Czech Technical University in Prague

Faculty of Electrical Engineering
Department of Computer Science

Lock-chart solving

Doctoral Thesis

Radomír Černoch, MSc.

Prague, October 2017

Ph.D. Programme:
Electrical Engineering and Information Technology

Branch of study:
Artificial Intelligence and Biocybernetics

Supervisor: prof. Ing. Filip Železný, Ph.D.

ABSTRACT

Lock-chart solving (also known as master key system solving) is a process for designing mechanical keys and locks so that every key can open and be blocked in a user-defined set of locks. This work is an algorithmic study of lock-chart solving.

Literature on this topic [34, 38, 44, 53] has established that the extension variant of the problem is $\mathcal{NP}$-complete, reformulated lock-chart solving as a constraint satisfaction problem (CSP) with set variables, applied a local search algorithm, and defined a symmetry-breaking algorithm using automorphisms.

However, the otherwise standard decision problem with a discrete search space has a twist. After a lock-chart is solved and its solution is fixed, new keys and locks may be added as a part of an extension, and the original solution should be prepared for this. In the first formal treatment of extensions, several scenarios are proposed, and effects on lock-chart solving algorithms are discussed.

First, we formalise lock-chart solving. 6 variants of lock-charts and 4 constraint frameworks of increasing generality and applicability to real-world problems are formulated. Their hierarchy is used to extend the classification of lock-chart solving problems into computational complexity classes. A close relationship between the most realistic framework and the Boolean satisfiability problem (SAT) is established. Mechanical profiles are shown to express $\mathcal{NP}$-complete problems as a complement to the previous result on the extension problem variant. We give the first proof that diagonal lock-charts (systems with only one master key) can be solved in $\mathcal{P}$ using an algorithm known as *rotating constant method*.

The practical part proposes several algorithms for lock-chart solving. The problem is translated into SAT, into CSP (with standard variables) and partly into the maximum independent set problem. The SAT translation inspires a model-counting algorithm tailored for lock-charts. Finally, we describe a customised depth-first-search (DFS) algorithm that uses the model-counter for pruning non-perspective parts of the search space. In the empirical evaluation, CSP and the customised DFS improve the performance of the previous automorphism algorithm.

ABSTRAKT

Řešením systému generálního a hlavních klíčů (SGHK) se myslí návrh uzávěrů mechanických klíčů a blokovacích prvků zámků. Návrh musí respektovat požadavek, aby každý klíč v systému otevíral uživatelem zadanou množinu zámků. Tato práce poskytuje algoritmickou analýzu SGHK.

Relevantní literatura [34, 38, 44, 53] již dokázala jednu variantu problému jako $\mathcal{NP}$-úplnou, přeformulovala problém jako programování s omezujícími podmínkami (CSP) s použitím množinových proměnných, aplikovala simulované žíhání a definovala symetrie stavového prostoru pomocí automorfismu.

Jinak běžná úloha s diskrétním stavovým prostorem má háček. Zákazník může objednat tzv. rozšíření – přidání nových klíčů a zámků do již vyrobeného SGHK. Prvotní řešení proto musí počítat s omezujícími podmínkami, jejichž přesná forma není v době návrhu známa. Tato práce je dle našich znalostí první formální studí problému rozšíření SGHK.

Práce nejdříve formalizuje pojem SGHK v několika variantách a navrhuje čtyři způsoby formalizace omezujících podmínek od nejjednodušší po nejrealističtější. Hierarchie rozhodovacích úloh je využita pro klasifikaci do tříd výpočetní složitosti. Nejdříve je popsána úzká vazba na úlohu splnitelnosti výrokových formulí (SAT). Mechanické profily se ukazují dostatečně expresivní pro překlad $\mathcal{NP}$-úplných úloh, což doplňuje již existující výsledek. Práce obsahuje první důkaz příslušnosti diagonální úlohy (SGHK s generálním klíčem, ale bez dalších hlavních klíčů) do třídy $\mathcal{P}$ pomocí tzv. *rotating constant method*.

Praktická část práce navrhuje několik algoritmů pro řešení SGHK. Problém je přeložen na SAT, na CSP a jeho část na úlohu hledání maximální nezávislé množiny. Pro počítání počtu klíčů splňující omezující podmínky je použito dynamické programování a princip inkluze a exkluze. Závěrem je popsán upravený algoritmus prohledávání do hloubky (DFS), který prořezává neperspektivní části stavového prostoru pomocí počitadla klíčů. V emprickém porovnání CSP a upravené DFS algoritmy prokázaly přínos stávajícímu algoritmu využívající automorfismy.

PUBLICATIONS

List of publications is presented for the purpose of dissertation defence. The degree of authorship is split between all authors of every publication equally.

IMPACTED JOURNAL ARTICLES RELEVANT TO THE TOPIC OF THIS DISSERTATION:

[1] Radomír Černoch, Ondřej Kuželka, and Filip Železný. "Polynomial and extensible solutions in lock-chart solving". In: *Applied Artificial Intelligence* 30.10 (2016), pp. 923–941.

OTHER ARTICLES IN IMPACTED JOURNALS:

[1] Roman Barták, Radomír Černoch, Ondřej Kuželka, and Filip Železný. "Formulating the template ILP consistency problem as a constraint satisfaction problem". In: *Constraints* 18.2 (2013), pp. 144–165.

OTHER PEER-REVIEWED CONFERENCE PAPERS:

[1] Radomír Černoch and Filip Železný. "Speeding Up Planning through Minimal Generalizations of Partially Ordered Plans". In: *Inductive Logic Programming: 20th International Conference, ILP 2010, Florence, Italy, June 27-30, 2010. Revised Papers*. Ed. by Paolo Frasconi and Francesca A. Lisi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 269–276. ISBN: 978-3-642-21295-6.

[2] Radomír Černoch and Filip Železný. "Subgroup Discovery Using Bump Hunting on Multi-relational Histograms". In: *Inductive Logic Programming: 21st International Conference, ILP 2011, Windsor Great Park, UK, July 31 – August 3, 2011, Revised Selected Papers*. Ed. by Stephen H. Muggleton, Alireza Tamaddoni-Nezhad, and Francesca A. Lisi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 76–90. ISBN: 978-3-642-31951-8.

[3] Radomír Černoch and Filip Železný. "Probabilistic Rule Learning through Integer Linear Programming". In: *Znalosti 2011, Czech and Slovak Knowledge Technology Conference*. Vol. 801. Workshop Proceedings. CEUR, 2011.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

1 INTRODUCTION

1.1 MECHANICAL LOCKS

There is hardly any technology as old, as trusted and as ubiquitous as mechanical keys and locks. The technology dates back to ancient Egypt [33, 46] and some sources put it even further in history [32]. Highly refined during the industrial revolution with notable improvements by famous Linus Yale, Sr. and Jr. [46], mechanical keys and locks are omnipresent. Given the modern industry, they can be produced in large quantities and for low prices. In other words – if you do not have a mechanical lock at work, your house is probably equipped with one.

Even to this day, mechanical locks have remained somewhat romanticised. It is hard to imagine the Wild West without a safe robbery and even today words like "vault" and "burgled" make newspaper headlines [27]. Lock-picking, the activity of overcoming locks without the proper key, has recently attracted many enthusiasts. Supported by a growing number of books [26, 40, 43], holding regular competitions, supporters practice it as a hobby or a so-called locksport without malicious intents and they emphasise an ethical code [55]. The emotional side of mechanical locks is best illustrated by publication titles such as "The Secret Life of Keys" [53], which carries on the 19th-century tradition of bold statements about "indestructible" and "pick-proof" [46] locks.

Mechanical locks are not only old but also resilient and refusing to die out. Admittedly, the most prominent competitor – the electronic lock – has certain advantages. Computer-equipped locks can, for example, restrict users on a time-scheduled basis or keep track of the users who opened them (or tried to open). Recent advancements [42] fit an electronic lock inside a standardised Euro-profile shell, which reduces the cost of electronic lock deployment. Especially in hotels, where keys are temporary, electronic locks possess a significant advantage.

Nevertheless, the mechanical lock still keeps a higher reliability, probably due to larger and sturdier components. Also, if a metal key is sunk in water, thrown in fire or exposed to freezing temperatures, there is still a good chance it will work. The implication on security is reflected in cheaper insurance of cars equipped with mechanical locks [33]. And finally, there is a prag-

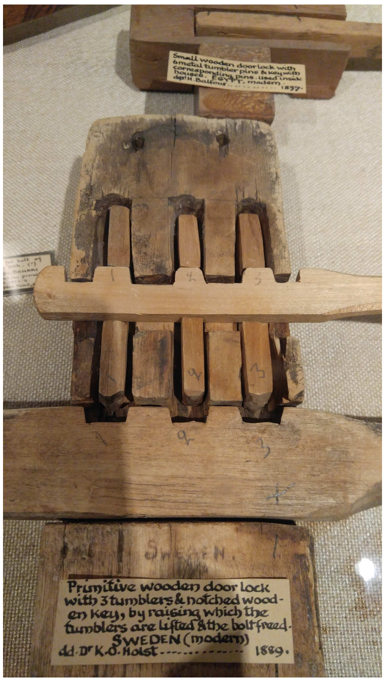

SWEDEN.
Primitive wooden door lock with 3 tumblers & notched wooden key, by raising which the tumblers are lifted & the bolt freed.
SWEDEN (modern)
dd. Dr K. O. Holst .... 1889.

Figure 1.1: Wooden door lock dated 1889 from the Pitt Rivers Museum, Oxford. Photo © Filip Železný.

matic argument. Many use cases do not need a high-tech solution, which makes the low price of mechanical locks a decisive advantage. It is likely that our homes will use mechanical locks for many years to come.

There is also a reason to believe in the future of mechanical locks in the long term. High-security applications might use locks, which are both electronic and mechanical, such as CLIQ [42]. Both components are independent – hence for a key to open a lock, the key's physical shape must be correct as well as the code inside its computer. An attacker attempting to make an illegal copy of a key must copy both components, which requires two different skill sets. The James Bond of the future will have to be a good hacker, merely to open a door.

TUMBLER LOCKS. The fundamental principle of mechanical locks has remained the same for thousands of years. Inside the lock, there is a small movable part, which obstructs the lock's opening. The movable part is hard to reach by hand or by tools but can be pushed out of its place by inserting a correctly shaped key.

Since the industrial revolution, this idea has been realized in many ways. Using [46] we can pinpoint two important types, which are relevant to lock-chart solving. They serve merely as an example, other lock types might be relevant as well.

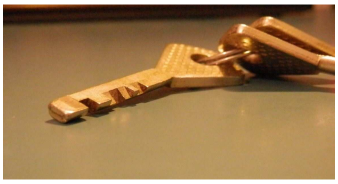


Figure 1.2: Keys for a pin tumbler lock (left) and disc tumbler lock (right). Images are in the public domain [11, 13].

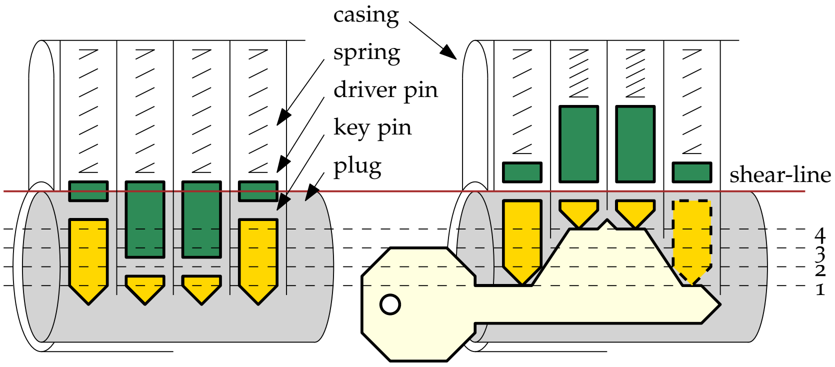


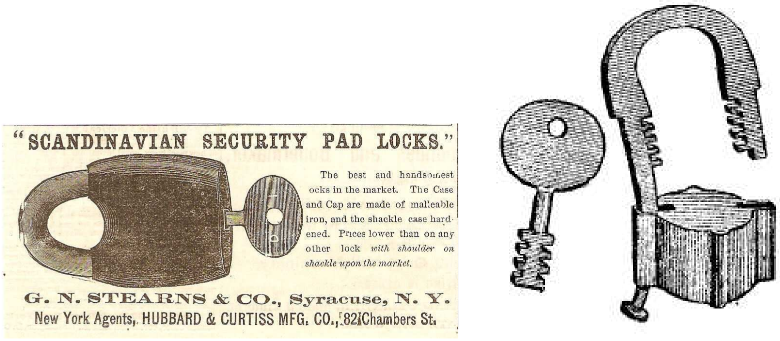
Figure 1.3: Internal components of a tumbler lock with 4 chambers in the locked (left) and unlocked (right) states.

First, there are *pin tumbler locks*, illustrated in Figure 1.3. Pin tumbler locks have several cylindrical holes called *chambers* drilled through the *plug* and the *casing*, in which spring-loaded *pins* can travel up and down. In the locked state (left) the plug cannot rotate in the casing, because *driver pins* are stuck between the two. Inserting a correct key (right) moves the *key pins* as well as the driver pins against the spring. Since all gaps between pins align with the *shear-line*, the key with the plug can be rotated.

Second, there are *disc tumbler locks*. Their mechanism is best explained using the *Scandinavian padlock* (shown in Figure 1.4), in Scandinavia better known as *Polhem padlock* after its inventor. It lacks some features of a modern disc tumbler lock (e.g. locking bars), but that only makes the principle easier to explain. The key has several *facets* aligned with the key's central axis, each rotated at a different angle (see Figure 1.2 on the right). The padlock's *body* contains several circular *discs* with protrusions on its inner perimeter, displayed in Figure 1.5. A key is inserted and starts turning. Facets start hitting the protrusions, each at a different angle and eventually turn the discs. When notches on the discs' outer perimeters are aligned, the *shackle* is released.

A particular type of locks together with the number of chambers, discs, cutting depths, facet angles etc. will be called a *platform*. Platforms come with constraints – either formal or informal rules that must be satisfied by all keys and locks.

Please keep in mind that the terminology differs between manufacturers and languages. A translation table can be found in [46].



“SCANDINAVIAN SECURITY PAD LOCKS.”

The best and handsomest ocks in the market. The Case and Cap are made of malleable iron, and the shackle case hardened. Prices lower than on any other lock *with shoulder on shackle upon the market.*

G. N. STEARNS & CO., Syracuse, N. Y.

New York Agents, HUBBARD & CURTISS MFG. CO., 82 Chambers St.

Figure 1.4: Left: Advertisement for a Scandinavian padlock, year 1874. Right: Drawing of an unlocked Scandinavian padlock. Images are in the public domain [12, 14].

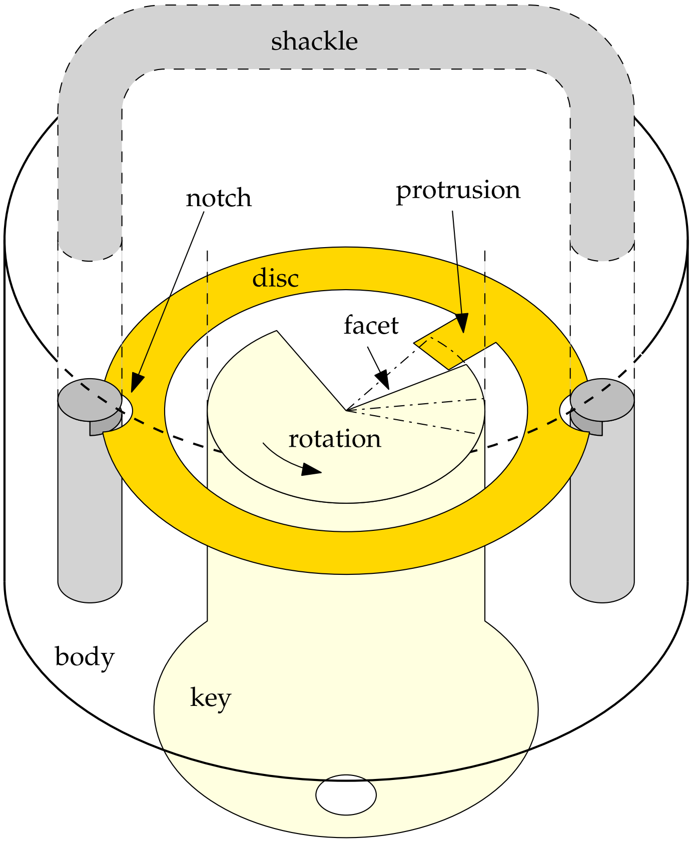


Figure 1.5: Cross section of a simplified Scandinavian padlock through one of its discs illustrates disc tumbler locks.

LOCK-CHARTS. For large buildings such as offices or factories, their owners specify access rights of each person to each room. Typically most users can only open their office, some have access to an entire floor and usually, there is one key which can open every door of the building. This is known as *master keying*, where each lock is opened by one associated key called an *individual* key and also by a limited number of *master* keys. The term is contrasted with *maison keying*, where a single lock is opened by many individual keys in the system [46], which can be used e.g. for the main door of the building.

However, in general, access rights can be arbitrary – a key can open any number of locks and vice versa. Such requirements are encoded in a *lock-chart*. Lock-charts can be visualised in many forms, the simplest of which is a table as in Figure 1.6. Its rows correspond to the locks and columns to the keys. When a key should open a lock, the respective cell in the table contains a tick. Lock-charts tend to have a large number of individual keys, which makes the table hard to visualise on a small screen. Figure 1.7 shows the same lock-chart in a format inspired by the Hlavatý calculation software, developed for the FAB company, now ASSA ABLOY Czech & Slovakia s.r.o. The format only displays locks and master keys. If a lock is opened by an individual key the "I" symbol is printed in the row.

The production of a master keyed system involves several parties. The customer's requirements are encoded in a lock-chart, which is sent to a key manufacturer. They must design the cuttings of the keys and the internal components of the locks so that all constraints and access rights are respected, manufacture them and send them to the customer. We call the process of designing keys and locks *lock-chart solving*. Algorithms for this task are the main contribution of this dissertation.

MASTER KEYED LOCKS. How can two or more keys open a mechanical lock? Ordinary pin tumbler lock in Figure 1.3 opens when gaps between all key pins and driver pins align with the shear-line. By introducing additional driver pins called *spacer pins* as in Figure 1.8, the number of gaps increases. Any combination of gaps from different chambers lets one key enter the lock. Disc tumbler locks can be modified similarly. If the disc's outer perimeter in Figure 1.5 had more notches, the disc could be rotated at multiple angles and still allow the shackle to be released.

Are all lock mechanisms suitable for master keying? The plethora of ideas accumulated over centuries of development makes this a hard question. A comprehensive encyclopedic answer is provided by [46]. However, for the purposes of this text, we consider a platform suitable for master keying if:

	g	m_1	m_2	m_3	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}	k_{11}	k_{12}
l_1	■	■			■											
l_2	■	■				■										
l_3	■	■					■									
l_4	■	■						■								
l_5	■		■						■							
l_6	■		■							■						
l_7	■		■								■					
l_8	■		■									■				
l_9	■			■									■			
l_{10}	■			■										■		
l_{11}	■			■											■	
l_{12}	■			■												■

Figure 1.6: Lock-chart with 12 locks and 16 keys, displayed as a table.

```
        +---------+
        | g m m m |
        |   1 2 3 |
        +---------+
        | M M M M |
+----+-+---------+
|l1  |I| * * . . |
|l2  |I| * * . . |
|l3  |I| * * . . |
|l4  |I| * * . . |
|l5  |I| * . * . |
|l6  |I| * . * . |
|l7  |I| * . * . |
|l8  |I| * . * . |
|l9  |I| * . . * |
|l10|I| * . . * |
|l11|I| * . . * |
|l12|I| * . . * |
+----+-+---------+
```

Figure 1.7: Lock-chart with 12 locks and 16 keys, displayed in a Hlavatý-like format.

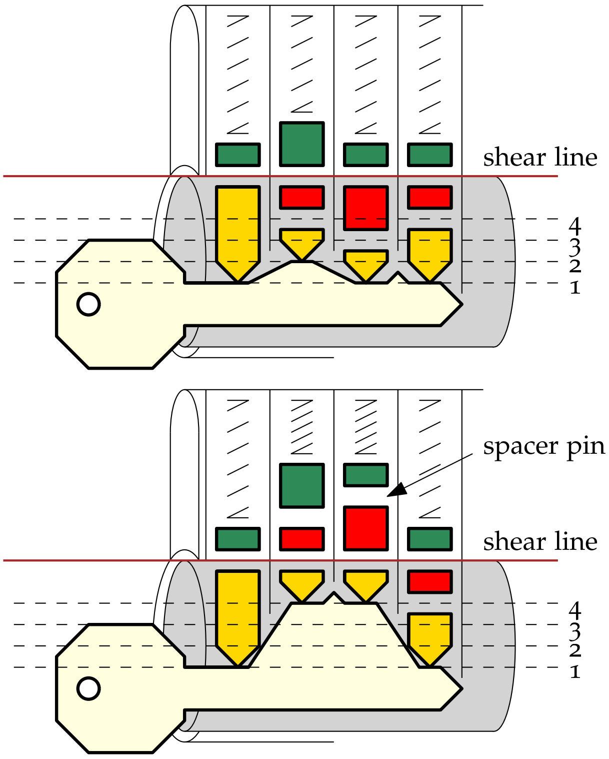


Figure 1.8: Master-keyed pin tumbler lock. Because of the added spacer pins, there are 8 different shear lines, each corresponding to a key cutting that opens the lock. Two of them are shown in the picture: $(1,2,1,1)$ and $(1,4,4,1)$.

> Given an arbitrary set of keys (from one platform), it is possible to manufacture a lock opened (at least) by all of them.

An example of technology outside of this definition is a *wafer tumbler lock*. Master-keyed wafer locks are opened by at most two keys [46]. Furthermore, the key's teeth are shifted either to the left or right. Consequently, no lock can be operated by two different "left" keys or by two different "right" keys, which dramatically reduces the set of lock-charts that can be manufactured. Hence wafer tumbler locks are not considered in this text.

Even platforms suitable for master keying have restrictions. Some master keyed locks are "forbidden", because of higher manufacturing costs or their susceptibility to lock picking. A typical example is an overly small spacer pin in a pin tumbler lock. However, from our experience, the effects of *lock* constraints are less significant than of those applied on *keys*. In this text, we considered only key constraints and assumed that lock constraints do not exist or that their effect can be nullified by a mathematical transformation (see Section 2.2).

On the other hand, our approach embraces many platforms and mechanical lock features. Besides pin tumbler locks and disc tumbler locks, this work is also well suited for *side pins*, both *active* and *passive* [46]. Side pins are similar to chambers in a pin tumbler lock, except they are binary. Usually located on the side of the key, side pin appears either as a dimple or a flat surface. Despite their low contribution to lock-picking protection, for lock-chart solving they behave exactly like a regular chamber.

INNOVATION. It may seem that a centuries-old technology is stable and does not evolve. In reality, the opposite is true.

One of the business promises made to customers is that a key cannot be duplicated without authorisation. Before issuing a duplicate key, customers must typically present a "key cutting ID card", otherwise kept at a safe place. This scheme ensures that a temporarily lost or lent key once returned is still the only existing copy.

How can a manufacturer ensure that no other party starts a business by issuing compatible key blanks and duplicating keys without authorisation? After all, a key cutting ID card has no legal status. However, by filing a patent for a particular platform, a legal protection of the scheme is guaranteed.

After a patent expires (20 years in the United States), the manufacturer is forced to update the platform and file a new patent. Sometimes, the old and new platforms are made incompatible only by changing the external dimensions, but often a chamber or a disc is added [46], which also affects the calculation.

EXTENSIONS. Our industrial partner introduced us to yet another aspect of lock-chart solving. Buildings get remodeled, new floors are added, dividing walls are moved, and all such changes affect lock-charts. For example, when a new room is built, the lock-chart receives a new lock. Or, when someone loses their key, a new key might be added. Such modification to an existing master key system is called an *extension*.

Extensions pose two challenges. First, a lock-chart solving software must anticipate such modifications. When a brand new master key system is calculated, which we call *from-scratch solving*, the keys and locks must be designed in a way to allow new keys and locks to be added in future. The tricky part is that the exact lock-chart with the extension is not known during the initial calculation. The second challenge called *extension lock-chart solving* is to accept a lock-chart with some of its keys and locks fixed during the from-scratch solution.

Our industrial partner stressed the importance of this aspect because, businesswise, the majority of their orders were extensions of already existing master-key systems.

This text focuses on lock-chart solving algorithms that are general, parametrizable, support a significant range of platforms, and are suitable for both the from-scratch and extension tasks.

1.2 RELATED LITERATURE

Despite a clean mathematical formulation of lock-chart solving and the constraints, the problem has not received adequate attention of computer scientists or the industry. From our experience of working together with a major player in the industry, many key manufacturers still solve lock-charts manually, merely with computer-assisted validation of design decisions made by a human operator. Most software we know only solve special cases of lock-charts or require their lengthy manual preprocessing. The underappreciation of lock-chart solving is reflected by a limited number of algorithmic studies.

The first study on lock-chart solving known to us is [34]. Junker used a constraint satisfaction problem (CSP) solver for solving lock-charts. In particular, his main contribution is a procedure, which decomposes a CSP representing a lock-chart into a tree hierarchy of subproblems of smaller size. Since the primal graph of the hierarchy is almost a tree, it can be solved in polynomial time in the size of the tree and in exponential time in the size of the subproblems [22]. The algorithm used the ILOG library and was verified on a set of problems, whose exact structure has not been disclosed.

Lawer's contribution to lock-chart solving in her thesis [38] is manyfold. The main theoretical contribution is a proof of $\mathcal{NP}$-completeness. In particular, the SAT problem was translated into an extension lock-chart. In the practical part, she focused on inter-customer security. Key manufacturers try to prevent keys belonging to one customer from opening locks belonging to another customer (if all produced in the same mechanical platform). Lawer proposed and tested several algorithms that minimise the number of keys unusable for future calculations. Also, the work contains a list of lock-charts used for benchmarking.

The newest contribution [53] to the field contains an explanation of $\mathcal{NP}$-completeness and reports good experience with a simulated annealing algorithm. The work is accessible and might serve as a good introduction. Since the algorithm's description and important claims somewhat lack formal precision, this text is more inspired by the former two works. Readers interested in this resource should be aware that the formalisation of a *key* and a *lock* seem swapped when compared to this text and all other work on lock-chart solving cited here.

Besides the computer science community, lock-chart solving is discussed in training manuals for professionals who solve lock-charts by hand [44]. For example, the description of the *rotating constant method* can be viewed as an executable algorithm. However, we were unable to find any optimality proofs or an analysis of the method's limitations. We decided to include the work here and fit it into our theoretical concepts.

EXTENSIONS. To the best of our knowledge, the extension aspect of lock-chart solving has not been studied before. Hence we also reviewed the literature about general combinatorial problems which are solved without the knowledge of future constraints.

The seminal work on *dynamic CSPs* in [18] attempts to reuse a solution to one CSP in a slightly modified CSP. [28] proposed *super solutions* as a method of preparing a solution for small modifications of the constraints and minimise the number of necessary changes. Other approaches allow using *soft constraints* in the form of *MAX-CSP* [21] or *weighted CSP* [15]. It looks promising to formulate extensibility as an optimisation criterion using a clever reformulation. Our initial attempt was to use existing CSP solvers, but even without the extensibility criterion, an off-the-shelf, state-of-the-art CSP solver Choco [45] proved orders of magnitude slower than the pruning procedure with the heuristics we present here.

1.3 GOALS AND DISCLAIMER

This text was written with several goals in mind:

- Capture our experience from 4-year long industrial collaboration with ASSA ABLOY Czech & Slovakia, ASSA ABLOY Belgium and the ASSA ABLOY EMEA.
- Formalise all known variations of the lock-chart solving problem and establish relationships to existing literature. Find a reasonable compromise between the theory's strength and its assumptions.
- Extend the $\mathcal{NP}$-completeness proof from [38] to non-extension variants of lock-chart solving. Find classes of polynomial instances.
- Find efficient algorithms for lock-chart solving. Evaluate them on a dataset of lock-chart with a focus on real-world problems. Describe means of increasing extensibility.
- Provide an introduction to lock-chart solving for computer scientists without a background in mechanical engineering.

We stress out topics which are not part of this text:

- Non-disclosure agreements with our partners were honoured. We tried to generalise our knowledge and provide theoretical insight without details about the particular technology, constraints or security measures.
- Despite the fact that some constraints affect lock-chart solving and they also affect resistance to lock-picking, we do not establish any relationship between the two here. Here, we assume that experts design all security measures and formulate their implications as constraints for lock-chart solving.
- Conversely, this text is not meant to help lock-pickers. If any property of an algorithm or an insight presented here leads to a new lock-picking technique (which we doubt), it should be disclosed publicly following the industry's best practices [46].
- Data structures for algorithms' efficient implementation are out of scope together with an analysis of asymptotic time complexity. First, most of the problems considered here are $\mathcal{NP}$-complete and entail an exponential runtime (based on the current knowledge) that hides effects of most efficient data structures. Second, computational aspects of lock-chart solving are too little explored. We felt that a broad exploration of different algorithms should precede fine-tuning of their implementation.

1.4 ACKNOWLEDGEMENTS

I would like to thank ASSA ABLOY for a long-term collaboration and for funding research, which has lead to publishing this text.

Several ideas in this text are products of teamwork, usually springing out of a lively discussion with colleagues. Chronologically, the CSP algorithm in Section 6.2 was suggested and first implemented by Ondřej Kuželka. The idea of using SAT solvers described in Section 4.2 and the first prototype shall be credited to Jiří Vyskočil. Several optimisations by Josef Hájíček and Martin Hořeňovský affected the code used here. The dynamic programming scheme in Section 5.2 came out of the collaboration with Václav Voráček.

I am also grateful to my supervisor, who was always there if the project needed a new direction to continue.

My biggest thanks belong to my partner and my son, who made the greatest sacrifice. Without their support and time, this text would never have been created.

2 LOCK-CHART FORMALISATION

The previous text provided an introduction to mechanical keys and lock-charts. This chapter will formalise these notions and give a mathematical model. The formalisation will be used as the input and output of a lock-chart solving algorithm.

2.1 PRELIMINARIES

This section is a quick review of mathematical notions used in the remaining text. Since all notions are within a common computer science curriculum, we tried to keep the definitions short and provided references for further investigation. Readers are encouraged to use this section merely as a "knowledge checklist".

Although some notions are complex enough to write a book, we have tried to find a reasonable compromise. For the reason of low relevancy rather then author's ignorance, we will not discuss the paradoxes of set theory, nor avoid them by axiomatization. Here, we define sets in a narrower sense.

SETS. Let there be finitely many symbols $d_1, \dots, d_n$. The collection $\{d_1, \dots, d_n\}$ is a *set*. Deleting arbitrary items from a set A yields a *set* B, (written $A \subseteq B$) called a *subset*. Sets are written with capital letters $A, B, C, \dots$, its items are referred as *members* of the set ($a \in A$) and the number of items in a set is its *cardinality*, written as $|A|$. An *empty set* $\emptyset$ is a set, whose cardinality is 0. If B is a subset of A and $|A| > |B|$, then B is a *proper subset* of A, written $B \subset A$. Given two sets A, B, their intersection, union and difference are $A \cap B$, $A \cup B$ and $A \setminus B$ respectively.

Given n sets $T_1, \dots, T_n$, a collection of n ordered items $t_1 \in T_1, \dots, t_n \in T_n$ is called a *tuple* $t = (t_1, \dots, t_n)$ of *arity* n. If $n = 2$, the tuple is a *pair*. The *cartesian product* $T_1 \times \cdots \times T_n$ is a set of all tuples on $T_1, \dots, T_n$. A subset of a cartesian product is called a *relation*. If $n = 1$, the relation is *unary* and if $n = 2$, the relation is *binary*.

Let R be a relation on $D \times D$. For all $d, d', d'' \in D$: If no $(d, d) \in R$, then R is *irreflexive*. If $(d, d) \in R$, then R is *reflexive*. If $(d, d') \in R$ and $(d', d) \in R$ implies $d = d'$, then R is *antisymmetric*. If $(d, d') \in R$ and $(d', d'') \in R$ implies $(d, d'') \in R$, the relation is *transitive*. A reflexive, antisymmetric and transitive relation is a *partial order*. The *inverse* relation R^{-1} is defined as $(d, d') \in R$ iff $(d', d) \in R^{-1}$.

FUNCTIONS. Let D and R be sets (called *function's domain* and the *function's range*). A *partial function* $f : D \rightharpoonup R$ is a binary relation on $D \times R$, which has the *right-unique* property — if $(d, r_1) \in f$ and $(d, r_2) \in f$ then $r_1 = r_2$. Furthermore, a partial function is a *function* $f : D \to R$ if it is *left-total* — if $d \in D$ then there must be $r \in R$ s.t. $(d, r) \in f$. A function is *surjective* if has the *right-total* property — if $r \in R$ then there must be $d \in D$ s.t. $(d, r) \in f$. A function is *injective* if it has the *left-unique* property — if $(d_1, r) \in f$ and $(d_2, r) \in f$ then $d_1 = d_2$. A function, which is both surjective and injective, is a *bijection*. Throughout the text, the standard notation $f(d) = r$ is used instead of $(d, r) \in f$ and also a less-standard notation $f \cup (d, h)$ to define an extension of a partial function f (partial functions are sets of pairs).

GRAPHS. *Graph* is a pair (V, E) of *vertices* and *edges*, where E is a binary relation on V. We say that vertex v' is *adjacent* to v if there is an edge from one vertex to the other $(v, v') \in E$. The graph is *simple* if E is irreflexive. Vertices adjacent to v are denoted $E(v)$. The *degree* of vertex v is $|E(v)|$. A graph is *undirected* if E is a symmetric relation: if $(v, v') \in V$, then $(v', v) \in V$. A graph is bipartite with *partite sets* U and U' if $U \cap U' = \emptyset$, $U \cup U' = V$ and $E \subseteq (U \times U') \cup (U' \times U)$, which will be written as $(U \cup U', E)$.

INTEGERS. The set of integers is $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, the set of whole numbers is $\mathbb{W} = \{0, 1, 2, 3, \ldots\}$ and the set of natural numbers $\mathbb{N} = \{1, 2, 3, \ldots\}$. Addition, subtraction, multiplication, quotient and remainder of natural numbers x, y will be denoted $x + y$, $x - y$, $x \cdot y$, $x \div y$ and $x \,\%\, y$. By $\frac{x}{y}$ we mean a division of two rational numbers.

2.2 CONSTRAINTS

The combinatorial problem of lock-chart solving consists of 2 parts. First, there is the lock-chart, which is specified by the customer. Second, there are some constraints, which capture the engineering limitations specified by the manufacturer. We start by describing the constraints in this section.

During the industrial collaboration, we met 12 mechanical platforms with roughly 20 types of constraints of various types. Together they make up a big „zoo“, which is hard to analyse as a whole. There are roughly two taxonomies of such constraints.

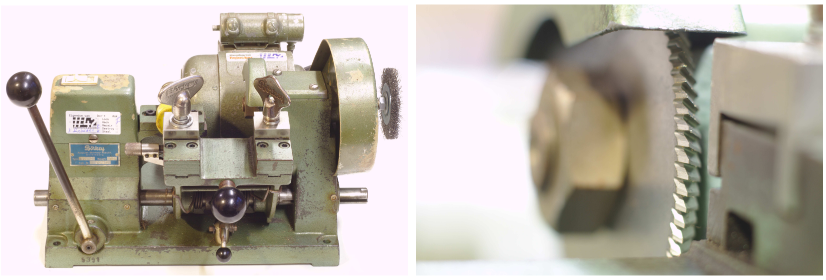


Figure 2.1: Börkey 954-2 Key Cutting Machine with a close-up of the milling wheel. © Dennis van Zuijlekom, CC BY-SA 2.0 [57].

SECURITY AND MANUFACTURING. There is an intuitive distinction between *security* and *manufacturing* constraints. An example of a manufacturing constraint is the "maximum difference between neighbouring positions on a key", which we call the *jump*. Figure 2.1 shows a detail of a milling machine. Its milling wheel is V-shaped, where the deepest cutting point cuts the desired position into a flat key blank. However, the shallower parts of the milling wheel might still cut too deeply in neighbouring positions. As a countermeasure, the jump constraint ensures that neighbouring positions still have enough metal to be milled properly.

An example of a security constraint is the *delta* constraint, which prescribes the "minimum difference of key's cutting depth from the lock's shear-line" if the key should be blocked in the lock.[1] Increasing the delta lowers the chances of an accidental opening, which makes it a security constraint.

Increasing the delta constraint also affects manufacturing. Assuming delta $= 3\,\text{mm}$, it makes no sense to manufacture pins of size $2\,\text{mm}$ or smaller, because they would be unable to block a key. Hence, the delta is often accompanied with the *forbidden pins* constraint, which "forbids prescribed pin heights from a lock".[2]

Consequently, increasing delta also affects manufacturing by increasing components' sizes and reducing the need for tight tolerances. The distinction between security and manufacturing constraints is used in the industry, but form the computational point of view, it is not very crisp.

COMPUTATIONAL CONSTRAINTS. Another distinction can be made from the formal point of view. Constraints can

1. affect a single key; e.g. the *jump*

[1] If the key should open the lock, then delta must be 0.

[2] A strong heuristic is to forbid pins, whose size is "not modulo delta" — in this case, pins of size $1\,\text{mm}$, $2\,\text{mm}$, $4\,\text{mm}$, $5\,\text{mm}$, $7\,\text{mm}, \ldots$ An even simpler heuristic is to ignore certain cuttings depths completely and reduce the entire code space.

2. affect a single lock; e.g. *forbidden pins*
3. affect the blocking between a key and a lock; e.g. the *delta*
4. affect a pair of keys

In this text, we decided to focus on the first type only. First, the example above shows that types 2 and 3 can be closely related or they can even be eliminated by reducing the code space. Moreover, out of the 20 types of constraints we know, the majority (15) is of the first type.

Finally, we know of only one constraint of the fourth type, namely "minimum difference of two key cuttings". Because it is almost always set to value 1, later in this section will show how to satisfy it as a by-product of Corollary 26.

TWO TYPES OF PLATFORMS. Different mechanical platforms call for different formalisms. On one side, there are old and low-cost platforms. Locks in these platforms have a small number of chambers, typically 5 or 6, and their components are manufactured with high tolerances. As an effect, lock's reliability must be increased by prescribing a large number of constraints. Even though these platforms can have up to 10 cutting depths, a high value of the delta constraint decreases this number to $3, 4$ or 5. In the end, these platforms can be characterized by $\sim 10^6$ or less keys that satisfy all constraints.

On the other side, locks in modern platforms typically have a large number of chambers. This value is usually larger than 10, the largest platform known to us has 30 chambers. Cheaper platforms may have a low number of cutting depths, say 2, 3 or 4, but some have 7. Tight tolerances allow delta $= 1$. In the end, these platforms have $\sim 10^7$ or more valid key cuttings.

The effect on computation can be dramatic. In older platforms, current computers can easily generate all key cuttings and keep them in memory. Even with 2 bytes per chamber, this requires roughly $12 \cdot 10^5\,\text{B} \cong 1\,\text{MiB}$ for all valid key codes. In newer platforms, it may not even be feasible to iterate through all valid key cuttings.

CUTTINGS AND CYLINDERS. This section will describe different ways of formalising constraints for calculating lock-charts, called a *framework*. Each framework will be associated with 2 numbers – the *number of positions* p, which counts the number of chambers, side pins, etc. and the number of *cutting depths* d, which counts available values that each position can take.

Definition 1 (Cutting, cylinder)**.** Let $p, d \in \mathbb{N}$. A *cutting depth* is a number $d_i \in \mathbb{N}$ s.t. $d_i \leqslant d$. A key *cutting* is a p-tuple $(d_1, \ldots, d_p)$

of cutting depths. A *cylinder* is a p-tuple $(D_1, \dots, D_p)$, where each D_i is a set of cutting depths.

From now on, we will distinguish between a key and a cutting and similarly between a lock and a cylinder. Cuttings and cylinders will represent physical metal objects. Keys and locks represent rows and columns in the lock-chart.

The definition of cuttings and cylinders requires p and d. Since every framework comes with the two numbers, we will omit them whenever they become obvious from the context.

A cutting will be denoted by γ and a cylinder by λ. The i-th cutting depth (resp. i-th set of cutting depths) will be γ_i (resp. λ_i). By a *union* of cylinders or cuttings, we mean a cylinder, whose cutting depths are a union of cutting depths in the respective positions. Example:

$$(1,3,2) \cup (\{1,2\},\{1\},\{2\}) = (\{1,2\},\{1,3\},\{2\})$$

Definition 2 (Shear-line)**.** Let λ be a cylinder. A cutting γ which satisfies $\gamma_i \in \lambda_i$ for all $1 \leqslant i \leqslant p$ is a *shear-line* of γ.

If γ is a shear-line of λ we will say that "γ *enters* λ". Otherwise "λ *blocks* γ".

Given p and d, there are at most d^p cutting depths. However, in general, not all of them satisfy the constraints. For that reason, each framework will be associated with a set of cuttings S called a *code space*. If $\gamma \in S$, we will say that "γ is a *valid* cutting".

We will also speak about the set of all cylinders T. Since constraints on cylinders are not in the scope of this text, any union of cuttings yields a valid cylinder

$$T = \left\{ \bigcup_{\gamma \in S'} \gamma \mid \text{for every } S' \subseteq S \right\}, \tag{2.1}$$

so that there are at most $2^{d \cdot p}$ different cylinders.

FOUR FRAMEWORKS. The simplest constraint framework, which is mostly of theoretical interest, is defined first and then we proceed to more complex ones, aiming at describing real-world platforms.

Definition 3 (Vanilla framework)**.** Let $p, d \in \mathbb{N}$. In the *vanilla* framework, all d^p cuttings are valid.

The vanilla framework has two major advantages. First, the code space is perfectly symmetric. Swapping two positions has no effect on the code space as well as swapping arbitrary cutting depths. Second, the code space is defined using two parameters only. This makes it suitable for benchmarking and easy-to-read plots.

How well do real-world platforms fit in the vanilla framework? During our industrial collaboration, we met only one platform, which has approx. 97% of d^p valid key cuttings,[3] hence the answer would be "almost". The encyclopedia of lock-chart platforms [46] mentions platforms (such as Mul-T-Lock) without constraints, merely with a varying number of cutting depths between positions. A step towards real-world platforms is to allow different positions to have a different number of cutting depths.

Definition 4 (Asymmetric framework)**.** Let $p, d \in \mathbb{N}$. The *deepest cutting* is a cutting denoted $(\tilde{d}_1, \dots, \tilde{d}_p)$. A cutting $(d_1, \dots, d_p)$ is valid in the *asymmetric framework* if all its cutting depths are below the deepest cut: $d_i \leqslant \tilde{d_i}$ for all $1 \leqslant i \leqslant p$.

The asymmetric framework needs $p + 1$ parameters to define the deepest cutting. When compared to the vanilla framework, the symmetry is somewhat reduced. Swapping two cutting depths on the same position is fine, swapping two positions i and j can be done as long as $\tilde{d}_i = \tilde{d}_j$.

In order to completely specify most real-world platforms, yet avoid the complexity of its constraints, the general framework is proposed.

Definition 5 (General framework)**.** Let $p, d \in \mathbb{N}$. A *general constraint* (gecon) is a p-tuple $(c_1, \dots, c_p)$, where either c_i is a cutting depth or $c_i = ?$ called a *wildcard*. A key cutting $(d_1, \dots, d_p)$ satisfies the gecon if there is a position i, which is not a wildcard $c_i \neq ?$ and where $c_i \neq d_i$. A key cutting is valid in the *general framework* if it satisfies all gecons.

Example 6. Let $p = 3$, $d = 5$ and assume there is a gecon $(1, 4, ?)$. Is the cutting $(1, 5, 5)$ valid? Yes, because on the second position $i = 2$, the cutting depth $d_2 = 5$ does not match the depth in the gecon $c_2 = 4$. The conditions $c_i \neq d_i$ and $c_i \neq ?$ are satisfied. Also all invalid cuttings are $(1, 4, 1)$, $(1, 4, 2)$, $(1, 4, 3)$, $(1, 4, 4)$ and $(1, 4, 5)$.

The intuitive meaning of the gecon from the example is: "First and second positions cannot have depths 1 and 4 respectively." The wildcard symbol ? means "any value". Hence a gecon without wildcards makes exactly 1 cutting invalid. A gecon with wildcards on all positions invalidates all cuttings, which is why it is rarely used.

[3] Other platforms had less than that, roughly between 1% and 50%.

Why do gecons have the word "general" in their name? Most constraints that we met can be translated into gecons.

Example 7. Let $p = 3$, $d = 5$ and take the jump $= 2$ constraint, which can be rephrased as "difference of neighbouring positions larger or equal than 3 is forbidden". This constraint can be expressed using 6 gecons $(1, 4, ?)$, $(1, 5, ?)$, $(2, 5, ?)$, $(?, 1, 4)$, $(?, 1, 5)$ and $(?, 2, 5)$.

In general, the jump constraint is equivalent to

$$\frac{1}{2} \cdot \text{jump} \cdot (\text{jump} + 1) \cdot (p - 1)$$

gecons. The number of gecons is polynomial, which is also true for most of the 15 constraints we know.

Software engineers have a proverb that the lengthiest task is not to develop the software, but to make the customer realize his/her expectations. In lock-chart solving, the same applies for obtaining a complete, non-ambiguous specification of constraints.

It helps a lot if the manufacturer provides a list of all valid key cuttings in a text file. When it is the case, some algorithms can use the list directly.

Definition 8 (Explicit framework)**.** In the *explicit framework*, the code space S is a part of the algorithm's input.

This framework is suited for older platforms, where $|S|$ is a feasible number for the current generation of computers.

HIERARCHY. There is a certain hierarchy among the 4 frameworks. The vanilla framework is a special case of the asymmetric framework. By setting the deepest cutting $(\tilde{d}_1, \ldots, \tilde{d}_p) = (d, \ldots, d)$, both frameworks generate the same code space.

The asymmetric framework is a special case of the general framework. Start from the deepest cutting $(\tilde{d}_1, \ldots, \tilde{d}_p)$ and for every positions $1 \leqslant i \leqslant p$ and every forbidden cutting depth $\tilde{d}_i < j \leqslant d$, generate 1 gecon:

$$(\underbrace{?, ?, \ldots, ?}_{i-1}, j, \underbrace{?, \ldots, ?}_{p-i}) \tag{2.2}$$

This gecon effectively forbids the cutting depth j at position i. Also, note that there are polynomially many such gecons; no more than $d \cdot p$.

The relationship between the general and the explicit frameworks is not that obvious. A brute force algorithm may iterate over all

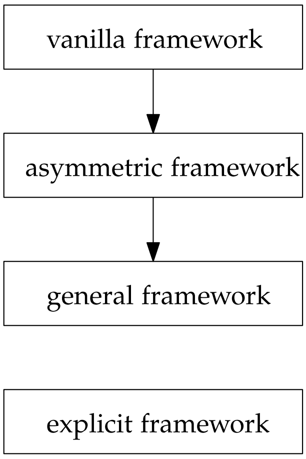


Figure 2.2: Hierarchy of constraint frameworks. An arrow points from a special case to a more general framework.

d^p cuttings, check if the cutting is in S or check against all gecons. However, this procedure is not polynomial in p. As we will see later, sub-exponential algorithms do not exist.[4]

Figure 2.2 concludes this section by a schema of frameworks' hierarchy, which will be referred in the future chapters.

2.3 LOCK-CHARTS

This section formalises the notion of a lock-chart. Their purpose is to define the computational task together with a constraint framework. A solution will be defined w.r.t. to a lock-chart.

The catch is that different manufacturers have different practices, which lead to different calculation procedures. Hence here we propose multiple definitions of "the lock-chart" together with their relationships, which will be used to analyse the computational complexity.

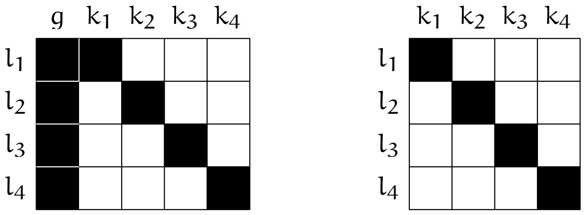
Definition 9 (Basic lock-chart)**.** A *basic lock-chart* (K, L, E) consists of a *key-set* K, a *lock-set* L, which are disjoint $K \cap L = \emptyset$ and a binary relation E on $K \times L$ called an *edge-set*.

Since we will speak about basic lock-charts most of the time, they will be referred simply as "lock-charts" if not stated otherwise.

There is an parallel between lock-charts and graphs [38]. Every lock-chart (K, L, E) is by definition an undirected bipartite graph $(K \cup L, E \cup E^{-1})$ with *partite sets* K and L. This simplifies the terminology. Incident vertices are denoted as

$$E(k) = \{l \in L \mid (k, l) \in E\},\ E(l) = \{k \in K \mid (k, l) \in E\},$$

[4] This holds unless $\mathcal{P} = \mathcal{NP}$.



	g	k_1	k_2	k_3	k_4
l_1	■	■			
l_2	■		■		
l_3	■			■	
l_4	■				■

	k_1	k_2	k_3	k_4
l_1	■			
l_2		■		
l_3			■	
l_4				■

Figure 2.3: Diagonal lock-chart (left) and key-to-differ lock-chart (right) with 4 individual keys.

If $(k, l) \in E$, then we say that "k *opens* l", otherwise "l *stops* k".

Example 10. Figure 1.6 shows a basic lock-chart. Keys are in columns, locks in rows. If the cell is black, the row-lock opens the key-column. The lock-chart is formalised as follows:

$$K = \{g, m_1, m_2, m_3, k_1, k_2, \dots, k_{12}\}, \; L = \{l_1, l_2, \dots, l_{12}\}\,,$$

$$E = \{(g, l_1), (g, l_2), \dots, (m_1, l_1), (m_1, l_2), \dots, (k_1, l_1), (k_2, l_2), \dots\}\,.$$

By the defined notation, the following statements hold: $E(g) = L$, $E(m_1) = \{l_1, l_2, l_3, l_4\}$ and $E(k_i) = E(l_i)$ for $1 \leqslant i \leqslant 12$.

Definition 11 (General, master and individual keys)**.** Let K be a key-set, L a lock-set, E an edge-set and $k \in K$. If $|E(k)| = 1$ then k is an *individual key*. If $|E(k)| > 1$ then k is a *master key*. If $|E(k)| = |L|$ then k is the *general key*.

Lock-chart in Figure 1.6 has the general key g, 4 master keys g, m_1, m_2, m_3 and 12 individual keys k_1 to k_{12}.

Perhaps the simplest class of lock-chart with the general key are diagonal lock-charts. Each is determined by a single parameter — the number of individual keys.

Definition 12 (Diagonal lock-chart)**.** A lock-chart with 1 general key and no other master key is a *diagonal* lock-chart.

Figure 2.3 left shows a diagonal lock-chart. It has 5 keys $K = \{g, k_1, \dots, k_4\}$ and 4 locks $L = \{l_1, \dots, l_4\}$. Each individual key $k_1, \dots, k_4$ opens exactly 1 lock $E(k_i) = \{l_i\}$, the general key g opens all locks $E(g) = L$.

If the general key is removed from a diagonal lock-chart, we get an even simpler lock-chart. Such lock-charts are occasionally used in practice, nevertheless they play a major role in the theoretical analysis.

Definition 13 (Key-to-differ)**.** A lock-chart without master keys is a *key-to-differ lock-chart*.

An example of a key-to-diff lock-chart is shown in Figure 2.3 on the right.

SOLUTIONS. Having formalised the input to a calculation algorithm, next we specify its output.

Definition 14 (Assignment)**.** Let K be a key-set, L a lock-set and S a code space. *Key-assignment* is a function $s : K \to S$ that assigns cuttings to keys and *lock-assignment* is a function $t : L \to T$ that assigns cylinders to locks. An *assignment* is a function $s \cup t$. We speak about a *partial assignment* if any of the two functions s or t is partial.

By inspecting each cell in the lock-chart, we check the correctness of an assignment.

Definition 15 (Solution)**.** Let (K, L, E) be a lock-chart and $s \cup t$ a (partial) assignment. If for every $(k, \gamma) \in s$ and for every $(l, \lambda) \in t$

$$\gamma \text{ enters } \lambda \text{ if and only if } k \text{ opens } l\ , \tag{2.3}$$

then $s \cup t$ is a *(partial) solution*.

For clarity, partial assignments and solutions will be denoted as $\hat{s}$ unless stated otherwise.

A possibly non-trivial finding is that a lock-assignment can be ignored by a deterministic extension of a key-assignment.

First note that the more cutting depths a cylinder has the more shear-lines it contains. There are exactly $|D_1| \times \cdots \times |D_p|$ shear-lines in a cylinder $(D_1, \ldots D_p)$. This motivates a method to generate cylinders that block as many cuttings as possible.

Proposition 16 (Least-cut)**.** *Let (K, L, E) be a lock-chart and $s \cup t$ its solution. Then the assignment $s \cup t'$, where*

$$t'(l) = \bigcup_{k \in E(l)} s(k) \tag{2.4}$$

is also a solution.

Proof. Take any lock l. In order to satisfy key openings,

$$t'(l) \subseteq \bigcup_{k \in E(l)} s(k) \subseteq t(l)\ . \tag{2.5}$$

Since the blocking ability only weakens with additional cutting depths, each set $t'(l)_i$ is as small as possible. □

Hence further in the text, when a key-assignment $s : K \to S$ is referred as a "solution", we speak about a solution $s \cup t'$, where t' is defined by (2.4). When it is the case, we took the liberty of writing $s(l)$ for $l \in L$, which really means $t'(l)$, just to avoid overly complex notation.

Example 17. Suppose that the diagonal lock-chart from Figure 2.3 is assigned cuttings as follows: $s(g) = (1, 1, 1, 1)$, $s(k_1) = (1, 1, 2, 2)$, $s(k_2) = (1, 2, 2, 1)$, $s(k_3) = (2, 2, 1, 1)$, $s(k_4) = (2, 1, 1, 2)$. Then the locks become

$$\begin{aligned} t'(l_1) &= (\{1\}, \{1\}, \{1, 2\}, \{1, 2\}) \\ t'(l_2) &= (\{1\}, \{1, 2\}, \{1, 2\}, \{1\}) \\ t'(l_3) &= (\{1, 2\}, \{1, 2\}, \{1\}, \{1\}) \\ t'(l_4) &= (\{1, 2\}, \{1\}, \{1\}, \{1, 2\}) \end{aligned}$$

The assignment is a solution: Key k_2 is blocked in l_1 in the 2nd position, because $s(k_2)_2 = 2$ is a cutting depth not present in $t'(l_1)_2 = \{1\}$. Similarly, key k_3 is blocked in l_1 in the 1st and 2nd position and key k_4 only in the 1st position. The same is true for locks $l_2, \dots, l_4$.

Note that the proposition can reduce the number of available cylinders T. It is no longer necessary to consider all combinations of all cuttings as in (2.1). The union in (2.4) iterates over $E(l)$, hence every lock l is assigned a cylinder, which is a union of at most $|E(l)|$ different cuttings.

Remark 18. Let (K, L, E) be a lock-chart and S a code space. Every lock-assignment t satisfies $t \subseteq L \times T'$, where

$$T' = \left\{ \bigcup_{\gamma \in S'} \gamma \mid \text{for every } S' \subseteq S \text{ s.t. } |S'| \leqslant \max_{l \in L} |E(l)| \right\}. \tag{2.6}$$

Can the proposition be applied to partial key-assignments $\hat{s} : K \rightharpoonup S$ as well? Adapting equation (2.4) needs only a cosmetic modification

$$t'(l) = \bigcup_{k \in E(l) \text{ and } \hat{s}(k) \text{ is defined}} \hat{s}(k)\,, \tag{2.7}$$

and the definition of partial solutions still holds.

Observe that using (2.7), a partial key-assignment $\hat{s}$ yields a lock-assignment t', which is not partial. Even if $\hat{s}$ is an empty function, the lock-assignment prescribes an empty cylinder to all locks $t'(l) = (\emptyset, \dots, \emptyset)$.

EXTENSION LOCK-CHARTS. An extension is a naturally occurring industrial problem when a customer orders master-key-systems in multiple batches. Each batch adds new keys or locks into the lock-chart, yet the solution to the original lock-chart remains fixed because it has already been manufactured and shipped.

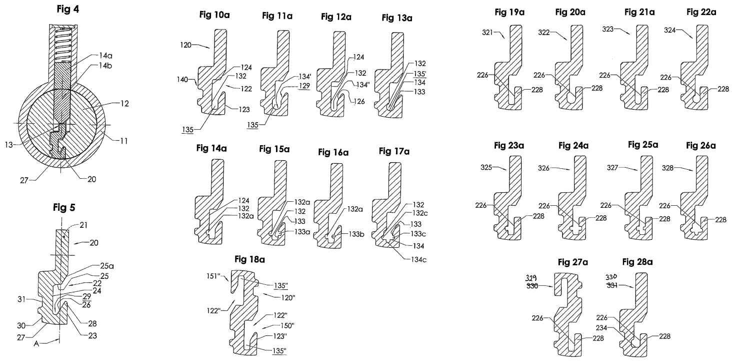


Figure 2.4: A physical profile on a cutting and in a cylinder (left) and key-profiles from a profile map (right).
Source: patent US 2011 0271723 A1 [54]

Definition 19 (Extension lock-chart)**.** Let (K, L, E) be a lock-chart and $\hat{s} \cup \hat{t}$ its partial solution. The *extension lock-chart* is $(K, L, E, \hat{s} \cup \hat{t})$. A (partial) assignment $s \cup t$ is a *(partial) solution* of $(K, L, E, \hat{s} \cup \hat{t})$ if $\hat{s} \subseteq s$, $\hat{t} \subset t$ and $\hat{s} \cup \hat{t}$ is a (partial) solution of (K, L, E).

Extension lock-charts with $\hat{s} = \hat{t} = \emptyset$ are called *from-scratch lock-charts*. Therefore basic lock-charts are a special case of extension lock-charts in the following sense: A assignment s of a basic lock-chart (K, L, E) is a solution if and only if it is a solution of the from-scratch lock-chart $(K, L, E, \emptyset)$.

PROFILE MAPS. Some mechanical platforms contain profiles and profile maps. A profile is a particular shape of a flat key's cross section or the keyway in a lock. Figure 2.4 shows examples of profile maps. Profiles can ensure that some flat keys cannot be physically inserted into certain keyways so that the blocking does not have to be done by pins, discs or any other elements inside the cylinder. This alleviates the number of blockings a cylinder must ensure, hence enlarging the capacity of the lock-chart.

Each key and lock in a lock-chart gets assigned one profile. The general key is usually assigned the profile with the least amount of metal. Like that, it enters all cylinders in the lock-chart. Typically, other master keys have more metal and individual keys have the most. This implies a reverse hierarchy among locks. Locks opened merely by a few master keys will be assigned profiles with more metal and less air than locks opened by individual keys.

Definition 20 (Profile map)**.** A *profile map* $(P, \preceq)$ consists of a set of *profiles* P and a partial order $\preceq$ on P. *Profile assignment* is a function $a : (K \cup L) \to P$, which assigns profiles to keys and locks.

Let be a profile map $(P, \preceq)$ and p and p' be profiles in P. If $p \preceq p'$ we say that "p is *above* p'" or "p' is *below* p". If $p \not\preceq p'$ and $p' \not\preceq p$ we say that "p and p' are *independent*".

Example 21. Figure 2.5 on the left shows a profile map with 5 profiles $P = \{1, 2, 3, 4, 5\}$. Arrows pointing from p to p' means $p \preceq p'$ (transitive and reflexive tuples in $\preceq$ are hidden for clarity). Profiles $2, 3$ and 4 are all pair-wise independent. No other two profiles are independent.

The strategy for dealing with profile maps varies. Some manufacturers let the software find a profile assignment, some hand-code it manually for better control of future extensions. It is not hard to formalise the first approach. The algorithm would have to find a solution s and a profile assignment a as its output.

There are two reasons why we focused on the second approach. First, there is the practical experience. We have dealt mostly with lock-charts, whose profiles have been hand-coded, and hence our knowledge of efficient methods for finding profile assignments is not very deep. Second, ideas that will be presented in the next chapter require a fixed profile assignment.

Assuming that a is fixed, the definition of the combinatorial problem can be simplified. Given an assignment, some blockings are ensured by means of profiles, not necessarily by pins in chambers.

Definition 22 (Profiled lock-chart)**.** Let (K, L, E) be a lock-chart chart, $(P, \preceq)$ a profile-map and $a : (K \cup L) \to P$ a profile assignment. A *profiled lock-chart* is a tuple $(K, L, E, P, \preceq, a)$. Let $s \cup t$ be an assignment. If for every $(k, \gamma) \in s$ and $(l, \lambda) \in t$

$$k \text{ opens } l \text{ if and only if } \gamma \text{ enters } \lambda \text{ and } a(k) \preceq a(l) \tag{2.8}$$

then $s \cup t$ is a *(partial) solution* of $(K, L, E, P, \preceq, a)$.

The definition might be clearer by rewriting (2.8) into an equivalent form using De Morgan's law:

$$l \text{ stops } k \text{ if and only if } \lambda \text{ blocks } \gamma \text{ or } a(k) \not\preceq a(l) \tag{2.9}$$

Basic lock-charts are a special case of profiled lock-charts in the following sense: An assignment s to a basic lock-chart (K, L, E) is a solution if and only if it is a solution to a profiled lock-chart $(K, L, E, \{p\}, \{(p, p)\}, (K \cup L) \times \{p\})$ with a single-profile profile map. This follows from $a(k) = p \preceq p = a(l)$ for all keys and locks and hence conditions (2.3) and (2.8) overlap.

Also note an immediate consequence of (2.8). For every lock l opened by some key k, the k's profile must be above the l's profile: $a(k) \preceq a(l)$; otherwise the lock-chart has no solution. Profile assignments that satisfy this condition are called *well-formed* *profile assignments*. Since other profile assignments are meritless, from now on we will assume that all profile assignments are well-formed.

Because of the "or" in (2.9), in a profiled lock-chart each key-lock pair (k, l) falls into 3 cases:

1. Either k opens l,
2. or k is blocked in l by means of profiles $a(k) \not\preceq a(l)$,
3. or k is blocked in l not by profiles $a(k) \preceq a(l)$.

Example 23. Let the lock-chart from Figure 1.6 be assigned profiles from the profile map in Figure 2.5 left as follows: $a(g) = 1$, $a(m_1) = a(k_i) = a(l_i) = 2$ for $1 \leqslant i \leqslant 4$, $a(m_2) = a(k_i) = a(l_i) = 3$ for $5 \leqslant i \leqslant 8$ and $a(m_3) = a(k_i) = a(l_i) = 3$ for $9 \leqslant i \leqslant 12$. Then Figure 2.5 on the right visualises the lock-chart, whose case 1 cells are filled black, case 2 are filled grey and case 3 are white.

These 3 cases of a profiled lock-chart's cells are reflected in the last definition of a lock-chart.

Definition 24 (Melted profiles lock-chart)**.** The *melted profiles lock-chart* (K, L, E, B) consists of a key-set K a lock-set L and two edge-sets E, B s.t. $K \cap L = \emptyset$ and $E \cap B = \emptyset$. Let $s \cup t$ be a (partial) assignment. If for every $(k, \gamma) \in s$ and $(l, \lambda) \in L$ these conditions hold:

$$\begin{aligned} &\text{if } (k,l) \in E \quad \text{then } s(k) \text{ enters } t(l) \text{ and} \\ &\text{if } (k,l) \in B \quad \text{then } t(l) \text{ blocks } s(k)\ , \end{aligned} \tag{2.10}$$

then $s \cup t$ is a *(partial) solution* to (K, L, E, B).

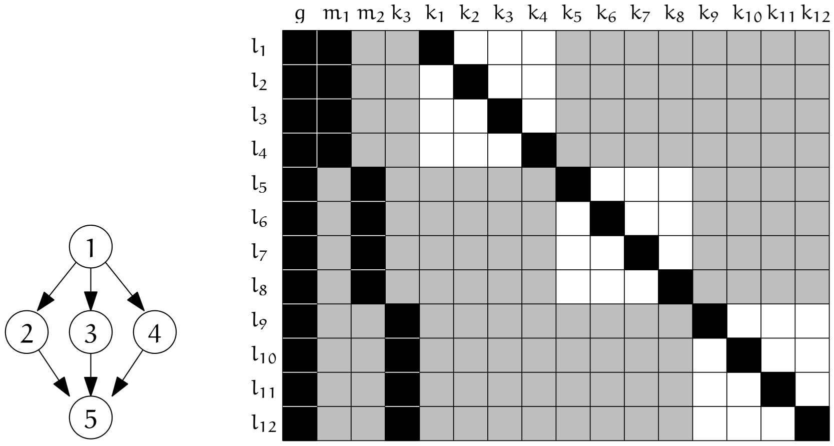
Profiled lock-charts are a special case of a melted profiles lock-charts in the following sense: An assignment $s \cup t$ is a solution to $(K, L, E, P, \preceq, a)$ if and only if it is a solution to (K, L, E, B), where

$$B = \{(k,l) \in K \times L \mid k \notin E(l) \text{ and } a(k) \preceq a(l)\}\ . \tag{2.11}$$

HIERARCHY. This section presented 5 types of lock-charts in Definitions 9, 12, 13, 19 and 24. Figure 2.6 shows their relationship graphically.

UNIQUENESS. Finally, we discuss an important assumption. Can two keys open the same set of locks (and two locks be opened by the same set of keys)? Theoretically yes, but there is a good reason not to allow this.

First, the "duplicate" keys k and k' in a basic lock-chart, whose $E(k) = E(k')$ can always be assigned the same cutting $s(k) =$



	g	m_1	m_2	k_3	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}	k_{11}	k_{12}
l_1	■	■	▨	▨	■	□	□	□	▨	▨	▨	▨	▨	▨	▨	▨
l_2	■	■	▨	▨	□	■	□	□	▨	▨	▨	▨	▨	▨	▨	▨
l_3	■	■	▨	▨	□	□	■	□	▨	▨	▨	▨	▨	▨	▨	▨
l_4	■	■	▨	▨	□	□	□	■	▨	▨	▨	▨	▨	▨	▨	▨
l_5	■	▨	■	▨	▨	▨	▨	▨	■	□	□	□	▨	▨	▨	▨
l_6	■	▨	■	▨	▨	▨	▨	▨	□	■	□	□	▨	▨	▨	▨
l_7	■	▨	■	▨	▨	▨	▨	▨	□	□	■	□	▨	▨	▨	▨
l_8	■	▨	■	▨	▨	▨	▨	▨	□	□	□	■	▨	▨	▨	▨
l_9	■	▨	▨	■	▨	▨	▨	▨	▨	▨	▨	▨	■	□	□	□
l_{10}	■	▨	▨	■	▨	▨	▨	▨	▨	▨	▨	▨	□	■	□	□
l_{11}	■	▨	▨	■	▨	▨	▨	▨	▨	▨	▨	▨	□	□	■	□
l_{12}	■	▨	▨	■	▨	▨	▨	▨	▨	▨	▨	▨	□	□	□	■

Figure 2.5: A small profile map (left) and a profiled lock-chart which use the profile map (right). Grey cells indicate blocking by profile.

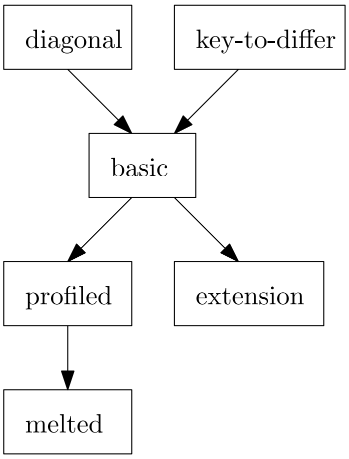


Figure 2.6: Hierarchy of lock-chart types. An arrow points from a special case to its generalisation.

$s(k')$. Assigning different cuttings would only weaken blocking properties of opened locks. Hence, from the calculation point of view, the duplicate keys are irrelevant. A similar reasoning goes for locks.

Assumption 25. *Let* K *be a key-set,* L *a lock-set and* E *an edge-set. For any* $x, y \in K \cup L$*, if* $x \neq y$ *then* $E(x) \neq E(y)$*.*

Corollary 26. *Let* $(K, L, E, \hat{s})$ *be an extension lock-chart and* s *its solution. Any two distinct keys* $k \neq k'$ *are assigned distinct cuttings* $s(k) \neq s(k')$*.*

Proof. Assume that the key k opens some lock l. Since s is a solution, $s(k)$ enters $s(l)$. Assume there is another key k' with the same cutting $s(k) = s(k')$. Since $s(k')$ enters $s(l)$ and s is a solution, k' opens l. This reasoning holds for any lock $l \in L$, hence $E(k) \subseteq E(k')$. Symmetric reasoning from k' to k yields $E(k) = E(k')$. This violates Assumption 25. □

The main purpose of this corollary will become apparent in Chapter 6. Here, it helps to understand the structure of a lock-chart.

Remark 27. No two individual keys open the same lock and no two locks are opened by the same individual key. Therefore, there the relation E contains a bijection between individual keys and their associated locks, which will be called *individual locks*.

Profiled and melted lock-charts are more complicated. We still apply the Assumption 25 (so that the previous remark is valid), but the Corollary 26 no longer holds. Its proof inferred that both k and k' opens l using (2.3). In profiled lock-charts, lock l does not have to exist if profiles of k and k' are independent.

2.4 OPTIMISATION

The lock-chart and its solution defined so far are sufficient to ask: "Does a solution exist?" or "Provide a solution, please." However, for various reasons, manufacturers often prefer some solutions over other ones. This naturally leads to finding a criterion to measure solution's quality and to rephrase lock-chart solving as an optimisation problem.

Especially high-end, patented platforms are advertised with a security guarantee that any key cutting issued for one customer is never reused for a different customer. Calculation software fulfils this promise by dividing the code space into regions, each assigned to one lock-chart. A natural manufacturer's requirement is to make such regions as small as possible so that it is possible to calculate as many lock-charts as possible using one platform.

Definition 28 (Global virtual cylinder)**.** Let K be a key-set and s an assignment. The *global virtual cylinder* Λ (abbreviated GVC) is a cylinder defined as the union of all cuttings assigned by s:

$$\Lambda = \bigcup_{k \in K} s(k)$$

The GVC is used to define two criteria, which minimise the size of a region allocated to a lock-chart.

Criterion 29 (*Maximizing prefix*)**.** *Let* (K, L, E) *be a lock-chart and* s *its solution. The* prefix *is the largest* r*-tuple* $(d_1, d_2, \dots, d_r)$ *of cutting depths s.t.* $\Lambda_i = \{d_i\}$ *for all* $1 \leqslant i \leqslant r$*. A solution with a longer prefix is preferred over a solution with a shorter prefix.*

The prefix is a well-established criterion in the industry. Its practical advantage is that every lock-chart is associated with 1 tuple, that defines its region in the code space. Then all new calculations cannot reuse the prefix. This can be done by translating prefixes into gecons. Adding a gecon $(3, 5, 2, ?, ?)$ to a platform prevents all keys from reusing the prefix $(3, 5, 2)$.

The prefix has a disadvantage when used as a criterion for comparing algorithms' performance. During prototyping, we often found that there is a critical prefix length r, which is achieved quickly by almost all algorithms, yet extending r by 1 is intractable by all algorithms. For that reason we use a different criterion in this work, taken from [38], where it was defined as "consumed key codes":

Criterion 30 (Minimise shear-lines)**.** *Let* (K, L, E) *be a lock-chart and* s *its solution. The number of* shear-lines in GVC*, denoted* $|\Lambda|$*, is defined as* $|\Lambda| = |\Lambda_1| \cdot |\Lambda_2| \cdots |\Lambda_p|$*. A solution with a smaller* $|\Lambda|$ *is preferred over a solution with a bigger* $|\Lambda|$*.*

Each GVC after a calculation defines shear-lines that cannot be used in new calculations. It is not feasible to translate a forbidden region defined by a GVC into gecons as it was done with prefixes.[5] Instead solving a basic lock-chart (K, L, E) with a forbidden GVC Λ is equivalent to solving an extension lock-chart $(K, L', E, \hat{s})$ with 1 additional lock $L' = L \cup \{v\}$, whose cylinder is fixed $\hat{s}(v) = \Lambda$. Since the new lock v stops all keys, no keys in K can be assigned a cutting which enters Λ.

2.5 EXTENSIONS

Empirical evidence indicates that finding a solution (or disproving its existence) to the extension lock-chart is easier than solving

[5] In the worst case $|\Lambda|$ grows exponentially in p. This would generate an exponential number of constraints.

a basic lock-chart.[6] The difficult part is to solve the from-scratch lock-chart so that its future extension is still solvable.

In general, this task is practically impossible. For almost any solution, there is an extension, which is unsolvable. Take two locks l, l', their cylinders $s(l)$ and $s(l')$ and assume they do not share any shear-line. If the extension contains a new key k s.t. $\{l, l'\} \subseteq E(k)$, the extension lock-chart has no solution.

Inevitably, one must specify which extensions should be expected and for which one optimises the solver. This section will discuss their various types. When all expected extensions of the chosen type are added, the largest solvable lock-chart is formed. Such lock-chart will be called the *extremal lock-chart*. This section will define two types of extremal lock-charts and further chapters will describe their solutions. A practical algorithm designed to solve from-scratch lock-charts can use cuttings from the extremal's lock-chart solution, which effectively prepares the solution for future extensions.

independent master keys. The first extremal lock-chart is suited for extensions, whose locks might be opened by an arbitrary combination of master keys. We formulate it for the vanilla framework with p positions and d cutting depths.

Definition 31. Let there be $|K| = p \cdot (d-1) + 1$ keys denoted $K = \{k_1, k_2, \ldots\}$ and $2^{|K|}$ locks denoted $L = \{l_1, l_2, \ldots\}$. In the *lock-chart of independent keys* (K, L, E) is a lock-chart, whose every lock is opened by a unique combination of keys:

$$E = \left\{(k_i, l_j) \in K \times L \,|\, (j-1)\%2^i < 2^{i-1}\right\}$$

Independent keys are meant for people, whose access to rooms or corridors might be arbitrarily changed. Say there is a lock l opened by keys $E(l)$ and the access of key k should be added or removed. The extension is formed by adding a lock opened by keys $E(l) \cup \{k\}$ or $E(l) \setminus \{k\}$. Knowing that a solution to the extremal lock-chart exists, the extension must be solvable.

The flexibility stems from the exponential number of locks. However, this comes at the price of a small number of keys. Essentially, there is a trade-off between knowing the structure of future extensions and maximizing the number of keys in the extension. Independent master keys are one side of a spectrum and individual keys are the on other one.

non-independent master keys. What happens if master keys are not marked as independent? We interpret this case by

[6] This is probably due to the fact that a partial solution reduces the number of free parameters in the calculation.

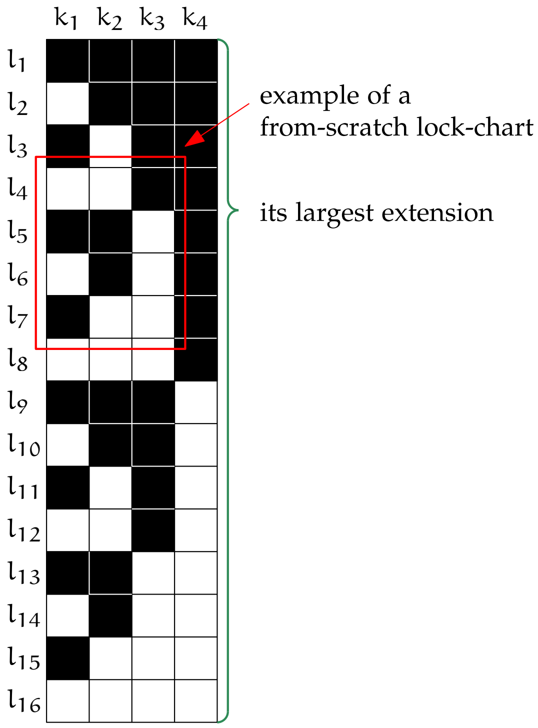


Figure 2.7: The lock-chart of 4 independent keys for $p = 3$, $d = 2$.

not expecting a new combination of master keys in an extension other than those combinations that appeared in the from-scratch problem.

The experience we gained when building commercial solvers suggests to "compress" master keys into the smallest code space possible. First individual keys are removed from the lock-chart and then the solver finds a solution which minimises $|\Lambda|$ (see Criterion 30) from master-keys' cuttings, which remained in the lock-chart. Practical algorithms with this objective will be presented in Section 6.4.

Admittedly, despite the practical experience, this approach lacks formal justification. Its analysis is one of the topics for future work.

INDIVIDUAL KEYS. A typical use-case for extensions is when a customer adds a new room to the building or loses a key and asks for a safe replacement. Both involve adding 1 individual key and 1 lock opened by the key to a lock-chart so that the new key doesn't open any previously shipped locks and the new lock blocks all existing individual keys.

The lock-chart, which maximizes the code space for individual keys, yet keeps at least some master keys, is the diagonal lock-chart. Considering diagonal lock-chart as the extremal, a natural question is: "What is the size of the largest diagonal lock-chart"?

Section 3.6 will give a definitive answer for the vanilla framework. The same question for the asymmetric and generic frameworks, to which we have found a partial answer, is presented in Chapter 5.

CENTRAL LOCKS. Central locks are used for the main door of a building or doors on corridors or floors. As such, they are expected to be opened by many keys, often by most keys in the lock-chart. Hence we define *central lock* as a lock, which is expected to be opened by new keys in an extension. Central locks are usually marked as such by the customer.

The issue with central locks is to estimate the number of shear-lines in their cylinders. Implicit solutions add as few cutting depths into cylinders as possible (see Proposition 16). The risk is that there are too few shear-lines in a cylinder and that they will not allow new keys from an extension to enter them. On the other hand, central locks still have to block some keys and going too far by adding all $d \cdot p$ cutting depths is certainly not desirable.

Mitigation of this issue is to add superfluous cutting depths into cylinders. How much and where? My experience says that this question is only of theoretical importance. Manufacturers usually specify roughly how many new keys are expected in new extensions. New cutting depths might be added until this criterion is reached.

CONCLUSION. Extensibility in all its forms can make a constraint for the decision problem or a criterion for the optimisation problem. Take independent keys, which can be rephrased both as "find a solution with all keys independent" or "find a solution with as many keys marked as independent as possible". My personal experience shows greater success with the former approach; nevertheless, such choice depends heavily on the business case and a particular algorithm employed.

3 COMPLEXITY CLASSES

The question of the highest interest to all the customers, manufacturers and programmers, who design lock-chart solving algorithms, is probably: "Can I get a solution to this particular lock-chart in a reasonable time?" But the question has a tricky part – the notion of a *reasonable time*. For manufacturers that we worked with, the reasonable time was somewhere between 1 minute and 1 hour. As programmers, we tried to persuade them that 1 day is also a reasonable time. And that within a few years, computers will become fast enough to meet the 1 hour deadline.

Such disputes are of no concern to computer scientists. Most computers ranging from mobile phones to Konrad Zuse's mechanical machines share the same mathematic model. The model works in time that is not proportional to our physical time and hence asking for the processing time of any particular lock-chart is meaningless. Instead, the right question to ask in this chapter is: "How does the run-time evolve when the size of the lock-chart grows?" Since the mathematical model more or less captures all current computers, it also shapes our expectations for practical algorithms.

3.1 PRELIMINARIES

TRACTABILITY. Let there be a combinatorial *problem*, whose instances are written in a formal language and let the instance size be n. A *decision problem* is a problem, to which the answer is *yes* or *no*. A Turing machine has *time complexity* $T(n)$ if it accepts the formal description of the problem instance and gives a yes/no answer in at most $T(n)$ time steps. A decision problem is in $\mathcal{P}$, if there exists a deterministic Turing machine with a time complexity polynomial in n. A decision problem is in $\mathcal{NP}$, if a solution to the yes-instance can be verified in $\mathcal{P}$. The $\mathcal{P} \overset{?}{=} \mathcal{NP}$ hypothesis, recently surveyed in [1] has not been proved at the time of writing this text.

A *polynomial reduction* is an algorithm that translates a formal description of one problem into another one. A decision problem is $\mathcal{NP}$-hard if every problem in $\mathcal{NP}$ can be polynomially reduced to it. A decision problem is $\mathcal{NP}$-complete if it is $\mathcal{NP}$-hard and in $\mathcal{NP}$ [29].

PROPOSITIONAL LOGIC. Let X denote the set of *variables* denoted $x_1, x_2, \dots, x_n$. Every variable is a *formula*. If F and F' are formulas, then a *negation* of a formula $\neg F$, a *conjunction* of formulas $F \wedge F'$, a *disjunction* of formulas $F \vee F'$, an *implication* of formulas $F \Rightarrow F'$, and an *equivalence* of formulas $F \Leftrightarrow F'$ are also formulas.

An *interpretation* is a function $\mathcal{I} : X \to \{0, 1\}$. The domain of $\mathcal{I}$ is recursively extended to all formulas: $\mathcal{I}(\neg F) = 1 - \mathcal{I}(F)$, $\mathcal{I}(F \wedge F') = \min\{\mathcal{I}(F), \mathcal{I}(F')\}$, $\mathcal{I}(F \vee F') = \max\{\mathcal{I}(F), \mathcal{I}(F')\}$, $s(F \Rightarrow F') = \mathcal{I}(\neg F \vee F')$, $\mathcal{I}(F \Leftrightarrow F') = \mathcal{I}((F \Rightarrow F') \wedge (F' \Rightarrow F))$. An interpretation $\mathcal{I}$ is a *model* of a formula F if $\mathcal{I}(F) = 1$. Two formulas F and F' are *semantically equivalent* if $\mathcal{I}$ is a model of F if and only if $\mathcal{I}$ is also a model of F'.

CONJUNCTIVE NORMAL FORM. Next, we consider a restricted class of formulas. Every variable x is associated a *positive literal* x and a *negative literal* $\bar{x}$. A disjunction of literals $C = \bar{x}_1 \vee \bar{x}_2 \vee \dots \vee \bar{x}_i \vee x_{i+1} \vee x_{i+2} \vee \dots \vee x_j$ is called a *clause* (with the first i literals negative). The size of a clause $|C|$ is the number of its literals (j in this case). A conjunction of clauses is called a *formula in the conjunctive normal form*, abbreviated as *CNF*. The size of a CNF $|F|$ is the number of its clauses.

The *Boolean satisfiability problem* (SAT) is a combinatorial problem of finding a model to a CNF. SAT is $\mathcal{NP}$-complete. The *2-satisfiability problem* (2SAT) is a SAT for clauses with at most 2 literals. The 2SAT problem is in $\mathcal{P}$. The *sharp-SAT problem* (#SAT) is the problem of counting the number of models to a CNF. The #SAT problem is in the class $\#\mathcal{P}$.

GRAPH PROBLEMS. Let (V, E) be an undirected graph and $k \in \mathbb{N}$. A set of nodes $I \subseteq V$ is *independent* if no vertices in I are adjacent. The *maximum independent set problem* (MIS) is a problem of finding some largest independent set [52]. The *independent set decision problem* asks if the graph contains an independent set of size at least k. The problem is $\mathcal{NP}$-complete. A *graph colouring function* is a function $c : V \to \{1, 2, \dots, k\}$, which assigns distinct values to all adjacent vertices. The *graph colouring problem* is an $\mathcal{NP}$-complete problem to determine if there exists a graph colouring function given k.

COMBINATORICS. Let $n, k \in \mathbb{W}$. *Factorial* is the function $n! = n \cdot (n-1) \cdots 2 \cdot 1$. The corner case is defined as $0! = 1$. The *binomial coefficient* $\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$ counts the choices of k items out of a basket with n items.

The ultimate goal of this chapter is to show $\mathcal{NP}$-completeness of the lock-chart solving. First, we deal with the easier part, namely proving that lock-chart solving is in $\mathcal{NP}$.

Lock-chart solving is in $\mathcal{NP}$ if the solution can be verified in polynomial time. This can be done by checking if an assignment is a solution using Definition 15, exemplified by Algorithm 3.1. Its runtime is polynomial in $|K|$, $|L|$, $|E|$, $|B|$, $|S|$, p and d.

The algorithm is formulated for the explicit constraint framework, where we can easily check $s(k) \in S$ by iterating over the set. However, if the code space S was not given explicitly, the algorithm can be easily extended. The general framework can be used simply by writing a procedure for checking a cutting against a gecon (see Example 6).[1]

This shows that all lock-charts defined in Chapter 2 are $\mathcal{NP}$.

```
input  : lock-chart $(K, L, E, B)$, partial solution $\hat{s}$, solution $s$
output : true if the solution is correct, otherwise false

foreach $k \in K$ do
    if $s(k) \neq \hat{s}(k)$ then
        return false
    end
    foreach $l \in L$ do
        if $(k, l) \in E$ then
            if $s(l)$ blocks $s(k)$ then
                return false
            end
        end
        if $(k, l) \in B$ then
            if $s(k)$ opens $s(l)$ then
                return false
            end
        end
    end
end
return true
```

Algorithm 3.1: Solution verifier for partially-solved melted profiles lock-charts.

[1] If still in doubt, fast forward to Section 4.2. An equivalence-version of CNF constructed with a partial solution $\hat{s} = s$ has all clauses with at most 2 literals (after unit propagation). Since 2SAT is in $\mathcal{P}$, lock-charts in the general framework are in $\mathcal{NP}$, which also entails vanilla and asymmetric frameworks.

GENERAL CONSTRAINTS. At the beginning of the discussion of tractability, we discuss the constraints themselves. Are gecons expressive enough to encode complex combinatorial problems?

Theorem 32. *Solving the* 1×0 *lock-chart* $(\{k\}, \emptyset, \emptyset)$ *in the general framework is* $\mathcal{NP}$*-complete.*

This theorem will be proved by creating a many-one polynomial reduction from SAT to the lock-chart problem. However, the first reduction is not presented in this section. The high-level idea of the reduction procedure uses an equivalence between a clause and a gecon. A gecon ensures that at least one cutting depth on a non-wildcard position must "deviate" from the constraint. The "at least one" corresponds to the disjunctions $\vee$ in a clause.

Claim 33. Let there be a CNF with variables $x_1, \ldots, x_n$. A lock-chart $(\{k\}, \emptyset, \emptyset)$ with $p = n$, $d = 2$ will be constructed. For every clause C let there be 1 gecon $(c_1, \ldots, c_p)$. For every positive literal $x_i \in C$, the constriant has $c_i = 1$. For every negative literal $\bar{x}_i \in C$, the constraint has $c_i = 2$. For all remaining variables $c_i =?$. The lock-chart has a solution s if and only if $\mathcal{I}$ is a model of the CNF, where $\mathcal{I}(x_i) = s(k)_i - 1$.

Proof. $(\Rightarrow)$ Assume s is a solution to the lock-chart, yet $\mathcal{I}$ is not a model of the CNF. Then there must be a clause $C = \bar{x}_1 \vee \bar{x}_2 \vee \cdots \vee \bar{x}_i \vee x_{i+1} \vee x_{i+2} \vee \cdots \vee x_j$ s.t. $\mathcal{I}(C) = 0$ and therefore all its literals evaluate to 0. For any negative literal $\mathcal{I}(\bar{x}_i) = 0 = 1 - \mathcal{I}(x_i)$, hence $s(k)_i = 2$. Similarly for all positive literals x_j, $s(k)_i = 1$. There is no position, where $s(k)$ deviates from the non-wildcard positions in the constraint created from C. The cutting $s(k)$ does not satisfy the gecon and therefore s is not a solution (a contradiction).

$(\Leftarrow)$ Assume $\mathcal{I}$ is a model of the CNF, yet there is no solution to the lock-chart. As there is no blocking cell in the lock-chart, a gecon must have been violated by $s(k)$. A contradiction is reached by similar reasoning as above. The violated constraint is associated with a clause, which must evaluate to 0 under $\mathcal{I}$. □

The result also applies to diagonal lock-charts. Observe that a diagonal lock-chart has as many locks as individual keys. A diagonal lock-chart with 0 individual keys has exactly 1 key (the general key) and no locks. Hence the 1×0 lock-chart is the smallest diagonal lock-chart and therefore solving diagonal lock-charts in the general framework are $\mathcal{NP}$-complete.

Extending the result to the 1×1 key-to-differ lock-chart can be made by adding a redundant lock. The proofs in this section would neither change a lot, neither provide additional insight. Instead, the result is generalised to the whole class of key-to-differ lock-charts.

KEY-TO-DIFFER LOCK-CHARTS. For clarity, a key-to-differ lock-chart is assumed to contain n keys and n locks $K = \{k_1, \dots k_n\}$, $L = \{l_1, \dots, l_n\}$ and both sets will indexed "conveniently" s.t. $E = \{(k_1, l_1), (k_2, l_2), \dots, (k_n, l_n)\}$.

Lemma 34. *A key-to-differ lock-chart with n keys/locks has a solution if and only if there are at least n available cuttings in the set of available keys S.*

Proof. All keys are individual. Hence all cylinders contain a single shear-line — namely the i-th lock's only shear-line is the i-th key's cutting. Such cylinder prevents all remaining keys $k_j \neq k_i$ from having the same cutting. By the pigeonhole principle it is possible to assign $|S|$ cuttings to n keys if and only if $n \leqslant |S|$. □

This lemma implies results to some trivial decision problems.

Corollary 35. *A key-to-differ lock-chart with n keys/locks is solvable in $\mathcal{P}$ if a) $n \leqslant d^p$ in vanilla framework, b) $n \leqslant \tilde{d}_1 \cdot \tilde{d}_2 \cdot \dots \cdot \tilde{d}_p$ in asymmetric framework with the deepest cutting $(\tilde{d}_1, \dots, \tilde{d}_p)$ or c) $n \leqslant |S|$ if the set S is given by the explicit framework.*

The only remaining question is whether key-to-diff lock-charts are solvable in the general framework.

Corollary 36. *The problem of finding the largest key-to-differ lock-chart in the general framework is in $\#\mathcal{P}$.*

Proof. As in the proof of Claim 33, every model is equivalent to 1 valid key cutting $s(k)$. By Lemma 34 every model corresponds to one key in a key-to-differ lock-chart. The number of models to a CNF is given by the #SAT problem, which is in $\#\mathcal{P}$. □

EXTENSIONS LOCK-CHARTS. The idea of having two cutting depths for two boolean values and one position per variable, that we used in Lemma 33, is not new. Lawer [38] used a similar translation procedure to turn SAT into an extension lock-chart in the vanilla framework. Theorem 32 will be related to Lawer's main result by a reduction of a basic lock-chart in general framework into an extension lock-chart in vanilla framework.

Lemma 37. *Let $(c_1, \dots, c_p)$ be a gecon. A* universal cylinder *is $(D_1, \dots, D_n)$, s.t. if $c_i = ?$ then $D_i = \{1, 2, \dots, d\}$; otherwise $D_i = \{c_i\}$. A key cutting $(d_1, \dots, d_p)$ satisfies the gecon if and only if it is blocked in its universal cylinder.*

Proof. If the cutting depth satisfies the constraint, there is a position i, s.t. $c_i \neq ?$ and $c_i \neq d_i$. Hence $d_i \notin D_i$ and the cylinder blocks the cutting. If the cutting depth violates the constraint, all $d_i \in D_i$ and hence the cylinder is opened by the cutting. □

The lemma gives a mechanism to turn constraints into extension's locks. Given a lock-chart (K, L, E) with n gecons, an extension lock-chart $(K, L', E, \hat{s})$ is constructed by adding one new lock for each constraint: $L' = L \cup \{u_1, \dots, u_n\}$. The partial solution $\hat{s}(u_i)$ fixes the i-th added lock to a universal cylinder associated with the i-th gecon. Because every key is blocked in every u_i, the newly added locks have the same effect on the solution as the gecons.

Corollary 38. *Extension lock-charts in the vanilla framework are $\mathcal{NP}$-complete.*

Since gecons are *slightly* less expressive than blocking by (universal) cylinders, our result in the previous section is a *slight* generalization of Lawer's result.

3.4 MELTED PROFILES LOCK-CHARTS

While designing commercial solvers, which all ran in exponential time, we were constantly bothered with the question whether basic lock-charts in the vanilla framework can't be in $\mathcal{P}$. Such eventuality would turn all such algorithms in vain. The closest we got to answer this question is a result on melted profiles lock-charts.

There is also a second motivation. $\mathcal{NP}$-completeness results so far relied on a growing number of positions. However, in some real-world platforms, the p parameter is fixed and quite low, sometimes as low as 5. Practically speaking, this makes it possible to iterate over d^p cuttings on current CPUs. From a theoretical point of view, having p fixed makes d^p a constant-factor slowdown. Hence a practical algorithm, which scales badly in d^p, but well in other parameters ($|K|, |L|, |E|, \dots$) might still exist.

Admittedly, we were unable neither to prove or disprove the existence of such algorithm for basic lock-charts. Instead, this section has an $\mathcal{NP}$-completeness result on melted profiles lock-charts with the property of having $p = 1$.

Theorem 39. *Let (V, F) be an undirected graph and k a natural number. The graph colouring problem is translated into a melted profile lock-chart (K, L, E, B). The lock-chart has a key k_i and a lock l_i for each vertex $v_i \in V$. The relation E connects i-th key with i-th lock only (as in the key-to-differ lock-chart). The relation of stopped keys* *and locks* B *connects* k_i *and* l_j *if* $(v_i, v_j) \in F$.[2] *The vanilla framework is used with* $p = 1$ *and* $d = k$. *Given a solution to the lock-chart* s, *a graph colouring function* f *defined as* $f(v_i) = s(k_i)_1$ *is a solution to the graph colouring problem. Solving melted profiles lock-charts in the vanilla framework is* $\mathcal{NP}$*-complete.*

The proof will show that f solves the graph colouring problem.

Proof. Graph colouring function cannot assign the same color to adjacent vertices. Suppose it does by having some $(v_i, v_j) \in F$, s.t. $f(v_i) = f(v_j) = x$. Then both keys k_i and k_j were assigned the cutting (x) and lock l_i (which is opened by k_i only) was assigned the cylinder $(\{x\})$. Therefore k_j is not blocked in l_i, which is a contradiction. □

This concludes the discussion of $\mathcal{NP}$-completeness. The rest of this chapter will identify problems that are solvable in $\mathcal{P}$.

3.5 LOCK-CHART OF INDEPENDENT KEYS

The $\mathcal{NP}$-completeness from previous sections motivates finding classes of lock-charts, whose solution can be found in polynomial time. Mechanical constraints, which can be almost arbitrary in practice, make it difficult to devise polynomial-time algorithms. Therefore from now on, the proofs will be restricted to a simple class of mechanical constraints to achieve a polynomial-time algorithm. This section starts with lock-charts of independent keys.

Theorem 40. *Let* K *be the set of keys in a lock-chart of independent keys. The* i*-th key is associated numbers* x, y, *where* $x = 1 + (i-2) \div (d-1)$ *and* $y = 2 + (i-2)\,\%(d-1)$, *except for the key* k_1 *which gets* $x = y = 1$. *The assignment* s *is a solution, where*

$$s(k_i) = (\underbrace{1, \ldots, 1}_{x-1}, y, \underbrace{1, \ldots, 1}_{p-x}) \;.$$

Proof. First note that every key has a unique tuple (x, y) (because the formula is reversible: $i = (x-1) \cdot (d-1) + y$). The proof follows Definition 15. Take any lock l. From each key $k \in E(l)$, the cylinder $s(l)$ contains the cutting depth y at the x-th position: $y \in s(l)_x$. Take any stopped key $k' \in K \setminus E(l)$ and its x' and y' values. By the same reasoning, $y' \notin s(l)_{x'}$, therefore $s(k')$ is blocked in $s(l)$. □

[2] Since (V, E) is simple, $E \cap B = \emptyset$, which ensures that (K, L, E, B) is a valid melted profiles lock-chart.

Key	x	y	cutting
k_1	1	1	$(1,1,1)$
k_2	1	2	$(2,1,1)$
k_3	2	2	$(1,2,1)$
k_4	3	2	$(1,1,2)$

Table 3.1: Solution to the lock-chart of 4 independent keys for $p = 3$, $d = 2$.

Example 41. Assume $p = 3$, $d = 2$. Figure 2.7 on page 30 shows the corresponding lock-chart of independent keys with 4 keys. The solution is shown in Table 3.1.

Corollary 42. *Lock-chart of independent keys in the vanilla framework is solvable in $\mathcal{P}$ merely by deciding $|K| \leqslant p \cdot (d-1)+1$.*

For clarity, the theory is formulated for the vanilla framework, but it can be easily extended to asymmetric ones. The principle is to spread x, y values across cutting depths on all positions. In vanilla framework, there are $1 + p \cdot (d-1)$ such pairs, meanwhile in the asymmetric framework, there are

$$1 + \prod_{1 \leqslant i \leqslant p} (\tilde{d}_i - 1) \text{ of them.}$$

3.6 DIAGONAL LOCK-CHARTS

The proof of diagonal lock-charts' feasibility in this section is one of the main results of the text. We were able to formulate the result in the vanilla framework.

The strategy for assigning key cuttings is not a new invention. In the industry, it is known as the *rotating constant method*. To the best of our knowledge, the only publication on this subject is [44], even though the method is widely recognized and used by many business partners. This section does not provide a wide introduction into rotating constant method; we focus on a proof of its completeness and assumptions behind it.

This section starts with a lower bound on the size of the largest solvable diagonal lock-chart without assuming any particular constraint framework. Then, assuming vanilla framework, this we prove an upper bound, which concludes the $\mathcal{P}$ time solvability of vanilla diagonal lock-charts.

Definition 43. Let γ_g be the general key's cutting and $q \in \mathbb{W}$ s.t. $q \leqslant p$, Then S_q is a set of all key cuttings, which are *equal to the general key* exactly in q positions.

Example 44. If the general key is $(1, 1, 1, 1)$, the S_1 cuttings include e.g. $(1, 2, 2, 2)$ and $(2, 2, 1, 2)$, but neither $(1, 2, 2, 1)$ nor $(2, 2, 2, 2)$. Example 17 contains shapes only from the S_2 set.

Even though S_p is a perfectly well defined set, $q < p$ is assumed in this section, which excludes S_p from the reasoning. Cuttings from S_q are used for individual keys in a diagonal lock-chart and since $S_p = \{\gamma_g\}$, it makes no sense to reuse the general key's cutting for the first and only individual key.

Theorem 45 (Lower bound)**.** *The largest solvable diagonal lock-chart has at least* $\max_q |S_q|$ *individual keys.*

Proof. Take any key k_i, which should be blocked $(i \neq j)$ in some lock l_j, which is opened only by g and k_j. Let key cuttings be $s(k_i) = (d^i_1, \dots, d^i_p)$, $s(k_j) = (d^j_1, \dots, d^j_p)$ and $s(g) = (d^g_1, \dots, d^g_p)$ and cylinder pins $s(l_j) = (D^j_1, \dots D^j_p)$. The proof is done by finding some blocking position r, s.t. $d^i_r \notin D^j_r$.

Let $A_i = \{r^i_1, \dots, r^i_q\}$ be the set of q positions, where the cutting of k_i is equal to the cutting of the general key g. Formally

$$A_i = \{r \in \mathbb{N} \mid s(k_i)_r = s(g)_r\} \ .$$

a) $A_i = A_j$. Since k_i, k_j and g have different cuttings (by Corollary 26) and $q < p$, there must be at least one position $r \notin A_i$ where $d^i_r \neq d^j_r$ and $d^i_r \neq d^g_r$. On r, pins $D^j_r = \{d^g_r, d^j_r\}$, which implies $d^i_r \notin D^j_r$.

b) $A_i \neq A_j$: Since $|A_i| = |A_j| = q$, there is at least one position $r \in A_j \setminus A_i$ where k_j equals g $(d^j_r = d^g_r)$, but k_i is not $d^i_r \neq d^j_r$. On r, pins $D^j_r = \{d^g_r\}$, which implies $d^i_r \notin D^j_r$.

In both cases, $s(k_i)$ is blocked in $s(l_j)$ on the r-th position. □

The proof of the upper bound is preceded with a few lemmas. First, a code-space symmetry is described, which proves that using the cutting $(1, \dots, 1)$ for the general key is as good as any other cutting.

Lemma 46. *Given* q*, in the vanilla framework, the size of* S_q *is constant, irrespective of the general key's cutting:*

$$|S_q| = \binom{p}{p-q} \cdot (d-1)^{p-q} \tag{3.1}$$

Proof. Suppose there is a general key cutting $(g_1, \dots, g_p)$ and a cutting $(d_1, \dots, d_p) \in S_q$. There are q positions where the cutting depth matches $g_i = d_i$. In the $p - q$ remaining positions, d_i can take any value $1 \leqslant d_i \leqslant d$ except for g_i. □

Definition 47. Given the general key g, and two cuttings γ, γ'. The cutting γ is *unsuitable for individual keys in a diagonal lock-chart* with γ' if the cylinder $g \cup \gamma$ is entered by γ' or the cylinder $g \cup \gamma'$ is entered by γ.

The plan behind unsuitable cuttings is to find the largest set of suitable cuttings, which then forms a solution for the diagonal lock-chart.

Lemma 48. *Let $0 \leqslant r \leqslant q \leqslant p$. In a single lock-chart, every cutting $\gamma_q \in S_q$ has $\binom{p-r}{q-r}$ unsuitable cuttings in S_r. Every cutting $\gamma_r \in S_r$ has $\binom{q}{q-r} \cdot (d-1)^{q-r}$ unsuitable keys in S_q. The ratio of these numbers is*

$$\Omega = \frac{\binom{p-r}{q-r}}{\binom{q}{q-r} \cdot (d-1)^{q-r}} = \frac{|S_q|}{|S_r|} \, . \tag{3.2}$$

Proof. A cutting $\gamma_r \in S_r$ differs from the general key in $p - r$ positions. By "overwriting" them in some $q - r$ positions by the depth of the general key, the newly obtained cutting γ_q differs from the general in $(p-r)-(q-r) = p-q$ positions, hence $\gamma_q \in S_q$. There are exactly $\binom{p-r}{q-r}$ such overwritings.

Similarly, there are exactly $\binom{q}{q-r}$ combinations of positions, where "diverting" from the general key moves a key from S_q to S_r. In each of them, there are exactly $d-1$ available depths different from the general. This gives the $\binom{q}{q-r} \cdot (d-1)^{q-r}$ formula.

Let γ_q and γ_r be cuttings obtained by overwriting or diverting. Cylinder $g \cup \gamma_r$ can only have more shear-lines than $g \cup \gamma_q$, which is entered by γ_q. Hence γ_q also enters $g \cup \gamma_r$ and γ_q and γ_r are not suitable.

The last equality for Ω is proven by simple algebra (using $x^y \cdot x^z = x^{y+z}$, $\binom{n}{k} = \binom{n}{n-k}$ and the definition of the binomial coefficient. □

Example 49. Let $p = 4$, $d = 2$ and $(1,1,1,1)$. The cutting $\gamma = (1,1,2,2) \in S_2$ is unsuitable with $\binom{4-2}{3-2} = 2$ cuttings from S_3, namely $(1,1,1,2)$ and $(1,1,2,1)$, because neither of them is blocked in cylinder $\gamma \cup g = (\{1\},\{1\},\{1,2\},\{1,2\})$. Similarly γ is unsuitable with $\binom{2}{2-1} \cdot (2-1)^{2-1} = 2$ cuttings from S_1, namely $\gamma' = (1,2,2,2)$ and $\gamma'' = (2,1,2,2)$, because γ is blocked neither in $g \cup \gamma'$, nor in $g \cup \gamma''$.

Lemma 50. *No two distinct cuttings $\gamma, \gamma' \in S_q$ are unsuitable with the same set of cuttings from a different S_r set.*

Proof. Let g be the general key's cutting. a) $r < q$: If two keys are assigned distinct γ, γ', their opened locks have cylinders $g \cup \gamma$, $g \cup \gamma'$ that are also distinct and therefore both are opened by different sets of cuttings from S_r. b) $r > q$: Reasoning is similar, by considering cylinders of keys unsuitable with γ and γ'.

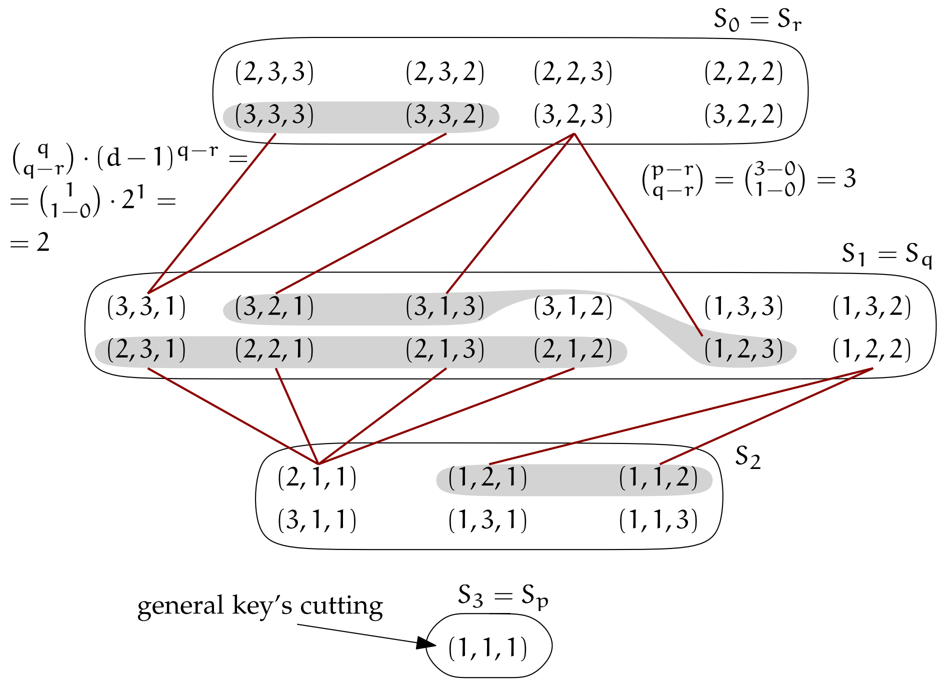


Figure 3.1: All cuttings from a vanilla framework $p = 3$, $d = 3$, grouped into S_q sets w.r.t. the general key $(1,1,1)$. Selected pairs of unsuitable cuttings are joined by red lines.

□

The last lemma needed for the main theorem generalises the binomial coefficient.

Lemma 51. *Let C be a set of all subsets of $\{1,\ldots,n\}$, whose cardinality is k ($\forall K \in C. |K| = k$). If every $i \in \{1,\ldots,n\}$ appears in at most r subsets, then $|C| \leqslant \frac{n}{k} \cdot r$. If moreover k and n are divisible, then $|C| = \frac{n}{k} \cdot r$.*

Example. For $n = 4$, $k = 2$ and r unrestricted, C has $\binom{4}{2} = 6$ subsets $C = \{\{1,2\}, \{1,3\}, \{1,4\}, \{2,3\}, \{2,4\}, \{3,4\}\}$. By deleting $\{1,2\}$ and $\{3,4\}$, every number appears exactly $r = 2$ times. Indeed $\frac{n}{k} \cdot r = 4$ remaining subsets.

Note. From the binomial coefficient $\frac{n}{k} \cdot r = |C| \leqslant \binom{n}{k}$. Therefore $r \leqslant \binom{n}{k}/\frac{n}{k}$ and hence $r \leqslant \binom{n-1}{k-1}$.

Proof. Inequality: Let C_i denote the elements of C which contain i. By definition $|C_i| \leqslant r$ and therefore $\sum_{i=1}^{n} |C_i| \leqslant n \cdot r$. Also $|C| \cdot k = \sum_i |C_i| \leqslant n \cdot r$ which yields the inequality.

Equality: 1. Initialise $r = 0$ and $C = \emptyset$. 2. Take a permutation $\pi = \{\pi_1, \ldots, \pi_n\}$ of the set $\{1,\ldots,n\}$ and chop it into chunks of size k (assuming n and k divisible) to obtain sets $c_1 = \{\pi_1, \pi_2, \ldots, \pi_k\}$, $c_2 = \{\pi_{k+1}, \ldots, \pi_{2k}\}$, …, $c_{\frac{n}{k}} = \{\pi_{n-k+1}, \ldots, \pi_n\}$. 3. If C does not contain any of the c sets, then increase r by 1 and add all c sets

($\frac{n}{k}$ many of them) into C. 4. Reiterate from step 2. This algorithm has 2 invariants: $|C| = \frac{n}{k} \cdot r$ and $C_i = r$. □

Theorem 52 (Upper bound)**.** *The largest solvable diagonal lock-chart has at most* $\max_q |S_q|$ *individual keys.*

Proof. Take any two distinct sets S_r and S_q. Suppose there is a lock-chart with a solution s', whose individual keys are assigned cuttings $S'_r \cup S'_q$, where $S'_r \subseteq S_r$ and $S'_q \subseteq S_q$. Their complements are denoted $\bar{S}'_r = S_r \setminus S'_r$ and $\bar{S}'_q = S_q \setminus S'_q$. This proof will show that there is another solvable lock-chart with at least $\max\{|S_r|, |S_q|\}$ individual keys.

Given that the solution is correct, all keys from $S'_r \cup S'_q$ are pairwise suitable. Theorem 45 implies that all keys from S_r are pairwise suitable, as well as keys from S_q. Therefore, the only pairs of unsuitable keys are I. in $S'_r \times \bar{S}'_q$, II. in $S'_q \times \bar{S}'_r$ and III. in $\bar{S}'_r \times \bar{S}'_q$.

Consider the case I. (resp. II.) and assume $r < q$ WLOG. By Lemma 48, every cutting in S'_r (resp. S'_q) has $\binom{p-r}{q-r}$ (resp. $\binom{q}{r} \cdot (d-1)^{q-r}$) unsuitable cuttings in $\bar{S}'_q$ (resp. $\bar{S}'_r$). By Lemma 50, for every cutting in S'_r (resp. S'_q), there is a distinct combination of cuttings in $\bar{S}'_q$ (resp. $\bar{S}'_r$). By the same reasoning, a cutting can appear in at most $\binom{q}{r} \cdot (d-1)^{q-r}$ (resp. $\binom{p-r}{q-r}$) such combinations (again by Lemma 48). This gives two formulas as applications of Lemma 51:

$$\frac{|\bar{S}'_q|}{\binom{p-r}{q-r}} \cdot \left[\binom{q}{q-r} \cdot (d-1)^{q-r}\right] = \frac{|S_q| - |S'_q|}{\Omega} \geqslant |S'_r|\,. \tag{3.3}$$

$$\left(\text{resp. } \frac{|S_r| - |S'_r|}{\binom{q}{q-r} \cdot (d-1)^{q-r}} \cdot \binom{p-r}{q-r} = \Omega \cdot \left(|S_r| - |S'_r|\right) \geqslant |S'_q|\right) \tag{3.4}$$

Next, assume $S_q \geqslant S_r$, which gives $\Omega \geqslant 1$ (resp. $S_r \geqslant S_q$, $\Omega \leqslant 1$). The equation can be rewritten as $|S_q| \geqslant |S'_r| \cdot \Omega + |S'_q| \geqslant |S'_r| + |S'_q|$ (resp. $|S_r| \geqslant \frac{|S'_q|}{\Omega} + |S'_r| \geqslant |S'_q| + |S'_r|$). Therefore $S'_r \cup S'_q$, arbitrarily chosen subsets of S_r and S_q, cannot be larger than S_q (resp. S_r). Since S_q is bigger than S_r (resp. S_r is bigger than S_q), there is a diagonal lock-chart, whose individual keys take cuttings form S_q (resp. S_r), which is bigger than $|S'_r \cup S'_q|$. By pair-wise comparison, we can maximize over all $S_0, \dots, S_p$. □

Theorems 45 and 52 together imply that by finding some optimal value $\hat{q} = \arg\max_q |S_q|$ the cuttings from $S_{\hat{q}}$ allow solving the largest possible lock-chart. Next, we show how to find $\hat{q}$ even faster than by iterating over all S_qs.

Theorem 53. *An optimal value* $\hat{q}$, *which maximizes* $|S_q|$ *is*

$$\hat{q} = \left\lfloor \frac{p+1}{d} \right\rfloor . \tag{3.5}$$

Proof. a) $|S_{\hat{q}}| \geqslant |S_{\hat{q}+1}|$:

$$\binom{p}{\hat{q}} \cdot (d-1)^{p-\hat{q}} \geqslant \binom{p}{\hat{q}+1} \cdot (d-1)^{p-(\hat{q}+1)} \tag{3.6}$$

$$(d-1) \geqslant \frac{p-\hat{q}}{\hat{q}+1} \tag{3.7}$$

$$\hat{q} \geqslant \frac{p+1}{d} - 1 \tag{3.8}$$

b) $|S_{\hat{q}}| \leqslant |S_{\hat{q}-1}|$. Similar reasoning yields $\hat{q} \leqslant \frac{p+1}{d}$. □

ACTUAL NUMBERS. The size of a diagonal lock-chart in the vanilla framework depends purely on two parameters (p and d) and can be easily computed using closed form formulas. Table 3.2 and Figure 3.2 show the actual numbers obtained by formulas (3.1 and 3.5) in Lemma 46 and Theorem 53. The $\frac{d^p}{|S_{\hat{q}}|}$ ratio in Table 3.3 tells how many cuttings in the code space are there per one individual key in the largest diagonal lock-chart. They might be used a quick reference for comparing different platforms.

3.7 CONCLUSION

Results of the entire chapter can be summarised as follows: In the general framework, all non-trivial lock-chart problems are $\mathcal{NP}$-complete. In the asymmetric framework, the only known fact is a trivial result on key-to-differ lock-charts, which are in $\mathcal{P}$. In the vanilla framework, melted profiles lock-charts and extension lock-charts are $\mathcal{NP}$-complete, but diagonal and key-to-differ lock-charts are in $\mathcal{P}$. The situation in the vanilla framework is visualised in Figure 3.3.

Finally, let us conjecture that profiles do not express additional complexity over basic lock-charts. Consider a key-to-differ profiled lock-chart with a profile-map $(P, \preceq)$ in the vanilla framework with. If $p = d = 1$, blockings must be ensured by the profile assignment a and therefore lock k_i is blocked in l_j if and only if $a(k_i)$ and $a(l_j)$ are independent. Finding the largest set of independent profiles in a profile-map $(P, \preceq)$ corresponds to finding the largest anti-chain in the partial order $\preceq$, which is a problem in the $\mathcal{P}$ class [20]. The interaction between profiles and positions is a good topic for future research.

	$d = 2$	$d = 3$	$d = 4$	$d = 5$	$d = 6$	$d = 7$	$d = 8$	$d = 9$	$d = 10$	$d = 11$
$p = 1$	1	2	3	4	5	6	7	8	9	10
$p = 2$	2	4	9	16	25	36	49	64	81	100
$p = 3$	3	12	27	64	125	216	343	512	729	1000
$p = 4$	6	32	108	256	625	1296	2401	4096	6561	$1 \cdot 10^4$
$p = 5$	10	80	405	1280	3125	7776	16807	32768	59049	$1 \cdot 10^5$
$p = 6$	20	240	1458	6144	18750	46656	117649	262144	531441	$1 \cdot 10^6$
$p = 7$	35	672	5103	28672	109375	326592	823543	2097152	4782969	$1 \cdot 10^7$
$p = 8$	70	1792	20412	131072	625000	2239488	6588344	16777216	43046721	$1 \cdot 10^8$
$p = 9$	126	5376	78732	589824	3515625	15116544	51883209	150994944	387420489	$1 \cdot 10^9$
$p = 10$	252	15360	295245	2949120	19531250	100776960	403536070	1342177280	3874204890	$1 \cdot 10^{10}$
$p = 11$	462	42240	1082565	14417920	107421875	665127936	3107227739	11811160064	38354628411	$11 \cdot 10^{10}$
$p = 12$	924	126720	4330260	69206016	644531250	4353564672	23727920916	103079215104	376572715308	$12 \cdot 10^{11}$
$p = 13$	1716	366080	16888014	327155712	3808593750	28298170368	179936733613	893353197568	3671583974253	$13 \cdot 10^{12}$
$p = 14$	3432	1025024	64481508	1526726656	22216796875	198087192576	1356446145698	7696581394432	35586121596606	$14 \cdot 10^{13}$
$p = 15$	6435	3075072	241805655	7633633280	128173828125	1371372871680	10173346092735	65970697666560	343151886824415	$15 \cdot 10^{14}$
$p = 16$	12870	8945664	967222620	37580963840	732421875000	9403699691520	81386768741880	562949953421312	3294258113514384	$16 \cdot 10^{15}$

Table 3.2: Exact size of the largest diagonal lock-chart in the vanilla framework. Notice how $\hat{q}$ changes between rows $p = 10$ and $p = 11$.

	$d = 2$	$d = 3$	$d = 4$	$d = 5$	$d = 6$	$d = 7$	$d = 8$	$d = 9$	$d = 10$	$d = 11$	$d = 12$	$d = 13$	$d = 14$	$d = 15$	$d = 16$
$p = 1$	2.00	1.50	1.33	1.25	1.20	1.17	1.14	1.12	1.11	1.10	1.09	1.08	1.08	1.07	1.07
$p = 2$	2.00	2.25	1.78	1.56	1.44	1.36	1.31	1.27	1.23	1.21	1.19	1.17	1.16	1.15	1.14
$p = 3$	2.67	2.25	2.37	1.95	1.73	1.59	1.49	1.42	1.37	1.33	1.30	1.27	1.25	1.23	1.21
$p = 4$	2.67	2.53	2.37	2.44	2.07	1.85	1.71	1.60	1.52	1.46	1.42	1.38	1.35	1.32	1.29
$p = 5$	3.20	3.04	2.53	2.44	2.49	2.16	1.95	1.80	1.69	1.61	1.55	1.49	1.45	1.41	1.38
$p = 6$	3.20	3.04	2.81	2.54	2.49	2.52	2.23	2.03	1.88	1.77	1.69	1.62	1.56	1.51	1.47
$p = 7$	3.66	3.25	3.21	2.72	2.56	2.52	2.55	2.28	2.09	1.95	1.84	1.75	1.68	1.62	1.57
$p = 8$	3.66	3.66	3.21	2.98	2.69	2.57	2.55	2.57	2.32	2.14	2.01	1.90	1.81	1.74	1.68
$p = 9$	4.06	3.66	3.33	3.31	2.87	2.67	2.59	2.57	2.58	2.36	2.19	2.06	1.95	1.86	1.79
$p = 10$	4.06	3.84	3.55	3.31	3.10	2.80	2.66	2.60	2.58	2.59	2.39	2.23	2.10	1.99	1.91
$p = 11$	4.43	4.19	3.87	3.39	3.38	2.97	2.76	2.66	2.61	2.59	2.60	2.41	2.26	2.14	2.03
$p = 12$	4.43	4.19	3.87	3.53	3.38	3.18	2.90	2.74	2.66	2.62	2.60	2.61	2.43	2.29	2.17
$p = 13$	4.77	4.36	3.97	3.73	3.43	3.42	3.06	2.85	2.72	2.66	2.62	2.61	2.62	2.45	2.31
$p = 14$	4.77	4.67	4.16	4.00	3.53	3.42	3.24	2.97	2.81	2.71	2.66	2.63	2.62	2.63	2.47
$p = 15$	5.09	4.67	4.44	4.00	3.67	3.46	3.46	3.12	2.91	2.78	2.70	2.66	2.63	2.63	2.63
$p = 16$	5.09	4.81	4.44	4.06	3.85	3.53	3.46	3.29	3.04	2.87	2.77	2.70	2.66	2.64	2.63

Table 3.3: The $\frac{d^p}{|S_{\hat{q}}|}$ ratio tells how many cuttings in the code space are there per one individual key in the largest diagonal lock-chart.

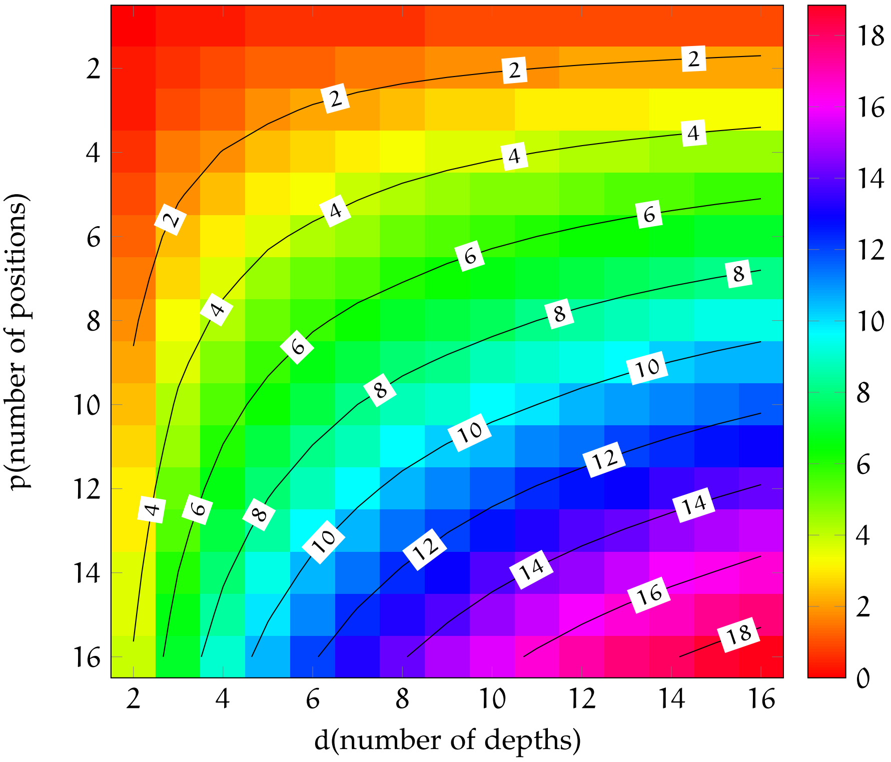


Figure 3.2: 2D contour plot of the largest diagonal lock-chart size in vanilla framework in $\log_{10}$ scale. E.g. for $p = 14$, $d_{\max} = 3$, we get $|S_{\hat{q}}| = |S_5| = 1025024 \simeq 10^6$, which corresponds to the value 6 in the chart.

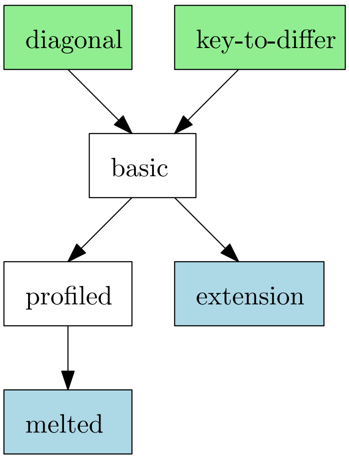


Figure 3.3: Lock-chart problem complexity in the vanilla framework. $\mathcal{P}$ variants are printed green, $\mathcal{NP}$-complete variants are printed blue.

4 PROPOSITIONALIZATION

In combinatorial optimisation, there is usually a trade-off between algorithm's generality and its performance. Given a specific problem, one can improve the runtime by better analysis, discovering symmetries in the search-space or by improving data structures and their implementation. But development takes time. Hence a "practical approach" to lock-chart solving should start from the other end. In this chapter, we try picking a state-of-the-art general solver geared towards $\mathcal{NP}$-complete problems and translating the lock-chart problem into its formalism.

We called the approach *propositionalization*, a term used by [37] to describe a process of rephrasing a problem in a higher-complexity language into a lower-complexity one. Here lock-charts defined in relational algebra are rewritten into propositional logic.

4.1 PRELIMINARIES

This chapter will use notions of SAT, clause, formula and CNF, defined in the preliminaries of Chapter 3. However, for better readability, we use several concise notations, all of which preserve semantic equivalence.

SYNTACTIC SUGAR FOR CNF. A clause with the first i literals negative can be written using the implication as $(x_1 \wedge x_2, \wedge \cdots \wedge x_i) \Rightarrow (x_{i+1} \vee x_{i+2} \vee \cdots \vee x_j)$. A clause with at most 1 positive literal is a *Horn clause*. An equivalence with a disjunction $x_0 \Leftrightarrow (x_1 \vee x_2 \vee \cdots \vee x_n)$ is rewritten into a CNF with $n+1$ clauses: $(x_0 \Rightarrow (x_1 \vee x_2 \vee \cdots \vee x_n)) \wedge (x_0 \Leftarrow x_1) \wedge (x_0 \Leftarrow x_2) \wedge \cdots \wedge (x_0 \Leftarrow x_n)$. An equivalence with a conjunction $x_0 \Leftrightarrow (x_1 \wedge x_2 \wedge \cdots \wedge x_n)$ is rewritten into a CNF with $n+1$ clauses: $(x_0 \Rightarrow x_1) \wedge (x_0 \Rightarrow x_2) \wedge \ldots \wedge (x_0 \Rightarrow x_n) \wedge (x_0 \Leftarrow (x_1 \wedge x_2 \wedge \cdots \wedge x_n))$. To remove most parentheses from the previous definitions, we define the *operator priority* to parse $\neg \bar{x}_1 \vee x_2 \wedge x_3 \Rightarrow x_4 \Leftrightarrow x_5$ into $((((\neg(\bar{x}_1)) \vee x_2) \wedge x_3) \Rightarrow x_4) \Leftrightarrow x_5$.

CONVEX OPTIMISATION. *Linear programming* is the problem of finding an n-dimensional vector $x = (x_1, \ldots, x_n)$ or rational numbers, which maximizes the dot product $c^T x$ subject to $Ax \leqslant b$, where c is a vector, A is a $n \times m$ matrix and b an

m-dimensional vector (the inequality $Ax \leqslant b$ describes m independent constraints, all of which must be satisfied). *Integer linear programming* (ILP) is a linear problem, where $x_1, \ldots, x_n \in \mathbb{Z}$. *Quadratic programming* is the problem of finding an n-dimensional vector $x = (x_1, \ldots, x_n)$, which minimises $\frac{1}{2}x^T Qx + c^T \cdot x$ subject to $Ax \leqslant b$, where Q is a $n \times n$ matrix. Linear programming is in $\mathcal{P}$, integer linear programming is $\mathcal{NP}$-hard.

4.2 BOOLEAN SATISFIABILITY

This section describes a procedure to translate the lock-chart problem in the general framework into a CNF, which can be solved by an off-the-shelf SAT solver.

There is a good motivation behind this idea. First, the quite large SAT solving community organizes yearly competitions,[1] where SAT solvers are benchmarked on various industrial and synthetic instances. Using the final scores, one can quickly pick state-of-the-art algorithms from the vast pool of existing libraries. Second, there is a widely adopted text format for representing CNFs called DIMACS.[2] A simple format shared by most SAT solving libraries allows quick prototyping and benchmarking.

Algorithms for SAT are roughly of two types. *Local search algorithms* start from a random interpretation (a point in the $|X|$-dimensional space), which might not satisfy the CNF. The assignment is gradually improved by flipping variable polarities (searching the neighbourhood of the point). A notable example is the GSAT algorithm [49], which picks the flipped variable by the number of unsatisfied clauses in which it appears. The danger of becoming trapped in local minima inspired the WalkSAT algorithm [48], which randomly picks an unsatisfied clause and flips one of its variables. Since WalkSAT is not guaranteed make the globally best move unlike GSAT, the two algorithms can be viewed as instances of hill-climbing and gradient descent respectively [35].

The second type of algorithms are descendants of the Davis–Putnam and Davis–Putnam–Logemann–Loveland [16, 17] algorithms, usually referred as *DPLL-based algorithms*. They are backtracking algorithms with a *decision* phase and a *unit propagation* phase. In the simplest form, decision phase picks a variable and assigns a polarity to it. Clauses which contain the variable in a positive literal are ignored (until a backtrack). Negative literals of the assigned variable are removed from all clauses (until a backtrack). Clauses with exactly 1 unassigned remaining literal are called

[1] http://www.satcompetition.org

[2] http://www.satcompetition.org/2009/format-benchmarks2009.html

unitary. Unit propagation is a procedure that assigns the truth value of the remaining literal, possibly rendering more clauses unitary – a process which resembles a chain-reaction. If a variable is assigned both 0 and 1 during unit propagation, a *conflict* causes the algorithm to backtrack. The strength of DPLL algorithms stems from clever data structures, well described in a seminal paper on the MiniSat algorithm [19]. The important message for this text is the polynomial runtime of unit propagation.

Modern DPLL (including MiniSAT) employ some sort of *Conflict-Driven Clause Learning* (CDCL). Particular implementations vary, but in general, CDCL algorithms can analyse the clauses which caused a conflict, simplify them and generate a *conflict clause*. The conflict has two effects. First, it prunes the search space by allowing the unit propagation to propagate variables earlier. Second, the backtrack does not have to include a single variable, but more of them – a technique known as *backjumping* (or non-chronological backtracking). Readers interested in more details of such techniques are once again encouraged to read the paper on MiniSat [19]. Other algorithms that we tried were Glucose [4] and CryptoMiniSat [50].

The translation of the lock-chart problem into SAT uses the idea of "grounding". Every cutting depth at every position both in keys and locks will be represented by 1 propositional variable. Albeit simple and straightforward, the procedure handles all features from Chapter 2: extensions and melted profiles, which imply support for profiled lock-charts. Since vanilla and asymmetric frameworks are instances of the general framework, the only unsupported framework is the explicit one.

NOTATION. This section will translate arbitrary extension melted-profiles lock-chart (K, L, E, B) with a partial solution $\hat{s} \cup \hat{t}$ into a CNF. For convenience, variable k will iterate over all keys K, l over all locks L, i over positions $\{1, \dots, p\}$ and j over all cutting depths $\{1, \dots, d\}$ if not stated otherwise.

KEYS AND CUTTINGS. Each position in a key must be assigned a cutting depth. Hence, for each position, we allocate d propositional variables and enforce only 1 of them to be true in every model. In coding theory, this is known as "1-of-N code".

The variable, which encodes cutting depth j on position i in key k, will be denoted $\mathsf{key}^k_{i,j}$. At least one such variable must be true on every position. Given k and i, the requirement is expressed in the clause

$$\mathsf{key}^k_{i,1} \vee \mathsf{key}^k_{i,2} \vee \dots \vee \mathsf{key}^k_{i,d} \ . \qquad (4.1)$$

Also, at most one variable must be true for every key k and position i. A clause for every two distinct cutting depths j and j' forbids any two cutting depths to be active at the same time:

$$\mathsf{key}^k_{i,j} \Rightarrow \overline{\mathsf{key}}^k_{i,j'} \tag{4.2}$$

In total there are $|K| \cdot p \cdot d$ variables and $|K| \cdot p \cdot \left(1 + \frac{d \cdot (d-1)}{2}\right)$ clauses.[3]

LOCKS AND CYLINDERS. A cylinder is encoded using the same idea as a key. We assign $p \cdot d$ propositional variables denoted as $\mathsf{lock}^l_{i,j}$. However, in this case, the 1-of-N constraints are not needed, because a cylinder can have an arbitrary number of cutting depths in each position. This gives $|L| \cdot p \cdot d$ variables and no clause.

OPENING. If a key k enters lock l, then its cutting must be among the shear-lines of the cylinder. We do so by including clauses

$$\mathsf{key}^k_{i,j} \Rightarrow \mathsf{lock}^l_{i,j} \tag{4.3}$$

for each position and cutting depth. This generates $p \cdot d \cdot |E|$ clauses.

There is one brawback of this formula. The implications allow models to add pins into locks even if no key needs it, which violates Proposition 16. As a counter-measure, we can replace the implication by an equivalence

$$\left(\bigvee_{k \in E(l)} \mathsf{key}^k_{i,j} \right) \Leftrightarrow \mathsf{lock}^l_{i,j} \ . \tag{4.4}$$

for each position and cutting depth, which generates $p \cdot d \cdot (|E| + |L|)$ clauses. Also note that the expansion of the equivalence (4.4) into a CNF will generate the implications (4.3).

However, any difference in SAT solvers' performance must be tested empirically. The equivalence generates a smaller search-space, because the lock variables are fully determined by the key variables. On the other hand, the additional clauses might clog data structures of a solver's engine and slow down its inference.

BLOCKING. Supppose a key k is blocked in lock l. Then there must be 1 position, where the cutting depth of the key is blocked

[3] Notice that $\mathsf{key}^k_{i,j} \Rightarrow \overline{\mathsf{key}}^k_{i,j'}$ and $\mathsf{key}^k_{i,j'} \Rightarrow \overline{\mathsf{key}}^k_{i,j}$ are two identical clauses. This explains the $\frac{1}{2}$ term.

by a missing pin in the lock. First, we define auxiliary variables that detect the blocking

$$\mathsf{key}^{k}_{i,j} \wedge \overline{\mathsf{lock}}^{l}_{i,j} \Rightarrow \mathsf{block}^{k,l}_{i,j} \tag{4.5}$$

for each position, cutting depth, lock and a stopped key. The key' cutting is blocked if at least one blocking variable is true

$$\bigvee_{i,j} \mathsf{block}^{k,l}_{i,j} \tag{4.6}$$

which gives $(pd+1) \cdot |B|$ clauses and $pd \cdot |B|$ variables.

Note that the auxiliary blocking variables appear a positive literal in both types of clauses. Therefore they cannot be removed by resolution, which is a common pre-processing step in SAT solving.

Similarly to the previous case, it is possible to replace the implications in (4.5) by equivalences.

PARTIAL SOLUTION. Extension lock-chart's solution must satisfy the partial solution $\hat{s} \cup \hat{t}$: In every solution, keys' cuttings and lock's cylinders must remain the same as in $\hat{s} \cup \hat{t}$. This is achieved by adding unitary clauses $\mathsf{lock}^{l}_{i,j}$ if $\hat{t}(l)_i = j$ and its negation $\overline{\mathsf{lock}}^{l}_{i,j}$ otherwise. The same idea applies to keys, even though the negative literals may be implied by unit propagation in DPLL algorithms. To see this, notice that a unitary clause $\mathsf{key}^{k}_{i,j}$ renders left side of the implication (4.2) true and hence the right side $\overline{\mathsf{key}}^{k}_{i,j'}$ must be also true for all $j' \neq j$.

The DPLL algorithms have a great advantage here. Since unit propagation affects all clauses in the CNF, the solver only deals with variables, literals and clauses not determined by the partial solution $\hat{s}$. In other words, the complexity of solving the generated CNF correlates with the number of unassigned keys, not the number of all keys. Consequently, DPLL solvers are very strong at finding small extensions of arbitrarily large lock-charts, which can be confirmed from our experience.

Including the redundant literals, there are $p \cdot d \cdot |\hat{s}|$ generated unit clauses.

CONSTRAINTS. This straightforward translation is well suited for incorporating mechanical constraints. One gecon translates to one clause for each key. Let $(c_1, \ldots, c_p)$ be a gecon with $p - r$ wildcards and $i_1, \ldots, i_r$ be positions, where the gecon has non-wildcard values. The clause

$$\overline{\mathsf{key}}^{k}_{i_1,c_{i_1}} \vee \overline{\mathsf{key}}^{k}_{i_2,c_{i_2}} \vee \cdots \vee \overline{\mathsf{key}}^{k}_{i_r,c_{i_r}} \tag{4.7}$$

	Variables		**Clauses**	
	Name	Allocated	$\Rightarrow$	$\Leftrightarrow$
keys	key	$pd \cdot \lvert K \rvert$	$p \cdot \left(1 + \frac{d \cdot (d-1)}{2}\right) \cdot \lvert K \rvert$	
locks	lock	$pd \cdot \lvert L \rvert$	0	
enterings			$pd \cdot \lvert E \rvert$	$pd \cdot (\lvert E \rvert + \lvert L \rvert)$
blockings	block	$pd \cdot \lvert B \rvert$	$(pd+1) \cdot \lvert B \rvert$	$(3pd+1) \cdot \lvert B \rvert$
constraints			number of gecons $\cdot \lvert K \rvert$	
extension			$pd \cdot \lvert \hat{s} \rvert$	

Table 4.1: Size of the straightforward translation. The number of clauses is given for both implication and equivalence models.

ensures that the cutting differs from the gecon on at least 1 non-wildcard position. This exactly follows the definition of gecons.

Since the asymmetric framework is a special case of the general framework, asymmetric constraints are covered by this approach as well. See the last paragraphs of Section 2.2 for details. Given that the gecons (2.2) translate to a unitary clause, they take advantage of unitary propagation.

The problem with adapting the explicit framework into the general one and the need for an exponential translation procedure is described in the same place.

Definition 54. Assume a general framework and let (K, L, E, B) be a melted profiles lock-chart and $\hat{s}$ its partial solution. The CNF which composes of all clauses (4.1)-(4.7) is called the lock-chart's *straightforward translation*.

Proposition 55. *The straightforward translation is satisfiable if and only if the translated lock-chart has a solution.*

This proposition is left without a formal proof. There is a 1:1 correspondence between every definition of the lock-chart problem and a clause in the straightforward translation; hence a proof would provide no additional insight.

Given this proposition and a model to the CNF, one can easily recreate a solution to the original lock-chart. For every key and position $s(k)_i = j$ if and only if $\mathcal{I}\left(\mathsf{key}^k_{i,j}\right) = 1$.

SUMMARY. Table 4.1 contains the size of the CNF generated by the straightforward translation. Since real-world lock-charts (for example diagonal lock-charts) have $|B| \sim |K| \cdot |L|$, we can see that the blocking clauses and variables dominate all other types.

EVALUATION. Before the SAT translation algorithm shows its strengths, which comes no earlier than in Chapter 6, I would like to discuss its limitations. The theoretical limits have been described in Chapter 3, but what about the practical ones?

As the simplest benchmark, we picked vanilla diagonal lock-charts, because such a benchmark only depends on 3 parameters – p, d and the number of individual keys. All results can be summarised in a single table.

Implementation-wise, we used a C++ binary API instead of exporting the CNFs to the DIMACS format. The API allows incremental solving, which was used for adding keys and locks one by one, until a the entire lock-chart was solved, the 1 hour timeout occurred or an out-of-memory error was raised.

One may question such benchmark on the grounds of "torturing" a SAT solver on a polynomial instance. Indeed, if the SAT solver could somehow discover the solution, which was proved optimal in Theorems 45 and 52, such a benchmark would certainly be meaningless.

We still believe that such a benchmark is a reasonable estimate of SAT solvers' performance on real-world lock-charts with many constraints. SAT solvers rarely attempt at analyzing its input or discovering symmetries in the search-space. They are designed and tuned for solving $\mathcal{NP}$-hard problems, largely relying on their brute-force performance.

Tables 4.2 and 4.3 show the results. They were obtained by using the MiniSat library on an Intel Core i5 CPU running at 2.70 GHz with 16 GiB RAM. The CNFs were obtained using the implication model both for lock and block variables.[4]

Until the diagonal lock-chart has 700 individual keys or less, the solver is very likely to find a solution. Apparently the solver's performance peaks at around 1700 keys. With increasing p and d, it fails to exploit the larger code space and even the absolute numbers tend to decrease.

The main reason for dwindling performance in large code spaces is memory depletion. Most instances, which did not achieve 100% score, ended up on an out-of-memory error. This can be attributed to the fact that DPLL-based algorithms rely on many clever data structures. For example, every variable is associated with a list of clauses, in which it appears [19] to speed up propagation. Since all relevant formulas in Table 4.1 have the pd factor and $|B|$ scales quadratically with $|K|$, MiniSat runs out of memory faster with larger code space.

[4] During internal testing, we found out that MiniSat beats Glucose and Crypto-MiniSat and that the implication model beats equivalence mode.

	d = 2	d = 3	d = 4	d = 5	d = 6	d = 7	d = 8	d = 9	d = 10	d = 11	d = 12	d = 13	d = 14	d = 15	d = 16
p = 1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
p = 2	2	4	9	16	25	36	49	64	81	100	121	144	169	196	225
p = 3	3	12	27	64	125	216	343	512	729	1000	*1327*	*1486*	*1451*	*1418*	*1372*
p = 4	6	32	108	256	*621*	*1293*	*1597*	*1556*	*1506*	*1436*	*1374*	*1321*	*1272*	*1229*	*1189*
p = 5	10	80	405	*1265*	*1662*	*1610*	*1506*	*1420*	*1347*	*1284*	*1229*	*1181*	*1138*	*1099*	*1063*
p = 6	20	240	*1378*	*1704*	*1587*	*1469*	*1399*	*1296*	*1229*	*1172*	*1122*	*1078*	*1039*	*1003*	*970*
p = 7	35	672	*1749*	*1610*	*1469*	*1393*	*1296*	*1200*	*1138*	*1085*	*1039*	*998*	*962*	*929*	*898*
p = 8	70	*1755*	*1749*	*1506*	*1418*	*1303*	*1190*	*1122*	*1064*	*1015*	*972*	*936*	*901*	*869*	*840*
p = 9	*122*	*1832*	*1681*	*1420*	*1337*	*1229*	*1122*	*1058*	*1003*	*957*	*921*	*880*	*848*	*819*	*792*
p = 10	252	*1836*	*1594*	*1404*	*1268*	*1138*	*1064*	*1003*	*952*	*915*	*874*	*835*	*804*	*777*	*751*
p = 11	462	*1778*	*1436*	*1339*	*1209*	*1085*	*1015*	*957*	*918*	*872*	*828*	*796*	*767*	*741*	*716*
p = 12	*911*	*1702*	*1452*	*1282*	*1158*	*1039*	*972*	*929*	*878*	*835*	*793*	*762*	*734*	*709*	*686*
p = 13	*1565*	*1635*	*1400*	*1232*	*1113*	*998*	*951*	*893*	*844*	*796*	*762*	*732*	*705*	*681*	*659*
p = 14	*1980*	*1581*	*1349*	*1187*	*1072*	*962*	*916*	*860*	*813*	*773*	*734*	*705*	*680*	*657*	*635*
p = 15	*1849*	*1528*	*1303*	*1147*	*1036*	*952*	*885*	*831*	*786*	*741*	*709*	*681*	*657*	*634*	*613*
p = 16	*1847*	*1479*	*1262*	*1111*	*972*	*922*	*857*	*805*	*761*	*717*	*687*	*660*	*636*	*614*	*594*

Table 4.2: Number of individual keys in a vanilla diagonal lock-chart, which were found by the MiniSat algorithm. Values in italics correspond to runs, which ended by a timeout or by an out-of-memory error.

	$d = 2$	$d = 3$	$d = 4$	$d = 5$	$d = 6$	$d = 7$	$d = 8$	$d = 9$	$d = 10$	$d = 11$	$d = 12$	$d = 13$	$d = 14$	$d = 15$	$d = 16$
$p = 1$	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
$p = 2$	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
$p = 3$	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	86%	66%	52%	41%
$p = 4$	100%	100%	100%	100%	99%	100%	67%	38%	23%	14%	9%	6%	4%	3%	2%
$p = 5$	100%	100%	100%	99%	53%	21%	9%	4%	2%	1%	10^{-2}	10^{-2}	10^{-3}	10^{-3}	10^{-3}
$p = 6$	100%	100%	95%	28%	8%	3%	1%	10^{-2}	10^{-3}	10^{-3}	10^{-3}	10^{-3}	10^{-4}	10^{-4}	10^{-4}
$p = 7$	100%	100%	34%	6%	1%	10^{-2}	10^{-3}	10^{-3}	10^{-4}	10^{-4}	10^{-4}	10^{-5}	10^{-5}	10^{-5}	10^{-5}
$p = 8$	100%	98%	9%	1%	10^{-3}	10^{-3}	10^{-4}	10^{-4}	10^{-5}	10^{-5}	10^{-5}	10^{-6}	10^{-6}	10^{-6}	10^{-6}
$p = 9$	97%	34%	2%	10^{-3}	10^{-3}	10^{-4}	10^{-5}	10^{-5}	10^{-6}	10^{-6}	10^{-6}	10^{-7}	10^{-7}	10^{-7}	10^{-8}
$p = 10$	100%	12%	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-6}	10^{-7}	10^{-7}	10^{-7}	10^{-8}	10^{-8}	10^{-9}	10^{-9}
$p = 11$	100%	4%	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-6}	10^{-7}	10^{-8}	10^{-8}	10^{-9}	10^{-9}	10^{-9}	10^{-10}	10^{-10}
$p = 12$	99%	1%	10^{-3}	10^{-5}	10^{-6}	10^{-7}	10^{-7}	10^{-8}	10^{-9}	10^{-9}	10^{-10}	10^{-10}	10^{-11}	10^{-11}	10^{-11}
$p = 13$	91%	10^{-2}	10^{-4}	10^{-5}	10^{-7}	10^{-7}	10^{-8}	10^{-9}	10^{-10}	10^{-10}	10^{-11}	10^{-11}	10^{-12}	10^{-12}	10^{-12}
$p = 14$	58%	10^{-3}	10^{-5}	10^{-6}	10^{-7}	10^{-8}	10^{-9}	10^{-10}	10^{-11}	10^{-11}	10^{-12}	10^{-12}	10^{-13}	10^{-13}	10^{-14}
$p = 15$	29%	10^{-3}	10^{-5}	10^{-7}	10^{-8}	10^{-9}	10^{-10}	10^{-11}	10^{-12}	10^{-12}	10^{-13}	10^{-13}	10^{-14}	10^{-14}	10^{-15}
$p = 16$	14%	10^{-4}	10^{-6}	10^{-8}	10^{-9}	10^{-10}	10^{-11}	10^{-12}	10^{-13}	10^{-13}	10^{-14}	10^{-15}	10^{-15}	10^{-16}	10^{-16}

Table 4.3: Percentage of individual keys solved in a vanilla diagonal lock-chart by MiniSat. Value 100% means all invidiual keys in the largest solvable lock-chart $|S_{\hat{q}}|$.

Beside SAT solvers there are ILP solvers, which are also employed in hard combinatorial tasks. This section summarises our reasoning about writing a similar translation procedure from the lock-chart problem to ILP.

Question 56. *Can ILP solvers perform better than SAT solvers at solving the lock-chart problem?*

Because $\mathcal{NP}$-hard solving algorithms are tricky to compare, a definitive answer needs an experimental evaluation. Nevertheless, we decided to skip it in order to focus on domain-specific algorithms. Still, we think there are good reasons to believe ILP solvers stand a good chance.

First, there is a pragmatic argument. Libraries such as CPLEX [30] or Gurobi [24] are big commercial endeavours. Merely judging from the code-base size (current binaries take up between 25 MB and 100 MB of disk space) and commercial price (Gurobi license costs between \$ 14 000 and \$ 56 000),[5] these libraries have been invested a lot of endeavour.

Second, the CNF obtained by the translation procedure from the previous section can be translated once again into a binary ILP instance easily. Assume that for every propositional variable in the CNF, there is one binary ($0 \leqslant x \leqslant 1$) integer variable in the ILP instance. A clause

$$C = \bar{x}_1 \vee \bar{x}_2 \vee \cdots \vee \bar{x}_i \vee x_{i+1} \vee x_{i+2} \vee \cdots \vee x_j \tag{4.8}$$

is satisfied under an interpretation $\mathcal{I}$ if $\mathcal{I}(C) = 1$. This condition is implied by an ILP constraint

$$(1 - x_1) + (1 - x_2) + \cdots + (1 - x_i) + x_{i+1} + x_{i+2} + \cdots + x_j \geqslant 1 \,. \tag{4.9}$$

This “retranslation” of the lock-chart problem might not be the most efficient encoding, but is certainly an easy start given that the procedure for generating CNF is implemented.

INITIAL SOLUTION. Next, consider a common technique to attack ILP problems: First, the “integer” constraint is relaxed, all variables are considered real, which gives an *LP relaxation*. Unlike the original problem, the relaxation is in $\mathcal{P}$ [36] and can be solved by many existing algorithms quickly. Given a solution to the relaxation, ILP solver starts the backtracking procedure by rounding off the variables.

[5] http://www.gurobi.com/products/licensing-pricing/commercial-pricing

Given a good optimisation criterion, the LP relaxation can bring the backtracking part close to an integer solution, which might reduce the algorithm's runtime. At least for diagonal lock-charts, that clog MiniSAT, a good criterion respects the "equal to the general key" value q from Section 3.6.

Assume that the general key is g and take any other key k. If the cuttings of the two keys differ on position i, then

$$\sum_{j=1}^{d} \left| \mathsf{key}^{g}_{i,j} - \mathsf{key}^{k}_{i,j} \right| = 2 \tag{4.10}$$

Then by summing over all positions and all keys and rephrasing the constraint as an optimisation criterion, the c vector can be formed to minimise

$$\left| \left(\sum_{k \in K \setminus \{g\}} \sum_{i=1}^{p} \sum_{j=1}^{d} \left| \mathsf{key}^{g}_{i,j} - \mathsf{key}^{k}_{i,j} \right| \right) - 2 \cdot q \cdot |K| \right| \tag{4.11}$$

This criterion is becomes linear by replacing every absolute value by auxiliary variables [10]. The absolute values will generate $pd \cdot |K|$ auxiliary variables, which is still linear in the number of keys.

COMPRESSION. Finally, the blocking constraints, which are behind MiniSat's high memory consumption, can be encoded more efficiently. To ensure a blocking on at least 1 position, the following constraint can be formulated:

$$\sum_{i,j} \mathsf{key}^{k}_{i,j} \cdot (1 - \mathsf{lock}^{l}_{i,j}) \geqslant 1 \tag{4.12}$$

The constraint is quadratic, which calls for quadratic programming. If for whatever reason, quadratic programming was undesirable, every binary quadratic integer program can be translated into a binary ILP [9].

5 CUTTING COUNTING

The cutting counting problem occurs when new mechanical platforms are created. While fiddling with the physical layout of cuttings and cylinders, the number of positions, cutting depths and mechanical constraints, engineers need to know how their design choices affect master-keyed systems. An algorithm for counting key cuttings does exactly that.

The counting problems are also related to some theoretical questions, because several problems in lock-chart solving reduce to counting problems. For example, finding a solution to a key-to-differ lock-chart is as hard counting solutions to a 1×0 lock-chart (see Section 3.3). Out of many similar questions, by *cutting counting*, we specifically mean the following question:

Question 57. *How to calculate the size of the code space $|S|$ or the size of the largest solvable diagonal-lock chart?*

The first major result on cutting counting is the formula (3.1) in the vanilla framework, which gives the size of largest solvable diagonal lock-chart as $|S_{\hat{q}}|$. Besides that, the simple formula $|S| = d^p$ can also be considered as a trivial result on count cutting. This chapter extends such results to the remaining constraint frameworks.

5.1 ASYMMETRIC FRAMEWORK

Definition 58. Let $(\tilde{d}_1, \dots, \tilde{d}_p)$ be the deepest cutting. The function $\Gamma : \{0, \dots, p\} \times T \times T \to \mathbb{W}$, denoted $\Gamma(q, E, B)$, counts the number of cuttings, which have exactly q cutting depths present in the cylinder E (on respective positions) and which are blocked in the cylinder B. Formally

$$\Gamma(q, E, B) = \left\{\gamma \in S \;\middle|\; B \text{ blocks } \gamma \text{ and } q = q^E_\gamma\right\}, \text{ where} \tag{5.1}$$

$$q^E_\gamma = \sum_{1 \leqslant i \leqslant p} \begin{cases} 1 & \text{if } \gamma_i \in E_i \\ 0 & \text{if } \gamma_i \notin E_i \end{cases} \tag{5.2}$$

This function has many applications. For a start, it answers the cutting counting question in the asymmetric framework:

1. Let $q = 0$ and $E = B = (\overbrace{\emptyset, \dots, \emptyset}^{p \text{ times}})$, which will be written concisely as $E = B = \emptyset$ in this chapter. Then every cutting γ is blocked in B and $q^E_\gamma = 0 = q$. Therefore every cutting satisfies the conditions and Γ function calculates the size of the code space $|S|$:

   $$\Gamma(0, \emptyset, \emptyset) = \tilde{d}_1 \cdot \tilde{d}_2 \cdot \ldots \cdot \tilde{d}_p = |S| \,. \tag{5.3}$$

   In the vanilla framework, all cutting depths of the deepest cutting $\tilde{d}$ are equal, and hence $\Theta(p, 0, \emptyset, \emptyset) = d^p = |S|$.

2. By definition, value $\Gamma(q, (\overbrace{\{1\}, \dots, \{1\}}^{p \text{ times}}), \emptyset)$ is the number of cuttings equal to any general key[1] on exactly q positions, a number known as $|S_q|$. By Theorem 45, which holds in all frameworks, the value $\max_q |S_q|$ is the lower bound on the size of the largest solvable diagonal lock-chart. In the vanilla framework, the Theorem 52 holds and hence the bound is tight.

Through this section, we will be finding a faster way to evaluate the Γ function. First, a recurrence relation is formulated, which avoids a brute-force iteration over all cuttings $\gamma \in S$. Its evaluation will reduce the number of operations from the order of d^p to 4^p.

Lemma 59. *The* $\Gamma(q, E, B)$ *function is equal to* $\Theta(p, q, E, B)$*, where*

$$\Theta(i, q, E, B) = \begin{cases} 0 & \textit{if } q < 0 \textit{ or } i < q \\ |E_1 \setminus B_1| & \textit{if } i = 1 \textit{ and } q = 1 \\ \tilde{d}_1 - |E_1 \cup B_1| & \textit{if } i = 1 \textit{ and } q = 0 \\ \theta(i, q, E, B) & \textit{otherwise} \end{cases} \tag{5.4}$$

$$\begin{aligned} \theta(i, q, E, B) = |B_i \setminus E_i| \cdot \Theta(i-1, q, E, B) + & \quad \textit{(case 1)} \\ + |B_i \cap E_i| \cdot \Theta(i-1, q-1, E, B) + & \quad \textit{(case 2)} \\ + |E_i \setminus B_i| \cdot \Theta(i-1, q-1, E, \emptyset) + & \quad \textit{(case 3)} \\ + (\tilde{d}_i - |E_i \cup B_i|) \cdot \Theta(i-1, q, E, \emptyset) & \quad \textit{(case 4)} \end{aligned} \tag{5.5}$$

The terminating condition is defined in the formula (5.4) and the recursion in (5.5). Note that the θ function is merely a substitution for better typography on a narrow page layout.

Before a general proof, we feel obliged to do a "sanity check".

[1] In the asymmetric framework, the size $|S_q|$ is constant regardless of the general key's cutting. The reasoning is similar to the proof of Theorem 46.

Example 60. Let $p = 3$, $d = 2$ (deepest cutting is $(2,2,2)$ in the asymmetric framework), $q = 2$, $E = (\{1\},\{1\},\{1\})$ and $B = (\{1,2\},\{1\},\{2\})$. The expected value of $\Theta(p,q,E,B)$ is 2, because only cuttings $(1,1,2)$, $(1,2,1)$ and $(2,1,1)$ have $q = 2$ cuttings depths within E, but 1 of them enters B, namely $(1,1,2)$.

$$\Theta(3,2,E,B) = \overbrace{1\cdot\Theta(2,2,E,B)}^{\text{by case 1}} + \overbrace{1\cdot\Theta(2,1,E,\emptyset)}^{\text{by case 3}} =$$

$$= \left(\overbrace{1\cdot\Theta(1,1,E,B)}^{\text{by case 2}} + \overbrace{1\cdot\Theta(1,2,E,\emptyset)}^{\text{by case 4}} \right) +$$

$$+ \left(\overbrace{1\cdot\Theta(1,0,E,\emptyset)}^{\text{by case 3}} + \overbrace{1\cdot\Theta(1,1,E,\emptyset)}^{\text{by case 4}} \right) =$$

$$= \left(\overbrace{|\{1\}\setminus\{1,2\}|}^{\text{by } i=q=1} + \overbrace{0}^{\text{by } i<q} \right) + \left(\overbrace{2-|\{1\}\cup\emptyset|}^{\text{by } i=1 \text{ and } q=0} + \overbrace{|\{1\}\setminus\emptyset|}^{\text{by } i=q=1} \right) = 2$$

A second "sanity check" is done by checking two special cases in the vanilla framework:

1. The code space size $|S|$ is given by $\Theta(p,0,\emptyset,\emptyset)$. This follows from $E_i = B_i = \emptyset$, which makes the factors in cases 1 to 3 evaluate to 0. Case 4 multiplies $\tilde{d}_i - |E_i \cup B_i| = d$ exactly $p-1$ times and the terminating condition adds the last d^{th} factor. Therefore $\Theta(p,0,\emptyset,\emptyset) = d^p$.

2. When deriving $|S_q| = \Theta(p,q,(\{1\},\ldots,\{1\}),\emptyset)$, cases 1 and 2 do not apply, because $B_i = \emptyset$. First note that the Θ function is a sum of products. Every call sequence picks $p-1$ times either case 3 or 4, arrives at some terminating condition and yields one product to the sum. If we were to arrive at the terminating condition $q = 1$, case 3 must be picked exactly $q-1$ times. Hence the $|\tilde{d}_i - \{1\}| = d-1$ value is multiplied in the remaining $p-q$ invocations of case 4. This gives the $(d-1)^{p-q}$ factor, which is obtained $\binom{p-1}{q-1}$-many times. If we were to arrive at the terminating condition $q = 0$, a similar reasoning gives the same factor $\binom{p-1}{q}$-many times. The sum of these two factors gives formula (3.1), because

$$\left[\binom{p-1}{q-1} + \binom{p-1}{q}\right] \cdot (d-1)^{p-q} = \binom{p}{p-q} \cdot (d-1)^{p-q}\,.$$

Proof of the lemma. The first parameter i iterates over all positions and lets the function "restrict all reasoning" only to positions $\{1,\ldots,i\}$ and ignore the remaining positions $\{i+1,\ldots,p\}$. Every recursive step “extends” the reasoning to cuttings one position longer. The proof is done by induction on i.

The base case assumes $i = 1$: The θ function is never invoked. Hence only the first 3 lines of (5.4) are proved. There is only 1 position to block a cutting, hence cutting depths B_1 are never counted. The second and third line count remaining depths either inside (resp. outside) E_i when $q = 1$ (resp. $q = 0$).

The inductive step proves correctness of $\Theta(i, q, E, B) = \theta(i, q, E, B)$ in (5.5) by assuming $\Theta(i-1, q, E, B)$ is correct. First note the partitioning of cutting depths. Every cutting depth on the i-th position is either in $E_i \cap B_i$, in $E_i \setminus B_i$, in $B_i \setminus E_i$ or outside both sets, of which there are $\tilde{d}_i - |E_i \cup B_i|$ many ones. Sizes of these sets are multiplicative factors in cases 1 to 4. This ensures that no cutting is counted twice.

Next, take any cutting γ which matches E on q positions between 1 and i. If $\gamma_i \in E_i$ then the cutting matches E on $q - 1$ positions between 1 and $i - 1$ (cases 2 and 3). Otherwise, the q remains the same between the recursive steps (cases 1 and 4). Hence the recursive formula does not depend on values $q - 2$ or less and no cutting is left out.

Similarly if the cutting depth $\gamma_i \in B_i$, the cutting is not blocked on the i-th position. Therefore it must have been blocked somewhere between positions 1 and $i - 1$. Therefore cases 1 and 2 extend $\Theta(i, *, E, B)$ cuttings. Cases 3 and 4 assume $\gamma_i \notin B_i$, which ensures (possibly additional) blocking on the i-th position and extend $\Theta(i, *, E, \emptyset)$. □

Having defined and understood Γ and Θ functions, there is the question of their evaluation. A third, equivalent, closed-form formula is unlikely to exist, and a naive execution of both functions is exponential in p. Observing that parameters i and q can only decrease during the execution of Θ calls for its evaluation via a simple dynamic programming scheme. Algorithm 5.1 implements the scheme and runs in quadratic time. The algorithm merely turns the terminating condition (5.4) into initialisation and then applies the recursive case (5.5). A better version would save memory (which implies better cache usage on modern CPUs) by eliminating the first dimension from the Θ array by observing that $\Theta(i, *, *)$ only depends on $\Theta(i - 1, *, *)$.

5.2 GENERAL FRAMEWORK

Having formulated a cutting counting procedure for the asymmetric framework, can the same be done for the general framework?

```
input  : The deepest cutting $(\tilde{d}_1, \dots, \tilde{d}_p)$, a whole number $q$
         and two cylinders $E, B$
output : Value of the $\Theta(1, q, E, B)$
Function count($(\tilde{d}_1, \dots, \tilde{d}_p), q, E, B$):
    declare $\Theta$ as an array of size $p \cdot (p+1) \cdot 2$, values
      initialised to $0$, for convenience indexed by
      $\{1, \dots, p\} \times \{0, \dots, p\} \times \{B, \emptyset\}$
    $\Theta(1, 1, B) \leftarrow |E_1 \setminus B_1|$
    $\Theta(1, 0, B) \leftarrow \tilde{d}_1 - |E_1 \cup B_1|$
    $\Theta(1, 1, \emptyset) \leftarrow |E_1|$
    $\Theta(1, 0, \emptyset) \leftarrow \tilde{d}_p - |E_1|$
    for $i \in \{2, \dots, p\}$ do
        for $q \in \{0, \dots, i\}$ do
            $\Theta(i, q, B) \leftarrow |B_i \setminus E_i| \cdot \Theta(i-1, q, B) +$
                        $+ |B_i \cap E_i| \cdot \Theta(i-1, q-1, B) +$
                        $+ |E_i \setminus B_i| \cdot \Theta(i-1, q-1, \emptyset) +$
                $+ (\tilde{d}_i - |E_i \cup B_i|) \cdot \Theta(i-1, q, \emptyset)$
            $\Theta(i, q, \emptyset) \leftarrow |E_i| \cdot \Theta(i-1, q-1, \emptyset) +$
                $+ (\tilde{d}_i - |E_i|) \cdot \Theta(i-1, q, \emptyset)$
        end
    end
    return $A(p, q, B)$
```

Algorithm 5.1: Cutting-counting algorithm for the asymmetric framework.

First, recall a fundamental limitation. Section 3.3 established that the problem is $\mathcal{NP}$-complete, namely, it sits in the $\#\mathcal{P}$ class, which kills any hope for a polynomial procedure.

SAT-BASED APPROACH. The SAT correspondence may yield the first practical algorithm, which would be easy to implement. Theorem 32 shows that every solution of the 1×0 lock-chart corresponds to one cutting in S. Can we take the straightforward translation of the 1×0 lock-chart to CNF (from Section 4.2) and use existing efficient libraries [8] to solve the #SAT problem?

Interested readers may try this approach, but we remain sceptical. There are several issues that would have to be addressed:

- Take the DPLL-Based Model Counters (discussed in Section 20.2.1 of [8]). Their idea is to obtain a partial solution $\mathcal{I} : X \rightharpoonup \{0, 1\}$, which do not have to assign all variables X, but stop once every clause has at least 1 positive literal. The remaining $|X| - |\mathcal{I}|$ variables can be assigned arbitrarily, and hence the partial solution captures $2^{|X|-|\mathcal{I}|}$ models. However, considering the CNF instances from Section 4.2, the solution cannot be partial ($|X| = |\mathcal{I}|$), because of the binary clauses coming from the "1-of-N encoding". If a variable $\mathsf{key}^k_{i,j}$ is assigned 1, all other variables $\mathsf{key}^k_{i,j'}$ (for $j \neq j'$) must be assigned 0, otherwise the binary clauses have no positive literal.
- Another strong idea of model counters is a *component analysis*. Connected components of the *constraint graph* [8] may be solved independently. Variables encoding different positions of a cutting indeed appear in different connected components, but only as long as they are not joined by a gecon, which is often the case. Hence this idea works well only in the asymmetric framework (for which there is a polynomial procedure).

On a different note, one may try solving a series of key-to-differ lock-charts with increasing size by a non-counting (aka standard) SAT-solver. Lemma 34 shows that a solution to the largest solvable key-to-differ lock-chart gives the set $|S|$. However, it is hard to believe that such a solver would get much further than ~ 2000 cuttings we obtained in Section 4.2 by MiniSAT on diagonal lock-charts. Instead, here we aim at results in the order of $\sim 10^7$.

INCLUSION-EXCLUSION PROCEDURE. The probably simplest form of the *inclusion-exclusion* principle says that given two sets A, B, their sizes are related as following:

$$|A \cup B| = |A| + |B| - |A \cap B|$$

The principle may be known from the high-school formula on the probability of two dependent events. Here, it will be applied to counting cuttings that are satisfied by a set of gecons.

Recall that a gecon is associated with a universal cylinder, formally defined in Lemma 37. A cutting satisfies the gecon if and only if the cutting is blocked in the associated universal cylinder. Therefore counting cuttings that satisfy a set of gecons is equivalent to the number of cuttings blocked in all associated universal cylinders.

Lemma 61. *Let* $C = (C_1, \ldots, C_p)$ *and* $D = (D_1, \ldots, D_p)$ *be two cylinders. Their intersection denoted as*

$$C \cap D = (C_1 \cap D_1, \ldots, C_p \cap D_p)$$

blocks exactly those cuttings that are blocked in both C *and* D*.*

Proof. A cutting $(c_1, \ldots, c_p)$ is blocked in C if there is $i \in \{1, \ldots, p\}$ s.t. $c_i \not\in C_i$. Therefore $c_i \not\in C_i \cap D_i$ and hence the cutting is blocked in $C \cap D$. If the cutting is blocked in $C \cap D$, then there is $i \in \{1, \ldots, p\}$ s.t. $c_i \not\in C_i \cap D_i$. Therefore either $c_i \not\in C_i$ or $c_i \not\in D_i$ and the cutting is blocked in C or D. □

Having defined an intersection of cylinders, we can count the number of blocked cuttings.

Corollary 62. *Let* C *and* D *be two cylinders. Number of cuttings blocked by both* C *and* D *is (I.) the number of cuttings blocked by* C plus *(II.) the number of cuttings blocked by* D minus *(III.) the number of cuttings blocked by* $C \cap D$*.*

Proof. If a cutting is blocked in the cylinder C only, it is only counted in the (I.) term. If a cutting is blocked by both C and D, it is counted in all 3 terms (I.), (II.) and (III.). Given the polarities, two of the terms cancel out. Hence the cutting is counted only once. Other cases are similar or trivial. □

The inclusion-exclusion principle generalises the corollary to an arbitrary number of cylinders.

Definition 63. Let $\bar{\Gamma}(q, E, \{B_1, \ldots, B_n\})$ be the generalised Γ function, which counts the number of cuttings, which have exactly q cutting depths present in the cylinder E (on respective positions) and which are blocked by all cylinders B_i. The inclusion-exclusion principle states

$$\bar{\Gamma}(q, E, \{B_1, \ldots, B_n\}) = \sum_{i=1}^{n} \Gamma(q, E, B_i) - \tag{5.6}$$

$$- \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Gamma(q, E, B_i \cap B_j) + \tag{5.7}$$

$$+ \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^{n} \Gamma(q, E, B_i \cap B_j \cap B_k) - \cdots \tag{5.8}$$

$$\cdots \Gamma(q, E, B_1 \cap \cdots \cap B_n) \, . \tag{5.9}$$

Example 64. Let $p = 3$, $d = 2$, $q = 2$, $E = (\{1\},\{1\},\{1\})$ and the three gecons are $(?, 1, 2)$, $(1, 1, ?)$ and $(?, 2, ?)$. The expected answer is $\bar{\Gamma}(q, E, \{B_1, B_2, B_3\}) = 1$, because from the 3 candidate cuttings $S_q = S_2 = \{(1, 1, 2), (1, 2, 1), (2, 1, 1)\}$ only $(2, 1, 1)$ satisfies all 3 gecons.

First state the universal cylinders associated with the 3 gecons:

$$B_1 = (\{1,2\},\{1\},\{2\}),\ B_2 = (\{1\},\{1\},\{1,2\})$$

$$\text{and } B_3 = (\{1,2\},\{2\},\{1,2\}) \, .$$

Let us save some work by noting that $B_2 \cap B_3 = (\{1\}, \emptyset, \{1,2\})$. Every cutting is blocked on the middle position, hence $\Gamma(q, E, B_2 \cap B_3) = \Gamma(q, E, \emptyset) = |S_q|$. The same is true for $B_1 \cap B_2 \cap B_3$, because the additional cylinder B_1 in the intersection cannot remove $\emptyset$ from the middle position. Therefore the terms $\Gamma(q, E, B_2 \cap B_3)$ and $\Gamma(q, E, B_1 \cap B_2 \cap B_3)$ cancel out:

$$\begin{aligned}
&\bar{\Gamma}(q, E, \{B_1, B_2, B_3\}) = \\
&\qquad = \Gamma(q, E, B_1) + \Gamma(q, E, B_2) + \Gamma(q, E, B_3) - \\
&\qquad\qquad - \Gamma(q, E, B_1 \cap B_2) - \Gamma(q, E, B_1 \cap B_3)
\end{aligned}$$

Example 60 already calculated $\Gamma(q, E, B_1) = 2$. By a similar calculation, by invoking `count(...)`, or by observing that both B_2 and B_3 block exactly 2 cuttings from S_2, we get $\Gamma(q, E, B_2) = \Gamma(q, E, B_3) = 2$. The cylinder $B_1 \cap B_2 = (\{1\},\{1\},\{2\})$ is entered only by the cutting $(1, 1, 2)$, hence $\Gamma(q, E, B_1 \cap B_3) = 2$. The cylinder $B_1 \cap B_3 = (\{1,2\}, \emptyset, \{2\})$ has the middle position empty, which yields $\Gamma(q, E, B_1 \cap B_3) = |S_q| = 3$. Substituting the values gives

$$\bar{\Gamma}(q, E, \{B_1, B_2, B_3\}) = 2 + 2 + 2 - 2 - 3 = 1 \, .$$

How to apply the formula to cutting counting?

1. **Code space size:** The code space size is given simply by

$$|S| = \sum_{q=0}^{p} \bar{\Gamma}(q, \emptyset, \{B_1, \ldots, B_n\}) , \tag{5.10}$$

   where B_i is the universal cylinder associated with the i-th gecon.

2. **Diagonal lock-charts:** Unfortunatelly, in the general framework, not all cutting depths are treated equally. Some may be more constrained by gecons, some less. Therefore not all general keys are equal and the lock-chart size depends on the general key's cutting. Assume for a moment that it is given as $(g_1, \ldots, g_p)$. Then the number of individual keys in a diagonal lock-chart is at least

$$\max_{q \in \{0,\ldots,p-1\}} \bar{\Gamma}(q, (\{g_1\}, \ldots, \{g_p\}), \{B_1, \ldots, B_n\}) \tag{5.11}$$

3. **Finding a good general key:** The last formula evaluates a cutting for the general key by the number of individual keys that can be added to its diagonal lock-chart. By sampling $(g_1, \ldots, g_p) \in S$, one can pick the one which maximizes the (5.11). However, the repeated evaluation of this formula may not be the fastest way to go. Faster algorithms are left for future research.

EVALUATION. The inclusion-exclusion procedure might explore all 2^n combinations of n gecons, which defines its worst-case time complexity. Depending on p, d and n, it may or may not be faster than searching d^p cuttings and checking against all constraints. In a joint project with Václav Voráček, an efficient procedure is currently being developed. The pseudo-code and details of the actual implementation will be published in his bachelor's thesis. Here is merely a short list of ideas for an efficient implementation and a short justification of the entire approach.

- **Pruning:** Once any intersection becomes empty, then its intersection with additional cylinders must yield an empty intersection. This idea was used in Example 64. A clever strategy inspired by the Apriori algorithm for mining itemsets [3] may avoid evaluating intersections, whose subsets are empty. From our experience, this idea is the major factor for the efficiency of the inclusion-exclusion principle.

- **Unitary constraints:** An unitary constraint is a gecon which has exactly 1 non-wildcard position. Such constraints can be eliminated in the same way as unitary propagation works in the DPLL algorithm.

- **Caching:** Assume $E = \emptyset$. Then the value of $\Gamma(0, \emptyset, B)$ depends purely on the number of cutting depths on each position, not their actual values. Hash-based caching can use a sorted array $[|B_1|, \ldots, |B_p|]$, which is shared among multiple different cylinders. This avoids redundant executions of the $\texttt{count}(..., B)$ procedure.

For a quick empirical evaluation, two real-world platforms were shortlisted, to which we had the complete list of gecons, and which had the highest known p and d values. Due to a non-disclosure agreement, we cannot report specific details. Speaking of approximate numbers, the first platform had 12 positions and between 6 and 7 cutting depths and the number of unconstrained cuttings (which satisfy merely $1 \leqslant d_i \leqslant \tilde{d}_i$) was $\tilde{d}_1 \cdots \tilde{d}_p \simeq 3 \cdot 10^9$. Such cuttings were constrained by ~ 70 gecons, most of which had 2 non-wildcard positions and the remaining ones had 3 non-wildcard positions. The second platform had 12 positions and between 2 and 4 cutting depths and the number of unconstrained cuttings was $\tilde{d}_1 \cdots \tilde{d}_p \simeq 108 \cdot 10^{15}$. The number of gecons was ~ 65 and they were structurally similar to the first platform.

The $\bar{\Gamma}$ function was evaluated on a 16-core Intel Xeon clocked at 3.10 GHz. On the first platform, the inclusion-exclusion approach was able to count $\sim 840 \cdot 10^6$ valid cuttings (and gave an exact value). The calculation took in 60 s and consumed $\sim$ 40 GB RAM. The results are summarised in Table 5.1. The table also indicate that 27% unconstrained keys indeed satisfy all gecons.

On the second platform, the calculation was not successful. For values $q < 25$, 128 GB RAM was not enough and resulted in an out-of-memory error. This might be surprising, since both platforms had similar constraints, both in their structure and total number. The reason is a less efficient pruning. With a larger p, more constraints might be intersected before reaching an empty cylinder.

The implementation is still evolving rapidly, and the presented results are still preliminary. Curious readers should wait for Václav Voráček's thesis to see the full potential of this algorithm. Nevertheless the $\bar{\Gamma}$ function implemented using the inclusion-exclusion principle will see an application in Section 6.3.

5.3 EXPLICIT FRAMEWORK

In the explicit framework, some counting problems become trivial. The set S is given as the algorithm's input and hence the code space size $|S|$ is obtained by a linear scan that adds 1 for

q	$\lvert S_q \rvert$	runtime
0	$64 \cdot 10^6$	7996 ms
1	$200 \cdot 10^6$	7298 ms
2	$260 \cdot 10^6$	7832 ms
3	$210 \cdot 10^6$	7044 ms
4	$110 \cdot 10^6$	6734 ms
5	$40 \cdot 10^6$	6754 ms
6	$11 \cdot 10^6$	3967 ms
7	$2 \cdot 10^6$	4011 ms
8	280 000	1755 ms
9	27 000	483 ms
10	1 700	144 ms
11	60	68 ms
12	1	64 ms
$\sum$	$840 \cdot 10^6$	60 s

Table 5.1: Code space size on a real-world platform with $p = 12$ and $\tilde{d}_i$ between 3 and 4. Values are rounded to 2 places.

every cutting in S. Also, given a cutting for the general key γ_g and a value q, the algorithm scans S and counts only those cuttings that are equal to γ_g on exactly q positions. Such linear algorithm gives the $|S_q|$ size, which is also a lower bound on the largest solvable diagonal lock-chart. A quadratic algorithm might consider all choices $\gamma_g \in S$ and pick the best cutting for the general key as:

$$\underset{\gamma_g \in S,\ 0 \leqslant q \leqslant p}{\arg\max} \quad |S_q| \,. \tag{5.12}$$

Question 65. *Are there diagonal lock-charts with more individual keys than the formula (5.12) gives?*

First, note the principal limitation. The answer to the question can only be based on the violation of Theorem 52. In other words, if all d^p cuttings from a vanilla framework were given in S explicitly, the Theorem would hold, and formula (5.12) would give the size of the largest diagonal lock-charts. Better algorithms must be able to exploit asymmetries in the code space.

TEMPLATE LOCK-CHARTS. One such algorithm, which will be described in a moment, is suited for a bigger class of lock-charts than the diagonal ones. It will be described in its general form and then applied to diagonal lock-charts as a special case.

Definition 66. The *template lock-chart* is a lock-chart (K, L, E) and the *expansion function* $e : L \to \mathbb{W}$ that assigns a number of individual keys to each lock in the template. Let $K = \{k_1, \dots, k_m\}$

and $L = \{l_1, \dots, l_n\}$. A template lock-chart is equivalent to the *expansion lock-chart* (K', L', E') s.t.

$$K' = \{k_1, \dots, k_m, \overbrace{k_1^1, \dots, k_1^{e(l_1)}}^{\text{expansion of } l_1}, \dots, \overbrace{k_n^1, \dots, k_n^{e(l_n)}}^{\text{expansion of } l_n}\}\,,$$

$$L' = \{\underbrace{l_1^1, \dots, l_1^{e(l_1)}}_{\text{expansion of } l_1}, \dots, \underbrace{l_n^1, \dots, l_n^{e(l_n)}}_{\text{expansion of } l_n}\}\,.$$

The keys k_i from the template will be referred as *master* keys and the additional keys k_i^x as *individual* keys. The lock l_j serves as the *prototype* for all individual keys $k_j^1, \dots, k_j^{e(l_j)}$ and locks $l_j^?$.

The edges E' are defined as follows: Individual key k_i^x opens lock l_j^y if and only if $i = j$ and $x = y$. Master key k_i opens lock l_j^y, formally $(k_i, l_j^y) \in E'$, if and only if k_i opens l_j in the template lock-chart $(k_i, l_j) \in E$.

Figure 5.1 shows an example of a template and its expansion.

A so-called individual key k_i^x in the expansion is always an individual key according to the Definition 11. This is because k_i^x opens l_i^x only. Master keys are more complicated.

A so-called master key k_i is also a master key by Definition 11 if and only if $e(l) \geqslant 2$ for some of opened locks $l \in E(k_i)$. For example k_1 in Figure 5.2 is not a master key, because $e(l_1)$ is only 1. You may also notice $E(k_1) = E(k_1^1)$, which violates Assumption 25.

However, such problems are only terminological and do not affect the algorithm presented here. To avoid confusion, by master and individual keys in this section we refer to Definition 66.

Remark 67. A diagonal lock-charts with n individual keys is an extension of a 1×1 lock-chart $(\{g\}, \{l\}, \{(g, l)\})$ with expansion function $s(l) = n$.

MOTIVATION. Template lock-charts serve as a way to dissociate algorithms for master keys and individual keys (a similar idea can be found in [38]). In this section, we assume that all master keys have been assigned by a suitable algorithm and here an algorithm is proposed to maximize the total number of individual keys that can be added to the system

$$\sum_{l \in L} e(l)\,. \tag{5.13}$$

Loose chaining of two different algorithms usually breaks completeness or explodes the practical time efficiency. We admit this, but only for a portion of lock-charts. It is a common practice for human operators to assign the general key manually to save "good" cuttings for hard lock-charts. In such a scenario there are only $|S|^{\max_l |E(l)|-2}$ such solutions to the template lock-chart. Moreover, iterating over all assignments of master keys is practically feasible, because from on our experience, many lock-charts have a small number of them. Approximate algorithms that work reasonably fast are evaluated in Chapter 6.

TRANSLATION TO MIS. The algorithm proposed here takes a template lock-chart with its solution and generates an undirected graph. The maximum independent set on this graph maximizes (5.13).

Definition 68 (Independence graph)**.** Let (K, L, E) be a template lock-chart and s its solution. The *independence graph* is an undirected graph (N, H), whose nodes are pairs of locks and cuttings $N \subseteq L \times S$. The graph contains a vertex (l, γ) if $s(l) \cup \gamma$ blocks all cuttings $s(k)$ of stopped keys $k \in K \setminus E(l)$. There is an edge between (l, γ) and (l', γ') if either γ' enters $s(l) \cup \gamma$ or γ enters $s(l') \cup \gamma'$.

Lemma 69. *Let (K, L, E) be a template lock-chart, s its solution and $I = \{(l_i, \gamma_x), (l_j, \gamma_y), \ldots\}$ be an independent set of the independence graph. Let (K', L', E') be an expansion from e, where*

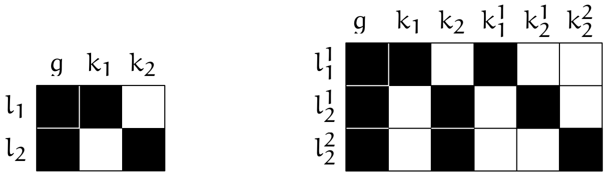
$$e(l) = |\{(l_i, \gamma_x) \in I \mid l_i = l\}| \ .$$

The assignment $s' = s \cup (k_i^x, \gamma_x) \cup (k_j^y, \gamma_y) \cup \cdots$ is a solution to (K', L', E').

Proof. Correctness is checked by inspecting all "empty cells" in the lock-chart. Take any lock l_i^x and an individual key k_j^y which is blocked in l_i^x ($i \neq j$ or $x \neq y$). By definition $s'(k_j^y) = \gamma_y$ and $s'(l_i^x) = s(l_i) \cup \gamma_x$. Since I is an independent set, vertices (l_i, γ_x) and (l_j, γ_y) not adjacent in (N, H) and therefore γ_y does not enter $s(l_i) \cup \gamma_x$. In other words, $s'(k_j^y)$ is blocked by $s'(l_i^x)$.

If a master key k_j should be blocked in l_i^x, then $s'(k_j) = s(k_j)$ must be blocked in $s'(l_i^x) = s(l_i) \cup \gamma_x$. If it were not, then by definition (l_i, γ_x) would not be a vertex in N. □

Example 70. Consider the lock-chart in Figure 5.1 in the vanilla framework $p = 2$, $d = 3$. The template is solved as follows: $s(k_G) = (1, 1)$, $s(k_1) = (2, 2)$ and $s(k_2) = (1, 3)$. The result is shown in Figure 5.2. Vertices with self-loops (vertices (l, γ), where γ is a shear-line of $s(l)$) were omitted from the figure, because they can never be a part of any independent set.



	g	k_1	k_2
l_1	■	■	□
l_2	■	□	■

	g	k_1	k_2	k_1^1	k_2^1	k_2^2
l_1^1	■	■	□	■	□	□
l_2^1	■	□	■	□	■	□
l_2^2	■	□	■	□	□	■

Figure 5.1: A template lock-chart (left) and its expansion from $e(l_1) = 1$ and $e(l_2) = 2$ (right).

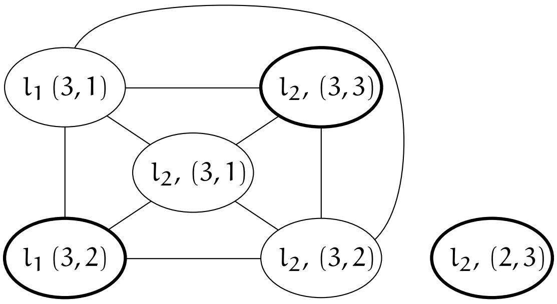


Figure 5.2: Example of an independence graph with a detailed description in Example 70. The maximum independent set is shown in bold.

Corollary 71. *Maximizing the number of individual keys, which can be added to a template lock-chart, can be done by finding the maximum independent set of the independence graph.*

Proof. This corollary is justified by the formula

$$\sum_{l \in L} e(l) = \sum_{l \in L} |\{(l_i, \gamma_x) \in I \mid l_i = l\}| = |I| \,,$$

because all sets in the sum are disjoint. □

Diagonal lock-charts are expansions of the 1×1 lock-chart defined in Remark 67. Hence finding the MIS of the independence graph created from 1×1 lock-chart constitutes the largest diagonal lock-chart and solves the cutting counting problem.

How to find I? The maximum independent set is an $\mathcal{NP}$-complete problem in general with an exponential best-known runtime. Our experience suggests a greedy approximation scheme, which picks the minimum degree vertex in each step [7, 25]. Since the guaranteed approximation ratio on graphs with a bounded degree is inverse proportional to the maximum degree in the independence graph (which is quite high), the idea should be evaluated empirically.

A final disclaimer. This translation is not a proof of $\mathcal{NP}$-hardness. If it were, the MIS would have to be translated into the problem of finding the largest diagonal lock-chart – not the other way round. In fact, whether the studied problem is in $\mathcal{P}$ or $\mathcal{NP}$ is still unknown to us and makes a good research question.

EVALUATION. Finally, the technique is evaluated by finding a solution to the largest diagonal lock-chart in the explicit framework. Here is the procedure to reproduce the experiments:

1. Uniformly sample $1 \leqslant p \leqslant 6$ and $2 \leqslant d \leqslant 8$.
2. Generate the S set using two different methods:
   a) In the *uniform* dataset there are no mechanical constraints and hence $|S| = d^p$. Theorem 52 holds in the uniform dataset, and formula (5.12) gives the size of the largest diagonal lock-chart.

   b) In the *realistic* dataset, mechanical constraints defined by our industrial partner were initialised to random values, and the size of S was reduced.[2]
3. Sample a uniform distribution over S to obtain the cutting of the general key $s(g)$.
4. Use a non-backtracking algorithm, which adds one individual key and a lock at a time. 1) Initialise the lock-chart with the general key g only and assign its cutting from the previous step. 2) Add 1 individual key k and a lock l, s.t. $E(l) = \{g, k\}$. 3) Pick a cutting from S by one of the evaluated heuristics, assign it to k. 4) If the solution is correct, reiterate from step 2. If not, find a different cutting in step 3. If there is no such cutting, terminate and report the size of the lock-chart.
   a) *Baseline*: The heuristic picks a random key from the uniform distribution over S.

   b) *Same-As-General heuristics* (SAG): First sort S_qs by their cardinality.[3] In step 3), start with keys from a larger S_q. Among keys with the same S_q, choose randomly. The result on the uniform dataset is guaranteed to be optimal.
5. Generate the independence graph by Definition 68 with a 1×1 lock-chart as a template, whose only key has the cutting of the general key.
   a) *Exact*: If size permits, find a MIS using an exact, exponential procedure. If a result is found, it is guaranteed to be the optimal one regardless of the dataset.

   b) *Greedy*: In each iteration of a greedy approximation [25], pick the min-degree vertex and remove its neighbours from the graph.

[2] We do not list the exact constraints due to a non-disclosure agreement.

[3] The cardinality was measured by iterating over S, just as the formula (5.12). The result from Theorem 53 was not used due to the bias in the realistic dataset.

		baseline	sag	greedy	exact
uniform	baseline	1	0.59	0.64	0.70
	sag	**1.68**	1	**1.07**	1.00
	greedy	**1.57**	0.93	1	1.00
	exact	**1.43**	1.00	1.00	1
realistic	baseline	1	0.76	0.79	0.86
	sag	**1.31**	1	**1.03**	0.97
	greedy	**1.27**	0.97	1	0.99
	exact	**1.16**	**1.03**	**1.01**	1

Table 5.2: Cutting counting algorithms' performance in the explicit framework. Value larger than 1 means that the row-algorithm beats the column-algorithm.

6. For all the methods above, we recorded the number of individual keys, referred as the *score*.
7. When two algorithms are compared, we report the ratio of their scores. To aggregate more runs we used the geometric mean,[4] shown in Table 5.2.

CONCLUSIONS. Let us focus on the uniform dataset first. The 1.0 value in the SAG column of Table 5.2 above corroborate the optimality proof of the SAG heuristic from Theorem 52. If $|S|$ is small enough for the exact procedure to produce a result, the greedy algorithm achieves a near-optimal score (up to the round-off error). However as $|S|$ grows, the greedy approximation loses 7% on average to the optimum.

Results in the realistic dataset are similar, merely with smaller differences in scores. Figure 5.3 above shows that the baseline can win over SAG only on very small lock-charts, by a factor of 2 at most. With more keys, the randomness inside the baseline heuristic is more likely to do wrong decisions and above 100 keys SAG always performed better.

We consider the Figure 5.4 below as the most interesting. It shows that until 10 keys, the greedy procedure achieves a better score than SAG. This is true up to ~ 100 keys, where both heuristics are roughly even. With increasing code space, the greedy procedure loses, which also explains the overall 3% loss. When focused on the small instances where the exact procedure found a result, the greedy heuristic won over SAG by 2%.

[4] Example: In one run, the baseline score is 10 and the SAG score 20. In the next one, the respective scores are 3 and 4. The final score would be $\sqrt{\frac{20}{10} \cdot \frac{4}{3}} = 1.63$. We say that on average, the SAG heuristic achieves 63% better results than the baseline.

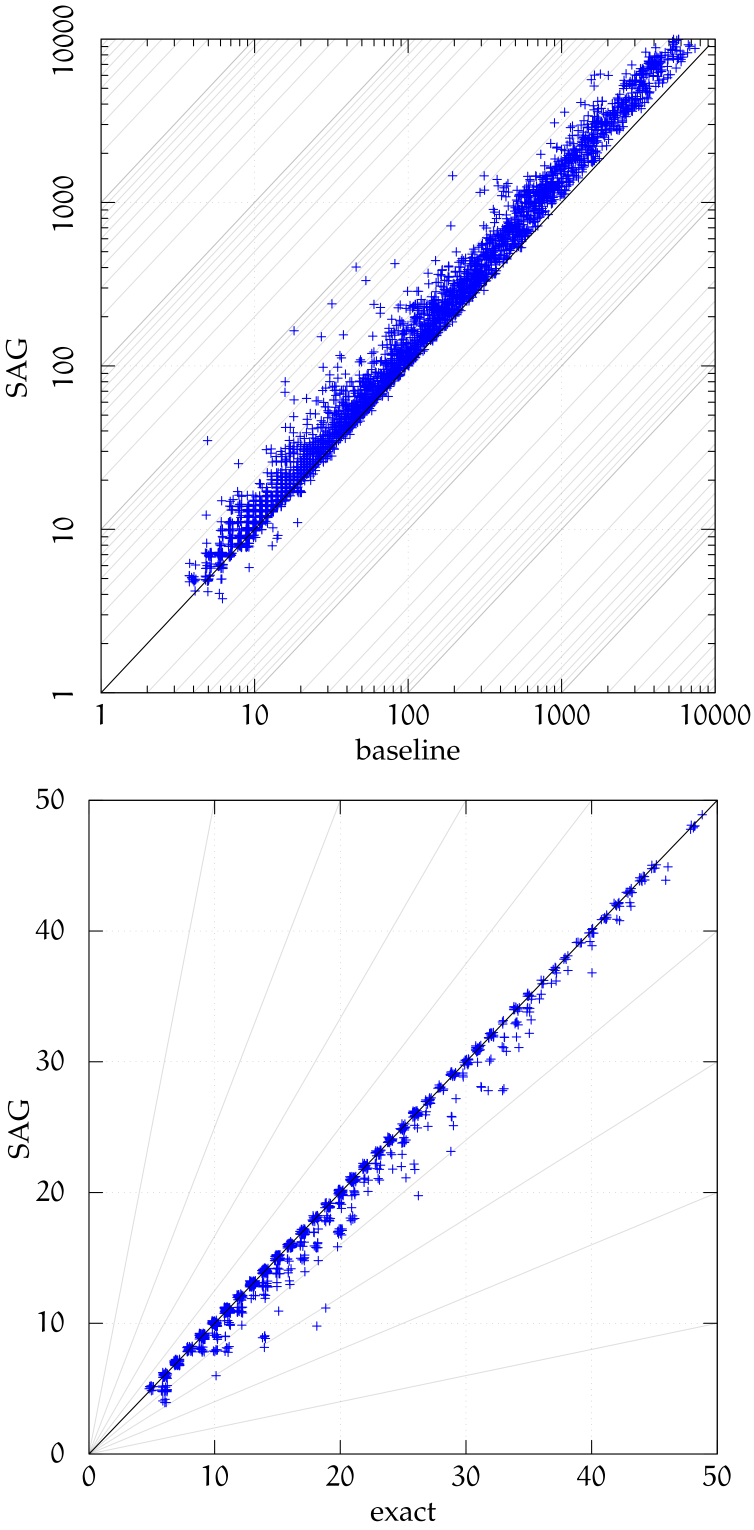


Figure 5.3: Scatterplot of the number of individual keys evaluates the same-as-general heuristic on the realistic dataset. Jitter is ±0.25.

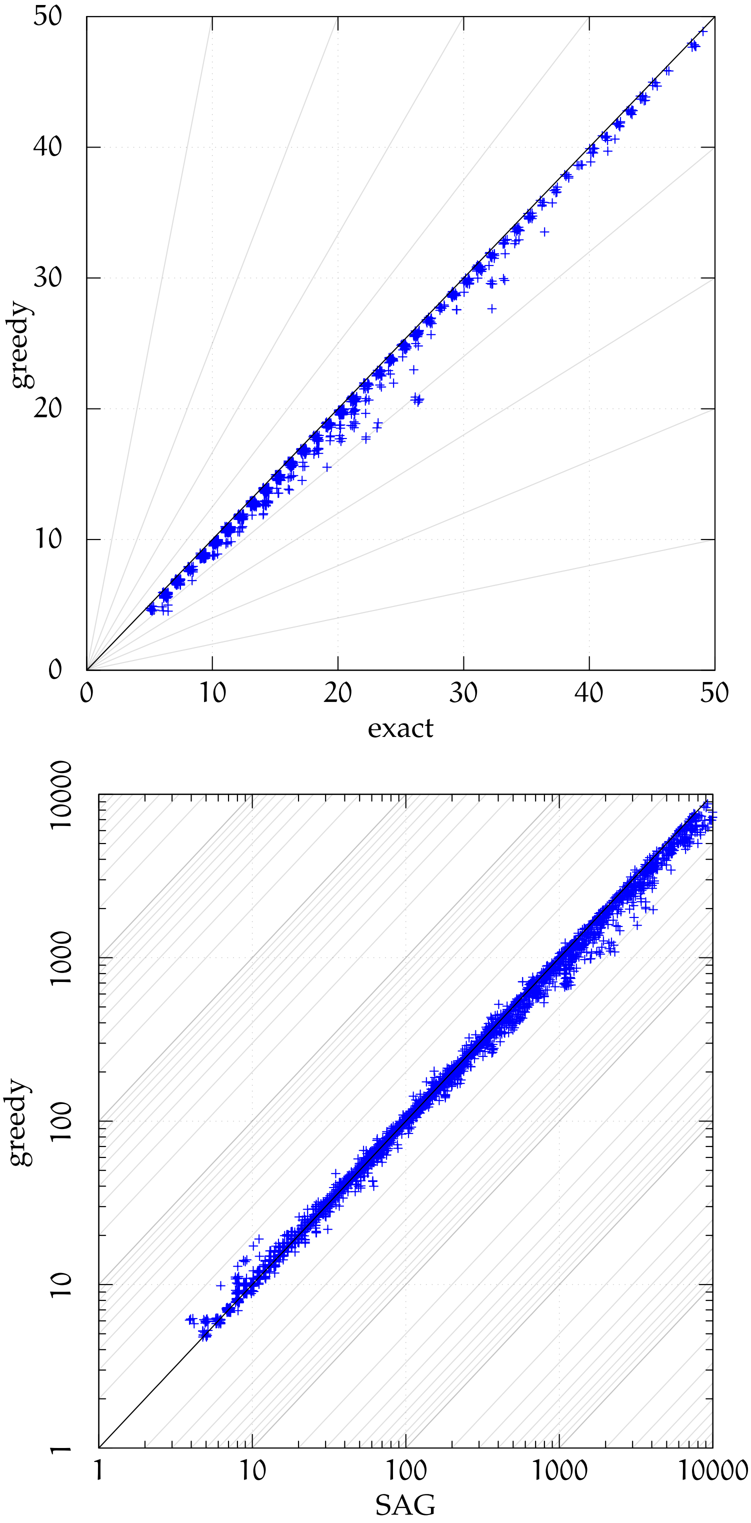


Figure 5.4: Scatterplot of the number of individual keys evaluates the greedy approximation on the realistic dataset. Jitter is ±0.25.

6 BACKTRACKERS

A solution is a function that assigns cuttings to keys. A function is a set of binary tuples. Hence by *backtrackers* we mean algorithms that systematically explore subsets of $K \times S$. Unlike most previous ideas, they do not rely on a translation to a different formalism (e.g. SAT).

Since the search space is vast, we only deal with depth-first-search algorithms. Unlike breadth-first-search, their memory requirements scale linearly with $|K|$.

input : Melted profiles lock-chart (K, L, E, B), a set of available key cuttings S and a partial solution $\hat{s}$
output: Algorithm `dfs(ŝ)` returns a solution $s : K \to S$ of the lock-chart or `null` if no solution exists

```
1  Function dfs($\hat{s} : K \rightharpoonup S$):
2      pick an unassigned key $k$
3      foreach candidate cutting $\gamma \in S$ do
4          $\hat{s}' \leftarrow \hat{s} \cup (k, \gamma)$
5          if $\hat{s}'$ is perspective then
6              $s \leftarrow$ dfs($\hat{s}'$)
7              if $s \neq$ null then
8                  return $s$
9              end
10         end
11     end
12     return null
```

Algorithm 6.1: Depth-first-search solver for melted profiles extension lock-charts.

Algorithm 6.1 shows the template for all algorithms in this chapter. As a domain-specific algorithm, it is easy to modify and tweak. Specifically, there are 4 questions that can be addressed:

1. Which keys to pick first on line 2?
2. Which cuttings to choose from on line 3?
3. Which cuttings to pick first on line 3?
4. Which assignment is perspective on line 5?

The plain version of the algorithm answers the questions as follows: 1. Pick keys sequentially from k_1 to $k_{|K|}$. 2. Try all cuttings. 3. Pick cuttings randomly. 4. Consider a partial assignment perspective if it is a solution.

Before finding better answers, please notice that the plain algorithm is not very bad. Section 5.3 used a non-backtracking version called the "baseline". On average it was able to solve a diagonal lock-chart with 59% keys of the largest solvable lock-chart. Speaking about the worst-case, it found at least 10% keys of the optimum (see Figure 5.3). With backtracking in place, the algorithm's performance might only increase.

6.1 AUTOMORPHISM ALGORITHM

Lawer's seminal work [38] on lock-chart solving provided a technique to prune large portions of the search-space. Perhaps surprisingly, the technique does not sacrifice completeness – if the lock-chart has a solution, the algorithm finds it.

In this section, its main idea is presented. We decided to reformulate the algorithm for two reasons. First, Lawer's work was hard to digest. A lengthier introduction might help some readers. Moreover, the actual implementation of her algorithm was never publicly released. For the experimental part of this text, we have reimplemented the idea exactly as described here. For a complete formal treatment, readers are encouraged to go through the original document.

PRELIMINARIES. Let (V, E) be a graph. An *automorphism* is a bijection $a : V \to V$ s.t. $a(v)$ is adjacent to $a(w)$ if and only if v is adjacent to w. A group of automorphisms on (V, E) *induces* an equivalence relation $\simeq$ on the set V: Two nodes are related $v \simeq w$ if there is an automorphism a on (V, E) s.t. $a(v) = w$. Every equivalence relation $\simeq$ defines a *partition* of V into sets $P_1, \dots, P_m$ called *classes*: Two vertices $v, w \in P_i$ if and only if $v \simeq w$. The classes are disjoint (for every two distinct classes P_i and P_j: $P_i \cap P_j = \emptyset$) and they partition V $(P_1 \cup \cdots \cup P_m = V)$. Hence a group of n automorphisms on (V, E) can be efficiently represented using $P_1, \dots, P_m$ classes of vertices s.t. $n = P_1! \cdot \cdots \cdot P_m!$.

For convenience and a concise notation, an empty partition $\emptyset$ does not modify the set of automorphisms. Hence $P_1, \dots, P_m$ and $P_1, \dots, P_m, \emptyset$ define exactly the same group of automorphisms. Given $0! = 1$, the sizes are consistent $P_1! \cdots P_m! = P! \cdots P_m! \cdot \emptyset!$.

As of writing this text, it is not known whether the problem of finding a non-trivial automorphism is $\mathcal{NP}$-complete or in $\mathcal{P}$ [41]. The algorithm with the best known asymptotic time complexity runs in quasipolynomial time [5, 6].

Let (V', E') be another graph. The *induced subgraph isomorphism* is a function $s : V \to V'$ s.t. $s(v)$ is adjacent to $s(w)$ in (V', E') if and only if v is adjacent to w in (V, E). Consequently, the subgraph of (V', E') induced by vertices mapped by s is isomorphic to (V, E). The problem of finding an induced subgraph isomorphism is $\mathcal{NP}$-complete [51]. If the two graphs are bipartite $V = V_1 \cup V_2$, $V' = V_1' \cup V_2'$, the *bipartite induced subgraph isomorphism* (BISI) is an induced subgraph isomorphism that maps only matching partite sets: $s \subseteq (V_1 \times V_1') \cup (V_2 \times V_2')$.

Let n, k be two integers. There *number of combinations with repetition* $\left(\!\binom{n}{k}\!\right)$ denotes the number of k-tuples $(x_1, x_2, \dots, x_k)$ s.t. $1 \leqslant x_i \leqslant n$ which are ordered $x_1 \leqslant x_2 \leqslant \cdots \leqslant x_n$.[1] The number can be evaluated using the binomial coefficient $\left(\!\binom{n}{k}\!\right) = \binom{n+k-1}{k}$.

SEARCH SPACE GRAPH. Lawer's algorithm will be presented in several steps. The first step reduces the lock-chart problem to finding a bipartite induced subgraph isomorphism.

Definition 72. The *search space graph* is a bipartite graph $(S \cup T, R)$, where S is the set of all valid cuttings, T the set of all cylinders and $(\gamma, \lambda) \in R$ if γ enters λ.

Remark 73. Lock-chart (K, L, E) has a correct solution s if and only s is a BISI from $(K \cup L, E)$ to the search space graph $(S \cup T, R)$.

Proof. Let s be a correct solution. Then k is adjacent to l iff $s(k)$ enters $s(l)$ by Definition 15 on page 21. This is equivalent to $s(k)$ being adjacent to $s(l)$ in the search space graph by Definition 72. Hence s is a BISI. The other direction is similar. □

SYMMETRIES. The second step provides a way to prune the search space by finding symmetries (automorphisms) in the search space graph. The motivation behind it is to somehow "skip" isomorphic solutions, but still ensure that the algorithm can reach a solution (if it exists).

Theorem 74. *Let (K, L, E) be a lock-chart, s its solution and a an automorphism on the search space graph. Assignment s' defined as $s'(x) = a(s(x))$ is a solution.*

Proof. If a is an automorphism in the search space graph, $s(k)$ and $s(l)$ are adjacent iff $a(s(k))$ and $a(s(l))$ are adjacent. Therefore s' is a BISI, and by Remark 73, s' is a solution. □

The theorem explains why having an automorphism prunes the search space. How to find automorphisms?

[1] If the ordering was strict $x_1 < x_2 < \cdots < x_n$, there would be $\binom{n}{k}$ such tuples. If there was no prescribed ordering, there would by n^k tuples.

AUTOMORPHISMS BY BRUTE-FORCE. Before proceeding, let us report some experience with a brute-force approach, which constructs a restricted version of the search space graph in memory. The restriction exploits Remark 18 and reduces the number of cylinders $|T|$ from $2^{d \cdot p}$ to $|S|^{\max_{l \in L} |E(l)|}$ without sacrificing completeness. For diagonal lock-charts, the size of $|T|$ is at most $|S|^2$ and the size of $|U|$ is at most $|S|^3$.

For experiments, we took the vanilla framework, generated all d^p cuttings, all $\left(\frac{1}{2} \cdot d \cdot (d+1)\right)^p$ cylinders and the respective connections R. Next we took a few small real-world platforms and their set S. For finding all automorphisms the `nauty` program [41] was used.

Quite surprisingly, `nauty` was able to find all $d^p!$ automorphisms at least for small vanilla code spaces (roughly $d, p \leqslant 5$) within the timeout. However, behind a certain border (around $d \sim p \sim 8$), the runtime suddenly spiralled, and no solution was found under 1 hour. We attributed this to a switch between modes inside `nauty`.

Out of the real-world platforms, none were processed in a reasonable time. Even with only $|S| \sim 700$ cuttings, the asymmetries in the code space prevented `nauty` from finding any isomorphism. The brute-force approach is probably not usable for a practical algorithm.

LAWER'S AUTOMORPHISMS. Instead, automorphisms in [38] are derived theoretically. They are composed of two separate isomorphisms, one on cutting depths and one on positions. For representing both isomorphisms, the algorithm keeps track of classes of depths and classes of positions. An automorphism on the search space graph is a composition of the cutting depth automorphism and position automorphism.

The group of automorphisms is "reduced" by a partial assignment $\hat{s}$ that enters the function `dfs`. Even though the automorphisms should be as "big" as possible to prune large numbers of candidate cuttings, they must also be defined in a way that the partial solution $\hat{s}$ remains unchanged. For any automorphism induced by the classes of depths and the classes of positions, we expect

$$\hat{s}(k) = a(\hat{s}(k)) \text{ for all already assigned keys in } \hat{s}. \tag{6.1}$$

The *depth-classes* partition the set $\{1, \dots, d\}$ and are represented by two disjoint sets $U, R \subseteq \{1, \dots, d\}$ s.t. $U \cup R = \{1, \dots, d\}$. The set U contains cutting depths that form *unitary* classes. In other words, if there is a unitary cutting depth $u_i \in U$, then there is one class $\{u_i\}$ in the partition. The set R of *remaining* cutting depths forms a separate class. Hence given sets $U = \{u_1, \dots, u_{|U|}\}$ and $R = \{1, \dots, d\} \setminus U$, the automorphism group on cutting depths is defined by the partition $\{u_1\}, \dots, \{u_{|U|}\}, R$ that generates $|R|!$ automorphisms.

The idea behind the isomorphism on depths is similar to the proof of Lemma 46. Despite cutting depths are numbers, there is no particular order among them. In any solution $\hat{s}$, by swapping any two cutting depths on a certain position in every cutting and every cylinder, one must arrive at a solution as well. Hence, the set U associated with the i-th position will contain cutting depths that appeared as $\hat{s}(k)_i$ for some already defined key k. Since R contains all unused cutting depths, a candidate cutting can try any arbitrarily chosen value from R, yet still retain completeness.

The *position-classes* partition the set of positions $\{1, \dots, p\}$ into m classes $P = \{P_1, \dots, P_m\}$. Every class P_x represents positions that can be "freely permuted". For any two positions $i, j \in P_x$, if cuttings d_i and d_j were swapped in every cuttings $\hat{s}(k)$ and sets of depths D_i and D_j were swapped in every cylidner $\hat{s}(l)$, the solution $\hat{s}$ would remain exactly the same.

Every position-class is *mapped* to one group of depth-classes by the function $u : P \to 2^{\{1,\dots,d\}}$. Let

$$U_x = u(P_x) \text{ and } R_x = \{1, \dots, d\} \setminus U_x \ .$$

Now we can describe how u prescribes all combinations of candidate depths $d_i, d_j, \dots$ to positions $P_x = \{i, j, \dots\}$. First note that by picking only one depth from the unused cutting depths R_x (if there is one) and all choices of U_x, every position can be assigned $|U_x| + 1$ candidate cutting depths. Second, depths on positions P_x can be freely permuted (see the previous paragraph), hence by considering only sequences of increasing cutting depths $d_i \leqslant d_j \leqslant \cdots$, completeness is not sacrificed. The number of candidate cutting depths assignable to $|P_x|$ positions is given by the z function, defined using the number of combinations with repetition

$$z(P_x, U_x, R_x) = \begin{cases} \left(\!\!\binom{|U_x| + 1}{|P_x|}\!\!\right) & \text{if } R_x \neq \emptyset \\[2ex] \left(\!\!\binom{|U_x|}{|P_x|}\!\!\right) & \text{if } R_x = \emptyset \end{cases} \tag{6.2}$$

By combining mupltiple position-classes, one arrives at the total number of candidate cuttings

$$\prod_{x=1}^{m} z(P_x, U_x, R_x) \; . \tag{6.3}$$

Example 75. Let $p = 4$, $d = 3$ and the partial solution $\hat{s}$ assigns two keys: $\hat{s}(k_1) = (1,1,1,1)$ and $\hat{s}(k_2) = (1,1,1,2)$. Which candidate cuttings are there for k_3? The first three positions have the same depth. Hence $P_1 = \{1,2,3\}$ and $P_2 = \{2\}$. On positions from P_1 only the cutting depth 1 has been used: $U_1 = \{1\}$ and $R_1 = \{2,3\}$. In P_2 two cutting depths were used: $U_2 = \{1,2\}$ and $R_2 = \{3\}$. We expect

$$\prod_{x=1}^{m} z(P_x, U_x, R_x) = \prod_{x=1}^{m} \binom{|P_x| + |U_x|}{|P_x|} = \binom{3+1}{3} \cdot \binom{1+2}{1} = 12$$

candidate cuttings to be generated (by using $\left(\!\binom{n}{k}\!\right) = \binom{n+k-1}{k}$ and $R_x \neq \emptyset$). The class P_1 generates 4 "partial" cuttings $(1,1,1,*)$, $(1,1,2,*)$, $(1,2,2,*)$ and $(2,2,2,*)$. The class P_2 generates $(*,*,*,1)$, $(*,*,*,2)$ and $(*,*,*,3)$. Their product is indeed 12 cuttings:

$$\begin{array}{cccc}
(1,1,1,1) & (1,1,2,1) & (1,2,2,1) & (2,2,2,1) \\
(1,1,1,2) & (1,1,2,2) & (1,2,2,2) & (2,2,2,2) \\
(1,1,1,3) & (1,1,2,3) & (1,2,2,3) & (2,2,2,3)
\end{array}$$

MODIFYING THE DFS ALGORITHM. A straightforward application of Lawer's algorithm is to modify line 3 of Algorithm 6.1 by adding two lines of code: 1) compute P and u from a partial solution $\hat{s}$ and 2) generate all candidate cuttings. However, the first step might be implemented more efficiently.

Given the freshly assigned cutting γ, the isomorphism group P, u can be "updated" more quickly just before passing the updated isomorphisms into the recursive call on line 6. Here we describe how to calculate the updated P', u', r' from P, u, r and γ.

Initialisation is simple. Since an empty partial solution $\hat{s} = \emptyset$ defines no cuttings, all positions are equal and no cutting depth has been used: $P = \{\{1, \ldots, p\}\}$ and $u(\{1, \ldots, p\}) = \emptyset$.

The update function will be defined concisely using the following notion. Let $\gamma = (d_1, \ldots, d_p)$. The *cutting-based partition* Q^{γ} is a partitioning of positions into y classes s.t. positions i and j are in the same class $i, j \in Q^{\gamma}_{y}$ if and only if $d_i = d_j$. For example the cutting $\gamma = (1,1,1,2)$ is associated with $Q^{(1,1,1,2)} = \{\{1,2,3\},\{4\}\}$.

Let there be m position-classes $P = P_1, \dots, P_m$ and n cutting-based classes $Q^\gamma = \{Q^\gamma_1, \dots, Q^\gamma_n\}$. Then the updated partition is

$$\begin{aligned} P' = \{ \quad & P_1 \cap Q^\gamma_1, & P_1 \cap Q^\gamma_2, & \quad \cdots, & P_1 \cap Q^\gamma_n, & \\ & P_2 \cap Q^\gamma_1, & P_2 \cap Q^\gamma_2, & \quad \cdots, & P_2 \cap Q^\gamma_n, & \\ & \vdots, & \vdots, & \quad \ddots, & \vdots, & \\ & P_m \cap Q^\gamma_1, & P_m \cap Q^\gamma_2, & \quad \cdots, & P_m \cap Q^\gamma_n & \ \} \,. \end{aligned} \tag{6.4}$$

Function u is updated as follows. Let Q^γ_y contain positions, where cutting γ has depth d_i. The updated mapping adds depth d_i to the set of used depths $u(P_x)$:

$$u'(P_x \cap Q^\gamma_y) = u(P_x) \cup \{d_i\} \,.$$

Example 76. Let $p = 4$, $d = 3$. Isomorphism on positions is initialised to $P = \{\{1, 2, 3, 4\}\}$. The only class of positions has depth isomorphism defined by $u(\{1, 2, 3, 4\}) = \emptyset$. The first key has only $\binom{4+0}{4} = 1$ candidate cutting, namely $(1, 1, 1, 1)$. Let's proceed to the second key. Since $Q^{(1,1,1,1)} = \{\{1, 2, 3, 4\}\}$ has only one class, P remains unchanged. By moving depth 1 from R to U, the only class of positions becomes mapped as $u(\{1, 2, 3, 4\}) = \{1\}$. Consequently, the next key has $\binom{4+1}{4}$ candidate cuttings. Indeed, there are 5 combinations with repetitions of cutting depths $\{1, 2\}$ to be assigned to all 4 positions: $(1, 1, 1, 1)$, $(1, 1, 1, 2)$, $(1, 1, 2, 2)$, $(1, 2, 2, 2)$ and $(2, 2, 2, 2)$.

Let's assume $(1, 1, 1, 2)$ was picked. Since $Q^{(1,1,1,2)} = \{\{1, 2, 3\}, \{4\}\}$, the partitioning P is updated to $\{\{1, 2, 3\}, \{4\}\}$. The first class $P_1 = \{1, 2, 3\}$ is still mapped to $u(P_1) = \{1\}$, because on these positions, the only used cutting depth is 1. The second class $P_2 = \{4\}$ becomes mapped to $u(P_2) = \{1, 2\}$. The automorphism is now exactly same as in Example 75 and the third key can has 12 candidate cuttings.

The example illustrates something we consider a "killer feature" of Lawer's automorphisms. In a diagonal lock-chart, the first key is the general key, which always gets only 1 candidate cutting. This is in line with Lemma 46, which can be summarised as: "In the vanilla framework, the cutting of the general key is irrelevant". For the first individual key, it attempts to assign exactly 1 candidate from every S_q set.

OTHER FRAMEWORKS. The automorphisms are defined for the vanilla framework. An extension to the asymmetric framework is straightforward. If the deepest cutting is $(\tilde{d}_1, \dots, \tilde{d}_p)$, the position-classes can be initialised to $P = Q^{(\tilde{d}_1, \dots, \tilde{d}_p)}$. Like this, the symmetries between positions with a different number of available cutting depths is broken but retained for positions with an equal number of available cutting depths. An extension to the explicit framework is hard to imagine (see the brute-force experiment above). What about the general framework?

Let us sketch an idea. Suppose there is a graph with vertices $(p, d) \in \{1, \dots, p\} \times \{1, \dots d\}$. It may be possible to translate gecons into edges of this graph so that (p, d) and (p, d') appear in the same partition class precisely if cutting depths d and d' are interchangeable on position p. How to formalise it precisely? Is finding automorphisms on a graph with $p \cdot d$ vertices doable in a reasonable amount of time? Moreover, will the automorphisms be large enough save enough backtracker's runtime? The answers need further research and empirical evaluation.

6.2 Constraint satisfaction

In this section, the lock-chart solving problem is formulated as a *constraint satisfaction problem* (CSP) and a pruning scheme is provided. The modified algorithm keeps a list of suitable cuttings for every key called a *scope*. By deleting "obviously unsuitable" cuttings from the scopes, one may arrive at a partial solution with less suitable cuttings that the number of unassigned keys. This idea is known as *all-different pruning* [47] and allows a backtracker to skip huge parts of the search space by modifying line 5 of Algorithm 6.1.

As the scope keeps a set of cuttings, we assume that one can iterate over S. Hence, the CSP is formulated for the explicit framework.

Preliminaries. Let D be a set called the *domain* and X a set of *variables*. A *constraint* is a pair (Σ, R), where $\Sigma \subseteq X$ is its *signature* (whose members will be denoted as $\sigma_1, \sigma_2, \dots$) and R a relation on D of arity $|\Sigma|$. A *constraint satisfaction problem* (CSP) is a tuple (D, X, C), where C is a set of constraints.

A *solution* to a CSP is a function $s : X \to D$, s.t. for all constraints $(\{\sigma_1, \dots, \sigma_m\}, R)$, the vector of solutions satisfies

$$(s(\sigma_1), \dots, s(\sigma_m)) \in R\,.$$

A solution is called *partial* if s is a partial function. The *domain function* $\delta : K \to 2^S$ assigns each variable a *scope.*

Lock-chart CSP. The lock-chart problem will be reformulated as a CSP. The CSP instance will have one variable for each key, code space as the domain and one constraint for every blocking cell in the lock-chart.

Definition 77. Let (K, L, E) be a lock-chart and S the code space. The (S, K, C) is a CSP if for every $k \in K$, $l \in L$ s.t. $(k, l) \notin E$, there is one constraint $(\{k\} \cup E(l), R_{k,l}) \in C$, where $R_{k,l} =$

$$\left\{(\gamma_0, \gamma_1, \dots, \gamma_{|E(l)|}) \in S^{|E(l)|} \middle| \gamma_1 \cup \dots \cup \gamma_{|E(l)|} \text{ blocks } \gamma_0 \right\} . \tag{6.5}$$

Remark 78. Let (K, L, E) be a lock-chart, S the code space and (S, K, C) the CSP from Definition 77. A solution s to (S, K, C) is a solution to the lock-chart (K, L, E).

Proof. By definition, CSP's solution s is an assignment. Let $k \in K$, $l \in L$. Proposition 16 ensures that if $(k, l) \in E$, then $s(k)$ enters $s(l)$. Consider the $(k, l) \notin E$ case. Since s is a solution to the CSP, there must be

$$(\gamma_0, \gamma_1, \dots, \gamma_{|E(l)|}) \in R_{k,l} .$$

By (6.5) γ_0 is blocked in $\gamma_1 \cup \dots \cup \gamma_{|E(l)|}$. Since $\gamma_0 = s(k)$ and $\gamma_1 \cup \dots \cup \gamma_{|E(l)|} = s(l)$, then $s(k)$ is blocked in $s(l)$. By checking all k, l, all cells in the lock-chart are satisfied. □

The largest relation has $|S|^{1+\max_l |E(l)|}$ tuples, which is too much for storing all relations in memory. Instead, there is Algorithm 6.2, which provides a pruning algorithm tailored for the lock-chart CSPs and which does not store relations explicitly.

The pruning algorithm provides a guarantee: If $\hat{s}$ was a solution and $\gamma \in \delta(k)$, then by assigning γ to a unassigned key k, the assignment $\hat{s} \cup (k, \gamma)$ is a solution.

Lemma 79 (Consistency). *Let (K, L, E) be a lock-chart, $\hat{s}$ a partial solution that assigns cuttings keys from a set $K' \subseteq K$ and*

$$\delta = \mathtt{prune}((K, L, E), \hat{s}) .$$

Assume that for any $\gamma, \gamma' \in S$, the cylinder $\gamma \cup \gamma'$ blocks some cuttings from S. Then for every constraint $(\{k_0, k_1, \dots, k_{|E(l)|}\}, R_{k_0,l})$, for every key k_i in its signature, for every cutting $\gamma_i \in \delta(k_i)$, there is a tuple $(\gamma_0, \gamma_1, \dots, \gamma_{|E(l)|}) \in R_{k,l}$ s.t.

$$\textit{if } k_j \in K' \textit{ then } \gamma_j = \hat{s}(k_j).$$

Proof. a) $i = 0$: Given $k_j \in K'$ then all locks $E(k_j)$ contain shear-lines induced by $\hat{s}(k_j)$. On line 5 the pruner removes such shear-lines from the domain of a blocked key k_0 and therefore $\gamma_0 \notin \hat{s}(k_0)$.

b) $i \geqslant 1$, $j = 0$: The state, when k_0 is blocked in l and $k_0 \in K'$ is checked on line 13. If assigning γ_i to k_i breaks this blocking, γ_i is removed from $\delta(k_i)$ on line 19.

```
Function $\delta = \texttt{prune}((K, L, E), \hat{s} : K \rightharpoonup S)$:
    input  : a lock-chart and its partial solution $\hat{s}$
    output : domain function $\delta : K \to 2^S$
    let $\delta \leftarrow$ array of size $|K|$, all cells filled with $S$, for
      convenience indexed by keys
    for $l \in L$ do
        $\Gamma \leftarrow$ cuttings that enter $\hat{s}(l)$
        /* $\Gamma$ will never be blocked in any extension of $\hat{s}$. */
        for $k \notin E(l)$ do
            remove $\Gamma$ from $\delta(k)$
        end
    end
    for $l \in L$ do
        $\Gamma \leftarrow \emptyset$
        for $\gamma \in S$ do
            /* Simulate adding $\gamma$ into $\hat{s}(l)$ and find keys
            already assigned in $\hat{s}$ that should be blocked by $l$.
            */
            $\lambda \leftarrow s(l) \cup \gamma$
            for $k \notin E(l)$ do
                if $k$ is assigned by $\hat{s}$ and $\hat{s}(k)$ enters $\lambda$ then
                    add $\gamma$ to $\Gamma$
                end
            end
        end
        /* $\Gamma$ contains cuttings that violate some already
        established blocking in $\hat{s}(l)$. Never use them for keys
        that open $l$.                                  */
        for $k \in E(l)$ do
            remove $\Gamma$ from $\delta(k)$
        end
    end
    return $\delta$
```

Algorithm 6.2: The domain pruning algorithm for a CSP in the explicit framework.

c) $i \geqslant 1$, $j \geqslant 1$. Values in $1, \ldots, |E(l)|$ columns come from the cartesian product $S^{|E(l)|}$. Since cylinder $\gamma \cup \gamma'$ blocks some other cutting, there must be a candidate for γ_0. Therefore the tuple with γ_j is in $R_{k,l}$. □

Note this is a weaker consistency than *generalised arc-consistency* [39]. GAC would replace the "if $k_j \in K'$ then $\gamma_j = s(k_j)$" by a stronger condition "if $k_j \in K$ then$\gamma_j \in \delta(k_j)$".

all-different pruning. Next, we add a pruning scheme for an early detection of non-perspective partial solutions. Proposition 26 ensures that all keys must be assigned different key cuttings, which allows to impose the *all-different constraint* [47] on K. Instead of the pruning algorithm based on bipartite matching [47], here we use a simpler technique which achieves a weaker consistency.

Let $\delta = \texttt{prune}((K, L, E), \hat{s})$, K' be any subset of K and

$$\Delta = \bigcup_{k \in K'} \delta(k) \tag{6.6}$$

be the union cuttings in their scopes. Next, every complete solution $s \supseteq \hat{s}$ (which extends $\hat{s}$) must assign all keys K'. Such keys will never be assigned cuttings outside Δ. Therefore, by the pidgeon-hole principle, there must be at least as many cuttings as there are keys

$$|\Delta| \geqslant |K'| \,. \tag{6.7}$$

If this formula does not hold, the partial solution $\hat{s}$ can never be extended to a complete correct solution.

Algorithm 6.3 is a fast test that attempts to find K' which violates the formula. It adds scopes of keys to a set Δ, one-by-one. If Δ is too small and too many scopes have been added, the algorithm returns `false`.

Finally, let constrcut a full CSP backtracking algorithm from ideas in this section. The skeleton copies Algorithm 6.1 with the following modifications:

- On line 3, the scopes are calculated by calling `prune`. Candidate cuttings for key k are taken from $\delta(k)$.
- On line 5, a partial solution is not considered perspective if `perspective`$((K, L, E), \hat{s}')$ returns `false`.

6.3 implicit domains

The last algorithm builds upon an empirical observation. When executed on real-world datasets, the `perspective`(δ) procedure

```
Function perspective(δ):
    input  : a scope function $\delta : K \to 2^S$
    output: true if it finds a $K' \subseteq K$ s.t. $|\Delta| < |K'|$
    let $\Delta$ be a set of cuttings
    let $K$ be an array of keys
    sort $K$ by $|\delta(k)|$ in ascending order
    initialise $\Delta = \emptyset$
    for $i \in \{1, \dots, |K|\}$ do
        $\Delta \leftarrow \Delta \cup \delta(k_i)$
        if $|\Delta| < i$ then
            return false
        end
    end
    return true
```

Algorithm 6.3: Simplified all-different pruner for the CSP backtracker.

usually returned `false` only if the set K' consisted of individual keys. This section formulates a different pruning algorithm tailored specifically for individual keys. Its advantage over the previous CSP approach is that it works in the general framework and hence it can handle larger code spaces.

TEMPLATE LOCK-CHARTS. The backtracker keeps track of the available code space for individual keys. For a succinct notation, let's assume that the backtracker always solves a template lock-chart (M, G, F) with an expansion function $e : G \to \mathbb{N}$ (see Definition 66). Keys in M represent master keys $k_1, \dots, k_m$ and locks $g_1, \dots, g_n$ will be called *groups*. The lock-chart (K, L, E), which actually enters Algorithm 6.1, will be the the expansion of this template. Keys K in the expansion will be denoted

$$K = \{k_1, \dots, k_m, \overbrace{k_1^1, \dots, k_1^{e(g_1)}}^{\text{expansion of } g_1}, \dots, \overbrace{k_n^1, \dots, k_n^{g_n}}^{\text{expansion of } g_n}\}. \tag{6.8}$$

Since master keys $k_1, \dots, k_m$ appear both in the template and the expansion ($M \subseteq K$), any partial solution $\hat{s}$ associated with the expansion lock-chart also prescribes cuttings for the template lock-chart.

Assume further that on line 2 of Algorithm 6.1 keys are picked in the same order as they appear in (6.8). Consequently, master keys will be assigned before individual keys from the first group g_1. Only after that, the keys in other groups will be assigned.

PRUNING SCHEME. The proposed pruning scheme checks if there are enough cuttings for individual keys in each group.

We are aiming at reusing the $\bar{\Gamma}(q, E, \{B_1, \ldots, B_n\})$ procedure for counting cuttings and comparing its size with $e(g_i)$. The procedure will be invoked when assigning the first individual key k_i^1 from group g_i. Before stating a similar inequality to (6.7), which justifies the pruning, let's define arguments q, E and Bs.

Natural candidates for cylinders B are universal cylinders (see Lemma 37) created from gecons in the general framework (see Definition 5). All cuttings must these gecons, obviously including cuttings assigned to individual keys. Let's call such cylinders the B cylinders of type 0.

Next, all blocking constraints in the lock-chart are translated to B cylinders of types 1 and 2. We start with cylinders of type 1, which ensure that a newly assigned cutting is blocked in all relevant cylinders.

Lemma 80. *Let* (K, L, E) *be the expansion s.t.* K *is indexed as in* (6.8) *and* $\hat{s}$ *its partial solution. If an individual key* k_i^x *is assigned a cutting that enters some cylinder* $\hat{s}(l_j^y)$ *for* $i \neq j$ *or* $x \neq y$*, partial assignment* $\hat{s}'$ *on line 4 of Algorithm 6.1 is not a solution.*

Proof. If $i \neq j$ or $x \neq y$, lock l_j^y is not opened by k_i^x by Definition 66 of the template lock-chart. Therefore a cutting assigned to k_i^x must be blocked in l_j^y. □

Since $\bar{\Gamma}$ counts cuttings that are blocked by cylinders B, cylinders $\hat{s}(l_j^y)$ can be passed as the last argument of $\bar{\Gamma}$. They constitute B cylinders of type 1. Also note that many such cylinders are equal:

Remark 81. When the individual key k_i^x is being assigned a cutting, all cylinders $\hat{s}(l_j^y)$ are equal to $\hat{s}(g_j)$ from the template if a) $j > i$ or b) $j = i$ and $y > x$.

Proof. The set of keys $E(l_j^y)$ contains some master keys and exactly 1 individual key k_j^y. Key k_i^x appears before k_j^y in (6.8) precisely if conditions a) or b) hold. Consequently k_j^y has not been assigned before k_i^x and $\hat{s}(l_j^y)$ contains cuttings of the master keys only. This is the exactly the cylinder $\hat{s}(g_j)$ from the template lock-chart. □

After removing identical cylinders, Lemma 80 generates

$$\sum_{j=1}^{i-1} e(g_j) + (n - i)$$

cylinders B of type 1. The sum counts all $\hat{s}(l_j^y)$ cylinders for $j < i$ and the $(n - i)$ term counts all the $\hat{s}(g_j)$ cylinders.

The cylinders B of type 2 ensure that a previously established blocking is not violated. The prune method in Algorithm 6.2 preserved already established blockings by iterating over all $\gamma \in S$. In the general framework, this is not feasible. Instead, an already established blocking of key k in lock l_i^x translates to a gecon.

Lemma 82. *Let (K, L, E) be the expansion s.t. K is indexed as in (6.8), $\hat{s}$ its partial solution, l_i^x be a lock and k one of its stopped keys $k \notin E(l_i^x)$. Individual key k_i^x can be assigned cutting γ without violating the blocking of $\hat{s}(k)$ in $\hat{s}(l_i^x)$ if γ satisfies the gecon $(d_1, \dots, d_p)$ defined as follows: The depth d_j is a wildcard if $\hat{s}(k)_j \in \hat{s}(l_i^x)_j$; otherwise $d_j = \hat{s}(k)_j$.*

Proof. A cutting γ violates a gecon $(d_1, \dots, d_p)$ precisely if $\gamma_j = d_j$ on all non-wildcard positions (by the Definition 5). If it does, cylinder $\hat{s}(l_i^x)$ will contain shear-line $\hat{s}(k)$ after having assigned γ to k_i^x and therefore k is not blocked in l_i^x. If it does not, there is some depth γ_j different from a non-wildcard d_j and therefore k is blocked in l_i^x on the j-th position. □

By satisfying gecons generated for every blocked cell in the l_i^1 row, the previously established blockings will not be violated. A gecon is satisfied by a cutting γ if it is blocked in the gecon's universal cylinder. Therefore the $\bar{\Gamma}$ function receives one B cylinder of type 2 for every blocked cell.

Together, by generating cylinders B of types 0, 1 and 2, cuttings counted by $\bar{\Gamma}(q, E, \{B_1, \dots, B_n\})$ will be blocked in existing cylinders and their cylinders will block all blocked keys. How to ensure that cuttings within one group will be blocked amongst each other?

Let's motivate this question by a special case. If $(K \cup L, E)$ was a diagonal lock-chart and cylinder E contained a single shear-line, namely the general key's cutting, then (5.11) would give a lower bound on the size (K, L, E), which is equal to the size of its only group $e(g_1)$. How to define E for non-diagonal lock-charts?

Lemma 83. *Let $\bar{\Gamma}(q, \hat{s}(g_i), \{\emptyset\})$ count cuttings $\gamma_1, \gamma_2, \dots$. If these cuttings are assigned to keys $k_i^1, k_i^2, \dots$ then key k_i^x will be blocked in l_i^y for any $x \neq y$.*

Proof. The proof generalises the idea from the proof of Theorem 45. Let A_x be the set of positions, where the cutting $\hat{s}(k_i^x)$ has cutting depths present in lock $\hat{s}(g_i)$:

$$A_x = \left\{ r \in \{0, \dots, p\} \mid \hat{s}(k_i^x)_r \in \hat{s}(l_i^y)_r \right\} .$$

a) $A_x = A_y$: There is a position $r \notin A_x$ s.t. $\hat{s}(k_i^x)_r \neq \hat{s}(k_i^y)_r$. Key k_i^x is blocked in l_i^y on position r as well as key k_i^y in l_i^x.

b) $A_x \neq A_y$: Since $q = |A_x| = |A_y|$, there is a position $r \in A_x \setminus A_y$ and $r' \in A_y \setminus A_x$. Key k_i^y is blocked in l_i^x on position r and key k_i^x is blocked in l_i^y on r'. □

Now it is tempting to state the condition for a perspective partial solution $\hat{s}$ as

$$\max_{0 \leqslant q \leqslant p} \bar{\Gamma}(q, \hat{s}(g_i), \text{B cylinders of all 3 types}) \geqslant e(g_i)\ . \qquad (6.9)$$

However, this is a *sufficient* condition, not a *necessary* one, because $\bar{\Gamma}$ provides merely a lower bound on the size of diagonal "areas" in lock-charts. Section 5.3 found diagonal lock-charts in the realistic dataset with more individual keys than this formula would give. Nevertheless, the margin between the lower bound and the exact value no bigger than 50% (see Figure 5.3 below) and with larger code-space sizes, the margin only closed. Relying merely on the following assumption, the possibly overly-zealous pruning scheme must be evaluated empirically.

Assumption 84. *If the inequality (6.9) does not hold, there are not enough cuttings for individual keys in group g_i.*

The resulting pruning procedure is presented in Algorithm 6.4. Line 1 starts with two preconditions. The procedure is executed only when starting to assign the first key in each group of individual keys. Also, the pruning is considered only for sufficiently large groups, governed by a fixed threshold. Then, the procedure constructs B cylinders of type 0 (starting on line 3), type 1 (line 6 onwards) and type 2 (line 15 onwards). Finally, the inequality (6.9) is checked in the loop on line 25.

6.4 GVC MINIMISATION

In the last section, all algorithms above are evaluated empirically. The methodology was taken from the automorphism algorithm's analysis in [38]. All algorithms were adapted to minimise the number of shear-lines in a global virtual cylinder (see Definition 30). The adaptations were minimal so that the decision variant of the problem (which asks if *any* solution can be found) could also be evaluated.

DATASETS. For the evaluation, three datasets were used. Two of them are real-world datasets; the last was synthetic, generated using a procedure described below.

1. The *real-world dataset* contains the lock-charts published in [38], which were provided by a German manufacturer IKON. The study does not mention the code space that

```
Function perspective($(M, G, F)$, $e$, $(K, L, E)$, $\hat{s}$, $k_i^x$):
    input  : The template lock-chart $(M, G, F)$, the expansion
             function $e$, the expansion $(K, L, E)$, the partial
             solution $\hat{s}$ and a key yet to be assigned $k_i^x$
    output: true if the partial solution is perspective
    if $x = 1$ and $e(g_i) \geqslant$ threshold then
        let $\bar{B}$ be a set of cylinders, initialised to $\emptyset$
        foreach gecon in the platform specification do
            convert it to a universal cylinder and add to $\bar{B}$
        end
        for $j \in \{1, \ldots, n\}$ do
            if $j < i$ then
                for $y \in \{1, \ldots, e(g_j)\}$ do
                    add $\hat{s}(l_j^y)$ to $\bar{B}$
                end
            else
                add $\hat{s}(g_j)$ to $\bar{B}$
            end
        end
        foreach master key $k \notin F(g_i)$ do
            create gecon by Lemma 82 from key $k$ and lock $g_i$
            convert it to a universal cylinder and add to $\bar{B}$
        end
        for $j \in \{1, \ldots, i - 1\}$ do
            for $y \in \{1, \ldots, e(g_j)\}$ do
                create gecon by Lemma 82 from $k_j^y$ and $g_i$
                convert it to a universal cylinder and add to $\bar{B}$
            end
        end
        for $q \in \{0, \ldots, p\}$ do
            if $\bar{\Gamma}(q, \hat{s}(g_i), \bar{B}) \geqslant e(g_i)$ then
                return true
            end
        end
        return false
    end
    return true
```

Algorithm 6.4: All-different pruner for the general framework.

was used to solve them, and hence we used the vanilla framework with p and d values chosen as small as possible, which allowed at least one competing algorithm to find a solution. The actual values are given in Table 6.1.

2. The *master-only dataset* contains lock-charts from the real-world dataset, whose individual keys were deleted. The deletion allows us to compare results presented here with Lawer's decision to use the GVC minimisation on master keys only and to assign individual keys using a separate procedure (e.g. those suggested in Section 5.3). Since lock-chart `M204` contains only individual keys, there was one less lock-chart in the master-only dataset than in the real-world dataset.

3. The *synthetic dataset* was constructed algorithmically. First, the parameter p (number of positions), d (number of cutting depths) and m (number of master keys) were chosen randomly from values $\{2, 6, 10\}$. A lock-chart of independent keys (Definition 31) with m keys was used as a template, but keeping only the first 2^{m-1} locks. This ensures that k_m was the general key. Since $m \geqslant 2$, no diagonal lock-chart was not included. Next, the total number of individual keys x was sampled as either 1%, 2%, 5%, 20%, 50% or 100% of $|S_{\hat{q}}|$. To generate "reasonably" sized lock-charts, yet larger than those in the real-world dataset, values outside of $20 \leqslant x \leqslant 200$ were omitted. The expansion function was designed to spread x individual keys between as many combinations of master keys as possible:

$$e(l_i) = (i + x - 1) \div 2^{m-1} \tag{6.10}$$

Table 6.2 contains the list of all lock-charts in the synthetic dataset.

Algorithms. The first contesting algorithm was the *automorphism algorithm* from Section 6.1, which reduces the number of candidate cuttings from $|S|$ to the number given by (6.3).

The other two backtrackers also used the automorphism pruning scheme but added the all-different pruning on top of that. The *CSP algorithm* followed ideas from Section 6.2 and considered a partial solution perspective according to Algorithm 6.3. The *implicit algorithm* from Section 6.3 considered a partial solution perspective according to Algorithm 6.4.

All three backtrackers were modified as follows:

- If a solution is found, the search does not stop. It stops when no more solutions are available.

Lock-chart		Keys m+x	Locks	Positions p	Depths d
easy	M103	5+6	6	2	6
	M108	5+14	19	4	3
	M109	7+8	8	3	4
	M111	4+10	10	5	2
	M112	3+13	13	5	2
	M201	10+9	9	4	3
	M203	3+19	19	4	3
	M204	0+2	2	2	2
	M209	6+16	16	2	6
medium	M101	8+29	29	5	3
	M104	8+23	23	3	5
	M106	13+19	29	3	5
	M107	4+31	31	5	3
	M202	9+19	19	2	9
	M206	9+27	27	9	2
	M208	7+34	34	9	2
hard	M100	20+21	28	3	6
	M102	12+60	60	3	6
	M200	17+15	29	5	3
	M205	20+15	33	3	5

Table 6.1: Real-world lock-charts from [38] with values of p and d used during the experiments. Number of keys is formatted as $m + x$ = master + individual keys.

Lock-chart	Keys ($m+x$)	Locks	Pos. p	Depths d
D02M0025I02P06D100R	2 + 25	25	2	6
D02M0040I02P10D050R	2 + 40	40	2	10
D02M0081I02P10D100R	2 + 81	81	2	10
D02M0020I06P02D100R	2 + 20	20	6	2
D02M0187I06P06D001R	2 + 187	187	6	6
D02M0025I10P02D010R	2 + 25	25	10	2
D02M0050I10P02D020R	2 + 50	50	10	2
D02M0126I10P02D050R	2 + 126	126	10	2
D06M0025I02P06D100R	6 + 25	25	2	6
D06M0040I02P10D050R	6 + 40	40	2	10
D06M0081I02P10D100R	6 + 81	81	2	10
D06M0020I06P02D100R	6 + 20	20	6	2
D06M0187I06P06D001R	6 + 187	187	6	6
D06M0025I10P02D010R	6 + 25	25	10	2
D06M0050I10P02D020R	6 + 50	50	10	2
D06M0126I10P02D050R	6 + 126	126	10	2
D10M0025I02P06D100R	10 + 25	25	2	6
D10M0040I02P10D050R	10 + 40	40	2	10
D10M0081I02P10D100R	10 + 81	81	2	10
D10M0020I06P02D100R	10 + 20	20	6	2
D10M0187I06P06D001R	10 + 187	187	6	6
D10M0025I10P02D010R	10 + 25	25	10	2
D10M0050I10P02D020R	10 + 50	50	10	2
D10M0126I10P02D050R	10 + 126	126	10	2

Table 6.2: Lock-charts from the synthetic dataset with values of p and d used during the experiments. Number of keys is formatted as $m + x =$ master + individual keys.

- The algorithm keeps track of the best solution b so far. When a new solution s is found, b is overwritten to s if the number of shear-lines of the GVC decreases:

$$\left|\bigcup_{k\in K} s(k)\right| < \left|\bigcup_{k\in K} b(k)\right| \tag{6.11}$$

- To gain some more efficiency, any perspective partial solution $\hat{s}$ must also satisfy

$$\left|\bigcup_{(k,\gamma)\in\hat{s}} \gamma\right| < \left|\bigcup_{k\in K} b(k)\right| . \tag{6.12}$$

  This follows from the fact that the numuber of shear-lines in a partial solution can only increase in its extension.

- Following the idea in [38] the assignment of individual keys preferred cuttings, whose q value was $\arg\max_q |S_q|$.

The fourth algorithm was not a backtracker. The *SAT algorithm* translates the lock-chart into a CNF as described in Section 4.2 and then it is solved by the MiniSAT library [19]. Algorithm 6.5 describes the adaptation needed to minimise the GVC using a SAT solver.

The algorithm can be viewed as a method to find an asymmetric framework with the deepest cutting $(\tilde{d}_1, \dots, \tilde{d}_p)$. Iteratively it tries to minimise $\tilde{d}_1$ by forbidding the respective variables on line 11. If that is not possible (line 18), it continues with the next position $\tilde{d}_2$ etc. A early-escape pruning (line 16) occurs if it fails to minimise $\tilde{d}_i$ at all. In such a case, $\tilde{d}_{i+1}$ can't be minimised neither and the algorithm stops. The algorithm minimises GVC shear-lines under the following assumption.

Assumption 85. *The Λ under the asymmetric framework with deepest cutting $(\tilde{d}_1, \dots, \tilde{d}_p)$ has less shear-lines than Λ under $(\tilde{d}'_1, \dots, \tilde{d}'_p)$ if there is a position i s.t.*

$$\tilde{d}'_1 = \tilde{d}'_1, \dots, \tilde{d}_{i-1} = \tilde{d}'_{i-1} \text{ and } \tilde{d}_i < \tilde{d}'_i . \tag{6.13}$$

Since there are trivial counter-examples to this assumption, the effects on the calculation results must be found experimentally.

Two known algorithms published by other authors were left out of this comparison due to their incomplete specification. Junker's translation of lock-chart solving to CSP [34] relied on *set variables*, a feature present in ILOG solver v4.0 [31], which is already 20 years old at the time of writing this text and no longer available. In the documentation of the current version of ILOG solver [30] (now acquired by IBM) we were unable to find any reference to set variables. Hence, to the best of our knowledge, the

```
Function sat(p, d, (K, L, E, B)):
    input  : Number of positions p, number of cutting depths
             d and a melted profiles lock-chart (K, L, E, B)
    output : Solution which minimises the number of
             shear-lines in the GVC or null if no solution
             exists.
    let b a solution, initialised to null
    let C be the straightforward translation of $(K \cup L, E)$
    if C has a model then
        update b to the solution from C's model
    else
        return null
    end
    for $i \in \{1, \dots p\}$ do
        for $j \in \{2, \dots d\}$ do
            foreach $k \in K$ do
                add a unitary clause $\overline{\mathsf{key}}^{k}_{i,j}$ to C
            end
            if C's model can be found within a conflict limit
              then
                update b to the solution from C's model
            else if $j = d$ then
                return b
            else
                break to try the next position
            end
        end
    end
    return b
```

Algorithm 6.5: SAT solver modified to minimise $|\Lambda|$ using the straightforward translation.

actual strategy for navigating through the search-space defined by Junker's translation procedure is buried somewhere in the IBM archives.

The recent study [53] by Vőmel et al. adapted *simulated annealing* [2] algorithm for lock-chart solving. Their objective function measured "the grade of deviation from a correct matching" and the move from one partial solution to another did not use the plain idea of "random exchanges", but was "restricted at runtime" to find "'reasonable' candidates from a promising neighbourhood". However, no formal description of *reasonable* or *promising* was given. Despite incomplete specification, we tried implementing a prototype of a local search procedure. Since its performance was orders of magnitude slower than results presented in [53], we thought it would be unfair to consider it a "reimplementation of Vőmel's algorithm". Merely as a "sluggish prototype of local search", we decided not to include it here.

HYPOTHESES. The automorphism and CSP algorithms are complete in the following sense: Given an infinite amount of time, they always find the optimal solution. The implicit algorithm is not complete, because of possible violations of Assumption 84 and neither is the SAT algorithm because of Assumption 85. However, given a limited amount time, even the complete algorithms may fail to find the optimal solution quickly enough. Will the incomplete algorithms outperform them?

The implicit algorithm has an unpleasant property of invoking an $\mathcal{NP}$-complete pruning procedure in some points of the search space. Will the overall runtime benefit from this?

Finally, how do these algorithms perform at the decision variant of the problem? All four algorithms received a "hook", which reports every update of the best solution b. The first invocation of this hook reports the runtime necessary for finding the first solution, regardless of its quality.

DISCLAIMER. A particular bias of these experiments should be pointed out. One fundamental limitation of the SAT algorithm was reported in Section 4.2. Due to high memory consumption and the MiniSAT's search strategy oblivious to symmetries in the search-space (such as picking cuttings from $S_{\hat{q}}$ for individual keys), SAT is unlikely to solve lock-charts with more than ~ 1700 keys. The largest lock-chart in the dataset has $x = 200$ individual keys, well below this limit. If this limit were raised, SAT's performance would deteriorate. Lock-charts in this section should be considered *small-sized* or *mid-sized*.

Individual keys in the synthetic dataset follow the Definition 11 exactly. Each key either opens exactly one lock (hence individ-

Dataset	Autom.	CSP	Implicit	SAT
real-world (20 lock-charts)	8	8	8	20
	40%	40%	40%	100%
master-only (19 lock-charts)	17	17	17	19
	89%	89%	89%	100%
synthetic (24 lock-charts)	0	8	8	24
	0%	33%	33%	100%

Table 6.3: Number of lock-charts solved within the 10^5 s timeout.

ual) or opens most locks (hence master). Our business experience confirms Lawer's observation [38] that the hardest industrial lock-charts are *unstructured*. They have a lot of "almost individual" or "border-line master" keys, which, e.g. complicate picking cuttings of the $S_{\hat{q}}$ for individual keys. Hence lock-charts here are computationally hard due to the number of master keys rather than the poor structure of individual keys.

We avoided overly well-structured problems, such as the diagonal ones. The implicit algorithm detects whether they have a solution in linear time.[2] If a solution exists, the same is true for Lawer's algorithm. Hence, including well-structured problems would merely test data structures and quality of implementation, which is not the aim of these experiments.

RESULTS. All experiments were performed on a Intel Xeon clocked at 3.10 GHz with 128 GiB RAM. All algorithms were implemented as single-threaded. The timeout was set to 10^4 s, which is ~ 2.8 hours.

Table 6.3 summarises the number of successfuly solved lock-charts within the timeout, referred as the *success rate*. It shows the algorithms' ability to solve the decision variant of the lock-chart solving problem. We can see that SAT unquestionably dominates in all datasets having the 100% success rate.

For small lock-charts in the real-world and master-only datasets, none of the two additional all-different pruning increases the success rate. However, as the lock-chart's size increases in the synthetic dataset, both pruning strategies avoid parts of the search space, where the automorphism algorithm got trapped. Tables 6.4 and 6.5 support this by showing absolute runtime needed for finding the first solution – the time referred as *decision runtime*.

[2] Right after setting the general key's cutting (for which there is only 1 candidate cutting), the $\bar{\Gamma}(q, \hat{s}(k_1), \{\emptyset\})$ function is invoked exactly $p+1$ times in the only group g_1. It reduces to calling $\theta(p, q, \hat{s}(g), \emptyset)$, for which Algorithm 5.1 provides a fast implementation.

In the pair-wise comparison, two algorithms are compared. The score (runtime, shear-lines in GVC, ...) is averaged by the geometric mean in the same way as in Section 5.3 from those lock-charts, which were solved by both compared algorithms.

Table 6.6 shows the pair-wise comparison of decision runtimes. Values confirm the prevailing dominance of the SAT algorithm. Between CSP and the implicit pruning schemes, the implicit algorithm achieves a better runtime – a difference especially pronounced in the synthetic dataset, whose lock-charts are bigger.

How do the algorithms perform at minimising the number of shear-lines in the GVC? Table 6.7 show the results in real-world and master-only datasets. In both, only minor differences can be found. In one case (lock-chart `M108`), the automorphism algorithm's brute-force paid off by finding a better solution than all other algorithms. In two cases (lock-charts `M107` and `N200`), the SAT's brute force won over all other algorithms.

Probably the most interesting result is lock-chart `N201`. It is the only lock-chart, where Assumption 85 of SAT's optimality was necessarily violated. Nevertheless, possible further violations in the synthetic dataset, shown in Table 6.8, have a lesser effect than the backtrackers' inability to escape local minima. In 3 lock-charts, the difference was by more than one magnitude.

Finally, we measured the *optimisation runtime* – time needed to find the best solution. Such value is useful in practice for determining a reasonable timeout. If a timeout is greater than the optimisation runtime, then the quality of the solution (Table 6.7) will be guaranteed. Table 6.9 summarises the optimisation runtime in real-world and master-only datasets and support the practicality of the SAT algorithm.

For illustrative purposes, Figure 6.1 shows how the shear-lines in GVC are reduced over time in the `N102` lock-chart.

	Dataset	Autom.	CSP	Implicit	SAT
real-world	M103	14.43 s	18.29 s	22.71 s	$2.6 \cdot 10^{-3}$ s
	M108	$2.8 \cdot 10^{3}$ s	$5.0 \cdot 10^{3}$ s	$5.0 \cdot 10^{3}$ s	0.01 s
	M109	1.16 s	2.91 s	1.97 s	$2.1 \cdot 10^{-3}$ s
	M111	0.01 s	$2.1 \cdot 10^{-3}$ s	$2.5 \cdot 10^{-3}$ s	$3.7 \cdot 10^{-3}$ s
	M112	3.85 s	4.48 s	6.14 s	0.03 s
	M201	$9.2 \cdot 10^{-3}$ s	0.01 s	0.01 s	0.03 s
	M203	–	–	–	0.21 s
	M204	$7.0 \cdot 10^{-6}$ s	–	–	$3.1 \cdot 10^{-5}$ s
	M209	248.94 s	739.17 s	373.76 s	0.08 s
	M101	–	–	–	0.94 s
	M104	–	–	–	0.01 s
	M106	–	–	–	0.01 s
	M107	–	$2.6 \cdot 10^{-3}$ s	$1.2 \cdot 10^{-3}$ s	0.02 s
	M202	–	–	–	0.01 s
	M206	–	–	–	0.11 s
	M208	–	–	–	0.09 s
	M100	–	–	–	0.07 s
	M102	–	–	–	0.08 s
	M200	–	–	–	0.01 s
	M205	–	–	–	0.04 s
master-only	N103	$3.1 \cdot 10^{-5}$ s	$1.1 \cdot 10^{-4}$ s	$2.0 \cdot 10^{-5}$ s	$1.1 \cdot 10^{-4}$ s
	N108	$2.1 \cdot 10^{-5}$ s	$1.4 \cdot 10^{-4}$ s	$1.6 \cdot 10^{-5}$ s	$3.8 \cdot 10^{-4}$ s
	N109	$3.2 \cdot 10^{-5}$ s	$1.7 \cdot 10^{-4}$ s	$2.4 \cdot 10^{-5}$ s	$1.1 \cdot 10^{-4}$ s
	N111	$1.1 \cdot 10^{-5}$ s	$3.6 \cdot 10^{-5}$ s	$9.0 \cdot 10^{-6}$ s	$4.9 \cdot 10^{-5}$ s
	N112	$4.0 \cdot 10^{-5}$ s	$2.4 \cdot 10^{-5}$ s	$5.0 \cdot 10^{-6}$ s	$1.5 \cdot 10^{-4}$ s
	N201	$4.9 \cdot 10^{-5}$ s	$5.8 \cdot 10^{-3}$ s	$1.0 \cdot 10^{-4}$ s	$1.7 \cdot 10^{-4}$ s
	N203	$1.3 \cdot 10^{-5}$ s	$3.5 \cdot 10^{-5}$ s	$1.6 \cdot 10^{-5}$ s	$6.7 \cdot 10^{-5}$ s
	N209	$9.5 \cdot 10^{-5}$ s	$1.8 \cdot 10^{-4}$ s	$2.4 \cdot 10^{-5}$ s	$1.1 \cdot 10^{-4}$ s
	N101	$2.9 \cdot 10^{-5}$ s	$4.5 \cdot 10^{-4}$ s	$8.3 \cdot 10^{-5}$ s	$4.4 \cdot 10^{-4}$ s
	N104	$5.4 \cdot 10^{-5}$ s	$2.0 \cdot 10^{-3}$ s	$4.6 \cdot 10^{-5}$ s	$3.8 \cdot 10^{-4}$ s
	N106	0.10 s	0.03 s	0.11 s	$5.8 \cdot 10^{-3}$ s
	N107	$2.2 \cdot 10^{-5}$ s	$6.4 \cdot 10^{-4}$ s	$2.9 \cdot 10^{-5}$ s	$2.3 \cdot 10^{-4}$ s
	N202	$1.5 \cdot 10^{-4}$ s	0.01 s	$8.8 \cdot 10^{-5}$ s	$5.1 \cdot 10^{-4}$ s
	N206	$5.5 \cdot 10^{-5}$ s	$5.4 \cdot 10^{-4}$ s	$4.4 \cdot 10^{-5}$ s	$2.7 \cdot 10^{-4}$ s
	N208	$3.1 \cdot 10^{-5}$ s	$4.5 \cdot 10^{-4}$ s	$3.6 \cdot 10^{-5}$ s	$1.4 \cdot 10^{-4}$ s
	N100	–	–	–	$9.7 \cdot 10^{-3}$ s
	N102	$1.7 \cdot 10^{-3}$ s	0.01 s	$1.9 \cdot 10^{-4}$ s	$1.9 \cdot 10^{-3}$ s
	N200	0.42 s	0.14 s	0.43 s	$5.5 \cdot 10^{-3}$ s
	N205	–	–	–	$8.4 \cdot 10^{-3}$ s

Table 6.4: Time required to find a solution (decision runtime) to the lock-charts in the real-world and master-only datasets.

Dataset		Autom.	CSP	Implicit	SAT
synthetic	D02M0025I02P06D100R	–	–	–	0.51 s
	D02M0040I02P10D050R	–	$9.2 \cdot 10^{-3}$ s	$8.4 \cdot 10^{-3}$ s	0.03 s
	D02M0081I02P10D100R	–	–	–	2.46 s
	D02M0020I06P02D100R	–	–	–	0.02 s
	D02M0187I06P06D001R	–	207.41 s	4.35 s	3.24 s
	D02M0025I10P02D010R	–	0.05 s	$7.0 \cdot 10^{-3}$ s	$8.6 \cdot 10^{-3}$ s
	D02M0050I10P02D020R	–	0.14 s	0.01 s	0.03 s
	D02M0126I10P02D050R	–	0.41 s	0.07 s	0.52 s
	D06M0025I02P06D100R	–	–	–	0.89 s
	D06M0040I02P10D050R	–	$3.4 \cdot 10^{-3}$ s	$3.9 \cdot 10^{-3}$ s	0.07 s
	D06M0081I02P10D100R	–	–	–	2.41 s
	D06M0020I06P02D100R	–	–	–	0.26 s
	D06M0187I06P06D001R	–	203.77 s	2.33 s	3.48 s
	D06M0025I10P02D010R	–	–	–	$8.4 \cdot 10^{-3}$ s
	D06M0050I10P02D020R	–	–	–	0.06 s
	D06M0126I10P02D050R	–	–	–	4.70 s
	D10M0025I02P06D100R	–	–	–	1.16 s
	D10M0040I02P10D050R	–	–	–	0.07 s
	D10M0081I02P10D100R	–	–	–	2.83 s
	D10M0020I06P02D100R	–	–	–	0.33 s
	D10M0187I06P06D001R	–	462.76 s	4.98 s	4.17 s
	D10M0025I10P02D010R	–	–	–	0.03 s
	D10M0050I10P02D020R	–	–	–	0.28 s
	D10M0126I10P02D050R	–	–	–	9.46 s

Table 6.5: Time required to find a solution (decision runtime) to the lock-charts in the synthetic dataset.

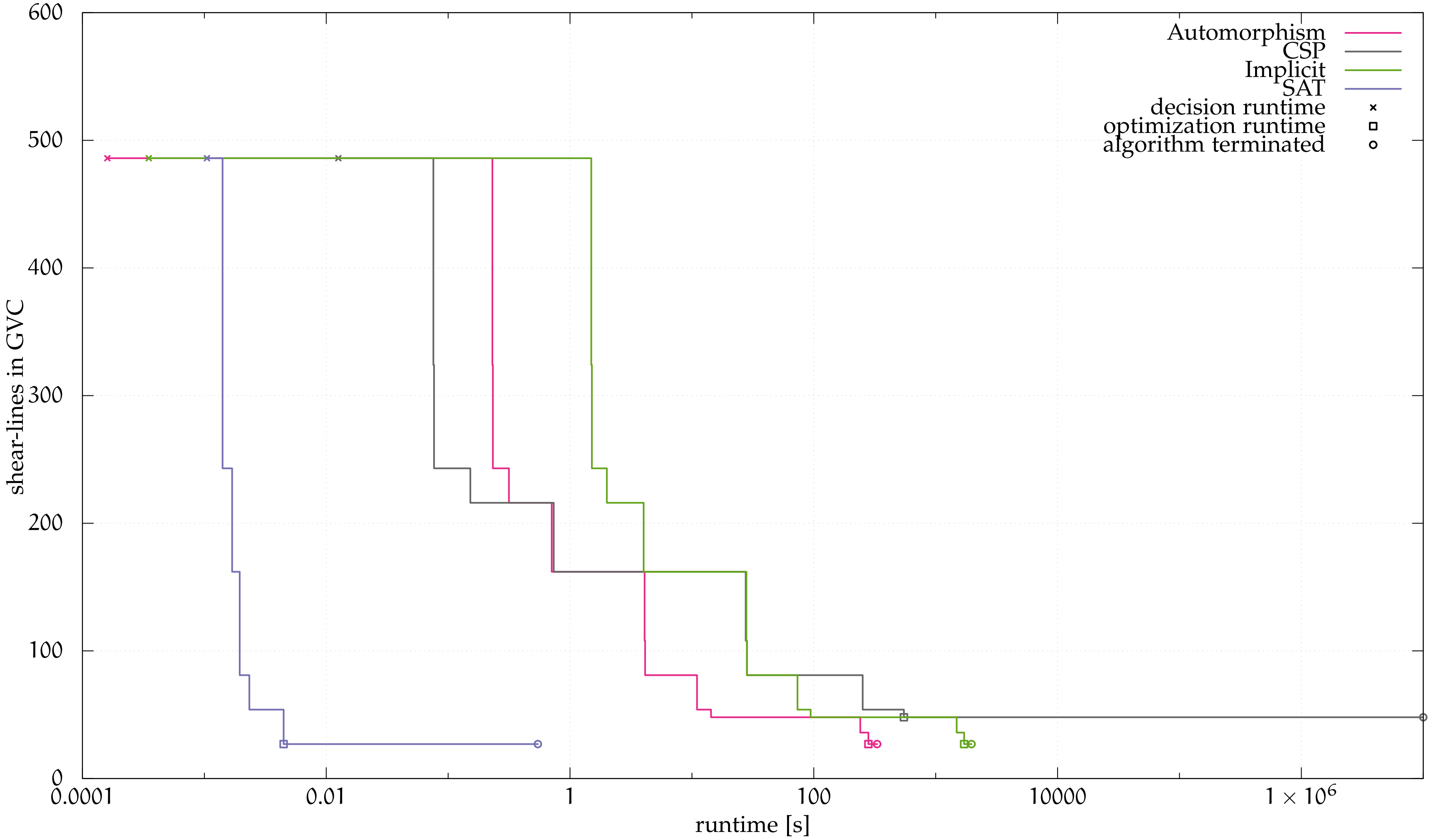


Figure 6.1: Convergence of $|\Lambda|$ in the best solution found so far (b) on the `N102` lock-chart.

real-world	Autom.	CSP	Implicit	SAT
Autom.	1.00	1.04	1.10	124.17
CSP	0.96	1.00	1.15	56.33
Implicit	0.91	0.87	1.00	48.96
SAT	$8.1 \cdot 10^{-3}$	0.02	0.02	1.00

synthetic	CSP	Implicit	SAT
CSP	1.00	4.90	9.04
Implicit	0.20	1.00	1.84
SAT	0.11	0.54	1.00

Table 6.6: Pair-wise comparison of decision runtimes, averaged by geometric mean. Values $\leqslant 1$ mean that the row-algorithm beats the column-algorithm. Master-only dataset is not listed, because of small absolute values in Table 6.4, whose ratio is numerically unstable.

Dataset		Autom.	CSP	Implicit	SAT
real-world	M103	64	64	64	64
	M108	48	64	64	64
	M109	27	27	27	27
	M111	25	25	25	25
	M112	25	25	25	25
	M201	64	64	64	64
	M203	–	–	–	64
	M204	2	–	–	2
	M209	64	64	64	64
	M101	–	–	–	125
	M104	–	–	–	162
	M106	–	–	–	162
	M107	–	125	125	100
	M202	–	–	–	512
	M206	–	–	–	81
	M208	–	–	–	81
	M100	–	–	–	729
	M102	–	–	–	486
	M200	–	–	–	100
	M205	–	–	–	243
master-only	N103	16	16	16	16
	N108	8	8	8	8
	N109	9	9	9	9
	N111	4	4	4	4
	N112	3	3	3	3
	N201	27	27	27	32
	N203	3	3	3	3
	N209	16	16	16	16
	N101	12	12	12	12
	N104	24	24	24	27
	N106	54	54	54	54
	N107	4	4	4	4
	N202	128	128	128	128
	N206	9	9	9	9
	N208	7	7	7	7
	N100	–	–	–	486
	N102	27	48	27	27
	N200	50	50	50	25
	N205	–	–	–	81

Table 6.7: $|\Lambda|$ by the best found solution in the real-world and master-only datasets.

	Dataset	Autom.	CSP	Implicit	SAT
synthetic	D02M0025I02P06D100R	–	–	–	36
	D02M0040I02P10D050R	–	70	70	60
	D02M0081I02P10D100R	–	–	–	100
	D02M0020I06P02D100R	–	–	–	64
	D02M0187I06P06D001R	–	19440	960	864
	D02M0025I10P02D010R	–	256	256	128
	D02M0050I10P02D020R	–	512	512	512
	D02M0126I10P02D050R	–	1024	1024	1024
	D06M0025I02P06D100R	–	–	–	36
	D06M0040I02P10D050R	–	80	80	80
	D06M0081I02P10D100R	–	–	–	100
	D06M0020I06P02D100R	–	–	–	64
	D06M0187I06P06D001R	–	17280	1944	864
	D06M0025I10P02D010R	–	–	–	256
	D06M0050I10P02D020R	–	–	–	512
	D06M0126I10P02D050R	–	–	–	1024
	D10M0025I02P06D100R	–	–	–	36
	D10M0040I02P10D050R	–	–	–	90
	D10M0081I02P10D100R	–	–	–	100
	D10M0020I06P02D100R	–	–	–	64
	D10M0187I06P06D001R	–	10368	8640	1296
	D10M0025I10P02D010R	–	–	–	512
	D10M0050I10P02D020R	–	–	–	1024
	D10M0126I10P02D050R	–	–	–	1024

Table 6.8: $|\Lambda|$ by the best found solution in the synthetic dataset.

Dataset		Autom.	CSP	Implicit	SAT
real-world	M103	14.43 s	18.29 s	22.71 s	$2.6 \cdot 10^{-3}$ s
	M108	$6.3 \cdot 10^{3}$ s	$5.0 \cdot 10^{3}$ s	$5.0 \cdot 10^{3}$ s	0.01 s
	M109	31.59 s	192.35 s	100.85 s	$6.4 \cdot 10^{-3}$ s
	M111	0.01 s	$2.1 \cdot 10^{-3}$ s	$2.5 \cdot 10^{-3}$ s	$3.7 \cdot 10^{-3}$ s
	M112	3.85 s	4.48 s	6.14 s	0.03 s
	M201	$9.2 \cdot 10^{-3}$ s	0.01 s	0.01 s	0.03 s
	M203	–	–	–	0.21 s
	M204	$7.0 \cdot 10^{-6}$ s	–	–	$3.1 \cdot 10^{-5}$ s
	M209	248.94 s	739.17 s	373.76 s	0.08 s
	M101	–	–	–	0.94 s
	M104	–	–	–	0.02 s
	M106	–	–	–	0.01 s
	M107	–	$2.6 \cdot 10^{-3}$ s	$1.2 \cdot 10^{-3}$ s	0.09 s
	M202	–	–	–	0.01 s
	M206	–	–	–	0.11 s
	M208	–	–	–	0.09 s
	M100	–	–	–	0.07 s
	M102	–	–	–	1.19 s
	M200	–	–	–	0.02 s
	M205	–	–	–	0.04 s
master-only	N103	$3.1 \cdot 10^{-5}$ s	$1.1 \cdot 10^{-4}$ s	$2.0 \cdot 10^{-5}$ s	$1.1 \cdot 10^{-4}$ s
	N108	$3.8 \cdot 10^{-5}$ s	$1.5 \cdot 10^{-4}$ s	$3.2 \cdot 10^{-5}$ s	$3.8 \cdot 10^{-4}$ s
	N109	0.01 s	0.03 s	0.01 s	$3.2 \cdot 10^{-4}$ s
	N111	$3.2 \cdot 10^{-5}$ s	$6.9 \cdot 10^{-5}$ s	$2.6 \cdot 10^{-5}$ s	$4.9 \cdot 10^{-5}$ s
	N112	$5.3 \cdot 10^{-5}$ s	$2.8 \cdot 10^{-5}$ s	$8.0 \cdot 10^{-6}$ s	$1.5 \cdot 10^{-4}$ s
	N201	0.36 s	1.01 s	0.35 s	$1.5 \cdot 10^{-3}$ s
	N203	$2.1 \cdot 10^{-5}$ s	$4.4 \cdot 10^{-5}$ s	$2.6 \cdot 10^{-5}$ s	$6.7 \cdot 10^{-5}$ s
	N209	$4.4 \cdot 10^{-3}$ s	$3.7 \cdot 10^{-3}$ s	$1.1 \cdot 10^{-3}$ s	$1.8 \cdot 10^{-4}$ s
	N101	$4.4 \cdot 10^{-3}$ s	0.22 s	0.02 s	0.01 s
	N104	0.42 s	14.49 s	0.38 s	$6.2 \cdot 10^{-4}$ s
	N106	567.38 s	$6.2 \cdot 10^{3}$ s	593.01 s	$6.8 \cdot 10^{-3}$ s
	N107	$6.4 \cdot 10^{-5}$ s	$1.3 \cdot 10^{-3}$ s	$9.6 \cdot 10^{-5}$ s	$2.3 \cdot 10^{-4}$ s
	N202	$4.4 \cdot 10^{-3}$ s	0.01 s	$1.4 \cdot 10^{-3}$ s	$7.2 \cdot 10^{-3}$ s
	N206	$4.7 \cdot 10^{-3}$ s	0.03 s	$2.7 \cdot 10^{-3}$ s	$2.7 \cdot 10^{-4}$ s
	N208	$6.3 \cdot 10^{-4}$ s	$7.4 \cdot 10^{-3}$ s	$9.4 \cdot 10^{-4}$ s	$1.4 \cdot 10^{-4}$ s
	N100	–	–	–	0.01 s
	N102	418.56 s	911.59 s	418.73 s	$5.1 \cdot 10^{-3}$ s
	N200	$1.3 \cdot 10^{3}$ s	$9.8 \cdot 10^{3}$ s	$1.6 \cdot 10^{3}$ s	0.02 s
	N205	–	–	–	$9.2 \cdot 10^{-3}$ s

Table 6.9: Time required to find the best solution (optimisation time) to lock-charts in the real-world and master-only datasets.

7 CONCLUSIONS

Lock-chart solving is a complex problem, both theoretically and practically. The formalism proposed in this study is relatively simple, yet expressive enough to capture all intricacies of several real-world mechanical platforms. Many real-world problems coming from industrial needs, such as finding largest diagonal lock-charts, are either polynomially solvable or can be approximated using a polynomial procedure to a satisfactory degree. Other real-world problems, such as solving large unstructured lock-charts or counting available cuttings, are not practically feasible given the state-of-the-art computers and algorithms.

Lock-chart solving is a problem overlooked by academia. We know only of 4 publicly available studies [34, 38, 44, 53] on the topic of computational lock-chart solving, which might be surprising given that the problem has many interesting properties. For example, the hierarchy of lock-chart formalisms contains the $\mathcal{P}/\mathcal{NP}$ boundary. This study proved that in vanilla framework, diagonal and key-to-differ lock-charts are solvable in $\mathcal{P}$, yet more expressive lock-charts are $\mathcal{NP}$-complete. Moreover, the simplistic formalism of lock-charts allows researchers to attack the problem using various discrete optimisation approaches. Here we have successfully applied SAT solvers and some CSP-inspired techniques.

Lock-chart solving is a great challenge for hackers.[1] Tackling $\mathcal{NP}$-complete problems has always been approached both by theoretical breakthroughs and dirty tricks. The apparent structure and symmetries in lock-charts call for many ideas of the latter kind. Given the popularity of contests in lock-picking, which is essentially a kind of mechanical hacking, there is no reason to believe that lock-chart solving cannot become a hacking quest for IT people.

Lock-chart solving is a good exercise for teaching algorithmization in schools, an observation already mentioned in [53]. Students need a low overhead to start. Since tuples are naturally represented by arrays and tree/hash sets are one of the earliest taught data structures, probably the hardest notion to understand before diving into lock-chart solving is a graph.

Here is a list of several opportunities for future development.

[1] We mean ethical hackers.

- Personally, the most intriguing open question is the status of basic lock-charts in the vanilla framework. Diagonal lock-charts are in $\mathcal{P}$, yet extension and melted profiles lock-charts are $\mathcal{NP}$-complete. In which class do the basic lock-charts belong to?

- Solving $\mathcal{NP}$-complete tasks is usually concerned with the runtime. Unless $\mathcal{P} = \mathcal{NP}$, the runtime is inevitably exponential, which limits the algorithm capabilities. However, in Section 4.2 we saw that the limitation of SAT solvers is not their runtime, but memory requirements. The work on reducing the memory footprint of SAT solvers focuses on reducing the footprint of learnt clauses [4, 23] or proofs of formula's unsatisfiability [56]. Another line of research can investigate methods of storing or reducing the CNF itself and keep a comparable runtime to the current generation of SAT solvers.

- Practical solvers that optimise for extensibility would benefit from a deeper analysis of independent keys. What is the largest lock-chart of independent keys if the code space is defined in the explicit framework?

- A natural variation of employing SAT solvers is to explore other formalisms for discrete optimisation, such as ILP as proposed in Section 4.3.

- The cutting counting algorithm with general constraints is in its early stages of development and offers many research directions. For example, can the code space S be sampled to get merely approximate values $|S_q|$, yet precise enough to evaluate $\hat{q} = \arg\max_q |S_q|$?

- Better cutting counting algorithm can improve the scalability of backtracking algorithms, which is the main drawback of SAT solvers (see Section 4.2). Even though the implicit algorithm (from Section 6.3) is not complete and does not ensure finding the optimal solution, it achieved better results than other backtrackers did due to performance gains. The trade-off between completeness and performance should be explored in more detail.

Finally, there is a personal wish. It would be great to see a small lock-chart solving community. Given a commercial interest, the undeniable romantic flair of mechanical locks and a potential of a healthy competition, there is no reason why the topic should not attract more enthusiasts. Hopefully, this text will help people willing to take up lock-chart solving. A second important component would be a publicly available dataset of constraints from real-world platforms and a dataset of lock-charts. I promise to try persuading our commercial partners towards this direction.

BIBLIOGRAPHY

[1] Scott Aaronson. $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$. 2017. URL: https://www.scottaaronson.com/blog/?p=3095.

[2] Emile Aarts and Jan K. Lenstra, eds. *Local Search in Combinatorial Optimization*. 1st. New York, NY, USA: John Wiley & Sons, Inc., 1997. ISBN: 0471948225.

[3] Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, eds. *Fast Algorithms for Mining Association Rules*. Morgan Kaufmann, 1994, pp. 487–499.

[4] Gilles Audemard and Laurent Simon. "Predicting Learnt Clauses Quality in Modern SAT Solvers". In: *Proceedings of the 21st International Jont Conference on Artifical Intelligence*. IJCAI'09. Pasadena, California, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 399–404.

[5] László Babai. "Graph Isomorphism in Quasipolynomial Time". In: *Proceedings of the Forty-eighth Annual ACM Symposium on Theory of Computing*. STOC '16. Cambridge, MA, USA: ACM, 2016, pp. 684–697. ISBN: 978-1-4503-4132-5. DOI: 10.1145/2897518.2897542.

[6] László Babai. *Fixing the UPCC case of Split-or-Johnson*. [Online; accessed 2-September-2017]. 2017.

[7] Piotr Berman and Martin Fürer. "Approximating Maximum Independent Set in Bounded Degree Graphs." In: *SODA*. Vol. 94. 1994, pp. 365–371.

[8] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009. ISBN: 1586039296, 9781586039295.

[9] Christian Bliek, Pierre Bonami, and Andrea Lodi. "Solving mixed-integer quadratic programming problems with IBM-CPLEX: a progress report". In: *Proceedings of the Twenty-Sixth RAMP Symposium*. Hosei University, Tokyo, Japan, 2014.

[10] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. ISBN: 0521833787.

[11] Wikimedia Commons. *File:Standard-lock-key.jpg — Wikimedia Commons, the free media repository*. [Online; accessed 20-October-2017]. 2010. URL: https://ja.wikipedia.org/wiki/%E3%83%95%E3%82%A1%E3%82%A4%E3%83%AB:Standard-lock-key.jpg.

[12] Wikimedia Commons. *File:Polhemslås skiss.png — Wikimedia Commons, the free media repository*. [Online; accessed 20-October-2017]. 2015. URL: \url{https://commons.wikimedia.org/w/index.php?title=File:Polhemsl%C3%A5s_skiss.png&oldid=152861961}.

[13] Wikimedia Commons. *File:Chiave simil-Abloy.JPG — Wikimedia Commons, the free media repository*. [Online; accessed 20-October-2017]. 2016. URL: https://commons.wikimedia.org/w/index.php?title=File:Chiave_simil-Abloy.JPG&oldid=197571188.

[14] Wikimedia Commons. *File:Ad for Scandinavian padlock.jpg — Wikimedia Commons, the free media repository*. [Online; accessed 20-October-2017]. 2017. URL: https://commons.wikimedia.org/w/index.php?title=File:Ad_for_Scandinavian_padlock.jpg&oldid=256874604.

[15] Martin C Cooper, Simon de Givry, Martı Sánchez, Thomas Schiex, Matthias Zytnicki, and Tomáš Werner. "Soft arc consistency revisited". In: *Artificial Intelligence* 174.7 (2010), pp. 449–478.

[16] Martin Davis, George Logemann, and Donald Loveland. "A Machine Program for Theorem-proving". In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557.

[17] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034.

[18] Rina Dechter and Avi Dechter. *Belief maintenance in dynamic constraint networks*. University of California, Computer Science Department, 1988.

[19] Niklas Eén and Niklas Sörensson. "An extensible SAT-solver". In: *Theory and applications of satisfiability testing*. Springer. 2003, pp. 502–518.

[20] Stefan Felsner, Vijay Raghavan, and Jeremy Spinrad. "Recognition Algorithms for Orders of Small Width and Graphs of Small Dilworth Number". In: *Order* 20.4 (2003), pp. 351–364. ISSN: 1572-9273. DOI: 10.1023/B:ORDE.0000034609.99940.fb.

[21] Eugene C Freuder and Richard J Wallace. "Partial constraint satisfaction". In: *Artificial Intelligence* 58.1-3 (1992), pp. 21–70.

[22] Georg Gottlob and Stefan Szeider. "Fixed-parameter algorithms for artificial intelligence, constraint satisfaction and database problems". In: *The Computer Journal* 51.3 (2008), pp. 303–325.

[23] Orna Grumberg, Assaf Schuster, and Avi Yadgar. "Memory Efficient All-Solutions SAT Solver and Its Application for Reachability Analysis". In: *Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings*. Ed. by Alan J. Hu and Andrew K. Martin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 275–289. ISBN: 978-3-540-30494-4.

[24] Inc. Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2017. URL: http://www.gurobi.com.

[25] Magnús M Halldórsson and Jaikumar Radhakrishnan. "Greed is good: Approximating independent sets in sparse and bounded-degree graphs". In: *Algorithmica* 18.1 (1997), pp. 145–163.

[26] Steven Hampton. *Secrets Of Lock Picking*. Paladin Press, 1987.

[27] *Hatton Garden safety deposit box vault burgled*. 2015. URL: http://www.bbc.com/news/uk-england-london-32207974.

[28] Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. "Super solutions in constraint programming". In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer, 2004, pp. 157–172.

[29] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321455363.

[30] *IBM ILOG CPLEX Optimizer*. 2017. URL: https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[31] *Ilog Solver. V4.0. Reference manual and user manual*. 1997.

[32] Peter James and Nick Thorpe. *Ancient Inventions*. New York: Ballantine Books, 1995.

[33] Luke Jones and Tom de Castella. *Is the traditional metal key becoming obsolete?* 2014. URL: http://www.bbc.com/news/magazine-29817520.

[34] Ulrich Junker. "Constraint-based Problem Decomposition for a Key Configuration Problem". In: *Principles and Practice of Constraint Programming* 1520 (1999), pp. 265–279.

[35] H Kautz, B Selman, and D McAllester. "Walksat in the SAT 2004 competition". In: *International Conference on Theory and Applications of Satisfiability Testing (SAT04)*. 2004.

[36] Leonid G Khachiyan. "A polynomial algorithm in linear programming". In: *Doklady Akademiia Nauk SSSR*. Vol. 244. 1979, pp. 1093–1096.

[37] Nicolas Lachiche. “Propositionalization”. In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 812–817. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_680.

[38] Anna Lawer. “Calculation of Lock Systems”. MA thesis. Royal Institute of Technology, 2004.

[39] Alan K. Mackworth. “On Reading Sketch Maps”. In: *Proceedings of the 5th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI’77. Cambridge, USA: Morgan Kaufmann Publishers Inc., 1977, pp. 598–606.

[40] Mark McCloud, Gonzalez de Santos, and Mirko Jugurdzija. *Visual Guide to Lock Picking*. third. Standard Publications, Inc., 2007.

[41] Brendan D. McKay and Adolfo Piperno. “Practical graph isomorphism, {II}”. In: *Journal of Symbolic Computation* 60.0 (2014), pp. 94 –112. ISSN: 0747-7171.

[42] *Mul-T-Lock CLIQ*. [Online; accessed 20-October-2017]. 2017. URL: http://www.mul-t-lock-cliq.com/.

[43] Deviant Ollam. *Practical Lock Picking. A Physical Penetration Tester’s Training Guide*. second. Syngress, 2012.

[44] Don O’Shall. *The Definitive Guide to RCM – Rotating Constant Method of Master Keying*. Locksmithing Education, 2015. ISBN: 9781937067137. URL: https://books.google.cz/books?id=5Hz_rQEACAAJ.

[45] Charles Prud’homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC; INRIA Rennes; LINA CNRS UMR 6241; COSLING S.A.S. 2015. URL: http://www.choco-solver.org.

[46] G.W. Pulford. *High-security Mechanical Locks: An Encyclopedic Reference*. Elsevier Butterworth-Heinemann, 2007. ISBN: 9780750684378. URL: https://books.google.cz/books?id=1WeVeMMGQKEC.

[47] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[48] Bart Selman, Henry A Kautz, Bram Cohen, et al. “Local search strategies for satisfiability testing.” In: *Cliques, coloring, and satisfiability* 26 (1993), pp. 521–532.

[49] Bart Selman, Hector Levesque, and David Mitchell. “A New Method for Solving Hard Satisfiability Problems”. In: *Proceedings of the Tenth National Conference on Artificial Intelligence*. AAAI’92. San Jose, California: AAAI Press, 1992, pp. 440–446. ISBN: 0-262-51063-4.

[50] Mate Soos. "CryptoMiniSat v4". In: *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*. Helsinki, Finland: University of Helsinki, 2014. ISBN: 978-951-51-0043-6.

[51] Maciej M. Syso. "The subgraph isomorphism problem for outerplanar graphs". In: *Theoretical Computer Science* 17.1 (1982), pp. 91 –97. ISSN: 0304-3975.

[52] Robert Endre Tarjan and Anthony E Trojanowski. "Finding a maximum independent set". In: *SIAM Journal on Computing* 6.3 (1977), pp. 537–546.

[53] Christof Vomel, Flavio de Lorenzi, Samuel Beer, and Erwin Fuchs. "The Secret Life of Keys: On the Calculation of Mechanical Lock Systems". In: *SIAM Review* 59.2 (2017), pp. 393–422.

[54] B. Widén. *Cylinder lock with profiled keyway*. US Patent App. 13/184,305. 2011. URL: https://www.google.cz/patents/US20110271723.

[55] Wikipedia. *Locksport — Wikipedia, The Free Encyclopedia*. [Online; accessed 20-October-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Locksport&oldid=798922145.

[56] Lintao Zhang and Sharad Malik. "Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications". In: *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*. DATE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 10880–. ISBN: 0-7695-1870-2.

[57] Dennis van Zuijlekom. *Börkey 954-2 Key Cutting Machine*. 2013. URL: https://www.flickr.com/photos/dvanzuijlekom/8489934067/.