did not discover a global minimum but the j^{th} loop iteration does.

Equation (11) restructures the series to clarify that it's a geometric series, and Equation (12) simplifies the upper bound of the series so that a standard formula may be applied later in Equation (16).

We now consider the limit of Equation (12) as $i \to \infty$:

$$\begin{aligned}
lim_{i\to\infty} P(i) &= lim_{i\to\infty}\left(p \times \left(\left(\sum_{j=0}^{i}(1-p)^j\right) - (1-p)^i\right)\right) \\
&= p \times lim_{i\to\infty}\left(\left(\sum_{j=0}^{i}(1-p)^j\right) - (1-p)^i\right) && (13) \\
&= p \times \left(\left(lim_{i\to\infty}\sum_{j=0}^{i}(1-p)^j\right) - \left(lim_{i\to\infty}(1-p)^i\right)\right) && (14) \\
&= p \times \left(\left(lim_{i\to\infty}\sum_{j=0}^{i}(1-p)^j\right) - 0\right) && (15) \\
&= p \times \left(\left(\frac{(1-p)^0}{1-(1-p)}\right) - 0\right) && (16) \\
&= 1 && (17)
\end{aligned}$$

Equation (13) simplifies by factoring p out of the limit expression.

Equation (14) separates terms in the limit expression to obtain two separate limits, and Equation (15) solves the second limit.

Equation (16) applies the standard formula for the limit of an infinite series of a geometric sequence, and Equation (17) simplifies, completing our proof. $\square$

CHAPTER 8
Proof of Concept

The optimization algorithms presented in this thesis were implemented to validate the correctness of the theoretical results, and to clarify the potential for this approach to optimizing the performance of task-parallel codes. The methodology and results are discussed below.

A model program, represented as an abstract DFG, was developed to generate and then sort a collection of pseudo-random floating-point numbers (see Section 8.1). Several alternative approaches were used optimize the DFG's task set:

- A program named `evolver1` , an implementation of the full-task-set optimization algorithm outlined in Algorithm 2 (page 37). See Section 8.2 for a more complete description of `evolver1` .
- A task set was selected based on the author's experience in developing multi-threaded software (see Subsection 8.3.3). This is to provide a baseline against which the learning algorithm's results are compared. The manual creation of a task set was tractable because the program being optimized was represented with a DFG having only 11 vertices (see Section 8.3).
- Two task sets representing extremes of the search space were manually chosen: the empty task set, and the task set in which each of the DFG's 11 vertices is assigned to its own separate task. Their performance was examined to confirm that the optimization problem treated by this thesis was not simply solved by trivial selection of one of these extrema.

8.1 Model Problem : Sorting an Array of Numbers

The model problem chosen is a program which sorts an array of floating-point number, as follows.

A DFG representation of this program is shown in Figure 4. For this work, we generate the sort program's DFG in terms of a single pre-runtime parameter, l. l gives the number of levels of mergesort vertices in the DFG. For a given value of l, this work produces a DFG with 2^l `randomize` vertices, 2^l `quicksort` vertices, and $2^l - 1$ `mergesort` vertices. The DFG is represented as a simple text file, and is produced by a program named `generate-DFG` .

Note that the value of l is only loosely related to the number of data to be sorted by the program at runtime. The number of data to be sorted is specified as a runtime parameter, whereas l is a pre-runtime parameter. Our only assumption is that the specified number of data to sort is an integer multiple of 2^l, to simplify the runtime logic governing the number of data produced by each `randomize` vertex.

The vertex types behave as follows.

- Each `randomize` vertex produces an array of pseudo-randomly generated floating-point numbers. The size of the array is provided as a runtime parameter.
- Each `quicksort` performs an in-place sorting of the array provided by its in-neighbor. The vertex passes a reference to that same array on to the vertex's out-neighbor.
- Each `mergesort` allocates a new array capable of storing the combined content of the vertex's two input arrays. It then uses a simple mergesort algorithm to merge the two (pre-sorted) input arrays, and provides a reference to that array to its out-neighbor.

This model problem was chosen for several reasons. Its simple, recursive structure per-

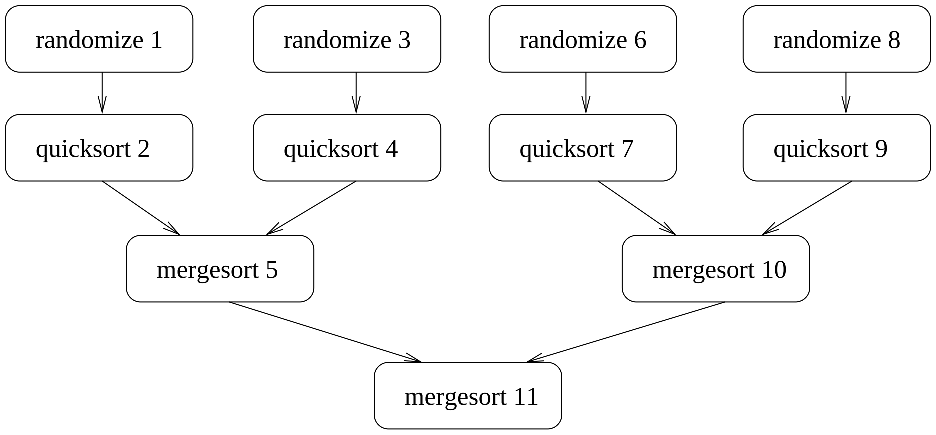


Figure 4: A generated DFG for the proof-of-concept. The number of mergesort levels is specified when the DFG is generated. The number within each box is its vertex number, a unique identifier for the vertex.

mits the easy generation of DFG's of vastly various sizes without altering the program's semantics. The regular structure also lends itself to intuitive parallelization by a human programmer, which simplifies the exercise of comparing the results produced by the proof-of-concept optimization algorithm to what a human programmer may be expected to produce.

With this approach, several parameters govern the static and runtime qualities of the sorting program:

- l. This is depth of the mergesort portion of the DFG, as described above.
- *Number of data to be sorted.* This is the number of floating-point numbers to be randomly generated and sorted.
- *Task set.* The collection of convex sets which forms the basis for Thread Building Block tasks in the resulting executable program.

8.2 Task-set Optimization Algorithm

`evolver1` implements a simple evolutionary algorithm whose structure is consistent with that of Algorithm 2 (page 37).

Note that Algorithm 2 is parameterized by which algorithm it uses to optimize a single convex set (the Z parameter). For this work, $Z =$ Algorithm 3 (see page 73). Additional details of `evolver1` are as follows.

8.2.1 Initialization

The initial generation is created by pseudo-randomly generating a user-specified number of task sets, `population_size`. A user-specified `probability_of_inclusion` parameter influences the content of each task in each task set of the initial population, as indicated in Algorithm 4.

Algorithm 4 Initialization Algorithm for Proof-of-Concept System

```
Function init : ($D = (\mathcal{V}, A)$, probability_of_inclusion, population_size)
                → $population$
 1: $population \leftarrow \emptyset$
 2: for $i = 1 \ldots$ population_size do
 3:   $task\_set \leftarrow \{\}$
 4:   for all $v \in \mathcal{V}$ do
 5:     if $random\_draw\_in\_range(0.0, 1.0) <$ probability_of_inclusion then
 6:       $task\_set \leftarrow task\_set \cup \{v\}$
 7:     end if
 8:   end for
 9:   $population \leftarrow population \cup \{task\_set\}$
10: end for
11: return $population$
```

8.2.2 Reproduction and Mutation

Each subsequent generation of task sets is formed as follows. `population_size` copies of the fastest-running task set from the previous generation are created. An additional

$$\left\lfloor \frac{\texttt{population\_size}}{3} \right\rfloor$$

copies are the fastest-running task set from *any* previous generation are created. These two groups are task sets are the basis for the subsequent generation. All of these task sets are then mutated as follows.

As an invocation of `evolver1` iterates from its first to its last evolved generation, a "temperature" value decreases linearly from 1.0 to 0.0. The temperature indicates the fractional number of tasks within each task set which are to be mutated during that iteration of the algorithm. This mechanism is provided to allow broad search of the entire solution space in early iterations of the algorithm, followed by more precise local refinement in the later stages so that locally optimum solutions are more likely to be be discovered.

Any task which is to be mutated is mutated using the convex-set-mutation approach described in Section 7.5.

8.2.3 Termination

The proof-of-concept evolutionary algorithm terminates after running for a user-specified number of iterations. A more sophisticated evolutionary algorithm might use convergence analysis, rather than a fixed number of iterations, to decide after which iteration to terminate execution.

8.3 Methodology

For this work, the computer used was a laptop computer with an Intel Core i7-Q820 microprocessor, with four CPU cores and a nominal clock speed of 1.73 GHz. The computer has 4 GB of RAM. The computer's operating system is the 64-bit Ubuntu version of Linux Mint 14. The compiler used is gcc 4.7.2. The version of the Thread Building Blocks library used is 4.0.

For each parallelization of the program, the resulting executable program was run three times to obtain the average running time. During these executions, the computer was running in the typical fashion, with one user logged into a desktop environment, and several desktop applications open but untouched by the user.

8.3.1 Problem Details

This experiment used a DFG with two mergesort levels ($l = 2$), as shown in Figure 4. This was based on several considerations, most notably that early prototyping showed no significant performance benefit to the optimized program by using values of l greater than 2.

All runs of the sort program generated and sorted 10 million floating-point numbers. This number was chosen to minimize the impact of other system activity on the sort program's measured running time, while keeping the sort program's running times brief enough to allow research to proceed at a reasonable pace.

8.3.2 Evolutionary Algorithm Parameters

Based on common practices for the evaluation of evolutionary algorithms, each run of this work's evolutionary algorithm ran for 50 generations. Each generation had $population\_size = 100$. As described in Subsection 8.2.2, this led to an effective population size of 133 in all generations after the first generation.

8.3.3 Manually Chosen Task Set

The manually designed task set for the DFG is

$$\{\{1,\ 2\},\ \{3,\ 4\},\ \{6,\ 7\},\ \{5\}\}$$

This task set is visually depicted in Figure 5f (page 89).

The following reasoning led to the task set in Subsection 8.3.3. The sort program was to run with 1×10^7 input data, and so each of the `randomize` vertices would likely require a quantity of CPU time which significantly exceeded the time required to launch a parallel task. Therefore it seemed profitable to in fact run all four `randomize` vertices concurrently. This is achieved by having three of those vertices run in separate child tasks, and the fourth `randomize` vertex run in the parent thread.

Task Set Name	Tasks	Best Time (Seconds)
Run 1	{ {9} {5 , 10} {6 , 7} {3 , 4} {11} {1 , 2} }	1.047
Run 2	{ {2 , 9} {3 , 4 , 6 , 7} {5} {10} }	1.315
Run 3	{ {2} {3 , 4} {8 , 9} {5 , 10} {11} {6 , 7} }	1.048
Run 4	{ {2 , 9} {3 , 4 , 7} {5} {10} {8} {6} }	1.315
Run 5	{ {2 , 9} {3 , 4 , 6 , 7} {5} }	1.318
Manually designed	{ {1 , 2} {3 , 4} {6 , 7} {5} }	1.387
Empty	{}	2.110
One task per vertex	{ {1} {2} . . . {11} }	1.383

Table 1: Average Running Times of a DFG with two mergesort levels (shown in Figure 4). The program generated for each of the listed task sets was run three times, and each run generated and sorted 1×10^7 floating-point numbers. The non-degenerate task sets from this table are graphically depicted in Figure 5.

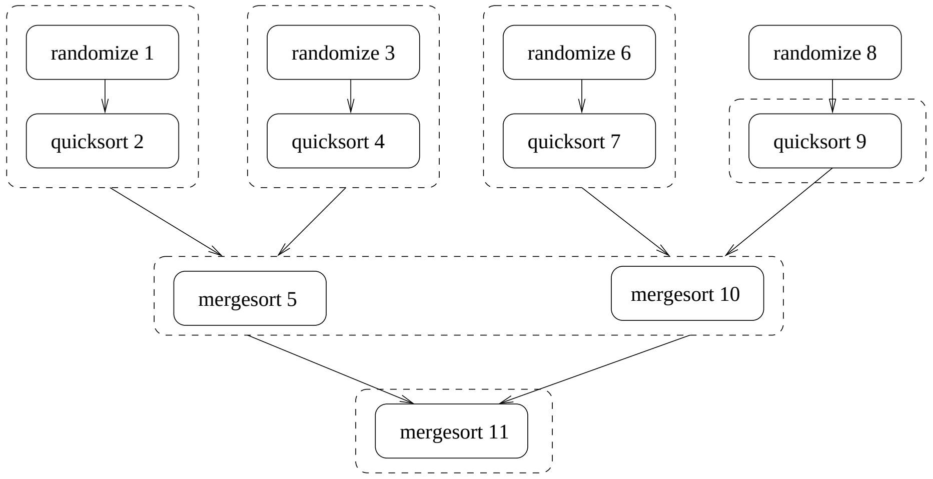
By similar reasoning, all four of the `quicksort` appeared suitable for running in parallel on separate CPU cores. Furthermore, each `quicksort` could be run in the same task as the `randomize` vertex which supplied its input data. This grouping appeared sensible because it might reduce the overhead of launching the `quicksort` vertices in new tasks, and there appeared to be no way in which it could cause additional delay in the start of execution of a `quicksort` vertex.

Each of the two non-DFG-sink `mergesort` vertices merges together two arrays of 2.5×10^6 floating-point numbers. Intuitively, this large CPU burden for each `mergesort` vertex appears to make a strong case for running the `mergesort` vertices in parallel if at all possible. For this reason that the task set {5} was specified to be in a child task and vertex 6 was left to run separately in the parent thread.

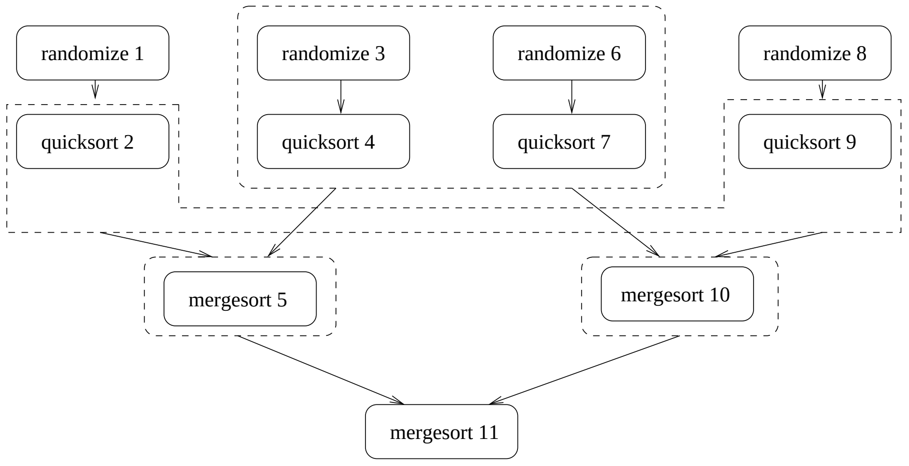
8.4 Proof-of-Concept Results

Each run of the stochastic optimization algorithms took approximately 12 hours.

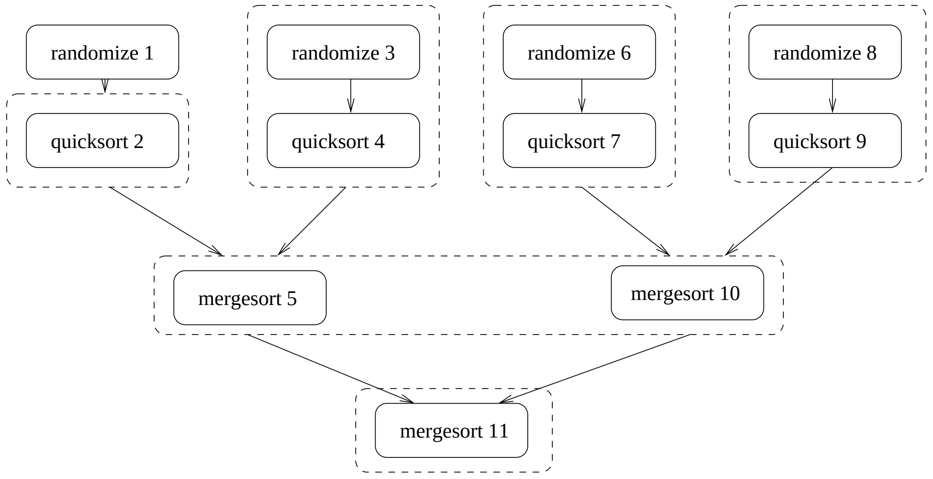
Table 1 presents the running times obtained from each of the task sets considered by this proof of concept. Six of the task sets are shown in Figure 5. (The two degenerate



(a) Stochastic Run 1

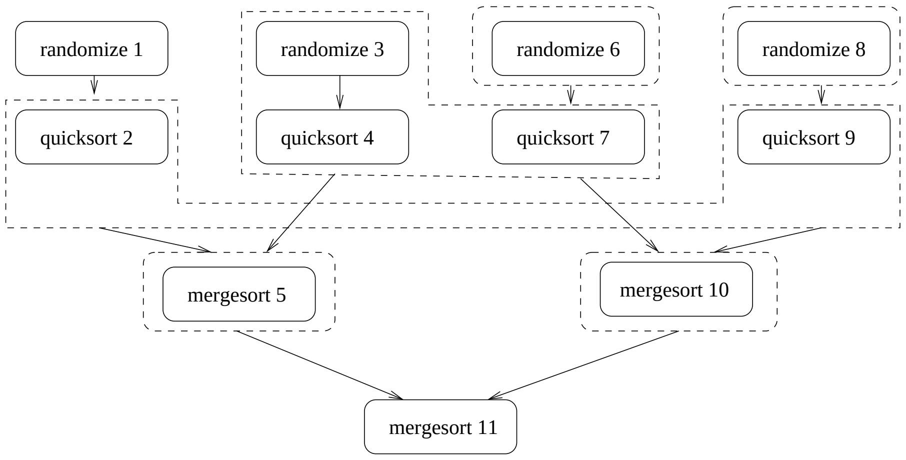


(b) Stochastic Run 2

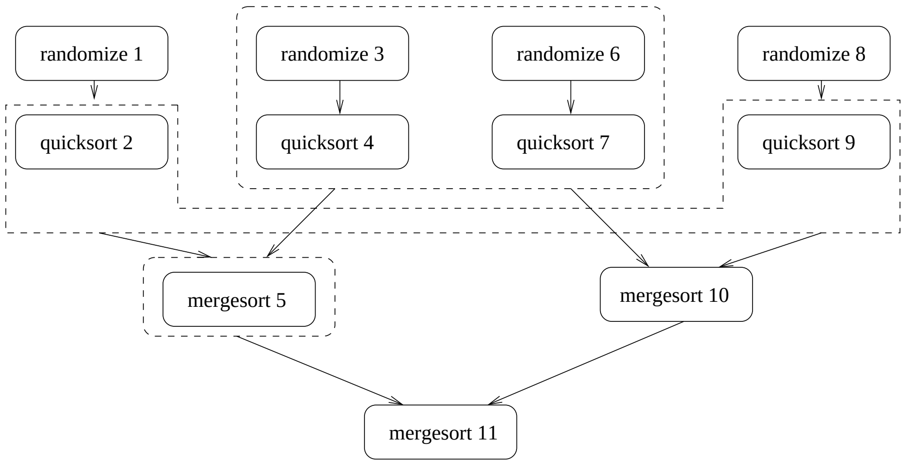


(c) Stochastic Run 3

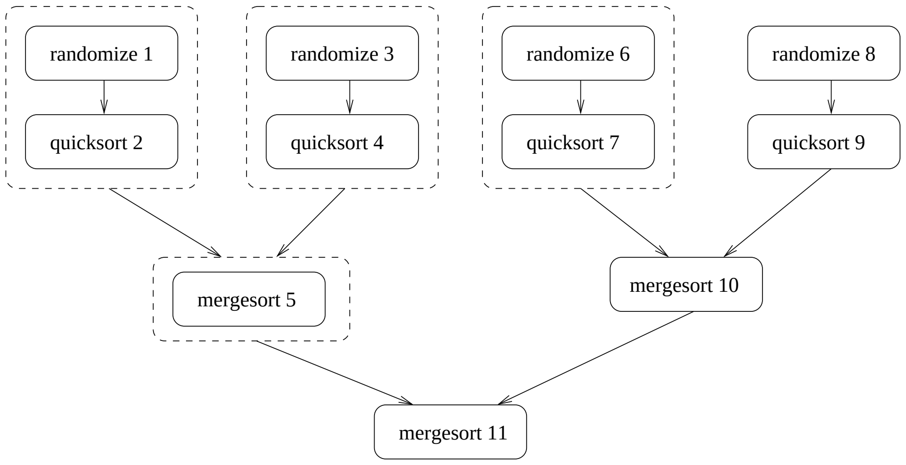
Figure 5: Visual depictions non-degenerate task sets in Table 1.



(d) Stochastic Run 4



(e) Stochastic Run 5



(f) Manually Designed Task Set

Figure 5: Visual depictions non-degenerate task sets in Table 1 (continued).

task sets are not shown.)

From Table 1 we see that all of the stochastically optimized task sets outperformed the manually designed one. The fastest stochastically optimized task set (Run 1) completed its work 32.2% faster than did the manually developed task set ($1.384 \div 1.047$ seconds).

Figure 5 (pages 88-89) shows shows the comparative results of the five stochastic optimization runs, on a per-generation basis.

We see that the stochastically developed task sets fall into two groups: a slower group (average runnings times of 1.315, 1.315, and 1.318 seconds), and a much faster group (average running times of 1.047 and 1.048 seconds).

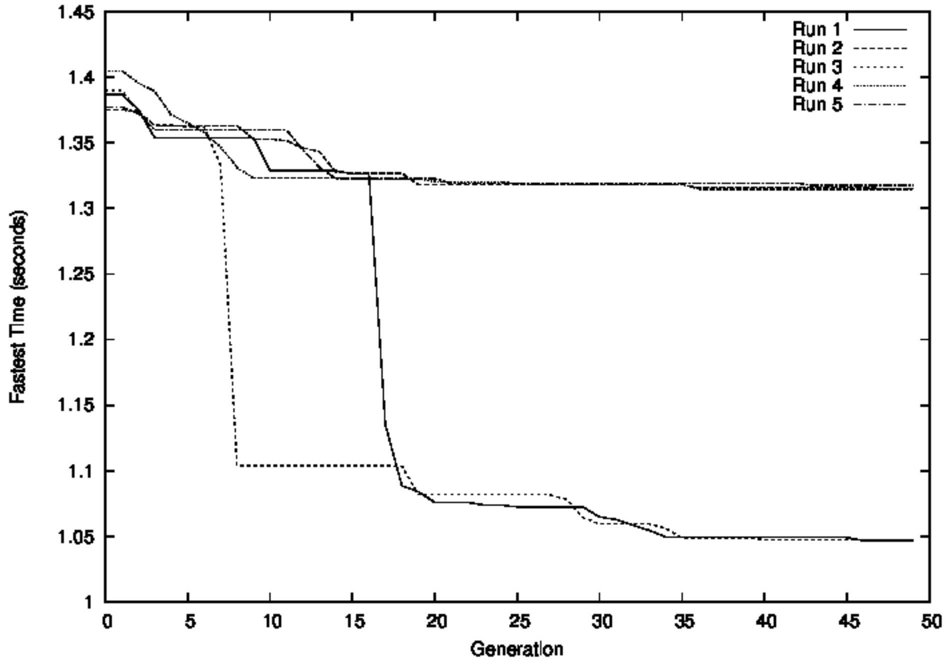
The best-performing stochastic optimization runs (Run 1 and Run 3) produced remarkably similar task sets. The worse-performing stochastic optimization runs (Runs 2, 4, and 5) also significantly similar task sets as each other, however with more apparent variation.

8.5 Limitations of Proof-of-Concept

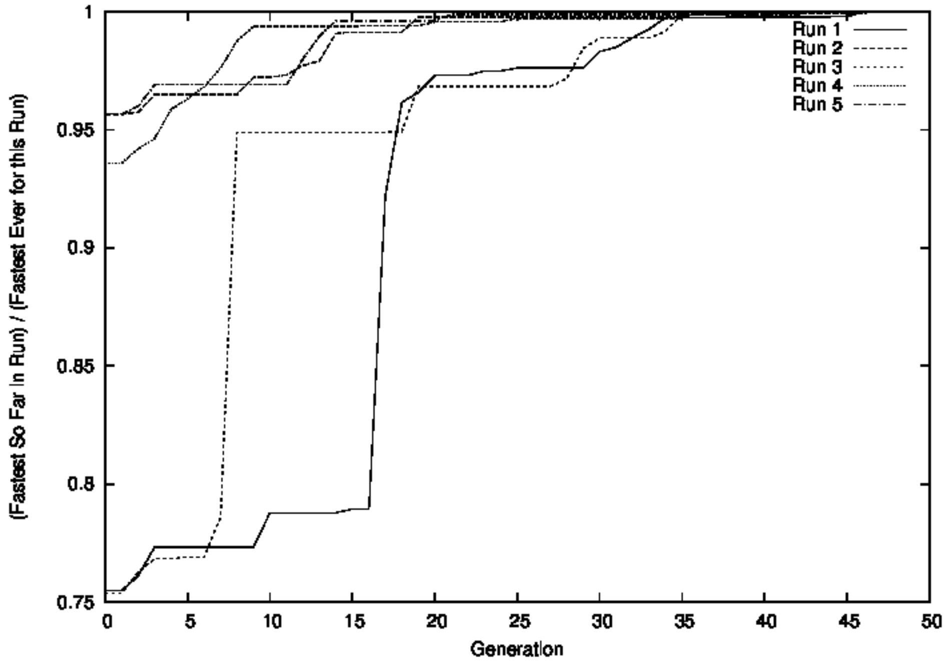
This proof-of-concept is provided to help validate the algorithms developed in this thesis. Care must be take to not draw overly broad conclusions from this exercise.

In particular, this exercise has not established that either the evolutionary algorithm or the human-crafted parallelization offers the best possible performance for that kind of endeavor. A programmer with more skill or luck than this author could perhaps produce a parallelization of the test programmer which outperformed any that the evolutionary algorithm is likely to discover. Conversely, a better-tuned evolutionary algorithm could perhaps outperform a human in nearly all circumstances.

Another limitation of this exercise is that the selected DFG was quite small, with only 11 vertices, and each those vertices is computationally intensive. Given the good efficiency



(a) Comparative absolute performances of the five runs of the stochastic optimization algorithm.



(b) Relative pace, in generations, at which each run of the stochastic optimization algorithm discovered its final result.

Figure 6: Performance Comparison of Five Auto-parallelization Runs

of the Thread Building Blocks library, this means that even a full parallelization of this DFG, in which each vertex is assigned to a separate task, will have very low book-keeping overhead for the tasks. Therefore this proof-of-concept does little to explore how this thesis' algorithms would perform on a very large DFG with shorter-running vertices.

This search algorithm initializes, grows, and shrinks convex sets using only algorithms based on topological sort, not the alternative algorithms based on predecessor and successor sets. This limitation was deemed acceptable because the primary goal of this proof-of-concept was to examine both the quality of the solutions produced by the task set optimization algorithm, and speed with which it obtained those solutions. Any difference in performance between these two groups of support algorithms were expected to be negligible for this purpose, in particular because the target program's DFG contained only 11 vertices.

8.6 Discussion

The machine-learning algorithm developed in this work outperformed what appears to be a sensibly chosen manual tuning of the same problem by 32%. This result was obtained from two of the five stochastic optimization runs, which collectively required approximately 60 hours of computer time.

This appears to validate several assumptions of this present research. The first is that on modern computing hardware, even a seasoned programmer's expectation about the ideal parallelization of a simple program isn't necessarily accurate.

The second validated assumption is that, at least in this one test case, the evolutionary algorithm can uncover a good parallelization of the subject program in a relatively small amount of time.

The solution space over which this proof of concept's evolutionary algorithm ranges is all task sets containing six or fewer convex sets. However, the deconfliction process (see

Appendix E) maps that space onto the smaller space of task sets which will not cause the parent DFG to contain a cycle. Having the evolutionary algorithm range over a search space larger than the space of acceptable answers may reduce its efficiency.

CHAPTER 9

Conclusions and Future Work

9.1 Central Conjecture Proven

This thesis proves the central conjecture that a single convex set can be evolved in a per-iteration efficient manner, such that as the number of iterations approaches infinity, the probability of all convex sets being explored approaches one.

The primary motivation for this work was to advance the state of the art in machine-learned task parallelism, so that parallel programs can run faster and be developed with less human labor. The proof-of-concept developed in this work shows that for the one subject program studied, the resulting parallel program was substantially faster (32%) than the version produced by a human programmer. This suggests that automatic task parallelization may continue to be a fruitful area for future research.

9.2 Alternative Search Spaces

Under the execution model assumptions made in this present work, the true optimization space is all valid task sets, not merely all valid convex sets. Either of two definitions of task-set validity is appropriate, depending on the execution model assumed: In the weak definition of task-set validity, a task set is valid merely if each contained task is a convex set, and no two tasks contain the same vertex. A more demanding definition of task-set-validity, based in this work's assumed execution models, is that it must also be possible to replace each task in the task set with a pair of `SPAWN` and `WAIT` vertices without inducing cycles in the parent DFG.

While this thesis shows that a single convex set can be efficiently evolved to cover the space of all convex sets, it does establish the existence of a search algorithm which covers precisely the space of all valid task sets, using either of the stated definitions of validity. The development of algorithms covering either of those valid task-set spaces, in a manner that is per-iteration efficient and which will cover the entire search space given enough iterations, may lead to better-performing machine learning systems than the one demonstrated in this work's proof-of-concept.

9.3 Enumeration of Search Spaces

[27] and Bang-Jensen and Gutin [1, section 17.2.3] present an efficient algorithm for enumerating all convex sets of any directed acyclic graph. Such an algorithm may be useful in studying the rate at which a stochastic search algorithm visits the full search space.

To this author's knowledge, no similar algorithm has been discovered for enumerating either of the valid task-set search spaces described above. The development of such algorithms may be a useful area of future endeavor.

LIST OF REFERENCES

[1] J. Bang-Jensen and G. Gutin, *Digraphs: Theory, Algorithms and Applications*, 2nd ed. Springer, 2009.

[2] V. Sarkar and J. Hennessy, "Partitioning parallel programs for macro-dataflow," in *LFP '86 Proceedings of the 1986 ACM conference on LISP and functional programming.* Assoc. of Computing Machinery, 1986.

[3] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press Cambridge, MA, USA, 1989.

[4] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, C. Williams, M. O'Boyle, P. Barnard, E. Ashton, E. Courtois, and F. Bodin, "Milepost gcc: machine learning based research compiler," in *Proceedings of the GCC Developers' Summit*, July 2008. [Online]. Available: http://hal.inria.fr/inria-00294704/fr/

[5] M. Stephenson, S. Amarasinghe, M. Martin, and U. O'Reilly, "Meta Optimization: Improving Compiler Heuristics with Machine Learning," *ACM SIGPLAN Notices*, vol. 38, no. 5, p. 90, 2003.

[6] G. G. Fursin, M. F. P. O'Boyle, and P. M. W. Knijnenburg, "Evaluating Iterative Compilation," in *Languages and Compilers for Parallel Computing, 15th Workshop, LCPC 2002*, ser. LNCS, no. 2481. Springer, 2005, pp. 362–376.

[7] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "Acme: adaptive compilation made efficient," in *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems.* New York, NY, USA: ACM, 2005, pp. 69–77.

[8] M. Frigo and S. Johnson, "FFTW: An adaptive software architecture for the FFT," in *IEEE International Conference on Acoustics Speech and Signal Processing*, vol. 3. Citeseer, 1998.

[9] M. Frigo and S. Johnson, "The design and implementation of FFTW 3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[10] M. Pueschel, J. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. Johnson, "Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Alogorithms," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, 2004.

[11] M. Olszewski and M. Voss, "An install-time system for the automatic generation of optimized parallel sorting algorithms," in *Proceedings of PDPTA'04: The International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2004.

[12] C. Chan, J. Ansel, Y. L. Wong, S. Amarasinghe, and A. Edelman, "Autotuning multigrid with petabricks," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.* New York, NY, USA: ACM, 2009, pp. 1–12.

[13] W. Najjar, L. Roh, and A. Wim Böhm, "An evaluation of medium-grain dataflow code," *International Journal of Parallel Programming*, vol. 22, no. 3, pp. 209–242, 1994.

[14] L. Roh, W. A. Najjar, and A. P. W. Böhm, "Generation and quantitative evaluation of dataflow clusters," in *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture.* New York, NY, USA: ACM, 1993, pp. 159–168.

[15] A. Smyk and M. Tudruj, "Genetic optimization of parallel fdtd computations," in *ISPDC '08: Proceedings of the 2008 International Symposium on Parallel and Distributed Computing.* Washington, DC, USA: IEEE Computer Society, 2008, pp. 155–161.

[16] E. Mohr, D. A. Kranz, and J. R. H. Halstead, "Lazy task creation: a technique for increasing the granularity of parallel programs," in *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming.* New York, NY, USA: ACM, 1990, pp. 185–197.

[17] V. Sarkar and D. Cann, "Posc—a partitioning and optimizing sisal compiler," in *ICS '90: Proceedings of the 4th international conference on Supercomputing.* New York, NY, USA: ACM, 1990, pp. 148–164.

[18] S. Rus, M. Pennings, and L. Rauchwerger, "Sensitivity analysis for automatic parallelization on multi-cores," in *ICS '07: Proceedings of the 21st annual international conference on Supercomputing.* New York, NY, USA: ACM, 2007, pp. 263–273.

[19] L. Huelsbergen, J. R. Larus, and A. Aiken, "Using the run-time sizes of data structures to guide parallel-thread creation," in *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming.* New York, NY, USA: ACM, 1994, pp. 79–90.

[20] A. Duran, J. Corbalán, and E. Ayguadé, "An adaptive cut-off for task parallelism," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing.* Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.

[21] L. Prechelt and S. U. Hänßgen, "Efficient parallel execution of irregular recursive programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 2, pp. 167–178, 2002.

[22] D. C. Cann, J. Feo, and R. Oldehoeft, "A report on the SISAL language project," *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, pp. 349–366, 1990.

[23] P. Beard, "An implementation of SISAL for distributed-memory architectures," Master's thesis, University of California, Davis, 1995.

[24] J. E. P. Sanchez and D. Trystram, "A new genetic convex clustering algorithm for parallel time minimization with large communication delays." in *Proceedings of the International Conference ParCo 2005*, ser. John von Neumann Institute for Computing Series, G. R. Joubert, W. E. Nagel, F. J. Peters, O. G. Plata, P. Tirado, and E. L. Zapata, Eds., vol. 33. Central Institute for Applied Mathematics, Jülich, Germany, 2005, pp. 709–716.

[25] J. Pecero and P. Bouvry, "An improved genetic algorithm for efficient scheduling on distributed memory parallel systems," in *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications - AICCSA 2010.* IEEE Computer Society, 2010.

[26] R. Andersen, D. F. Gleich, and V. Mirrokni, "Overlapping clusters for distributed computation," in *Proceedings of the fifth ACM international conference on Web search and data mining - WSDM '12.* Assoc. of Computing Machinery, 2012.

[27] P. Balister, S. Gerke, G. Gutin, A. Johnstone, J. Reddington, E. Scott, A. Soleimanfallah, and A. Yeo, "Algorithms for generating convex sets in acyclic digraphs," 2008.

[28] G. Rudolph, "Convergence analysis of canonical genetic algorithms," *IEEE Transactions on Neural Networks*, vol. 5, pp. 96–101, 1994.

[29] G. Rudolph, "Convergence of evolutionary algorithms in general search spaces," in *In Proceedings of the Third IEEE Conference on Evolutionary Computation.* IEEE Press, Piscataway, NJ, USA, 1996, pp. 50–54.

[30] R. A. Iannucci, "A dataflow / von neumann hybrid architecture," Ph.D. dissertation, Massachusetts Inst. of Technology, 1988.

[31] Y. Leung, Y. Gao, and Z. Xu, "Degree of population diversity - a perspective on premature convergence in genetic algorithms and its markov chain analysis," *IEEE Transactions on Neural Networks*, vol. 8, pp. 1165–1176, 1997.

[32] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[33] D. Bertsimas and J. Tsitsiklis, "Simluated annealing," *Statistical Science*, vol. 8, no. 1, pp. 10–15, 1993.

APPENDIX A

Algorithm Assumptions and Primitive Operations

In this appendix we provide asymptotic running times, and when appropriate the randomness properties, of primitive operations and algorithms used within this work. The operations and algorithms described below are not the ones of primary interest in this work. They are called by this work's more interesting algorithms, and are presented here to facilitate the analysis of the running times and randomness properties of the algorithms which call them.

Some primitive operations on sequences backed by linked lists have running times that are proportional to the current size of the sequence, as opposed to the sequence's maximum possible size. However, for the sake of simplified worst-case running-time analyses, for each we primitive operation we indicate a worst-case asymptotic running time, typically in terms of $|\mathcal{V}|$, the order of the overall DAG being considered.

The remainder of this appendix is structured as follows.

Table A.2 summarizes the asymptotic running times the operations and algorithms presented in this appendix.

Appendix A.1 provides an overview of the role of random primitive operations in this work.

Appendices A.2-A.8 describe primitive operations on the collections underlying this work's algorithms.

Appendix A.9 states additional assumptions used in this work's analysis of algorithm asymptotic running-times, and discusses the use of $|\mathcal{V}|$ as the non-constant value in the stated bounds.

Operation	**Time**
`random_integer` (a, b)	$O(1)$
`symset_random_elem` (L)	$O(1)$
`symset_init` (S)	$O(1)$
`symset_add` (S, t)	$O(1)$
`dag_rand_topo_sort` (D)	$O(\|\mathcal{V}\|^2)$
`dag_transitive_closure` (D)	$O(\|\mathcal{V}\|^{2.376})$
`vset_clear` ($\mathcal{X}$)	$O(\|\mathcal{V}\|)$
`vset_init_by_vertex` ($\mathcal{X}, v$)	$O(\|\mathcal{V}\|)$
`vset_clone` ($\mathcal{X}$)	$O(\|\mathcal{V}\|)$
`vset_union` ($\mathcal{X}, \mathcal{Y}$)	$O(\|\mathcal{V}\|)$
`vset_isect` ($\mathcal{X}, \mathcal{Y}$)	$O(\|\mathcal{V}\|)$
`vset_diff` ($\mathcal{X}, \mathcal{Y}$)	$O(\|\mathcal{V}\|)$
`vset_is_member` ($\mathcal{X}, v$)	$O(1)$
`vset_is_subset` ($\mathcal{X}, \mathcal{Y}$)	$O(\|\mathcal{V}\|)$
`vset_is_equal` ($\mathcal{X}, \mathcal{Y}$)	$O(\|\mathcal{V}\|)$
`vset_random_subset` ($\mathcal{X}, n$)	$O(\|\mathcal{V}\|)$
`vset_iterate` ($\mathcal{X}$)	$O(\|\mathcal{V}\|)$
`vset_add` ($\mathcal{X}, v$)	$O(1)$
`aset_clear` (B)	$O(\|\mathcal{V}\|^2)$
`aset_add` (B, c)	$O(1)$
`aset_del` (B, c)	$O(1)$
`aset_iterate` (B)	$O(\|\mathcal{V}\|^2)$
`aset_iterate` (B)	$O(\|\mathcal{V}\|^2)$
`aset_in_nbors` ($B, \mathcal{X}$)	$O(\|\mathcal{V}\|^2)$
`aset_out_nbors` ($B, \mathcal{X}$)	$O(\|\mathcal{V}\|^2)$
`vseq_random_slice` ($\vec{X}, n$)	$O(\|\mathcal{V}\|)$
`vseq_slice` ($\vec{X}, i, j$)	$O(\|\mathcal{V}\|)$
`vseq_find` ($\vec{X}, v$)	$O(\|\mathcal{V}\|)$
`vseq_find_bounds` ($\vec{X}, \mathcal{Y}$)	$O(\|\mathcal{V}\|^2)$
`vseq_to_vset` ($\vec{X}$)	$O(\|\mathcal{V}\|)$
`vss_append` ($\overline{\mathcal{T}}, \mathcal{V}$)	$O(1)$
`vss_concat` ($\overline{\mathcal{T}}, \overline{\mathcal{U}}$)	$O(1)$
`vss_reverse` ($\overline{\mathcal{T}}$)	$O(\|\overline{\mathcal{T}}\|)$
`tc_vtx_pred` ($TC(D), x$)	$O(\|\mathcal{V}\|)$
`tc_vtx_succ` ($TC(D), x$)	$O(\|\mathcal{V}\|)$
`tc_vset_pred` ($TC(D), \mathcal{X}$)	$O(\|\mathcal{V}\|^2)$
`tc_vset_succ` ($TC(D), \mathcal{X}$)	$O(\|\mathcal{V}\|^2)$

Table A.2: Summary of Primitive Operations

A.1 Primitive Random Operations

Primitive random operations particular to some primitive data structure presented in the appropriate subsection below.

A critical property of all primitive random operations used in this work is that each valid outcome of the operation has a finite, positive probability each time the operation is performed. From this foundation we establish certain randomness properties in the algorithms presented in Appendix C and Section 7.5.

We present here our one primitive random on integers. All random operations on primitive data structures are described in appropriate subsection below.

`random_integer` : (**[in]** a : integer, **[in]** b : integer) $\rightarrow$ integer

This returns an integer from the interval $[a, b]$. For any integer $i \in [a, b]$, there is a non-zero probability that this operation returns i. This operation requires time $O(1)$.

A.2 Primitive Operations on Literal Sets

A literal set is a set of zero-or-more pseudocode literal values. In particular, the set {`GROW`, `SHRINK`, `SWAP`} in Algorithm 3.

We assume each instance of this kind of set to be a random-access array of Boolean values, having the following operations.

`symset_random_elem` : (**[in]** S : set of symbols) $\rightarrow$ symbol

This returns one symbol from L. For any symbol $s \in S$, there is a finite, positive probability that this operation returns s. This operation requires time $O(1)S$.

`symset_init` : (**[in]** S : literal set of symbol values) $\rightarrow$ literal value

This is the type constructor for symbol sets using literal values. This operation requires time $O(1)$.

`symset_add` : ([**in,out**] S : set of symbols, [**in**] t : symbol)

This modifies S to be the set $S \cup \{t\}$. This operation requires time $O(1)$.

A.3 Primitive Operations on a DAG

We assume for this work that the order (i.e., number of vertices) in DAG are known *a priori*, and that all data structures are pre-allocated or whose allocation requires a negligible amount of running-time.

Let $D = (\mathcal{V}, A)$ be the DAG. Each vertex in the DAG is named by a distinct number in the range $[1, |\mathcal{V}|]$. Unless stated otherwise, any mention of D, $\mathcal{V}$ or A in this appendix refers to the DAG over which the primitive operations work.

D is represented by a random-access $|\mathcal{V}| \times |\mathcal{V}|$ array of Boolean values, where the value of element (x, y) indicates the presence or absence of an arc from vertex x to vertex y.

`dag_rand_topo_sort` : ([**in**] D : DAG) $\rightarrow$ vertex sequence

This returns a topological sort of D. For any topological sort $\vec{Q}$ of D, there is a finite, positive probability that this operation returns $\vec{Q}$ each time this operation executes.

As shown in [1, Thm. 2.1.4] and elsewhere, some arbitrary topological sorting for a DAG $D = (\mathcal{V}, A)$ may be deterministically calculated in time $O(|\mathcal{V}| + |A|)$. We assume but do not prove that a non-deterministic algorithm exists and has running time $O(|\mathcal{V}| + |A|)$.

In Section 2.3 we show that $|A|$ is $O(|\mathcal{V}|^2)$, giving us a simplified running-time of $O(|\mathcal{V}|^2)$ for this operation.

`dag_transitive_closure` : ([**in**] D : DAG) $\rightarrow$ (vertex set, arc set)

This returns $TC(D)$, the transitive closure of D, as described in Section 2.7.

A.4 Primitive Operations on a Set of Vertices

A vertex set is represented as a random-access vector of $O(|\mathcal{V}|)$ Boolean values, with the following operations.

`vset_clear` : ([**out**] $\mathcal{X}$: vertex set)

Modifies $\mathcal{X}$ to have the empty-set value. The running time of this operation is $O(|\mathcal{V}|)$.

`vset_init_by_vertex` : ([**out**] $\mathcal{X}$: vertex set, [**in**] v : vertex)

Modifies $\mathcal{X}$ to have precisely the value $\{v\}$. The running time of this operation is $O(|\mathcal{V}|)$.

`vset_clone` : ([**in**] $\mathcal{X}$: vertex set) $\rightarrow$ vertex set

Returns an independent copy of $\mathcal{X}$. Later modifications to $\mathcal{X}$ are not visible in the clone returned by this function, and *vice versa*. Assuming that the domains of $\mathcal{X}$ and $\mathcal{Y}$ are the subsets of $\mathcal{V}$, the running time of this operation is $O(|\mathcal{V}|)$.

`vset_union` : ([**in**] $\mathcal{X}$: vertex set, [**in**] $\mathcal{Y}$: vertex set) $\rightarrow$ vertex set

Returns the set $\mathcal{X} \cup \mathcal{Y}$. Assuming that the domains of $\mathcal{X}$ and $\mathcal{Y}$ are the subsets of $\mathcal{V}$, the running time of this operation is $O(|\mathcal{V}|)$.

`vset_isect` : ([**in**] $\mathcal{X}$: vertex set, [**in**] $\mathcal{Y}$: vertex set) $\rightarrow$ vertex set

Returns the set $\mathcal{X} \cap \mathcal{Y}$. Assuming that the domains of $\mathcal{X}$ and $\mathcal{Y}$ are the subsets of $\mathcal{V}$, the running time of this operation is $O(|\mathcal{V}|)$.

`vset_diff` : ([**in**] $\mathcal{X}$: vertex set, [**in**] $\mathcal{Y}$: vertex set) $\rightarrow$ vertex set

Returns the set $\mathcal{X} \setminus \mathcal{Y}$. Assuming that the domains of $\mathcal{X}$ and $\mathcal{Y}$ are the subsets of $\mathcal{V}$, the running time of this operation is $O(|\mathcal{V}|)$.

`vset_is_member` : ([**in**] $\mathcal{X}$: vertex set, [**in**] v : vertex) $\rightarrow$ Boolean

Returns **true** if $v \in \mathcal{X}$, and **false** if not. The running time of this operation is $O(1)$.

`vset_is_subset` : ([**in**] $\mathcal{X}$: vertex set, [**in**] $\mathcal{Y}$: vertex set) $\rightarrow$ Boolean

Returns **true** if $\mathcal{X} \subseteq \mathcal{Y}$, and **false** if not. The running time of this operation is $O(|\mathcal{V}|)$.

`vset_is_equal` : ([**in**] $\mathcal{X}$: vertex set, [**in**] $\mathcal{Y}$: vertex set) $\rightarrow$ Boolean

Returns **true** if $\mathcal{X} = \mathcal{Y}$, and **false** if not. The running time of this operation is $O(|\mathcal{V}|)$.

`vset_random_subset` : ([**in**] $\mathcal{X}$: vertex set, [**in**] n : integer) $\rightarrow$ vertex set

This operation assumes $1 \leq n \leq |\mathcal{X}|$, and returns some subset of $\mathcal{X}$ with order n. For every subset $\mathcal{Y}$ of $\mathcal{X}$, where $|\mathcal{Y}| = n$, there is a finite, positive probability that $\mathcal{Y}$ is returned by this operation. The running time of this operation is $O(|\mathcal{V}|)$.

`vset_iterate` : ([**in**] $\mathcal{X}$: vertex set)

This is a pseudo-operation which generates each vertex in $\mathcal{X}$ for use in loop iteration. We document it here to facilitate running-time analyses of "for all"-style loops over vertex sets. This pseudo-operation requires time $O(|\mathcal{V}|)$ to emit all vertices of $\mathcal{X}$ over time.

`vset_add` : ([**in,out**] $\mathcal{X}$: vertex set, [**in**] v : vertex)

Modifies $\mathcal{X}$ to be the set $\mathcal{X} \cup \{v\}$. This operation requires time $O(1)$.

A.5 Primitive Operations on a Set of Arcs

For simplicity we assume that a set of arcs has the same representation as a DAG: A set of arcs potentially connecting any two vertices in some vertex set $\mathcal{V}$ is represented as a $|\mathcal{V}| \times |\mathcal{V}|$ array of Boolean values.

`aset_clear` : ([**out**] B : arc set)

Modifies B to have the empty-set value. The running time of this operation is $O(|\mathcal{V}|^2)$.

`aset_add` : ([**in,out**] B : arc set, [**in**] c : arc)

Modifies B to be the set $B \cup \{c\}$. This operation has running time $O(1)$.

`aset_del` : ([**in,out**] B : arc set, [**in**] c : arc)

Modifies B to be the set $B \setminus \{c\}$. This operation has running time $O(1)$.

`aset_iterate` : ([**in**] B : arc set)

This pseudo-operation generates each arc in B for use in loop iteration in some unspecified order. We document it here to facilitate running-time analyses of "for all"-style loops over arc sets. This pseudo-operation requires time $O(|\mathcal{V}|^2)$ to emit all vertices of B over all iterations occurring during one activation of the loop.

`aset_in_nbors` : $([\textbf{in}]\ B : \text{arc set}, [\textbf{in}]\ \mathcal{X} : \text{vertex set})$
Returns the in-neighbors of $\mathcal{X}$: $\{v \mid ((v, \mathcal{X}) \in B) \wedge (v \notin \mathcal{B})\}$. This operation has running time $O(|\mathcal{V}|^2)$.

`aset_out_nbors` : $([\textbf{in}]\ B : \text{arc set}, [\textbf{in}]\ \mathcal{X} : \text{vertex set})$
Returns the out-neighbors of $\mathcal{X}$: $\{v \mid ((\mathcal{X}, v) \in B) \wedge (v \notin \mathcal{B})\}$. This operation has running time $O(|\mathcal{V}|^2)$.

A.6 Primitive Operations on a Sequence of Vertices

We assume that each vertex $\mathcal{V}$ is unambiguously identified by an integer, and that a vertex sequence is represented as a linked list of these integers. We further assume that allocation of storage for linked list elements requires negligible running time.

No vertex sequence manipulated by this work's algorithms contain multiple occurrences of the same vertex. (This is ultimately stems from this work's focus on *acyclic* graphs.) Therefore, for the sake of asymptotic running-time analyses, we assume that the length of a given vertex sequences is $O(|\mathcal{V}|)$.

The primitive operations on this type are as follows.

`vseq_random_slice` :

$$([\textbf{in}]\ \vec{X} : \text{vertex sequence}, [\textbf{in}]\ n : \text{integer}) \rightarrow \text{vertex sequence}$$

This operation assumes $1 \leq n \leq |\mathcal{X}|$, and returns some contiguous subsequence of $\vec{X}$ of length n. Let $\vec{Y}$ be any contiguous subsequence of $\vec{X}$ with length n. Then there is a finite, positive probability that this operation returns $\vec{Y}$. This operation requires time

$O(|\mathcal{V}|)$.

`vseq_slice` :

$$([\textbf{in}]\ \vec{X} : \text{vertex sequence}, [\textbf{in}]\ i : \text{integer}, [\textbf{in}]\ j : \text{integer}) \to \text{vertex sequence}$$

This operation assumes $1 \leq i \leq j \leq |\mathcal{X}|$. It returns the subsequence $\vec{X}[\![i \ldots j]\!]$. This operation requires time $O(|\mathcal{V}|)$.

`vseq_find` :

$$([\textbf{in}]\ \vec{X} : \text{vertex sequence}, [\textbf{in}]\ v : \text{vertex}) \to integer$$

This operation assumes that $v \in \mathcal{Y}$. It returns the index into $\vec{X}$ at which v is located. This operation requires time $O(|\mathcal{V}|)$.

`vseq_find_bounds` :

$$([\textbf{in}]\ \vec{X} : \text{vertex sequence}, [\textbf{in}]\ \mathcal{Y} : \text{vertex set}) \to (integer, integer)$$

This operation assumes that $\mathcal{X} \cap \mathcal{Y} \neq \emptyset$. It returns $(first, last)$, where $first$ and $last$ are, respectively, the lowest- and highest-numbered indices into $\vec{X}$ containing any member of $\mathcal{Y}$. This operation requires time $O(|\mathcal{V}|^2)$.

`vseq_to_vset` :

$$([\textbf{in}]\ \vec{X} : \text{vertex sequence}) \to vertexset$$

This operation returns a vertex set with precisely the same vertices as those comprising $\vec{X}$. This operation requires time $O(|\mathcal{V}|)$.

A.7 Primitive Operations on a Sequence of Vertex Sets

We assume that a sequence of vertex sets is represented as a linked list of references to existing vertex set objects, and that allocating nodes in the linked list requires negligible time. The primitive operations on this type are as follows. A sequence of vertex sets is implicitly empty until modified.

`vss_append` :

([**in,out**] $\overline{\mathcal{T}}$: sequence of vertex sets, [**in**] $\mathcal{V}$: vertex set)

Modifies $\overline{\mathcal{T}}$ by appending a reference to $\mathcal{V}$. This operation has running time $O(1)$.

`vss_concat` : (

[**in,out**] $\overline{\mathcal{T}}$: sequence of vertex sets,

[**in,out**] $\overline{\mathcal{U}}$: sequence of vertex sets)

$\rightarrow$ sequence of vertex sets

This operation modifies $\overline{\mathcal{T}}$ to have the value $[\overline{\mathcal{T}}, \overline{\mathcal{U}}]$, and modifies $\overline{\mathcal{U}}$ to be the empty sequence. This operation requires time $O(1)$.

`vss_reverse` :

[**in**] $\overline{\mathcal{T}}$: sequence of vertex sets, $\rightarrow$ sequence of vertex sets

This returns a copy of $\overline{\mathcal{T}}$ in which the order of the elements has been reversed. This operation requires time $O(|\overline{\mathcal{T}}|)$.

A.8 Primitive Operations on the Transitive Closure of a DAG

We assume the following representation for the transitive closure of a DAG. Suppose $D = (\mathcal{V}, A)$ is a DAG with transitive closure $TC(D) = (\mathcal{V}, A_{TC})$.

The transitive closure has two components: a vertex set and a vertex adjacency matrix. The D and $TC(D)$ have the same vertex set, and we assume that $TC(D)$ merely holds a reference to the vertex set of D.

Both A_{TC} and A are $|\mathcal{V}| \times |\mathcal{V}|$ adjacency matrices, in which the indices along both axes are in correspondence with the elements of $\mathcal{V}$. The difference between A and A_{TC} is the meaning of a given cell in the matrix. $A(i, j) = \textbf{true}$ indicates that the arc $(v_i, v_j) \in A$. $A_{TC}(i, j) = \textbf{true}$ indicates that a path of the form $v_i \ldots v_j$ exists in D.

We first provide the operation for computing the transitive closure of a DAG. We then provide operations which use the DAG's transitive closure to computing $\hat{P}$ and $\hat{S}$ (see Chapter 5).

The predecessor set ($\hat{P}$) and successor set ($\hat{S}$) for a vertex or vertex set can be computed in various ways. The approach used in this work's algorithms is to pre-compute the predecessor and successor set for each individual vertex in the DAG. Note that this is directly given by a DAG's transitive closure, which is given as a primitive operation in Appendix A.3.

Once we have $\hat{P}$ and $\hat{S}$ for each individual vertex in the DAG, computing $\hat{P}$ and $\hat{S}$ for a vertex set $\mathcal{X}$ is simply a matter of computing the obtaining and merging the predecessor and successor sets, respectively, of each vertex in $\mathcal{V}$. These operations are as follows.

`tc_vtx_pred` : (**[in]** $TC(D)$: transitive closure, **[in]** x : vertex) $\rightarrow$ vertex set

This operation assumes that $TC(D)$ is the transitive closure of the DAG $D = (\mathcal{V}, A)$. The operation returns the vertices $\hat{P}(x)$. It has running time $O(|\mathcal{V}|)$.

`tc_vtx_succ` : (**[in]** $TC(D)$: transitive closure, **[in]** x : vertex) $\rightarrow$ vertex set

This operation assumes that $TC(D)$ is the transitive closure of the DAG $D = (\mathcal{V}, A)$. The operation returns the vertices $\hat{S}(x)$. It has running time $O(|\mathcal{V}|)$.

`tc_vset_pred` : (**[in]** $TC(D)$: transitive closure, **[in]** $\mathcal{X}$: vertex set) $\rightarrow$ vertex set

This operation assumes that $TC(D)$ is the transitive closure of the DAG $D = (\mathcal{V}, A)$. The operation returns the vertices $\hat{P}(\mathcal{X})$.

Note however that $1 \leq |\mathcal{X}| \leq |\mathcal{V}|$. Therefore the running time of this algorithm in terms of $|\mathcal{V}|$ is $O(|\mathcal{V}|^2)$.

`tc_vset_succ` : (**[in]** $TC(D)$: transitive closure, **[in]** $\mathcal{X}$: vertex set) $\rightarrow$ vertex set

This operation assumes that $TC(D)$ is the transitive closure of the DAG $D = (\mathcal{V}, A)$. The operation returns the vertices $\hat{S}(\mathcal{X})$.

Note however that $1 \leq |\mathcal{X}| \leq |\mathcal{V}|$. Therefore the running time of this algorithm in terms of $|\mathcal{V}|$ is $O(|\mathcal{V}|^2)$.

A.9 Other Assumptions

Below are the remaining assumptions made in our analyses of this work's algorithms.

We assume that assignments (denoted $x \leftarrow y$) have negligible running times, and are treated as having time $O(1)$. For simple objects such as integers, this is clearly an appropriate assumption for modern computers. For complex objects such as vertex sets, we assume that the assignment is is the binding of an identifier to a pre-existing object, and not the creation of a new object or a new copy of the object.

Simple scalar operations on numbers (addition, comparison, etc.) or Boolean values (negation, conjunction, etc.) are assumed to have time $O(1)$.

As noted in Section 2.3, $|A|$ is bounded as $|A| \leq |\mathcal{V}|(|\mathcal{V}| - 1)/2$. From this we have $|A| = O(|\mathcal{V}|^2)$. This result is used in various running-time analyses throughout this work.

The asymptotic running times of the primitive operations and algorithms presented in this work are generally bounded by some function of $|\mathcal{V}|$, the order of the overall DAG under consideration. It is often possible to obtain tighter bounds on the worst-case running time of a primitive operation when additional terms are considered, such as the particular number of elements in a sequence, or the particular number of arcs in an arc set. However, for the sake of simplified running-time analyses, $|\mathcal{V}|$ is the only non-constant value to appear in our statements of asymptotic running time.

APPENDIX B

Utility Algorithms

In this appendix we present algorithms which are not the main focus of this work, but which are called by the more interesting algorithms elsewhere in the work.

The algorithms presented here differ from the operations in Appendix A in that their function, asymptotic running time, and/or randomness properties merit more explanation than do the operations presented in Appendix A.

Table B.3 below provides a the subroutine name and running time for each algorithm described in this appendix. Sections B.1-B.4 follow with a detailed presentation of each algorithm.

Operation	**Time**
`contract` (D, $\mathcal{X}$, x')	$O(\lvert\mathcal{V}\rvert^2)$
`induce_subdigraph` (D, $\mathcal{X}$)	$O(\lvert\mathcal{V}\rvert^2)$
`get_strict_dir_innbrs` ($TC(D)$, $\mathcal{X}$)	$O(\lvert\mathcal{V}\rvert^2)$
`get_strict_dir_outnbrs` ($TC(D)$, $\mathcal{X}$)	$O(\lvert\mathcal{V}\rvert^2)$
`topsort_with_embedded_cvx_set` (D, $\mathcal{X}$)	$O(\lvert\mathcal{V}\rvert^2)$

Table B.3: Summary of Utility Algorithms

B.1 Contracting a Digraph

We present below an algorithm for computing the contraction of a digraph (see Section 2.5).

This algorithm is a trivial implementation of the definition given in Section 2.5, and we consider its correctness to be easily verified by inspection. The algorithm's asymptotic running time is as follows.

The loop spanning lines 2-6 may have up to $|\mathcal{V}|$ iterations, and each iteration of the loop

Algorithm 5 Contract a Digraph

Function `contract` : $(D,\, \mathcal{X},\, x') \;\rightarrow\; D'$

Require:

(R1) $D = (\mathcal{V}, A)$ is a DAG.

(R2) $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$

(R3) $x' \notin \mathcal{X}$

Ensure:

(E1) $D' = (\mathcal{V}', A')$ is the contraction of $\mathcal{X}$ to x' in D.

```
 1: $\mathcal{V}' \leftarrow x'$                          vset_init_by_vertex( $\mathcal{V}', x'$ )   $O(|\mathcal{V}|)$
 2: for all $\mathcal{V} \in v$ do                        vset_iterate( $v$ )                          $O(|\mathcal{V}|)$
 3:    if $v \notin \mathcal{X}$ then                     vset_is_member( $\mathcal{X}, v$ )           $O(1)$
 4:       $\mathcal{V}' \leftarrow \mathcal{V}' \cup \{v\}$   vset_add( $\mathcal{V}', v$ )            $O(1)$
 5:    end if
 6: end for

 7: $A' \leftarrow \emptyset$                            vset_clear( $A'$ )                           $O(|\mathcal{V}|)$
 8: for all $A \in (u_1, v_1)$ do                        aset_iterate( $(u_1, v_1)$ )                 $O(|\mathcal{V}|^2)$
 9:    if $u_1 \notin \mathcal{V}$ then                   vset_is_member( $\mathcal{V}, u_1$ )         $O(1)$
10:       $u_2 \leftarrow u_1$                            (bind identifier)                            $O(1)$
11:    else
12:       $u_2 \leftarrow x'$                             (bind identifier)                            $O(1)$
13:    end if
14:    if $v_1 \notin \mathcal{X}$ then                   vset_is_member( $\mathcal{X}, v_1$ )         $O(1)$
15:       $v_2 \leftarrow v_1$                            (bind identifier)                            $O(1)$
16:    else
17:       $v_2 \leftarrow x'$                             (bind identifier)                            $O(1)$
18:    end if
19:    $A' \leftarrow A' \cup \{(u_2, v_2)\}$             aset_add( $A', (u_2, v_2)$ )                 $O(1)$
20: end for
21: return $D' = (\mathcal{V}', A')$
```

body is $O(1)$. Therefore the running time of lines 1-6 is clearly $O(|\mathcal{V}|)$.

As shown in Section 2.3, $|A| = O(|\mathcal{V}|^2)$. Therefore the loop spanning lines 8-20 has $O(|\mathcal{V}|^2)$ iterations, each of which has an running time of $O(1)$ (lines 9-19). From this we have a running time of lines 8-20 of $O(|\mathcal{V}|^2)$.

The running time of lines 8-20 dominates, giving the overall algorithm a running time of $O(|\mathcal{V}|^2)$.

B.2 Inducing a Subgraph

Algorithm 6 computes an induced subdigraph, as described in Section 2.6.

Algorithm 6 Induce Subgraph

```
Function induce_subdigraph : $(D, \mathcal{X}) \rightarrow D{<}\mathcal{X}{>}$
Require:
    (R1)        $D = (\mathcal{V}, A)$ is a DAG.
    (R2)        $\mathcal{X} \subseteq \mathcal{V}$
Ensure:
    (E1)        $D{<}\mathcal{X}{>} = (\mathcal{X}, A')$ is the subdigraph of $D$ induced by $\mathcal{X}$.

1: $A' \leftarrow \emptyset$                                   aset_clear( $A'$ )                 $O(|\mathcal{V}|^2)$
2: for all $A \in (u, v)$ do                                    aset_iterate( $(u, v)$ )          $O(|\mathcal{V}|^2)$
3:     if $(u \notin \mathcal{X}) \wedge (v \notin \mathcal{X})$ then                vset_is_member( $\mathcal{X}, u$ )         $O(1)$
                                                                vset_is_member( $\mathcal{X}, v$ )         $O(1)$
4:         $A' \leftarrow A' \cup \{(u, v)\}$                        aset_add( $A', (u, v)$ )           $O(1)$
5:     end if
6: end for
7: return $D{<}\mathcal{X}{>} = (\mathcal{X}, A')$
```

This algorithm is a trivial implementation of the definition given in Section 2.6, and we consider its correctness to be easily verified by inspection. The algorithm's asymptotic running time is as follows.

Lines 1 and 2 are each time $O(|\mathcal{V}|^2)$. As shown in Appendix A.9, $|A| = O(|\mathcal{V}|^2)$. Therefore the loop body on lines 3-4 executes $O(|\mathcal{V}|^2)$ times, and each iteration of the body is $O(1)$. Therefore the overall running time of this algorithm $O(|\mathcal{V}|^2)$.

B.3 Computing $\mathcal{N}^{\ominus}(\mathcal{X})$ and $\mathcal{N}^{\oplus}(\mathcal{X})$

Algorithms 7 and 8 below compute the strictly direct in-neigbors and out-neigbors, respectively, of a vertex set (see Section 2.8).

Algorithm 7 Compute Strictly Direct In-Neighbors

```
Function get_strict_dir_innbrs : $(TC(D),\, \mathcal{X}) \rightarrow \mathcal{N}^{\ominus}(\mathcal{X})$
Require:
    (R1)        $D = (\mathcal{V}, A)$ is a DAG.
    (R2)        $TC(D)$ is the transitive closure of $D$.
    (R3)        $\emptyset \subset \mathcal{X} \subset \mathcal{V}$ is a convex set of $D$.
Ensure:
    (E1)        $\mathcal{N}^{\ominus}(\mathcal{X}) = \{\, u \,|\, ((u,\mathcal{X}) \in A) \wedge (u \notin \mathcal{X}) \wedge (\hat{S}(v) \cap \hat{P}(\mathcal{X}) \subseteq \mathcal{X}) \,\}$

 1: $\mathcal{N}^{\ominus}(\mathcal{X}) \leftarrow \emptyset$                    vset_clear($\mathcal{N}^{\ominus}(\mathcal{X})$)          $O(|\mathcal{V}|)$
 2: $\mathcal{I}_{\mathcal{X}} \leftarrow$ call aset_in_nbors( $A, \mathcal{X}$ )                                   $O(|\mathcal{V}|^2)$
 3: $\mathcal{P}_{\mathcal{X}} \leftarrow \hat{P}(\mathcal{X})$                    tc_vset_pred( $TC(D), \mathcal{X}$ )      $O(|\mathcal{V}|^2)$
 4: for all $v \in \mathcal{I}_{\mathcal{X}}$ do                 vset_iterate( $\mathcal{I}_{\mathcal{X}}$ )              $O(|\mathcal{V}|)$
 5:     use $\leftarrow (\hat{S}(v) \cap \mathcal{P}_{\mathcal{X}}) \subseteq \mathcal{X}$      tc_vtx_succ( $TC(D), v$ )         $O(|\mathcal{V}|)$
                                            vset_isect( ... )                 $O(|\mathcal{V}|)$
                                            vset_is_subset( ... )             $O(|\mathcal{V}|)$
 6:     if use then                         (scalar op)                       $O(1)$
 7:         $\mathcal{N}^{\ominus}(\mathcal{X}) \leftarrow \mathcal{N}^{\ominus}(\mathcal{X}) \cup \{v\}$      vset_add($\mathcal{N}^{\ominus}(\mathcal{X}), v$ )       $O(1)$
 8:     end if
 9: end for
10: return $\mathcal{N}^{\ominus}(\mathcal{X})$
```

Algorithms 7 and 8 are trivial implementations of the formulae given in Equation (1) and Equation (2), respectively. Using the assumptions stated in Appendix A, these algorithms' runnings times are clearly $O(|\mathcal{V}|^2)$.

Algorithm 8 Compute Strictly Direct Out-Neighbors

```
Function get_strict_dir_outnbrs : $(TC(D),\ \mathcal{X}) \rightarrow \mathcal{N}^{\oplus}(\mathcal{X})$
Require:
    (R1)    $D = (\mathcal{V}, A)$ is a DAG.
    (R2)    $TC(D)$ is the transitive closure of $D$.
    (R3)    $\emptyset \subset \mathcal{X} \subset \mathcal{V}$ is a convex set of $D$.
Ensure:
    (E1)    $\mathcal{N}^{\oplus}(\mathcal{X}) = \{\, u \,|\, ((\mathcal{X}, u) \in A) \wedge (u \notin \mathcal{X}) \wedge (\hat{P}(v) \cap \hat{S}(\mathcal{X}) \subseteq \mathcal{X}) \,\}$

 1: $\mathcal{N}^{\oplus}(\mathcal{X}) \leftarrow \emptyset$                      vset_clear( $\mathcal{N}^{\oplus}(\mathcal{X})$ )          $O(|\mathcal{V}|)$
 2: $\mathcal{O}_{\mathcal{X}} \leftarrow$ call aset_out_nbors( $A, \mathcal{X}$ )                                           $O(|\mathcal{V}|^2)$
 3: $\mathcal{S}_{\mathcal{X}} \leftarrow \hat{S}(\mathcal{X})$                        tc_vset_succ( $TC(D), \mathcal{X}$ )       $O(|\mathcal{V}|^2)$
 4: for all $v \in \mathcal{O}_{\mathcal{X}}$ do                  vset_iterate( $\mathcal{O}_{\mathcal{X}}$ )             $O(|\mathcal{V}|)$
 5:     $use \leftarrow (\hat{P}(v) \cap \mathcal{S}_{\mathcal{X}}) \subseteq \mathcal{X}$        tc_vtx_pred( $TC(D), v$ )          $O(|\mathcal{V}|)$
                                           vset_isect( ... )                  $O(|\mathcal{V}|)$
                                           vset_is_subset( ... )              $O(|\mathcal{V}|)$
 6:     if $use$ then                      (scalar op)                        $O(1)$
 7:         $\mathcal{N}^{\oplus}(\mathcal{X}) \leftarrow \mathcal{N}^{\oplus}(\mathcal{X}) \cup \{v\}$     vset_add( $\mathcal{N}^{\oplus}(\mathcal{X}), v$ )        $O(1)$
 8:     end if
 9: end for
10: return $\mathcal{N}^{\oplus}(\mathcal{X})$
```

B.4 Topological Sorting with Embedded Convex Set

Given $\mathcal{X}$, a convex set of D, Algorithm 9 returns some topological sort of D in which the vertices of $\mathcal{X}$ appears as a contiguous subsequence.

B.4.1 Running Time of Algorithm 9

From inspection, algorithm clearly has running time $O(|\mathcal{V}|^2)$. We proceed below with an overview of the algorithm's approach, followed by more formal arguments its correctness.

B.4.2 Overview of Algorithm 9

This algorithm ensures that $\vec{Q}$ a topological sort of D by establishing that for every arc $(u, v) \in A$, u appears before v in $\vec{Q}$.

The arcs of D are implicitly divided into four groups, differentiated on whether or not a

Algorithm 9 Topologically Sort with an Embedded Contiguous Convex Set

Function `topsort_with_embedded_cvx_set` : $(D,\ \mathcal{X}) \rightarrow \vec{Q}$

Require:

(R1) $D = (\mathcal{V}, A)$ is a DAG.

(R2) $\emptyset \subset \mathcal{X} \subseteq V$ is a convex set of D.

Ensure:

(E1) $\vec{X}$ is a topological sort of $D{<}\mathcal{X}{>}$.

(E2) $\vec{Q}$ is a topological sort of D.

(E3) $\vec{Q}$ has the structure $\vec{Q} = [\ldots,\ \vec{X},\ \ldots]$.

(E4) Let $\vec{Z}$ be any topological sort of D with structure $[\ldots,\ u,\ \vec{X},\ \ldots]$. Then there is a non-zero probability that each invocation of this algorithm returns $\vec{Q}$ such that $\vec{Q} = \vec{Z}$.

(E5) Let $\vec{Z}$ be any topological sort of D with structure $[\ldots,\ \vec{X},\ u,\ \ldots]$. Then there is a non-zero probability that each invocation of this algorithm returns $\vec{Q}$ such that $\vec{Q} = \vec{Z}$.

```
 1: $(\mathcal{V}', A') \leftarrow$ call contract( $D, \mathcal{X}, x'$ )                                  $O(|\mathcal{V}|^2)$
 2: $A' \leftarrow A' \setminus (x', x')$                          aset_del( $A', (x', x')$ )   $O(1)$

    (Comment: $\vec{T} = \vec{W} x' \vec{Y}$ for some $\vec{W}$, $\vec{Y}$)
 3: $\vec{T} \leftarrow$ call dag_rand_topo_sort( $D_C = (\mathcal{V}', A')$ )                    $O(|\mathcal{V}|^2)$
 4: $xpos \leftarrow$ call vseq_find( $\vec{T}, x'$ )                                              $O(|\mathcal{V}|)$
 5: $\vec{W} \leftarrow \vec{T}[\![1 \ldots (xpos-1)]\!]$                 vseq_slice( $\vec{T}, \ldots$ )   $O(|\mathcal{V}|)$
 6: $\vec{Y} \leftarrow \vec{T}[\![(xpos+1) \ldots |\vec{T}|]\!]$          vseq_slice( $\vec{T}, \ldots$ )   $O(|\mathcal{V}|)$

 7: $D{<}\mathcal{X}{>} \leftarrow$ call induce_subdigraph( $D, \mathcal{X}$ )                     $O(|\mathcal{V}|^2)$
 8: $\vec{X} \leftarrow$ call dag_rand_topo_sort( $D{<}\mathcal{X}{>}$ )                            $O(|\mathcal{V}|^2)$
 9: $\vec{Q} \leftarrow \vec{W}\vec{X}\vec{Y}$                          vss_concat( $\vec{W}, \vec{X}$ )     $O(1)$
                                                  vss_concat( $\vec{W}\vec{X}, \vec{Y}$ )  $O(1)$
10: return $\vec{Q}$
```

given arc originates in $\mathcal{X}$, and whether or not the arc terminates in $\mathcal{X}$.

Although $\vec{T}$ is a topological sort of D_C rather than D, D_C may have some arcs in common with D. The algorithm ensures that for the vertices ordered by these common arcs, the relative ordering of these vertices as provided by $\vec{T}$ is preserved during the construction of $\vec{Q}$.

Similarly, although $\vec{X}$ is a topological sort of $D{<}\mathcal{X}{>}$ rather than of D, $D{<}\mathcal{X}{>}$ and D, and $(\mathcal{X}, \mathcal{X})$-arc in D is also an arc in $D{<}\mathcal{X}{>}$. By ensuring that $\vec{Q}$ has the same relative ordering of all vertices in $\mathcal{X}$ as does $\vec{X}$, the algorithm guarantees that $\vec{Q}$ is consistent with all $(\mathcal{X}, \mathcal{X})$ arcs of D.

Line 1. Recall that in the contraction $\mathcal{X} \to x'$ of D, every arc in D that terminates in $\mathcal{X}$ is replaced by an arc terminating at the vertex x'. Similarly, any arc originating in $\mathcal{X}$ in D is replaced by an arc originating at x'. In all other regards, the directed graph $(\mathcal{V}', A')$ is identical to D.

Line 2. Although D is by assumption acyclic, the possibility exists that $(\mathcal{V}', A')$ contains a cycle. If A has any arc of the form $(\mathcal{X}, \mathcal{X})$, then the arc (x', x') is be present in $(\mathcal{V}', A')$. The presence of such an arc in $(\mathcal{V}', A')$ is not problematic for our overall purposes, but if present, $(\mathcal{V}', A')$ is not acyclic and therefore has no topological sort. Therefore on line 2 we ensure $(\mathcal{V}', A')$ is a DAG by ensuring $(x', x') \notin A'$.

Lines 3-6. Here we create a topological sort of $(\mathcal{V}', A')$. A crucial correspondence exists between $\vec{T}$ and every topological sort of D. The correspondence is discussed in terms of the four categories of arcs in A identified above. All four categories are covered in our analyses of the remaining lines of this algorithm. Lines 4-6 simply decompose $\vec{T}$ into the components $\vec{W}$, x', and $\vec{Y}$.

Lines 7-9. The induced digraph $D{<}\mathcal{X}{>}$ contains all vertices in $\mathcal{X}$, and all of the $(\mathcal{X}, \mathcal{X})$ arcs in A. Because $D < \mathcal{X} >$ has a subset of the arcs in a DAG (D), it is not possible that $D{<}\mathcal{X}{>}$ contains any cycle. Therefore $D{<}\mathcal{X}{>}$ is also a DAG. All DAG's have at least one topological sort [1], which here we call $\vec{X}$.

Line 8 obtains an arbitrary topological sort of $D{<}\mathcal{X}{>}$, and line 9 constructs a total ordering of $\mathcal{V}$. In the following subsections we show that $\vec{Q}$ satisfies all of this algorithm's postconditions.

B.4.3 Correctness of Algorithm 9 postconditions (E1) - (E3)

Postconditions (E1) and (E3) are clearly satisfied by lines 7 an 9 of the algorithm. We now prove postcondition (E2).

$\vec{Q}$ is a topological sort of D if and only if for every arc $(u, v) \in A$, u appears before v in $\vec{Q}$. We divide the arcs of A into four groups, and for each group show that this ordering requirement is satisfied by D.

Case 1: $(u, v) \in A$, *with* $u \in \mathcal{X} \wedge v \in \mathcal{X}$*:*

Every arc of this form is also an arc in the DAG $D{<}\mathcal{X}{>}$. $\vec{X}$ is a topological sort of $D{<}\mathcal{X}{>}$. Therefore for every arc $(u, v) \in A$, u appears before v in $\vec{X}$.

$\vec{Q}$ contains $\vec{X}$, and therefore if u appears before v in $\vec{X}$, u also appears before v in $\vec{Q}$. Thus every arc in A covered by Case 1 is respected by $\vec{Q}$.

Case 2: $(u, v) \in A$, *with* $u \notin \mathcal{X} \wedge v \in \mathcal{X}$*:*

$\vec{T}$ is a topological sort of the DAG $(\mathcal{V}', A')$. Therefore for every arc of the form $(u, x') \in A'$, u appears before x' in $\vec{T}$.

From lines 4-5 of the algorithm, we have $u \in \mathcal{W}$. From lines 7-8 of the algorithm, we have that if $v \in \mathcal{X}$, then $v \in \vec{X}$. Therefore every arc covered by Case 2 is an arc of the form $(\mathcal{W}, \mathcal{X})$.

Line 9 of the algorithms constructs $\vec{Q}$ such that all vertices in $\mathcal{W}$ occur before all vertices

[1] See [1, Prop. 2.1.3].

of $\mathcal{X}$. Therefore every arc in A covered by Case 2 is respected by $\vec{Q}$.

Case 3: $(u, v) \in A$, *with* $u \in \mathcal{X} \wedge v \notin \mathcal{X}$:

The proof for Case 3 has the same structure as the proof for Case 2, above.

Case 4: $(u, v) \in A$, *with* $u \notin \mathcal{X} \wedge v \notin \mathcal{X}$:

Because neither u nor v is a member of $\mathcal{X}$, the arc (u, v) is an arc in both D and in the contracted digraph $(\mathcal{V}', A')$. $\vec{T}$ is a topological sort of $(\mathcal{V}', A')$, therefore for every arc (u, v) covered by Case 4, u appears before v in $\vec{T}$.

We now show that for the arcs (u, v) covered by Case 4, u also appears before v in $\vec{Q}$.

There are three possible locations for u and v: $u, v \in \mathcal{W}$, or $u, v \in \mathcal{Y}$, or $u \in \mathcal{W} \wedge v \in \mathcal{Y}$.

If $u, v \in \mathcal{W}$, then any topological sort with $\vec{W}$ as an embedded subpath respects the (u, v) arc. $\vec{Q}$ is one such topological sort, and therefore u appears before v in $\vec{Q}$.

By similar reasoning, if $u, v \in \mathcal{Y}$, then u appears before v in $\vec{Q}$.

Finally, if $u \in \mathcal{W} \wedge v \in \mathcal{Y}$, then u appears before v in $\vec{Q}$ because line 9 of the algorithm $\vec{Q}$ such that all vertices of $\vec{W}$ appear before all vertices of $\vec{Y}$.

This concludes our proof of postcondition (E3).

B.4.4 Correctness of Algorithm 9 postconditions (E4)-(E5)

We now demonstrate that postcondition (E4) holds: If $\vec{Z} = \ldots u, \mathcal{X} \ldots$ is a topological sort of D for any $u \notin \mathcal{X}$, then there's a non-zero probability that this algorithm returns $\vec{Z}$.

This follows directly from our assumptions about the randomness of the *dag_rand_topo_sort* subroutine as detailed in Appendix A.3.

Postcondition (E5) holds by the same logic.

APPENDIX C

Basic Convexity Algorithms

In this appendix we present algorithms for initializing a convex set according to either of two criteria, and for growing or shrinking an existing convex set by some specified number of vertices.

For each of these four kinds of algorithms, we provide two variations based on their underlying approach. One group of algorithms is implemented using the transitive closure of the DAG and $\hat{P}$ and $\hat{S}$ sets, and the other group is implemented in terms of topological sorts of the DAG. Section 7.3 discusses the relative merits of using these different implementation approaches.

Operation	**Time**
`init_convex_set_by_seeds_PS` ($TC(D), \mathcal{X}$)	$O(\|\mathcal{V}\|^2)$
`init_convex_set_by_seeds_TS` ($D, \mathcal{X}$)	$O(\|\mathcal{V}\|^2)$
`init_convex_set_by_order_PS` ($TC(D), \sigma$)	$O(\|\mathcal{V}\|^3)$
`init_convex_set_by_order_TS` (D, σ)	$O(\|\mathcal{V}\|^2)$
`grow_convex_set_PS` ($TC(D), \mathcal{X}, \delta$)	$O(\|\mathcal{V}\|^3)$
`grow_convex_set_TS` ($D, \mathcal{X}, \delta$)	$O(\|\mathcal{V}\|^2)$
`shrink_convex_set_PS` ($TC(D), \mathcal{X}, \delta$)	$O(\|\mathcal{V}\|^2)$
`shrink_convex_set_TS` ($D, \mathcal{X}, \delta$)	$O(\|\mathcal{V}\|^2)$

Table C.4: Summary of Basic Convexity Algorithms

C.1 Initializing Convex Set by Seeds using $\hat{P}$ and $\hat{S}$

Algorithm 10 computes the smallest convex superset of the specified set of seed vertices. If the seed set $\mathcal{X}$ is a convex set of D, this simply returns $\mathcal{X}$.

The correctness of Algorithm 10 is clearly established by Theorem 5.3.4 and Corollary 5.3.5. From inspection the running time of this algorithm is clearly $O(|\mathcal{V}|^2)$.

Algorithm 10 Initialize a Convex Set by Seeds using $\hat{P}$ and $\hat{S}$

```
Function init_convex_set_by_seeds_PS : $(TC(D), \mathcal{X}) \rightarrow \mathcal{W}$
Require:
    (R1)    $D = (\mathcal{V}, A)$ is a DAG.
    (R2)    $TC(D)$ is the transitive closure of $D$.
    (R3)    $\emptyset \subset \mathcal{X} \subset \mathcal{V}$
Ensure:
    (E1)    $\mathcal{W}$ is the smallest superset of $\mathcal{X}$ that is a convex set of $D$.

1: $\mathcal{PX} \leftarrow \hat{P}(\mathcal{X})$                 tc_vset_pred( $TC(D), \mathcal{X}$ )    $O(|\mathcal{V}|^2)$
2: $\mathcal{SX} \leftarrow \hat{S}(\mathcal{X})$                 tc_vset_succ( $TC(D), \mathcal{X}$ )    $O(|\mathcal{V}|^2)$
3: $\mathcal{W} \leftarrow \mathcal{X} \cup (\mathcal{PX} \cap \mathcal{SX})$       vset_isect( $\mathcal{PX}, \mathcal{SX}$ )        $O(|\mathcal{V}|)$
                                       vset_union( $\mathcal{X}, \dots$ )         $O(|\mathcal{V}|)$
4: return $\mathcal{W}$
```

C.2 Initializing Convex Set by Seeds using Topological Sort

Algorithm 11 computes some convex superset of the specified set set. Note that unlike Appendix C.1, this algorithm makes no assurance that it returns the *smallest* convex superset of the seed set. This stems from this algorithm's use of a random topological sort, which may not arrange the vertices of $\mathcal{X}$ into the smallest possible subsequence of the topological sort.

Algorithm 11 Initialize a Convex Set by Seeds using Topological Sort

```
Function init_convex_set_by_seeds_TS : $(D, \mathcal{X}) \rightarrow \mathcal{W}$
Require:
    (R1)    $D = (\mathcal{V}, A)$ is a DAG.
    (R2)    $\emptyset \subset \mathcal{X} \subseteq \mathcal{V}$
Ensure:
    (E1)    $\mathcal{Z}$ is a convex set of $D$.
    (E2)    $\mathcal{X} \subseteq \mathcal{Z}$

1: $\vec{T} \leftarrow$ call dag_rand_topo_sort( $D$ )                              $O(|\mathcal{V}|^2)$
2: $(p, q) \leftarrow$ call vseq_find_bounds( $\vec{T}, \mathcal{X}$ )                       $O(|\mathcal{V}|^2)$
3: $\vec{Z} \leftarrow \vec{T}[\![p \dots q]\!]$                 vseq_slice( $\vec{T}, \dots$ )            $O(|\mathcal{V}|)$
4: $\mathcal{Z} \leftarrow$ call vseq_to_vset( $\vec{Z}$ )                                  $O(|\mathcal{V}|)$
5: return $\mathcal{Z}$
```

From inspection, the running time of this algorithm is clearly $O(|\mathcal{V}|^2)$.

We establish the algorithm's correctness as follows. Clearly $\mathcal{X} \subseteq \mathcal{Z}$, because the subsequence of $\vec{T}$ from which $\mathcal{Z}$ is formed contains all members of $\mathcal{X}$.

What remains is to show that $\mathcal{Z}$ is a convex set of D. This follows from Theorem 4.2.1, which states that any contiguous subsequence of any topological sort of D is itself a convex set of D.

C.3 Initializing Convex Set by Order using $\hat{P}$ and $\hat{S}$

Algorithm 12 creates a convex set of the specified order σ using the predecessor and successor set operations.

Algorithm 12 Initialize a Convex Set by Order using $\hat{P}$ and $\hat{S}$

```
Function init_convex_set_by_order_PS : $(TC(D),\, \sigma) \rightarrow \mathcal{Z}$
Require:
    (R1)    $D = (\mathcal{V}, A)$ is a DAG.
    (R2)    $TC(D)$ is the transitive closure of $D$.
    (R3)    $1 \le \sigma \le |\mathcal{V}|$
Ensure:
    (E1)    $\mathcal{Z}$ is a convex set of $D$.
    (E2)    $|\mathcal{Z}| = \sigma$

 1: $\mathcal{X} \leftarrow$ call vset_random_subset($\mathcal{V}, 1$)                      $O(|\mathcal{V}|)$
 2: $\mathcal{Z} \leftarrow$ call grow_convex_set_PS($TC(D), \mathcal{X}, (\sigma - 1)$)      $O(|\mathcal{V}|^3)$
 3: return $\mathcal{Z}$
```

From inspection, the running time of this algorithm is clearly dominated by line 2, and is therefore $O(|\mathcal{V}|^3)$.

The correctness of this algorithm is trivially given by the correctness of the subroutine `grow_convex_set_PS` (Algorithm 14), which is established in Appendix C.5.

C.4 Initializing Convex Set by Order using Topological Sort

Algorithm 13 creates a convex set with the specified number of vertices, without regard to which particular vertices appear in the set.

Algorithm 13 Initialize Convex Set by Order using Topological Sort

```
Function init_convex_set_by_order_TS : $(D, \sigma) \to \mathcal{X}$
Require:
    (R1)        $D = (\mathcal{V}, A)$ is a DAG.
    (R2)        $1 \leq \sigma \leq |\mathcal{V}|$
Ensure:
    (E1)        $\mathcal{X}$ is a convex set of $D$.
    (E2)        $|\mathcal{X}| = \sigma$

1: $\vec{T} \leftarrow$ call dag_rand_topo_sort( $D$ )                      $O(|\mathcal{V}|^2)$
2: $\mathcal{X} \leftarrow \vec{T}[\![1 \ldots \sigma]\!]$          vseq_slice( $\vec{T}, \ldots$ )    $O(|\mathcal{V}|)$
3: return $\mathcal{X}$
```

The running time of this algorithm is dominated by line 1, and is therefore $O(|\mathcal{V}|^2)$.

This algorithm is trivially correct due to Theorem 4.2.1, which states that any contiguous subsequence of any topological sort of D is itself a convex set of D.

C.5 Growing Convex Sets Using $\hat{P}$ and $\hat{S}$

Given a convex set $\mathcal{X}$, Algorithm 14 returns some superset of $\mathcal{X}$ having $|\mathcal{X}| + \delta$ vertices.

C.5.1 Running Time of Algorithm 14

We now consider the running time of Algorithm 14, using the assumptions stated in Appendix A.

Lines 3-6 have the highest asymptotic running-times of any in the algorithm, $O(|\mathcal{V}|^2)$ per activation. Furthermore, these lines are within a loop that might execute up to δ times.

δ represents the number of vertices to be added to $\mathcal{X}$. Because $|\mathcal{X}| \geq 1$, and the set produced by this algorithm must be an improper subset of $\mathcal{V}$, it must always be true that $\delta < |\mathcal{V}|$.

Therefore the overall running time for this algorithm is $O(\delta \times |\mathcal{V}|^2)$, or $O(|\mathcal{V}|^3)$.

Algorithm 14 Grow Convex Set Using Predecessor and Successor Sets

Function `grow_convex_set_PS` : $(TC(D),\, \mathcal{X},\, \delta) \rightarrow \mathcal{Z}$

Require:

(R1)	$D = (\mathcal{V}, A)$ is a DAG.
(R2)	$TC(D)$ is the transitive closure of D.
(R3)	$\emptyset \subset \mathcal{X} \subset \mathcal{V}$ is a convex set of D.
(R4)	$1 \leq \delta \leq \lvert\mathcal{V} \setminus \mathcal{X}\rvert$

Ensure:

(E1)	$\mathcal{Z}$ is a convex set of D.
(E2)	$\mathcal{X} \subset \mathcal{Z}$
(E3)	$\lvert\mathcal{Z}\rvert = \lvert\mathcal{X}\rvert + \delta$
(E4)	Let $u \notin \mathcal{X}$ be any vertex such that $\mathcal{X} \cup \{u\}$ is a convex set of D. Then if $\delta = 1$, there is a non-zero probability that this algorithm returns with $\mathcal{Z} = \mathcal{X} \cup \{u\}$.

```
 1: $\mathcal{Z} \leftarrow$ *call* vset_clone( $\mathcal{X}$ )                                        $O(|\mathcal{V}|)$
 2: for $i = 1$ to $\delta$ do                        (scalar op)              $O(\delta - 1)$
 3:     $\mathcal{PZ} \leftarrow \hat{P}(\mathcal{Z})$                       tc_vset_pred( $TC(D), \mathcal{Z}$ )  $O(|\mathcal{V}|^2)$
 4:     $\mathcal{SZ} \leftarrow \hat{S}(\mathcal{Z})$                       tc_vset_succ( $TC(D), \mathcal{Z}$ )  $O(|\mathcal{V}|^2)$
 5:     $\mathcal{N}^{\ominus}(\mathcal{Z}) \leftarrow$ *call* get_strict_dir_innbrs( $TC(D), \mathcal{Z}$ )          $O(|\mathcal{V}|^2)$
 6:     $\mathcal{N}^{\oplus}(\mathcal{Z}) \leftarrow$ *call* get_strict_dir_outnbrs( $TC(D), \mathcal{Z}$ )         $O(|\mathcal{V}|^2)$
 7:     $\mathcal{R} \leftarrow (\mathcal{V} \setminus (\mathcal{Z} \cup \mathcal{PZ} \cup \mathcal{SZ}))$         vset_union( ... )         $O(|\mathcal{V}|)$
                                              vset_union( ... )         $O(|\mathcal{V}|)$
                                              vset_diff( ... )          $O(|\mathcal{V}|)$
 8:     $\mathcal{Y} \leftarrow \mathcal{N}^{\ominus}(\mathcal{Z}) \cup \mathcal{N}^{\oplus}(\mathcal{Z}) \cup \mathcal{R}$          vset_union( ... )         $O(|\mathcal{V}|)$
                                              vset_union( ... )         $O(|\mathcal{V}|)$
 9:     $\mathcal{W} \leftarrow$ *call* vset_random_subset( $\mathcal{Y}, 1$ )                       $O(|\mathcal{V}|)$
10:     $\mathcal{Z} \leftarrow \mathcal{Z} \cup \mathcal{W}$                        vset_union( $\mathcal{Z}, \mathcal{W}$ )        $O(|\mathcal{V}|)$
11: end for
12: return $\mathcal{Z}$
```

C.5.2 Correctness of Algorithm 14 postconditions (E1)-(E3)

Each activation of the loop body spanning lines 3-9 grows $\mathcal{Z}$ by one vertex, using the formula given by Theorem 5.4.3. The algorithm's preconditions ensure that $\mathcal{Z}$ is a convex set of D prior to the first activation of the loop body, and Theorem 5.4.3 ensures that the new value of $\mathcal{Z}$ is a convex set of D after each completion of the loop body.

The for-loop spanning lines 2-10 repeatedly applies Theorem 5.4.3 to obtain a convex set of order $|\mathcal{X}| + \delta$.

C.5.3 Correctness of Algorithm 14 postcondition (E4)

Theorem 5.4.3 states that $\mathcal{Y}$ is the precise vertex set such that $\mathcal{X} \cup \{y\}$ is a convex set of D if and only if $y \in \mathcal{Y}$.

When $\delta = 1$, this algorithm clearly ensures that whichever vertex was added to $\mathcal{X}$ is a vertex from $\mathcal{Y}$. What remains is to show that this algorithm has a non-zero probability of selecting *each* vertex $y \in \mathcal{Y}$. This comes from our definition of the `vset_random_subset` operation, which in Appendix A.4 is defined to have a non-zero probably of returning any subset of the specified order.

C.6 Growing Convex Sets Using Topological Sort

Given a convex set $\mathcal{X}$, this algorithm returns some superset of $\mathcal{X}$ having $|\mathcal{X}| + \delta$ vertices.

From inspection, the running time of this algorithm is clearly $O(|\mathcal{V}|^2)$. We now consider the algorithm's correctness.

C.6.1 Correctness of Algorithm 15 postconditions (E1)-(E3)

Line 1 computes a topological sort of D in which the members of $\mathcal{X}$ are contiguous. Theorem 4.2.2 demonstrates that it is always possible to attain such an ordering.

Algorithm 15 Grow Convex Set Using Topological Sort

```
Function grow_convex_set_TS : $(D,\, \mathcal{X},\, \delta) \;\to\; \mathcal{Z}$
Require:
    (R1)    $D = (\mathcal{V}, A)$ is a DAG.
    (R2)    $\emptyset \subset \mathcal{X} \subset \mathcal{V}$ is a convex set of $D$.
    (R3)    $1 \leq \delta \leq |\mathcal{V} \setminus \mathcal{X}|$
Ensure:
    (E1)    $\mathcal{Z}$ is a convex set of $D$.
    (E2)    $\mathcal{X} \subset \mathcal{Z}$.
    (E3)    $|\mathcal{Z}| = |\mathcal{X}| + \delta$.
    (E4)    Let $u \notin \mathcal{X}$ be any vertex such that $\mathcal{X} \cup \{v\}$ is a convex set of $D$.
            Then if $\delta = 1$, there is a non-zero probability that this algorithm
            returns with $\mathcal{Z} = \mathcal{X} \cup \{v\}$.

    (Comment: $\vec{T} = \vec{W}\vec{X}\vec{Y}$ for some $\vec{W}$, $\vec{X}$, and $\vec{Y}$)
1: $\vec{T} \leftarrow$ call topsort_with_embedded_cvx_set( $D, \mathcal{X}$ )                      $O(|\mathcal{V}|^2)$
2: $(x_{start}, x_{end}) \leftarrow$ call vseq_find_bounds( $\vec{T}, \mathcal{X}$ )                   $O(|\mathcal{V}|^2)$

3: $num\_w \leftarrow$ call random_integer( $0, (x_{start} - 1)$ )                              $O(1)$
4: $num\_y \leftarrow (\delta \;-\; num\_w)$                      (scalar op)                  $O(1)$

5: $slice_{start} \leftarrow (x_{start} - num\_w)$                (scalar op)                  $O(1)$
6: $slice_{end} \leftarrow (x_{end} + num\_y)$                    (scalar op)                  $O(1)$
7: $\vec{Z} \leftarrow \vec{T}[\![slice_{start} \dots slice_{end}]\!]$            vseq_slice( $\vec{T}, \dots$ )          $O(|\mathcal{V}|)$
8: $\mathcal{Z} \leftarrow$ call vseq_to_vset( $\vec{Z}$ )                                            $O(|\mathcal{V}|)$
9: return $\mathcal{Z}$
```

Lines 2-8 randomly select a contiguous subsequence $\vec{Z}$ of $\vec{T}$. $\vec{Z}$ has $|\mathcal{X}|+\delta$ elements, and contains $\vec{X}$. Because $\vec{X}$ contains precisely the vertices of $\mathcal{V}$, $\mathcal{Z}$ is a superset of $\mathcal{X}$.

Because $\vec{Z}$ is a contiguous subsequence of a topological sort of D, we have from Theorem 4.2.1 that $\mathcal{Z}$ is a convex set of D.

C.6.2 Correctness of Algorithm 15 postcondition (E4)

From Theorem 4.2.5, we have that for any vertex $v \in \mathcal{V} \backslash \mathcal{X}$ such that $\mathcal{X} \cup \{v\}$ is a convex set of D, there exists at least one topological sort of the form $\vec{T} = [\ldots,\ v,\ \vec{X},\ \ldots]$ or of the form $\vec{T} = [\ldots,\ \vec{X},\ v,\ \ldots]$. We show that when $\delta = 1$, line 1 of this algorithm has a non-zero probability of producing such a topological sort for any such vertex v. We then show that given such a topological sort, there's a non-zero probability of v subsequently being selected to appear in $\mathcal{Z}$.

In Appendix B.4 it is shown that `topsort_with_embedded_cvx_set` if a topological sort exists having either of those two forms, there is a non-zero probability that an invocation of `topsort_with_embedded_cvx_set` returns a topological sort having that form. And so in line 1 of this algorithm we have a non-zero probability that for any $u \in \mathcal{V} \setminus \mathcal{X}$ such that $\mathcal{X} \cup \{v\}$ is a convex set of D, $\vec{T}$ has either the structure $\vec{T} = [\ldots,\ v,\ \vec{X},\ \ldots]$ or of the form $\vec{T} = [\ldots,\ \vec{X},\ v,\ \ldots]$.

Line 3 determines which neighbors in $\vec{T}$ of $\vec{X}$ appear in $\mathcal{Z}$. Because the `random_integer` is defined to have a non-zero probability of returning any value in the closed interval $[0, (x_{start} - 1)]$, there is a non-zero probability that line 3 causes $\mathcal{Z}$ to contain either the vertex in $\vec{T}$ that immediately precedes $\vec{X}$, and/or the vertex that immediately follows $\vec{X}$.

C.7 Shrinking Convex Sets Using $\hat{P}$ and $\hat{S}$

Given a convex set $\mathcal{X}$, this algorithm returns some subset of $\mathcal{X}$ having $|\mathcal{X}| - \delta$ vertices. The basis for our approach to shrinking existing convex sets is provided in Theorem 5.4.5.

Algorithm 16 Shrink Convex Set Using Predecessor and Successor Sets

```
Function shrink_convex_set_PS : $(TC(D), \mathcal{X}, \delta) \rightarrow \mathcal{Z}$
Require:
    (R1)    $D = (\mathcal{V}, A)$ is a DAG.
    (R2)    $TC(D)$ is the transitive closure of $D$.
    (R3)    $\emptyset \subset \mathcal{X} \subseteq V$ is a convex set of $D$.
    (R4)    $|\mathcal{X}| > 1$
    (R5)    $1 \leq \delta < |\mathcal{X}|$
Ensure:
    (E1)    $\mathcal{Z}$ is a convex set of $D$.
    (E2)    $\mathcal{Z} \subset \mathcal{X}$
    (E3)    $|\mathcal{Z}| = |\mathcal{X}| - \delta$
    (E4)    Let $u \notin \mathcal{X}$ be any vertex such that $\mathcal{X} \cup \{v\}$ is a convex set of $D$.
            Then if $\delta = 1$, there is a non-zero probability that this algorithm
            returns with $\mathcal{Z} = \mathcal{X} \setminus \{v\}$.

 1: $\mathcal{PX} \leftarrow \hat{P}(\mathcal{X})$                        tc_vset_pred( $TC(D), \mathcal{X}$ )   $O(|\mathcal{V}|^2)$
 2: $\mathcal{SX} \leftarrow \hat{S}(\mathcal{X})$                        tc_vset_succ( $TC(D), \mathcal{X}$ )   $O(|\mathcal{V}|^2)$
 3: $\mathcal{Z} \leftarrow$ call vset_clone( $\mathcal{X}$ )                                          $O(|\mathcal{V}|)$
 4: $to\_remove \leftarrow \delta$                          (bind identifier)              $O(1)$
 5: while $to\_remove > 0$ do                   (scalar op)                    $O(1)$
 6:     $\mathcal{Y} \leftarrow \mathcal{Z} \setminus (\hat{P}(\mathcal{Z}) \cap \hat{S}(\mathcal{Z}))$          vset_isect( $\hat{P}(\mathcal{Z}), \hat{S}(\mathcal{Z})$ )   $O(|\mathcal{V}|)$
                                                vset_diff( $\mathcal{Z}, \ldots$ )             $O(|\mathcal{V}|)$
 7:     $\mathcal{W} \leftarrow$ call vset_random_subset( $\mathcal{Y}, min(|\mathcal{Y}|, to\_remove)$ )     $O(|\mathcal{V}|)$
 8:     $\mathcal{Z} \leftarrow \mathcal{Z} \setminus \mathcal{W}$                     vset_diff( $\mathcal{Z}, \mathcal{W}$ )           $O(|\mathcal{V}|)$
 9:     $to\_remove \leftarrow to\_remove - |\mathcal{W}|$      (scalar op)                    $O(1)$
10: end while
11: return $\mathcal{Z}$
```

C.7.1 Running Time of Algorithm 16

Because δ represents the number of vertices to be removed from $\mathcal{X}$, and the resulting set $\mathcal{Z}$ cannot be the empty set, we have $\delta < |\mathcal{V}|$. This provides the upper limit for the number of iterations of the loop spanning lines 5-10.

The longest-running statements within the loop body are $O(|\mathcal{V}|)$. Therefore the overall loop has a running time of $O(|\mathcal{V}|^2)$. This matches the running times of lines 1-2, and so the overall running time of this algorithm is $O(|\mathcal{V}|^2)$.

C.7.2 Correctness of Algorithm 16 postconditions (E1)-(E3)

The correctness of this algorithm comes from Theorem 5.4.5, which indicates that for a given convex set $\mathcal{X}$ of D, any subset of $\mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$ may be removed from $\mathcal{X}$ to obtain a new convex set of D.

The loop body spanning lines 5-9 deletes as many vertices as possible from $\mathcal{Z}$ using the formula given by Theorem 5.4.5. The algorithm's preconditions ensure that $\mathcal{Z}$ is a convex set of D before the first activation of the loop body, and Theorem 5.4.5 ensures that the new version of $\mathcal{Z}$ is a convex set of D after each completion of the loop body.

Corollary 5.4.6 ensures that $\mathcal{Y} \neq \emptyset$, and therefore each activation of the loop body always reduces the order of $\mathcal{Z}$ by at least one. Therefore this algorithm always terminates.

Because there is no assurance that $|\mathcal{Y}| \geq \delta$, repeated applications of the formula from Theorem 5.4.5 may be necessary, provide by the loop spanning lines 5-10.

C.7.3 Correctness of Algorithm 16 postcondition (E4)

We begin by showing that when $\delta = 1$, the loop body spanning lines 6-9 is executed precisely once. When $\delta = 1$, *to_remove* > 0 is clearly true, and so at least one loop iteration occurs.

From Corollary 5.4.6, we have that for every activation of line 6, $|\mathcal{Y}| > 0$. From this if follows that lines 7-9 always cause *to_remove* to be decremented by at least 1. When $\delta = 1$, this ensures that after the first iteration of the loop, *to_remove* is set to 0.

We now show that when the loop body executed precisely one time, this algorithms returns $\mathcal{Z} = \mathcal{X} \setminus \{y\}$, where there is a non-zero probability that y is any of the vertices in $\mathcal{V} \setminus \mathcal{X}$ such that $\mathcal{X} \setminus \{y\}$ is a convex set of D.

Observe that in the first (and in this case, only) iteration of the loop, we have $\mathcal{Z} = \mathcal{X}$. Therefore line 6 computes the formula $\mathcal{Y} = \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$. Theorem 5.4.7 establishes that this is the precise vertex set such that the deletion of any one of them from $\mathcal{X}$ yields another convex set of D.

Our goal now is to show that for any $y \in \mathcal{Y}$, there is a non-zero probability that $\mathcal{Z} = \mathcal{X} \setminus \{y\}$. This comes from our definition of the `vset_random_subset` operation, which in Appendix A.4 is defined to have a non-zero probably of returning any subset of the specified order.

C.8 Shrinking Convex Sets Using Topological Sort

Given a convex set $\mathcal{X}$, this algorithm returns some superset of $\mathcal{X}$ having $|\mathcal{X}| + \delta$ vertices.

Algorithm 17 Shrink Convex Set Using Topological Sort

```
Function shrink_convex_set_TS : $(D, \mathcal{X}, \delta) \to \mathcal{Z}$
Require:
    (R1)    $D = (\mathcal{V}, A)$ is a DAG.
    (R2)    $\emptyset \subset \mathcal{X} \subseteq V$ is a convex set of $D$.
    (R3)    $|\mathcal{X}| > 1$
    (R4)    $1 \leq \delta < |\mathcal{X}|$
Ensure:
    (E1)    $\mathcal{Z}$ is a convex set of $D$.
    (E2)    $\mathcal{Z} \subset \mathcal{X}$.
    (E3)    $|\mathcal{Z}| = |\mathcal{X}| - \delta$
    (E4)    Let $u \notin \mathcal{X}$ be any vertex such that $\mathcal{X} \cup \{v\}$ is a convex set of $D$.
            Then if $\delta = 1$, there is a non-zero probability that this algorithm
            returns with $\mathcal{Z} = \mathcal{X} \setminus \{v\}$.

1: $D{<}\mathcal{X}{>} \leftarrow$ call induce_subdigraph( $D, \mathcal{X}$ )                 $O(|\mathcal{V}|^2)$
2: $\vec{U} \leftarrow$ call dag_rand_topo_sort( $D{<}\mathcal{X}{>}$ )                     $O(|\mathcal{V}|^2)$
3: $\vec{Z} \leftarrow$ call vseq_random_slice( $\vec{U}, |\mathcal{X}| - \delta$ )          $O(|\mathcal{V}|)$
4: $\mathcal{Z} \leftarrow$ call vseq_to_vset( $\vec{Z}$ )                                    $O(|\mathcal{V}|)$
5: return $\mathcal{Z}$
```

From inspection the running time of this algorithm is clearly $O(|\mathcal{V}|^2)$.

C.8.1 Correctness of Algorithm 17 postconditions (E1)-(E3)

Lines 1-2 compute a topological sort of $D{<}\mathcal{X}{>}$, from which a subsequence is drawn in line 3.

As shown in the discussion of Algorithm 15, it is possible to compute a topological sort $\vec{T}$ of all of D, where $\vec{U}$ is a subsequence of $\vec{T}$. However, determining a topological sort for all of D is unnecessary for Algorithm 17, because only the relative ordering of elements in $\mathcal{X}$ is relevant here.

Because $\vec{Z}$ is a contiguous subsequence in of some topological sort of D, $\mathcal{Z}$ is a convex set of D (see Theorem 4.2.1), satisfying postcondition (E1).

$\vec{U}$ is a total ordering of $\mathcal{X}$, and so any proper subsequence of $\vec{U}$ is an ordering of a proper subset of $\mathcal{X}$. Because $delta > 0$, line 3 establishes $\vec{Z}$ as one such proper subsequence of $\vec{U}$, and therefore $\mathcal{Z} \subset \mathcal{X}$, satisfying postcondition (E2).

Postcondition (E3) is obtained directly by the definition of
`vseq_random_slice` (see Appendix A.6).

C.8.2 Correctness of Algorithm 17 postcondition (E4)

From Theorem 5.4.7 we have that $\mathcal{X} \setminus \{y\}$ is a convex set of D if and only if $y \in \mathcal{X} \setminus (\hat{P}(\mathcal{X}) \cap \hat{S}(\mathcal{X}))$. It follows then that either $y \notin \hat{P}(\mathcal{X})$, and/or $y \notin \hat{S}(\mathcal{X})$.

If $y \notin \hat{P}(\mathcal{X})$, then there must be some topological sort of $D{<}\mathcal{X}{>}$ in which y is the last element of the sequence. If no such topological sort existed, it would indicate that $D{<}\mathcal{X}{>}$ contains a $(y, \mathcal{X} \setminus \{y\})$ arc, contradicting our assumption that $y \notin \hat{P}(\mathcal{X})$.

For similar reasons, if $y \notin \hat{S}(\mathcal{X})$, then there must exist a topological sort of $D{<}\mathcal{X}{>}$ in which y appears as the first element of the topological sort.

The function `dag_rand_topo_sort` is defined to have a non-zero probability of returning any topological sort of the supplied DAG, and therefore has a non-zero probability of returning a topological sort with y as the first or last element.

When $\delta = 1$, `vseq_random_slice` is constrained to return the subsequence of $\vec{U}$ containing all elements of $\vec{U}$ except either the first or last. Furthermore, `vseq_random_slice` is defined to have a non-zero probability of yielding either subsequence. Therefore regardless of whether y appears as the first or last element of $\vec{U}$, there is a non-zero probability of $\mathcal{Z} = \mathcal{X} \setminus \{y\}$.

APPENDIX D

Arbitrary Transformation Algorithms

Below we present the convex set transformation algorithms supporting the arbitrary transformation of convex sets as discussed in Chapter 6.

D.1 Transforming Convex Set into a Single Vertex with Convex Intermediate Sets

Given a convex set $\mathcal{X}$ of some DAG D, Algorithm 18 discovers a sequence $\overline{\mathcal{R}}$ of single-vertex removals that may be progressively applied to $\mathcal{X}$, such that the set resulting from each single-vertex removal is itself a convex set of D. That is, for any $n \in [1, |\overline{\mathcal{R}}|]$, removing from $\mathcal{X}$ the first n vertices of $\overline{\mathcal{R}}$ yields a convex set.

Algorithm 18 Transform Convex Set into a Single Vertex with Convex Intermediate Sets

```
Function evolve_convex_set_to_vertex : $(D, \mathcal{X}) \rightarrow \overline{\mathcal{R}}$
Require:
    (R1)    $D = (\mathcal{V}, A)$ is a DAG.
    (R2)    $\emptyset \subset \mathcal{X} \subset \mathcal{V}$
Ensure:
    (E1)    $\overline{\mathcal{R}}[\![1]\!] = \mathcal{X}$
    (E2)    $|\overline{\mathcal{R}}[\![\ |\overline{\mathcal{R}}|\ ]\!]| = 1$
    (E3)    $\forall_{i \in 1 \ldots |\mathcal{X}|-1}, \exists u \in (\mathcal{V} \setminus \mathcal{R}[\![i]\!])$ such that $\mathcal{R}[\![i+1]\!] = \mathcal{R}[\![i]\!] \setminus \{u\}$

1: $\vec{Q} \leftarrow$ call topsort_with_embedded_cvx_set( $D, \mathcal{X}$ )
2: $(x\_low, x\_high) \leftarrow$ call vseq_find_bounds( $\vec{Q}, \mathcal{X}$ )
3: $\overline{\mathcal{R}} \leftarrow \emptyset$
4: for $i = x\_low$ to $x\_high$ do
5:     $\vec{Y} \leftarrow \vec{Q}[\![i \ldots x\_high]\!]$
6:     $\mathcal{Y} \leftarrow$ call vseq_to_vset( $\vec{Y}$ )
7:     call vss_append( $\overline{\mathcal{R}}, \mathcal{Y}$ )
8: end for
9: return $\overline{\mathcal{R}}$
```

D.1.1 Correctness of Algorithm 18

The correctness of this algorithm is rooted in the correctness of `topsort_with_embedded_cvx_set` and Theorem 4.2.1.

For any given convex set such as $\mathcal{X}$, we can use `topsort_with_embedded_cvx_set` to produce a topological sort $\vec{Q}$ of D in which the elements $\mathcal{X}$ form a contiguous subsequence of $\vec{Q}$. This is accomplished by line 1.

Lines 2-8 cause each $\overline{\mathcal{R}}[\![i]\!] \in \overline{\mathcal{R}}$ to contain the vertices from a contiguous subsequence of $\vec{Q}$. Because each $\overline{\mathcal{R}}[\![i]\!]$ is a contiguous subsequence of some topological sort of D, it is a convex set of D (see Theorem 4.2.1).

When $i = x\_low$, line 5 gives us $\overline{\mathcal{R}}[\![1]\!] = \overline{\mathcal{Q}}[\![x\_low \ldots x\_high]\!]$, satisfying postcondition (E1).

When $i = x\_high$, line 5 gives us $\overline{\mathcal{R}}[\![\ |\overline{\mathcal{R}}|\ ]\!] = \{\vec{Q}[\![x\_high]\!]\}$, satisfying postcondition (E2).

Now consider each iteration of the loop spanning lines 4-8. i is incremented by exactly one during each loop iteration. From this it follows that for the i^{th} iteration of the loop, where $i > 1$, $\mathcal{Y}$ is the same set as was produced by the previous iteration, except that the vertex $\vec{Q}[\![i-1]\!]$ has been deleted. This ensures postcondition (E3).

D.2 Transforming between Arbitrary Convex Sets

Algorithm 19 provides a constructive proof that given any two convex sets $\mathcal{X}$ and $\mathcal{Y}$ of some digraph D, one can always evolve $\mathcal{X}$ into $\mathcal{Y}$ using a sequence of single-vertex additions, removals, and/or replacements, such that the set produced by each of those single-vertex changes is also itself convex.

Algorithm 19 Transform between Arbitrary Convex Sets

```
Function evolve_between_convex_sets : $(D, \mathcal{X}, \mathcal{Y}) \rightarrow \overline{\mathcal{T}}$
Require:
    (R1)    $D = (\mathcal{V}, A)$ is a DAG.
    (R2)    $\emptyset \subset \mathcal{X} \subseteq V$ is a convex set of $D$.
    (R3)    $\emptyset \subset \mathcal{Y} \subseteq V$ is a convex set of $D$.
Ensure:
    (E1)    Each vertex set in $\overline{\mathcal{T}}$ is a convex set of $D$
    (E2)    $\overline{\mathcal{T}}[\![1]\!] = \mathcal{X}$
    (E3)    $\overline{\mathcal{T}}[\![\ |\overline{\mathcal{T}}|\ ]\!] = \mathcal{Y}$
    (E4)    $\forall_{i<|\overline{\mathcal{T}}|}(|\overline{\mathcal{T}}[\![i+1]\!]| < |\overline{\mathcal{T}}[\![i]\!]|) \implies \exists u \in \overline{\mathcal{T}}[\![i]\!]$ such that
            $\overline{\mathcal{T}}[\![i+1]\!] = \overline{\mathcal{T}}[\![i]\!] \setminus \{u\}$
    (E5)    $\forall_{i<|\overline{\mathcal{T}}|}(|\mathcal{T}_{i+1}| = |\mathcal{T}_i|) \implies (|\mathcal{T}_{i+1}| = |\mathcal{T}_i| = 1) \wedge (\mathcal{T}_{i+1} \neq \mathcal{T}_i)$
    (E6)    $\forall_{i<|\overline{\mathcal{T}}|}(|\overline{\mathcal{T}}[\![i+1]\!]| > |\overline{\mathcal{T}}[\![i]\!]|) \implies \exists u \in (\mathcal{V} \setminus \overline{\mathcal{T}}[\![i]\!])$ such that
            $\overline{\mathcal{T}}[\![i+1]\!] = \overline{\mathcal{T}}[\![i]\!] \cup \{u\}$

1: $\overline{\mathcal{Q}} \leftarrow$ *call* evolve_convex_set_to_vertex( $D, \mathcal{X}$ )
2: $\overline{\mathcal{R}} \leftarrow$ *call* evolve_convex_set_to_vertex( $D, \mathcal{Y}$ )
3: $\overline{\mathcal{S}} \leftarrow$ *call* vss_reverse( $\overline{\mathcal{R}}$ )
4: if $\overline{\mathcal{Q}}[\![\ |\overline{\mathcal{Q}}|\ ]\!] = \overline{\mathcal{S}}[\![1]\!]$ then
5:     $\overline{\mathcal{T}} = [\overline{\mathcal{Q}},\ \overline{\mathcal{S}}[\![2 \ldots]\!]]$
6: else
7:     $\overline{\mathcal{T}} = [\overline{\mathcal{Q}},\ \overline{\mathcal{S}}]$
8: end if
9: return $\overline{\mathcal{T}}$
```

D.2.1 Correctness of Algorithm 19

We consider each of the algorithm's postconditions in turn.

Every vertex set in $\overline{\mathcal{T}}$ is one of the vertex sets produced by `evolve_convex_set_to_vertex` (Algorithm 18), and is therefore a convex set of D. Thus postcondition (E1) holds.

The first vertex set of $\overline{\mathcal{T}}$ is also the first vertex set of $\overline{\mathcal{Q}}$, which `evolve_convex_set_to_vertex` ensures has the value $\mathcal{X}$. This satisfies postcondition (E2).

The last element $\overline{\mathcal{T}}$ is the first element of $\overline{\mathcal{S}}$, which in turn is the last element of $\overline{\mathcal{R}}$. `evolve_convex_set_to_vertex` ensures that that vertex set is $\mathcal{Y}$. Therefore postcondition (E3) is satisfied.

We now consider postconditions (E4)-(E6).

Recall that `evolve_convex_set_to_vertex` returns a sequence of vertex sets such that for elements i and $(i+1)$ of some returned sequence $\overline{\mathcal{Z}}$, we have $|\overline{\mathcal{Z}}[\![i+1]\!]| = |\overline{\mathcal{Z}}[\![i]\!]| - 1$. That is, as one proceeds through the sequence, each vertex set in the sequence is one vertex smaller than the previous. This corresponds to the antecedent predicate in postcondition (E4).

Because $\overline{\mathcal{S}}$ is the reversal of a sequence produced by `evolve_convex_set_to_vertex`, in general for the elements of $\overline{\mathcal{S}}$ we have $|\overline{\mathcal{S}}[\![i+1]\!]| = |\overline{\mathcal{S}}[\![i]\!]| + 1$. This corresponds to the antecedent predicate in postcondition (E6).

We consider two possibilities, corresponding to the test on line 4 of the algorithm.

Case 1: $\overline{\mathcal{Q}}[\![\ |\overline{\mathcal{Q}}|\ ]\!] = \overline{\mathcal{S}}[\![1]\!]$*:*

In Case 1, lines 1 and 2 of the algorithm yielded values for $\overline{\mathcal{Q}}$ and $\overline{\mathcal{R}}$, respectively, in which both vertex sets evolved down to the same single-vertex set. That is, for some

$u \in (\mathcal{X} \cap \mathcal{Y})$, $\overline{\mathcal{Q}}[\![\ |\overline{\mathcal{Q}}|\ ]\!] = \{u\}$ and $\overline{\mathcal{R}}[\![\ |\overline{\mathcal{R}}|\ ]\!] = \{u\}$:

$$\overline{\mathcal{T}} = [\ \underbrace{\overline{\mathcal{Q}}[\![1\ldots|\overline{\mathcal{Q}}|-1]\!],\ \{u\}}_{\overline{\mathcal{Q}}},\ \underbrace{\overline{\mathcal{S}}[\![2\ldots|\overline{\mathcal{S}}|]\!]}_{\overline{\mathcal{S}}}\ ]$$

It follows then that the only adjacent pairs of vertices $(\overline{\mathcal{T}}[\![i]\!], (\overline{\mathcal{T}}[\![i+1]\!])$ to which postcondition (E4) applies are those pairs with lie entirely within the part of $\overline{\mathcal{T}}$ that is identical to $\overline{\mathcal{Q}}$. Therefore, every adjacent pair of vertices in $\overline{\mathcal{T}}$ to which postcondition (E4) applies is known to satisfy postcondition (E4) because all of $\overline{\mathcal{Q}}$ satisfies postcondition (E4).

Postcondition (E6) is satisfied by similar reasoning, with $\overline{\mathcal{S}}$ rather than $\overline{\mathcal{Q}}$.

Postcondition (E5) does not pertain to Case 1, because there is no pair of adjacent elements $(\overline{\mathcal{T}}[\![i]\!], \overline{\mathcal{T}}[\![i+1]\!])$ in $\overline{\mathcal{T}}$ such that $|\overline{\mathcal{T}}[\![i]\!]| = |\overline{\mathcal{T}}[\![i+1]\!]|$.

Case 2: $\overline{\mathcal{Q}}[\![\ |\overline{\mathcal{Q}}|\ ]\!] \neq \overline{\mathcal{S}}[\![1]\!]$*:*

In Case 2, from line 7 of the algorithm we have:

$$\overline{\mathcal{T}} = [\overline{\mathcal{Q}}[\![1\ldots|\overline{\mathcal{Q}}|]\!],\ \overline{\mathcal{S}}[\![1\ldots|\overline{\mathcal{S}}|]\!]]$$

As discussed in Case 1 above, the first $|\overline{\mathcal{Q}}|$ elements of $\overline{\mathcal{T}}$ comprise the only portion of $\overline{\mathcal{T}}$ to which postcondition (E5) applies, and is satisfied by that portion's equality with $\overline{\mathcal{Q}}$.

Similarly, the remaining portion of $\overline{\mathcal{T}}$ is the only portion to which postcondition (E6) applies, and is satisfied by that portion's equality with $\overline{\mathcal{S}}$.

Unlike Case 1 above, postcondition (E5) does pertain to Case 2. Both the final element of $\overline{\mathcal{Q}}$ (i.e., $\overline{\mathcal{T}}[\![|\overline{\mathcal{Q}}|]\!]$) and the first element of $\overline{\mathcal{S}}$ (i.e., $\overline{\mathcal{T}}[\![|\overline{\mathcal{Q}}|+1]\!]$) containing just one vertex. It is this pair of vertices to which postcondition (E5) applies. However, this if-then test on line 4 ensures that in Case 2, we have $\overline{\mathcal{T}}[\![|\overline{\mathcal{Q}}|]\!] \neq \overline{\mathcal{T}}[\![|\overline{\mathcal{Q}}|+1]\!]$.

APPENDIX E

Task Set Deconfliction

This appendix presents a pair of algorithms to modify a task set P such that no two tasks overlap, and that the cycle-induction problem described in Subsection 3.3.1.3 is avoided.

Suppose $D = (\mathcal{V}, A)$ is a DFG, and $P = \{\mathcal{T}_1,\ \mathcal{T}_2,\ \ldots,\ \mathcal{T}_n\}$. For each set $\mathcal{T}_i \in P$, let $\emptyset \subset \mathcal{T}_i \subseteq \mathcal{V}$ and let $\mathcal{T}_i$ be a convex set of D.

Here we make no assumption that the sets in P are mutually disjoint (that is, $\forall i \neq i, \mathcal{T}_i \cap \mathcal{T}_j = \emptyset$). We also make no assumption that if P were translated to a collection of graphs $\{D_{parent},\ D_{child\_1},\ D_{child\_2},\ \ldots,\ D_{child\_m}\}$, that D_{parent} would be free of cycles.

E.1 Obtaining Disjoint Tasks

One approach to ensuring that independently evolved convex sets do not overlap is as follows.

Let $\mathcal{X}$ and $\mathcal{Y}$ be two convex sets of some DFG, in which $\mathcal{X} \cap \mathcal{Y} \neq \emptyset$. Either $\mathcal{X}$ or $\mathcal{Y}$ is chosen as a "victim" set. Let us assume that $\mathcal{Y}$ is chosen as the victim. Vertices shall be deleted from $\mathcal{Y}$ to yield a set $\mathcal{Y}'$, such that $\mathcal{Y}'$ is a convex set and $\mathcal{X} \cap \mathcal{Y}' = \emptyset$.

Note that, but the definition of convex sets, the empty set is not a convex set. However, it may be necessary for the victim set to become the empty set in order to eliminate overlap. This is most easily seen in the example $\mathcal{X} = \mathcal{Y} = \{v\}$ for some vertex v.

We assert but do not prove that either of the two formulae in Equation (E.18) ensure that $\mathcal{Y}'$ is a subset of $\mathcal{Y}$', and is either a convex set of the DFG or is the empty set.

$$\begin{array}{rcl} \mathcal{Y}' & = & \mathcal{Y} \setminus ((\mathcal{X} \cap \mathcal{Y}) \cup \hat{P}(\mathcal{X} \cap \mathcal{Y}))) \\ \mathcal{Y}' & = & \mathcal{Y} \setminus ((\mathcal{X} \cap \mathcal{Y}) \cup \hat{S}(\mathcal{X} \cap \mathcal{Y}))) \end{array} \tag{E.18}$$

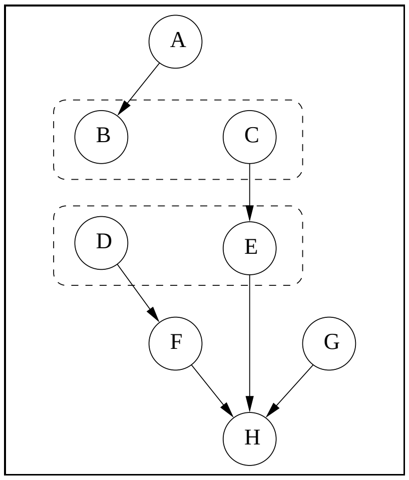
Informally, the formulae in Equation (E.18) may be understood as follows. By removing $\mathcal{X} \cap \mathcal{Y}$ from $\mathcal{Y}$, we potentially introduce concavity in the resulting set, $\mathcal{Y} \setminus (\mathcal{X} \cap \mathcal{Y})$. This is because there may exist a path in the DFG which originates within $\mathcal{Y} \setminus (\mathcal{X} \cap \mathcal{Y})$, then passes through $\mathcal{X} \cap \mathcal{Y}$, and then re-enters $\mathcal{Y} \setminus (\mathcal{X} \cap \mathcal{Y})$. Any such path is eliminated by deleting from $\mathcal{Y}$ not just the vertices $\mathcal{X} \cap \mathcal{Y}$, but also any vertices which complete a cycle between $\mathcal{X} \cap \mathcal{Y}$ and any vertices which are to remain in $\mathcal{Y}'$. This cycle may be broken by further removing either $\hat{P}(\mathcal{X} \cap \mathcal{Y})$ or $\hat{S}(\mathcal{X} \cap \mathcal{Y})$, which is why two variations of the this approach are presented in Equation (E.18).

This approach for eliminating overlap within a pair of convex sets may be generalized to eliminating any such pairwise overlap of convex sets in a task set of arbitrary order. Algorithm 20 presents one such algorithm.

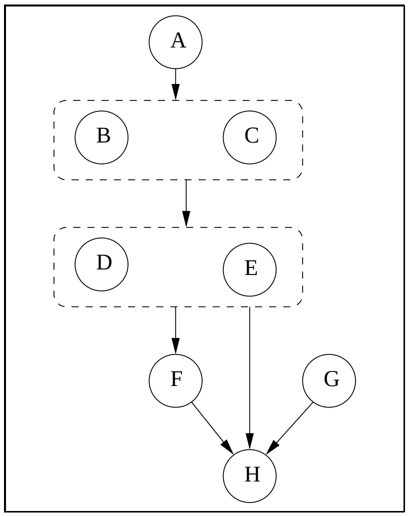
Algorithm 20 Elimination of Overlap in Multiple Convex Sets

```
Function eliminate_overlap : $(D, P) \rightarrow P'$
Require:
    (R1)    $D = (\mathcal{V}, A)$ is a DAG.
    (R2)    $P = \{\mathcal{T}_1 \dots \mathcal{T}_n\}$ is a set of tasks of $D$.
Ensure:
    (E1)    $P' = \{\mathcal{T}'_1 \dots \mathcal{T}'_n\}$ is a set of tasks of $D$.
    (E2)    Each vertex set in $P'$ is a convex set of $D$ or is the empty set.
    (E3)    $\forall i \in 1 \dots n, \mathcal{T}'_i \subseteq \mathcal{T}_i$

 1: $P' \leftarrow$ call vset_clone( $P$ )
 2: for $i = 1$ to $n - 1$ do
 3:   for $j = i + 1$ to $n$ do
 4:     $\mathcal{X} \leftarrow P'[\![i]\!]$
 5:     $\mathcal{Y} \leftarrow P'[\![j]\!]$
 6:     $\mathcal{I} \leftarrow \mathcal{X} \cap \mathcal{Y}$
 7:     if $\mathcal{I} \neq \emptyset$ then
 8:       $P'[\![j]\!] \leftarrow \mathcal{Y} \setminus (\mathcal{I} \cup \hat{P}(\mathcal{I})))$
 9:     end if
10:   end for
11: end for
12: return $P'$
```



(a) DFG D with the task set $P' = \{\mathcal{T}'_1,\ \mathcal{T}'_2\}$, and tasks $\mathcal{T}'_1 = \{B, C\}$, $\mathcal{T}'_2 = \{D, E\}$.



(b) For hyper-$\hat{P}$, if even one vertex u of some task $\mathcal{T}'_i$ precedes a vertex v, then *all* vertices in $\mathcal{T}'_i$ are considered to precede v.

Figure E.7: Example of hyper-$\hat{P}$. $\hat{P}(\{F\}) = \{D\}$. However, hyper-$\hat{P}(\{F\}) = \{A,\ B,\ C,\ D,\ E\}$.

E.2 Preventing Cycle-Induction during Translation

Algorithm 21 assumes that the input sets are individually convex sets of D, and are collectively disjoint with each other. It produces a new collection of sets which are guaranteed to be free of the cycle-induction problem described in Subsection 3.3.1.3.

E.2.1 Hyper-predecessor Sets and Hyper-successor Sets

Algorithm 21 makes use of the concepts of **hyper-$\hat{P}$** and **hyper-$\hat{S}$** sets, described below. An illustration of hyper-$\hat{P}$ is provided in Figure E.7.

Let $D = (\mathcal{V}, A)$ be a DFG. Recall that the predecessor set of some vertex set $\mathcal{X}$, denoted $\hat{P}(\mathcal{X})$, is the collection of all vertices $u \notin \mathcal{X}$ such that a $(u, \mathcal{X})$ path exists within D (see Chapter 5).

Suppose that $P' = \{\mathcal{T}'_1 \dots \mathcal{T}'_n\}$ is a set of convex, mutually disjoint tasks of D, such as produced by Algorithm 20.

We define the hyper-$\hat{P}(\mathcal{X})$ as follows:

- If D contains an arc $(u, \mathcal{X})$, then every member of hyper-$\hat{P}(u)$ is also a member of hyper-$\hat{P}(\mathcal{X})$.
- If there exists a task $\mathcal{T}_i' \in P'$ such that $u \in \mathcal{T}_i'$, then for all $v \in \mathcal{T}_i'$, every vertex in hyper-$\hat{P}(v)$ is also a member of hyper-$\hat{P}(\mathcal{X})$.

Hyper-$\hat{S}$ is defined in a similar manner, using successors rather than predecessors. In general, hyper-$\hat{P}(\mathcal{X}) \supseteq \hat{P}(\mathcal{X})$, and hyper-$\hat{S}(\mathcal{X}) \supseteq \hat{S}(\mathcal{X})$.

Hyper-$\hat{P}/\hat{S}$ are defined so as to anticipate the (non-hyper) $\hat{P}/\hat{S}$ sets that would arise from replacing, in D, every task P' with the corresponding pair of `SPAWN` ans `WAIT` vertices.

Note that these definitions depend upon D being acyclic, but remain well-defined even when if the graph resulting from replacing each task $\mathcal{T}_i' \in P'$ with a pair of `WAIT` and `SPAWN` vertices contains a cycle.

E.2.2 Hyper-Predecessor/Successor Sets as Cycle Predictors

Algorithm 21 produces a set of tasks $P'' = \{T_1'', T_2'', \ldots, T_m''\}$.

This crucial difference between the supplied task set P' and P'' is as follows. If the DFG D is modified by replacing each task $\mathcal{T}_i' \in P'$ with the two vertices `SPAWN(`$\mathcal{T}_i'$`)` and `WAIT(`$\mathcal{T}_i'$`)` , the resulting graph D_{parent} may contain a cycle, as described in Subsection 3.3.1.3.

In contrast, if D is modified by replacing each task in P'', rather than each task in P', with `SPAWN(`$\mathcal{T}_i'$`)` and `WAIT(`$\mathcal{T}_i'$`)` vertices, the resulting graph D_{parent} is guaranteed to by acyclic, solving the cycle-induction problem described in Subsection 3.3.1.3.

The insight which drives this algorithm's design is the following: Suppose P is a task set for some DFG D. We assert but only informally prove the following: If P contains two tasks $\mathcal{T}_i$ and $\mathcal{T}_j$ such that

$$\textit{hyper–}\hat{P}(\mathcal{T}_i) \cap \textit{hyper–}\hat{S}(\mathcal{T}_i) \cap \mathcal{T}_j \neq \emptyset \tag{E.19}$$

then producing D_{parent} as described above will cause D_{parent} to contain a cycle.

That proposition is based on the observation that hyper-$\hat{P}(\mathcal{T}_i)$, as computed in the context of the DFG D, is identical to the set $\hat{P}(\texttt{SPAWN(}\mathcal{T}_i\texttt{)})$ as computed in the graph D_{parent}.

We may similarly observe that hyper-$\hat{S}(\mathcal{T}_i)$, as computed in the context of the DFG D, is identical to the set $\hat{S}(\texttt{WAIT(}\mathcal{T}_i\texttt{)})$ as computed in the graph D_{parent}.

Let D^i_{parent} be the graph produced by replacing $\mathcal{T}_i$ with $\texttt{SPAWN(}\mathcal{T}'_i\texttt{)}$ and $\texttt{WAIT(}\mathcal{T}'_i\texttt{)}$ in D. Further assume that D^i_{parent} still contains the vertices $\mathcal{T}_j$ (i.e., $\mathcal{T}_j$ has not yet been replaced by $\texttt{SPAWN(}\mathcal{T}'_j\texttt{)}$ and $\texttt{WAIT(}\mathcal{T}'_j\texttt{)}$ vertices in D^i_{parent}).

Let $\hat{P}_D$ and $\hat{S}_D$ be the functions $\hat{P}$ and $\hat{S}$, respectively, evaluated in the context of the DFG D. Let $\hat{P}_{D^i_{parent}}$ and $\hat{S}_{D^i_{parent}}$ be the functions $\hat{P}$ and $\hat{S}$, respectively, evaluated in the context of the graph D^i_{parent}. Then we have the following:

$$\textit{hyper–}\hat{P}_D(\mathcal{T}_i) \cap \textit{hyper–}\hat{S}_D(\mathcal{T}_i) = \hat{P}_{D^i_{parent}}(\texttt{SPAWN(}\mathcal{T}_i\texttt{)}) \cap \hat{S}_{D^i_{parent}}(\texttt{WAIT(}\mathcal{T}_i\texttt{)}) \tag{E.20}$$

From Equation (E.20) it follows that ($\mathcal{T}_j$ intersects both hyper-$\hat{P}(\mathcal{T}_i)$ and hyper-$\hat{S}(\mathcal{T}_i)$ in the DFG D) if any only if ($\mathcal{T}_j$ intersects both $\hat{P}(\texttt{SPAWN(}\mathcal{T}_i\texttt{)})$ and $\hat{S}(\texttt{WAIT(}\mathcal{T}_i\texttt{)})$ in the graph D^i_{parent}).

From this one can extend the principles used in Theorem 5.3.3 to show that further modifying D^i_{parent} by replacing $\mathcal{T}_j$ with the vertices $\texttt{SPAWN(}\mathcal{T}_j\texttt{)}$ and $\texttt{WAIT(}\mathcal{T}_j\texttt{)}$ would induce a cycle in the graph.

Our conclusion then is the following.

- Let D be a DFG, and let P' be a task set with the qualities ensured by Algorithm 20.
- Let D^i_{parent} be the DFG produced by replacing some $\mathcal{T}'_i \in P'$ with the vertices `SPAWN(`$\mathcal{T}_i$`)` and `WAIT(`$\mathcal{T}_i$`)` .
- Let $\mathcal{T}'_j \in P'$ be a task which has not yet been replaced by the vertices `SPAWN(`$\mathcal{T}_j$`)` and `WAIT(`$\mathcal{T}_j$`)` in D^i_{parent}.
- Let D^j_{parent} be the graph produced by replacing $\mathcal{T}'_j$ with the vertices `SPAWN(`$\mathcal{T}_j$`)` and `WAIT(`$\mathcal{T}_j$`)` in D^i_{parent}.
- Then D^j_{parent} contains a cycle if and only if $\mathit{hyper}\text{–}\hat{P}_D(\mathcal{T}_i) \cap \mathit{hyper}\text{–}\hat{S}_D(\mathcal{T}_i) \cap \mathcal{T}_j \neq \emptyset$.

In summary, once can use the hyper-$\hat{P}$ and hyper-$\hat{S}$ operators to anticipate whether or not two tasks are destined to participate in a parent-graph cycle should all of the task set's tasks be replaced with `SPAWN` and `WAIT` vertices.

E.2.3 Algorithm Overview

Algorithm 21 visits every pair of tasks in the supplied task set P'. For each pair of tasks, Equation (E.19) is used to predict whether or not those two tasks will ultimately participate in the creation of a parent-graph cycle.

If a cycle is indicated, then vertices are deleted from one of the two tasks (the "victim" task) such that Equation (E.19) no longer indicates that a cycle would be formed.

Line 9 of the algorithm is certain to produce a subset of $\mathcal{Y}$ which not only has no cycle with $\mathcal{X}$, but is also a convex set of D. This is because $HSX \cap \mathcal{Y} = \hat{P}(\mathcal{Y})$, and removing all predecessors or all successors of a set ensures that what remains is acyclic.

E.2.4 Algorithm Observations

Algorithm 21 has several noteworthy qualities, in light of its intended use to ensure that machine-learned task sets can be translated into task-parallel programs.

The first observation is that this algorithm may be systematically biased toward modifying high-numbered tasks within the supplied task set P', because when two tasks are found to be in conflict, the higher-numbered task is chosen as the victim. The impact of this bias on the performance of the overall machine-learning system is unclear.

Secondly, without further study it's not obvious that Algorithm 21 makes the smallest-possible modifications to the tasks in P' in order to obtain P''. It may in fact be the case that the algorithm's statement $P''[\![j]\!] \leftarrow \mathcal{Y} \setminus HSX$ is equivalent to simply deleting the victim task, $P''[\![j]\!]$. This possibly over-aggressive modification of the victim task may lead to small changes in the task set P' causing unnecessarily large and unintuitive modifications in the modified task set P''. It may be the case that this causes avoidable performance problems in the machine-learning system which evolves the task sets.

Algorithm 21 Eliminating Cycle-Induction by Multiple Convex Sets

Function `eliminate_cycle_induction` : $(D,\, P') \rightarrow P''$

Require:

(R1) $D = (\mathcal{V}, A)$ is a DAG.

(R2) $P' = \{\mathcal{T}_1' \ldots \mathcal{T}_n'\}$ is a task set.

(R3) Each task in $\mathcal{T}_i \in P'$ is a convex set of D, or is the empty set.

(R4) $\forall i \neq j, \mathcal{T}_i' \cap \mathcal{T}_j' = \emptyset$

Ensure:

(E1) Each task in $P'' = \{\mathcal{T}_1'' \ldots \mathcal{T}_n''\}$ is a convex set of D, or is the empty set.

(E2) $\forall i \in 1 \ldots n, \mathcal{T}_i'' \subseteq \mathcal{T}_i'$

(E3) The graph formed by replacing every set $\mathcal{T}_i' \in P'$ with the vertex pair `SPAWN(`$\mathcal{T}_i$`)` and `WAIT(`$\mathcal{T}_i$`)`, as described in Subsection 3.3.1.1, is acyclic.

```
1: $P'' \leftarrow$ call vset_clone( $P'$ )
2: for $i = 1$ to $n - 1$ do
3:    for $j = i + 1$ to $n$ do
4:       $\mathcal{X} \leftarrow P''[\![i]\!]$
5:       $\mathcal{Y} \leftarrow P''[\![j]\!]$
6:       $HPX \leftarrow$ compute hyper-$\hat{P}(\mathcal{X})$ using current value of $P''$
7:       $HSX \leftarrow$ compute hyper-$\hat{S}(\mathcal{X})$ using current value of $P''$
8:       if $HPX \cap HSX \cap \mathcal{Y} \neq \emptyset$ then
9:          $P''[\![j]\!] \leftarrow \mathcal{Y} \setminus HSX$
10:      end if
11:   end for
12: end for
13: return $P''$
```

APPENDIX F

Experimentally Derived Source Code

```
#include <string_util.h>
#include <parse_util.h>
#include <tbb/tick_count.h>
#include <tbb/atomic.h>
#include <tbb/task_group.h>
#include <limits>
#include <cstdlib>
#include <memory>
#include <vector>
#include <cassert>
#include <sstream>
#include <iostream>

using namespace std;

tbb::atomic<int> randomizer_seed;

//==============================================================================

void verify_sort( unsigned int num_data, const vector<double> & v ) {
   if (v.size() != num_data) {
      ostringstream os;
      os << "verify_sort: num_data=" << num_data << ", but v->size()="
         << v.size();
      throw os.str();
   }

   if (num_data == 0) {
      return;
   }

   // Confirm that the vector data are strictly non-decreasing...
   double prev = v[0];
   for (unsigned int i = 1; i < v.size(); ++i) {
      if (v[i] < prev) {
         ostringstream os;
```

```
            os << "Unsorted data encountered: v[" << i << "]=" << v[i]
               << ", prev=" << prev;
            throw os.str();
        }
        prev = v[i];
    }
}

//==============================================================================
// C++ functions for vertices
//==============================================================================

void op_RANDOMIZE(
    int num_elements,
    std::shared_ptr<std::vector<double>> & output_receiver
    )
{
    std::shared_ptr<std::vector<double>> output( new std::vector<double> );
    output->reserve( num_elements );

    // Create a pseudo-random seed value.  This will lead to the same set
    // of seeds for all runs of a given DFG, but that's not a problem for
    // our application.  We're just generating load, anyway.
    unsigned int seed = randomizer_seed++;

    for (int i = 0; i < num_elements; ++i) {
        int r = rand_r( & seed );
        output->push_back( r / double(RAND_MAX) );
    }

    output_receiver = output;
}

//==============================================================================

static int less_than( const void * pa, const void * pb ) {
    const double a = *(reinterpret_cast<const double *>(pa));
    const double b = *(reinterpret_cast<const double *>(pb));
    if (a < b) {
        return -1;
    }
    else if (a > b) {
        return 1;
    }
    else {
        return 0;
    }
}

/// Performs an in-place sort of @c v.
void op_QUICKSORT(
    std::shared_ptr< std::vector<double>> & v1,
    std::shared_ptr< std::vector<double>> & output_receiver
    )
{
    assert(! v1->empty());
//      double * data = v1->begin();
    double * data = &(v1->operator[](0));
    qsort( data, v1->size(), sizeof(double), less_than );
    output_receiver = v1;
}


//==============================================================================

void op_MERGESORT(
    std::shared_ptr<std::vector<double>> v1,
    std::shared_ptr<std::vector<double>> v2,
    std::shared_ptr<std::vector<double>> & output_receiver
    )
{
    assert(! v1->empty());
    assert(! v2->empty());

    std::shared_ptr<std::vector<double>> v_out( new std::vector<double>() );
    v_out->reserve( v1->size() + v2->size() );

    auto p1 = v1->begin();
    auto p2 = v2->begin();

    // This could be done somewhat faster, but I don't really care.
    // We're just trying to generate CPU workload anyway.
    while ((p1 != v1->end()) || (p2 != v2->end())) {
        if (p1 == v1->end()) {
            v_out->push_back( *p2 );
            ++p2;
        }
        else if (p2 == v2->end()) {
            v_out->push_back( *p1 );
            ++p1;
        }
        else {
            // We need to actually compare numbers...
            if ( (*p1) < (*p2) ) {
                v_out->push_back(*p1);
                ++p1;
            }
            else {
                v_out->push_back(*p2);
                ++p2;
            }
        }
    }

    output_receiver = v_out;
}

//==========================================================================

void execute_DFG( int num_data, bool verify );

extern int NUMBER_OF_RANDOMIZE_VERTICES;

//==========================================================================

int main( int argc, const char* argv[] ) {
    try {
        if (argc != 3) {
            throw "Usage: dfg-sort <num-data> <true|false>  <--- verify "
                    "sortedness";
        }

        const int num_data = parse_int( argv[1] );
        const string verify_str = argv[2];

        bool verify;
        if (verify_str == "true") {
            verify = true;
        }
        else if (verify_str == "false") {
            verify = false;
        }
        else {
            throw "The 'verify' argument must be either 'true' or 'false'";
        }

        if (num_data < 1) {
            throw "<num-data> must be positive.";
        }

        if ((num_data % NUMBER_OF_RANDOMIZE_VERTICES) != 0) {
            ostringstream os;
            os << "<num-data> must be an integer multiple of "
               << NUMBER_OF_RANDOMIZE_VERTICES;
            throw os.str();
        }

        execute_DFG( num_data, verify );
    }
    catch (const char * e) {
        cerr << "Exception: " << e << endl;
        return 1;
    }
    catch (const string & e) {
        cerr << "Exception: " << e << endl;
        return 1;
    }

    return 0;
}

int NUMBER_OF_RANDOMIZE_VERTICES = 4;

//===========================================================================

void f1(
    shared_ptr<vector<double>> output_1,
    shared_ptr<vector<double>> output_8,
    shared_ptr<vector<double>> & output_2,
    shared_ptr<vector<double>> & output_9,
    int num_data,
    bool verify )
{
    // Vertex # 9
    op_QUICKSORT( output_8, output_9 );
    output_8.reset();

    // Vertex # 2
    op_QUICKSORT( output_1, output_2 );
    output_1.reset();
}

class Functor_f1 {
    public:
        Functor_f1(
            shared_ptr<vector<double>> & output_1,
            shared_ptr<vector<double>> & output_8,
            shared_ptr<vector<double>> & output_2,
            shared_ptr<vector<double>> & output_9,
            int num_data, bool verify ) :
            m_output_1( output_1 ),
            m_output_8( output_8 ),
            m_output_2( output_2 ),
            m_output_9( output_9 ),
            m_num_data( num_data ),
            m_verify( verify )
        {
        }

    void operator() ()
    {
        f1(
            m_output_1,
            m_output_8,
            m_output_2,
            m_output_9,
            m_num_data, m_verify );
    }

    private:
        shared_ptr<vector<double>> & m_output_1;
        shared_ptr<vector<double>> & m_output_8;
        shared_ptr<vector<double>> & m_output_2;
        shared_ptr<vector<double>> & m_output_9;
        int m_num_data;
        bool m_verify;
};

//==========================================================================

void f2(
    shared_ptr<vector<double>> output_3,
    shared_ptr<vector<double>> & output_4,
    shared_ptr<vector<double>> & output_7,
    int num_data,
    bool verify )
{
    // Vertex # 6
    shared_ptr<vector<double>> output_6;
    op_RANDOMIZE( (num_data / 4), output_6 );

    // Vertex # 7
    op_QUICKSORT( output_6, output_7 );
    output_6.reset();

    // Vertex # 4
    op_QUICKSORT( output_3, output_4 );
    output_3.reset();
}

class Functor_f2 {
    public:
        Functor_f2(
            shared_ptr<vector<double>> & output_3,
            shared_ptr<vector<double>> & output_4,
            shared_ptr<vector<double>> & output_7,
            int num_data, bool verify ) :
            m_output_3( output_3 ),
            m_output_4( output_4 ),
            m_output_7( output_7 ),
            m_num_data( num_data ),
            m_verify( verify )
        {
        }

    void operator() ()
    {
        f2(
            m_output_3,
            m_output_4,
            m_output_7,
            m_num_data, m_verify );
    }

    private:
        shared_ptr<vector<double>> & m_output_3;
        shared_ptr<vector<double>> & m_output_4;
        shared_ptr<vector<double>> & m_output_7;
        int m_num_data;
        bool m_verify;
};

//==========================================================================

void f3(
    shared_ptr<vector<double>> & output_8,
    int num_data,
    bool verify )
{
    // Vertex # 8
    op_RANDOMIZE( (num_data / 4), output_8 );
}

class Functor_f3 {
    public:
        Functor_f3(
            shared_ptr<vector<double>> & output_8,
            int num_data, bool verify ) :
            m_output_8( output_8 ),
            m_num_data( num_data ),
            m_verify( verify )
        {
        }

    void operator() ()
    {
        f3(
            m_output_8,
            m_num_data, m_verify );
    }

    private:
        shared_ptr<vector<double>> & m_output_8;
        int m_num_data;
        bool m_verify;
};

//==========================================================================

void f4(
    shared_ptr<vector<double>> output_2,
    shared_ptr<vector<double>> output_4,
    shared_ptr<vector<double>> & output_5,
    int num_data,
    bool verify )
{
    // Vertex # 5
    op_MERGESORT( output_2, output_4, output_5 );
    output_2.reset();
    output_4.reset();
}

class Functor_f4 {
    public:
        Functor_f4(
            shared_ptr<vector<double>> & output_2,
            shared_ptr<vector<double>> & output_4,
            shared_ptr<vector<double>> & output_5,
            int num_data, bool verify ) :
            m_output_2( output_2 ),
            m_output_4( output_4 ),
            m_output_5( output_5 ),
            m_num_data( num_data ),
            m_verify( verify )
        {
        }

    void operator() ()
    {
        f4(
            m_output_2,
            m_output_4,
            m_output_5,
            m_num_data, m_verify );
    }

    private:
        shared_ptr<vector<double>> & m_output_2;
        shared_ptr<vector<double>> & m_output_4;
        shared_ptr<vector<double>> & m_output_5;
        int m_num_data;
        bool m_verify;
};

//==============================================================================

void execute_DFG(
    int num_data,
    bool verify )
{
    // Vertex # 10003
    shared_ptr<vector<double>> output_8;
    tbb::task_group tgroup_3;
    Functor_f3 functor_instance_f3(
        output_8,
        num_data, verify );
    tgroup_3.run( functor_instance_f3 );

    // Vertex # 3
    shared_ptr<vector<double>> output_3;
    op_RANDOMIZE( (num_data / 4), output_3 );

    // Vertex # 10002
    shared_ptr<vector<double>> output_4;
    shared_ptr<vector<double>> output_7;
    tbb::task_group tgroup_2;
    Functor_f2 functor_instance_f2(
        output_3,
        output_4,
        output_7,
        num_data, verify );
    tgroup_2.run( functor_instance_f2 );

    // Vertex # 1
    shared_ptr<vector<double>> output_1;
    op_RANDOMIZE( (num_data / 4), output_1 );

    // Vertex # 10001
    tgroup_3.wait();
    shared_ptr<vector<double>> output_2;
    shared_ptr<vector<double>> output_9;
    tbb::task_group tgroup_1;
    Functor_f1 functor_instance_f1(
        output_1,
        output_8,
        output_2,
        output_9,
        num_data, verify );
    tgroup_1.run( functor_instance_f1 );

    // Vertex # 10004
    tgroup_1.wait();
    tgroup_2.wait();
    shared_ptr<vector<double>> output_5;
    tbb::task_group tgroup_4;
    Functor_f4 functor_instance_f4(
        output_2,
        output_4,
        output_5,
        num_data, verify );
    tgroup_4.run( functor_instance_f4 );

    // Vertex # 10
    shared_ptr<vector<double>> output_10;
    op_MERGESORT( output_7, output_9, output_10 );
    output_7.reset();
    output_9.reset();

    // Vertex # 11
    tgroup_4.wait();
    shared_ptr<vector<double>> output_11;
    op_MERGESORT( output_5, output_10, output_11 );
    output_5.reset();
    output_10.reset();


    if (verify) {
        verify_sort( num_data, *output_11);
    }
}
```

Listing F.1: Primary C++ source file generated by a run of the evolutionary algorithm run discussed in Chapter 8. Comments and formatting of this file have been manually adjusted for cosmetic purposes.

BIBLIOGRAPHY

Andersen, R., Gleich, D. F., and Mirrokni, V., "Overlapping clusters for distributed computation," in *Proceedings of the fifth ACM international conference on Web search and data mining - WSDM '12.* Assoc. of Computing Machinery, 2012.

Balister, P., Gerke, S., Gutin, G., Johnstone, A., Reddington, J., Scott, E., Soleimanfallah, A., and Yeo, A., "Algorithms for generating convex sets in acyclic digraphs," 2008.

Bang-Jensen, J. and Gutin, G., *Digraphs: Theory, Algorithms and Applications*, 2nd ed. Springer, 2009.

Beard, P., "An implementation of SISAL for distributed-memory architectures," Master's thesis, University of California, Davis, 1995.

Bertsimas, D. and Tsitsiklis, J., "Simluated annealing," *Statistical Science*, vol. 8, no. 1, pp. 10–15, 1993.

Cann, D. C., Feo, J., and Oldehoeft, R., "A report on the SISAL language project," *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, pp. 349–366, 1990.

Chan, C., Ansel, J., Wong, Y. L., Amarasinghe, S., and Edelman, A., "Autotuning multigrid with petabricks," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.* New York, NY, USA: ACM, 2009, pp. 1–12.

Cooper, K. D., Grosul, A., Harvey, T. J., Reeves, S., Subramanian, D., Torczon, L., and Waterman, T., "Acme: adaptive compilation made efficient," in *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems.* New York, NY, USA: ACM, 2005, pp. 69–77.

Duran, A., Corbalán, J., and Ayguadé, E., "An adaptive cut-off for task parallelism," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing.* Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.

Frigo, M. and Johnson, S., "FFTW: An adaptive software architecture for the FFT," in *IEEE International Conference on Acoustics Speech and Signal Processing*, vol. 3. Citeseer, 1998.

Frigo, M. and Johnson, S., "The design and implementation of FFTW 3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

Fursin, G. G., O'Boyle, M. F. P., and Knijnenburg, P. M. W., "Evaluating Iterative Compilation," in *Languages and Compilers for Parallel Computing, 15th Workshop, LCPC 2002*, ser. LNCS, no. 2481. Springer, 2005, pp. 362–376.

Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Bonilla, E., Thomson, J., Leather, H., Williams, C., O'Boyle, M., Barnard, P., Ashton, E., Courtois, E., and Bodin, F., "Milepost gcc: machine learning based research compiler," in *Proceedings of the GCC Developers' Summit*, July 2008. [Online]. Available: http://hal.inria.fr/inria-00294704/fr/

Huelsbergen, L., Larus, J. R., and Aiken, A., "Using the run-time sizes of data structures to guide parallel-thread creation," in *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming.* New York, NY, USA: ACM, 1994, pp. 79–90.

Iannucci, R. A., "A dataflow / von neumann hybrid architecture," Ph.D. dissertation, Massachusetts Inst. of Technology, 1988.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P., "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

Leung, Y., Gao, Y., and Xu, Z., "Degree of population diversity - a perspective on premature convergence in genetic algorithms and its markov chain analysis," *IEEE Transactions on Neural Networks*, vol. 8, pp. 1165–1176, 1997.

Mohr, E., Kranz, D. A., and Halstead, J. R. H., "Lazy task creation: a technique for increasing the granularity of parallel programs," in *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming.* New York, NY, USA: ACM, 1990, pp. 185–197.

Najjar, W., Roh, L., and Wim Böhm, A., "An evaluation of medium-grain dataflow code," *International Journal of Parallel Programming*, vol. 22, no. 3, pp. 209–242, 1994.

Olszewski, M. and Voss, M., "An install-time system for the automatic generation of optimized parallel sorting algorithms," in *Proceedings of PDPTA'04: The International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2004.

Pecero, J. and Bouvry, P., "An improved genetic algorithm for efficient scheduling on distributed memory parallel systems," in *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications - AICCSA 2010.* IEEE Computer Society, 2010.

Prechelt, L. and Hänßgen, S. U., "Efficient parallel execution of irregular recursive programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 2, pp. 167–178, 2002.

Pueschel, M., Moura, J., Singer, B., Xiong, J., Johnson, J., Padua, D., Veloso, M., and Johnson, R., "Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Alogorithms," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, 2004.

Roh, L., Najjar, W. A., and Böhm, A. P. W., "Generation and quantitative evaluation of dataflow clusters," in *FPCA '93: Proceedings of the conference on Functional* *programming languages and computer architecture.* New York, NY, USA: ACM, 1993, pp. 159–168.

Rudolph, G., "Convergence analysis of canonical genetic algorithms," *IEEE Transactions on Neural Networks*, vol. 5, pp. 96–101, 1994.

Rudolph, G., "Convergence of evolutionary algorithms in general search spaces," in *In Proceedings of the Third IEEE Conference on Evolutionary Computation.* IEEE Press, Piscataway, NJ, USA, 1996, pp. 50–54.

Rus, S., Pennings, M., and Rauchwerger, L., "Sensitivity analysis for automatic parallelization on multi-cores," in *ICS '07: Proceedings of the 21st annual international conference on Supercomputing.* New York, NY, USA: ACM, 2007, pp. 263–273.

Sanchez, J. E. P. and Trystram, D., "A new genetic convex clustering algorithm for parallel time minimization with large communication delays." in *Proceedings of the International Conference ParCo 2005*, ser. John von Neumann Institute for Computing Series, Joubert, G. R., Nagel, W. E., Peters, F. J., Plata, O. G., Tirado, P., and Zapata, E. L., Eds., vol. 33. Central Institute for Applied Mathematics, Jülich, Germany, 2005, pp. 709–716.

Sarkar, V., *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press Cambridge, MA, USA, 1989.

Sarkar, V. and Cann, D., "Posc—a partitioning and optimizing sisal compiler," in *ICS '90: Proceedings of the 4th international conference on Supercomputing.* New York, NY, USA: ACM, 1990, pp. 148–164.

Sarkar, V. and Hennessy, J., "Partitioning parallel programs for macro-dataflow," in *LFP '86 Proceedings of the 1986 ACM conference on LISP and functional programming.* Assoc. of Computing Machinery, 1986.

Smyk, A. and Tudruj, M., "Genetic optimization of parallel fdtd computations," in *ISPDC '08: Proceedings of the 2008 International Symposium on Parallel and Distributed Computing.* Washington, DC, USA: IEEE Computer Society, 2008, pp. 155–161.

Stephenson, M., Amarasinghe, S., Martin, M., and O'Reilly, U., "Meta Optimization: Improving Compiler Heuristics with Machine Learning," *ACM SIGPLAN Notices*, vol. 38, no. 5, p. 90, 2003.