A Verified Program for the Enumeration of All Maximal Independent Sets

A thesis presented to
the faculty of
the Russ College of Engineering and Technology of Ohio University

In partial fulfillment
of the requirements for the degree
Master of Science

Samuel A. Merten
December 2016
© 2016 Samuel A. Merten. All Rights Reserved.
This thesis titled
A Verified Program for the Enumeration of All Maximal Independent Sets

by
SAMUEL A. MERTEN

has been approved for
the School of Electrical Engineering and Computer Science
and the Russ College of Engineering and Technology by

David Juedes
Professor of Electrical Engineering and Computer Science

Dennis Irwin
Dean, Russ College of Engineering and Technology
ABSTRACT

MERTEN, SAMUEL A., M.S., December 2016, Computer Science

A Verified Program for the Enumeration of All Maximal Independent Sets (58 pp.)

Director of Thesis: David Juedes

Errors in programs and mathematical results are a common cause for wasted time and resources. While peer-review and software-engineering techniques are helpful in mitigating the occurrence of these errors, the human element at the center of these techniques is fallible, and for very large or complex results the application of these techniques is often infeasible. Towards this end, researchers have developed a number of tools, proof assistants, which aid the user in constructing machine-checkable formal proofs. Tools and theories verified using these tools carry a much stronger assurance of correctness than many alternatives.

Currently, a number of fundamental theorems and programs have been proven correct within these proof assistants, as well as a number of proofs which were previously deemed too large for review. However, despite these successes, and the ubiquity of graph algorithms in computer science, little work has been done towards generating verified implementations of graph algorithms.

This thesis describes my attempt to address this weakness through the construction of a verified implementation of Tsukiyama et al.’s algorithm for enumerating all maximal independent sets of a graph. The enumeration of all maximal independent sets has a number of immediate applications in domains such as computer vision, computational chemistry and information systems, as well as being an important subroutine in efficient algorithms for exact graph coloring. Such a verified algorithm could lay the foundation for a library of verified graph algorithms suitable for use in critical systems.
Dedicated to Julia, who has been fighting proofs for my attention since we first met.
ACKNOWLEDGMENTS

I would like to begin by thanking my advisor, Dr. David Juedes. Over the past few years, his support has been invaluable. Even during the busiest of times, Dr. Juedes has been available to talk over difficult problems and critique my work. He is responsible for introducing me to the discipline of formal verification; without his guidance, I doubt that I would have found my program over the past few years to be as personally fulfilling as it has been.

I would also like to acknowledge Dr. Gordon Stewart for his help during the past year and a half. His expertise and assistance has been essential in the completion of this thesis. Through my work with him, I have had the opportunity to explore topics and connections that previously had escaped my notice.

Additionally, I would like to thank my committee members, Dr. Chang Liu and Dr. Archil Gulisashvili, not only for their commitment to this thesis, but also for their pedagogy. The techniques and constructions I learned in their courses often provided me with the tools or inspiration I needed to overcome a difficult proof or construction.

My work over the past few years would have been impossible without the support of my family. As such, I would like to take this opportunity to express my deepest gratitude to my partner Julia, my mother Susan, and Travis for everything they have done. Their sacrifices are too many to enumerate (unlike maximal independent sets), but their companionship and care, were essential to my sanity.

A number of my peers have had a profound influence on me throughout my tenure in the master’s program: Mackenzie Crabtree, Patrick Gray, Charlie Murphy, and Joe Scott. Working and learning with them spurred me to be better, and their criticism kept me honest.
# Table of Contents

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
</tr>
<tr>
<td>Dedication</td>
</tr>
<tr>
<td>Acknowledgments</td>
</tr>
<tr>
<td>List of Tables</td>
</tr>
<tr>
<td>List of Figures</td>
</tr>
<tr>
<td>1 Introduction</td>
</tr>
<tr>
<td>2 Preliminaries</td>
</tr>
<tr>
<td>2.1 Graph Fundamentals</td>
</tr>
<tr>
<td>2.2 Maximal Independent Sets</td>
</tr>
<tr>
<td>2.3 Proof Assistants</td>
</tr>
<tr>
<td>2.3.1 Verification in Coq</td>
</tr>
<tr>
<td>2.3.2 Coq Fundamentals</td>
</tr>
<tr>
<td>3 Tsukiyama’s Algorithm</td>
</tr>
<tr>
<td>3.1 Tsukiyama et al.’s Algorithm</td>
</tr>
<tr>
<td>3.2 Correctness</td>
</tr>
<tr>
<td>3.2.1 A Small Example</td>
</tr>
<tr>
<td>4 Formalization of an Algorithm for the Enumeration of All Maximal Independent Sets</td>
</tr>
<tr>
<td>4.1 Sets and Lexicographic Orderings</td>
</tr>
<tr>
<td>4.1.1 Lexicographic Ordering</td>
</tr>
<tr>
<td>4.2 Graphs</td>
</tr>
<tr>
<td>4.2.1 Graph Predicates</td>
</tr>
<tr>
<td>4.3 A Greedy Algorithm for Generating Lexicographically First Maximal Independent Sets</td>
</tr>
<tr>
<td>4.3.1 Specifications</td>
</tr>
<tr>
<td>4.3.2 Proof Techniques</td>
</tr>
<tr>
<td>4.4 Enumeration of Maximal Independent Sets</td>
</tr>
<tr>
<td>4.4.1 Soundness</td>
</tr>
<tr>
<td>4.4.2 Completeness</td>
</tr>
<tr>
<td>4.4.3 Form Lemmas</td>
</tr>
<tr>
<td>4.4.4 Uniqueness</td>
</tr>
</tbody>
</table>
5 Conclusions ................................................................. 51
  5.1 Refinements .......................................................... 51
  5.2 Extensions ............................................................ 53

References ................................................................. 54
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>32</td>
</tr>
</tbody>
</table>

3.1 Application of EnumMIS with $G = C_4$
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Modus Ponens in Coq</td>
<td>23</td>
</tr>
<tr>
<td>2.2 Basic Arithmetic in Coq</td>
<td>24</td>
</tr>
<tr>
<td>3.1 ENUMMIS: Compute All Maximal Independent Sets in $G$</td>
<td>27</td>
</tr>
<tr>
<td>3.2 Cyclic Graph with 4 Vertices: $C_4$</td>
<td>31</td>
</tr>
<tr>
<td>4.1 Lexicographic Ordering</td>
<td>36</td>
</tr>
<tr>
<td>4.2 Section Based Graph Representation</td>
<td>37</td>
</tr>
<tr>
<td>4.3 Dependently Typed Graph Representation</td>
<td>38</td>
</tr>
<tr>
<td>4.4 Section Based Graph Predicates, Parameterized by $G = (V, E)$</td>
<td>39</td>
</tr>
<tr>
<td>4.5 Dependently Typed Graph Predicates</td>
<td>39</td>
</tr>
<tr>
<td>4.6 LiftGraph Definition</td>
<td>40</td>
</tr>
<tr>
<td>4.7 Induction Principle by Induced Subgraphs</td>
<td>41</td>
</tr>
<tr>
<td>4.8 MkMaximalIndSet Definition, Parameterized by $G = (V, E)$</td>
<td>42</td>
</tr>
<tr>
<td>4.9 MkMaximalIndSet Specifications</td>
<td>43</td>
</tr>
<tr>
<td>4.10 IndSetConstraint_gen Definition</td>
<td>43</td>
</tr>
<tr>
<td>4.11 vertexConnected_spec Definition</td>
<td>44</td>
</tr>
<tr>
<td>4.12 AllMIS: Compute All Maximal Independent Sets in $G$</td>
<td>46</td>
</tr>
<tr>
<td>4.13 Soundness of AllMIS</td>
<td>47</td>
</tr>
<tr>
<td>4.14 State of Soundness Proof Following Induction</td>
<td>47</td>
</tr>
<tr>
<td>4.15 Induction Lemma for Soundness</td>
<td>48</td>
</tr>
<tr>
<td>4.16 Completeness of AllMIS</td>
<td>48</td>
</tr>
<tr>
<td>4.17 Form Lemmas</td>
<td>49</td>
</tr>
<tr>
<td>4.18 Uniqueness of AllMIS and noDuplicates Specification</td>
<td>50</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

Mathematicians and computer scientists are often haunted by the specter of errors in their proofs and bugs in their code. Even subtle, small mistakes can have huge consequences. History provides ample evidence of this fact: a race condition in software for the Therac-25, a radiation therapy machine, resulted in the death of four and the injury of two others [LT93]; an overflow error in recycled code from the Ariane-4 rockets caused the first test flight of the Ariane-5 rockets to self-destruct, with an estimated cost of $500 million [Lio]; and Kempe’s incorrect proof of the four color theorem stood for 11 years before a critical flaw was recognized [Hea90] (a correct proof was finally presented almost a century later by Appel and Haken [AH77, AHK77]). Over the years, theorists and software engineers have developed a large repertoire of tools and techniques for minimizing the occurrence of errors in finished products, including the peer-review of published papers, rigorous testing methods and software development processes that promote security and correctness. While these methods are effective in catching mistakes they are heavily reliant on human oversight, and this poses two major problems. First, reviewers and test writers are fallible and may fail to catch an error, or overlook a critical case, as in the examples above. Secondly, some developments are too large or complex for efficient application of these methods. In the past century a number of mathematical proofs with this flavor have emerged, including Hale’s proof of the Kepler conjecture [Hal05], regarding configurations for optimal sphere packing, as well as Appel and Haken’s (correct) proof of the four-color theorem [AH77, AHK77]. Each of these proofs were hundreds of pages long and relied on auxiliary programs to explore numerous individual cases of their proofs, resulting in only tenuous acceptance by their reviewers. Given the size and complexity of many programs, software too is susceptible to this weakness and software testing is rarely exhaustive.

One alternative to these standard techniques, which has seen recent success, is formal verification of proofs and programs using proof assistants. These tools allow the user to
construct proofs, and state axioms, definitions, and propositions using a formal language familiar to standard mathematics. Each statement is checked by the assistant in order to ensure that it is well formed, and in the case of proofs, ensures that each step in the proof follows from a valid deduction rule. Furthermore, due to a result known as the Curry-Howard Isomorphism, which establishes an equivalence between proofs and programs, these tools are able to construct and reason about programs by reusing the same techniques and rules used to check the validity and well-formedness of propositions.

The process of encoding a theory in these systems takes additional effort; the obligations imposed by the proof assistant are much more thorough than those imposed by the standard audience of mathematicians and programmers; and the proof assistant allows for no omissions in its proofs. However, this extra effort conveys two major benefits. First, and most clearly, results verified by one of these proof assistants possess an increased degree of certainty. Rather than placing trust in the hands of potentially fallible reviewers or incomplete testing, one need only place trust in the correctness of the theorem prover and the underlying hardware. Barring hardware failure or an error in the proof assistant itself, the proofs and program specifications written within the system are guaranteed to be valid with respect to the axioms of the theorem assistant’s logic and those introduced by the user. Strengthening this assurance of correctness, the logical kernels of many theorem provers are intentionally small and easily checkable. In many cases, there is little question about the result’s validity, or the risk and cost of a program’s failure is not, by itself, worth the endeavor of formal verification. However, as a second, more widely applicable benefit of verification, the rigor imposed by the theorem prover acts as a crucible for the current computational and mathematical machinery. The act of verification results in new tools and techniques that provide both a deeper understanding of the problem at hand and the means to address more novel problems.
Given these benefits, it is no surprise that researchers have envisioned a future in which new developments and theories are accompanied by a formal proof. A theory is only as strong as its foundation, and building a verification attempt on top of an unverified base undermines the attempt. Just as programmers rely on APIs and libraries to avoid writing code from scratch and mathematicians build their theories on those of their predecessors, before it becomes commonplace for novel results to be accompanied by a formal proof, it is essential to develop libraries that encode fundamental results within proof assistants. Already, researchers have provided formalizations for a number of fundamental results in logic [O05], abstract algebra [GPWZ02] and real analysis [BLM15]. Simultaneously, at conferences such as Principles of Programming Languages (POPL), it has become increasingly common for developments to be accompanied by formal proofs of correctness [DPCGC15, AK16]. However, one domain which has yet to see significant development, despite its wide range of application and maturity, is verified implementations of graph algorithms [BRA10].

In this thesis, I outline my attempt to address this dearth of development through the implementation and verification of an efficient algorithm for enumerating all maximal independent sets of a graph, within the proof assistant Coq [BC13]. This work consists of four notable results:

1. A Coq library of graph theoretic concepts suitable for both theorem proving and programming. This library includes two implementations of graphs, functions relating these implementations, fundamental predicates over graphs, and most notably, an induction principle via induced subgraphs.

2. The construction of a lexicographic ordering over sets of vertices.

3. A program for finding the first (according to the aforementioned lexicographic ordering) maximal independent set of a graph containing a given independent set.
4. An implementation, along with proofs of correctness, of an algorithm (due to Tsukiyama et al.) for finding all maximal independent sets, constructed using the above results.

Sections of this thesis are based on a manuscript entitled “Certified Program for Computing All Maximal Independent Sets”, by Samuel Merten, Gordon Stewart, and David Juedes [MSJ16], pending publication.

At this point, it is worth delimiting the scope of these contributions by discussing the limitations of their verification. The theories and programs enumerated above are guaranteed to be correct under the assumption that Coq itself behaves correctly, and this is no small assumption. The correct behavior of Coq is not only a product of the code used to construct the proof assistant’s kernel and the theory behind Coq, it is also reliant on (at least) the OCaml compiler used to compile this code, the OCaml runtime system which executes the proof assistant, the C compiler used to build the OCaml runtime system, the operating system responsible for managing these components, and the hardware on these programs are executed. An error in any of these systems could compromise the correct behavior of Coq and invalidate these results. The assumption that all of these components function correctly, or at least function in a way which does not compromise the performance of Coq, is an assumption that underlies most other unverified programs. However, Coq still has a number of advantages. First, the Coq kernel is kept intentionally small to mitigate the possibility of implementation errors, and it uses only the most basic features of OCaml to minimize the chance that some fault in the OCaml system introduces some inconsistency into Coq. Second, as a tool developed for formal verification, Coq and the theory underneath it have been under intense scrutiny throughout its almost 30 year lifetime. At this point, the occurrence of bugs in the kernel is rare, and their effect on developments which do not intentionally exploit these errors is minimal. Trusting the correct behavior of a small, singular, well-tested tool such as Coq is a significantly lighter obligation than trusting
the behavior of more complicated, less tested programs. In this sense, the trusted computing base of the program verified in this thesis is significantly smaller than that of an unverified implementation.

Beyond these caveats, the existence of such a verified program is useful. The enumeration of maximal independent sets emerges as a natural solution in many practical applications, including networking, computer vision and computational chemistry [MW05, JNŠ01, SN95]. Outside of these immediate applications, the enumeration of all maximal independent sets is an important subroutine in efficient exact algorithms for graph coloring and other related problems [Law76]. With this wide range of applications, a verified algorithm for enumerating maximal independent sets has immediate use, but it is my hope that the techniques applied in the process of verification could serve as the foundation for a library of verified graph algorithms and results.
2 PRELIMINARIES

This chapter is intended to provide the reader with the fundamentals necessary to understand the contributions detailed in Chapter 3. This entails a review of necessary concepts from graph theory, an introduction to theorem proving in Coq, a review of the literature surrounding maximal independent sets and verified graph algorithms, as well as a detailed look at Tuskiyama et al.’s algorithm for enumerating all maximal independent sets of a graph.

2.1 Graph Fundamentals

Following are definitions of fundamental notions regarding graphs and their properties which are relevant to this thesis. Such a review is helpful, given the formalism inherent to theorem proving and software verification within a proof assistant.

1. A graph $G = (V, E)$ is an ordered pair of two sets, such that $E \subseteq V \times V$. The term vertex denotes a member of $V$, while edge denotes a member of $E$.

2. Two vertices, $v_1, v_2$, are connected if $(v_1, v_2) \in E$.

3. A graph in which no vertex is connected to itself is simple.

4. An undirected graph is one in which all edges are symmetric, that is $(v_1, v_2) \in E$ implies that $(v_1, v_2) \in E$.

5. For a graph $G$, the neighborhood of a vertex, $v$, denoted by $\Gamma_G(v)$ (or $\Gamma(v)$ when the parent graph is unambiguous), is the set of all vertices connected to $v$. Equivalently, $\Gamma(v) = \{v'|(v, v') \in E\}$.

6. A simple, undirected graph in which each vertex is connected to all other vertices is complete. We denote complete graphs with $v$ vertices by $K_v$. 
7. The subgraph induced by a subset of its vertices, $V'$, is the graph $(V', E')$, where $E'$ consists of all edges connecting pairs of vertices from $V'$ in the original graph.

8. An independent set of a graph is a subset of vertices such that there are no edges between any pairs of vertices.

9. A maximal independent set (MIS) of a graph is an independent set in which the addition of any new vertex to the set violates the conditions for independence.

10. A clique is a subset of vertices in which every vertex in the clique is connected to every other vertex by an edge.

11. A maximal clique is a clique in which it is impossible to introduce a new vertex to the set which results in a clique.

12. The graph complement of a graph is a new graph with the same vertices, but whose edges consist of the complement of the initial set of edges, with respect to $V \times V$.

Finally, it is worth noting that every independent set of a graph is a clique in the graph complement, and vice-versa. The same holds for maximal independent sets and maximal cliques. Since generating graph complements is trivial, the problem of finding (maximal) independent sets is considered equivalent to that of finding (maximal) cliques.

2.2 Maximal Independent Sets

Fundamental results regarding the enumeration of maximal independent sets (equivalently cliques) emerged during the later half of the twentieth century, with publications by Moon and Moser [MM65], as well as Bron and Kerbosch [BK73]. In their paper, “On Cliques in Graphs”, Moon and Moser derived an upper bound of $O(3^n)$ on the number of maximal cliques in a graph with $n$ vertices. Their proof reaches this bound by providing a MIS-preserving transformation of a graph into copies of $K_3$ and potentially one
copy of $k_2$ or $k_1$ and then deriving a bound on graphs of this form. Following this development, Bron and Kerbosch presented one of the first efficient algorithms for enumerating the maximal cliques of a graph. Their algorithm works by greedily generating maximal cliques for each vertex, while avoiding redundant computation by maintaining a list of vertices already used to generate maximal cliques. For a graph with $n$ vertices, their algorithm runs in $O(3^n)$, an optimal bound with respect to the aforementioned result by Moon and Moser.

Over the years, these two seminal results have seen small refinements. Erdős managed to extract slightly tighter bounds from Moon and Moser’s proof [Erd66], while Croitoru [Cro79] used a similar argument to bound the number of maximal independent sets in terms of the size of the largest independent set of the graph. In the 2000s, both Eppstein [Epp01] and Nielsen [Nie02] produced tight bounds on the number of maximal independent sets. Eppstein’s bound only considers maximal independent sets smaller than some parameter $k$, while Nielsen’s bounds hold in general.

With respect to enumerative algorithms for maximal independent sets, multiple variations of the Bron-Kerbosch algorithm have emerged in the time since its publication. These include the family of algorithms presented by Johnston [Joh76], as well as those presented by Johnson, Papadimitriou, and Yannakakis [JPY88]. In general these variations provide better performance than the original Bron-Kerbosch algorithm, however, they explore essentially the same search space and possess the same asymptotic time complexity.

Not all efficient algorithms for the enumeration of maximal independent sets are modifications of the Bron-Kerbosch algorithm. In 1977 Tsukiyama et al. [TIAS77] presented a new enumerative algorithm with a running time that, unlike the Bron-Kerbosch algorithm, is output sensitive: its asymptotic runtime is bounded by the total number of maximal independent sets in the graph. It achieves this by building maximal independent sets of a graph from the maximal independent sets of its induced subgraphs. Tsukiyama et
al.’s algorithm, made an ideal candidate for investigation because of this structural recursion. This property lends itself well to the logic of Coq, and for this reason the algorithm by Tsukiyama et al. was chosen to be the focus of this verification attempt.

Beyond new methods of enumeration, researchers have identified a number of interesting properties of and applications for the enumeration of all maximal independent sets. One of the earliest results of this kind was Lawler’s dynamic programming algorithm for graph coloring. That algorithm uses the enumeration of all maximal independent sets as a subroutine to construct an efficient algorithm for exact graph colorings [Law76]. Johnson, Yannakakis and Papadimitriou developed an algorithm which enumerates all maximal independent sets in lexicographic order with only polynomial time delay between generating new maximal independent sets, and showed that no such algorithm exists which enumerates all maximal independent sets in reverse lexicographic order, unless \( P = NP \) [JPY88].

2.3 Proof Assistants

This chapter concludes by providing a brief overview of proof assistants, a review of notable results and a brief tutorial on the use of Coq, the theorem prover used in our development.

As their name suggests, proof assistants are programs that aid their user in the construction of formal proofs. They will not automatically derive proofs for the user (excluding the most trivial of cases), and they are not of particular use in developing the intuition that goes into the derivation of novel proofs. Instead, proof assistants aid in the formal aspect of formal proofs. A formal proof is not the semi-formalized arguments based in natural language that pervade math textbooks and journal articles. Rather a formal proof is a series of statements written in a formal language that proceed only according to the rules dictated by a particular logic.
Currently there a number of proof assistants seeing wide use, each with its own notation and based on different logical systems. Among those in current use are Coq, F* [SCF+11], Isabelle [NPW02], HOL [GM93], Mizar [miz] and PVS [ORS92]. For this development Coq was chosen for its support of dependent types and an intuitive method of code extraction. Dependent types are types whose construction depends on the value of elements used in its construction. Common examples of dependent types include lists of fixed length and ordered pairs where the second element is strictly greater than the first. In this project, dependent types made it possible to incorporate the propositional facts that distinguish simple undirected graphs into the constructors for these objects. Code extraction allows the user to transform functions and definitions written in the theorem prover into functions in other, more efficient languages while preserving the behavior of the initial function. With the aim of this development being the verification of an algorithm, being able to extract our Coq implementation into working code was an attractive attribute.

## 2.3.1 Verification in Coq

As mentioned in the introduction, a number of theories and algorithms have been successfully verified in Coq, including the four color theorem, the Kepler conjecture and the Feit-Thompson theorem [Gon08, HAB+15, GAA+13]. However, beyond verification of mathematical proofs, Coq has also been successfully applied in generating verified software and algorithms. The most notable of these contributions is perhaps CompCert [Ler09], a verified compiler for the majority of the C language. Other examples of verified algorithms include implementations of Huffman encoding/decoding [The04], the perceptron algorithm from machine learning [MGS16] and the IRC algorithm for register allocation [BRA10].

Two of these aforementioned developments are related to graph theory (the four color theorem is a statement about planar graphs and the IRC algorithm reduces the problem of register allocation to graph coloring), and there have been a number of other developments
focusing on the formalization of graph theory and graph algorithms. A number of libraries containing formalizations of basic graph theoretic concepts such as paths, cycles, induced subgraphs, vertex degree, and special graph types have been developed for a number of systems including HOL [Cho94], Mizar [Hry90] and Coq [Dup01]. Unfortunately, it is worth noting that the aforementioned Coq library is ill-suited towards the verification of graph algorithms, as its definitions cannot be extracted [BRA10, p.18]. Beyond these basic libraries, researchers have generated code for classic algorithms in graph theory such as Dijkstra’s (Mizar and ACL2) [MZ05, Che03], and Prim’s (Mizar and B) [LR05, ACM03]. While Mizar has been the source of a number of these results, it is worth noting that Mizar does not support code extraction, and these results cannot be extended to produce verified programs outside of Mizar.

Outside of these fundamental results, however, little work has been done on verifying more complex graph algorithms. In particular, to my knowledge, no work has been done on the verification of non-polynomial time or NP-complete graph algorithms (the coloring that occurs in the IRC algorithm is not necessarily minimal and runs in polynomial time). And while the verification of the four color theorem was a sizable achievement, no other theoretic results of that magnitude have undergone verification.

2.3.2 Coq Fundamentals

The language underlying Coq is a system known as the Calculus of Inductive Constructions (CIC). The CIC is a higher-order, typed lambda calculus supporting dependent types. The lambda calculus was initially developed by Alonzo Church [Chu36] as a model of computation, and from this perspective CIC can be viewed as a programming language. However, the Curry-Howard Correspondence ensures that it is possible to encode the rules of propositional intuitionistic logic within CIC, and from this perspective, CIC can be seen as a logical framework for the construction of proofs and propositions. The scope of
this thesis does not permit a full presentation of either the CIC, or Coq’s interpretation of this system. Instead the remainder of this section attempts to provide the reader with a working understanding of these components. Those interested in more detailed presentations of these subjects are directed to the Coq reference manual [dt16] as well as Benjamin Pierce’s “Software Foundations” [PCG+10] and Adam Chilpala’s “Certified Programming with Dependent Types” [Chl11].

Using the CIC as a basis, Coq achieves its role as a theorem prover through two other languages: Gallina, a specification language for constructing and relating terms in CIC; and Ltac, a tactic language for facilitating the manipulation of CIC terms. A Coq development takes the form of a proof script written in Gallina with Ltac and CIC subterms. As a simple example consider the Gallina statement shown in Figure 2.1, which shows the validity of modus ponens within Coq. This statement consists of two major components. The first is the declaration indicated by the Gallina keyword Lemma, which states that the name MP will be bound to an element with the type \( \forall p q \). . . . The second component of this Gallina term is the construction of this element using commands from the Ltac language.

Figure 2.1 illustrates how this process unfolds. Within this figure, the text to the left indicates the commands provided to the theorem prover, while the text to the right shows the state of the proof following these declarations and applications of tactics. The declaration of the object (Lemma . . . ), causes Coq to enter theorem proving mode and begin displaying a proof state. In this mode, Coq displays a list of current goals below the dashed line and a list of facts and variables in context above the line. Since the declaration does not introduce any assumptions into the context, following its declaration Coq presents an empty context above the line followed by the content of the lemma as a single goal to be proven. The intros tactic performs the role of universal generalization, introducing specific, arbitrary elements corresponding to the propositions \( p, q \) and the facts that \( p \) is true and \( p \rightarrow q \) (if \( p \) then \( q \)). Following the application of the tactic, these terms appear in the context and can be
manipulated at later steps of the proof. The use of the \textit{apply} tactic modifies the goal via
backwards chaining: if $p$ implies $q$ then in order to prove $q$, it suffices to prove $p$. From this
use of the \textit{apply} tactic, the final step of the proof involves showing that $p$ is true, but $p$’s
truth is assumed as part of the proof and is named in the proof’s context as $H$. The \texttt{exact}
tactic encapsulates this reasoning, allowing a user to discharge a goal, provided an
equivalent term already exists in context. At this point, the proof is complete and the
command \texttt{Qed} attempts to close the proof-editing mode (if the proof were not complete,
Coq would not allow the proof-script to progress).
Lemma MP : \( \forall p \ q : \text{Prop}, \ p \rightarrow (p \rightarrow q) \rightarrow q. \)
Proof.

1 subgoal

\[ \forall p \ q : \text{Prop}, \ p \rightarrow (p \rightarrow q) \rightarrow q \]

>> intros p q H0 H1.

1 subgoal

p, q : Prop
H0 : p
H1 : p \rightarrow q

q

>> apply H1.

1 subgoal

p, q : Prop
H0 : p
H1 : p \rightarrow q

p

>> exact H0.

No more subgoals.

>> Qed.

MP is defined.

Figure 2.1: Modus Ponens in Coq

The process and underlying mechanisms of the declaration, binding and type checking that occur in the above example should be familiar to individuals with prior programming experience. The only strange thing is that within Coq, an element with type \( \forall p \ q : \text{Prop}, \ p \rightarrow (p \rightarrow q) \rightarrow q \) corresponds to a proof of that statement. In this way, Gallina statements like the one above are used to bind and label propositions to their proofs.
However, Gallina permits more than the declaration of propositions. Using various
keywords, users can introduce new types and axioms, provide definitions for functions and
programs, as well as construct complex tactics. Figure 2.2 shows a proof script that
illustrates a small subset of these features. For those unfamiliar with Coq, it may be helpful
to work through this script and see how the proofs unfold.

**Inductive** nat : Set ≜
| O : nat
| S : nat → nat.

**Fixpoint** add (n m : nat) : nat ≜
match m with
| O ⇒ n
| S m' ⇒ S (add n m')
end.

**Theorem** addOr : ∀n, add n O = n.
Proof.
  intros n.
  simpl.
  reflexivity.
Qed.

**Theorem** addOl : ∀n, add O n = n.
Proof.
  intros n.
  induction n.
  simpl.
  reflexivity.
  simpl.
  rewrite IHn.
  reflexivity.
Qed.

Figure 2.2: Basic Arithmetic in Coq
The **Inductive** keyword allows the user to define inductive types, types generated solely with respect to their introduction rules. This means that any element of this type has a canonical construction and provides the basis for proofs by structural induction and case analysis. In the example above, the inductive definition of `nat` provides two constructors for the type – `O` which is primitive and `S` which generates a new `nat` from an existing one, corresponding to the Peano construction of natural numbers [Pea79].

In a similar manner, recursive functions in Coq can be constructed using the **Fixpoint** keyword. However, one of the quirks of Coq is that in order to ensure the consistency of the logic, all functions are required to terminate. Coq ensures that functions declared using **Fixpoint** terminate by requiring the recursive calls of the fixpoint be called on structurally smaller terms. The example above contains a recursive definition for addition over elements of type `nat`. The **match** statement in the definition of `add` performs case analysis over `m`, according to its constructors. In the case that `m` was `O` the function would simply return `n`. However if `m` was built by applying the constructor `S` to another `nat` `m'`, the function makes a recursive call to `add` using, the necessarily smaller, value of `m'` as an argument.

With these constructions, it is possible to show that `O` acts as an additive identity, both by addition on the left and the right. The two lemmas in the above example prove these facts. These proofs are relatively simple, and use only three tactics that were not used in Figure 2.1. The tactic **simpl** applies a simplification procedure to the goal. For example, its application in `addOr` transforms the state of the goal from `add n O = O` to `n = n` by unfolding and evaluating the definition of `add`. The **reflexivity** tactic solves a goal involving equality when both sides of the equality are syntactically equal. For example, if the goal has the form `n = n`, like the state resulting from the application of **simpl**, **reflexivity** will solve the goal. Finally, the **induction** tactic, when applied to an object with an inductive type will generate goals for each of the constructors of the type. In the case of the `nat` type described above, these goals take the standard form for induction over natural numbers. Applying
induction in the proof addOl above transforms the goal from add 0 n = n into two subgoals:
add 0 0 = 0 and add 0 (S n) = S n, and introduces an induction hypothesis into the context
of the second subgoal add 0 n = n. In conjunction with the tactics discussed earlier, it is a
simple process to conclude these proofs.

While this tutorial is necessarily incomplete, it does highlight the basic properties of
Coq. In the process of verifying Tsukiyama et al.’s algorithm, at times it was necessary to
use constructions and tactics more complicated than those illustrated in this section,
however, the majority of the development can be understood in the context of the techniques
shown here.
3 TSUKIYAMA’S ALGORITHM

This section provides the reader with a detailed analysis of Tsukiyama et al.’s algorithm. The analysis consists of three major parts: the presentation of the algorithm, a proof of the algorithm’s correctness and a small example of the algorithm’s execution. Having an understanding of the algorithm in its standard presentation helps to navigate the more technical presentation that the proof assistant mandates and motivates some of the constructions that emerged in the process of formalization and verification.

3.1 Tsukiyama et al.’s Algorithm

The algorithm shown in Figure 3.1 calculates all of the maximal independent sets. This algorithm is due to Tsukiyama et al. The version presented here is based closely on the description provided by Johnson, Papadimitriou and Yannakakis [JPY88].

```plaintext
1: REQUIRE n = number of vertices of G.
2: if n = 0 then
3:   return the list containing ∅
4: else
5:   ℓ = ENUMMIS(G_{n-1}).
6:   ℓ₁ = empty list.
7: for all S ∈ ℓ do
8:   if S ∩ Γ(n - 1) = ∅ then
9:     add S ∪ {n - 1} to ℓ₁.
10: else
11:    S’ = S − Γ(n - 1) ∪ {n - 1}.
12:    S” = LFMIS(G_{n-1}, S − Γ(n - 1)).
13:    if (S’ is a maximal independent set in G) and (S = S”) then
14:      add S and S’ to ℓ₁.
15: else
16:    add S to ℓ₁
17: return ℓ₁

Figure 3.1: ENUMMIS : Compute All Maximal Independent Sets in G
```
At a high level, this algorithm works by constructing the MISs in a graph $G_n$ (with vertices labeled $\{0, \ldots, n-1\}$) from the MISs in $G_{n-1}$. In the case of the empty graph, the algorithm builds the set of MISs explicitly (lines 2 & 4). For each MIS $S$ in $G_{n-1}$ the algorithm builds new MISs in $G$ by considering three possible cases:

1. $S \cup \{n-1\}$ is an independent set in $G_n$ (lines 8 & 9). In this case, the algorithm adds $S \cup \{n-1\}$ to the list of MISs of $G_n$.

2. The condition for the first case cannot be satisfied, and it is the case that both $S \cup \{n-1\} - \Gamma(n-1)$ is an MIS in $G$ and $S$ is the lexicographically first maximal independent set in $G_{n-1}$ containing $S - \Gamma(n-1)$ (lines 13 & 14). In this case, this algorithm adds both $S$ and $S \cup \{n-1\} - \Gamma(n-1)$ to the list of MISs of $G_n$.

3. If $S$ fails to satisfy any of the two above conditions, the algorithm simply adds $S$ to the list of MISs of $G_n$ (lines 15 & 16).

### 3.2 Correctness

Here, we present an informal proof of correctness for Tsukiyama’s algorithm, that for any graph $G$, the output of $\text{MIS}(G)$ indeed enumerates all MISs of $G$.

In order to prove the correctness of the $\text{EnumMIS}$ algorithm, it suffices to show that every set in the output of $\text{EnumMIS}$ is an MIS and that every MIS is an element of the output of $\text{EnumMIS}$.

**Lemma 3.2.1.** For all graphs $G$ and sets of vertices $X$, if $X \in \text{EnumMIS}(G)$ then $X$ is an MIS of $G$.

The proof proceeds by induction via the induced subgraphs of $G$. In the base case, if $G$ is the empty graph, then the empty set is clearly an MIS of the graph. In the inductive step, assuming that $\text{EnumMIS}(G_{n-1})$ outputs only MISs of $G_{n-1}$, it can be shown that
EnumMIS($G_n$) produces only MISs of $G_n$ by considering the new sets generated by the three cases outlined above for an arbitrary element $S$ of the output of EnumMIS($G_{n-1}$).

1. Here, $S \cup \{n - 1\}$ is an independent set in $G_n$ and the algorithm adds $S \cup \{n - 1\}$ to the set of MISs in $G_n$. Since $S$ is a maximal independent set in $G_{n-1}$ the addition of any new vertices in the range $[0, \ldots, (n - 2)]$ to $S$ results in a set that is no longer independent in either $G_n$ or $G_{n-1}$. Thus, $S \cup \{n - 1\}$ must be a maximal independent set since no additional vertices can be added without destroying independence.

2. If the new sets are produced under this case, then $S \cup \{n - 1\} - \Gamma(n - 1)$ is an MIS in $G_n$, and $S \cup \{n - 1\}$ is not an independent set in $G_n$. Two sets are added to the list of MIS in $G_n$: $S$ and $S \cup \{n - 1\} - \Gamma(n - 1)$. $S$ must be an MIS of $G_n$ since $S$ was maximal in $G_{n-1}$ and $S \cup \{n - 1\}$ is not independent. $S \cup \{n - 1\} - \Gamma(n - 1)$ must be maximal by assumption.

3. This case only produces one maximal independent set $S$, and only if $S$ fails to satisfy any of the prior cases. Notably, this means that $S \cup \{n - 1\}$ is not an independent set in $G_n$, and $S$ is an MIS in $G_n$ by the same reasoning as in Case 2.

Thus, any element in the output of EnumMIS($G_{n-1}$) when processed in the execution of EnumMIS($G_n$) produces only MISs in $G_n$. Since only elements in the output of EnumMIS($G_{n-1}$) are processed, the output of EnumMIS($G_n$) consists of only MISs in $G_n$. Since this property holds for both the base case and the inductive case, the output of EnumMIS consists of only MISs.

**Lemma 3.2.2.** For all graphs $G$ and sets of vertices $X$, if $X$ is an MIS of $G$ then $X \in EnumMIS(G)$.

Similar to the prior lemma, this proof proceeds by induction on the induced subgraphs of $G$. In the case of the empty graph, the algorithm produces all maximal independent sets
explicitly. For all other cases, suppose that $S$ is a maximal independent set in $G_n$. To show that EnumMIS produces all maximal independent sets, it suffices to show that there exists some $S'$ which is an MIS in $G_{n-1}$ such that when evaluated according to one of the three cases, $S'$ results in the production of $S$. For some such $S$, either $n - 1 \in S$, or $n - 1 \notin S$. If $n - 1 \notin S$ then let $S' = S$. $S$ is clearly an MIS in $G_{n-1}$ and $S \cup \{n - 1\}$ is not an MIS in $G_n$, since $S$ is an MIS in $G$. As $S \cup \{n - 1\}$ is not an MIS in $G_n$ then it will be processed by either Case 2 or Case 3, both of which result in the inclusion of $S$ in the output of EnumMIS. If $n - 1 \in S$ then let $S' = LFMIS(S \setminus \{n - 1\})$, the lexicographically first maximal independent set in $G_{n-1}$ which contains $S \setminus \{n - 1\}$. It may be the case that $S'$ is $S \setminus \{n - 1\}$. In this case $S'$, will be evaluated by Case 1 and produce $S$ by readding $n - 1$ to $S'$. In any other case, there are elements added to $(S \setminus \{n - 1\}$ in generating $S'$. However, each of these new elements are connected to $n - 1$ in $G_n$. Thus, $S' \cup \{n - 1\} - \Gamma(n - 1) = S$ and by construction, $S'$ is the LFMIS in $G_{n-1}$ which contains $S \setminus \{n - 1\}$ as a subset. Thus, $S'$ is processed by the second case and produces $S$. Thus, for any MIS in $G_n$ there exists an MIS in $G_{n-1}$ which results in its production via the EnumMIS algorithm, and by induction EnumMIS produces every maximal independent set in a graph.

3.2.1 A Small Example

In order to illustrate how this algorithm generates all MISs, consider its operation on the cyclic graph of four vertices, $C_4$, and its induced subgraphs (shown in Figure 3.2). Unfolding the recursion, of EnumMIS, the algorithm first generates the MISs of the empty graph ($G_0$) explicitly and derives the remainder according to the three cases outlined above.
Since adding 0 to $\emptyset$ produces an independent set in $G_1$, $\text{EnumMIS}(G_1)$ produces only one MIS $\{0\}$ from the MIS in $G_0$, by the application of the first rule to $\{0\}$. In $\text{EnumMIS}(G_2)$, the MIS from $G_1$, $\{0\}$, generates two MISs ($\{0\}$ and $\{1\}$) through the second production rule, since $\{0, 1\}$ is not an MIS in $G_2$, $\{0\} \cup \{1\} - \Gamma(1) = \{1\}$ is an MIS in $G_2$ and $\{1\}$ is (trivially) the LFMIS in $G_1$ containing $\{0\} - \Gamma(1) = \emptyset$. $\text{EnumMIS}(G_3)$ produces only two maximal independent sets $\{0\}$ and $\{1, 2\}$. $\{1, 2\}$ is produced by the application of the first production rule to the MIS of $G_2$ $\{1\}$, since $\{1\} \cup \{2\}$ is an MIS in $G_3$ and $\{0\}$ is produced by the third rule since the conditions for the other rules fail. Finally, the two MISs for $C_4$ are generated from the MISs in $G_3$ by applying the third production rule to $\{1, 2\}$, resulting in the occurrence of $\{1, 2\}$ in the output, and the application of the first rule to $\{0\}$, resulting in the occurrence of $\{0, 3\}$. Table 3.1 provides a simple overview of the process described above. The right most entry of each row shows how each MIS in the preceding subgraph is used to generate new MISs for the current graph.
Table 3.1: Application of EnumMIS with \( G = C_4 \)

<table>
<thead>
<tr>
<th>Graph Label</th>
<th>Graph</th>
<th>Maximal Independent Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G_0 )</td>
<td></td>
<td>( \ell := {0} )</td>
</tr>
</tbody>
</table>
| \( G_1 \)  | ![Graph 1](image1.png) | \( S_0 = \emptyset \xrightarrow{\text{Case 1}} \{0\} \)
            |                   | \( \ell := \{0\} \)    |
| \( G_2 \)  | ![Graph 2](image2.png) | \( S_0 = \{0\} \xrightarrow{\text{Case 2}} \{0\}, \{1\} \)
            |                   | \( \ell := \{0, \{1\}\} \) |
| \( G_3 \)  | ![Graph 3](image3.png) | \( S_0 = \{0\} \xrightarrow{\text{Case 3}} \{0\} \)
            |                   | \( S_1 = \{1\} \xrightarrow{\text{Case 1}} \{1, 2\} \)
            |                   | \( \ell := \{0, \{1, 2\}\} \) |
| \( G \)    | ![Graph G](imageG.png) | \( S_0 = \{0\} \xrightarrow{\text{Case 1}} \{0, 3\} \)
            |                   | \( S_1 = \{1, 2\} \xrightarrow{\text{Case 3}} \{1, 2\} \)
            |                   | \( \ell := \{0, 3, \{1, 2\}\} \) |
4 **Formalization of an Algorithm for the Enumeration of All Maximal Independent Sets**

This chapter provides a description of the process of formalizing Tsukiyama et al.’s algorithm within Coq. There are four major components to this contribution:

1. The construction of a type capable of acting as sets of vertices, as well as the definition of a lexicographic ordering over this type.

2. The creation of a library of graph theory that contains two implementations of graphs; definitions for a number of fundamental graph predicates and relations; an induction principle for induced subgraphs; and methods for relating the two implementations.

3. The construction of an algorithm which produces the lexicographically first (according to the ordering above) maximal independent set of a graph which contains a given independent set as input.

4. An implementation and verification of Tsukiyama et al.’s algorithm for the enumeration of all maximal independent sets.

4.1 **Sets and Lexicographic Orderings**

In the type theory underlying Coq, sets are not elementary objects. This poses a problem, as the standard definition of graphs, along with Tsukiyama et al.’s algorithm are defined in terms of sets. As such, the first task of this development was the identification of a type which could act as a substitute for this standard mathematical object. Ultimately, the lists from Coq’s standard library were selected for this purpose. While not optimal from a performance perspective, the lists included in the Coq standard library are simple to reason about and require minimal modification to act as sets.

Coq’s definition of lists follows the standard construction used in functional programming languages. For an arbitrary type $A$, there is one elementary list of that type,
the empty list, indicated by nil. All other lists are formed by the constructor cons which forms a new list by appending an element of type A to the front of an existing list. In order to view lists as sets, one need only identify nil with the empty set, and cons with the addition of a single element to a set. Under this interpretation, many elementary set operations have simple representations as existent list operations. As examples, list membership is equivalent to set membership and list concatenation corresponds to set union.

Beyond the aforementioned computational inefficiency of this structure, there are two weaknesses in this choice of representation. First, while a set in general might be infinite, a list following this representation is always finite. However, for the purpose of this development, as well as many graph theoretic results and algorithms, this is not significant, as they concern themselves only with finite graphs. Additionally, with respect to programming in Coq, the finiteness of lists is a significant boon, as the termination of programs that recur over lists can often be automatically derived. The second, and more impactful difference is that with this implementation the representation of a set is not unique: the set \{a, b\} can be represented with both the list cons a (cons b nil) and the list cons b (cons a (cons a nil)). Coq’s definition of equality is strict, two terms are equivalent iff one can be rewritten to be syntactically identical to the other. However, since the nil and cons constructors for lists are primitive, no such rewriting exists in cases like the example above, and they are provably not equal. In order to work around this, the development makes use of an extensional equality over lists:

**Definition** \( \text{list
deq} \ X \ Y \triangleq \forall x, \ln x X \leftrightarrow \ln x Y. \)

An additional consideration is that arbitrary types in Coq are not equipped with decidable equality. For many of the predicates over lists, a lack of decidable equality over the type implies that the predicate is also undecidable. However, such undecidable predicates are unsuitable for the construction of programs. For this reason, along with others related to the construction of graphs, new predicates and functions for these lists-as-sets are
defined not for lists over an arbitrary type, but for lists of natural numbers (a type equipped with decidable equality and ultimately chosen to represent the vertices of graphs).

4.1.1 Lexicographic Ordering

Using this implementation of lists as sets, the next step in the development was the definition of a lexicographic ordering over sets of natural numbers (equivalently sets of vertices). The lexicographic order (on subsets of a well-ordered set $V$) is defined as follows. Given two subset $X, Y \subseteq V$, such that $X \neq Y$, $X < Y$ according to the lexicographic ordering if and only if the minimum element of $X - Y$ is less than the minimum element of $Y - X$ or $Y - X = \emptyset$.

The construction used in this formalization is shown in Figure 4.1. Here, $\text{delta_min } X \ Y$ computes the least element (using the function $\text{LeastElem}$) in $X$, but not $Y$, and returns an $\text{option nat}$ in order to handle the case in which there are no elements in $X$ not shared by $Y$. $\text{list_order}$ is an inductively defined type, with constructors which enumerate the result of $\text{dec_order}$. Finally, $\text{dec_order} \ X \ Y$ implements a procedure for determining the lexicographic ordering of $X$ and $Y$. The theorems $\text{dec_order_dual_spec}$ and $\text{dec_order_eq_spec}$ are two important results about this lexicographic order which are used later in proving the specifications of Tsukiyama et al.’s algorithm. The lemma $\text{dec_order_dual_spec}$ proves that $\text{lt_list}$ and $\text{gt_list}$ are duals, while $\text{dec_order_eq_spec}$ shows that two sets are equal iff the result of their comparison (via $\text{dec_order}$) is $\text{eq_list}$. Given that the comparison function is total, these lemmas ensure that there is always a least (lexicographically first) set of vertices for any collection of sets of vertices.
Fixpoint LeastElem (X : list nat) : option nat ≜
  match X with
  | nil ⇒ None
  | cons x X’ ⇒
    match LeastElem X’ with
    | None ⇒ Some x
    | Some y ⇒ if (lt_dec x y) then Some x else Some y
    end
  end.

Definition delta_min (X Y : list nat) : option nat ≜
  LeastElem (Subtract_list X Y).

Inductive list_order ≜
  | lt_list : list_order
  | eq_list : list_order
  | gt_list : list_order.

Fixpoint dec_order (X Y : list nat) : list_order ≜
  match delta_min X Y with
  | None ⇒
    match delta_min Y X with
    | None ⇒ eq_list
    | Some y ⇒ gt_list
    end
  | Some x ⇒
    match delta_min Y X with
    | None ⇒ lt_list
    | Some y ⇒
      if lt_dec x y
      then lt_list
      else gt_list
    end
  end.

Theorem dec_order_dual_spec : \forall X Y : list nat,
  dec_order X Y = lt_list ↔ dec_order Y X = gt_list.

Theorem dec_order_eq_spec : \forall X Y : list nat,
  dec_order X Y = eq_list ↔ list_eq X Y.

Figure 4.1: Lexicographic Ordering
4.2 Graphs

We use two variants of graphs. The first and simplest variation, shown in Figure 4.2 provides a minimal interpretation of graphs and is suited for reasoning about singular graphs. This representation makes use of Coq’s sectioning mechanism to allow the user to reason about a set of natural numbers (those natural numbers less than \(V\)) corresponding to the vertices in a graph, and a set of edges denoted by \(E\), which together with the axioms \text{NoSelfEdges}, \text{Edges}_{lt}V\ and \text{EdgesBidirectional}\ suffice to characterize a simple, undirected graph. The notion of subsets of the set of vertices is captured by the predicate \text{ValidSet}. Within the section IS the user can provide definitions and derive theorems about these terms as if they were concrete objects. However, outside of the section any constructions dependent on these axioms or the values of \(E\) or \(V\) are generalized. For example, within the section, the definition \text{ValidSet} has type \(\text{list nat} \rightarrow \text{Prop}\), while outside the section, it has type \(\text{nat} \rightarrow \text{list nat} \rightarrow \text{Prop}\). The major use of this representation within the development is in defining and proving the specifications of the algorithm for finding the lexicographically first MIS containing a given independent set.

\textbf{Section IS.}\n\textbf{Variable} \(V : \text{nat}\).\n\textbf{Definition} \(\text{Edge} \triangleq (\text{nat} \times \text{nat})\%\text{type}\).\n\textbf{Variable} \(E : \text{list Edge}\).\n\textbf{Variable} \(\text{NoSelfEdges} : \forall x : \text{nat}, \neg \ \text{In (x,x) E}\).\n\textbf{Variable} \(\text{Edges}_{lt}V : \forall x, y, \text{In (x, y) E} \rightarrow x < V\).\n\textbf{Variable} \(\text{EdgesBidirectional}\) :\n\quad \forall x, y : \text{nat}, \text{In (x,y) E} \rightarrow \text{In (y,x) E}\).\n\textbf{Definition} \(\text{ValidSet (X : list nat)} \triangleq \)
\quad \forall x : \text{nat}, \text{In x X} \rightarrow x < V\).

Figure 4.2: Section Based Graph Representation
The second graph implementation, shown in Figure 4.3 is a dependently typed construction using Coq’s record syntax. Coq records behave similarly to inductive types, generating the same set of induction lemmas, while also automatically generating a set of accessor functions for the components of the record. This construction is useful for proofs and programs that rely on relations between graphs, and this variation is used to derive an induction scheme for induced subgraphs, as well as the object which serves as the input to the implementation of Tsukiyama et al.’s algorithm.

One notable distinction about this dependently typed implementation is that, rather than carrying a proof which relates the entire set of edges to the set of vertices, like \( \text{Edges}_{\text{lt} V} \), this implementation relates the elements of each edge to the set of vertices through the use of the \( \text{Ix} N \) type. This type, based on the index types of Ssreflect [GM10], is a dependent pair of a natural \( i \) and a proof \( i < N \).

\[
\text{Record } \text{lGraph} \triangleq \mkGraph \{ \text{lV} : \text{nat}; \text{lE} : \text{list (Ix lV * Ix lV)}; \\
\text{llIrref} : \forall x : \text{Ix lV}, \neg \text{In} (x, x) \text{lE}; \\
\text{llSymm} : \forall x y : \text{Ix lV}, \text{In} (x, y) \text{lE} \leftrightarrow \text{In} (y, x) \text{lE} \}.
\]

Figure 4.3: Dependently Typed Graph Representation

### 4.2.1 Graph Predicates

Using these formulations of graphs, many of the requisite graph predicates have simple formulations in Coq. Figure 4.4 contains definitions for independence and maximal independent sets over the section-based graphs, parameterized by a \( \text{nat} V \) (the vertex count) and a \( \text{list (nat*nat)} E \) (the graph’s edge list). Equivalent notions for predicates over the record-based implementation of graphs are defined using those from Figure 4.4 and the use of a function \text{flatten_edges} which removes the indexing from the elements of a \text{list Ix} to produce a \text{list nat}. The definitions for these predicates are shown in Figure 4.5.
Definition ValidSet \((X : \text{list nat}) \triangleq \forall x : \text{nat}, \mathrm{In} x X \rightarrow x < V.\)

Definition Independent \((X : \text{list nat}) \triangleq \forall x y, \mathrm{In} x X \rightarrow \mathrm{In} y X \rightarrow \neg \mathrm{In} (x, y) E.\)

Definition IndSet \((X : \text{list nat}) : \text{Prop} \triangleq \text{ValidSet} X \land \text{Independent} X.\)

Definition MaximalIndSet \((X : \text{list nat}) : \text{Prop} \triangleq \text{IndSet} X \land (\forall x, \text{IndSet} (x::X) \rightarrow \mathrm{In} x X).\)

Figure 4.4: Section Based Graph Predicates, Parameterized by \(G = (V, E)\)

Definition ValidSet_lGraph \((G : l\text{Graph}) : \text{list nat} \rightarrow \text{Prop} \triangleq \text{ValidSet} (lV G).\)

Definition Independent_lGraph \((G : l\text{Graph}) : \text{list nat} \rightarrow \text{Prop} \triangleq \text{Independent} (\text{flatten}_\text{EdgesGraph} G).\)

Definition independent_lGraph \((G : l\text{Graph}) : \text{list nat} \rightarrow \text{bool} \triangleq \text{independent} (\text{flatten}_\text{EdgesGraph} G).\)

Definition Ind_lSet_lGraph \((G : l\text{Graph}) : \text{list nat} \rightarrow \text{Prop} \triangleq \text{IndSet} (lV G) (\text{flatten}_\text{EdgesGraph} G).\)

Definition MaximalIndSet_lGraph \((G : l\text{Graph}) : \text{list nat} \rightarrow \text{Prop} \triangleq \text{MaximalIndSet} (lV G) (\text{flatten}_\text{EdgesGraph} G).\)

Figure 4.5: Dependently Typed Graph Predicates

With the dependently typed representation, it is possible to generate a restricted notion of induced subgraphs, sufficient for implementing Tsukiyama et al.’s algorithm. This is done through through the use of the function LiftGraph shown in Figure 4.6. (The function is called LiftGraph because it is used to lift the indices in a graph \(G\) to \(n > lV G\) as well as to induce subgraphs.) The function reduceLEdges, used in the construction of LiftGraph, filters the edges from \(lE G\) that contain vertices greater than or equal to \(n\), while reduceLEdges_irref and reduceLedges_symm generate new proofs that the edges from reduceLEdges remain irreflexive and symmetric. Applying liftGraph to a graph \(G\) and an \(n\) such that \(n < (lV G)\) produces the subgraph of \(G\) induced by the first \(n\) vertices in \(G\), exactly the subgraphs that need to be generated in the recursive calls of Tsukiyama et al.’s
algorithm. Using this construction, it is possible to write a recursive function which generates increasingly smaller subgraphs by recursing downward over some natural number $n$ which generates, at each level, the subgraph of the first $n$ vertices.

**Definition** LiftGraph $(V' : \text{nat}) : IGraph \rightarrow IGraph \triangleq$

$\lambda G : IGraph \Rightarrow$

- $\text{mkListGraph } V'$
- $(\text{reduceLEdges } (\text{IV } G) ((\text{IE } G) V'))$
- $(\text{reduceLEdges}_\text{irref } (\text{IV } G) ((\text{IE } G) ((\text{Iirref } G) V'))$
- $(\text{reduceLEdges}_\text{symm } (\text{IV } G) ((\text{IE } G) ((\text{ISymm } G) V'))$.

Figure 4.6: LiftGraph Definition

Beyond providing a simple method of generating subgraphs, liftGraph can also be used to generate an induction principle, relating the properties of graphs produced by liftGraph to those of the input graph. Note that for any graph $G$, liftGraph $(\text{IV } G) G$ has no effect on either the number of vertices or the list of edges. Furthermore, both the input and output graphs carry proofs of the propositions regarding the irreflexivity and symmetry of their edges. Were it not for a small wrinkle, these values would act as a fixpoint for LiftGraph. The problem resides in the fact that while both graphs contain proofs of the same propositions, the proofs themselves are not equivalent. The proofs related to the graph output by LiftGraph are derived from those in the original graph and this derivation introduces new steps to the proof which prevent the new proofs from being identical, under the standard interpretation of Coq’s logic. In order to make these values a fixpoint of the function, it was necessary to introduce an axiom known as proof-irrelevance. This axiom states that all proofs of a proposition are (syntactically) equivalent. While not an elementary part of Coq’s logic, it can safely be added. By introducing this axiom and making these values a fixed point, it is possible to derive the induction principle shown in 4.7. Once
establishing the aforementioned fixpoint of $\text{LiftGraph}$, the proof of this principle is straightforward, and requires only induction over the natural number argument to $\text{LiftGraph}$.

**Theorem** $\text{InducedGraph}_{\text{ind}}$

$$\forall (P : \text{lGraph} \rightarrow \text{Prop}),$$

$$P \text{nil}_{\text{lGraph}} \rightarrow$$

$$(\forall G \; x, \; P (\text{LiftGraph} \; x \; G) \rightarrow P (\text{LiftGraph} \; (S \; x) \; G))$$

$$\rightarrow \forall G, \; P \; G.$$

Figure 4.7: Induction Principle by Induced Subgraphs

### 4.3 A Greedy Algorithm for Generating Lexicographically First Maximal Independent Sets

The final aspect of prerequisite work was the construction and verification of a greedy algorithm which takes as input a graph $G$ and an independent set $S$, and produces the first (according to the lexicographic order constructed earlier) maximal independent set in $G$ which contains $S$. This simple algorithm is an important subroutine in the implementation of Tsukiyama et al.’s algorithm. Figure 4.8 shows the implementation of this algorithm, $\text{MkMaximalIndSet}$, using the section based implementation of graphs from Figure 4.2. The function $\text{IndSetDec}$ is a simple decision procedure for determining if a particular set of vertices is an independent set with respect to $E$, the set of edges.
Fixpoint StepMaximalIndSet (X : list nat) (n : nat) \triangleq
  match n with
  | O \Rightarrow if IndSetDec (n::X) then (n::X) else X
  | S m \Rightarrow if IndSetDec (n::(StepMaximalIndSet X m))
    then (n::(StepMaximalIndSet X m))
    else StepMaximalIndSet X m
  end.

Definition MkMaximalIndSet (S : list nat) : list nat \triangleq
  StepMaximalIndSet S (pred V).

Figure 4.8: MkMaximalIndSet Definition, Parameterized by \( G = (V, E) \)

This greedy algorithm maintains a set \( S \), initialized by the independent set given as input. It attempts to add, in order from least to greatest, each vertex of the graph to the set \( S \). If the addition of the vertex still results in an independent set, the vertex is added to \( S \) for further iterations, otherwise, it is discarded.

4.3.1 Specifications

This program is used as a subroutine in the implementation of Tsukiyama et al.’s algorithm. As such, it was essential to prove that this program meets the specification mentioned above; that it produces the lexicographically first maximal independent set of a graph which contains a given independent set as a subset.

In the Coq development this specification is captured by the theorem MkMaximalIndSet_spec3 shown in Figure 4.9. In constructing the proof of this theorem, it was helpful to first prove the weaker claims made by theorems MkMaximalIndSet_spec1, that the output is an independent set, and MkMaximalIndSet_spec2, that the output is a maximal independent set.
Theorem MkMaximalIndSet_spec1:
\[ \forall X, \text{IndSet } X \rightarrow \text{IndSet } (\text{MkMaximalIndSet } X) \].

Theorem MkMaximalIndSet_spec2:
\[ \forall X, \text{IndSet } X \rightarrow \text{MaximalIndSet } (\text{MkMaximalIndSet } X) \].

Theorem MkMaximalIndSet_spec3:
\[ \forall X, Y : \text{list nat}, \text{IndSet } X \rightarrow \text{MaximalIndSet } Y \rightarrow \text{incl } X Y \rightarrow \text{dec_order } (\text{MkMaximalIndSet } X) = \text{lt_list} \lor \text{dec_order } (\text{MkMaximalIndSet } X) = \text{eq_list} \].

Figure 4.9: MkMaximalIndSet Specifications

4.3.2 Proof Techniques

The proof of MkMaximalIndSet_spec1 is relatively straightforward, and entails showing that independence holds as a loop invariant of the algorithm through induction on \( V \). However, the statements of the other two theorems are less straightforward. While the independence of the candidate set is preserved at every iteration of the program, the maximality of this set is a property that emerges only in the final iteration.

Maximality of the output of MkMaximalIndSet is derived by contradiction: assuming that there exists a new element which can be added to the output of MkMaximalIndSet which preserves independence. Essential to the derivation of this contradiction is the lemma IndSetConstraint_gen shown in Figure 4.10.

Theorem IndSetConstraint_gen:
\[ \forall (X : \text{list nat}) (n m : \text{nat}), \text{IndSet } X \rightarrow n < V \rightarrow m \leq n \rightarrow \neg \text{In } m \ (\text{StepMaximalIndSet } X n) \rightarrow \text{vertexConnected } m \ (\text{StepMaximalIndSet } X n) = \text{true} \].

Figure 4.10: IndSetConstraint_gen Definition
This lemma is proved by considering the order in which elements are added into the output of \texttt{StepMaximalIndSet} (and thus, \texttt{MkMaximalIndSet}). Using this lemma, makes it possible to show that any of these addiditional elements must in turn be connected to the output of \texttt{MkMaximalIndSet}, and thus their addition must violate independence.

The constructive logic which underlies Coq does not possess the law of the excluded middle (\( \forall P : \text{Prop}, P \lor \lnot P \)). This means, that for general propositions, it is not possible to construct proofs by contradiction. However, for certain propositions, there are computable methods for determining their truth or falsity. For such computable propositions, the law of the excluded middle can be shown to hold, and the truth or falsity of these propositions can be analyzed for use in proofs by contradiction. The lemma shown in Figure 4.10 uses one of these computable methods in its statement, \texttt{vertexConnected}. The function \texttt{vertexConnected} produces a boolean value by iterating through a list of vertices and checking to see if an edge exists between some input vertex and any member of the lists, returning true if such an edge exists and false otherwise. The (boolean) truth value of this function’s output is shown to relate to a proposition by the lemma shown in Figure 4.11.

\textbf{Theorem} \texttt{vertexConnected_spec}:

\( \forall (v : \text{nat}) \ (S : \text{list nat}), \)

\[ \texttt{vertexConnected} \ v \ S = \text{true} \Leftrightarrow \exists (v' : \text{nat}), \ln \ v' \ S \land \ln (v, v') \ E. \]

Figure 4.11: \texttt{vertexConnected_spec} Definition

Unlike propositions, booleans (and other inductive types) are susceptible to case analysis. By performing this case analysis over the output of \texttt{vertexConnected}, it is possible to analyze the truth or falsity of the corresponding proposition, and produce proofs by contradiction similar to those in classical logic.
In proving \texttt{MkMaximalIndSet-spec3}, we use a process similar to that used in the proof of \texttt{MkMaximalIndSet-spec2}. The prior lemmas ensure that the algorithm produces a maximal independent set, the only additional fact that needs to be proved is that the output of the algorithm is first according to the definition of lexicographic ordering. The proof acts by considering the result of \texttt{dec.order} when applied to the output of the program and an arbitrary set satisfying the constraints of the theorem. In the event that the output is less than or equal to this arbitrary set, the theorem holds trivially. In the case where it is not, the theorem derives a contradiction. This is accomplished by considering the element in the arbitrary set which causes it to be less than the output set, and showing that \texttt{MkMaximalIndSet} would have included that element in the output set, through the same sort of structural decomposition used to prove \texttt{IndSetConstraint-gen}.

### 4.4 Enumeration of Maximal Independent Sets

Building on the definitions and procedures described in the previous sections made the process of implementing Tsukiyama et al.’s algorithm straightforward. Figure 4.12 contains the implementation of Tsukiyama et al.’s algorithm within Coq. The cases shown in Figure 4.12 correspond to those in Figure 3.1.

This implementation is proved correct with respect to three specifications:

1. **Soundness**: Every $S$ in $\texttt{AllMIS}(G)$ is an MIS in $G$.

2. **Completeness**: Every MIS in $G$ is contained in $\texttt{AllMIS}(G)$.

3. **Uniqueness**: Every element in the output of $\texttt{AllMIS}(G)$ is unique.

The first and second specifications are identical to the properties proved when reviewing the algorithm in Chapter 2. The uniqueness specification is important here because of the choice to represent sets as lists. With this representation, there exists the possibility of generating duplicate sets, and this assertion ensures that only unique sets are generated by the program.
Using \texttt{InducedGraph\_ind} makes it easy to capitalize on the recursive structure of the \texttt{AllMIS} algorithm, and produces proofs whose structure closely mirrors the informal proofs of correctness from Chapter 2. At a high level, all of the proofs regarding the specifications of \texttt{AllMIS} proceed by double induction, first by applying \texttt{InducedGraph\_ind} to the graph under consideration, followed by induction over the output of the recursive call to \texttt{mkSetsAllMIS}.

\begin{verbatim}
Fixpoint mkCandidateSets (G : lGraph) (ℓ : list (list nat)) : list (list nat) ≜
  match lV G with O ⇒ nil::nil | S i ⇒
    match ℓ with nil ⇒ nil | cons cand ℓ' ⇒
      let ℓ₁ ≜ mkCandidateSets G ℓ' in
      if independent_lGraph G (i :: cand) then (*Case 1*) (i :: cand) :: ℓ₁
      else if is_MIS G (i::rmvNeighbors i G cand)
        then if LFMIS_dec (LiftGraph i G) (rmvNeighbors i G cand) cand
          then (*Case 2*) (i :: rmvNeighbors i G cand) :: cand :: ℓ₁
          else (*Case 3*) cand :: ℓ₁
        else (*Case 3*) cand :: ℓ₁ end end.

Fixpoint mkSetsAllMIS (V : nat) (G : lGraph) : list (list nat) ≜
  match V with O ⇒ nil :: nil
  | S i ⇒ mkCandidateSets G (mkSetsAllMIS i (LiftGraph i G)) end.

Definition AllMIS G ≜ mkSetsAllMIS (lV G) G.

\end{verbatim}

Figure 4.12: \texttt{AllMIS}: Compute All Maximal Independent Sets in $G$

\subsection{Soundness}

Soundness of the algorithm is shown by the theorem shown in Figure 4.13. The first step of the proof proceeds by applying \texttt{InducedGraph\_ind} to the goal, generating two subproofs. The initial states of these subproofs are shown in Figure 4.14.
Theorem AllMIS_sound : \( \forall G \ell, \text{In } \ell (\text{AllMIS } G) \rightarrow \text{MaximalIndSet}_{\text{IGraph}} G \ell. \)

Figure 4.13: Soundness of AllMIS

\[ \begin{align*}
L : \text{list nat} \\
H : \text{In } L (\text{PrintMIS } \text{nil}_{\text{IGraph}}) \\
\text{-----------------------------}(1/2) \\
\text{MaximalIndSet}_{\text{IGraph}} \text{nil}_{\text{IGraph}} L
\end{align*} \]

G : IGraph \\
x : nat \\
IHG : \forall L : \text{list nat}, \\
    \text{In } L (\text{PrintMIS } (\text{LiftGraph } x G)) \rightarrow \\
    \text{MaximalIndSet}_{\text{IGraph}} (\text{LiftGraph } x G) L

\[ \begin{align*}
L : \text{list nat} \\
H : \text{In } L (\text{PrintMIS } (\text{LiftGraph } (S x) G)) \\
\text{-----------------------------}(2/2) \\
\text{MaximalIndSet}_{\text{IGraph}} (\text{LiftGraph } (S x) G) L
\end{align*} \]

Figure 4.14: State of Soundness Proof Following Induction

The first goal is solved by simplifying PrintMIS in hypothesis H. Following this simplification, H transforms into an assertion that L is equal to nil. The remainder of this subproof involves showing that nil, corresponding to the empty set, is indeed an MIS in the empty graph.

The proof of the second subgoal involves unfolding the term PrintMIS (LiftGraph (S x) G) into an application of mkCandidateSets, and then showing that mkCandidateSets transforms MISs in LiftGraph \( x G \) into MISs in LiftGraph (S x) G, using the theorem shown in Figure 4.15.
Theorem mkCandidateSets_correct_ind :
\( \forall \ell \ x \ G, \)
\( (\forall y, \text{In} \ y \ \ell \rightarrow \text{MaximalIndSet}_l\text{Graph} (\text{LiftGraph} \ x \ G) y) \rightarrow \)
\( (\forall y, \text{In} \ y \ \text{mkCandidateSets} (\text{LiftGraph} (S \ x) \ G) \ell) \rightarrow \)
\( \text{MaximalIndSet}_l\text{Graph} (\text{LiftGraph} (S \ x) \ G) y). \)

Figure 4.15: Induction Lemma for Soundness

4.4.2 Completeness

The completeness of this algorithm is shown Figure 4.16. The proof of this statement begins using the same techniques as the proof of soundness: applying \text{InducedGraph}\_\text{ind} to perform induction via induced subgraphs, and solving the base case by simplifying the application of \text{PrintMIS} to the empty graph. In the inductive step of the proof, the proof follows essentially the same analysis as the informal proof of correctness from Chapter 3: for every MIS in \text{LiftGraph} (S \ x) \ G, the proof finds a corresponding MIS in \text{LiftGraph} \ x \ G which produces the MIS in \text{LiftGraph} (S \ x) \ G via \text{mkCandidateSets}. While understanding this proof at high level is not difficult, the proof in Coq is relatively tedious, encompassing over 300 lines. A large portion of the length of this proof is due to the fact that it is not enough to simply examine the execution of \text{mkCandidateSets} over the output of \text{LiftGraph} \ x \ G. Instead, this analysis must be repeated with respect to a number of potential forms that the output could take.

Theorem AllMIS_complete :
\( \forall G \ \ell, \text{MaximalIndSet}_l\text{Graph} G \ \ell \rightarrow \exists \ell', \text{list}_\text{eq} \ ell' \ \ell \wedge \text{In} \ \ell' (\text{AllMIS} G). \)

Figure 4.16: Completeness of AllMIS
4.4.3 Form Lemmas

The proof of uniqueness is similar to the proof of completeness described above, in that its proof involves relating elements in the output of the program to particular sets of vertices in the induced subgraph of the input. However, in the proof of completeness it was sufficient to show that this relation could be established between one element in the output and some MIS in the induced subgraph of the input. In the case of uniqueness, however, it is necessary to establish this relation between an element of the program’s output and a particular element in the prior recursive call of the program. To facilitate this, it was helpful to prove a series of three inversion lemmas (called “form” lemmas in our code) establishing this relation. These lemmas are shown in Figure 4.17.

**Lemma form1:** \( \forall x \ G \ L_{in} \ L_{out} \ l, \)

\[ \text{IV} \ G = S \ x \ \rightarrow \ \neg \text{In} \ x \ l \ \rightarrow \ \text{In} \ L_{out} \ \rightarrow \]

\[ \text{mkCandidateSets} \ (\text{IV} \ G) \ G \ L_{in} = L_{out} \ \rightarrow \ \text{In} \ L_{in}. \]

**Lemma form2:**

\( \forall x \ G \ L_{in} \ L_{out} \ l, \)

\[ \text{IV} \ G = S \ x \ \rightarrow \ \text{In} \ x \ l \ \rightarrow \ \text{In} \ L_{out} \ \rightarrow \]

\[ (\forall \ l', \ \text{In} \ l' \ L_{in} \ \rightarrow \ \text{MaximalIndSet} (\text{Graph} (\text{LiftGraph} x G) \ l')) \ \rightarrow \]

\[ \text{MaximalIndSet} (\text{Graph} (\text{LiftGraph} x G) (\text{rmv} x l)) \ \rightarrow \]

\[ \text{mkCS} \ (\text{IV} \ G) \ G \ L_{in} \ L_{out} \ \rightarrow \]

\[ (\forall \ l', \ \text{In} \ l' \ L_{out} \ \rightarrow \ \text{MaximalIndSet} (\text{Graph} G \ l')) \ \rightarrow \]

\[ \text{In} \ (\text{rmv} x l) \ L_{in}. \]

**Lemma form3:**

\( \forall x \ G \ L_{out} \ L_{in} \ l, \)

\[ \text{IV} \ G = S \ x \ \rightarrow \ \text{In} \ x \ l \ \rightarrow \ \text{In} \ L_{out} \ \rightarrow \]

\[ (\forall \ l', \ \text{In} \ l' \ L_{in} \ \rightarrow \ \text{MaximalIndSet} (\text{Graph} (\text{LiftGraph} x G) \ l')) \ \rightarrow \]

\[ \neg \text{MaximalIndSet} (\text{Graph} (\text{LiftGraph} x G) (\text{rmv} x l)) \ \rightarrow \]

\[ \text{mkCS} \ (\text{IV} \ G) \ G \ L_{in} \ L_{out} \ \rightarrow \]

\[ \exists l', \ \text{LFMIS} (\text{Graph} (\text{LiftGraph} x G) (\text{rmv} x l)) \ l' \wedge \text{In} \ l' \ L_{in}. \]

Figure 4.17: Form Lemmas
4.4.4 Uniqueness

As a final property of the algorithm, we show that AllMIS produces no duplicate entries in its output. This fact is captured by the theorem (and the predicate NoDuplicates) shown in Figure 4.18.

**Inductive** noDuplicates : list (list nat) → Prop △
| NoDup_nil : noDuplicates nil
| NoDup_cons : ∀l L, ~ list_eq_in l L → noDuplicates L → noDuplicates (l::L).

**Theorem** AllMIS_unique : ∀G, noDuplicates (AllMIS G).

Figure 4.18: Uniqueness of AllMIS and noDuplicates Specification

This proof is similar in length and structure to the earlier proof of completeness. In the proof of completeness, we related each element in the output to an MIS in the output of an earlier recursive call. Here, we relate each element in the output to a unique element in the output of a recursive call, via the form lemmas from earlier, and show that duplicate elements in the output could only be generated by duplicate elements in the earlier recursive calls.
5 Conclusions

The major contribution of this thesis is the program shown in Figure 4.12, as well as the proofs of correctness shown in Figures 4.13, 4.16, and 4.18. While the proof scripts and intermediate derivations are lengthy, and at times complex, it is easy to survey these results. Admitting the correctness of the proof assistant, the only portions of this development which require human review are the specifications regarding AllMIS and the definitions they rely on. A reviewer need only ensure that our definitions of graphs, maximal independent sets, uniqueness and list membership are defined correctly; the intermediate steps are ensured to be correct by the proof assistant.

However, in the pursuit of this algorithm and its specifications, we have developed a number of notable artifacts and techniques. The most promising of these is the derivation of an induction principle for graphs via induced subgraphs. This property, while seeing wide use in the derivation of many graph theoretic concepts, is notably absent from other formal developments. Similarly, lacking from other developments are formalizations of independence as well as lexicographic orderings over sets.

Despite the creation of these techniques and the successful implementation and verification of Tsukiyama’s algorithm, this avenue of research is by no means finished. Not only do these results possess a number of immediate applications, but there are a series of improvements to the development in its current state which would increase its utility and applicability.

5.1 Refinements

In the current development, there are two notable paths for improvement. First, a number of choices regarding the implementation of the program are responsible for decreased efficiency, and refactoring the development to include more efficient implementation choices would greatly increase the applicability of the generated code.
Additionally, in its current implementation, the algorithm uses fixed objects, such as natural numbers and lists in its representation of sets and vertices. As a second avenue for improvement, the implementation could be generalized to operate over arbitrary objects (possessing the necessary properties). Not only does such an abstraction better highlight the properties of an algorithm rather than an implementation, but in this case it could broaden the application of the development, as users could tailor the results to fit their requirements.

With respect to the inefficiencies in the current development, the most obvious of these is the choice to represent sets as lists. While their simplicity was useful in producing simple proofs, the inefficiency of operations such as checking for membership and removal of elements, could be reduced by using a more efficient structure to represent sets, such as balanced binary search trees. However, this inefficiency probably does not need to be addressed directly, as the aforementioned process of generalization will likely suffice to permit this substitution. Another cause for inefficiency is the large memory footprint of this implementation. In large part, this results from the fact that the program generates all of the maximal independent sets of the subgraph $G_n$ before generating those of $G_{n+1}$, essentially executing a breadth first traversal of the tree of possible MISs. A refactorization of the code which executed a depth first search of such a tree would significantly reduce the amount of memory required by the program.

Already, work on these improvements is underway. In our current attempt at incorporating these changes, we abstract the logical properties of graphs from their actual implementation via an interface similar to that presented by Erwig in his paper “Inductive graphs and functional graph algorithms” [Erw01]. Work on the refactorization of the program has not yet begun, but, using the developments presented in this thesis as a guide, we do not anticipate any significant difficulties in its formulation or verification.
5.2 Extensions

Beyond potential improvements, there are a number of immediate applications for the development presented in this thesis. The most immediate of these applications would be the use of this development in the implementation and verification of Lawler’s algorithm for graph coloring [Law76]. Additionally, since our implementation generates all maximal independent sets uniquely, there is some hope that this implementation can be used in the construction of a formal proof placing an upper bound on the number of maximal independent sets, similar to those derived by Moon and Moser [MM65]. Even if we are unable to derive such results directly from our implementation of Tsukiyama et al.’s algorithm, the derived induction principle and construction of graphs could be used in a more standard derivation of these bounds, as well as other results such as those by Johnson, Papadimitriou and Yannakakis [JPY88].
REFERENCES


