Graph homomorphisms

Dissertation

Course	MT4796: Joint Project
MT Supervisor	Dr James Mitchell
CS Supervisor	Prof Stephen Linton
Matriculation number	150015116
Date	April 28, 2019

Declaration

I certify that this project report has been written by me, is a record of work carried out by me, and is essentially different from work undertaken for any other purpose or assessment.

Contents

Abstract

For finite simple graphs, homomorphism cores form a distributed lattice. The join-irreducible and meet-irreducible elements are exactly connected cores and exactly multiplicative cores respectively. This arises to an equivalent statement to the Hedetniemi Conjecture; complete graphs are meet-irreducible. This questions the nature of the lattice, which can be analysed from visualizing Hasse diagrams. From the resulting visualizations, it can be proved that: for $n \geq 6$ vertices, there is exactly one such core that sits strictly between K_{n-3} on K_{n-2} on up to n vertices and for $n \geq 7$ vertices, there are exactly eight cores strictly between K_{n-4} and K_{n-3} on up to n vertices, with ordering preserved.

1 Introduction

Graph homomorphisms are a subject of study in pure mathematics and computer science, many aspects of which are still conjectured but not proven [HN92, Hed66]. Other than that, they have been used in various areas and applications, including frequency assignment problem, social networks analysis and statistical physics [FK02, WR83, BW04].

This work aims to provide the reader with theoretical background and understanding of homomorphisms on graphs, with respect to their context in lattice theory and relations on graphs. It will also guide the reader through the design and implementation of software that will be used to comprehend and analyse the structure of the homomorphic relation algorithmically and visually.

In general cases, determining whether graphs are homomorphic is an NP-Complete problem, which is considered even “harder” than graph isomorphism or graph coloring problems [HN04]. In practice, the problem takes an especially long time to solve, when no such homomorphism exists, as for these cases an algorithm would inefficiently attempt to find a solution without having a systematic way of knowing that there is none.

One approach of finding homomorphisms is reducing graphs under consideration to their cores, which are at most as big as the graphs. Though finding a core of a graph is a co-NP-Complete problem, it happens that most of the graphs on up to a given number of vertices are isomorphic to only few small cores [HN92]. Therefore, an algorithm that makes use of memoizing join-irreducible elements of the lattice of finite cores, or connected graphs, can significantly improve practical performance in an average case. It is, therefore, instrumental to investigate the structure of such lattice and prove observations.

Through the introduction, some preliminaries and aspects of graph homomorphisms will be introduced, with a focus on the homomorphic relation of cores. However, Understanding of the notations and terminology for set theory, binary relations, graphs and groups is required. The preliminary section includes additional terminology on graphs, which, on a certain level, will be specific to this dissertation, and lattices, since they are most relevant to the focus of this project and may not be assumed as a part of any taken course at the University. The reader also should have a background and understanding of complexity theory, constraint programming, scientific computing or programming, and information visualization, in order to be able to understand the software design and problems associated with solving homomorphisms. The main audience for this work is, therefore, deemed to be a senior honours student, taking a joint degree in computer science or mathematics at The University of St Andrews.

1.1 Lattice

Lattices turn out to be central to the properties of graph homomorphisms, so this section will give definitions and list important facts associated with general and distributive lattices.

Recall. Preorder is a binary relation that is reflexive, transitive, and partial order is a relation that is also antisymmetric: $a \sim b, b \sim a \implies a = b$.

Recall. Hasse Diagram of a poset is a transitively reduced direct acyclic graph that represents its relations [Wei]

A lattice is a poset, on which it is possible to define two well-defined operations, join and meet: any two elements have unique nearest common "parent" and unique nearest common "child".

Join and meet are sometimes also called greatest lower bound or lowest upper bound, and are define as follows:

Definition 1.1.1. Let $L = (S, \preceq)$ be a poset, and a, b be two elements in S.

Then if there is a unique c such that $a \preceq c, b \preceq c$, so that there is no such element d that $a \preceq d, b \preceq d, d \prec c$, then c is called *join* of a, b.

Definition 1.1.2. Let $L = (S, \preceq)$ be a poset, and a, b be two elements in S.

Then if there is a unique c such that $c \preceq a, c \preceq b$, so that there is no such element d that $d \preceq a, d \preceq b, c \prec d$, then c is called *meet* of a and b.

Lattices are posets, for which join and meet are defined for all elements:

Definition 1.1.3 ([Bir40])**.** Let $L = (S, \preceq)$ be a poset. Then L is a *meet-semilattice* if it has a meet defined for all pairs of its elements, and *join-semilattice* if it has a join defined for all pars of its elements. If L is both, a meet- and join- semilattice, it is called a *lattice.*

It is worth mentioning that by definition a *meet* and a *join* are unique for any pair of elements, and can hence be seen as a well-defined operations.

Notation. Let $a, b \in S$. Then denote their join as $a \vee b$ and meet as $a \wedge b$.

Example 1.1.4. Let $(\mathbb{N}, \mathcal{D})$ be a poset of natural numbers under divisibility. Then $L = (\mathbb{N}, \mathcal{D})$ is a lattice with $a \wedge b = \mathrm{lcm}(a, b)$ and $a \vee b = \gcd(a, b)$. By fundamental theorem of algebra, any integer can be uniquely represented as a product of prime powers. lcm is then a union of two factorizations, and gcd is an intersection of them.

Equivalently, lattices can be defined as an algebraic structure with join and meet operations. This way, lattices connect the properties of its operations to the properties of partial order relation:

Definition 1.1.5 ([SB81])**.** Let $L = (S, \vee, \wedge)$ be a set with two operations. Then it is a lattice if:

- $\vee, \wedge$ are commutative.
- $\vee, \wedge$ are associative.
- $\vee, \wedge$ are idemponent.
- For any $a, b, c : a = a \vee (b \wedge c) = a \wedge (b \vee c)$ (Absorption).

Theorem 1.1.6 ([SB81])**.** Definitions 1.1.3, 1.1.5 are equivalent.

The proof can be done by simply confirming that idemponence, associativity and transitivity, together with absorption, of the operations $\wedge$ and $\vee$ bidirectionally correspond to reflexivity, transitivity and anti-symmetry respectively, with $a \sim b \iff a = a \wedge b$ and $b = a \vee b$. The details of this proof are irrelevant to this project.

Some elements of the lattice are special in the sense that they can only be join (or meet) of some two elements if they are one of them. Such elements are called irreducible:

Definition 1.1.7 ([SB81])**.** Let $L = (S, \preceq)$ be a lattice. Then $x \in S$ is *join-irreducible* if for any $a, b \in S$, $x = a \vee b \implies x = a$ or $x = b$. Similarly, x is *meet-irreducible* if for any $a, b \in S$, $x = a \wedge b \implies x = a$ or $x = b$.

On a Hasse diagram, meet-irreducible elements will have only one out-bound neighbour, while join-irreducible elements will have only one in-bound neighbour.

A similar concept is a prime element. As opposed to having just in-bound or out-bound neighbour, an element can be a prime if that neighbour is also a prime of the same kind:

Definition 1.1.8 ([SB81])**.** Let $L = (S, \preceq)$ be a lattice. Then $x \in S$ is *join-prime* if for any $a, b \in S$, $x \preceq a \vee b \implies x \preceq a$ or $x \preceq b$. Similarly, x is *meet-prime* if for any $a, b \in S$, $x \succeq a \wedge b \implies x \succeq a$ or $x \succeq b$.

Proposition 1.1.9. If an element is join- (meet-) prime, it is also join- (meet-) irreducible.

Proof. Suppose not: let a be such that there are such b, c that $a = b \vee c$ and $a \neq b$ and $a \neq c$. Hence, $b \prec a$ and $c \prec a$. But if a is join-prime, then $a \preceq b \vee c$, then $a \preceq b$ or $a \preceq c$, which contradicts the previous bound on a. Therefore, it is not join-prime.

The dual argument on meet-primes can be proven in a similar way.

Hence, any join- (meet-) prime is also a join- (meet-) irreducible. □

Lattices also may have upper and lower bounds, whether the underlying set is finite or infinite:

Definition 1.1.10 ([SB81])**.** Let L be a lattice. Then if there exists $a = \wedge_{x \in L} x$, then a is called *lattice's bottom* of L and is denoted as 0_L. Similarly, if there exists such b, that $b = \vee_{x \in L} x$, then b is called *lattice's top* of L and is denoted as 1_L.

Proposition 1.1.11. If a boundary exists, it is unique.

Proof. Let L be a lattice and a, b be its distinct lower bounds. Then let $a = a \wedge b = b$. Contradiction.

By dual argument, 1_L is also unique if it exists. The proof is complete. □

Example 1.1.12. For lattice from Example 1.1.4, $0_L = 1$, since its factorization is an empty set, and 1_L is undefined, since any infinitely large set of prime powers corresponds to multiplication of them.

Definition 1.1.13 ([SB81])**.** Let L be a lattice, for which that 0_L and 1_L exist. Then L is *bounded.*

Remark. It is trivial that any finite lattice is bounded.

1.1.1 Sublattices

Similar to how subgroups are defined for groups, or subsets are defined for sets, the notion of sub-structure can similarly be extended for lattices:

Definition 1.1.14. Let $L = (S, \preceq)$ be a lattice, and $T \subseteq S$. Then $M = (T, \preceq)$ is a called a *sublattice* of L if it is also a lattice under the same join and meet operations.

Following the analogy, homomorphisms and isomorphisms can be defined on lattices too. Homomorphisms are a mapping of structures that preserve operations/relation defined on them. More broadly, groups, lattices, graphs and other structures form a category, which consists of two objects, e.g. lattices, and a morphism, e.g. homomorphism, between them. For a lattice, preservation of its relation implies preservation of its operations, and vice-versa, as shown by Theorem 1.1.6.

Definition 1.1.15. Let $L = (S, \wedge_L, \vee_L)$ and $M = (T, \wedge_M, \vee_M)$ be lattices. Then a *homomorphism* from L to M is a mapping $\phi : L \to M$ that preserves the operation for any $a, b \in S$: $\phi(a \wedge_L b) = \phi(a) \wedge_M \phi(b)$ and $\phi(a \vee_L b) = \phi(a) \vee_M \phi(b)$.

Isomorphism preserves not only the operation, but also the poset. It is therefore defined to be a bijective homomorphism:

Definition 1.1.16. Let L and M be lattices. Then $L \simeq M$, or L is *isomorphic* to M, if there exists a bijective homomorphism from L to M.

1.1.2 Distributivity

For some lattices, $\wedge$ and $\vee$ are distributive, and it turns out that lattices with such operations are isomorphic to some lattices of sets under $\subseteq$ relation, as will be shown in Theorem 1.1.24.

Definition 1.1.17 ([SB81])**.** Let $L = (S, \preceq)$ be a lattice. Then L is distributive if for any $a, b, c \in S$:

1. $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
2. $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$.

In fact, it is sufficient to prove either one the two conditions:

Theorem 1.1.18 ([Dav02])**.** Let L be a lattice. Then for $\vee$ distributes over $\wedge$ iff $\wedge$ distributes over $\vee$.

Proof. Let $a, b, c \in L$. If $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ holds, then the following are equivalent:

$$(a \vee b) \wedge (a \vee c) \tag{1.1.1}$$
$$((a \vee b) \wedge a) \vee ((a \vee b) \wedge c) \tag{1.1.2}$$
$$a \vee ((a \wedge c) \vee (b \wedge c)) \tag{1.1.3}$$
$$a \vee (a \wedge c) \vee (b \wedge c) \tag{1.1.4}$$
$$a \vee (b \wedge c) \tag{1.1.5}$$

Therefore, $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$. By dual argument, the converse also holds. Hence the proof is complete. □

As a property of the operation, distributivity is preserved on sublattices:

Proposition 1.1.19. Let L be a distributive lattice, and M be its sublattice. Then M is also distributive.

Proof. Since $\vee$ and $\wedge$ operations are similar on L and on M, they distribute over each other in M. Hence, M is distributive. □

And it turns out that irreducibles are exactly the primes for distributive lattices, which is a very useful fact:

Theorem 1.1.20 ([Dav02])**.** In a distributive lattice, an element is join-irreducible iff it is a join-prime, and meet-irreducible iff it is a meet-prime.

Proof. This will be proven both ways separately:

($\Leftarrow$) This is true for lattices in general by Proposition 1.1.9.

($\Rightarrow$) Let $L = (S, \wedge, \vee)$ be a distributive lattice.

Consider join-irreducible a, and any b, c: $a \preceq b \vee c$. Then $a = a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c) \implies a = a \wedge b$ or $a = a \wedge c$. Therefore, $a \preceq b$ or $a \preceq c$, i.e. a is a join-prime.

By a dual conclusion, any meet-prime element is also meet-irreducible.

Hence, join- (meet-)irreducible elements of a distributed lattice are exactly the join- (meet-)prime elements. □

An important kind of distributive lattice are lattices of sets, defined below:

Definition 1.1.21. Let S be a set of sets. Then a lattice $(S, \subseteq)$ is called a *lattice of sets*, with $\wedge = \cap$ and $\vee = \cup$.

Proposition 1.1.22. Lattices of sets is distributive.

Proof. From assumed knowledge on set theory, $\cap$ and $\cup$ distribute over each other. Therefore, lattices of sets are distributive. □

The position of an element can often be described by considering sets of elements that precede (or succeed) a given element. These sets uniquely determine the element.

Definition 1.1.23. Let L be a lattice. Then for any $a \in L$, *lower set* of a is a set of all elements b in L such that $b \preceq a$, and the *upper set* of a is a set of all elements b in L such that $a \preceq b$.

It turns out that for finite distributive lattices, join and meet of two elements is essentially the same as join and union of some sets that they can bijectively map to:

Theorem 1.1.24 (Birkhoff's representation theorem [B$^+$37, Dav02])**.** Let L be a finite lattice. Then L is distributive iff it is isomorphic to a lattice of sets.

Proof [Wel82]. Let L be a finite lattice. The theorem can be proven both ways as follows:

($\Leftarrow$) By Proposition 1.1.22, any lattice of cores is distributive. Let ϕ be an isomorphism to L. Then for any a, b in the lattice of cores, $\phi(a) \preceq \phi(b) \iff a \subseteq b$. Hence, L is also distributive.

($\Rightarrow$) Let $\mathcal{J}(a)$ be the set of all join-prime elements of a that belong to its lower set, and let $\mathcal{L}(a)$ denote the lower set of a, so $\mathcal{J}(a) \subseteq \mathcal{L}(a)$. Then, clearly, it is well-defined, since any element has its own lower set, and clearly $\preceq$ order is preserved on it as set inclusion, so it is a homomorphism. Prove by induction on $|\mathcal{L}(a)|$.

Base case is that $\mathcal{L}(a) = \{x\}$. Then $0_L = \bigwedge_{e \in L} L \preceq x$ is contained in $\mathcal{L}(a)$, hence $x = 0_L$. Clearly, the only element with lower set of size 1 is 0_L, and the join of $\mathcal{J}(a) = \{0_L\}$ is 0_L.

The assumption is that $\mathcal{J}$ is a bijection for all $|\mathcal{L}(a)|$. Let b be such that $\mathcal{J}(a) = \mathcal{J}(b)$ and b is a join of $\mathcal{J}(b)$. Since $b = \bigvee \mathcal{J}(b)$, $b \preceq a$. If a is join-prime, it is contained in its own $\mathcal{J}(a)$, then $b \succeq a$, therefore $b = a$. Otherwise, it is a join of two or more strictly smaller elements, so by assumption all those elements are joins of their $\mathcal{J}$s, which are contained in $\mathcal{J}(a)$, since $\mathcal{J}$ it is a homomorphism that preserves inclusion. Therefore, $a \preceq b$, which together with the previous bound yields that $a = b$, and thus $\mathcal{J}$ is a bijection.

Hence any finite lattice is distributive iff it is isomorphic to a lattice of sets. □

This way, any element of a finite distributive lattice can be constructed from join-primes of its lower set, similar to factorization for numbers, so that the join and meet operations on elements are essentially union and intersection of such sets of smaller join-primes.

The topology of infinite lattices (see Figure 1.1.1) has been studied by Stone, Maclane, Preistley and others [Mac38, Pri70]. The result of Theorem 1.1.24 for infinite case has been known for Boolean algebras, and the sufficient generalization for distributive lattices can be found in a 1970 publication "Representation of distributive lattices by means of ordered Stone spaces", authored by Priestley [Pri70].

The proof of that result will not be a part of this dissertation, and isomorphisms of distributed lattices are a subject of study of representation theory. The theory ise used to describes algebras in terms of their isomorphisms to each other, and distributed lattices in particular, to lattices of sets, ordered Stone's spaces and others [Pri70].

An important property of non-finite lattices is density, is a convenient notion for describing their structure.

Definition 1.1.25. Let L be a lattice. If for any a, b in it there exists c such that $a \prec c \prec b$, then L is called *dense*.

Distributed and dense lattices will be used in Section 1.5 to describe the order of homomorphism on finite graphs.

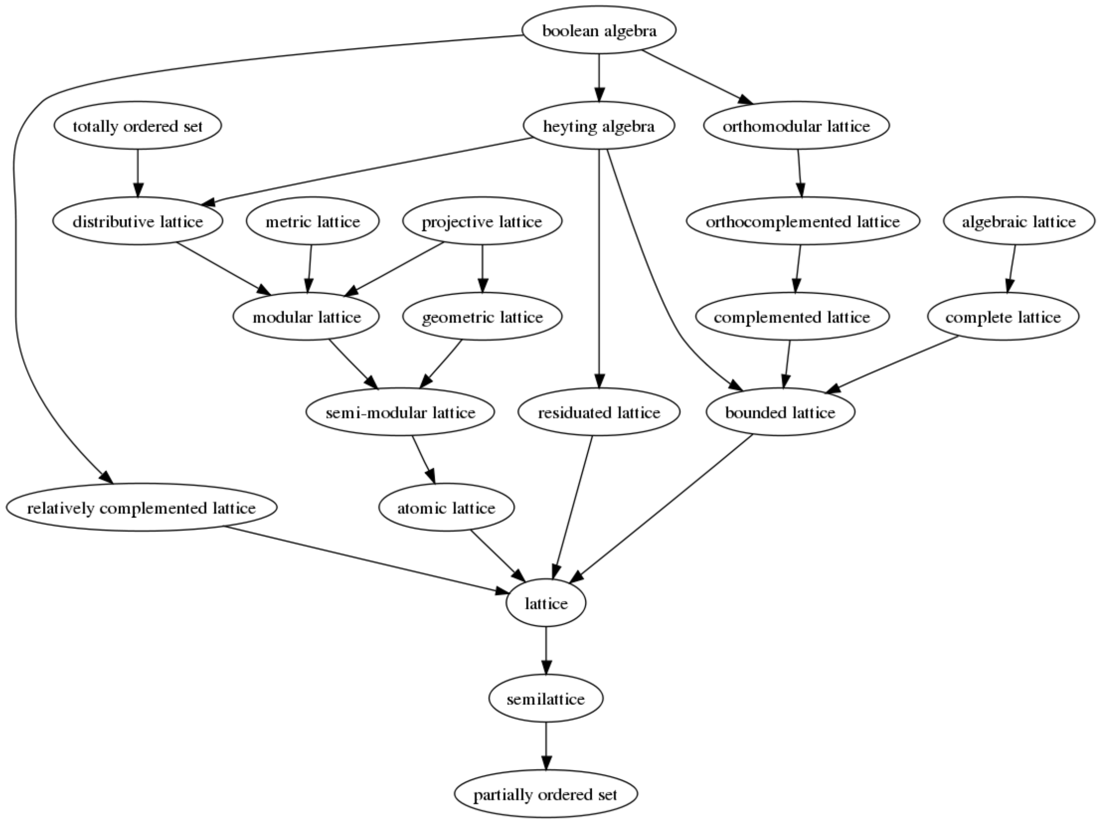

Figure 1.1.1: Hasse diagram of lattices [Wik12].

1.2 Graph

This section will clarify the language used to describe subgraphs and graph compositions to resolve possible ambiguity in the notation.

1.2.1 Subgraphs

The following clarifies subgraph terminology used in this writing:

Definition 1.2.1. *Node-induced subgraph* H of G is a graph, such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$ with a difference that the connectivity of any two nodes in H is the same as in G. If $H \neq G$, H is called a *proper node-induced subgraph* of G. G is then called *node-induced supergraph*, or *proper node-induced supergraph* respectively.

A node-induced subgraph is obtained by removing vertices from a graph and all edges connected to them with either end.

Example 1.2.2. An example of the above is shown on Figure 1.2.1: a subgraph on red nodes is node-induced from the graph on red and grey nodes, and hence only edges between red nodes are a part of the subgraph.

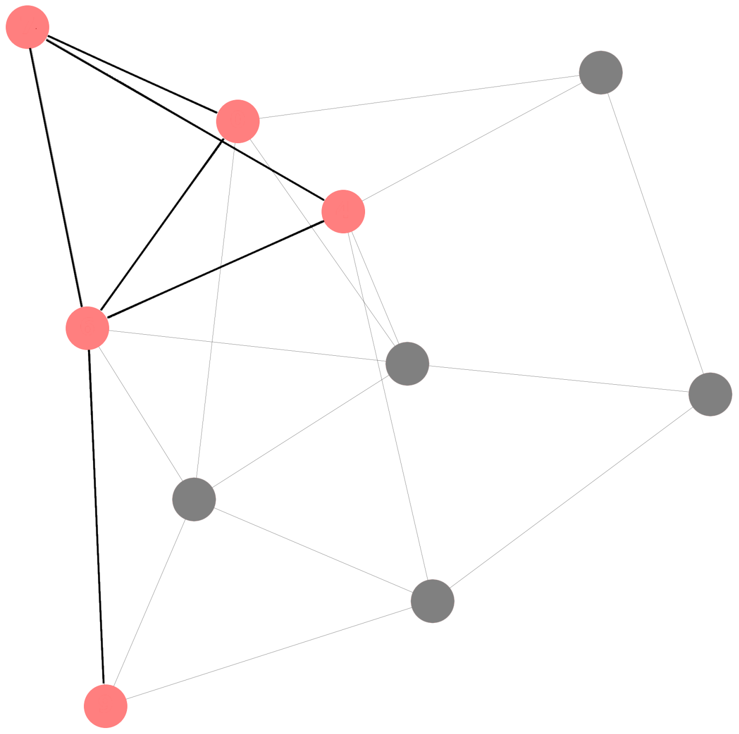

Figure 1.2.1: Example of a node-induced subgraph.

A graph can therefore be split into node-induced subgraphs:

Definition 1.2.3. Let G be a graph, and $\{X_1, \ldots, X_n\}$ be partitioning of $V(G)$. Then say that G is *node-partitioned* into $H_1, \ldots, H_n$ if each H_i is a node-induced subgraph with $V(H_i) = X_i$.

A similar, but more generalized notion of node-induced subgraphs is (edge-)induced subgraphs:

Definition 1.2.4. *(Edge-)induced subgraph* H of G is a graph, such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. If $H \neq G$, H is called *proper induced subgraph.* G is then called *(edge-)induced supergraph* and *proper induced supergraph* respectively.

Induced subgraphs are obtained by removing not only vertices, but also edges from a given graph. For example, P_3 is an (edge-)induced subgraph of C_3, since it can be obtained by removing any one edge from C_3. However, it is not a node-induced subgraph, as any proper node-induced subgraph would have at most 2 nodes. For C_4, it is on the contrary also a node-induced subgraph, which is obtained by removing any one of its nodes.

Example 1.2.5. Consider the graph on red and grey vertices, as shown on Figure 1.2.2. The node-induced subgraph on red nodes will contain both, black and light-blue edges, while the edge-induced subgraph on red nodes may contain any subset of edges between them, for example, only black edges.

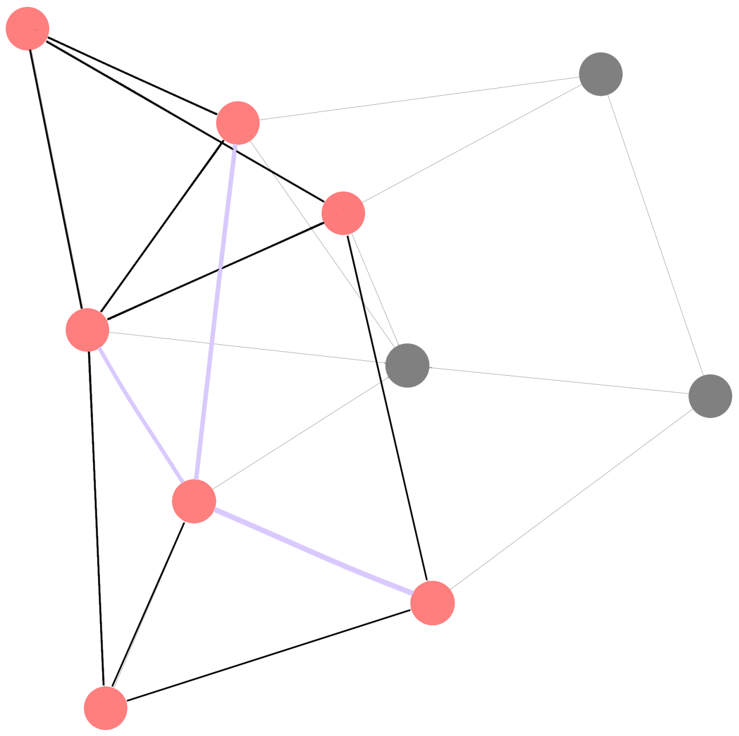

Figure 1.2.2: Example 1.2.5: Node- and edge-induced subgraphs.

1.2.2 Operations

This note will define graph products and composition that will be relevant for analysis of further content, related to graph colouring and homomorphisms.

Definition 1.2.6. Let G and H be graphs. Then $G \cup H = (V(G) \cup V(H), E(G) \cup E(H))$.

Remark. When $V(G) \cap V(H) \neq \varnothing$, which will never be the case in this dissertation, the two vertex sets will be considered as disjoint with respect to the graphs that they represent.

This operation simply takes two graphs, and produces a graph that contains both graphs, with no edges between G and H in it. G and H then node-partition $G \cup H$.

Similarly to union, define $G \setminus H$:

Definition 1.2.7. Let G - graph, and H be an induced subgraph or a vertex set. Then denote a node-induced subgraph of G that consists of all its nodes that are not in H to be $G \setminus H$.

Example 1.2.8. Trivially, $(G \cup H) \setminus H = G$.

The next operation takes a union of graphs and adds all edges between them:

Definition 1.2.9. [[Wei]] Let H, K be graphs. Then a *graph join* of H and K, defined as $G = H + K$, is a graph such that any $a \in H, b \in K$, are connected and $\{H, K\}$ node-partition G.

Remark. The case when $V(G) \cap V(H) = \varnothing$ is dealt with in precisely the way described in Remark 1.2.2. This will, too, never be the case in this dissertation.

Proposition 1.2.10. Graph join is well-defined and is commutative and associative.

Proof. Let H, K be graphs. Then $G = H + K = (V(H) \cup V(K), E(H) \cup E(K) \cup \{(a, b) \mid a \in H, b \in K\})$. Hence, graph join is well-defined. Since $\cup$ is associative and commutative, graph join is also associative and commutative. □

A special case of graph join is a join of complete graphs:

Proposition 1.2.11. Let n, m be integers. Then $K_n + K_m = K_{n+m}$.

Proof. Trivially, $K_n + K_m$ has $n + m$ vertices. Let $G = K_n$ and $H = K_m$. Then any vertex in G is connected to all vertices in G and all vertices in H, by Definition 1.2.9. Similarly, any vertex in H is connected to all other vertices. Hence by definition $G + H$ is complete on $n + m$ vertices. □

Another operation, that plays an important role in this dissertation, is direct product:

Definition 1.2.12. Let G, H be graphs. Then $K = G \times H$ is a graph such that $V(K) = V(G) \times V(H)$ and for any two vertices a, b in G and any pair of vertices m, n in H, $((a, m), (b, n)) \in E(K) \iff (a, b) \in E(G), (m, n) \in E(H)$. This operation is called *tensor product*, *categorical product* and *direct product*.

Example 1.2.13. Consider two graphs, shown on Figure 1.2.3. For convenience, nodes a and c of G are coloured in red, and nodes b and d - in blue. Note that the vertex set of the product is a set of pairs. Consider vertex $(a, 0)$ in K. In G, a is connected to b and d. In H, 0 is connected to $1, 2$ and 3. Hence, the neighbours of $(a, 0)$ is $\{b, d\} \times \{1, 2, 3\}$, and other edges can be deduced similarly, as shown on Figure 1.2.4. Hence, K, shown on Figure 1.2.4, is a categorical product of G and H.

This section is a preliminary for more complex properties and relations on graphs, which will be defined and explained in the further content of this dissertation.

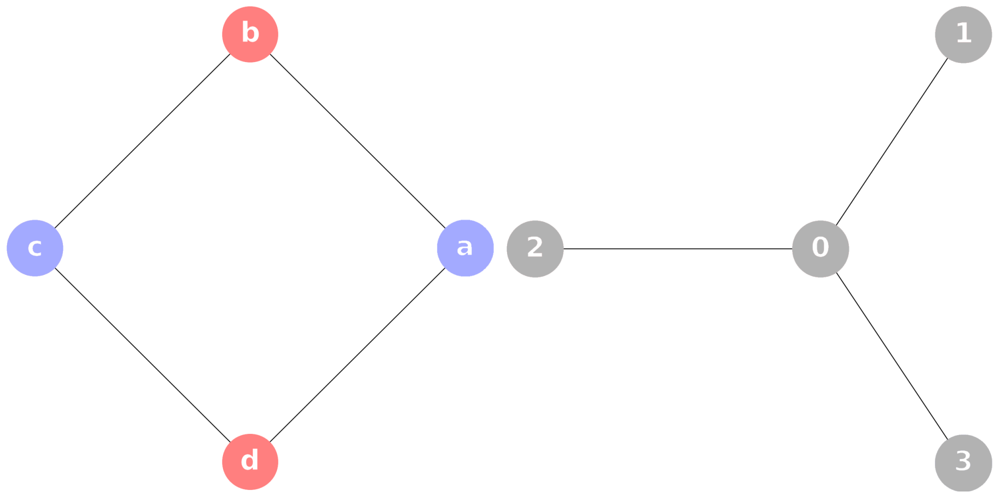

Figure 1.2.3: Example 1.2.13: direct product between G (on the left) and H (on the right).

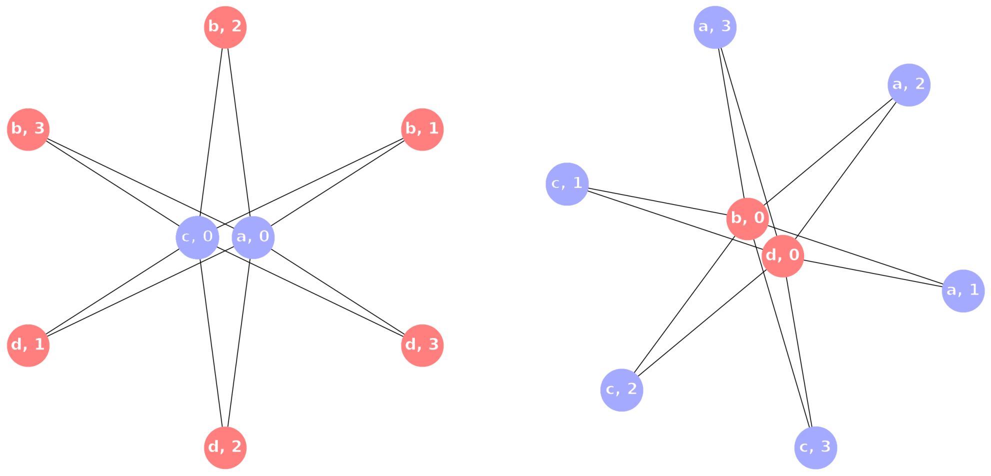

Figure 1.2.4: Example 1.2.13: $K = G \times H$.

1.3 Graph isomorphism

This section will re-define what graph isomorphisms are, together with associated concepts, and explain their connection to group theory.

Similarly to groups, lattices and others, graph isomorphisms preserve the structure of a graph. For example, given two graphs, shown on Figure 1.3.1, the problem is to determine whether there is a way to re-label the first graph to obtain the second.

Definition 1.3.1. Let G and H be graphs.Then G is *isomorphic* to H, denoted as $G \simeq H$, iff there exists a bijective mapping $\phi : G \to H$, such that for any $a, b \in V(G)$, $(a, b) \in E(G) \iff (\phi(a), \phi(b)) \in E(H)$. ϕ is then called a *(graph)isomorphism.*

Isomorphism forms equivalence classes on graphs, each of which can be represented by

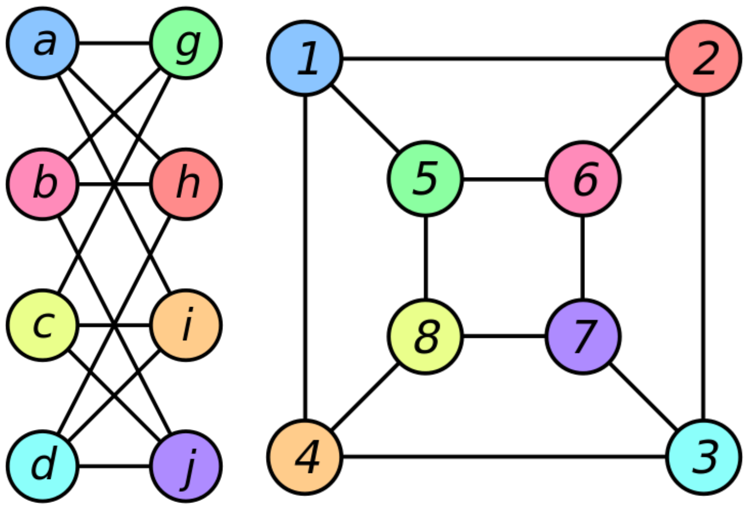

Figure 1.3.1: Two isomorphic graphs [Com14a, Com14b].

a so-called "unlabelled" graph, with each member of the class being a specific labelling of it. If these labellings are somehow "minimized", they are said to be reduced to a canonical form.

Definition 1.3.2 ([MP14])**.** *Canonical form* C of G, denoted as $\text{Canon}(G)$, is any function $G \to C$, such that:

- $G \simeq C$
- For any graph H, $G \simeq H \implies \text{Canon}(H) = C$.

Essentially, canonical form of a graph is a function that determines canonical, e.g. lexicographically smallest, labelling of a given graph.

Proposition 1.3.3. Canonical form is a well-defined function.

Proof. All graphs are partitioned by isomorphism classes, and by definition any canonical form reduces two graphs to the same graph if and only if they are isomorphic. Therefore, it reduces each graph to a unique representative of its class.

Hence, any such mapping $\text{Canon}(G)$ is well defined. □

A special kind of isomorphisms is an automorphism. As opposed to forming a class of graphs that is isomorphic to an "unlabelled" graph, automorphism is a "labelled" isomorphism, or an isomorphism that keeps the structure of labels. It can also be thought of as an isomorphism to itself, in terms of the category of graphs:

Definition 1.3.4. Let G be a graph. Then $\phi : G \to G$ is called an *automorphism* of G if it is an isomorphism (to itself).

Similarly to isomorphisms, automorphisms form an equivalence class. It turns out that this class is a group:

Theorem 1.3.5 ([W+96])**.** Let G be a graph, and $Aut(G)$ be a set of its automorphisms. Then $Aut(G)$ is a group.

Proof. This can be proven directly from definition:

- **Closure**. Let ϕ and ψ be automorphisms of G. Then $\phi\psi$ is an automorphism, since isomorphism is transitive.
- **Identity**. Identity map is clearly the identity element of $Aut(G)$.
- **Inverse**. As shown before, an isomorphism function is symmetrical, hence ϕ^{-1} is an isomorphism $\iff$ ϕ is an isomorphism.
- **Associativity**. Let ψ, ψ, λ be isomorphisms. From definition it is trivial that all automorphisms are permutations of vertices by Cayley's theorem.

Hence $Aut(G)$ is a group. □

In other words, every graph has an automorphism group that describes all of its symmetries. Therefore, each isomorphism class can be partitioned by automorphism groups:

Unless stated otherwise, all groups will be considered the same if they are isomorphic.

Corollary 1.3.6. Let $Ism(G)$ be an isomorphism class of G. Then any finite subset of labellings of G forms a group, which is constructed from its disjoint automorphism groups.

Proof. Trivially, any such subset of an isomorphism class can be partitioned by automorphism classes, which are automorphism groups. Then the isomorphism group can be constructed as a direct product of all its (identical) automorphism groups. □

The operation in that group would be a composition of changing labelling from e.g. $\{a, b, c, d\}$ to $\{1, 2, 3, 4\}$ and a permutation within the labelling. An example of both is shown on Figure 1.3.1, where the colors of vertices indicate the transformation.

Remark. Similarly, any finite subset of all graphs can be partitioned by such finite subsets of isomorphism classes, and such subset forms a group. The operation is a composition of changing an isomorphism class, e.g. adding an edge; relabelling and consequent permutation of vertices. As opposed to a subset of an isomorphism class, however, automorphism groups can differ between different isomorphism classes.

The natural question is which groups are represented by graph automorphisms. It turns out that the quantitative relation of groups to automorphism groups is one to many. In help of proving that it will be useful to introduce Lemma 1.3.7:

Lemma 1.3.7 ([Lov07])**.** Let $\Gamma = \{g_1, \dots, g_n\}$ be a group, and let G be a digraph such that $V(G) = \Gamma$, and $E(G)$ is defined by joining (g_i, g_j) with a color k such that $g_k = g_i g_j^{-1}$. Then $Aut(G) \simeq \Gamma$.

Proof [Lov07]. Let g_i, g_j be arbitrary. Consider $g_k = g_i g_j^{-1}$. Then for any $x \in \Gamma$, $g_k = (g_i x)(g_j x)^{-1}$. Thus, multiplication by x maps edges of color k to edges of color k. Therefore, it permutes edges in the induced subgraph on vertices of any color, hence multiplication by any element is an automorphism.

Let ϕ be any automorphism of G. Since $g_i = (g_i x)x^{-1}$, $g_i x$ and x are joined by an edge of color i. On the other hand, for $x = g_i^{-1}$, g_i is connected to id_Γ with an i-colored edge. Since ϕ is automorphism, $\phi(g_i)$ is joined to id_Γ by an edge of color i. Therefore $\phi(g_i) = g_i x$, i.e. any automorphism is a multiplication by some x.

Hence, automorphisms are exactly multiplications by elements in Γ, i.e. $Aut(G) \simeq \Gamma$. □

Theorem 1.3.8 (Frucht's theorem [Fru39, Bab96])**.** Given any group Γ there exists a graph G such that $\Gamma \simeq Aut(G)$.

Proof [Bab96, Lov07]. Assume $|\Gamma| \geq 2$. Consider graph $\overrightarrow{G}$ from Lemma 1.3.7.

Let F_k be a graph constructed in the following way: take P_{k+2}, and attach a connected vertex to nodes from first to second last node of P_{k+2}, or path of length 2 if it is the last node. An example of such graph for $k = 2$ is shown on Figure 1.3.2, where the first node is connected to g_i and the last node is connected to g_j. Notice, that F_k does not have any symmetries. Then construct a new graph G from $\overrightarrow{G}$ by replacing directed edges of color k by node-induced subgraphs F_k.

Clearly, any automorphism of $\overrightarrow{G}$ induces a unique automorphism on G. $\overrightarrow{G}$ can be recovered from node-induced subgraphs of G that are isomorphic to F_k: the ending of such graph, to which a path on two nodes is attached, indicates the direction of a colored edge, and the length of the path from first to last node corresponds to the parameter k in F, and color therefore. This way, any g_i is connected to any g_j as shown on Figure 1.3.2.

Notice also, that any vertex in F_k is a cut-point or end-point. Therefore, a subset of vertices that are not an end-point nor a cut-point in $\overrightarrow{G}$ is exactly the vertices of such property $V(\overrightarrow{G})$ in $V(G)$. Therefore, all node-induced graphs that are isomorphic to F_k bijectively correspond to the colored edges in $\overrightarrow{G}$. Thus, automorphism ϕ of G will connect $\phi(g_i)$ to $\phi(g_j)$, and therefore, ϕ corresponds to an automorphism in $Aut(\overrightarrow{G})$.

Hence, $\Gamma \simeq Aut(G)$. Given that Γ is a non-trivial group that is chosen arbitrarily, the proof is complete. For a trivial group, clearly, the automorphism of the graph on Figure 1.3.2 is trivial. □

Proposition 1.3.9. Any group is an automorphism group of infinitely many non-isomorphic graphs.

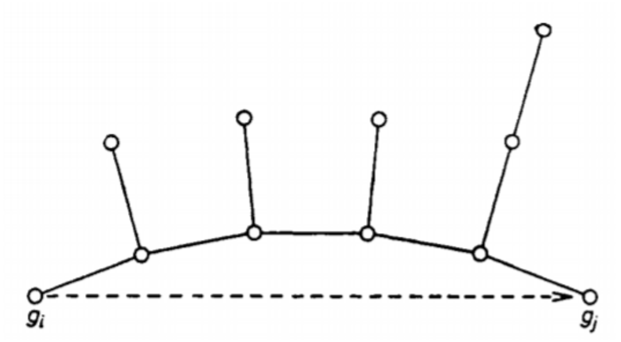

Figure 1.3.2: F_k which replaces colored directed edge.

Proof. Consider F_k described in Proof of Frucht's Theorem 1.3.8. For a given k, set the lengths of the attached paths to be k different numbers in ascending order. Then clearly an automorphism group of such graph is trivial, and any vertex is a cut-point or end-point. The direction of it can therefore be identified by the size of the last attached path, as it is greater than the length of the first attached path. Then the graph G in the proof can be constructed using such family of graphs, of which there is an infinite amount.

Hence, any group corresponds to infinitely many automorphism groups of infinitely many non-isomorphic graphs. □

This way, each group is represented by infinitely many automorphism groups of graphs that belong to different isomorphism classes. Similarly, any finite subset of subsets of isomorphism classes that satisfies the condition set in Corollary 1.3.6 forms a group, which is also an automorphism group of some graph in another class. Graph isomorphism is an extreme case of a homomorphic equivalence on graphs, which will be introduced in Section 1.5. Unless stated otherwise, from this point on-wards two graphs will be considered the same if they are isomorphic to each other.

1.4 Graph colouring

Colouring a graph with n colours means that a colour from 1 to n is given to each vertex in such a way that no two connected vertices are allowed to have the same colour. It is a special case of graph homomorphisms.

The following is a possible definition of graph colouring:

Definition 1.4.1. Let G be a graph, and $n \in \mathbb{N}$. Then G is *n-colourable*, if there exists a mapping $\phi : G \to \mathbb{Z}_n$, such that for any two connected nodes a and b, $\phi(a) \neq \phi(b)$.

Naturally, chromatic number is a closely related concept:

Definition 1.4.2. Let G be a graph. Then *chromatic number*, denoted as $\chi(G)$, is the smallest n such that G is n-colourable.

Theorem 1.4.3. Let G be a graph, and H be its (node-/edge-)induced subgraph. Then $\chi(G) \geq \chi(H)$.

Proof. Assume the opposite. Let $n = \chi(G), m = \chi(H)$ and $n < m$. Then let ϕ be n-colouring of G. Let $\lambda : H \to \mathbb{Z}_n$ be defined as $\lambda(x) = \phi(x)$. Then λ is an n-colouring of H. Contradiction. □

Bounds on chromatic numbers are an open topic in mathematics and computer science, and finding them is an NP-Complete problem, as will be explained in Section 1.6.11. There are other unsolved problems related to it, which will be introduced shortly.

The result from Theorem 1.4.3 has many implications. For example:

Theorem 1.4.4. Let G, H, K be graphs such that $G = H \cup K$. Then $\chi(G) = \max(\chi(H), \chi(K))$.

Proof. Without loss of generality, assume $\chi(H) \geq \chi(K)$. By Theorem 1.4.3, $\chi(G) \geq \chi(H)$, since H is a node-induced subgraph of G. Then there exists a colouring $\phi : H \to \mathbb{Z}_{\chi(H)}$. Similarly, there exists a colouring $\psi : K \to \mathbb{Z}_{\chi(K)}$. Since $\chi(K) \leq \chi(H)$, there also exists a colouring $\phi : K \to \mathbb{Z}_{\chi(H)}$. Then construct a mapping $\lambda : G \to \mathbb{Z}_{\chi(H)}$ so that given a vertex v in G, $\lambda(v) = \begin{cases} \phi(v) & v \in H \\ \psi(K) & v \in H \end{cases}$. The mapping is well-defined, since by definition H and K node-partition G, and, trivially, λ is a valid $\chi(H)$-colouring, since there are no edges between H and K. Hence, $\chi(G) \leq \chi(H) \leq \chi(G)$.

Therefore, $\chi(G) = \max(\chi(H), \chi(K))$. □

The "opposite" of Theorem 1.4.4 can be summarized by Theorem 1.4.5:

Theorem 1.4.5. Let G be a graph, and its node-partitioning be H, K. Then $\chi(G) \leq \chi(H) + \chi(K)$. The equality is attained, but only necessarily, when $G = H + K$.

Proof. Let G' be an induced supergraph of G, where any two nodes $a \in H, b \in K$ are connected. Then by Theorem 1.4.3 $\chi(G) \leq \chi(G') = \chi(H) + \chi(K)$. □

Remark. Specifically, let $G = H + K$. Then $\chi(G) = \chi(H) + \chi(K)$.

Example 1.4.6. Let G and H be graphs, as shown on Figure 1.4.1. Notice, that both graphs are bipartite, meaning $\chi(G) = \chi(H) = 2$. Their graph join is shown on Figure 1.4.2. Clearly, the graph contains both G and H as its node-induced subgraphs, as indicated by colours. Both graphs can only be coloured in two separate sets of colours each, which do not intersect. Hence, $\chi(G + H) = 4$.

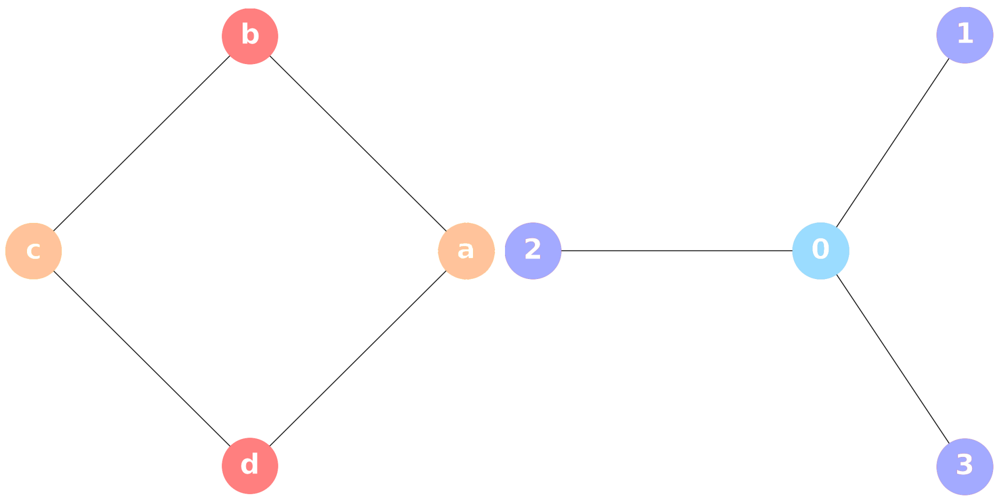

Figure 1.4.1: Example 1.4.6: G (on the left) and H (on the right).

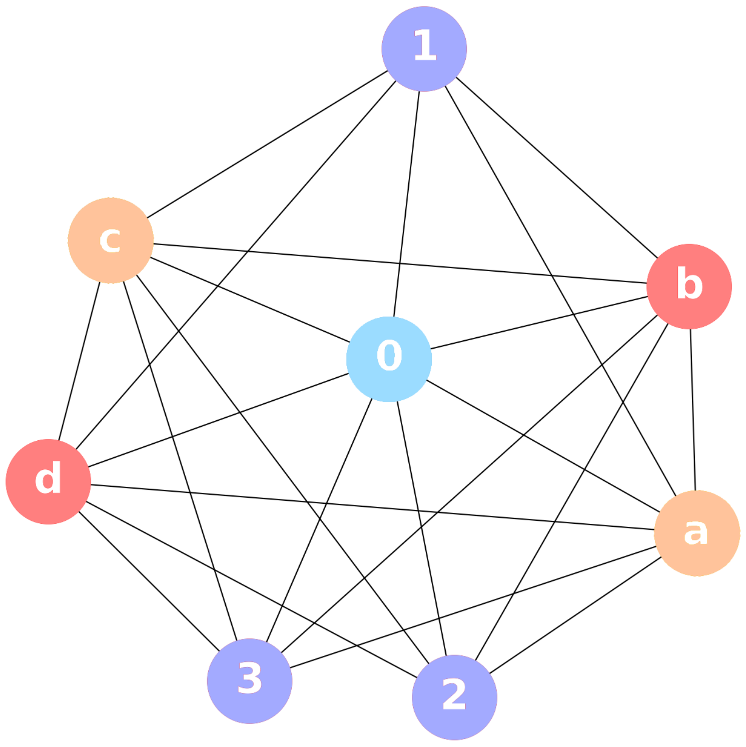

Figure 1.4.2: Example 1.4.6: $G + H$.

Another kind of composition, which has been introduced in Section 1.2.2, is direct product (Definition 1.2.12). The chromatic number of a direct product is not yet discovered for two arbitrary graphs, but there is a conjecture which is often attributed to Hedetniemi, who formulated it in his technical report in 1966 [Hed66]. Sauer in his work also claimed that other graph theorists, including Lovasz, knew of such hypothesis earlier [Sau01].

Conjecture 1.4.7 (Hedetniemi [Hed66])**.** Let $G = H \times K$, and $n = \chi(G)$. Then H is n-colourable or K is n-colourable.

The conjecture is also known as Hedetniemi-Lovasz Conjecture [Z^+98].

It has been proven for when the chromatic number of H or K does not exceed 3 [Bod15]. In particular, it is claimed that the case when $H \times K$ is 2-colourable is trivial, and is proven for 3-colourable graphs [Z^+98, Bod15, EZS85].

Example 1.4.8. Consider $G = H \times K$, where $H = K_3$ and $K = K_4$, as shown on Figure 1.4.3. The left and the right graphs on the picture are, respectively, H and K. Then $G = H \times K$ can be coloured as shown on Figure 1.4.4. In this case, nodes are not connected iff they have the same letter or number, so, trivially, the smaller of the two coordinates indicates the best colouring.

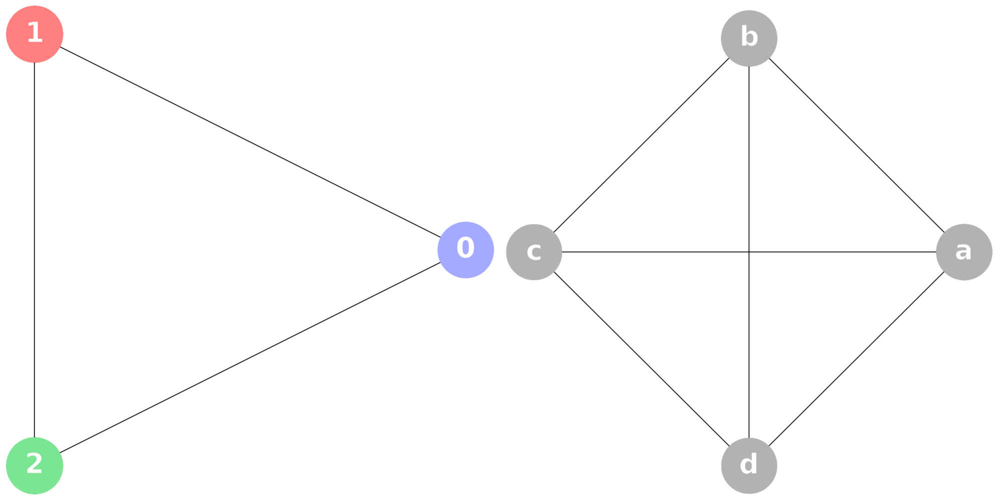

Figure 1.4.3: Example 1.4.8: Graphs H and K.

The following sections will aim to improve and generalize these facts and the conjecture in the light of graph homomorphisms.

1.5 Graph homomorphism

In a way that a group homomorphism preserves an operation between two groups, a graph homomorphism preserves the relation between the elements, or edges of a graph.

This section will define and explain graph homomorphisms and establish some basic principles and identities.

Hence, the definition:

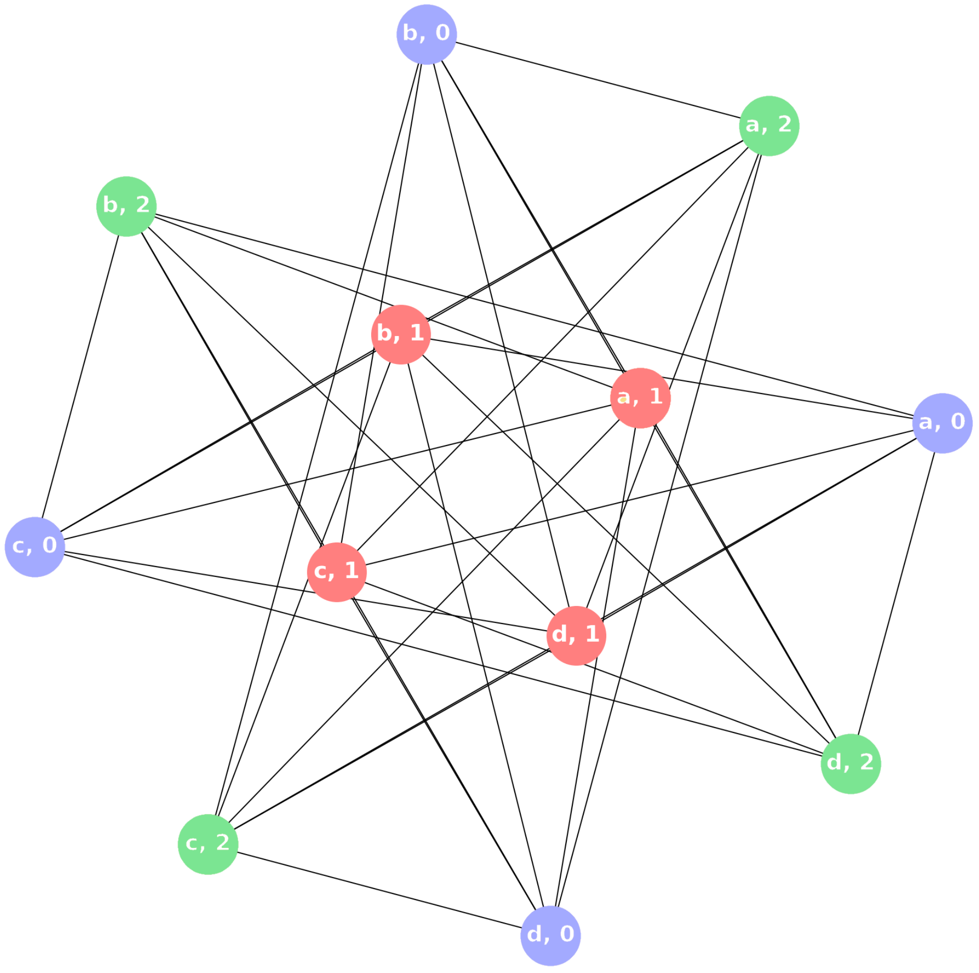

Figure 1.4.4: Example 1.4.8: Minimal colouring for $G = H \times K$.

Definition 1.5.1 ([Cam06])**.** Let G and H be two graphs. Then G is *homomorphic* to H, denoted as $G \to H$, if $\exists \phi : G \to H : \forall u, v \in V(G) : (u, v) \in E(G) \implies (\phi(u), \phi(v)) \in E(H)$. ϕ is then called a *homomorphism* from G to H.

In a way, it is a generalization of graph colouring. If $G \to H$, it means that G is $n = |V(H)|$ colourable, with the colors corresponding to the elements of $V(H)$, but two adjacent vertices in G are allowed to only be coloured with adjacent colours in H. This way, essentially, H as a graph defines which colours are allowed to be next to each other, as opposed to permitting any two colors to be adjacent.

Proposition 1.5.2. Let G be a graph and n be an integer. Then G is n-colourable $\iff$ $G \to K_n$.

Proof. ($\Rightarrow$) Let G be n-colourable, and let H be a complete graph K_n, such that $V(H) = \mathbb{Z}_n$. Then there exists a mapping $\phi : V(G) \to \mathbb{Z}_n$, such that for any $a, b \in G$, $(a, b) \in$

$E(G) \implies \phi(a) \neq \phi(b)$. Let $c = \phi(a)$ and $d = \phi(d)$. Then $(c, d) \notin E(H) \iff c = d$ by definition of a complete graph. Therefore, $(a, b) \in E(G) \implies (\phi(a), \phi(b)) \in E(H)$. Hence, ϕ is a homomorphism from G to H.

($\Leftarrow$) Let G be a graph, and H be a complete graph K_n, such that $V(H) = \mathbb{Z}_n$. Since $G \to H$, there exists a mapping $\phi : G \to H$, such that for any two connected vertices $a, b \in G$, $(\phi(a), \phi(b)) \in E(H)$. Since H is a complete graph, $(\phi(a), \phi(b)) \in E(H) \iff \phi(a) \neq \phi(b)$. Therefore, for any $(a, b) \in E(G)$, $\phi(a) \neq \phi(b)$. By definition, ϕ is an n-colouring, so G is n-colourable.

Hence, G is n-colourable $\iff G \to K_n$. □

All graphs that will be considered with respect to homomorphism will be loop-less. The reason for it is given below:

Theorem 1.5.3. Let G be a graph, and let $H = (\{a\}, \{(a, a)\})$. Then $G \to H$.

Proof. Mapping from every node of G to a preserves all edges, hence it is a valid homomorphism. □

Remark. Any graph with a loop will map loops to loops, so it can not map to a loop-less graph.

Example 1.5.4. Consider the following example of a graph homomorphism.

On Figure 1.5.1, there is a left graph, and a right graph. Denote left graph as G and right graph as H. In this example a homomorphism will be established between G and H. It is indifferent which node to colour the first node with, since all nodes in H are symmetric. Let the first node be picked and coloured as shown on Figure 1.5.2. The neighbouring nodes are colored with a subset of colors that are connected to the cyan node in H, as shown on Figure 1.5.3. Then the two other nodes can be deduced as shown on Figure 1.5.4. It can be verified that the mapping on that picture is a valid homomorphism from G to H. Hence, $G \to H$.

In the above example, each color is used exactly once, which does not have to always be so: a color might be used several times or not used at all. Below is a slightly more complicated case:

Example 1.5.5. Consider the following colouring of a bigger graph with a smaller graph, as shown on Figure 1.5.5. Just like in Example 1.5.4, a vertex is picked and coloured with some node in H, as shown on Figure 1.5.6. This is an assumption, which may or may not produce a valid homomorphism. Then, map neighbours of that node in G to a subset of neighbours of the colour H, as shown on Figure 1.5.7. It can then be deduced that the next node with no colour, as shown on Figure 1.5.8, can only be mapped to green or blue. For the connecting colours to this node are pink and teal and the only nodes that connect

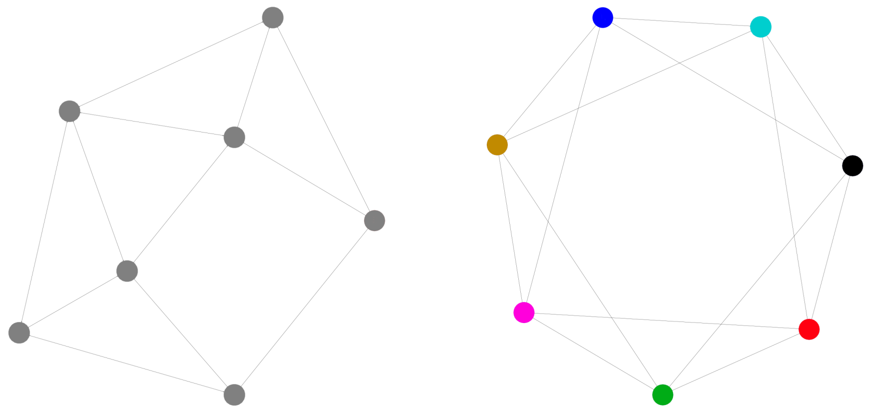
Figure 1.5.1: Example 1.5.4: Colouring the G with H.

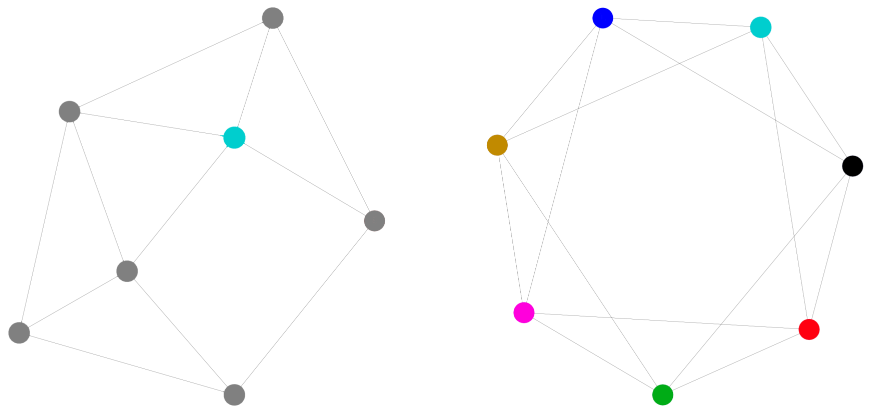
Figure 1.5.2: Example 1.5.4: Picking a node to start with.

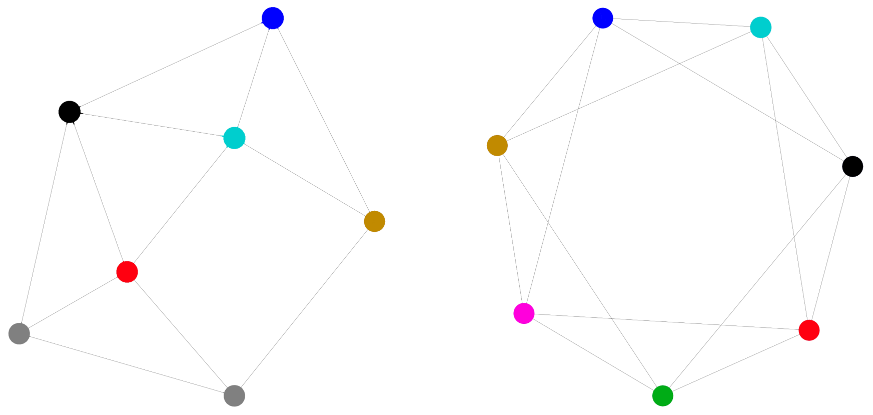
Figure 1.5.3: Example 1.5.4: Mapping neighbours in G to neighbours in H.

these two colours in H, are green and blue. In this example, green was selected. It can be deduced that the remaining node can be coloured in blue, as shown on Figure 1.5.9, and

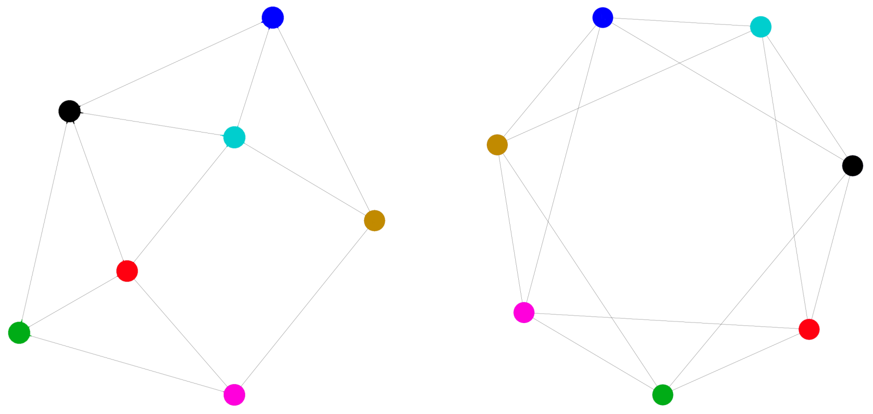

Figure 1.5.4: Example 1.5.4: valid homomorphism from G to H.

then verified that the diagram shows a valid homomorphism from G to H, hence $G \to H$.

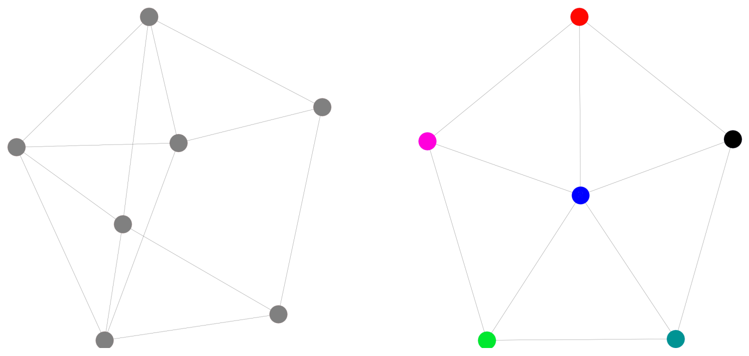

Figure 1.5.5: Example 1.5.5: colouring a bigger graph with a smaller graph.

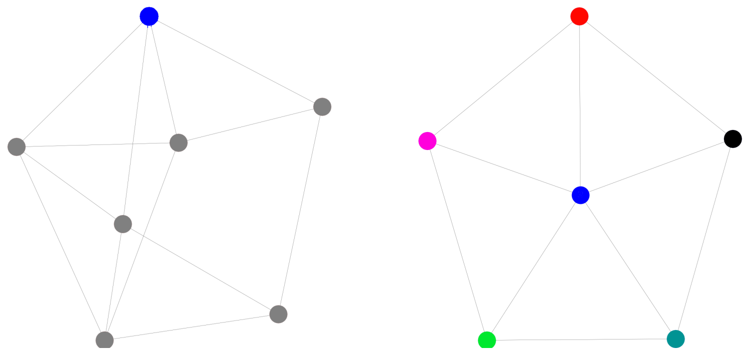

Figure 1.5.6: Example 1.5.5: Picking a vertex.

Theorem 1.5.6. Let G be a graph and S be a (node-/edge-)induced subgraph of G. Then $S \to G$.

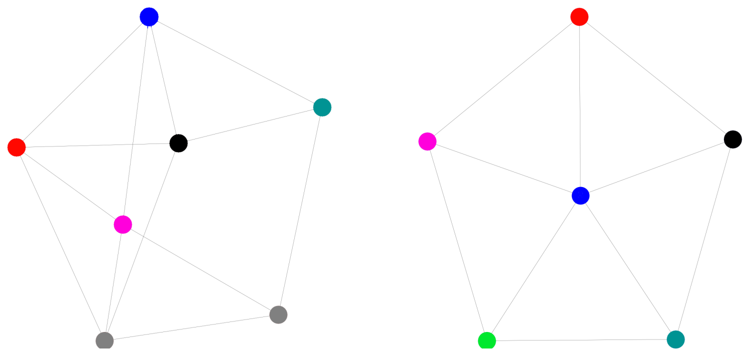
Figure 1.5.7: Example 1.5.5: Mapping neighbours in G to neighbours in H.

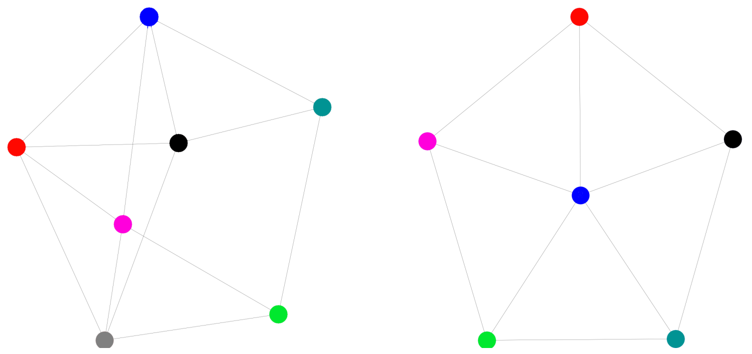
Figure 1.5.8: Example 1.5.5: Deducing next colour: either, green or pink.

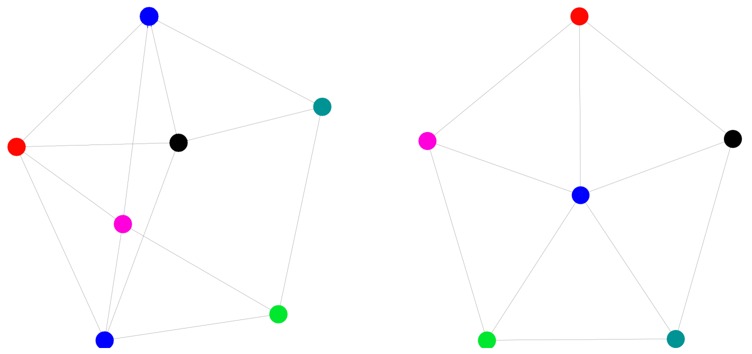
Figure 1.5.9: Example 1.5.5: Re-using blue.

Proof. Let $\phi : S \to G : \phi(a) = a$. Then ϕ is trivially a homomorphism, since $(a, b) \in E(S) \implies (a, b) \in E(G)$. $\square$

The material below requires that the notions of connectedness and disconnectedness are specified unambiguously. While many of the results can be generalized to directed graphs, from this point all considered graphs will be undirected, unless stated otherwise [Gra14].

Proposition 1.5.7. Let G, H be undirected graphs such that $G \to H$, and ϕ be a homomorphism. Then let $X_1 \dots X_n$ be connected components of G, and $Y_1 \dots Y_m$ be connected components of H. Then ϕ will map any X_i to some Y_j.

Proof. In this proof, write $a \curvearrowright b$ to denote that a reaches b.

Then let $A = X_i$. Take arbitrarily $a, b \in A$. Let $\phi(a) \in Y_j$. Then by definition $(a,b) \in E(G) \implies (\phi(a), \phi(b)) \in E(H) \implies \phi(a) \curvearrowright \phi(b)$. Therefore, $\phi(b) \in Y_j$.

Consider a path in X_i, such that the first node maps to Y_j with ϕ. Then ϕ maps the second node to Y_j, and thence the third node, etc. This way, $a \curvearrowright b \implies \phi(a) \curvearrowright \phi(b)$. a reaches every node in X_i, therefore it maps each of them to Y_j. Because reachability is an equivalence relation on undirected graphs, $\phi(X_i) \in Y_i$.

Hence, any homomorphism between any two graphs maps connected components to connected components. □

Similar to groups and lattices, graph isomorphism can be defined from homomorphism. Below is an alternative definition:

Definition 1.5.8. Let $G \to H$ such $\exists \phi : \phi$ is a bijective (in terms of edge-sets) homomorphism from G to H. Then ϕ is an isomorphism.

Theorem 1.5.9. Definitions 1.3.1, 1.5.8 are equivalent for simple graphs.

Proof. It is trivial that in both cases a map bijectively maps all edges in G to all edges in H. □

It was briefly mentioned in Sectioned 1.3, that isomorphisms are a special case of homomorphic equivalence. The latter is defined as follows:

Definition 1.5.10. Graphs G and H are *homomorphically equivalent* if $G \to H$ and $H \to G$. This will be denoted as $G \longleftrightarrow H$.

Remark. If a graph is homomorphic to its induced subgraph, it is homomorphically equivalent to it, since by Theorem 1.5.6, there is a homomorphism in the other direction.

The difference between homomorphic equivalence and isomorphism is that homomorphically equivalent graphs will contain similar induced subgraphs that they are both equivalent to. An example of such graphs is shown below:

Example 1.5.11. Let G and H be two connected bipartite graphs, such that $|G| > 1$ and $|H| > 1$. Then $G \longleftrightarrow H$.

Proof of Example 1.5.11. Since G, H are bipartite, by Proposition 1.5.2, $G \to K_2$ and $H \to K_2$.

Let (a, b) be an edge in G. let X be a node-induced subgraph of G on nodes a, b. Since all permitted edges are in X, X is a complete graph K_2. By Theorem 1.5.6, $X \to G$, since X is a node-induced subgraph of G. Therefore, $K_2 \to G$. Similarly, $K_2 \to H$.

Hence, $G \to K_2 \to H$, and $H \to K_2 \to G$. An example of such is shown on Figure 1.5.10. It will be shown later, that homomorphic equivalence is transitive (Theorem 1.5.12), therefore, $G \to H$ and $H \to G$, or $G \longleftrightarrow H$. □

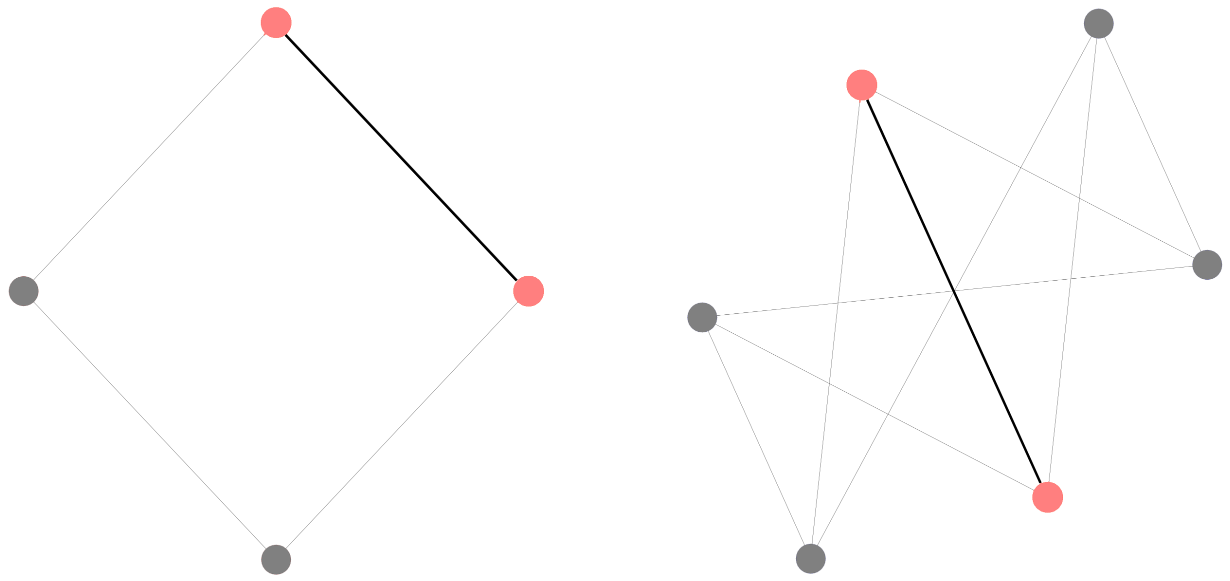

Figure 1.5.10: Example 1.5.11: Two homomorphically equivalent bipartite graphs.

It turns out, that graph homomorphism as a binary relation on graphs has some fascinating properties:

Theorem 1.5.12. Graph homomorphism is a preorder relation on graphs.

Proof. Any graph is homomorphic to itself under the identity map, hence graph homomorphism is reflexive.

Let G, H, K be graphs such that $G \to H$, $H \to K$, with ϕ and ψ homomorphism functions respectively. Then $\forall a, b \in V(G) : (a, b) \in E(G) \implies (\phi(a), \phi(b)) \in E(H) \implies (\psi(\phi(a)), \psi(\phi(b))) \in E(K) \implies \phi\psi$ is a homomorphism from G to K. Hence graph homomorphism is transitive.

Therefore graph homomorphism defines preorder on finite graphs. □

Example 1.5.13. The preorder on finite connected simple graphs is shown on Figure 1.5.11. Complete graphs are coloured in green, paths are coloured in grey, cycles are coloured in yellow, and other graphs of orders 5, 6 and 7 are coloured in cyan, blue and pink respectively. Black arrows are one-directional, and pink arrows are bi-directional.

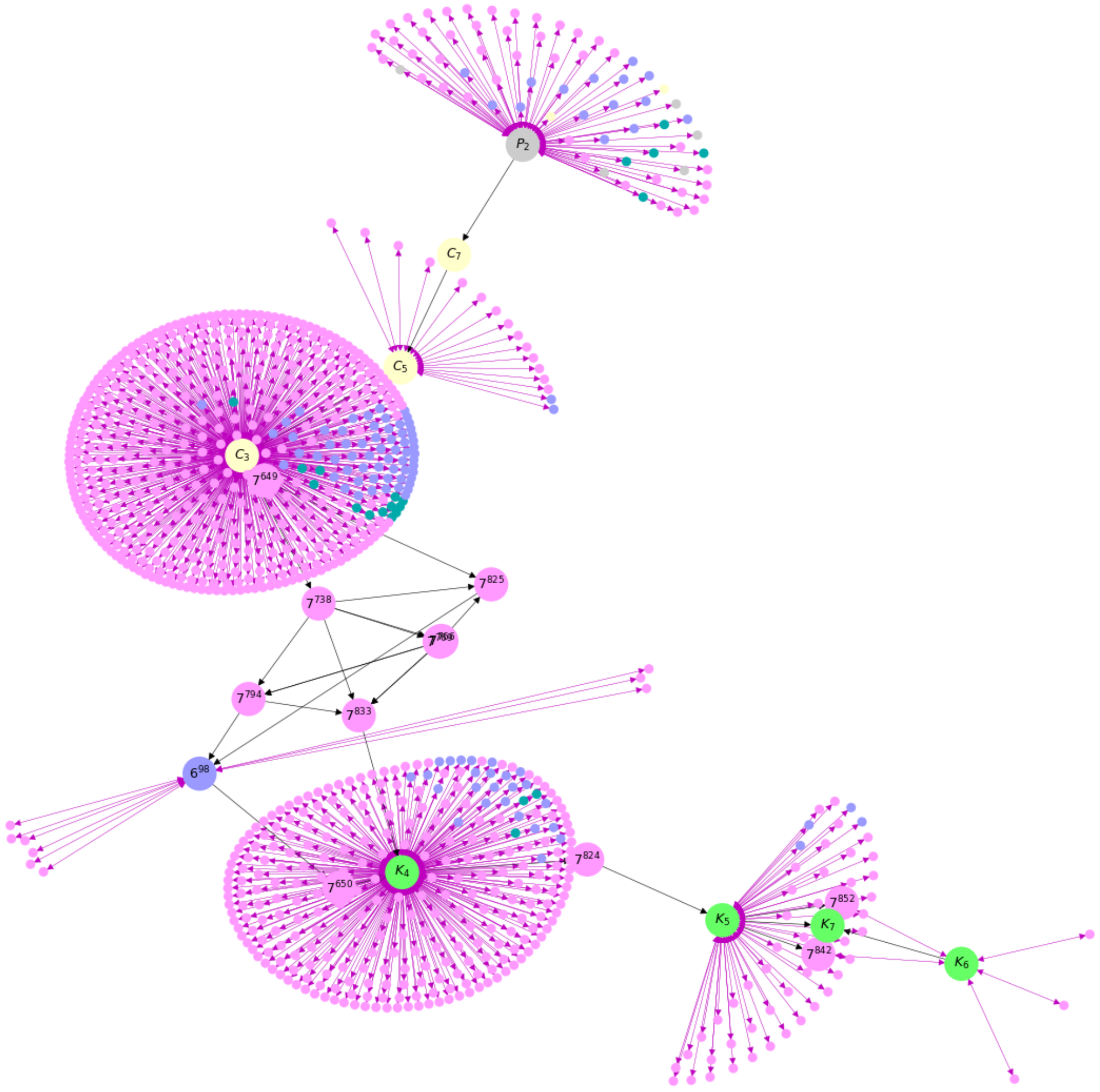

Figure 1.5.11: Preorder on connected graphs of size up to 7 under homomorphism.

It can be seen from Example 1.5.13, that graphs “cluster” together, and these “clusters” form strongly connected components of the graph on the diagram. These happen to be equivalence classes, which is implied by the preorder relation, since it is not antisymmetric.

Corollary 1.5.14. Homomorphic equivalence of graphs defines an equivalence relation.

Proof. By Theorem 1.5.12, graph homomorpisms are a preorder on graphs. Let $\rightarrow$ relation be then denoted as $\preceq$. By Definition 1.5.10, $G \longleftrightarrow H \iff G \preceq H, H \preceq G.$ since homomorphism is a preorder, homomorphic equivalence defines an equivalence relation. $\square$

Therefore finite graphs can be partitioned into homomorphism (equivalence) classes, which are exactly what the connected components are on Figure 1.5.11. Within each class all graphs are homomorphically equivalent to each other.

Since it is established that homomorphic relation is a preorder, the notation of precedence and homomorphism will be interchangeable:

Notation. Since $\to$ is a preorder, it can be denoted as $\preceq$, or $\prec$ in case it is a strict precedence.

Extending connections to group theory from Section 1.3, a union of finite subsets of homomorphism classes are a subgroup of some finite subset of graphs. For each graph can be represented by a finite set of its automorphism groups which belong to its isomorphism class. The operation of such subgroup would be a homomorphism, which includes both, re-labelling and an automorphism.

1.5.1 Products and homomorphisms

Similar to that in colouring, operations on graphs defined in Section 1.2 reserve certain properties.

Join product on graphs yields an analogous, but stronger property that is proven for chromatic numbers with Theorem 1.4.5

Theorem 1.5.15 ([hf])**.** *Let G, H, K be finite graphs such that $G + K \to H + K$. Then $G \to H$.*

Proof [hb]. This can be proven by induction.

Base: Let $|K| = 0$. Then, trivially, $G + K \to H + K$.

Assumption: The assumption is correct for any $|K| < n$.

Step: Let $|K| = n$. Let ϕ be a homomorphism from $G + K$ to $H + K$. Then let $G_K = \phi(G) \cap K$. If $|G_K| = 0$, then ϕ maps G entirely to H, hence $G \to H$. Assume $|G_K| > 0$. Then let $K_K = \phi(K) \cap K, K_H = \phi(K) \cap H$ and $G_H = \phi(G) \cap H$. Then by definition ϕ preserves edges between $\phi(G)$ and $\phi(K)$ in K, and since H and K are joined, so are K_H and K_K. Therefore, $\phi(G+K) \cap K = \phi(G_K) \cup \phi(K_K) = \phi(G_K + K_K) = K_H + K_K$, which means that $G_K + K_K \to K_H + K_K$. Since $|G_K| > 0$ and therefore $|K_K| < |K|$, it is implied by assumption that $G_K \to K_H$. Let ψ be a homomorphism from G_K to K_H. Then consider mapping $\lambda : G \to H$, so that given a vertex $v \in G$, $\lambda(v) = \begin{cases} \phi(v) & v \in G_H \\ \psi(\phi(v)) & v \in G_K \end{cases}$. Trivially, λ is a homomorphism, hence the step is proven.

Therefore, $G + K \to H + K \implies G \to K$. □

Remark. The similar property on chromatic numbers, shown by Theorem 1.4.5, would be that $\chi(G+K) \leq \chi(H+K) \implies \chi(G) \leq \chi(H)$, and Theorem 1.5.15 generalizes that notion.

Direct products also have analogous properties, which provide a weaker, but proven identity than that of Hedetniemi Conjecture 1.4.7.

Theorem 1.5.16. Let $G = H \times K$. Then $G \to H$ and $G \to K$.

Proof. Let $(a,m),(b,n)$ be arbitrary nodes in G, so that a,b are arbitrary in H and m,n are arbitrary in K. If $((a,m),(b,n)) \in E(G)$, then $(a,b) \in E(H)$ and $(m,n) \in E(K)$ by definition.

Let $\phi : G \to H$ be a mapping such that $\phi((a,m)) = a$. Then for any connected node (b,n), $(\phi((a,m)), \phi((b,n))) = (a,b) \in E(H)$. Hence, ϕ is a homomorphism from G to H. By symmetry of deduction, it is also true that $G \to K$.

Hence $G = H \times K \implies G \to H, G \to K$. □

Remark ([Bod15])**.** By Theorem 1.5.2, Hedetniemi Conjecture 1.4.7 can be formulated as follows:

$G = (H \times K) \to K_n \implies H \to K_n$ or $K \to K_n$.

This is a stronger assumption than Theorem 1.5.16, which implies that $\chi(G) \leq \min(\chi(H), \chi(K))$.

It turns out that the direct product of any two graphs is maximal with that property, which will be shown in Theorem 1.5.27. The next section will introduce a special kind of graphs, which are called cores.

1.5.2 Core

Since unique representatives of homomorphism classes form a poset, the particular selection of them is irrelevant to the properties of a poset; but it could be used to determine the existence of classes. From a certain angle, this section will produce an alternative of canonization, described in Section 1.3, in the setting of homomorphism classes.

From this point on-wards, all mentioned graphs will be finite, unless otherwise stated.

Definition 1.5.17. Graph $\mathcal{C}$ is a *(homomorphism) core* if any homomorphism $\mathcal{C} \to \mathcal{C}$ is an isomorphism. In other words, $\nexists C$:

- $\mathcal{C}$ is a proper induced subgraph of C.
- $\mathcal{C} \longleftrightarrow C$.

It can be observed from Figure 1.5.11 that some graphs are drawn in a bigger circle. It turns out that these graphs are cores. Some examples of them are given below:

Example 1.5.18. The following graphs are cores:

- Any complete graph K_n is a core, since the chromatic number of any of its induced subgraphs is less than n.
- Any odd cycle is a core, since there it has no induced subgraph it could be homomorphic to: any such subgraph is either, disconnected or a path, both of which have smaller chromatic numbers.
- Graphs shown on Figure 1.5.12 [Wel82].

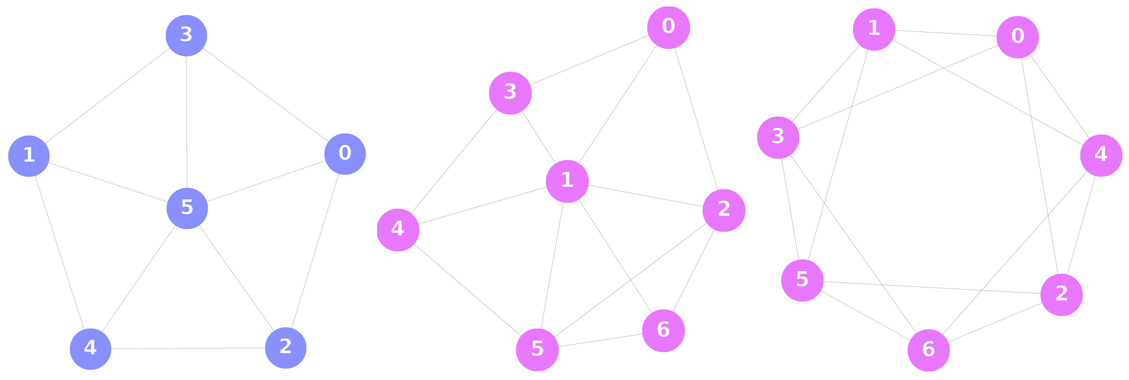

Figure 1.5.12: Cores on 6 and 7 vertices [Wel82].

Proposition 1.5.19. Any graph G contains a core $\mathcal{C}$ that it is equivalent to as its induced subgraph. $\mathcal{C}$ is then said to be a core of G.

Proof. Assume G has no proper induced subgraph that is its core. Then G is its own core by Definition 1.5.17. Otherwise, the core exists. □

This means, that any graph will have at least one core as its induced subgraph. In fact, the graphs on Figure 1.5.10 contain a core, which is colored in red, but there are other induced subgraphs, some of which are isomorphic to it.

Theorem 1.5.20. Let G be a graph, that is not a core. Then its core $\mathcal{C}$ will have less vertices than G.

Proof. Assume the opposite. By definition, $\mathcal{C}$ is an induced subgraph of G, hence there is an injective homomorphism from $\mathcal{C}$ to G. Since $|G| = |\mathcal{C}|$, the homomorphism is bijective. Since $G \neq \mathcal{C}$, there are more edges in G than in $\mathcal{C}$. Let ψ be a homomorphism from G to $\mathcal{C}$.

Then if ψ is injective, it is isomorphism. Contradiction. Hence any homomorphism will map some two vertices to the same vertex in $\mathcal{C}$. This means that some other vertex in $\mathcal{C}$ will not be mapped to, meaning that there is an induced subgraph of $\mathcal{C}$ that G is homomorphic to, but since it is an induced subgraph, this homomorphism is an bidirectional. Contradiction.

Hence G and $\mathcal{C}$ can not have the same number of vertices. □

Theorem 1.5.21. Two cores are equivalent iff they are isomorphic to each other.

Proof. In this context, two isomorphic graphs are considered the same, and, trivially, any graph is equivalent to itself. Therefore it is sufficient to prove in only one direction.

($\Rightarrow$) Let there be two equivalent cores. Then if one of them is equivalent to a proper induced subgraph of the other, then that subgraph will be equivalent to both graphs, hence a contradiction. Otherwise, they must have the same sizes and all homomorphisms from one to the other must map vertices injectively. Since vertices are mapped injectively, all edges are also mapped injectively, hence any homomorphism is bijective, i.e. an isomorphism.

Therefore, proof is complete. □

It is then possible to summarize the above facts as a logical connection between homomorphism classes and cores.

Corollary 1.5.22. Each of the following is true:

1. Every graph contains has a core as an induced subgraph, unique up to isomorphism.
2. Any homomorphism class has a core, unique up to isomorphism.
3. Core is exactly the smallest graph in any equivalence class.

Proof. The three facts can be proven one-by-one:

1. By Proposition 1.5.19, any graph has at least one core. If a graph G contains at least two distinct cores, they are equivalent to G, hence they are equivalent to each other. By Theorem 1.5.21, they are then isomorphic. Contradiction. Hence, any graph contains exactly one core as its induced subgraph
2. All graphs have exactly one core, therefore, two graphs are equivalent if their cores are equivalent. Therefore, within each equivalence class, all graphs have the same unique core.
3. It was shown that each class has a unique core. That core is an induced subgraph of each member of the class, as well as its own. Consider a graph in the class on the same number of vertices as the core. Then any homomorphism from it to the core will injectively map the vertices, and hence edges. Then it is isomorphic to the core, i.e. it is the core. Hence, the core has less vertices than any other graph in the class.

□

Remark. An interesting implication of this is that given a core, all graphs that can be obtained by adding and removing edges from it will belong to other equivalence classes.

Example 1.5.23. Consider the left-most (blue) graph from Figure 1.5.12, and add an edge between 0 and 1. Then the obtained graph will contain a clique on vertices $0, 1, 3, 5$, and it will be equivalent to it, since 4 can be mapped to 0 and 2 can be mapped to 3.

Summarizing the above, there is a way to unambiguously define a canonization-like function for homomorphism classes, which will map each class to the smallest element in it, called a core. Cores are contained in every graph of the class as induced subgraphs, and have strictly less vertices than any other homomorphically equivalent to it graph.

The next section will focus on cores rather than graphs, with respect to the homomorphic relation between them.

1.5.3 Order on cores

It was established with Theorem 1.5.12, that graph homomorphisms are a preorder relation on graphs, and that homomorphic equivalence is a relation of equivalence. The classes forming this relation can be represented uniquely by a core. The question of the nature of binary relations on cores is very interesting, and giving answers to that question is a vast area of research, as well as theoretical theoretical emphasis in this dissertation.

For starters, it is straightforward that cores form a partially ordered set. Some cores are homomorphic to some others, but some are incomparable.

Proposition 1.5.24. Cores are a poset under homomorphic relation.

Proof. By Corollary 1.5.22, cores represent each equivalence class exactly once, so the preorder on graphs is also antisymmetric on cores. Hence, it is a partial order. □

Example 1.5.25. Hasse diagrams for connected cores of sizes up to 6, 7 and 8 will be shown on Figures 3.0.1, 3.0.2 and 3.2.2 respectively.

Notation. Let G be a graph. Denote core of G as $[G]$.

The result from Proposition 1.5.24 can be extended even further. It turns out that a join operation can be defined on that poset, with the join of two graphs being homomorphically equivalent to their graph union:

Theorem 1.5.26 ([Bod15])**.** Poset of cores is a join-semilattice, where $G \wedge H = [G \cup H]$.

Proof. Let G and H be graphs. Then Let K be a graph such that $G \to K$ and $H \to K$ with homomorphisms α, β respectively. Then let ϕ be a mapping from $G \cup H$ to K, such that given a vertex v, $\phi(v) = \begin{cases} \alpha(v) & v \in G \\ \beta(v) & v \in H \end{cases}$. Then all edges in G get mapped to edges in K and all edges in H get mapped to edges in K, and since there are no other edges in $G \cup H$, ϕ is a valid homomorphism.

Since G and H are node-induced subgraphs of $G \cup H$, by Theorem 1.5.6, G and H are homomorphic to $G \cup H$.

Therefore, $[G \cup H]$ is a join. □

It also happens that a meet operation can also be defined on the poset of cores. A meet of two cores is the core of a direct product of them:

Theorem 1.5.27 ([Bod15])**.** Poset of cores is a meet-semilattice, where $G \vee H = [G \times H]$.

Proof. Let G, H, K be cores such that $K \to G, K \to H$, with two homomorphisms α, β respectively. Then let $\phi : K \to (G \times H)$ be a mapping such that given a vertex $v \in K$, $\phi(v) = (\alpha(v), \beta(v))$. Let u be such that $(u, v) \in E(K)$. Then by Definition 1.2.12 $(\phi(u), \phi(v)) = ((\alpha(u), \beta(u)), (\alpha(v), \beta(v)))$ is an edge in $G \times H$ iff $(\alpha(u), \alpha(v)) \in E(G)$ and $(\beta(u), \beta(v)) \in E(H)$. By declaration, α and β preserve edges from K to G and H, hence ϕ preserves edges from K to $G \times H$. Therefore by ϕ is a homomorphism from K to $G \times H$.

Hence, any graph that precedes G and H is also homomorphic to $G \times H$. Therefore, $[G \times H]$ is a meet of G and H. □

It is therefore straight forward, that under those operations cores form a lattice:

Corollary 1.5.28. Poset on cores is a lattice.

Proof. Since the poset of cores is a meet-semilattice by Theorem 1.5.27 and join-semilattice by Theorem 1.5.26, it is a lattice. □

Recall. By Theorem 1.1.6, the homomorphic relation on cores implies associativity, commutativity, idemponence and absorbtion properties of join and meet operation.

An important fact about disconnected cores, which extends Proposition 1.5.7, is that it is a graph union of connected cores.

Theorem 1.5.29. Let G be a core. Then its connected components are cores that are pairwise incomparable.

Proof. Let G be a core, and its component C be a non-core. by Theorem 1.5.7, connected components homomorphically map to connected components. Hence any such map ϕ from

G to G will map components to components. Then by Corollary 1.5.22, C contains a core. Let ϕ then map each connected components onto itself, and C to its core. Then ϕ is a homomorphism to a proper induced subgraph. Hence G is not a core. Contradiction.

Therefore, all components are cores. Assume there is a component A and component B such that $A \to B$. Then let ϕ be a homomorphism from G to itself that maps every component to itself, except for it maps A to B. Then ϕ maps G to $G \setminus A$, hence it is a homomorphism to a proper induced subgraph. Then G is not a core. Contradiction.

Let G then consist of connected components that are cores that are pairwise incomparable. By Proposition 1.5.7, any homomorphism from G to itself will map a component, which is a core, to a component, which is also a core. Since components are pairwise incomparable, such homomorphism can only map components to themselves. Hence, it is bijective, and G is a core by definition.

Hence it is proven that any (disconnected) core consists of pairwise incomparable connected components which are also cores. □

The join-irreducible elements and meet-irreducible elements are specific families of cores, which will now be introduced. The first of them is the connected cores:

Lemma 1.5.30 ([HN04])**.** The join-irreducible elements of the lattice of cores are exactly the connected cores.

Proof. Let L be a lattice of cores, as shown by Corollary 1.5.28.

($\Rightarrow$) Let $G, H, K \in L$ be such that $G = H \vee K \implies G = H$ or $G = K$. By Theorem 1.5.26, $G \approx [H \cup K]$. If $G = H$, then by Proposition 1.5.7, K as a component of $H \cup K$ maps to H, therefore $K \to H$. Similarly, if $G = K$, then $H \to K$. Assume that G is disconnected. Let A then be its connected component, and $B = G \setminus A$. By Lemma 1.5.29, connected components of G are pairwise incomparable cores. Hence A is a core, and B is a core by Lemma 1.5.29. Then by Theorem 1.5.26, $G = A \vee B$, but $G \neq A$ and $G \neq B$. Contradiction.

($\Leftarrow$) Let G be a connected core, and H, K be such that $G = H \vee K$. Then by Theorem 1.5.26, $G = [H \cup K]$. By Proposition 1.5.7, since G is connected, $H \to K$ or $K \to H$. Hence, trivially, $G = H$ or $G = K$.

Therefore, connected cores are exactly the join-irreducible elements of the lattice. □

Recall. Recall from proof of Theorem 1.1.24, the results of which extend to non-finite lattices, that any element can be reconstructed of the set of its join-irreducible elements. In lattice of cores, the connected components from Theorem 1.5.29 represent subsets of smaller elements, the join of which uniquely determines a core.

The dual of join-irreducibles - meet-irreducible elements of the lattice - are a special family of graphs that would require a separate definition:

Definition 1.5.31 ([DS96])**.** Let G be a graph, such that for any $H, K : H \times K \to G$, $H \to G$ or $K \to G$. Then G is called *multiplicative.* If G is also a core, it is called *multiplicative core.*

This definition essentially repeats the definition of a meet-prime element of a lattice, except that it uses direct product instead of the meet operation.

Lemma 1.5.32 ([HN04])**.** The meet-irreducible elements of the lattice of cores are exactly the multiplicative cores.

Proof. Let L be a lattice of cores, as shown by Corollary 1.5.28.

($\Rightarrow$) Let G be a meet-irreducible element. Then for any H, K: $H \wedge K = [H \times K] = G$, it has to be that $G = H$ or $G = K$. Since $[H \times K] = G$, it is trivial that $H \times K \to G \implies G \to H$ or $G \to K$. Hence, G is multiplicative by definition.

($\Leftarrow$) Let M be a multiplicative graph, and let $G = [M]$. Trivially, G is a multiplicative core. Let H, K be such graphs that $H \wedge K = [H \times K] = G$. Then, $H \times K \to G$. Since G is multiplicative, $G \to H$ or $G \to K$. Therefore, $G = H$ or $G = K$. Hence, G is meet-irreducible.

Thus, meet-irreducible elements are exactly the multiplicative cores. □

Remark. As noted earlier, Lemma 1.5.32 proves to only be meet-irreducible, which is a stronger condition, as shown by Proposition 1.1.9. This will become clear later.

The lattice of cores is non-finite, so it is useful to know if it is bounded below or above.

Theorem 1.5.33. Let L be the lattice of cores. Then $0_L = [(\{a\}, \varnothing)]$.

Proof. Clearly, such graph is homomorphic to any graph, since there are no edges to preserve with the mapping. □

Theorem 1.5.34. The lattice of cores is not bounded from above.

Proof. Assume there exists 1_L, such that 1_L is the top of the lattice. Since 1_L is finite, let $n = \chi(1_L)$. But then $1_L \to K_{n+1}$, hence $1_L \vee K_{n+1} = [1_K \cup K_{n+1}] \neq 1_L$. Contradiction.

Therefore, the lattice is not bounded from above. □

It turns out that the lattice of cores abides by a very strong condition:

Theorem 1.5.35 ([Wel82, Gra14])**.** The lattice on cores is distributive.

Proof. Distributed lattice is a lattice where $\wedge$ and $\vee$ operations distribute over each other. By Theorem 1.1.18, it is sufficient to prove that for any G, H, K: $G \wedge (H \vee K) = (G \wedge H) \vee (G \wedge K)$. By Theorems 1.5.26, 1.5.27, the statement can be simplified as follows:

$$\begin{aligned} G \wedge (H \vee K) &= (G \wedge H) \vee (G \wedge K) \\ [G \times [H \cup K]] &= [[G \times H] \cup [G \times K]] \\ [G \times (H \cup K)] &= [(G \times H) \cup (G \times K)] \end{aligned}$$

Let $A = G \times (H \cup K)$. Then $V(A) = V(G) \times (V(H) \cup V(K)) = (V(G) \times V(H)) \cup (V(H) \times V(K))$. Consider $B = (G \times H) \cup (G \times K)$. Then $V(B) = V(A)$. Now, prove $E(A) = E(B)$. Let $a, b \in G$ and $c, d \in (H \cup K)$. Then by definition, $((a, c), (b, d)) \in E(A) \iff \Big((a, b) \in G$ and $(c, d) \in (H \cup K)\Big) \iff \Big((a, b) \in G$ and either, $(c, d) \in E(H)$ or $(c, d) \in E(K)\Big)$. Therefore, $E(A) = E(G \times (H \cup K)) = E((G \times H) \cup (G \times K)) = E(B)$. Thus, $A = B$, and therefore $[A] = [B]$.

Hence, lattice of cores is distributive. □

It was mentioned before (Section 1.1.2) that any distributed lattice is isomorphic to a lattice of sets. In Theorem 1.5.29, it was shown that any core is a set of connected cores, which are pairwise incomparable. Consequently, it was noted in Remark 1.5.3 that each such core represents a subset of its lower set that consists of all connected cores that are homomorphic to it. However, such sets are never finite, which will be shown later.

There are several implications of the lattice being distributive. For example, that primes and irreducibles are the same:

Corollary 1.5.36. Join prime and meet prime elements of the lattice of cores are exactly the connected and multiplicative cores respectively.

Proof. By Theorem 1.5.35, lattice of cores is distributive, and by Theorem 1.1.20, its join-primes are exactly the join-irreducibles, and its meet-primes are exactly the meet-irreducibles. By Lemmas 1.5.30, 1.5.32 those are exactly the connected and the multiplicative cores respectively. □

The result of Theorem 1.5.35 can be extended to that cores also form a Heyting algebra, which is a stronger condition, as shown on Figure 1.1.1 [Sau01]. Theorem 1.5.33 already shows that the lattice is bounded from below. This project, however, will not go into defining or proving it, as it lies outside of its focus.

It has also been shown that the lattice of cores is dense, which is why a lower set of any element is infinite. That is, between any cores G and H such that $G \prec H$ there exists a core K such that $G \prec K \prec K$. The proof of that fact is long and irrelevant to this dissertation, and can be found in 1982 paper by Welzl [Wel82]:

Theorem 1.5.37 ([Wel82])**.** Lattice of cores is dense (excluding the zero and K_2).

Example 1.5.38. An example of an infinite family of cores between K_2 and K_3 is the family of odd cycles. The chromatic number of any odd cycle is 3, and a bigger odd cycle will always be homomorphic to a smaller one, with $\phi : C_{2k+3} \to C_{2k+1}$ being $\phi(v) = \begin{cases} v & v \leq 2k+1 \\ 2k & v = 2k+2 \\ 2k+1 & v = 2k+3 \end{cases}.$

The above facts can help to re-formulate Hedetniemi Conjecture 1.4.7, through obtaining an even stronger bound.

Theorem 1.5.39. Let L be a lattice of cores, and a, b be integers such that $3 \leq a \leq b$. Then a set of graphs G such that $K_{a-1} \preceq G \preceq K_b$ is a sublattice of L, which is distributive, bounded below by K_{a-1} and above by K_b.

Proof. Consider two graphs G and H in the subset. Since $G \preceq K_b$ and $H \preceq K_b$, it follows that $G \vee H \leq K_b$. Assume G and H are not $(a-1)$-colorable. Then $K_{a-1} \prec G$ and $K_{a-1} \prec H$, thus $K_{a-1} \preceq G \wedge H$. Otherwise, if one of them is K_{a-1}, trivially, K_{a-1} is their meet.

Denote the sublattice as M. Since L is distributive, and M is its sublattice, it is distributive by Proposition 1.1.19. From above, K_{a-1} is the smallest element, and K_b is the largest element, hence the lower and upper bounds. Therefore, M is distributive and bounded. □

Remark. It is trivial that for any two graphs G and H, $G \wedge H = [G \times H]$ is in the sublattice described in Theorem 1.5.39, with $a = \min(\chi(G), \chi(H))$ and $b = \max(\chi(G), \chi(H))$. If, supposedly, K_{a-1} is meet-irreducible, then if $[G \times H] = K_{a-1}$ it would imply that $G = K_{a-1}$ or $H = K_{a-1}$, i.e. $a = a-1$. This way, an alternative statement of Hedetniemi Conjecture for simple graphs is that all complete simple graphs are meet-irreducible (meet-prime), or multiplicative cores.

It is not yet proven whether K_n are meet-irreducible for $n \geq 4$. One of the reasons for this is because the lattice of cores is dense (Theorem 1.5.37). Between any K_n and any core that it is homomorphic to, there is another core. If the conjecture is proved wrong, there would be two cores on n vertices, whose meet is K_{n-1}. Therefore, by Proposition 1.5.19, their direct product would contain $(n-1)$-clique, and from definition of direct product it follows that both graphs would contain the same clique as well.

This way, graph homomorphisms are a preorder relation on finite simple graphs, which in turn imposes two other relations: homomorphic equivalence and a partial order. Each equivalence class of homomorphic equivalence has a unique, minimal representative, called a homomorphism core. These cores form a partially ordered set, which is a distributive lattice under graph union and graph join. This lattice is dense and bounded from below. The join- irreducible elements in it are multiplicative and connected cores respectively. It is not yet known whether complete graphs, which separate parts of the lattice, are meet-reducible. In addition, any finite subset of each equivalence class defines a group under homomorphism, as a cross-product of all groups that are defined from its isomorphism classes. This way, cores define two algebraic structures under homomorphism, for one of which, the lattice, homomorphism is seen as a relation, and for the other, the group, is an operation.

This concludes the material that is directly related to graph homomorphisms. The next section will be a note on the complexity of graph homomorphisms, which are a part of another unsolved problem.

1.6 Complexity

The problem of finding graph homomorphisms plays in important role in study of complexity. One of important areas of study in this field is proving whether or not for a certain class of problems, being able to verify a solution in order of n^k number of steps implies that the solution can be found in the order of n^m steps, where k and m are fixed and n is a variable. The class of such problems, for which a solving algorithm that works in the number of steps that is order of a polynomial, is not known, are called NP-Complete and NP-Intermediate, depending on the kind of problem [Pap03].

Definition 1.6.1. A problem is said to be in P class, or just P, if there exists a machine for which there is an algorithm that solves it in polynomial order of steps.

Definition 1.6.2. A problem is said to be in NP class, or just NP, if there exists a machine for which there is an algorithm that non-deterministically [1] solves it in polynomial order of steps.

The following definition is useful to classify problems in P and NP:

Definition 1.6.3. A problem X is said to be NPH, or NP-Hard, if for any problem Y in NP the problem Z of "translating" or "transcoding" Y into X is P.

Some problems in NPH are known to not have polynomial solutions in normal setting. Those, for which it is not known, are called are called NP-Complete:

Definition 1.6.4. A problem is NP-Complete, or NPC, if it is NP-Hard and NP.

[1]The difference is that if a problem is solved non-deterministically, the knowledge of the computation decision tree obtained to solve a problem is required a-posteriori, as if all non-consecutive decisions were taken simultaneously.

An NP-Complete problem is a problem in NP for which doesn't exist a machine for which there is a known algorithm that solves the problem in general case in polynomial time [Pap03].

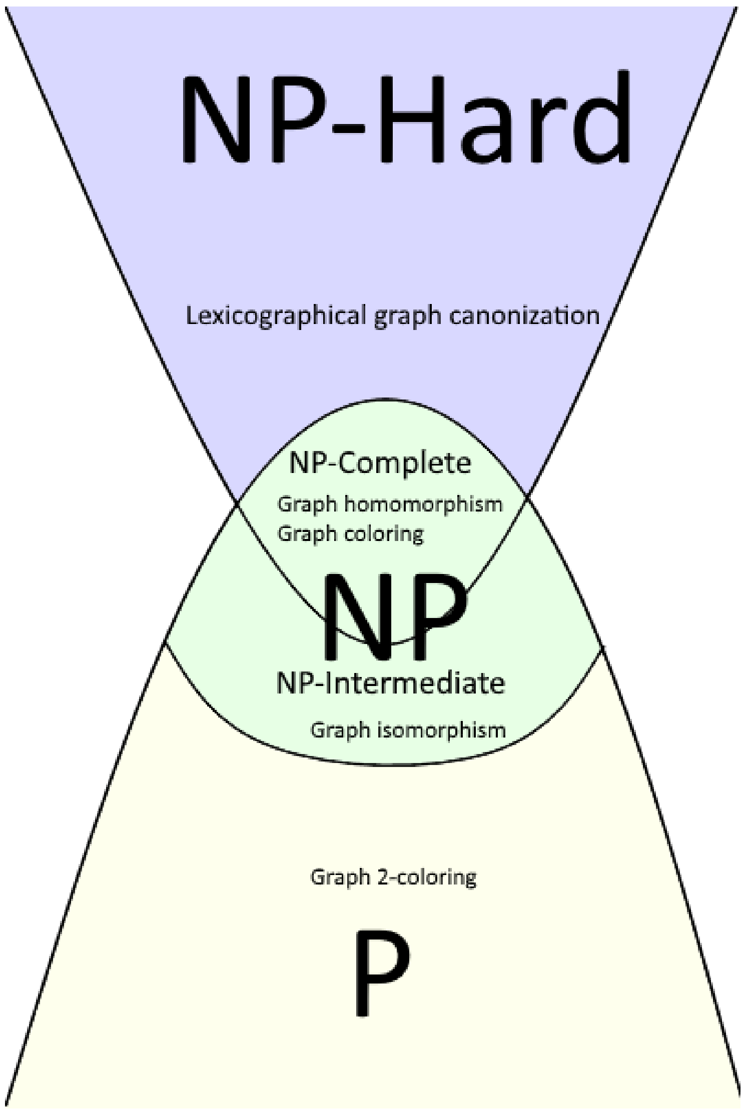

Figure 1.6.1: P vs NP diagram

It is conjectured that there is no algorithm that solves an NP-complete deterministically in polynomial time, which is a widely known unsolved problem in the study of complexity (Figure 1.6.1):

Conjecture 1.6.5 ([Pap03])**.** There are problems in NP which are not in P.

Example 1.6.6. Let a, b, c be symbols that represent either, *true* or *false*. Then let $Expr$ be an expression on these variables, using $\vee$ and $\wedge$ operators and the Problem($Expr = true$) be denoted as 3-SAT. It has been shown that 3-SAT, as well as n-SAT.

The NP-completeness of *SAT* and other problems has been researched by many scientists [Coo71, Bod15, Lev73, Kar72]. The 1971 paper by Cook states that any recognition procedure in theorem-proving is at least as hard as its translation into a SAT Problem [Coo71]. A paper in 1973 authored by Levin states that any brute-force problem of a certain kind, that he names "Universal", is of the same complexity type [Lev73]. Another research paper, authored by Karp and others, lists 21 combinatorial search problems that belong to NPC class [Kar72].

NP-Complete problems are not the only class of problems in NP for which a polynomial-time solution is still unknown. Some NP problems are conjectured to be neither, P nor NP-Complete. Such problems are called NP-Intermediate:

Definition 1.6.7 ([Zoo])**.** A problem is NP-Intermediate if it is NP, but neither P nor NP-Complete.

The following theorem shows that Conjecture 1.6.5 holds iff the class NPI of problems is empty:

Theorem 1.6.8 (Ladner's theorem [Lad75])**.** If $P \neq NP$, NPI is not empty.

If a problem is reducible to *SAT* problem in polynomial time, it is also NP-Complete. It occurs that the problem of some of the properties defined earlier fall into these classes:

Conjecture 1.6.9 ([Sch88, Bab16])**.** Graph isomorphism problem is NP-Intermediate.

It has recently been shown that the graph isomorphism problem can be solved in quasi-polynomial time [Bab16]. A similar problem, graph canonization, is at least as hard as graph isomorphism. It has not yet been proven if the two problems belong to the same time complexity class [ADK07]. A more extreme case of this problem is finding a canonical form, which is the smallest lexicographical order on graphs, with order being established by the binary string of the adjacency matrices of graphs:

Theorem 1.6.10 ([ADK08])**.** Graph canonization by lexicographic order of adjacency matrices is NP-Hard.

Graph colouring also happens to be a class of problems that are conjectured to be solvable in polynomial time, but the algorithm for which is not yet found:

Theorem 1.6.11 ([Pap03])**.** Graph colouring problem is NP-Complete.

Importantly, graph homomorphisms are "harder" than both, colouring or isomorphism, as these are special cases. Yet, the solution can be verified in polynomial time, meaning it is NP-Complete. The following theorem thus extrapolates Theorems 1.6.9, 1.6.11:

Theorem 1.6.12 (Hell–Nešetřil theorem [HN04])**.** Let H be a graph. Then the problem of H-colouring is P if H is bipartite and NP-Complete otherwise.

The current algorithms which solve NP-complete problems most often use a brute-force approach by recursively attempting all possible options. Such solvers often use heuristics which improve their performance by identifying when a recursive search in certain places is unnecessary or when a certain assumption is more likely to have a valid solution than another. This technique is often called *backtracking*, and will be covered in more detail in Section 2.2.1.

This concludes the introductory section of this dissertation. Section 2 will describe the software design for this project, which will include design and implementation of a solver, partial ordering program and visualization components. Section 3 will analyse the results produced by the software, as well as make observations and prove them right or wrong in a generalized setting.

2 Software

The software for this project was designed with the purpose of analysing graph homomorphisms algorithmically and visually. This section will give an introduction into the design objectives in Section 2.1 of the programs, design decisions and implementation details of them, including testing, profiling and demonstration.

2.1 Objective

The objective of the software design is to find the correct tools and approach to develop tools that would help to achieve a better understanding of how the lattice of finite cores is formed for small graphs. It is essential that the conclusions of this study are easily reproducible, and hence using the software to collect and visualize the data is a key component of this project. Analysis would be an important part of this investigation, so it is of high priority that the data visualization components highlight relevant information, and allow to view the results intuitively and seamless to interact with.

The software part of the project can therefore be split into three main parts: homomorphism solver, lattice builder and visualization, which visualizes all parts of this project. There are some other components of this project, which perform testing, profiling, debugging and interactive debugging routines. The further sections will focus on key details of their design and implementation decisions.

2.1.1 Implementation decisions

The framework and language choice is very uncomplicated and simple for this project. Since the objective is not the performance, but an ability to visualize, profile, and potentially integrate written code with other languages, there are advantages for using *Python* [S$^+$99]. The language has a variety of libraries, such as *networkx*, which is the a feature-relevant and well-documented framework for the purposes of this project, and *matplotlib*, which integrates with *networkx* and allows to visualize graph [Hun07, HSSC08]. There is an easy integration with *C/C++* languages through frameworks like *Boost Python* and *Cython*, and the language is easy to use from a bash script [AGKO03, BBC$^+$11, Bou78].

Integration with GAP has been primarily done through *bash*, as the script is central to system integration and GAP is only instrumental [G$^+$]. The scientific part of the project makes heavy use of *networkx* utilities, which have been tested separately and should not contain any significant defects.

2.2 Homomorphism solver

Though the solver is a crucial part of this project, it is not the main focus to obtain its ultimate performance by investigating techniques for improving speed of the algorithm. Rather, it is a central utility that has to produce the correct results and should not be the main performance bottleneck of other routines.

As shown by Theorem 1.6.12, graph homomorphism problem is NP-Complete, and as discussed in Section 1.6, one of the fastest ways to solve an NP-Complete problem is to use a backtracking approach to find whether there is a solution that satisfies the definition of graph homomorphism, or claim there is none otherwise. The algorithm design will, hence, be aiming to efficiently solve a constraint satisfaction problem by iterating its search tree using d-way approach. That is, solving a homomorphism between graph G and graph H,, the solver will pick a vertex in G, and attempt all possible colourings in H for that vertex before rolling back up the search tree. The order of iteration, aided by heuristics, as well as pruning sub-trees, can easily be done by modifying specific parts of the algorithm.

2.2.1 Design

The objective of a usable backtracking solver implementation is to make the problem solving time increase as flat as possible as the problem, and hence the search tree, grows exponentially. The choice of correct arc consistency and heuristics procedure are therefore the main focus of the solver's design. For that reason, it is a debate between an iterative and recursive approach, d-way and 2-way approach, arc consistency algorithm, whether to use back-jumping and which heuristics to use. These options will now be discussed. In scope of this section, G will mean a graph from which a homomorphism is being found, and H is the graph that the homomorphism is being found to. Variable is a node in G, and value, or colouring, is a node in H that the partial solution maps or assigns a variable it to.

d-way vs 2-way The difference between d-way and 2-way approach is how many sub-trees a solver iterates in each decision. A d-way solver typically picks an unused variable and iterates its possible values, until it finds a solution or goes a level up the tree. 2-way solver branches two ways: for taking a certain assumptions and rejecting it. In a trivial case, it can assume a value for a variable, and also assume otherwise, but in more complicated case it can force certain restriction on a solution. The trivial case would be very similar to the d-way solver, but allows for changing the order in which values are assigned to variables. It is important to note that a basic d-way solver is simpler to implement, as it is less abstract.

Arc consistency Arc consistency algorithm performs *pruning*, a procedure of removing a decision, or sub-tree, which can not have a solution. For example, when a node is coloured, its neighbors can not be coloured in conflicting colours, and can be pruned from subsequent search. Even further pruning is possible using algorithm families such as *MAC*, which stands for *Maintaining Arc Consistency* algorithm [LS04]. In general, for some problems extensive pruning significantly reduces the search tree, while for others it can add have a notable amount of time per decision and not reduce the problem accordingly.

Back-jumping Back-jumping is a technique with which a solver is able to undo assumptions in a different order than it made them. For example, it might assign variables $x_1 = 1, x_2 = 0, x_3 = 3$ and then decide that if x_2 is unassigned, a complete solution is possible. That is, a solver is capable of undoing assumptions as a choice, though it should not do so repeatedly. In that sense, the decision tree may become significantly higher, though is still polynomial in depth. Back-jumping is more convenient, for that and other reasons, to do iteratively, as the recursion might take too much time.

Recursive vs Iterative Recursive approach is simpler, than iterative, but there are some drawbacks, including back-jumping and performance.

Heuristics Heuristics does not affect the search tree itself, but affects the order in which it is iterated. It might indirectly influence arc consistency, which might change the number of steps taken to find out that there is no homomorphism, but other than that it is only instrumental for finding a solution if there is one. An example heuristic would be to choose to find a variable that has the most available colourings for it. This heuristic is called *SDF*, which stands for "Smallest domain first". Heuristics can be categorized into static and dynamic. Static heuristics are computed for more than one decision, while dynamic heuristics are specific to every decision.

For this project, due to Python runtime's inefficiency of calling functions, it was decided to use iterative solver, with d-way branching, as it is simpler and is sufficient. It was decided to not perform any extraneous pruning, other than forward-checking, which prunes according to the definition directly, as the number of conflicts between constraints grows, presumably, slowly for this problem. No back-jumping is performed, as it could be a secondary optimization, while the solver is sufficiently fast, but a careful choice of heuristics was made, and will be introduced further.

The model implementation therefore shown on Listing 2.2.1. There, the heuristics are preceded by a hash, indicating that it is optional for validity of the algorithm.

Listing 2.2.1: Iterative backtracking algorithm

```
function find_possible_map():
    value = current_value()
    while is_valid_option(value):
        value = value.next()
        if is_pruned(value):
            continue
        break
    return value

function find_optimal_colouring():
    return sorted(colourings[srcs[i]], lambda c: -rating(c))

function choose_best_node():
    return get_unused_variables()
        .filter_minimal(lambda x: rating(x))[0]

function find_homomorphism(g, h):
    while true:
        while i in [0..n - 1]:
            if action == FORWARD:
                # choose g-node
                srcs[i] = i # choose_best_node()
                # choose order in which h-colourings will be iterated
                # colourings[srcs[i]] = find_optimal_colouring()
                if is_last_option():
                    set_rollback()
            assert i >= -1
            if action == BACKTRACK and current_value() != UNDEFINED:
                # propagate value
                pass
            mapto = find_possible_map()
            if is_valid_option(mapto):
                forward_node(mapto)
            else:
                set_rollback()
        if i < 0:
            break
        assert is_valid_solution()
        return soln
        # i -= 1
        # action = BACKTRACK
    if i != -1:
        return soln
```

The heuristics used in this solver are described below:

Neighbors heuristic This variable ordering heuristic gives priority to the nodes that have more neighbors, since they are more constrained by definition of graph homomorphism, and should have a smaller domain. It is an adaptation of the *SDF* heuristic, described before, for graph homomorphisms. It is dynamic, as it is affected by the number of pruned values for a variable in a given state.

Colourings order heuristic This value-ordering heuristic reshuffles colourings of a node each time a colouring is reset, so that the solver iterates the ways to colour a specific node in different order. Such order is determined by the number of neighbors a specific colour has, and by some other criteria, such as cardinality in the node-induced subgraph of unassigned variables and prune count heuristic. This is a static heuristic, which is pre-computed each time a variable is selected.

Error heuristic This heuristic counts how many times the solver rolled back from colouring a specific node, and will prioritise taking the same option over others. This advises the solver to take more problematic nodes earlier in the search, thus increasing potential of the entire search, as the more "risky" nodes are left at the end. This is a variable ordering heuristic, and is dynamic.

Pruning count heuristic This heuristic counts how many times a specific colouring has been pruned. Similarly to the error heuristic, it influences the solver to apply most problematic colourings earlier in the search. This is a part of the value ordering heuristic, described above.

The performance of the solver could also have been consolidated by finding odd girth of the domain and image of a homomorphic relation, and finding automorphism groups in advance to avoid iterating symmetric solutions, but it would be more suitable for 2-way solver, and is unnecessary, as the implementation turned out to be sufficient.

2.2.2 Implementation

While use of a constraint satisfaction problem solver has been considered, it did not appear to be practically useful: an optimal specification would have required more time than implementation of a solver in an imperative programming language, and the integration would take too much time to start the solver. Another option would be to use a CSP module that can be used within the language, which in case of *Python* would not appear practical again, as the overhead of calling functions and loading modules would be too complex to alleviate without very good library, which has not been found. For that reason, the solver implementation started in *Python*, as integration with *networkx* allows to use its tested interface directly [S$^+$99, HSSC08]. However, the profiling information showed that CPython runtime is the major bottleneck of the algorithm, with *networkx.has_edge* being the main performance problem, so the implementation was adapted to *Cython*, a language that allows to write functions for both, C and interpreter levels of a Python program [BBC$^+$11]. This allowed for, majorly, reduction of function calling overhead, which is significant when the number of calls to a function grows exponentially to the size of the problem.

2.2.3 Profiling

The point of profiling is to analyse which functions take most of the time during run-time of the program. For example, the mentioned *networkx.has_ edge* was replaced with a C-layer subroutine, which looks up a cached adjacency matrix, stored in a one-dimensional bitset, with no global interpreter lock overhead. This improvement caused reduction of the number of instructions performed per call by a large constant factor [bh].

The profiling information, obtained from *pycallgraph* and *cProfile*, showed no accountable major bottleneck for larger problems, other than the number of calls performed by a solver [Kas16]. This only indicates that the heuristics and solver are not sufficiently effective for the problem, which for large enough problem would be an issue with any existing implementation.

2.2.4 Testing

The solver debugging was carried out manually using small graphs and *matplotlib* visualizations, as well as tested using *GAP* solver and on edge cases. The advantages of that decision is that the *GAP* solver is relatively fast and would also be a good tool to estimate relative performance of the *Cython* implementation. A shell script tests the solver using randomly generated connected graphs, and the lattice testing program checks that the program is correct on small graphs.

2.3 Data

It was decided to analyse poset of cores, since any non-core "behaves" like a core in terms of its structure, but is bigger in size, and therefore, for the purposes of this software is irrelevant. There is also no point in analysing isomorphic graphs, since in the scope of this analysis, isomorphic graphs are seen as similar.

The number of non-isomorphic graphs on n vertices can be described as combinatorial explosion, meaning it grows faster than exponentially [Slo14, McK]. The number of disconnected graphs on n vertices is by the order of magnitude higher than that of connected vertices. Moreover, since homomorphisms between disconnected graphs map connected components to connected components, it would make sense to only consider connected cores, i.e. join-irreducibles, which are also easier to find compared to meet-irreducibles. Recall that any core consists of connected cores from Theorem 1.5.29. For a similar reason, only simple graphs will be considered, since the number of directed graphs escalates too fast [Slo14, McK].

The data would therefore include small connected cores of sizes up to 9, of which there is almost 300000. For $n = 10$, the count would exceed 10^7, which induces serious input-output performance problems. Brendan McKay's datasets were utilized for the purposes of this study, as it contains all the necessary. It lists all graphs of size up to 10 with the desired properties [McK].

2.4 Lattice

The purpose of the lattice implementation is to organize given graphs in the order of homomorphism. A side objective is to be fast to load, since loading graph relations database can be a part of an extended solver. Hypothetically, if all finite graphs were given, the program would have had an algorithm to order them, and hence would construct a lattice. Due to density and bounds properties, however, it is only able to approximate the lattice of cores, but will normally not represent one.

2.4.1 Design

In terms of representation, Hasse diagram is a directed acyclic graph. By Theorem 1.5.20, it was established that all graphs that are equivalent to a core on the same number of vertices are isomorphic, and from Proposition 1.5.7, a core of a connected graph is a connected graph. Hence a valid Hasse diagram of cores can be generated providing small graphs as input in any order, and establishing if a simple graph introduced to the system is equivalent to the current smallest representative of equivalence classes, and if it is, the bigger of them based on the number of vertices is definitely not a core. In the meanwhile, since all existing graphs smaller than the introduced one should have already been added in the system, the new graph is definitely a homomorphism core. As a total of processing the provided input, the smallest of representatives are cores. Therefore the insertion algorithm will work correctly for the purposes of this project, given that the data suffices the aforementioned requirement.

From further experimentation it was learned that the number of homomorphism cores grows also exponentially with respect to n the number of vertices, i.e. $\{3, 5, 7, 17, 80, 1304, \dots\}$, yet it grows much slower than the number of graphs [Slo14], as it was important to avoid too many checks in the first place anyway, and as the graph becomes larger, it takes longer and longer to search, load and unload it. While the engine of this software does not perform any visualizations, it also meant that the emphasis had to be put on the relations of homomorphism cores, and visualization of the entire dataset is only feasible for a relatively small n, while visualization of the cores relation is yet possible for slightly larger n.

Another possible approach worth mentioning would be divide and conquer, that is, constructing multiple systems of representatives, and then merging them together. This will be further discussed as a part of implementation.

2.4.2 Implementation

This functionality was implemented in python using *networkx* package, as it offers a flexible functionality for working with directed graphs, and direct acyclic graphs in particular, which is what a Hasse diagram is. The lattice consists of the following components: lattice interface, path finder and cache.

Lattice interface is an entity responsible for merging homomorphism solver with the graph of the lattice. It defines several important functions, such as adding a new element, checking if two graphs are homomorphic using lattice graph, and construction. From the implementation point of view, it is mostly an interface to different parts of the program, such as the solver, the path finder and the cache management, and its main purpose is not to change throughout implementation process.

Lattice path finder is an entity responsible for finding and memoizing relations in a graph, and is an instrument for use by the interface. As well as a graph of the relations between cores, path finder also keeps a graph of non-existing relations between cores, meaning it memoizes when two representatives are non-homomorphic. In particular, given a new graph the program attempts to find homomorphisms in both directions and existing cores in the order of sizes of their equivalence classes. When a homomorphic relation is being established between some G and some H, the program calls path finder to deduce if it is possible to establish the relation without calling the solver. It does so by storing graphs of edges between representatives, and a graph of non-edges between them. When a new graph is added, until an equivalence is found, or not, it represents its own class. Therefore, a relation between G and H is being established, the path finder attempts to find either, a core that G has an edge to that has a path to H, which would imply that $G \to H$ by transitivity, or that there is some graph that H is homomorphic to but G is not, which would imply that $G \not\to H$. If no such deduction is possible, a solver is run and edge or non-edge is memoized, and when an equivalence is found, the new graph is removed from both path finder graphs. It replaces existing representative if it has less vertices or goes to the list of its equivalent graphs otherwise.

Lattice cache is an entity responsible for keeping all representatives in sync with the loading cache. Since the representatives are the only graphs that will be loaded multiple times in a scenario when new graphs are being inserted into the system, it is imperative to alleviate the bottleneck of reading and writing the same graphs repeatedly, which has been discovered through repeated profiling.

The above implementation proved sufficient to load graphs of size up to 9, of which there are about 300.000. It turns out that the number of graphs of size 10 is more than 11.000.000, and it would take too much time simply to decode them from $g6$ file format or to download in any other.

It was found that the vast majority (more than 95%) of the graphs belong to equivalence classes of K_3, K_4 or K_2. For that reason, the cores are sorted by the size of their homomorphism classes, and the program scales almost linearly for larger input, with exceptions for more complex graphs. Importantly, graph homomorphism solver has not been a bottleneck of the program at any point.

The aforementioned divide and conquer approach did not appear very useful, as the program scaling is not the major issue when attempting to find more graphs of size 10. The main issue, however, would be the system input/output, which would be too complex to alleviate, as though the program scales almost linearly to the size of the input, the size of the input itself grows faster than exponentially [Slo14].

2.4.3 Profiling

The profiling was performed in order to verify that no implementation detail is taking most of the run-time. The lattice interface was profiled using *cProfile*, and all major performance issues have been resolved. As a result, the profiling shows no specific time-prevalent function calls.

2.4.4 Testing

The interface was tested with a shell script that compared existing relations in the lattice with the output of the homomorphism solver. In case of an error the script would automatically check if the problem was in the solver or in the lattice. All tests have passed successfully.

The solver which uses a lattice has been tested together with the regular implemented homomorphism solver against solver in GAP Digraphs package [DBJM$^+$18, G$^+$]. All problems have been resolved and the automated testing reveals no errors.

2.5 Visualizations

Visualization is one of the key parts of this software, as it not only helps to analyse the results of the above components, but also complements error resolution. For that reason the functionality for plotting separate graphs, homomorphisms and lattices has been written alongside the development of those interfaces.

One of the most important parts of visualization is to choose which information is the most relevant to show, including both, the particular choice of data to illustrate and which properties of them to put emphasis on, as too much information will have the opposite effect by confusing the user.

It is important to mention that the central visualization part of this project is the Hasse diagram, in line with the problem. There have been multiple visualization prototypes, but only two were kept: an image generated using *matplotlib*, which is more suitable for the report, and an interactive visualization using *VivaGraph* [Hun07, Kas19].

2.5.1 Image-based visualization

This visualization is not as useful for viewing results as interactive visualization, but it can be well utilized for debugging and written presentation.

While it makes sense to visualize all graphs when there are relatively few of them, as shown on Figure 1.5.11, it is not very practical for bigger datasets. Besides, the diversity of graphs inside equivalence classes is not particularly of high importance, though the size of equivalence classes can be quite interesting. For that reason, this visualization will avoid drawing any graphs other than homomorphism cores, assuming the rest of the information to be unimportant.

This visualization uses different colours to indicate different types of graphs. For example, it uses green colour to show that the graph is complete, or yellow to indicate that the graph is a cycle. The graph labelling is also done in a way that highlights the important information, showing graph sizes and the ids of the graphs in the normal case or specific names otherwise. An example of effectiveness of this visualization method can be found on Figure 3.0.2, where it can be observed, for instance, that all odd-cycles are between two complete graphs on two and three vertices, with the number of vertices increasing towards K_2.

2.5.2 Interactive visualization

As can be judged from Figure 3.2.2, it would be quite difficult to see the core graph using image-based visualization for $n = 9$, as there would be too many objects. Another drawback would be an extra challenge of customizing a layout function to find the correct way to draw the diagram.

An important side objective of interactive visualization, compared to the image visualization, is to provide the user with a quick way to look up the graphs in the lattice. This can be achieved by substituting each disc of a node with an image.

Design The interactive visualization therefore needs to provide a convenient interface for navigating to and from specific areas of the lattice, and for looking up the elements of the lattice graph. It also has to be able to display as many elements of the core graph as necessary. It makes sense to make use of the intensity of the colours to signify the size of the equivalence class of a specific homomorphism core.

Implementation The above functionality was implemented using VivaGraphJs framework, as demonstrated on Figure 2.5.1 [Kas19]. The framework was chosen amongst many others, including, for example, d3, as its tutorial set was a better match for the project. The graphs are draggable and can be zoomed into and zoomed out of. The number next to the name of the graph indicates how big the equivalence class of that core is: that is, the count of elements which have been found homomorphic to it, and the background intensity of the small image is set depending to how high that value is. The icons for nodes are pre-generated using *matplotlib*, and the images are loaded into the HTML page. The implementation suffices performance requirements for all used datasets.

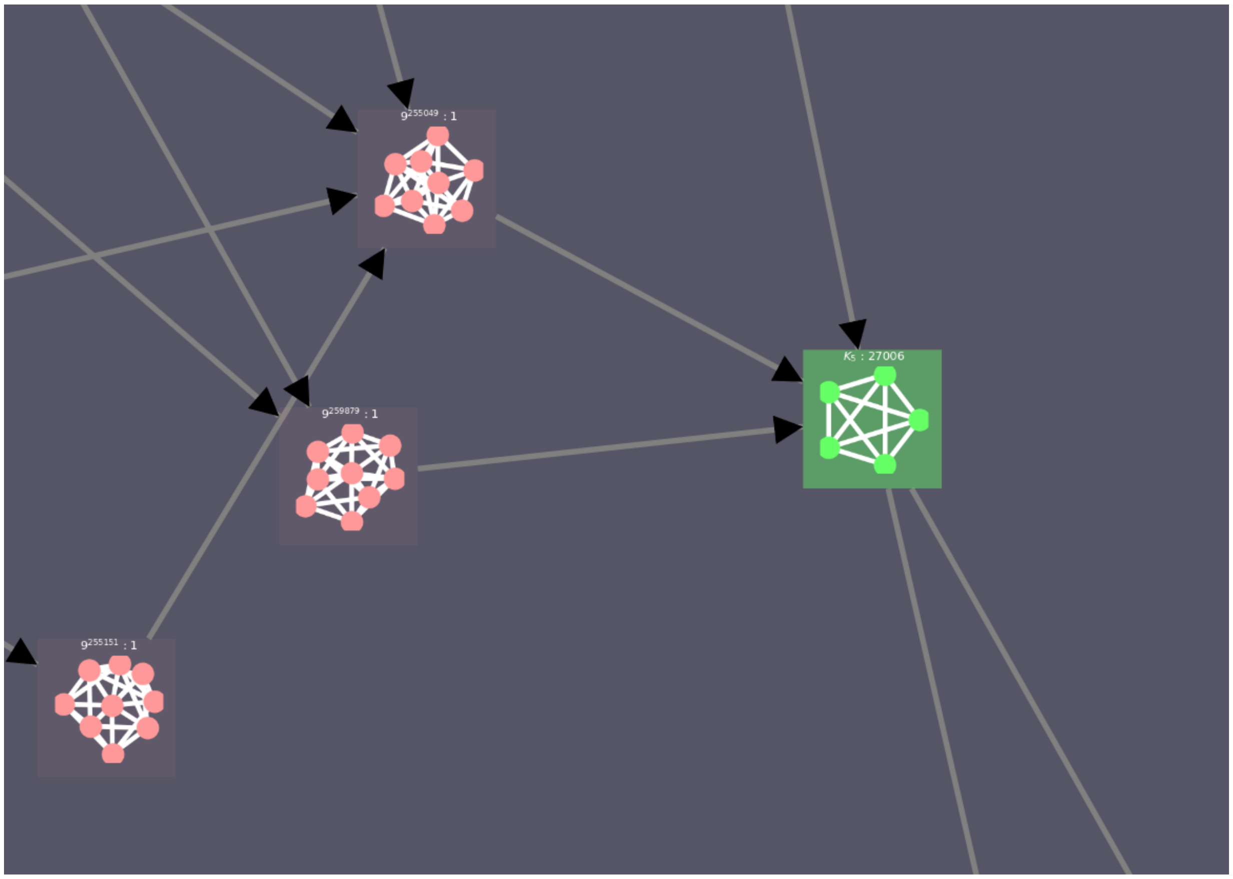

Figure 2.5.1: Screenshot of the interactive visualization software.

2.6 Experimentation

The above software therefore can be used on the data to generate and visualize lattices of sizes 6, 7, 8 and 9. This will allow to see the progression as the lattice “grows”, on multiple scales.

The software described above contains three main parts: homomorphism solver, a part that orders graphs according to their relation and visualization part. All parts have been profiled and tested, and the results will be presented and analysed in the next section.

3 Analysis

The software was used to visualize parts of the lattice which only include connected cores. Figure 3.0.1 illustrates a poset of them of size up to 6. The complete graphs are coloured in green, and the cycles are coloured in yellow. It can be observed, that there is also a graph, coloured in blue, which does not belong to either of those families. When connected cores of size 7 are added to this poset, which can be observed on on Figure 3.0.2, the diagram still features three complete graphs at its end and odd cycles in its beginning. This pattern repeats further on Figure 3.2.2, and the observation can therefore be extrapolated to Conjecture 3.1.1, which will shortly be formalized. These, however, require a certain termonology to be defined that is specific to the context of this analysis.

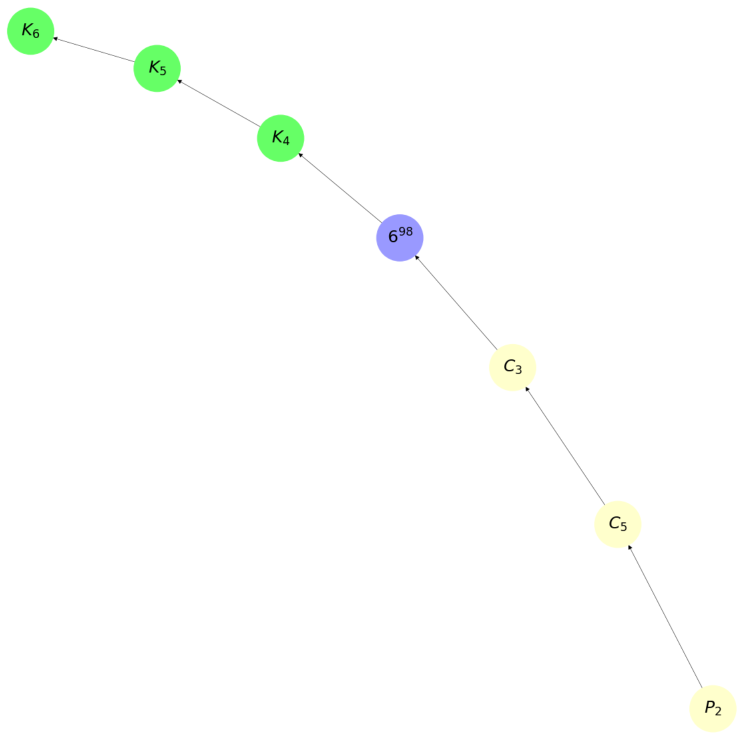

Figure 3.0.1: Hasse diagram of connected cores of size up to 6. The interactive version can be accessed online.

Unless stated otherwise, in this section two graphs will be considered the same if they are isomorphic to each other.

3.1 Cores between K_{n-2} and K_{n-1}

Conjecture 3.1.1. There is no such core $\mathcal{C}$ on n vertices that $K_{n-2} \prec \mathcal{C} \prec K_{n-1}$.

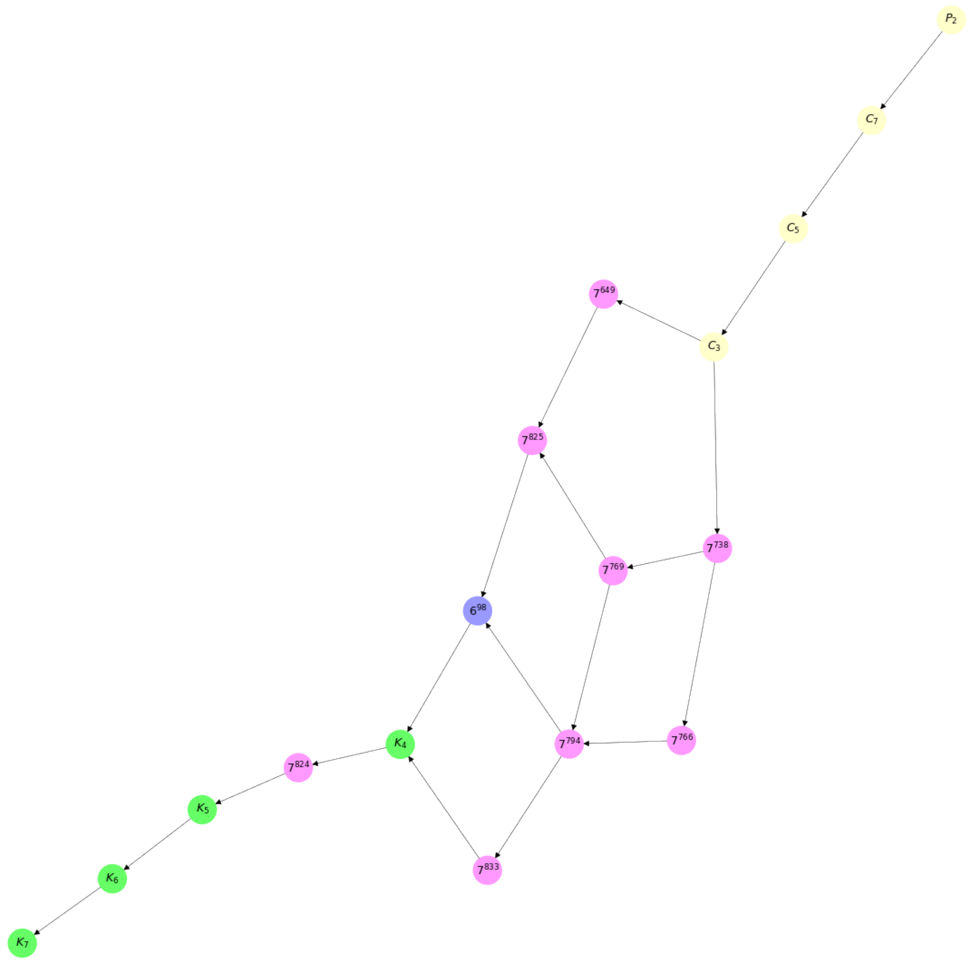

Figure 3.0.2: Hasse diagram of connected cores of size up to 7. The interactive version can be accessed online.

Proof of the Conjecture 3.1.2. This proof can be split into three stages:

1. **Prove**: G has $\geq n$ nodes:

   Clearly, it is required that $\chi(G) = n-1$, which is not possible for a graph on less than $n-1$ vertices, and is only possible for complete graph on $n-1$ vertices. But $K_{n-1} \not\prec K_{n-1}$. Contradiction. Hence there can only be suitable cores with at least n vertices.

2. **Prove**: G does not contain a clique on $n-1$ vertices.

   Assume the opposite. Then by Theorem 1.5.6, $K_{n-1} \preceq G$. Contradiction.

3. Assume G is a graph on n vertices that does not have a clique on $n-1$ vertices. Now, take two distinct nodes that are not connected to each other: a and b. Say that the

node-induced subgraph H is such that $V(H) = V(G) \setminus \{a, b\}$. Then $\chi(H) \leq n-2$. Since a is not connected to b, a node-induced subgraph on nodes a and b will be 1-chromatic. Then, by Theorem 1.4.5, $\chi(G) = n-1 \leq \chi(H)+1 \leq (n-2)+1$. Hence, $\chi(H) = n-2 \implies H = K_{n-2}$. If a is connected to every node in H, G has a $(n-1)$-clique, which would lead to a contradiction. Hence there is a node in H that a is not connected to, and therefore a can be assigned the same colour as that node. With dual argument on b it follows that G is $(n-2)$-colourable. Contradiction.

Hence there is no such graph G on up to n vertices such that $K_{n-2} \prec G \prec K_{n-1}$. □

Corollary 3.1.2. Any graph G on n vertices such that $K_{n-2} \prec G \preceq K_{n-1}$ has a $(n-1)$-clique.

Proof. Let G be a graph on n vertices, such that $K_{n-2} \prec G \preceq K_{n-1}$. By Conjecture 3.1.1, there is no such core that sits between K_{n-2} and K_{n-1}, hence $G \longleftrightarrow K_{n-1}$. By Proposition 1.5.19, each graph contains its core as an induced subgraph, therefore it has $(n-1)$-clique. □

3.2 Unique core between K_{n-3} and K_{n-2}

Similarly to an observation which led to Conjecture 3.1.1, it can be noticed that on Figures 3.0.1, 3.0.2 and 3.2.2 there is exactly one core between K_{n-3} and K_{n-2}. Such core is a graph join of a 5-cycle and a clique, as shown for 5 to 9 vertices on Figure 3.2.1. One of these graphs, of size 6, was noted before on Figure 3.0.1 as a blue graph. Conjecture 3.2.9 will therefore be summarized.

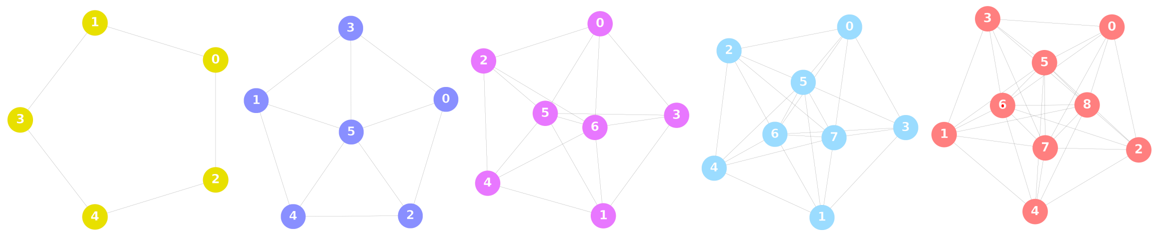

Figure 3.2.1: Unique cores observed between K_{n-3} and K_{n-2}.

It turns out, that for further analysis it is convenient to define a property of a graph, when adding a vertex to it only increases its chromatic number if this vertex is joined. Such graphs will be called χ-dense for the purposes of this section:

Definition 3.2.1. Let G be a graph, and G' be its node-induced supergraph, which contains one node more than G. Let a be that node. Then if a is not connected to all nodes in G and $\chi(G) + 1 = \chi(G')$, then let G be called *χ-dense*.

One of such graphs is a complete graph:

Proposition 3.2.2. Complete graphs are χ-dense.

Proof. Assume the opposite. Let G be a complete graph, and let a, G' be defined as in Definition 3.2.1. Let b be a node in G that a is not connected to. Then a can be coloured with the colour of b, since they are not connected, and no other node has the same colour. Thus $\chi(G') = \chi(G)$. Contradiction.

Hence complete graphs are χ-dense. $\square$

Recall. Recall from Theorem 1.4.5, that given G and its node-partitioning H, K, it turns out that $\chi(G) \leq \chi(H) + \chi(K)$, and the equality can be attained when $G = H + K$.

The nice property of such graphs (Definition 3.2.1) is that, the equality is attained only if they are joined.

Theorem 3.2.3. Let G, H, P be a graphs, such that $G = H + P$. Then G is χ-dense iff H is χ-dense and P is χ-dense.

Proof. Let a and G' be defined from G similarly to that in Definition 3.2.1. Then let H' and K' be node-induced supergraphs of H and K respectively, so that $H = H' \setminus \{a\}$ and $K = K' \setminus \{a\}$.

($\Leftarrow$). Assume a is not connected to some node in H. By Theorem 1.4.5, $\chi(G') \leq \chi(H') + \chi(K)$. Since H is χ-dense, $\chi(H') = \chi(H)$. Therefore, $\chi(G') \leq \chi(H) + \chi(K) = \chi(G)$. Hence, G is not χ-dense. Contradiction. By a dual assumption on K it is obtained that G is χ-dense.

($\Rightarrow$) Assume the opposite. Without loss of generality, assume H is not χ-dense. Then there is a way to connect a to nodes of H in such a way, that a is not connected to all nodes in H but $\chi(H') = \chi(H) + 1$. Let a be connected to all nodes in K then. Then $G' = H' + K$, therefore, $\chi(G') = \chi(H') + \chi(K) = \chi(H) + \chi(K) + 1 = \chi(G) + 1$. Since a is not connected to some node in H, G is not χ-dense. Contradiction.

Hence G is χ-dense $\iff$ H and K are χ-dense. $\square$

Lemma 3.2.4. Odd cycles are χ-dense.

Proof. Assume the opposite. Let $G = C_{2k+1}$, and G' be a node-induced supergraph of G on $2k + 2$ vertices. Denote $a = G' \setminus G$. Then there exists $b \in G$ that a is not connected to. Then denote $G' \setminus \{a, b\} = G \setminus \{b\} = P_{2k}$ as X. Since path graphs are bipartite, it can be coloured in two colours 1 and 2, and b is therefore assigned color 3; then colour a into colour 3. It is then trivial that G' has a valid 3-colouring, and since G is 3-chromatic, the assumption is contradicted.

Hence odd cycles are χ-dense. $\square$

For the following facts, it will be useful to note the following fact:

Lemma 3.2.5. Let $\phi : G \to H$ be a graph homomorphism. Then ϕ maps nodes that are connected to all other nodes injectively.

Proof. Consider node a, which is connected to all other nodes, and some other node b. Then $\phi(a) = \phi(b) \implies (a, b) \notin G$. Contradiction.

Hence, ϕ maps such nodes injectively. □

Theorem 3.2.6. Let G be a core, such that $\exists a \in G$: a is connected to all other nodes. Then in a graph join $G = C + a$, C is a core.

Proof. Assume the opposite. Then there exists a homomorphism ϕ from C to its proper induced subgraph, there would be a homomorphism ψ from G to a proper induced subgraph of G, which maps a to a, since it is connected to all other nodes and is not in C, and every other node x to $\phi(x)$. Contradiction.

Hence, C is a core. □

Theorem 3.2.7. Let $\mathcal{C}$ be a homomorphism core. Then $\mathcal{C} + K_1$ is also a homomorphism core.

Proof. Let H be any induced subgraph of G, such that $G \to H$ with homomorphism ϕ. Then $\phi(\mathcal{C}) \simeq \mathcal{C}$, since $\mathcal{C}$ is a core, and $\phi(C)$ is a homomorphism that maps its edges. By Lemma 3.2.5, ϕ will also map a injectively, i.e. $\phi(a) \notin \phi(\mathcal{C})$, and since a is connected to every node in $\mathcal{C}$, $\phi(a)$ will be connected to every node in $\phi(\mathcal{C})$. Therefore, $H \simeq G$, and hence G is a core. The proof is complete. □

Theorem 3.2.8. Let $\mathcal{C}$ be a graph, and let $n \geq 0$. Then $\mathcal{C}$ is a core iff $\mathcal{C} + K_n$ is a core.

Proof. This can be proven by induction on n.

Base: Let $n = 0$. $\mathcal{C} + K_0 = \mathcal{C}$, the case is trivial.

Assumption: $n = k$.

$\mathcal{C}$ is a core $\iff$ $\mathcal{C} + K_k$ is a core.

Step: $n = k + 1$.

By Proposition 1.2.11, $K_{k+1} = K_k + K_1$. Hence $\mathcal{C} + K_{k+1} = \mathcal{C} + K_k + K_1 = (\mathcal{C} + K_k) + K_1$. Let $\mathcal{D} = \mathcal{C} + K_k$. By assumption, $\mathcal{D}$ is a core iff $\mathcal{C}$ is a core. Then by Theorem 3.2.7, $\mathcal{D}$ is a core $\implies$ $\mathcal{D} + K_1$ is a core. And by Theorem 3.2.6, if $\mathcal{D} + K_1$ is a core, then $\mathcal{D}$ is a core. Therefore, $\mathcal{C} + K_k + K_1 = \mathcal{D} + K_1$ is a core $\iff$ $\mathcal{D}$ is a core $\iff$ $\mathcal{C}$ is a core.

Hence, $\mathcal{C}$ is a core $\iff$ $\mathcal{C} + K_n$ is a core for any $n \geq 0$. □

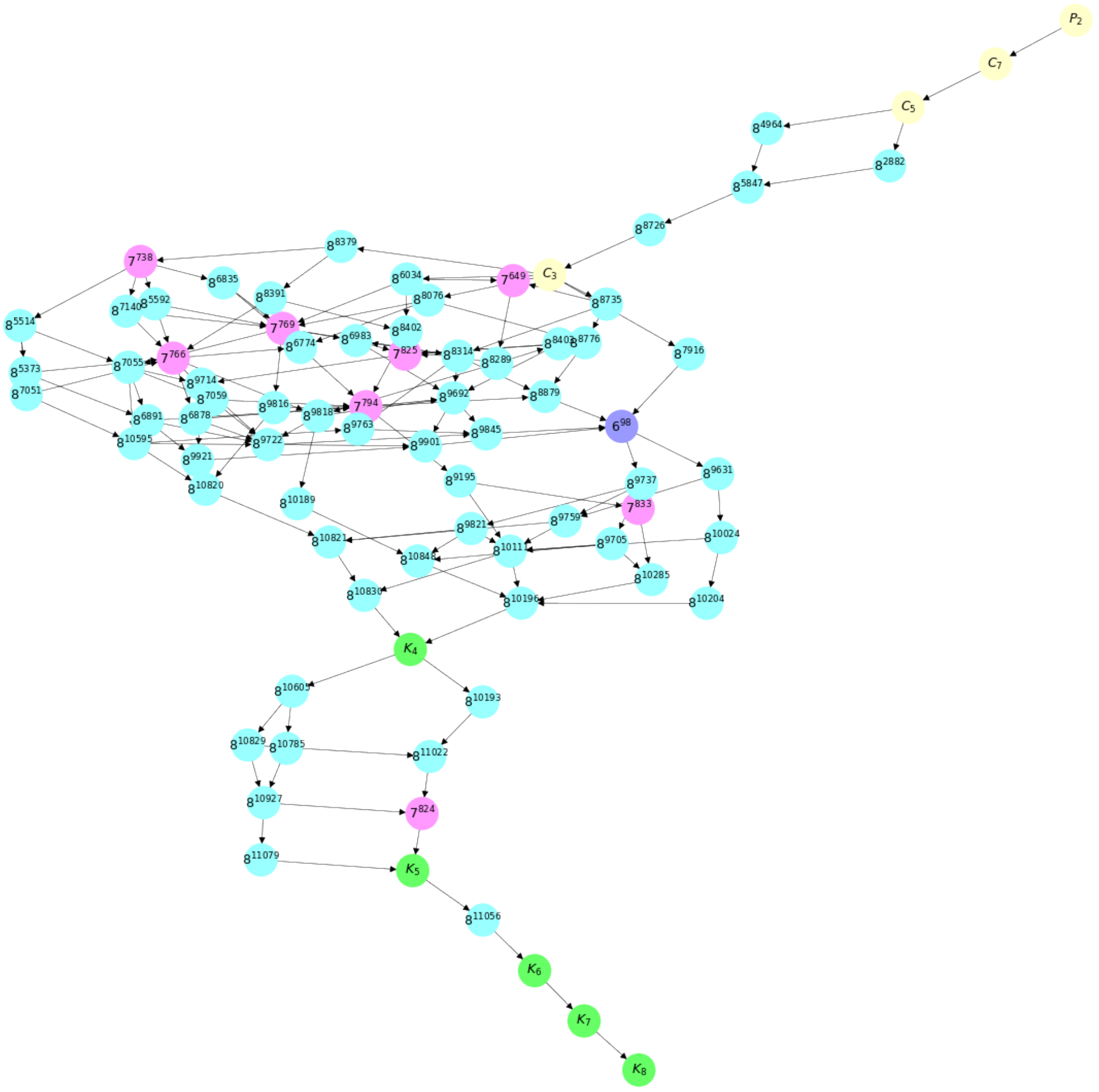

Figure 3.2.2: Hasse diagram of connected cores of size up to 8. The interactive version can be accessed online.

Conjecture 3.2.9. There is exactly one core $\mathcal{C}$ such that $K_{n-3} \prec \mathcal{C} \prec K_{n-2}$ on up to n vertices.

Before proving the Conjecture 3.2.9, it would be useful to introduce derivatives of C_5, which can be seen as unique graphs on Figures 3.0.1, 3.0.2, and 3.2.2.

Definition 3.2.10. Let $n \geq 5$. Then define $Q_n = C_5 + K_{n-5}$.

Proposition 3.2.11. Q_n is χ-dense.

Proof. By Lemma 3.2.4, C_5 is χ-dense. By Proposition 3.2.2, K_{n-5} is χ-dense. Hence by Theorem 3.2.3, $Q_n = C_5 + K_{n-5}$ is χ-dense. □

Proposition 3.2.12. Q_n is a core.

Proof. $Q_n = C_5 + K_{n-5}$. Hence, by Theorem 3.2.8, Q_n is a core iff C_5 is a core. C_5 is a trivial core, hence Q_n is a core. □

Proof of Conjecture 3.2.9. By Conjecture 3.1.1, there are no such cores on up to $n-1$ vertices.

Hence, only graphs on n vertices should be considered. The proof will use induction:

Base: $n = 5$, it is experimentally shown that only such connected core is C_5 (Figure 3.0.1). It is also trivial that no disconnected core satisfies this property.

Assumption: $n = k$, Q_k is the only suitable graph.

Step: $n = k+1$. Let $\mathcal{C}$ be a core on $k+1$ nodes.

Then the following two cases are possible:

1. Let $a \in \mathcal{C}$ be a node such that $\chi(\mathcal{C} \setminus a) + 1 = \chi(\mathcal{C})$. Then $\mathcal{C} \setminus a$ is a graph on $n-1$ nodes that is $(n-3)$-colourable. By assumption it is then either, equivalent to K_{n-3}, or Q_{n-1}. If it is equivalent to Q_{n-1}, then by Proposition 3.2.11, it is χ-dense, hence $\mathcal{C} = Q_{n-1} + a = Q_n$. If, on the other hand, it is equivalent to K_{n-3}, then by Corollary 3.1.2, $\mathcal{C} \setminus a$ has a clique on $(n-3)$ vertices. Let H represent that clique, and let $P = \mathcal{C} \setminus H$. By Theorem 1.4.5, $n-2 = \chi(\mathcal{C}) \leq \chi(H) + \chi(P) = n-3+\chi(P) \leq n-3+1 = n-2$. Therefore, P is 1-colourable, i.e. P has no edges. By Proposition 3.2.2, H is χ-dense, Therefore, one of the nodes in P has to be connected to all nodes in H. Then $\mathcal{C}$ contains a clique on $(n-2)$ nodes. By Theorem 1.5.6, $K_{n-2} \preceq \mathcal{C}$. Contradiction.
2. Otherwise, $(C \setminus a)$ is $(n-2)$-colourable, and by Conjecture 3.1.1 must have a $(n-2)$-clique. Therefore, $K_{n-2} \preceq C$. Contradiction.

Hence Q_n is a unique core between K_{n-3} and K_{n-2} on n vertices. □

The following therefore summarizes Conjecture 3.2.9:

Corollary 3.2.13. Let G be a graph on n vertices, such that $\chi(G) = n-2$. Then G is either of two kinds: the first kind has $(n-2)$-clique and the second kind is Q_n Conjecture 3.2.9.

Proof. Let G be a graph on n vertices. Since $\chi(G) = n-2$, $K_{n-3} \prec G \preceq K_{n-2}$. Then by Conjecture 3.2.9, G is Q_n or $G \iff K_{n-2}$. In the latter case, by Corollary 1.5.22, K_{n-2} is a node-induced subgraph of G, since it is its core. Hence, the proof is complete. □

This is where the χ-density is important: if there were at least two ways of increasing chromatic number of a graph, it would be difficult to count how many graphs are derived from each smaller graph, as there would be no guarantee that they would be distinct.

3.3 Seven cores between K_{n-4} and K_{n-3}

It can be noticed that for $n = 7, 8$ there are exactly 8 connected cores between K_{n-4} and K_{n-3}, as shown on Figures 3.0.2, 3.2.2. Moreover, these cores are ordered in a similar way. The 8 cores that can be seen on Figure 3.0.2 are topologically in the same order on Figure 3.2.2. That is, they form the same topological minor on the picture. This is also the case for $n = 9$, the diagram for which is not available as an image[2]. This leads to Conjecture 3.3.1. The eight cores are shown on Figure 3.3.

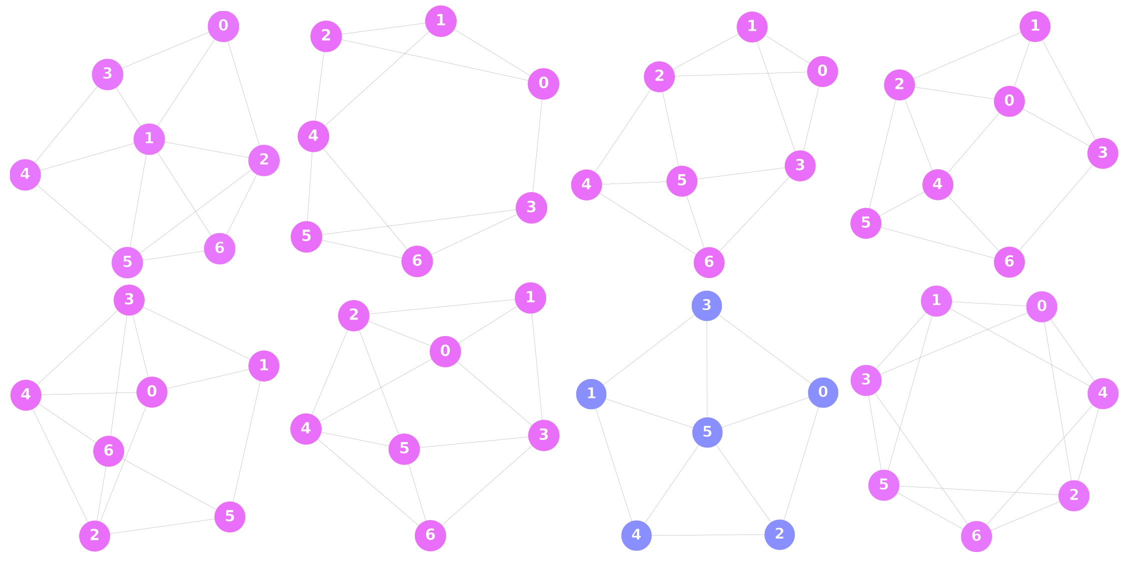

Figure 3.3.1: Cores in a substructure observed between K_{n-4} and K_{n-3}.

Conjecture 3.3.1. There are exactly 7 homomorphism cores on n vertices between K_{n-4} and K_{n-3}, denote $T_n^1 \dots T_n^7$, and 1 such core on $n-1$ vertices, which is Q_{n-1}. Moreover, their precedence to each other is preserved for any $n > 8$.

From Figure 3.3, it can be observed that one of them is a core from Conjecture 3.2.9, which is proven to be there for $n-1$; the rightmost graph on the bottom is a circular-complete graph $K_{7/3}$. An interesting observation is that a half of these graphs can be turned into Q_6 by a non-edge contraction.

Definition 3.3.2. Let $n \geq 7$, and $k \in \{1, \dots, 7\}$. Then T_7^k will be graphs, shown on Figure 3.3, and T_n^k will be defined to be $T_7^k + K_{n-7}$ for $n \geq 8$.

[2]Hasse diagram of connected cores of size up to 9 can be accessed at online.

Similar to Q_n, T_n^k needs to be proven to be χ-dense.

Proposition 3.3.3. Let $n \geq 7, k \in \{1 \dots 7\}$. Then T_n^k is χ-dense.

Proof. By Theorem 3.2.3, $T_n^k = T_7^k + K_{n-7}$ is χ-dense iff T_7^k is χ-dense and K_{n-7} is χ-dense. The latter is shown by Proposition 3.2.2, and the first is shown experimentally. Hence, T_n^k is χ-dense. □

Proposition 3.3.4. Let $n \geq 7, k \in \{1 \dots 7\}$. Then T_n^k is a core.

Proof. By Theorem 3.2.8, $T_n^k = T_7^k + K_{n-7}$ is a core $\iff$ T_7^k is a core. The latter is shown experimentally. Hence T_n^k is a core. □

Proof of Conjecture 3.3.1. This can be proven by induction.

Base: For $n = 7$, it is shown experimentally that $T_7^1 \dots T_7^1$ are the only such connected cores. If the core is disconnected, its connected components would have to be pairwise incomparable by Theorem 1.5.29, but there are no incomparable connected cores on $n < 7$. Hence, there is no such disconnected core that suffices the base condition.

Assumption: Let $n = k$. Then the only cores on k vertices which adhere to the set property are cores $T_k^1 \dots T_k^7$.

Step: Let $n = k + 1$. Let G be a core on $k + 1$ vertices, such that $K_{k-3} \prec G \prec K_{k-2}$.

Assume there exists such vertex a that $\chi(G \setminus a) + 1 = \chi(G)$. Then $G \setminus a$ is a graph on k nodes that is $(k - 3)$-colourable. By assumption, $G \setminus a$ is either, equivalent to K_{k-3}, in which case it is not a core, or it is T_k^p for some $p \in \{1 \dots 7\}$. Since T_k^p is χ-dense, $\chi(G) = \chi(G \setminus a) + 1 \implies G = T_k^p + a = T_{k+1}^p$. Therefore, in this case, $T_{k+1}^1 \dots T_{k+1}^7$ are exactly the cores on n vertices.

If, however, there is no such vertex a that $\chi(G \setminus a) + 1 = \chi(G)$, then let x be any vertex of G, and let $H = G \setminus x$. H is a graph on k vertices that is $k - 2$-colourable. By Corollary 3.2.13, there are only two kinds of such graphs: Q_k and graphs, which contain $(k - 2)$-clique. Below those cases can be considered separately:

1. In the latter case, K_{k-2} is a node-induced subgraph of G, so by Theorem 1.5.6, $K_{k-2} \preceq G \implies G \not\prec K_{k-2}$. Contradiction.

2. Therefore, let $H = Q_k = C_5 + K_{k-5}$. Denote the joined $(k - 5)$-clique as P and $F = G \setminus P$, a node-induced subgraph with a 5-cycle and x:

   If x is connected to any vertex y in P, then P contains a node that is connected to all other nodes. Then $G = y + (G \setminus y)$, and by Theorem 3.2.8, G is a core iff $G \setminus y$ is a core. By assumption, there is no such core $G \setminus y$, hence G is not a core.

Then, x is not connected to any node in P. Let y be any node in P and define mapping $\phi : G \to G$, such that $\phi(v) = \begin{cases} y & v = x \\ v & v \neq x \end{cases}$. Then for any v in G, if x is connected to v, then $\phi(x) = y$ is connected to $\phi(v) = v$, and all other edges are preserved because ϕ preserves $G \setminus x$. Hence, ϕ is a valid homomorphism onto a node-induced subgraph $G \setminus x$. By Theorem 1.5.6, $G \setminus x \to G$, since it is a node-induced subgraph. Thus, $G \iff Q_k \implies G$ is not a core. Contradiction.

Therefore, $T_n^1 \dots T_n^7$ are the only cores on n vertices the chromatic number for which is $(n-3)$.

The proof of ordering preservation can be constructed as follows. Let $\mathcal{B}_n$ be the 8 cores for $n \geq 7$. In the base-case, the ordering between elements of $\mathcal{B}_7 = \{T_7^1 \dots T_7^7, Q_6\}$ can be found on Figure 3.0.2. By inductive proof, it is true that any core $X_n \in \mathcal{B}_n$ is in fact $X_7 + K_{n-7}$. Let then X_n and Y_n be two elements of $\mathcal{B}_n$. Then if $X_7 \not\to Y_7$, it follows by Theorem 1.5.15 that $X_n = X_7 + K_{n-7} \not\to Y_7 + K_{n-7}$. Similarly, if $X_7 \to Y_7$, then trivially $X_n = X_7 + K_{n-7} \to Y_n = Y_7 + K_{n-7}$. Hence, the ordering between any X_n and Y_n is the same as it is between X_7 and Y_7.

Hence, the proof is complete. □

Corollary 3.3.5. Let G be a graph on n vertices such that $K_{n-4} \prec G \preceq K_{n-3}$. Then G is equivalent to one of the following graphs: $\{Q_{n-1}, T_n^1, \dots, T_n^7, K_{n-3}\}$.

Proof. By Conjecture 3.3.1, there are only 9 different cores on up to $n \geq 7$ vertices, the chromatic number of which is $n-3$. Therefore, any graph G such that $\chi(G) = n-3$ belongs to an equivalence class of one of those 9 graphs. □

Therefore, the result of the above that for given sufficiently large n, any sublattice that ranges from K_{n-4} to K_n (see Theorem 1.5.39), it will have exactly no join-irreducible non-complete cores between K_{n-2} and K_n on up to n vertices, it will have exactly one between K_{n-3} and K_{n-2}, some of which are shown on Figure 3.2.1 and it will have eight, between K_{n-4} and K_{n-3}, some of which are shown on Figure 3.3, in the order shown on Figure 3.0.2.

This concludes the analysis section of the dissertation, with all observations successfully proven to be correct in more general setting.

4 Conclusion

Graph homomorphisms are a mapping between graphs that preserves edges. It is a generalization of graph isomorphisms and graph colouring, and it forms a preorder on finite simple graphs. On the equivalence side of it, any finite subset of graphs in the class corresponds to a group under homomorphism. Additionally, all elements of the same class contain a the smallest element of it as a subgraph, which is called a core. These, in turn, form a distributive lattice, the join- and meet-irreducible elements of which are exactly the connected and the multiplicative cores. This way, any core could be constructed from its connected (or multiplicative) cores. Furthermore, since it has been externally shown that cores form a Heyting algebra, finite simple graphs set two algebraic structures under homomorphism. Between any two graphs there are infinitely many others. In particular, for any graph G there exists an integer n, such that $K_n \preceq G \prec K_{n+1}$, and the question of whether every complete graph is a meet-irreducible element remains unanswered to this day.

It is also not yet known, whether graph homomorphism is solvable in polynomial time, but it is conjectured to not be possible. As such, the best known algorithms on any existing machines solve it in an exponential number of steps. Because of that, practical performance of an algorithm will heavily depend on a choice of heuristics and design approach, such as arc consistency, etc, and can benefit from reducing given graphs to their cores. The latter is aided by deducing whether two graphs are homomorphic based on their relation to smaller graphs. Hence, constructing a database of smaller graphs practically benefits a solver, and can also be visualized as a Hasse diagram. The latter was further analysed, and it turned out, that the diagram ends with the same set of graphs, which preserve the order of their relation for any n.

4.1 Further work

A further research could, potentially, include graph polymorphisms, which are closely related to the problem of finding multiplicative and exponential graphs [Bod15, HHML88, Tar05]. A direct product of any two cores is homomorphic to both of them, so the problem of establishing the order of cores of the products should be reducible. This, however, would mainly require a more focused analysis of multiplicative graphs with respect to software implementation and solver design, as the size of such graphs grows exponentially.

References

[ADK07] Vikraman Arvind, Bireswar Das, and Johannes Köbler. The space complexity of k-tree isomorphism. In *International Symposium on Algorithms and Computation*, pages 822–833. Springer, 2007.

[ADK08] Vikraman Arvind, Bireswar Das, and Johannes Köbler. A logspace algorithm for partial 2-tree canonization. In *International Computer Science Symposium in Russia*, pages 40–51. Springer, 2008.

[AGKO03] David Abrahams, Ralf W Grosse-Kunstleve, and Operator Overloading. Building hybrid systems with boost. python. *CC Plus Plus Users Journal*, 21(7):29–36, 2003.

[B$^+$37] Garrett Birkhoff et al. Rings of sets. *Duke Mathematical Journal*, 3(3):443–454, 1937.

[Bab96] László Babai. Automorphism groups, isomorphism, reconstruction. In *Handbook of combinatorics (vol. 2)*, pages 1447–1540. MIT Press, 1996.

[Bab16] László Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 684–697. ACM, 2016.

[BBC$^+$11] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31, 2011.

[bh] bios (https://stackoverflow.com/users/943657/bios). Cython: Inline function not pure c. StackOverflow. URL:https://stackoverflow.com/q/13912726/4811285 (version: 2014-06-01).

[Bir40] Garrett Birkhoff. *Lattice theory*, volume 25. American Mathematical Soc., 1940.

[Bod15] Manuel Bodirsky. Graph homomorphisms and universal algebra course notes. *Institut fur Algebra, TU Dresden*, 1:1–56, 2015.

[Bou78] SR Bourne. Unix time-sharing system: The unix shell. *The Bell System Technical Journal*, 57(6):1971–1990, 1978.

[BW04] Graham R Brightwell and Peter Winkler. Graph homomorphisms and long range action. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 63:29–48, 2004.

[Cam06] Peter J Cameron. Graph homomorphisms. *Combinatorics Study Group Notes*, pages 1–7, 2006.

[Com14a] Wikimedia Commons. File:graph isomorphism a.svg — wikimedia commons, the free media repository, 2014. [Online; accessed 20-April-2019].

[Com14b] Wikimedia Commons. File:graph isomorphism b.svg — wikimedia commons, the free media repository, 2014. [Online; accessed 20-April-2019].

[Coo71] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[Dav02] PriestleyHA DaveyBA. Introductiontolatticesandorder, 2002.

[DBJM$^+$18] Jan De Beule, Julius Jonušas, James D Mitchell, Michael Torpey, and Wilf A Wilson. Digraphs–gap package. *Version 0.13. 0, September*, 2018.

[DS96] Dwight Duffus and Norbert Sauer. Lattices arising in categorial investigations of hedetniemi's conjecture. *Discrete Mathematics*, 152(1-3):125–139, 1996.

[EZS85] Mohamed El-Zahar and Norbert Sauer. The chromatic number of the product of two 4-chromatic graphs is 4. *Combinatorica*, 5(2):121–126, 1985.

[FK02] Jirí Fiala and Jan Kratochvíl. Partial covers of graphs. *Discussiones Mathematicae Graph Theory*, 22(1):89–99, 2002.

[Fru39] Robert Frucht. Herstellung von graphen mit vorgegebener abstrakter gruppe. *Compositio Mathematica*, 6:239–250, 1939.

[G$^+$] GAP Group et al. Gap–groups, algorithms, and programming, version 4.9. 2; 2018.

[Gra14] Charles T Gray. The digraph lattice. 2014.

[hb] Ilya Bogdanov (https://mathoverflow.net/users/17581/ilya bogdanov). Cancelling a graph join from a graph homomorphism. MathOverflow. URL:https://mathoverflow.net/q/281425 (version: 2017-09-18).

[Hed66] Stephen T Hedetniemi. Homomorphisms of graphs and automata. Technical report, MICHIGAN UNIV ANN ARBOR COMMUNICATION SCIENCES PROGRAM, 1966.

[hf] Tobias Fritz (https://mathoverflow.net/users/27013/tobias fritz). Cancelling a graph join from a graph homomorphism. MathOverflow. URL:https://mathoverflow.net/q/281414 (version: 2017-09-18).

[HHML88] Roland Häggkvist, Pavol Hell, Donald J. Miller, and V Neumann Lara. On multiplicative graphs and the product conjecture. *Combinatorica*, 8(1):63–74, 1988.

[HN92] Pavol Hell and Jaroslav Nešetřil. The core of a graph. *Discrete Mathematics*, 109(1-3):117–126, 1992.

[HN04] Pavol Hell and Jaroslav Nesetril. *Graphs and homomorphisms*. Oxford University Press, 2004.

[HSSC08] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[Hun07] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90, 2007.

[Kar72] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[Kas16] Gerald Kaszuba. Python call graph. *Internet: https://pycallgraph. readthedocs. io/en/master*, 2016.

[Kas19] Andrei Kashcha. Vivagraphjs. *Github Repository*, 2019.

[Lad75] Richard E Ladner. On the structure of polynomial time reducibility. *Journal of the ACM (JACM)*, 22(1):155–171, 1975.

[Lev73] Leonid Anatolevich Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

[Lov07] László Lovász. *Combinatorial problems and exercises*, volume 361. American Mathematical Soc., 2007.

[LS04] Javier Larrosa and Thomas Schiex. Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004.

[Mac38] Saunders MacLane. Stone mh. topological representations of distributive lattices and brouwerian logics. časopis pro pěsiování matematiky a fysiky, vol. 67 (1937–1938), pp. 1–25. *The Journal of Symbolic Logic*, 3(2):90–91, 1938.

[McK] Brendan McKay. Combinatorial data: Graphs, non-isomorphic simple graphs. [Online; accessed 20-April-2019].

[MP14] Brendan D McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014.

[Pap03] Christos H Papadimitriou. *Computational complexity.* John Wiley and Sons Ltd., 2003.

[Pri70] Hilary A Priestley. Representation of distributive lattices by means of ordered stone spaces. *Bulletin of the London Mathematical Society*, 2(2):186–190, 1970.

[S$^+$99] Michel F Sanner et al. Python: a programming language for software integration and development. *J Mol Graph Model*, 17(1):57–61, 1999.

[Sau01] Norbert Sauer. Hedetniemi's conjecture—a survey. *Discrete Mathematics*, 229(1-3):261–292, 2001.

[SB81] Hanamantagouda P Sankappanavar and Stanley Burris. A course in universal algebra. *Graduate Texts Math*, 78, 1981.

[Sch88] Uwe Schöning. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*, 37(3):312–323, 1988.

[Slo14] NJA Sloane. Sequence a001349. the on-line encyclopedia of integer sequences, 2014.

[Tar05] Claude Tardif. Multiplicative graphs and semi-lattice endomorphisms in the category of graphs. *Journal of Combinatorial Theory, Series B*, 95(2):338–345, 2005.

[W$^+$96] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, NJ, 1996.

[Wei] Eric W Weisstein. Hasse diagram. from mathworld–a wolfram web resource.

[Wel82] Emo Welzl. Color-families are dense. *Theoretical Computer Science*, 17(1):29–41, 1982.

[Wik12] Wikipedia. File:lattice v4.png, 2012. [Online; accessed 20-April-2019].

[WR83] Douglas R White and Karl P Reitz. Graph and semigroup homomorphisms on networks of relations. *Social Networks*, 5(2):193–234, 1983.

[Z$^+$98] Xuding Zhu et al. A survey on hedetniemi's conjecture. *Taiwanese Journal of Mathematics*, 2(1):1–24, 1998.

[Zoo] Complexity Zoo. Complexity zoo:n. [Online; accessed 20-April-2019].