Calculation of Lock Systems

ANNA LAWER

KTH Numerical Analysis
and Computer Science

Master's Degree Project
Stockholm, Sweden 2004

TRITA-NA-E04081

Numerisk analys och datalogi
KTH
100 44 Stockholm

Department of Numerical Analysis
and Computer Science
Royal Institute of Technology
SE-100 44 Stockholm, Sweden

Calculation of Lock Systems

ANNA LAWER

TRITA-NA-E04081

Master's Thesis in Computer Science (20 credits)
at the School of Engineering Physics,
Royal Institute of Technology year 2004
Supervisor at Nada was Mikael Goldmann
Examiner was Johan Håstad

Calculation of Lock Systems

Abstract

A lock system is a set of keys and cylinders where each key opens some subset of cylinders. Lock system calculation is concerned with designing cylinders and keys in order to make the lock system work in the specified way and at the same time minimise the number of keys that are *consumed*. A key is considered to be consumed if it fits in the smallest cylinder that accepts every manufactured key in the lock system. By *smallest cylinder* we refer to how many keys (counting both manufactured and not manufactured keys) that open the cylinder. This is a definition that is used by many companies in the Assa-Abloy group. A key that is *consumed* may not be manufactured and sold to a customer other than the owner of the lock system that *consumed* the key. Today there is no known polynomial time algorithm that can calculate the design of the keys and cylinders in order to make them fulfil the requirements of a customer *consuming* as few keys as possible. Most of the calculations are made manually. It is not known if the problem is NP-complete or not. The main goal of this Master's project was to create a tool for calculating lock systems that are particularly hard to calculate manually.

We implemented a backtracking algorithm combined with pruning and some heuristics. The pruning techniques are based on automorphisms on the search space for the backtracking algorithm. The implementation outperformed manually found solutions, with regard to *consumed* key codes, in most cases when the lock system was considered as difficult by the lock system calculators. We must add that the search spaces for manual calculations and the search space given to the algorithm were not entirely identical. The solutions are thus nor directly comparable.

Another result is the NP-completness of a problem faced by lock system calculators when a lock system is to be extended.

Beräkning av låssystem

Sammanfattning

Ett låsystem är en mängd nycklar och lås där varje nyckel öppnar någon delmängd lås. Låssystemberäkning handlar om att utforma nycklar och lås så att låssystemet fungerar på ett i förväg definierat sätt och på samma gång minimera antalet nycklar som anses vara *förbrukade* av låssystemet. En nyckel anses vara förbrukad om den passar i det minsta lås som öppnas av alla tillverkade nycklar i låssystemet. Med minsta syftar vi här på antalet nycklar som passar i låset. Detta är en definition som används av många företag i Assa-Abloy koncernen. En nyckel som anses vara *förbrukad* får ej tillverkas och säljas till någon annan kund än ägaren av det låssystem som *förbrukade* nyckeln. Idag finns det inte någon polynomiell algoritm som kan beräkna hur nycklarna skall se ut för att dels uppfylla kundens krav på låssystemet, dels *förbruka* så få nycklar som möjligt. En stor del av arbetet görs manuellt. Det är inte känt om problemet är NP-fullständigt eller ej. Det huvudsakliga målet med detta examensarbete var att skapa en algoritm som gör de nödvändiga beräkningarna för att tillverka låssystem som upplevs som speciellt svåra av dem som arbetar med beräkning av låssystem.

Vi implementerade en algoritm som gjorde djupet-först-sökning. För att minska sökrymden använde vi oss av automorfier på denna. Vidare införde vi en heurestik för att ytterligare begränsa sökträdet. Algoritmen *förbrukade* i regel mindre nycklar än lösningar som hittats med hjälp av manuella kalkyler understödda av annan mjukvara. Man bör tillägga att sökrymden för manuella beräkningar och automatiska var ej identiska. Lösningarna är därmed ej helt jämförbara.

Ett annat resultat är reduktionen som bevisar att det beräkningstekniska problem man stöter på när man vill utvidga ett redan existerande låssystem är NP-fullständigt.

Contents

Chapter 1

Introduction

A lock system is a set of keys and cylinders installed in one or serveral buildings. You can find lock systems everywhere, in office and factory buildings as well as in schools, churches and apartment buildings. A lock-system is tailor made for each customer. For each user you decide which doors they should be able to open with their key. Often one user can open more than one door. Different users can open different sets of doors, based on their specific needs.
When inserting a key in a lock a number of pins are pushed up against the edge of the key measuring the height at certain places. If all these heights are compatible with the cylinder, the lock will open. Today these key cut heights are calculated manually by lock system calculators. Even though there exists a range of tools to facilitate the work of the lock system calculator, none of these tools are entirely automatic. The crucial work, and the quality of the solution relies on the lock system calculator.
For one specific cylinder technology there is a limit to how deap/shallow we can make one cut. There is also a lower bound for the minimal difference between two key cuts. Because of this there will be a limited number of different keys that can be produced for one specific cylinder technology. When calculating a specific master key system the calculator will *reserve* a certain number of different keys for this system. This reservation will make it possible to extend the system and change user access in the future. When all the keys in a cylinder technology are reserved then this cylinder technology is exhausted and you must invent a new technology. The development of a new technology is rather costly because if often entrails the purchase of new expensive machines. A tool capable of automatic calculation of lock systems in a way that minimises key codes allocated to a system, would thus save a lot of time and labour. This Master's project was initiated with the intent to discover how far you could take the automation of lock system calculation, by using an algorithmic approach.

1.1 Project Goal and Limitations

The goals of this master's project are:

1. The development of an algorithm that specifies the key cuts for keys in a specific lock system, in a way that satisfies the lock chart of this lock system, for unstructured and relatively small problems (up to about 100

keys). The number of master keys is limited to no more than 30, the rest of the keys are individual keys.

2. An estimation of the quality of these solutions with regard to consumed key codes.

1.2 Methods and Main Results

After the evaluation of several different algorithms a backtracking algorithm was chosen and implemented. To reduce the search space for the algorithm we used a pruning technique and a heuristic. The pruning technique is based on automorphisms on the search space. The heuristic consists in estimating the quality of partial solutions and ignoring partial solutions that do not look promising. The algorithm outperformed/did as well as manually found solutions, with regard to *consumed* key codes, on 19 out of the 23 problems it was benchmarked on.

Another result is the NP-completness of a problem faced by lock system calculators when the lock system is going to be extended.

1.3 Report Outline

The second chapter describes how pin tumbler cylinders and lock systems work.

The third chapter presents an abstraction away from the real world technology to a representation of keys and cylinders that is more convenient for the implementation of the algorithms presented in Chapter 6.

In the fourth chapter we explain some elementary complexity theory and apply it to the lock system calculation problem.

Next the different approaches to attack the lock system calculation problem are evaluated.

The last three chapters treat the implementation of the chosen strategy, a presentation of the obtained results, and finally a conclusion and some suggestions for further work.

Chapter 2

Problem Definition

This chapter is devoted to how a pin tumbler cylinder and a lock system function. Readers familiar with lock system technology can skip this section.

2.1 Pin-tumbler Technology

The classical pin-tumbler cylinder consists of two mechanisms that decide if the key will be able to open the door or not; the *profile* of the key and its *key cuts*. The profile consists of a number of side wards. See Figure 2.1. The cylinder in this Figure accepts side wards in position 1, 4, 5, 7, 9, 10, 11 and 13. The key inserted in Figure 2.1 has a side ward in position 1, 4, 9, 10, 11, 13. As you can see from this Figure it will be possible to insert any key that has a subset of the allowed sidewards.

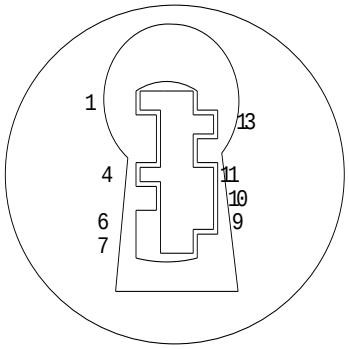


Figure 2.1: Key inserted in cylinder

When inserting the key in the cylinder a number of pins will be pushed up against the key cuts. In Figure 2.2 we have 4 pins that have been pushed up against the edge of a key. The pins are segmented in one or several places. If the key cuts have the appropriate height then the pins will stop in a position where they are segmented on the same level: the level of the shear line which is on the edge of the cylinder core. One will thus be able to turn the cylinder core and the door will open. If the key cuts have the wrong heights then one or

several pins will not be segmented at the shear line and will block any attempt to turn the cylinder core.

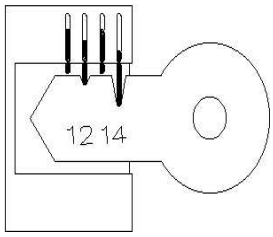


Figure 2.2: Cylinder pins and the shear line

2.2 Lock-system Technology

A lock system consists of a number of cylinders and keys. Each key opens a certain subset of cylinders in the lock system. One way to specify a lock system is to complete a lock chart. Keys are represented by rows and cylinders by columns. A cross in row x and column y indicates that key x should open cylinder y. A blank on the other hand means that key x does not open cylinder y, see Table 2.1 for an example of a lock chart. Keys that open several cylinders are referred to as *master keys*. If a key opens only one door it is called an individual key. A structured lock system is a system that can be represented by a tree. Each node in the tree represents a key. Each key open all the doors that are opened by the keys in its subtree. The key in the root of this tree opens all the doors in the lock system and is referred to as the *grand master key*.

All other lock systems are referred to as unstructured lock systems. Table 2.1 represents a lock chart for a structured lock system and Table 2.2 shows a lock chart for an unstructured lock system.

Key/Cylinder	1	2	3	4
MK1	X	X	X	X
MK2	X	X		
MK3			X	X
K4	X			
K5		X		
K6			X	
K7				X

Table 2.1: Structured lock system

Key/Cylinder	1	2	3	4
MK1	X	X	X	X
MK2	X	X		
MK3			X	X
K4	X		X	
K5		X		X
K6			X	
K7	X			X

Table 2.2: Unstructured lock system

Chapter 3

Lock System Calculation Theory

This chapter defines and explains some terminology and notation that will be used throughout this report.

3.1 Representation of the Elements Necessary for Lock System Calculation

This section contains mathematical representations of the key and the cylinder.

3.1.1 The Key

The different depths of the key cuts can be represented by numbers. A sequence of numbers can therefore represent the key cuts. For example

$$< 1, 2, 1, 4 >$$

means that the corresponding key has a cut of depth 1 in position 1, a cut of depth 2 in position 2 etc. See Figure 2.2. This sequence of numbers is referred to as a *key code*.

3.1.2 The Cylinder

The cylinder is specified by an ordered sequence of sets $S_1 ... S_n$. The first set indicates which key cuts are accepted in the first position, the second set which cuts are accepted in the second position etc. There are $|S_1| \cdot |S_2| \cdot \ldots \cdot |S_n|$ potential keys that open a cylinder.

Once you have fixed the appropriate key cuts, the design of the cylinders will become fixed; each cylinder will minimise the number of keys that operate it and still fulfil the lock chart requirements. How you construct this cylinder is explained in the next section.

3.1.3 The Cost Function

When manufacturing a lock system you want to *consume* a minimal number of key codes. A key code is considered as consumed if it operates the *minimal cylinder* that is operated by all keys in a given lock system.

Definition 3.1.1 *The* minimal cylinder $S_1 \cdot \ldots \cdot S_n$*, is the cylinder where* S_i *is the set of key cuts that occur in keys belonging to the lock system in position* i*.*

This cost function was chosen over the more intuitive cost function (the cost function that counts a key code as consumed if the corresponding key opens an existing lock in the lock system), because it allocates the consumed key codes to the lock system and allows future expansions of the lock system.

Definition 3.1.2 *A sub cylinder,* $R_1 \times \ldots \times R_n$*, is a cylinder where:*

$$R_i \subseteq S_i \quad \forall i = 1, \ldots, n$$

Here $S_1 \times \ldots \times S_n$ *is the* minimal cylinder *of the lock system.*

3.1.4 The Search Space

The search space is defined by taking the Cartesian product of sets of numbers that represent key cuts. For example:

$$S_1 \times S_2 \times S_3$$

where

$$S_1 = \{0, 1, 2\}, S_2 = \{0, 1\}, S_3 = \{0\}$$

defines a cylinder that also can be considered as a search space.

Chapter 4

NP Theory

The following chapter introduces the reader to basic definitions used in algorithmics to classify the difficulty level for solving differnet problems. The definitions can be found in for instance [2]. These definitions will then be applied to the lock system calculation problem and some conclusions regarding the problem's difficulty level are drawn.

4.1 NP Theory

Definition 4.1.1 *A problem is said to be in P if there is an algorithm that solves the problem in a time that is polynomial in its size.*

Definition 4.1.2 *A decision problem is a problem that has two possible answers: yes or no.*

Definition 4.1.3 *A certificate to a decision problem is an object of the right kind that proves a yes answer. This check is done in polynomial time.*

Definition 4.1.4 *A nondeterministic algorithm is an algorithm that has three phases:*

1. *A nondeterministic guessing phase where an arbitrary certificate s to the problem is designed.*
2. *A deterministic verifying phase where the certifiacate s may or may not be used. Eventually it returns either* true *or* false *or it may get in an infinite loop.*
3. *The output step. If the verifying phase returned* yes *the output is yes. Otherwise there is no output.*

Definition 4.1.5 *The problem is said to be in NP if there is a polynomially bounded nondeterministic algorithm for the problem.*

Definition 4.1.6 *When talking about* NP-complete decision problems *you refer to problems which are in NP and where you can, for every problem in NP, find an algorithm that runs in polynomial time and transforms an instance of that*

problem to an instance of the NP-complete problem. The transformation should be such that the answer to this instance of our NP-complete problem is the same as the answer to the original problem.

Definition 4.1.7 *A* problem instance *is the combination of the problem and a specific input. Usually the statement of an optimisation problem has two parts:*

1. *The* instance descripton *part that defines the information expected in the input.*
2. *The* objective function*, given an instance description of the problem, returns a value. It is this value that we try to minimise/maxmimise.*

Definition 4.1.8 *An optimisation problem belongs to the class NPO if:*

1. *The set of instances is recognizable in polynomial time.*
2. *The objective function is computable in polynomial time*
3. *There exists a polynomial q such that, given an instance x of the problem, there is a solution y to this problem such that $|y| \leq q(|x|)$. And for any y such that $|y| \leq q(|x|)$ it is decidable in polynomial time if y is a solution to x.*

From Definitions 4.1.1, 4.1.5 and 4.1.6 we can draw the conclusion that if you would find a polynomial algorithm for any NP-complete problem this would mean that NP=P. For any optimisation problem in NPO the corresponding decision problem belongs to NP, see [1].

4.2 Reduction from the Sat-problem to the Lock Calculation Problem

You can reduce the sat-problem to a subproblem of the lock calculation problem.

Definition 4.2.1 *A formula in* conjunctive normal form *is a formula consisting of* clauses *separated by the Boolean and ($\wedge$) operator. A* clause *is a sequence of* literals *separated by the Boolean or ($\vee$) operator. A* literal *is a boolean variable or the negation of a boolean variable.*

Definition 4.2.2 *SAT is the following problem. Given a set of boolean variables $x_1 \dots x_y$ and a boolean formula in conjunctive normal form, is there an assignment that satisfies the formula the formula?*

Definition 4.2.3 *The Lock System Extension Problem (LSEP) is the following problem. Given the cylinder technology with n pins and only two different allowed cut depths for each pin-position. Given a set of cylinders C and a search space that corresponds to the entire cylinder technology. Is there a key in the search space that does not open any of the cylinders?*

This problem (LSEP) is one problem lock-system calculators are facing when they want to extend an existing lock-system: From the base of installed cylinders C_A, the user selects a subset C_B that the new key has to open. The set

C=C$_A$\C$_B$ is the set that the new key is not supposed to open. There is no problem in making the new key fit into C$_A$ because a lock-smith can always exchange the existing pins in these cylinders, adding more split pins that ensure that the new key will fit. We can not remove split pins from the set C because this would change the access of exisiting keys. Adding split pins makes no sense because the new key shall not fit into these cylinders. This is why this set must remain constant and is the only set that figures in the LSEP problem.

Theorem 4.2.1 *LSEP is NP-complete.*

Proof:
In order to prove that a problem is NP-complete you have to show two things. The first step is to show that the problem in question really is in the class NP. The class NP is defined in Definition 4.1.5. The second step is to find a polynomial reduction of a known NP-complete problem to an instance of our problem. If we manage to do this we have proved that our problem is NP-complete. This because given any problem in NP we can first reduce it (in polynomial time) to an instance of the known NP-complete problem (by definition of NP-completeness). We can then reduce this problem in polynomial time to an instance of our problem (according to the second step of our proof). This means that we can reduce any problem in NP to an instance of our problem which means that our problem is NP-complete.

The problem is in NP:
This problem is clearly in NP, since for any given key code $c_1, \ldots, c_n$, we can verify if the selected key opens any cylinder in C. This is done by checking if there is a cylinder that accepts all key cuts $c_1, \ldots, c_n$. Since this can be done in time proportional to $|C| \cdot n$, we see that the problem is in NP.

Reduction of the SAT problem to the problem in Definition 4.2.3:
Numerate the variables $1, ..., n$ by the order in which they are encountered in the CNF-formula. The number of variables is also the number of pins our lock system will have. Now we can construct the cylinder codes $R_1, \ldots, R_x$ from the clauses in the formula. Each clause represents a cylinder code. These cylinders are the cylinders in the set C. If a variable j is not represented in clause C_i then the corresponding cylinder will accept all key cuts for pin j (i.e. both key cut 0 and 1). If variable j occurs in C_i but is not negated: add key cut 0 as an allowed key cut for pin j in the corresponding cylinder. If the variable is negated: add 1 instead.
The search space for the key code is:

$$\underbrace{S \times \ldots \times S}_{n-times}, \qquad S = \{0, 1\}$$

This reduction can clearly be made in time linear in the size of the CNF-formula. The LSEP-problem obtained from this reduction can be solved if and only if there is a truth assignment for the CNF-formula, since a key that does not open any of the constructed cylinders will correspond to a truth assignement of the CNF-formula.

This means that the problem of extending an existing lock system is NP complete.

4.3 Reduction to the Induced Sub-graph Isomorphism Problem

In this section we will reduce the lock-system calculation problem to the induced sub-graph isomorphism problem. But first we start with some definitions.

Definition 4.3.1 *A bipartite graph kan be denoted $G = \{X, Y, E\}$. Here X and Y are sets of nodes, and E is the set of edges between set X and Y.*

Definition 4.3.2 *This is the induced sub graph problem for bipartite graphs: Given two bipartite graphs, $G = \{X_G, Y_G, E_G\}$ and $H = \{X_H, Y_H, E_H\}$. Find a mapping F: $G \to H$ such that two conditions are fulfilled:*

1. *$F(X_G) \subseteq X_H$ and $F(Y_G) \subseteq Y_H$.*
2. *G, and the graph induced by $F(G)$ are isomorphic. This means that $(F(x), F(y))$ is an edge in H if and only if (x, y) is an edge in G.*

In order to reduce the lock calculation problem to the induced sub-graph isomorphism problem, you represent the search space by a bipartite graph. All key codes and all cylinder codes within the search space will be nodes in this graph. A key and a cylinder will be connected by an edge if the key operates the cylinder in question.

The lock chart can also be represented by a bipartite graph in the same way as the search space. If you are able to find an isomorphism from the lock chart graph to a subgraph of the search space graph you have found a valid assignment for the lock system.

4.3.1 Automorphisms on the Search Space Graph

The search space graph has a lot of symmeteries and in this section we will show how you could explore these in the lock-system calculation problem.

Definition 4.3.3 *An automorphism $\mathfrak{a}$ on a graph G is a bijective mapping of the vertices of G onto themselves such that $(\mathfrak{a}(x), \mathfrak{a}(y))$ is an edge in G if and only if (x, y) is an edge in G.*

If you can define an automorphism on the search space you can reduce your search in the sub-graph isomorphism problem by considering the equivalence classes generated by this automorphism. If you find that one node in an equivalence class can not be assigned to a node in the lock-chart graph then you can conclude that the other nodes in this eqivalence class can not be assigned to this node either. The next theorem tells us how we can define all the automorphisms on the serach space.

Theorem 4.3.1 *Every automorphism on the search space graph $G = (X, Y, E)$, that maps $X \to X$ and $Y \to Y$, can be expressed by a combination of two different types of permutations. The first is a permutation of pin channels with an identical number of allowed key cuts. The second is a permutation of key cuts for each pin channel. Given that all key cuts are virtual key cuts: that means that the first key cut is always denoted 0, the second 1 and so on.*

Notation:
$G = (X, Y, E)$ = search space graph
X = key nodes of G
Y = cylinder nodes of G
E = edges in G
$\mathfrak{a}$ = arbitrary automorphism on G
$\mathfrak{a}^{-1}$ = inverse of $\mathfrak{a}$
$y \in Y,\ \mathfrak{a}(y) = y^{\mathfrak{a}},\ \mathfrak{a}^{-1}(y^{\mathfrak{a}}) = y$
$x \in X,\ \mathfrak{a}(x) = x^{\mathfrak{a}},\ \mathfrak{a}^{-1}(x^{\mathfrak{a}}) = x$
$c_1, c_2, \ldots, c_l$= number of key cuts in the search space represented by G
$c_1 > c_2 > \ldots > c_l$
$n_1, n_2, \ldots, n_l$= number of pins with $c_1, c_2, \ldots, c_l$ different key cuts
$N(x)$ = set of cylinder nodes connected to x by an edge
$N(y)$ = set of key nodes connected to y by an edge
$N(y)^{C_X} = X \setminus N(y)$
y_0 = the node that accepts all keys in the search space
$Y^{ONE} =< y_1, y_2, \ldots, y_r >$, ordered set of nodes in Y that accept all key cuts except for one. The order on Y^{ONE} is the degree of the node.
Nodes in Y^{ONE} will have degrees on the form:

$$c_1^{n_1} \cdot c_2^{n_2-1} \cdot (c_2 - 1) \cdot \ldots \cdot c_l^{n_l}, \quad c_1^{n_1} \cdot c_2^{n_2} \cdot c_3^{n_3-1} \cdot (c_3 - 1) \cdot \ldots \cdot c_l^{n_l}$$

Proof for Theorem 4.3.1:

Idea:
Take an arbitrary automorphism $\mathfrak{a}$ on G and identify the permutation of pin channels and key cuts that define it.

Nodes in Y^{ONE}can not be mapped to nodes outside Y^{ONE}:
$\mathfrak{a}(y_0) = y_0$ for every automorphism on G because all other nodes have lower degree than y_0.

$$\forall y \in Y^{ONE},\ N(y) \subseteq N(y') \Rightarrow y' = y \ or \ y' = y_0 \tag{4.1}$$

For every node

$$y'' \in (Y \setminus Y^{ONE} \setminus y_0) \quad \exists\, y \in Y^{ONE} \ such \ that \ N(y'') \subseteq N(y) \tag{4.2}$$

So by 4.1 and 4.2 we have that:
:

$$N(\mathfrak{a}(y)) \subseteq N(y') \ if \ and \ only \ if \ y' = \mathfrak{a}(y) \ or \ y' = y_0$$

since $\mathfrak{a}$ is an automorphism.

The mapping of the nodes in Y^{ONE} fixes a permutation of key cuts and pin channels. These permutations will not be violated by $\mathfrak{a}$:s mapping of nodes not belonging to Y^{ONE}:
Consider y_1 that has the degree: $c_1^{n_1-1} \cdot (c_1 - 1) \cdot c_2^{n_2} \cdot \ldots \cdot c_l^{n_l}$. Identify $N(y_1)^{C_X}$ and $N(y_1^{\mathfrak{a}})^{C_X}$. The keys in $N(y_1)^{C_X}$ all have exactly one key cut in common. Let us call this key cut k_{1,p_1}. Here p_1 denotes that key cut k_{1,p_1} is on pin p_1. Keys in $N(y_1^{\mathfrak{a}})^{C_X}$ also have one key cut in common; $k^{\mathfrak{a}}_{1,p_1^{\mathfrak{a}}}$. now we can fix a

permutation of pin channels and key cuts:

$$p_1 \rightarrow p_1^{\mathfrak{a}}$$

$$k_{1,p_1} \rightarrow k_{1,p_1^{\mathfrak{a}}}^{\mathfrak{a}}$$

This mapping will not be violated by $\mathfrak{a}$:s mapping of the rest of the nodes because the nodes in $N(y_1)^{C_X}$ can only be mapped on nodes in $N(y_1^{\mathfrak{a}})^{C_X}$

Take the next node y_2 in Y^{ONE}, if y_2 and y_1 have the forbidden key cut on the same pin, then $y_1^{\mathfrak{a}}$ and $y_2^{\mathfrak{a}}$ will also have the forbidden key cut on the same pin. This is realised by observing that since:

$$|N(y_1) \cap N(y_2)| \ = \ (c_1 - 2) \cdot c_1^{n_1 - 1} \cdot c_2^{n_2} \cdot \ldots \cdot c_l^{n_l}$$

if the forbidden key cut is on the same pin for both y_1 and y_2 and

$$|N(y_1) \cap N(y_2)| \ = \ (c_1 - 1) \cdot (c_1 - 1) \cdot c_1^{n_1 - 2} \cdot c_2^{n_2} \cdot \ldots \cdot c_l^{n_l}$$

if the forbidden key cuts are on different pins then the same must be valid for

$$|N(y_1^{\mathfrak{a}}) \cap N(y_2^{\mathfrak{a}})|$$

This implies that the pin channel permutation is not violated.

Since $|Y^{ONE}|$ = the number of key cuts in the lock system, and since the mapping of each lock node fixes the permutation of a different key cut, then all key cut and pin channel permutations will be fixed at the end of this iterative procedure.

Since $\mathfrak{a}$ was an arbitrary automorphism all automorphisms are decomposable in a unique combination of permutations of key cuts and pin channels.

It is trivial that any combination of these two different types of permutations is an automorphism.

The permutations on pin channels and cut depths will completely define the automorphism. We can se this by taking an arbitrary key code and applying the permutations of key cuts and pin positions on it. This will give us a new completely defined key code where all the cut depths on all pins are consistent with the automorphism function. Now take an arbitrary cylinder node. A cylinder node must be mapped onto another node with the same cardinality. Once we have applied the permutation rules to the original cylinder code we have obtained a new cylinder code with the same cardinality as the original cylinder where all the cut dephs on the different pins are consistent with the original automorphism. We have therfore completely defined the automorphism.

Definition 4.3.4 is a well known algebraic definition that you can find in [4].

Definition 4.3.4 *An equivalence relation $\Re$ is a relation on a nonempty set S that is:*

- *reflexive, $x \Re x$*

- *symmetric, if* $x \Re y$ *then* $y \Re x$
- *transitive, if* $x \Re y$ *and* $y \Re z$ *then* $x \Re z$

Theorem 4.3.2 *The group of automorphisms A on a graph induces an equivalence relation* $\Re_A$ *on the nodes in the graph. More specifically:*

$$x \ \Re_A \ y \ \textit{if there is an } a \in A \textit{ such that } a(x) = y$$

It is trivial that it is reflexive because the identity mapping is an automorphism. Furthermore it fulfils the symmetry property because every automorphism has an inverse function that is also an automorphism. Finally it is also transitive because any composition of automorphisms is also an automorphisms.

Definition 4.3.5 *Let V be the set of vertices in a graph G, A a group of automorphisms on G and* $\Re_A$ *the equivalence relation induced by A.* $\Re_A$ *yields a partition on V:*

$$\{v_1 \ \Re_A v_2 \mid \quad v_1, \ v_2 \in V\}$$

The cells of the partition are the equivalence classes of V induced by $\Re_A$*.*

It is obvious that all key codes belong to the same equivalence class in the initial search space graph. If you want to identify the search space nodes that induce the lock chart graph you can do this by performing an exhaustive search on the search space key nodes. This can be done by, for instance, the backtracking algorithm to be described in Section 5.3. To reduce the number of possible combinations you can use the information obtained from the automorphisms.

Theorem 4.3.3 *Let* $\mathscr{F}$ *be the set of nodes in the search space graph to which you have assigned lock chart nodes. Define an equivalence relation on the set of key nodes in the search space graph by the group of automorphisms* $A_{\mathscr{F}}$ *on G, that satisfies:*

$$\forall f \in A_{\mathscr{F}}, \ \forall v \in \mathscr{F}, \ f(v) = v$$

The number of unique assignments[1] *of the next key node is equal to the number of equivalence classes of key nodes, yielded by* $\Re_{A_{\mathscr{F}}}$*.*

This result follows immediately from the definition of equivalence classes and automorphisms.

A direct conclusion from this theorem is that it is unimportant which key code the first key obtains since all key nodes belong to the same equivalence class before any key has been assigned.

4.4 Automatic Assignment of Individual Keys

Let us assume that all the pins in a search space have the same number of allowed key cuts, c. One way to automatically generate individual key codes is:

Fix the number of pin channels for the individual keys that are going to have key cuts different from the grand master key[2]. Generate all possible key

[1] Let $\mathscr{C}$ be a set of key codes. The set consists of unique key codes if $\forall c_1, \ c_2 \in \mathscr{C}$ c_1 is not equivalent to c_2, with the equivalence relation $\Re_{\mathscr{F}}$. If the set of key codes to be tested on a key is unique then the assignments are also unique.

[2] The key that opens all cylinders in the lock system.

codes that have this property, these will be the individual key codes. You can not add any other key codes to this set and still have a set of individual keys. To maximise the number of individual key codes by this assignment you fix the number of pin channels that are going to be different from the grand master key by taking the maximum of 4.3

$$\max_{x \in 1,2,\ldots,n} \frac{(c-1)^x \cdot n!}{(n-x)! \cdot x!} \tag{4.3}$$

This formula can be generalised for a lock system with n pins, and $c_1, \ldots, c_j$ different allowed key cuts where p_i pins have c_i allowed key cuts. $i = 1, ..., j$. From equation 4.3 we have that we can pick C_i different partial key codes, where C_i is given by 4.4, provided that x_i key cuts should be different from the master key cuts and that we have c_i allowed key cuts to choose from (including the master key cut).

$$C_i = \frac{(c_i - 1)^{x_i} \cdot p_i!}{(p_i - x_i)! \cdot x_i!} \tag{4.4}$$

When $x_1, \ldots, x_j$ is fixed then the number of different key codes with x key cuts different from the master key is given by 4.5

$$C_1 \cdot \ldots \cdot C_j \tag{4.5}$$

If this equation is summed over all combinations of $x_1, \ldots, x_j$ whose sum is x, then we will have the number of single keys we can produce, who have x positions different from the master key. If we maximise this formula with $x \in 1, \ldots, n$, then we obtain the maximum number of single keys we can produce, provided that all single keys must have the same number of key cuts different from the master key. The equation that gives this number of single keys is written bellow:

$$\max_{x \in 1,2,\ldots,n} \sum_{x_1=\max(0,x-p_2-\ldots-p_j)}^{\min(x,p_1)} \sum_{x_2=\max(0,x-x_1-p_3-\ldots-p_j)}^{\min(x-x_1,p_2)} \cdots \sum_{x_j=\max(0,x-x_1-\ldots-x_{j-1})}^{\min(x-x_1-\ldots-\ x_{j-1},p_j)} C_1 \cdot \ldots \cdot C_j$$

In this equation we assumed that $\sum_{i=0}^{j} p_i \ \geq x$.

Chapter 5

Method Selection

5.1 Simulated Annealing

The simulated annealing method imitates the physical process of cooling molten materials down to the solid state. If the temperature is lowered at a sufficiently slow rate then the resulting metal will be in a low-energy state. The cooling down process can be described by transitions of particles from one energy state to another. If the transition of a particle from an energy level ϵ_i to ϵ_j is considered, and $\epsilon_j < \epsilon_i$, then the transition is always accepted. But if the transition of a particle from a lower energy level to a higher energy level is considered then the transition will be accepted with probability:

$$P(\epsilon_i, \epsilon_j, T) = e^{\frac{\epsilon_i - \epsilon_j}{k_B T}} \tag{5.1}$$

Here k_B is Boltzmann's constant and T the temperature of the system. From formula 5.1 we conclude that the probability of jumping from a lower energy state to a higher is non-zero and the probability of transition increases with the temperature of the system.
The generic algorithm will be given in Algorithm 5.1.1:

Algorithm 5.1.1
Notation: *T(t)= function that decreses the temperature. It takes the current temperature as an input.*
N(s)= A neigbour solution to the solution s. That is a $N(s)$ and s differ by only one key code.
m(x,s)= measure of the fitness of the solution
FROZEN = stopping predicate based on the difference between subsequet solutions
input *Problem instance x*
output *Solution s*
τ:= t, inital temperature
s:= initial feasible solution.
while *FROZEN $\neq$ false*
  for *l times* ***do***
    select any unvisited s' $\in$ N(s);
    if *m(x,s) < m(x,s')*

s:=s'
else
$\delta := m(x, s') - m(x, s)$
s:= s', with probability $e^{\frac{-\delta}{\tau}}$
$\tau := T\tau$)
return *s*

There are three steps that have to be performed efficiently in order to obtain a good simulated annealing algorithm:

1. Problem representation: This includes a representation of the solution space and an easily computable cost function. The cost function must be a good indicator of the quality of a given solution.

2. Transition mechanism between solutions: A transition mechanism should alter the solution as little as possible. Effects of the change must be calculated in time proportional to the size of the change (typically constant).

3. Cooling schedule: These parameters govern the decrement of the temperature of the system, the number of iterations between each change, acceptance criteria for a modified solutions and stop criteria for the algorithm.

The main problem if you were to implement a simulated annealing algorithm lies in steps one and two. A simulated annealing algorithm could start by assigning arbitrary key codes that are feasible, expanding the search space when necessary, and then focusing on trying to reduce this search space while maintaining a feasible solution. A transition could be the assignment of a new key code to a key. Here we have the first problem, because in order to find a suitable key code you have to use the trial and error method which could be time consuming. The next problem is to find an appropriate cost function. The simple minimal cylinder function found in definition 3.1.1 for all keys in the lock system is a very rough measure of the quality of the solution and will not distinguish between many solutions of different qualities. You could correct this measure by taking into consideration how close each key cut (except for the key cuts of the master key) is to be eliminated, in other words how many keys share the same key cut, favouring solutions that have key cuts close to being eliminated. This correction is not necessarily a good one because it favours keys that share many key cuts with the master key. These keys tend to eliminate many alternative key codes for other keys and should be avoided in order to obtain optimal solutions for problems of considerable sizes. Since there were no good intuitive ways to perform neither step one or two I estimated that the algorithm would not work optimally if these steps did not have good solutions and that another approach maybe would be more efficient. I therefore did not spend very much time in search for ways to implement these steps. If you could find good solutions to these problems then simulated annealing would be a good candidate algorithm since it has performed very well on several problems, see [9] for further details.

5.2 Genetic Algorithms

Genetic algorithms are inspired by evolution and natural selection. You maintain a population of solution candidates. New solutions are created by letting

two arbitrary solutions reproduce by combining some aspects. The probability that a solutions is chosen depends on the fitness of this solution. This approach was eliminated mainly because of three reasons:

1. Historically they have not obtained as good results as for instance simulated annealing. See [9] for further details about the reasons for this.
2. There is no intuitive way to combine two solutions.
3. The intuitive cost function in Definition 3.1.1 does not differentiate between many different solutions, it is thus a rather poor judge of the quality of a current solution.

5.3 Backtracking with Pruning

A backtracking algorithm is an algorithm that tries all configurations of a certain search space in a systematic way. In our case it means trying all possible combinations of key codes and testing if these key code assignments satisfy the lock system matrix. The solution is constructed step by step by assigning one key code at the time. As long as the solution is not complete we try to extend it. When there is no possible extension of the current partial solution we backtrack and replace the last assigned item. Backtracking is equivalent to a depth first search on the search tree where each node is a partial solution. The children of a node represent the same solution except for one key code.

We can prune[1] nodes corresponding to partial solutions that do not fulfil the input lock chart requirements.

A heuristic that further reduces the size of the search consists of invoking a measure of how good a partial solution is, and cutting of the branches of solutions that are indicated as poor by the chosen measure. This measure is preferably easy to calculate. This method is used to avoid exploring subtrees of the search tree that are unlikely to result in a final solution.

The obvious drawback of the backtracking algorithm is that the execution time has an exponential growth with the size of the problem. The search space can however be reduced by exploiting symmetries of the problem and pruning solutions. This was the type of algorithm that finally was chosen for a test implementation. The main advantage of the algorithm is the natural way in which you can explore symmetries in the graph (see Section 4.3) and the measure in Definition 3.1.1 that can be used for pruning since it gives an indication of the quality of the solution.

[1] To prune a node means that you do not explore its children

Chapter 6

Implementation

Here we present the three different development phases of the backtracking algorithm. The first implementation is the naive algorithm that simply explores all allowed combinations of key code assignments. The second algorithm uses symmetries of the search space graph to prune the search tree. The third and final algorithm uses the minimal cylinder measure to prune partial solutions that do not look promising.

6.1 The Naive Algorithm

This algorithm takes a search space and a lock chart as input and then it performs an exhaustive, depth first search to find suitable key codes. This first naive implementation tries all combinations of key codes that fulfil the lock chart requirements.

A depth first search has the advantage of not requiring a lot of memory allocation compared to a breadth first search. In this implementation of the naive algorithm the key codes are tested in a predefined order, this means that you only have to store the currently assigned key code for every key. One version of cylinder codes is stored in order to speed up the checking if the newly assigned key code is consistent with the target structure. Algorithm 6.1.1 contains the pseudo-code for the naive algorithm.

Algorithm 6.1.1

Notation:
S = the search space of the algorithm, defined as all keys that open a specific cylinder
s_x = key code x in S
k_x = key x in the lock system
s_0 = master key code.

INPUT*(Lock chart L, Search space S)*
$k_0 := s_0$
Make necessary variable updates
$i := 1$

$k_1 := s_0$
while(*there are unassigned keys left AND* $\neg$(*i = 0 AND* $k_0 = s_{|S|}$))
  while(*there are unassigned keys left AND next*(k_i) *produces a valid key code*)
    $k_i := next(k_i)$
    increase i
    make necessary updates
  if(*there are unassigned keys left*)
    unassign key k_i
    decrease i
  if(s_{last} *is assigned*)
    return$(L, k_1, \ldots, k_{last})$

return(*impossible to solve lock system*)

The next() function takes the current key code as an argument. It returns the next key code in the search space that fulfils the requirements of the lock chart.

To check if a new assignment of key k_i is valid, two steps are performed. If the key is not accepted a third step is carried out.

1. We add the key cuts of the newly assigned key to the set of accepted key cuts for cylinders that are supposed to be opened by the key.
2. We check if the new key does not open any cylinders that it is not supposed to open according to the lock chart.
3. If the key does not pass step 2, we reverse the changes of the cylinder codes made in step 1.

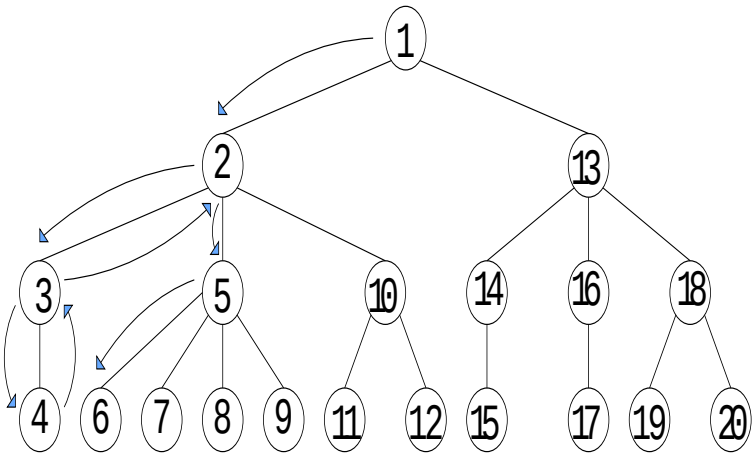


Figure 6.1: Depth first search

Figure 6.1 shows how a search tree is traversed by the naive algorithm. The numbers in the nodes are the order in which they are encountered by the algorithm. When you descend the tree you assign the key code of the node to the next key. This means that node 1 represents the assignment of the master

key code, node 2 and 13 are key codes for key 2 that are compatible with key code 1. Node 3, 5, and 10 represent key codes for key 3 that are compatible with the assignments in node 1 and 2 and so on.

The search space for this naive algorithm grows rapidly. For example, an exhaustive search for a lock chart with 5 keys and a search space of 300 keys would have to explore 300*299*298*297*296 $\approx$ 2 349 940 million possibilities! Many of these combinations are not explored by the algorithm because they do not fulfil the lock chart requirements. Since this is a problem where you have to choose x elements among y possible you have $O((y!/((y-x)!))$ different combinations. This algorithm was only able to solve rather small problems (See Chapter 7).

6.2 The Search Space Automorphism Algorithm

The key code symmetry algorithm works as the naive algorithm, with one exception: it uses the information obtained from automorphisms on the search space graph that is explained in Section 2.3. According to Theorem 4.3.1 all automorphisms on the search space graph can be represented by two different types of permutations (permutations of pin channels and cut depths). You have to identify the equivalence classes of key code nodes from the equivalence relation induced by the group of automorphisms that keep the set of previously assigned keys fixed. The search space automorphism algorithm chooses one key code from each equivalence class and cuts the branches corresponding to the other key codes in that equivalence class. The cut of one branch eliminates an entire subtree.

6.2.1 Identification of Equivalence Classes

Before any node is fixed, all key code nodes belong to the same equivalence class. All permutations of cut depths are allowed, and all permutations of pin channels with the same number of different cut depths are allowed. This can be represented in a compact way by the following notation:

1	2	3	4	5	6
0	0	0	0	0	0
1	1	1	1	1	1
2	2	2			
3					

Table 6.1: Automorphisms on the search space

The notation in Table 6.1 represents a cylinder technology with 6 pin channels and a search space where 2 different cut depths are allowed for pins 4-6, 3 different cut depths for pins 2 and 3 and finally 4 different cut depths for pin number 1. Vertical lines delimit the different pin channel permutation groups and horizontal lines delimit the cut depth permutation groups. In the example above, pin channels 4-6, and 2-3 can be permuted. There are no restrictions on allowed permutations for cut depths for each specific pin channel.

Select key code (0,0,0,0,0,0) for the first key in the lock system. Once this key code has been set we have to find the group of automorphisms that keep (0,0,0,0,0,0) fixed. This group of automorphisms has restrictions on permutations of cut depths, cut depth 0 must always be mapped to itself. This group of automorphisms is indicated in Table 6.2. If key code (0,0,0,1,0,0) is assigned to

1	2	3	4	5	6
0	0	0	0	0	0
1	1	1	1	1	1
2	2	2			
3					

Table 6.2: Automorphisms that keep (0,0,0,0,0,0) fixed

the second key in the lock system then the equivalence relation will be the set of automorphisms on the search space that keep both (0,0,0,0,0,0) and (0,0,0,1,0,0) fixed. In order to keep the two vertices fixed pin channel 4 must always be mapped to itself. For instance say that pin channel 4 would be mapped to pin channel 5 and (0,0,0,1,0,0) would still be mapped to itself. Then key cut 1 in pin channel 4 would have to be mapped to key cut 0. This is impossible since key cut 0 is always mapped to itself and the mapping is a permutation. The automorphisms that respect these rules map the two vertices onto themselves.
When the third key obtains key code (0,1,1,1,0,0) the equivalence relation must

1	2	3	4	5	6
0	0	0	0	0	0
1	1	1	1	1	1
2	2	2			
3					

Table 6.3: Automorphisms that keep (0,0,0,0,0,0) and (0,0,0,1,0,0) fixed

keep this key code fixed as well. In terms of permutations this means that key cut 1 must be kept fixed on pin 2 and 3. See Table 6.4

1	2	3	4	5	6
0	0	0	0	0	0
1	1	1	1	1	1
2	2	2			
3					

Table 6.4: Automorphisms that keep (0,0,0,0,0,0), (0,0,0,1,0,0) and (0,1,1,1,0,0) fixed

The update rules for the group of permutations on pin channels and key cuts are:

Algorithm 6.2.1

Notation:
Equivalence classes of pin channels: $P_1 \dots P_i$
Equivalence classes of key cuts for pin channel l: $K_{1,l} \dots K_{j,l}$
Newly assigned key code: $k_1 \dots k_n$

for *(l = 1, ..., i:)*
S = set of newly assigned key cuts corresponding to pin channels in the equivalence class P_l*.*
for*(x = 1, ... j)*
identify the key cuts both in S and $K_{x,l}$*. The pin channels corresponding to these key cuts will constitute a new equivalence class of pins.*

If *(a newly assigned key cut* k_r *belongs to an equivalence class* $K_{l,r}$ *with more than one key cut.)*
Split this equivalence class into two equivalence classes one containing k_r *and the other* $K_{l,r} \setminus k_r$*.*

Once the equivalence classes of pin channels and key cuts have been updated, we can enumerate the unique key codes. If we have $p_r,\ r = 1, \dots, |P|$ number of pins in equivalence class P_i of pins and equivalence class P_j of pin channels has c_j equivalence classes of key cuts. (Here j denotes to which equivalence class of pin channels the equivalence classes of key cuts correspond.) Then we have:

$$C_1 \cdot \ldots \cdot C_{|P|} \tag{6.1}$$

unique key codes. Here

$$C_i = \frac{(p_i + c_i - 1)!}{c_i! \cdot (p_i - 1)!} \tag{6.2}$$

Formula 6.2 is a well known formula that we can find in [8] that tells us in how many ways we can take c_i elements from a population of p_i elements without regard to order and with replacement. The problem of how many unique partial key codes on an equivalence class of pin channels we can select is equivalent to this problem. We can see that the two problems are equivalent because we can obtain the number of pins that have key cut $1, 2, \dots, c_{|P|}$ by the following procedure:
K_i= number of key cuts that have cut depth i.

- Order the numbers you have picked, $n_1, \dots, n_{c_{|P|}}$
- for $i = 1, \dots, c_{|P|} - 1 \quad K_i = n_{i+1} - n_i$
- $K_{c_{|P|}} = p_i - n_{c_i}$

Now it is obvious that:

- $\sum_{i=0}^{c_{|P|}} K_i = p_i$ for all sequences $n_1, \dots, n_{c_{|P|}}$
- Two different sequences $n_1, \dots, n_{c_{|P|}}$ and $n'_1, \dots, n'_{c_{|P|}}$ will not produce identical sequences of $K_{c_{|P|}} = p_i - n_{c_i}$

The pseudo code for the search space automorphism algorithm:

Algorithm 6.2.2

Notation:
S_{auto} = The search space of the algorithm, represented in the compact notation found in Table 6.1.
s_x = key code x in S_{auto}
k_x = key x in the lock system
s_0 = master key code.
next$_{auto}(k, S_{k_1,\ldots,k_i})$ *= next function of the automorphism algorithm that keeps key codes $k_1, \ldots, k_i$ fixed.*

INPUT*(Lock chart L, Search space S_{auto})*
$k_0 := s_0$
Run algorithm 6.2.1 to update the automorphism structure
Make necessary variable updates
$i := 1$
$k_1 := s_0$
while*(there are unassigned keys left AND $\neg$(i = 0 AND $next_{auto}(k_i, S_{auto}) ==$ null))*
    while*(there are unassigned keys left AND $next_{auto}(k_i, S_{k_1,\ldots,k_{i-1}})$ produces a valid key code)*
        $k_i := next_{auto}(k_i, S_{k_1,\ldots,k_{i-1}})$
        increase i
        make necessary updates
        Run Algorithm 6.2.1
    if*(there are unassigned keys left)*
        unassign key k_i
        decrease i
    if*(s_{last} is assigned)*
        return*$(L, k_1, \ldots, k_{last})$*

return*(Could not solve lock system)*

Automorphisms in the target structure are more complicated to identify, and are not considered by the algorithm. If the target structure is very irregular then the automorphisms should be few.

6.3 The Final Algorithm

Often the search space will be too big even after the elimination of symmetric nodes. By examining the most promising solutions first we could increase the probability of finding a solution within a reasonable time, if there is such a solution for the given search space. In order to do so you must have a measure of how good the current partial solution is. The final algorithm is an attempt to do this. The core of this algorithm is the automorphism algorithm. The addition to the final algorithm is that the fitness of every partial solution is estimated and the partial solution that has the most promising measure is explored first. We use the following measure of the fitness of the solution.

Algorithm 6.3.1

Notation:
$L_{K+1,d}$ = *The lock-chart consisting of key K+1 - K+d+1. (in other words lines K+1 - K+d+1 of the lock-chart).*
$\textbf{\textit{next}}_{auto}(k, S_{c_1,\ldots,c_K})$ = *next function of the automorphism algorithm that keeps key codes* $c_1, \ldots, c_K$ *fixed.*
$c_1, \ldots, c_K$ = *previously assigned key codes*
c_{K+1} *assigned to key K+1 = assignment whose fitness is going to be estimated.*
$S_{c_1 \ldots c_K}$ = *Automporphism search space that keeps key code* $c_1, \ldots, c_K$ *fixed.*

INPUT$(L_{K+1,K+1+d}, S_{c_1 \ldots c_K}, c_1, \ldots, c_{K+1})$
Assign $k_{K+1} = c_{K+1}$
Update the automorphism structure by algorithm 6.2.1.
$i := K+2$
$k_{K+2} \ldots k_{K+1+d} = null$
$LOW = null$
while$(\neg(i == K+2$ *AND* $next_{auto}(k_i, S_{c_1,\ldots,c_{K+1}}) == null)$ *)*
    while*(there are unassigned keys left AND* $next_{auto}(k_i, S_{c_1,\ldots,c_{K+1},\ldots k_{i-1}}) ==$ *null != null)*
        $k_i := next_{auto}(k_i, S_{c_1,\ldots,c_{K+1},\ldots k_{i-1}})$
        increase i
        Run algorithm 6.2.1 *on* $S_{c_1 \ldots c_K \ldots c_i}$

    if*(there are unassigned keys left)*
        $k_i == null$
        decrease i
    else
        $LOW = min(LOW, MinCyl(k_1, \ldots k_{K+1+d}))$
        decrese i
return *LOW*

This measure clearly has the property stated above if d does not decrease faster than one step at a time, since the number of consumed key codes can not decrease when more key codes are assigned. The smaller the measure the better is the solution. The algorithm cuts of branches[1] representing partial solutions that are judged not sufficiently good with regard to the number of consumed key codes.

Figure 6.2 demonstrates an execution of the final algorithm on a search tree. Besides the target structure and code resources the algorithm has got the backtracking depths for the measure function which is 2, 2, 1, 1 for this execution. The numbers beside the circles are the consumed key codes once the code corresponding to the node has been assigned. The numbers inside the circles are the order in which the nodes are encountered. Once the master key 1 has been assigned the algorithm explores nodes numbered 2-8 in order to find out if assignment of node 2 or 5 is most fit. Since node 2 will have measure 6 and node 5 will have measure 8 (we have descended to the first horizontal line and returned the smallest value on each subtree to node 2 and 5) we will chose node

[1]This means that the algorithm does not explore the search tree corresponding to the branch that has been cut of.

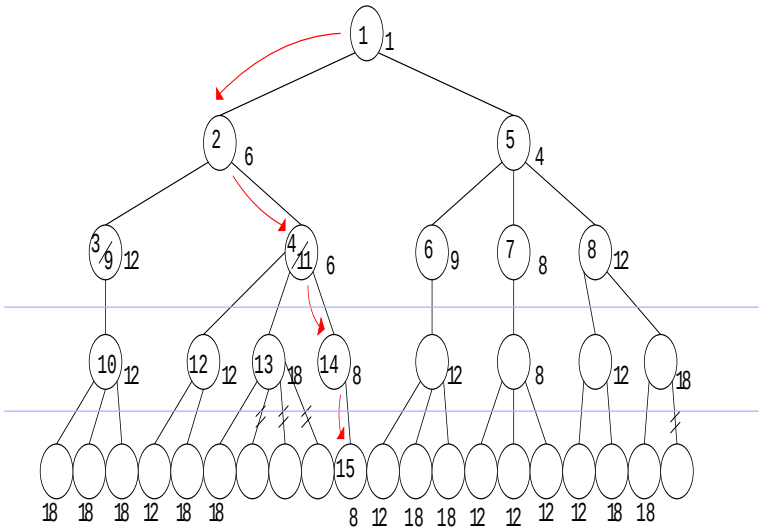


Figure 6.2: The final algorithm on a search tree with input 2, 2, 1, 1

2. We continue to explore the two subtrees two key codes ahead to determine the measure of the solution. Node 3/9 will have measure 12 and node 4/11 will have measure 8 so we chose node 4/11. Now we only look one step ahead and find that node 14 has the best measure and finally we only have one node to descend to which is node 15. If the algorithm would not have found a solution by descending to node 14, that is if node 15 would not have been allowed, it would have backtracked to node 4/11 and then continue to descend to node 12.

The pseudo-code for the final algorithm will be:

Algorithm 6.3.2

Notation:
S_{auto} = *The search space of the algorithm, represented in the compact notation found in Table 6.1.*
$k_x =$ *key x in the lock system*
$s_0 =$ *master key code.*
$next_{final}(k_i, S_{k_1,\ldots,k_{i-1}}, d)$ = *Function that returns the next unique, (with regard to automorphisms) key code that has the lowest measure with regard to algorithm 6.3.1. This function ignores key codes whose measure is bigger than twice the lowest measure at the current node.*

INPUT*(Lock chart L, Search space* $S_{auto}, d = \{4, 4, 4, 3, 2, 1, \ldots, 0\}$*)*
$k_0 := s_0$
Run algorithm 6.2.1 to update the automorphism structure
Make necessary variable updates
$i := 1$

$k_1 := s_0$
while(*there are unassigned keys left AND* $\neg$(*i = 0 AND* $next_{final}(k_0, S_{auto}, d_0)$)
)
    while(*there are unassigned keys left AND* $next_{final}(k_i, S_{k_0,\ldots,k_{i-1}}, d_i)$ *produces a valid key code)*
        $k_i := next_{final}(k_i, S_{k_0,\ldots,k_{i-1}}, d_i)$
        increase i
        make necessary updates
        Run Algorithm 6.2.1
    if(*there are unassigned keys left*)
        unassign key k_i
        decrease i
    if(s_{last} *is assigned*)
        return$(L, k_1, \ldots, k_{last})$

return(*Could not solve lock system)*

The values of d was fixed to $d = \{4, 4, 4, 3, 2, 1, \ldots, 0\}$ because this value performed experimentally well on all test lock-charts. If you wanted to further increase the performance you could adapt the values of d to each problem.

Chapter 7

Evaluation and Results

In order to measure the performance of the different algorithms they were tested on some lock charts that IKON, a German lock system manufacturer part of the Assa Abloy group, had calculated and produced. All these problems were chosen by Gudrun Brennecke as representative of the unstructured problems encountered by lock system calculators at IKON.

The lock charts have been divided into three groups. The easy problems are lock charts where it is possible to find optimal solutions. Medium hard problems are lock charts where the automorphism algorithm has found solutions using up to 600 key codes. Hard problems are lock charts where more than 600 key codes have been consumed by the automorphism algorithm.

The figures in this chapter present two types of solutions found by the algorithms, *Backtracking* and *Combined Backtracking and automatic assignment of individual keys*. The *Backtracking* solutions are solutions where the entire lock chart has been solved by the algorithm. *Combined Backtracking and automatic assignment of individual keys* refer to solutions where the master key codes have been found by the algorithms and the individual keys assigned using the procedure described in Section 3.4.

When benchmarking the implemented algorithms one must take some things into consideration: Lock charts of identical sizes are not necessarily equally difficult to solve. The hardness of the lock chart does not depend on its regularity. Lock charts that are easy to calculate manually may be difficult for the algorithm to solve and vice versa. The measure *consumed key codes* depends on how the allowed key cuts are distributed on the different pins. This means that we must choose the search space in the same way in order to be able to compare the performance of the different algorithms and the manually found solutions.

The benchmarking for the automorphism and final algorithm were performed on a Pentium 4, 1.7Ghz laptop. The execution time was limited to 2 hours. This time-limit was chosen because the order administration time for small lock-systems is limited to 3 hours for the quickest deliver times within the ASSA ABLOY group. Two of these three hours can be dedicated to lock-system calculation.

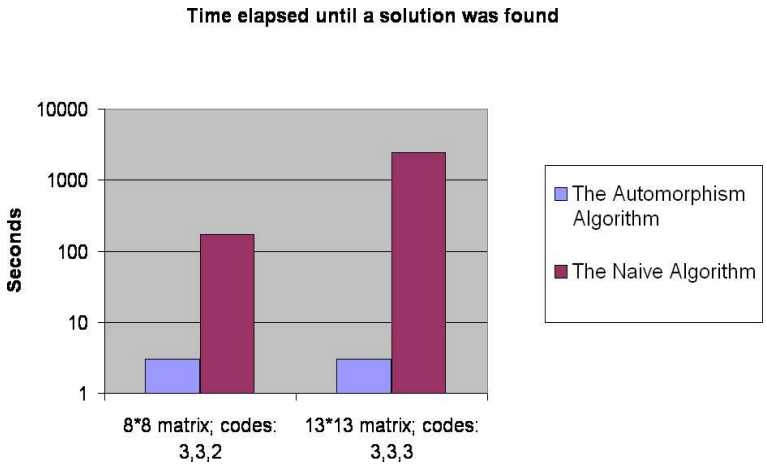


Figure 7.1: Comparison between the naive and the automorphism algorithm. The program ran on a Unix Ultra5

7.1 The Naive Algorithm

On average the naive algorithm managed to find solutions for problems of 5-8 keys within a couple of minutes, depending on the complexity of the lock chart. As the size of the lock chart grows the time until a solution is found grows rapidly. Figure 7.1 shows the performance of the naive and the automorphism algorithm for two lock charts one with 8 keys and 8 locks and the other with 13 keys and 13 locks. As you can see the time until a solution is found grows rapidly with the size of the problem for the naive algorithm.

7.2 The Automorphism Algorithm

The automorphism algorithm was benchmarked for several problems, easy, medium hard and hard problems. You can find their corresponding lock charts in Appendix A, Appendix B and Appendix C. As Figure 7.2 shows the results are very good for small problems. Often it was possible to finish the exhaustive search, which means that the obtained results are optimal. The columns representing IKON solutions are included as a reference of the amount of key codes normally consumed for these types of problems. They are not comparable to the solutions found by the algorithm because of two reasons:

- The IKON solutions include several profiles while the solutions found by the algorithm do not include different profiles. The column named IKON pin codes are the pin codes consumed by IKON in order to solve the problem. The column named IKON pin codes*profiles displays the total number of consumed pins codes because it includes the consumed profiles as well.

- The manufacturing restrictions are checked for the IKON solutions. These are not checked for the solutions found by the algorithm. A very small

percentage of combinations are eliminatied by these manufacturing restrictions,

IKONS extensive use of profiles is something rather unusal for lock system manufacturers. Most companies use only one profile for master key systems. The found solutions are thus interesting from a practical point of view. The total amount of unique key codes in one cylinder thechnology are comparable regardles of weather the codes are created from profiles or not. For instance IKON had approximately 4 096 000 unique key codes, including profiles, for one of their old cylinder technologies. ASSA (company that use only one profile for master key systems) had 4 782 969 unique key codes for their cylinder technology at the same time. It is impossible to say wether these two technologies are comparable by only looking at the number of unique keys. But these figures tell us that the comparisons are not completly irrelevant since the two technologies existed simultaniously.

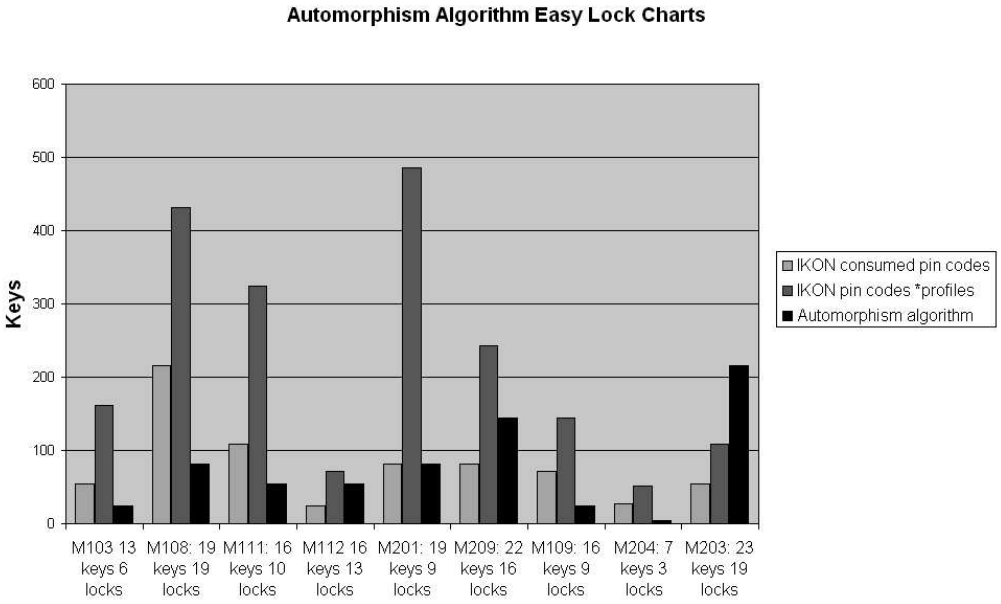


Figure 7.2: The automorphism algorithm on Easy lock charts

For medium sized problems the results are comparable to the manually found solutions. Often the medium sized problems are solved by dividing the problem into two subproblems, one consisting of individual keys and the other of all other keys. You allocate two separate pieces of code resources for these problems. The individual keys are assigned in an automatic way which consumes a minimal amount of code resources. The procedure of this assignment is found in Section 3.4. The second *master key problem* is solved by the automorphism algorithm.

The algorithm fails to find satisfactory solutions for hard problems, as you can see in Figure 7.4. Satisfactory solutions refer to solutions whose number of consumed key codes are comparable to the manually found solutions.

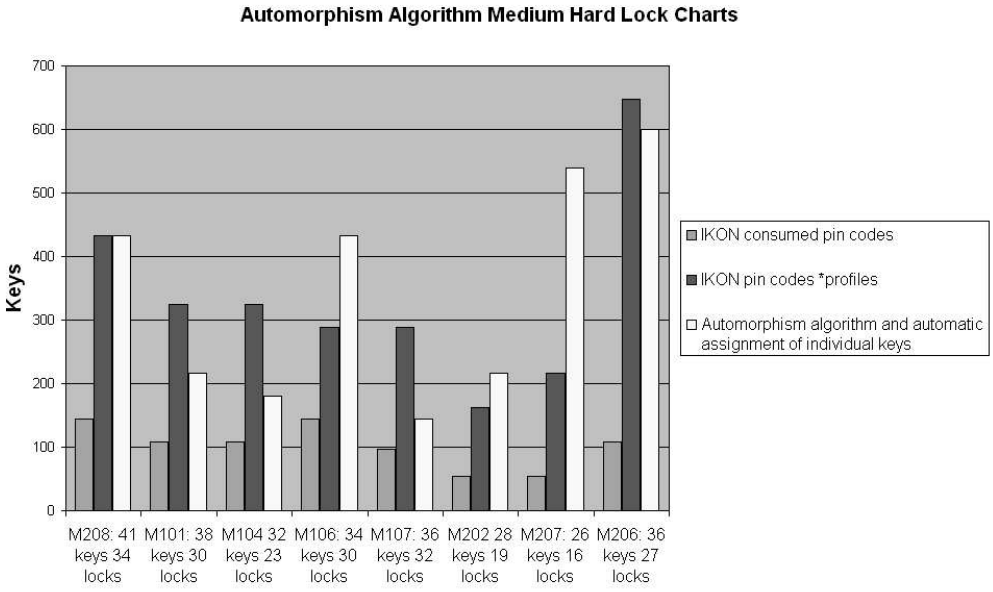


Figure 7.3: The automorphism algorithm on Medium hard lock systems

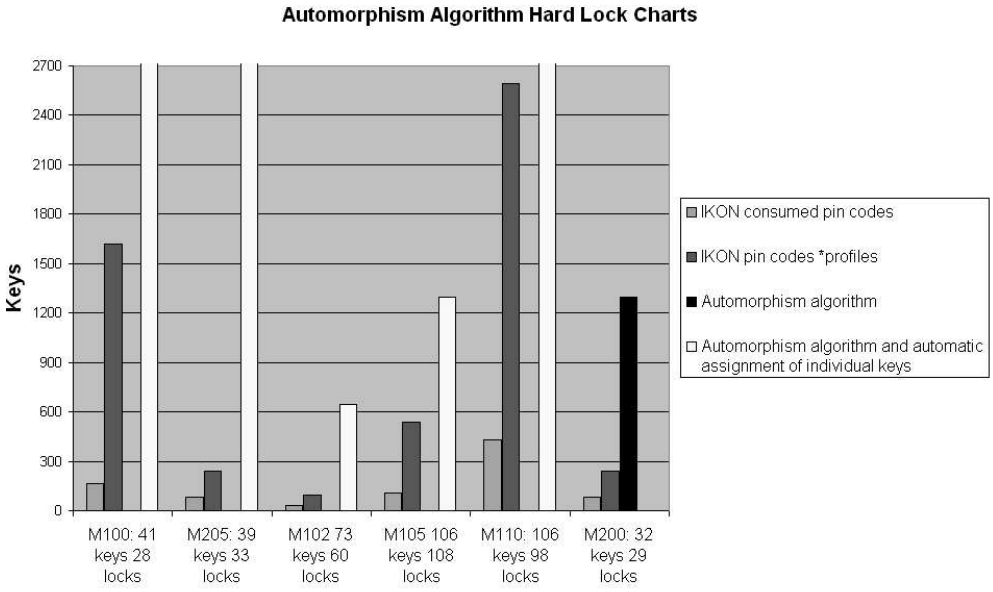


Figure 7.4: The automorphism algorithm on Hard lock systems

7.3 The Final Algorithm

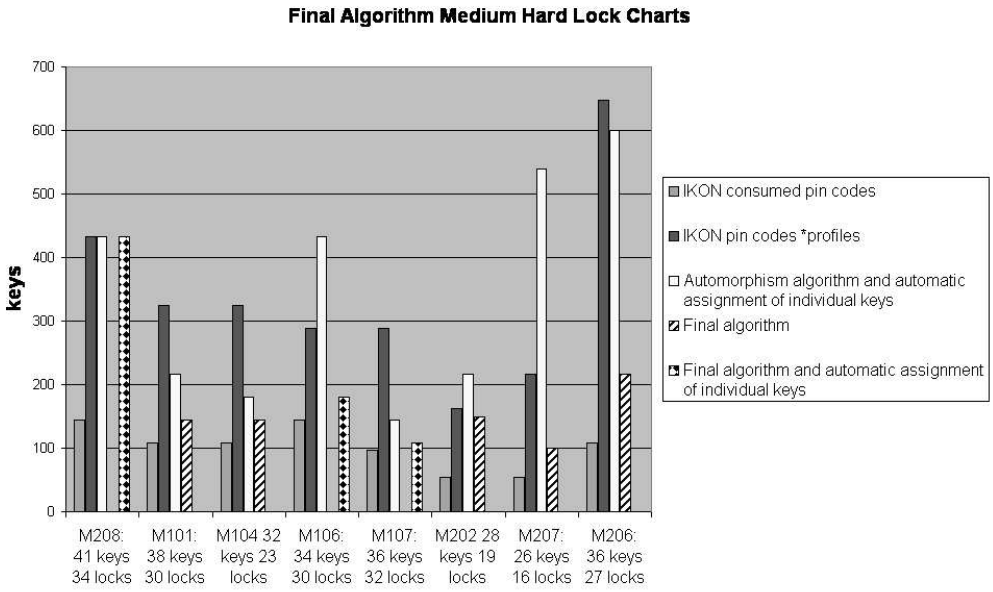


Figure 7.5: The final algorithm on Medium hard lock systems

The final versions performance depends on an appropriate choice of search depth and threshold value for pruning. If those variables are set correctly then the algorithm will outperform the automorphism algorithm for all benchmarking problems as you can see in Figure 7.5 and 7.6. It does as well as the automorphism algorithm for two problems. A very important difference between the final algorithm and the automorphism algorithm that is not directly seen in the diagram is the fact that the final algorithm obtains better solutions than could be obtained by dividing the problem into a master key and an individual key problem for many medium sized and large problems. The largest problem that has been solved by the final algorithm in a way that is comparable to solutions obtained by dividing the problem is the lock chart M102 with 72 keys and 60 locks. Solving problems of this size directly, in a satisfactory way, was not possible with the automorphism algorithm.

The program obtains better solutions than manually found solutions when lock charts are unstructured. Among the problems used for benchmarking M205, M102 and M105 are relatively structured. As you can see these problems are also the ones where the results found where worse than manual solutions. If you compare the result obtained for M105 with the result obtained for M110 that has the same number of keys then you see that the solution found by the program is compatible with the manual solution for M110 whereas the solution for M105 is worse than the manual solution. M110 has 15 keys on master level that are very unstructured whereas M105 only has 11 unstructured master keys. The generally poor performance of the automorphism algorithm compared to the final algorithm depends on that the automorphism algorithm does not find

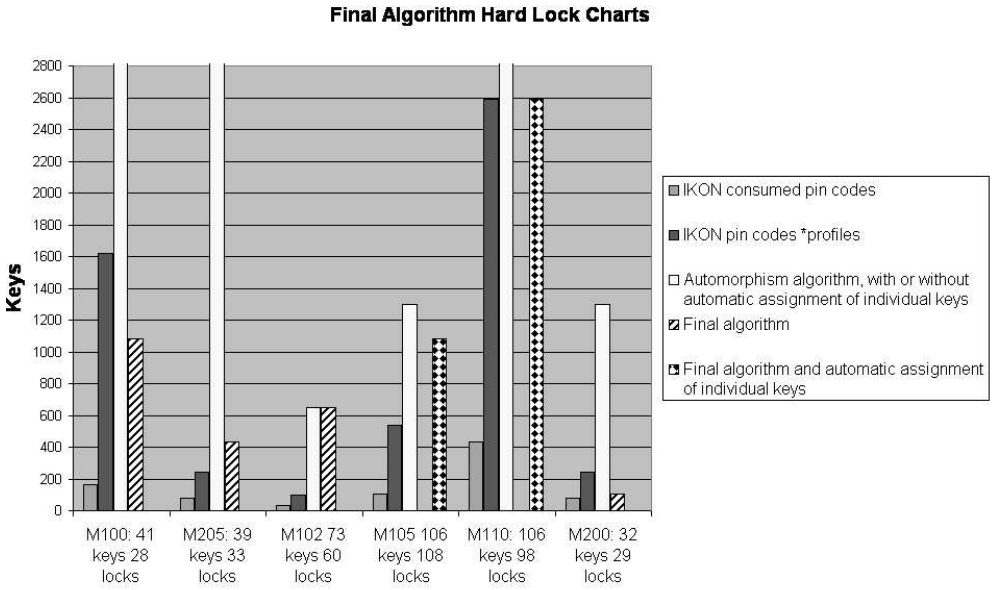


Figure 7.6: The final algorithm on Hard lock systems

solutions within a reasonable time (= two hours), when the search space grows smaller. In order to find a solution we must extend the search space.

Chapter 8

Conclusion and Further Work

8.1 Conclusion

The final algorithm reduces the number of consumed key codes substantially compared to manually found solutions for small and medium sized unstructured problems. Even if the algorithmic method is not directly comparable to manual calculations we can se that in general the final algorithm consumes much less key codes that the manual solutions. This indicates that these key codes could be used to fullfill the manufacturing restrictions if they are not already fullfilled. The differances in search space are sometimes large (several profiles are used in the manual calculations) but many times they are small or none (only 1-3 defferent key profiles are used). For hard relatively unstructured problems the solutions of the final algorithm are comparable to manual solutions, here we do not have much flexibility to alter the solution if it does not fullfill manufacturing restrictions. When the problem is relatively hard, structured manual solutions are generally better than the solutions produced by the final algorithm.

8.2 Further Work

8.2.1 Improvements on the Pruning Algorithm

There are several improvements that can be made on different parts of the final algorithm. I believe that these improvements could increase the performance of the algorithm substantially:

- **Automatic assignment of input variables:** In the current final implementation of the backtracking algorithm you have to give the backtracking depths for the different keys as an input to the algorithm. If the backtracking depth is too deep the algorithm will not finish within a reasonable time, and you have to interrupt it and execute it again with different backtracking depths. If a maximum runtime is given as an input to the program it could be made to assign a certain search time for each key. During the time assigned to each key the program would evaluate the

valid key codes and assign the best current code once the time run out. The evaluation could be, as in the current implementation, a depth first search with a measure function.

- **Improvement on the measure function:** As the size of the partial solution grows the measure increases until it reaches the maximum number of allowed keys. After reaching this number, the choice between the different solutions becomes arbitrary. The choice between two keys that *consume* the same amount of keys is arbitrary. A measure function that could distinguish between the quality of these soultions would improve the performance of the algorithm substantially. This would be particularly valuable for large problems where lot of different key assignments can be made even when the maximum number of allowed key cuts are reached.

- **Elimination of keys similar to the master key:** Keys assigned early in the backtracking procedure tend to be keys that have few positions different from the master key. This is because the algorithm strives to minimise the consumed key codes. These codes could be a restriction on further key assignments because the keys will fit in many locks. To avoid this you could update master key codes regularly and change the key code for a code where the number of positions different from the master key are maximised in the existing lock system.

- **Exploring lock chart automorphisms:** Lock chart automorphisms could be explored much in the same way as search space automorphisms are explored. If the lock chart is very irregular the number of automorphisms will be few and the speedup will be marginal but if the lock chart is regular, as is the case for structured problems, the number of automorphisms will be very important. There is already a program called Nauty that can determine these automorphisms efficiently for graphs the size of lock charts. It is hard to tell how large graphs the program can handle because the hardness of the problem depends on the regularity of the graphs. In (*Practical graph isomorphism. 1981 [6]*) examples of efficient solutions for random graphs with up to 1000 nodes are given. See *Nauty Users's Guide [7], Practical graph isomorphism [6] and The Graph Isomorphism Problem, [3]* for further details. This could be the solution that would make the program output compatible with manual solutions for structured problems as M205, M105, M102.

- **Cooperation with lock system calculators to assign the first master key codes:** Lock system calculators have developed a systematic way to assign the first master key codes. The principles that they use when assigning those key codes could be formalised and used by the program to automatically assign the first master key codes.

- **Measure limits for large problems:** The problem for large lock systems is that the program does not backtrack to the initially assigned key codes within a reasonable time because the problem is too large even when automorphisms are eliminated and pruning is performed on solutions estimated as not promising by the measure function.

- **Implementation of automatic individual key code assignment:** If the procedure of assigning individual key codes was implemented then the search space used to assign individual keys could be reused when assigning master keys.
- **Automatic verification of manufacturing constraints:** Once a solution to the lock chart has been found, one task remains to be performed. Since the output of the program is a lock system specification in virtual key cuts we can map these key codes in more than one way. By *virtual key cuts* I mean that that the first key cut on every pin channel is represented by a 1 the second by a 2 and so on. A program that enumerates all these combinations looking for a lock system that fulfils the manufacturing restrictions as well as the lock chart requirements would be another useful tool for the lock system calculator.

8.2.2 Implementation of Profile Assignment

Today different profiles are often used when manufacturing a lock system. The assignment of different profiles to different keys could be used to break the lock chart down to smaller partial lock-charts.

Bibliography

[1] G. Ausiello, P.Crescenzi, G.Gambosi, V.Kann, A.Marchetti-Spaccemela M.Protasi. 1999: *Complexity and Approximation, Combinatorial Optimization Problems and Their Approximability Properties.* Springer. ISBN 3-540-65431-3

[2] S. Baase, Allen Van Gelder 2000 *Computer Algorithms.* Addison Wesley. ISBN 0-201-61244-5

[3] S. Fortin 1996 *The Graph Isomorphism Problem* Department of Computing Science, The University of Alberta

[4] J. Fraleigh, 1998: *Abstract Algebra*, Addison Wesley Longman. ISBN 0-201-33596-4

[5] G. Luger & William A Stubblefield. 1997: *Artificial Intelligence, Structures and Strategies for Complex Problem Solving.* Addison-Wesley. ISBN 0-805-31196-3

[6] B. McKay. *Practical graph isomorphism.* Congressus Numerantium, 30:45-87 1981

[7] B. McKay *Nauty Users's Guide(version 2.2).* Computer Science Department, Australian National University.

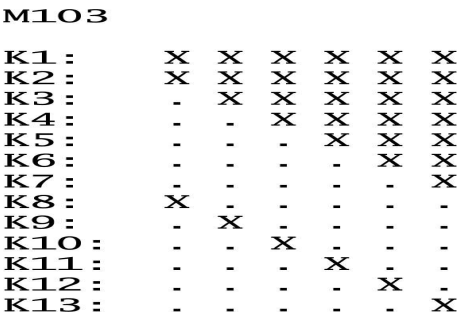
[8] L. Rade, Nertil Westergren 1989: *BETA Mathematics Handbook for Science and Engineering* Studentlitteratur. ISBN: 91-44-008392

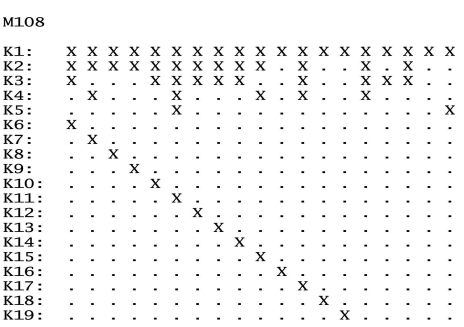
[9] S. Skiena 1997: *The Algorithm Design Manual.* Telos/Springer-Verlag. ISBN 0387948600

Appendix

.1 Lock Charts Easy Problems

Cylinders are represented by vertical columns and keys by horizontal lines in the lock charts in this Appendix.





M111

```
K1:    X  X  X  X  X  X  X  X  X  X
K2:    .  X  X  .  .  .  .  .  .  .
K3:    .  X  .  .  .  .  .  .  .  .
K4:    .  X  X  .  .  .  .  .  .  .
K5:    X  X  X  X  X  X  .  .  .  .
K6:    X  X  X  .  X  X  .  .  .  .
K7:    X  .  .  .  .  .  .  .  .  .
K8:    .  X  .  .  .  .  .  .  .  .
K9:    .  .  X  .  .  .  .  .  .  .
K10:   .  .  .  X  .  .  .  .  .  .
K11:   .  .  .  .  X  .  .  .  .  .
K12:   .  .  .  .  .  X  .  .  .  .
K13:   .  .  .  .  .  .  X  .  .  .
K14:   .  .  .  .  .  .  .  X  .  .
K15:   .  .  .  .  .  .  .  .  X  .
K16:   .  .  .  .  .  .  .  .  .  X
```

M112

```
K1:    X  X  X  X  X  X  X  X  X  X  X  X
K2:    X  X  X  X  X  X  .  .  .  .  .  X  .
K3:    .  .  .  .  .  .  X  X  X  X  .  .  .
K4:    X  .  .  .  .  .  .  .  .  .  .  .  .
K5:    .  X  .  .  .  .  .  .  .  .  .  .  .
K6:    .  .  X  .  .  .  .  .  .  .  .  .  .
K7:    .  .  .  X  .  .  .  .  .  .  .  .  .
K8:    .  .  .  .  X  .  .  .  .  .  .  .  .
K9:    .  .  .  .  .  X  .  .  .  .  .  .  .
K10:   .  .  .  .  .  .  X  .  .  .  .  .  .
K11:   .  .  .  .  .  .  .  X  .  .  .  .  .
K12:   .  .  .  .  .  .  .  .  X  .  .  .  .
K13:   .  .  .  .  .  .  .  .  .  X  .  .  .
K14:   .  .  .  .  .  .  .  .  .  .  X  .  .
K15:   .  .  .  .  .  .  .  .  .  .  .  X  .
K16:   .  .  .  .  .  .  .  .  .  .  .  .  X
```

M201

```
K1:     X  X  X  X  X  X  X  X  X
K2:     X  .  .  X  X  X  .  .  X
K3:     .  X  .  X  X  .  X  .  X
K4:     .  .  .  .  X  .  .  .  X
K5:     X  .  X  .  .  X  .  X  X
K6:     .  X  X  .  .  .  X  X  X
K7:     .  .  .  .  .  .  .  X  X
K8:     .  .  X  .  X  .  .  .  X
K9:     .  .  .  X  .  .  .  X  X
K10:    .  X  .  .  .  .  .  .  X
K11:    X  .  .  .  .  .  .  .  .
K12:    .  X  .  .  .  .  .  .  .
K13:    .  .  X  .  .  .  .  .  .
K14:    .  .  .  X  .  .  .  .  .
K15:    .  .  .  .  X  .  .  .  .
K16:    .  .  .  .  .  X  .  .  .
K17:    .  .  .  .  .  .  X  .  .
K18:    .  .  .  .  .  .  .  X  .
K19:    .  .  .  .  .  .  .  .  X
```

M209

```
K1:   X X X X X X X X X X X X X X X X
K2:   X . . . . . X X . . . . . . . .
K3:   . . . . X X . . X . . . . . . .
K4:   . X X X . . . . . . . . . . . .
K5:   . . . . . . . . . X X X . . . .
K6:   . . . . . . . . . . . X . X X .
K7:   X . . . . . . . . . . . . . . .
K8:   . X . . . . . . . . . . . . . .
K9:   . . X . . . . . . . . . . . . .
K10:  . . . X . . . . . . . . . . . .
K11:  . . . . X . . . . . . . . . . .
K12:  . . . . . X . . . . . . . . . .
K13:  . . . . . . X . . . . . . . . .
K14:  . . . . . . . X . . . . . . . .
K15:  . . . . . . . . X . . . . . . .
K16:  . . . . . . . . . X . . . . . .
K17:  . . . . . . . . . . X . . . . .
K18:  . . . . . . . . . . . X . . . .
K19:  . . . . . . . . . . . . X . . .
K20:  . . . . . . . . . . . . . X . .
K21:  . . . . . . . . . . . . . . X .
K22:  . . . . . . . . . . . . . . . X
```

M109

```
K1:     X  X  X  X  X  X  X  X  X
K2:     X  X  X  X  .  .  .  .  X
K3:     .  .  .  .  X  X  X  X  X
K4:     X  X  .  .  .  .  .  .  X
K5:     .  .  X  X  .  .  .  .  X
K6:     .  .  .  .  X  X  .  .  X
K7:     .  .  .  .  .  .  X  X  X
K8:     X  .  .  .  .  .  .  .  X
K9:     .  X  .  .  .  .  .  .  X
K10:    .  .  X  .  .  .  .  .  X
K11:    .  .  .  X  .  .  .  .  X
K12:    .  .  .  .  X  .  .  .  X
K13:    .  .  .  .  .  X  .  .  X
K14:    .  .  .  .  .  .  X  .  X
K15:    .  .  .  .  .  .  .  X  X
K16:    .  .  .  .  .  .  .  .  X
```

M204

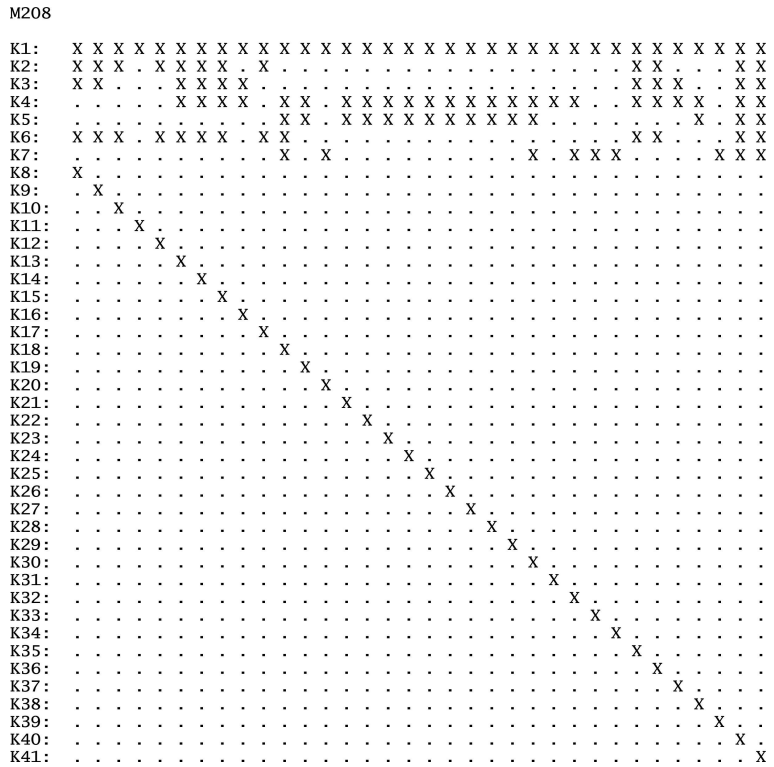
```
K1:    X X .
K2:    X . .
K3:    X . .
K4:    X . .
K5:    X . .
K6:    X X .
K7:    X . X
```

M203

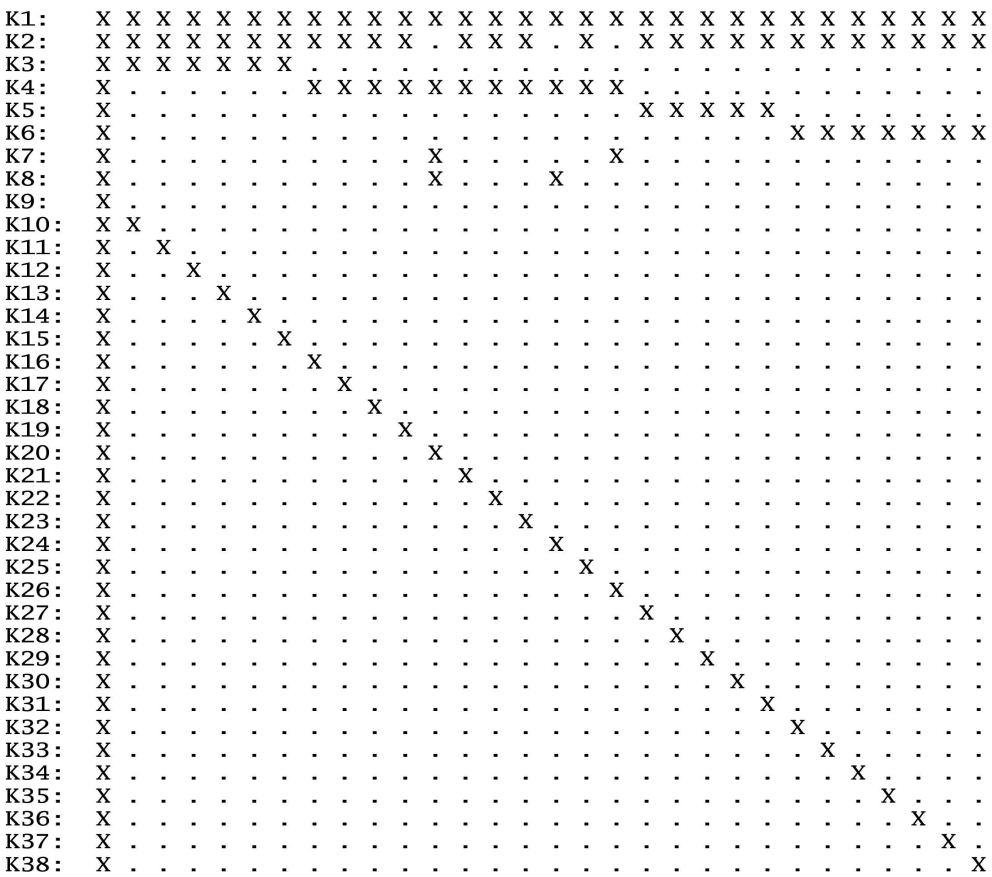
```
K1:   X X X X X X X X X X X X X X X X X X X X X
K2:   X X X X X X X X X X X X X X X X . . . . .
K3:   X . . . . . X X X X . . . X X X . . . . .
K4:   X . . . . . . . . . . . . . . . . . . . .
K5:   . X . . . . . . . . . . . . . . . . . . .
K6:   . . X . . . . . . . . . . . . . . . . . .
K7:   . . . X . . . . . . . . . . . . . . . . .
K8:   . . . . X . . . . . . . . . . . . . . . .
K9:   . . . . . X . . . . . . . . . . . . . . .
K10:  . . . . . . X . . . . . . . . . . . . . .
K11:  . . . . . . . X . . . . . . . . . . . . .
K12:  . . . . . . . . X . . . . . . . . . . . .
K13:  . . . . . . . . . X . . . . . . . . . . .
K14:  . . . . . . . . . . X . . . . . . . . . .
K15:  . . . . . . . . . . . X . . . . . . . . .
K16:  . . . . . . . . . . . . X . . . . . . . .
K17:  . . . . . . . . . . . . . X . . . . . . .
K18:  . . . . . . . . . . . . . . X . . . . . .
K19:  . . . . . . . . . . . . . . . X . . . . .
K20:  . . . . . . . . . . . . . . . . X . . . .
K21:  . . . . . . . . . . . . . . . . . X . . .
K22:  . . . . . . . . . . . . . . . . . . X . .
```

.2 Lock Charts Medium Hard Problems

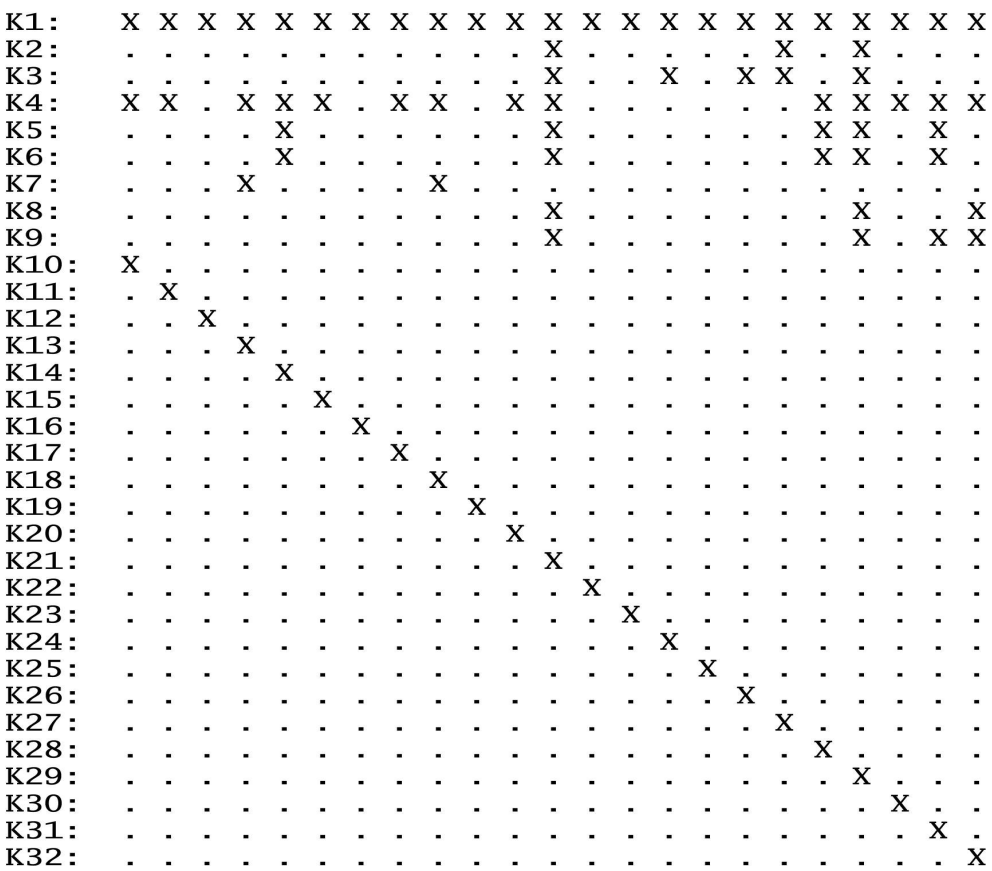
Cylinders are represented by vertical columns and keys by horizontal lines in the lock charts in this Appendix.



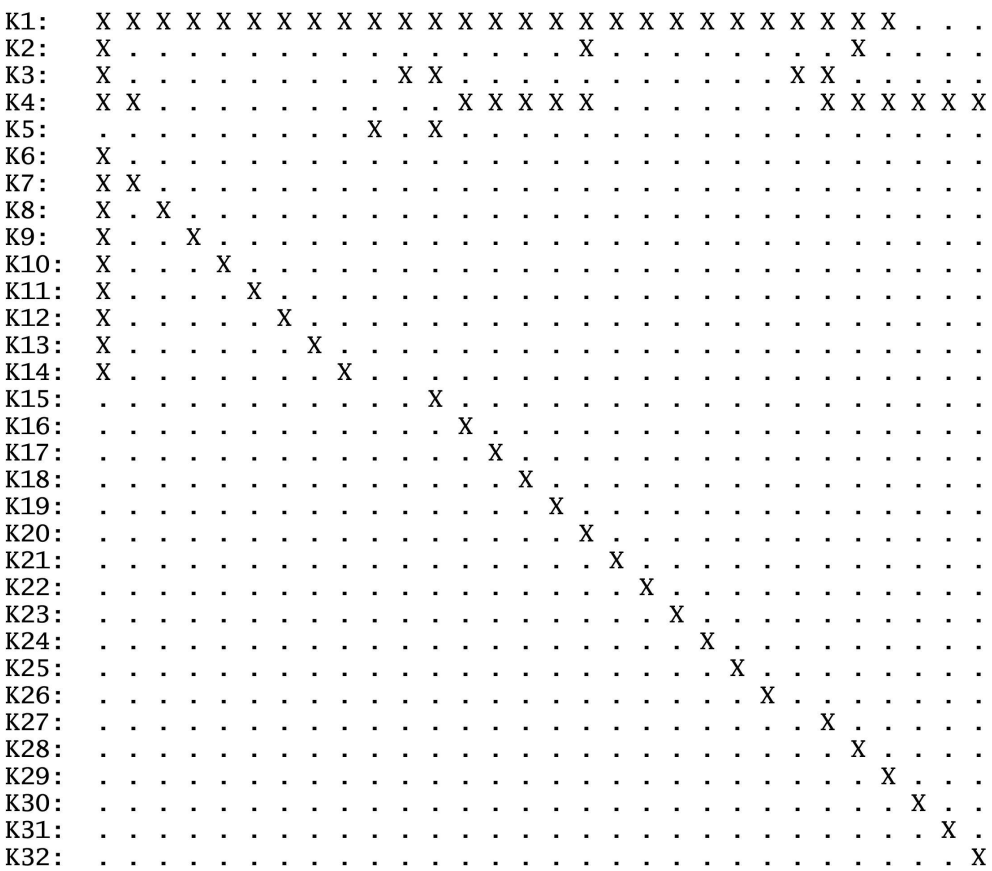
M101


K1: X X X X X X X X X X X X X X X X X X X X X X X X X X X X X
K2: X X X X X X X X X X X . X X X . X . X X X X X X X X X X X
K3: X X X X X X X . . . .
K4: X . . . . . . X X X X X X X X X X X . . . .
K5: X . . . . X X X X X . . . .
K6: X . . . . X X X X X X X
K7: X . . . . X . . . . . X . . . .
K8: X . . . . X . . . X . . . .
K9: X . . . .
K10: X X . . . .
K11: X . X . . . .
K12: X . . X . . . .
K13: X . . . X . . . .
K14: X . . . . X . . . .
K15: X . . . . . X . . . .
K16: X . . . . . . X . . . .
K17: X . . . . X . . . .
K18: X . . . . X . . . .
K19: X . . . . X . . . .
K20: X . . . . X . . . .
K21: X . . . . X . . . .
K22: X . . . . X . . . .
K23: X . . . . X . . . .
K24: X . . . . X . . . .
K25: X . . . . X . . . .
K26: X . . . . X . . . .
K27: X . . . . X . . . .
K28: X . . . . X . . . .
K29: X . . . . X . . . .
K30: X . . . . X . . . .
K31: X . . . . X . . . .
K32: X . . . . X . . . .
K33: X . . . . X . . . .
K34: X . . . . X . . . . . .
K35: X . . . . X . . . . .
K36: X . . . . X . . . .
K37: X . . . . X . . .
K38: X X

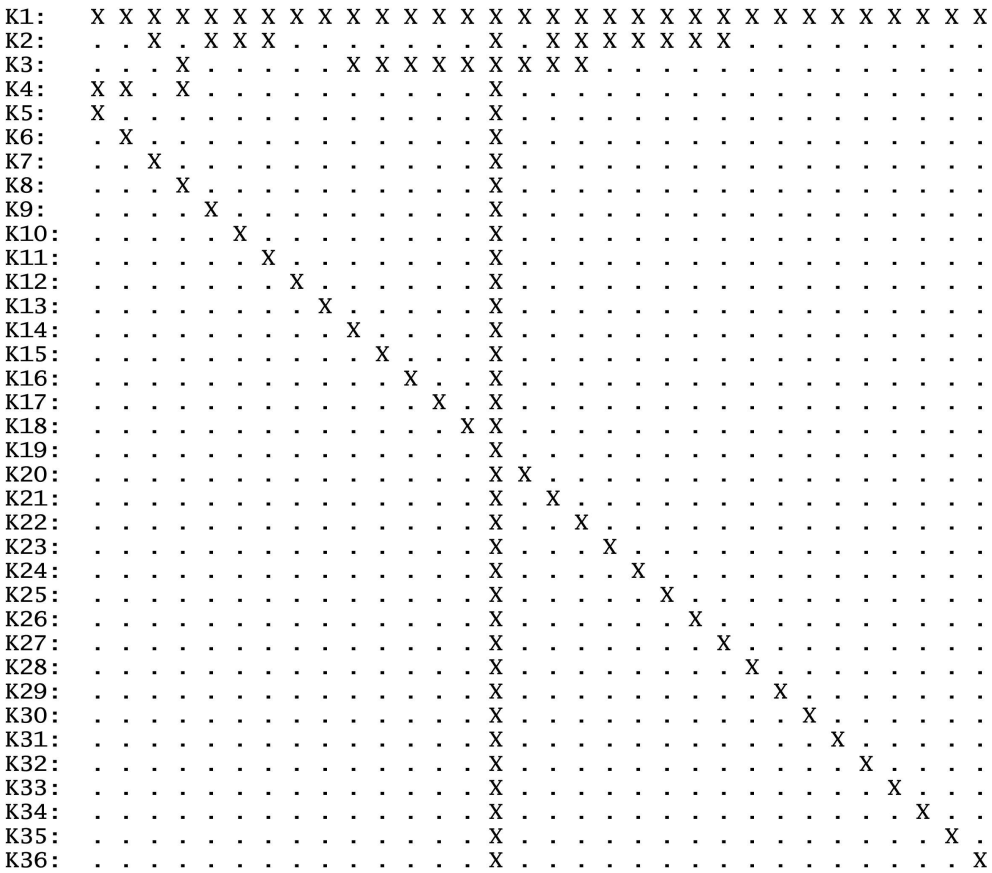
M104


K1: X X X X X X X X X X X X X X X X X X X X X X
K2: . . . . X . . . . . X . X . . .
K3: . . . . X . . X . X X . X . . .
K4: X X . X X X . X X . X X . . . . . . X X X X X
K5: . . . . . X . . . . X . . . . . . X X . X .
K6: . . . . . X . . . . X . . . . . . X X . X .
K7: . . . X . . . . X . . . .
K8: . . . . X . . . . X . . X
K9: . . . . X . . . . X . X X
K10: X . . . .
K11: . X . . . .
K12: . . X . . . .
K13: . . . X . . . .
K14: . . . . X . . . .
K15: . . . . . X . . . .
K16: . . . . . . X . . . .
K17: . . . . X . . . .
K18: . . . . X . . . .
K19: . . . . X . . . .
K20: . . . . X . . . .
K21: . . . . X . . . .
K22: . . . . X . . . .
K23: . . . . X . . . .
K24: . . . . X . . . .
K25: . . . . X . . . .
K26: . . . . X . . . .
K27: . . . . X . . . . . .
K28: . . . . X . . . . .
K29: . . . . X . . . .
K30: . . . . X . . .
K31: . . . . X . .
K32: X

M106


K1: X X X X X X X X X X X X X X X X X X X X X X X X X . . .
K2: X . . . . X . . . . X . . . .
K3: X . . . . X X . . . . X X . . . . .
K4: X X . . . . X X X X X . . . . X X X X X X X
K5: . . . . X . X . . . .
K6: X . . . .
K7: X X . . . .
K8: X . X . . . .
K9: X . . X . . . .
K10: X . . . X . . . .
K11: X . . . . X . . . .
K12: X . . . . . X . . . .
K13: X . . . . . . X . . . .
K14: X . . . . X . . . .
K15: . . . . X . . . .
K16: . . . . X . . . .
K17: . . . . X . . . .
K18: . . . . X . . . .
K19: . . . . X . . . .
K20: . . . . X . . . .
K21: . . . . X . . . .
K22: . . . . X . . . .
K23: . . . . X . . . .
K24: . . . . X . . . .
K25: . . . . X . . . .
K26: . . . . X . . . .
K27: . . . . X . . . .
K28: . . . . X . . . .
K29: . . . . X . . . .
K30: . . . . X . . . . . .
K31: . . . . X . . . . .
K32: . . . . X . . . .

M107


K1: X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X
K2: . . X . X X X . . . . X . X X X X X X X . . . .
K3: . . . X . . . . . X X X X X X X X X . . . .
K4: X X . X . . . . X . . . .
K5: X . . . . X . . . .
K6: . X . . . . X . . . .
K7: . . X . . . . X . . . .
K8: . . . X . . . . X . . . .
K9: . . . . X . . . . X . . . .
K10: . . . . . X . . . . X . . . .
K11: . . . . . . X . . . . X . . . .
K12: . . . . X . . . . X . . . .
K13: . . . . X . . . . . . X . . . .
K14: . . . . X . . . . . X . . . .
K15: . . . . X . . . . X . . . .
K16: . . . . X . . . X . . . .
K17: . . . . X . . X . . . .
K18: . . . . X X . . . .
K19: . . . . X . . . .
K20: . . . . X X . . . .
K21: . . . . X . X . . . .
K22: . . . . X . . X . . . .
K23: . . . . X . . . X . . . .
K24: . . . . X . . . . X . . . .
K25: . . . . X . . . . . X . . . .
K26: . . . . X . . . . . . X . . . .
K27: . . . . X . . . . X . . . .
K28: . . . . X . . . . X . . . .
K29: . . . . X . . . . X . . . .
K30: . . . . X . . . . X . . . . . .
K31: . . . . X . . . . X . . . . .
K32: . . . . X . . . . X . . . .
K33: . . . . X . . . . X . . .
K34: . . . . X . . . . X . .
K35: . . . . X . . . . X .
K36: . . . . X X

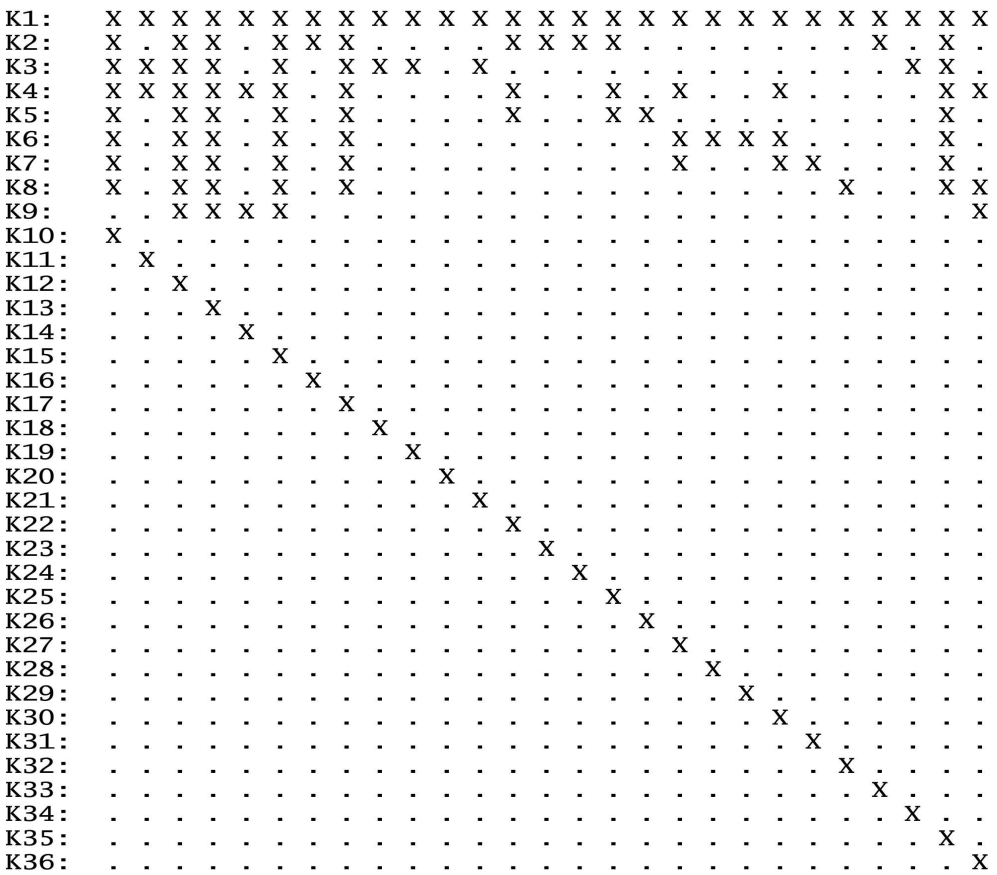
M202

```
K1:   X X X X X X X X X X . X X . X X X X X
K2:   X X X X . . . . . X . . . . . X . X X .
K3:   . X X . X . . . . . X . . . . . . X X .
K4:   X X X . . . X . X . . . . . X . . . .
K5:   X . . . . . . . . X . . . . . . . . .
K6:   . . . . . . . . . . X . . . . . . X X .
K7:   X X X . . X . X X X . . . . X X X X .
K8:   . . . . . . . . . . . X X . X X X . . .
K9:   . . . . . . . . . . . . X X . . X . . X
K10:  X . . . . . . . . . . . . . . . . . . .
K11:  . X . . . . . . . . . . . . . . . . . .
K12:  . . X . . . . . . . . . . . . . . . . .
K13:  . . . X . . . . . . . . . . . . . . . .
K14:  . . . . X . . . . . . . . . . . . . . .
K15:  . . . . . X . . . . . . . . . . . . . .
K16:  . . . . . . X . . . . . . . . . . . . .
K17:  . . . . . . . X . . . . . . . . . . . .
K18:  . . . . . . . . X . . . . . . . . . . .
K19:  . . . . . . . . . X . . . . . . . . . .
K20:  . . . . . . . . . . X . . . . . . . . .
K21:  . . . . . . . . . . . X . . . . . . . .
K22:  . . . . . . . . . . . . X . . . . . . .
K23:  . . . . . . . . . . . . . X . . . . . .
K24:  . . . . . . . . . . . . . . X . . . . .
K25:  . . . . . . . . . . . . . . . X . . . .
K26:  . . . . . . . . . . . . . . . . X . . .
K27:  . . . . . . . . . . . . . . . . . X . .
K28:  . . . . . . . . . . . . . . . . . . X .
```

M201

K1:	X	X	X	X	X	X	X	X	X
K2:	X	.	.	X	X	X	.	.	X
K3:	.	X	.	X	X	.	X	.	X
K4:	.	.	.	.	X	.	.	.	X
K5:	X	.	X	.	.	X	.	X	X
K6:	.	X	X	.	.	.	X	X	X
K7:	.	.	.	.	.	.	.	X	X
K8:	.	.	X	.	X	.	.	.	X
K9:	.	.	.	X	.	.	.	X	X
K10:	.	X	.	.	.	.	.	.	X
K11:	X	.	.	.	.	.	.	.	.
K12:	.	X	.	.	.	.	.	.	.
K13:	.	.	X	.	.	.	.	.	.
K14:	.	.	.	X	.	.	.	.	.
K15:	.	.	.	.	X	.	.	.	.
K16:	.	.	.	.	.	X	.	.	.
K17:	.	.	.	.	.	.	X	.	.
K18:	.	.	.	.	.	.	.	X	.
K19:	.	.	.	.	.	.	.	.	X

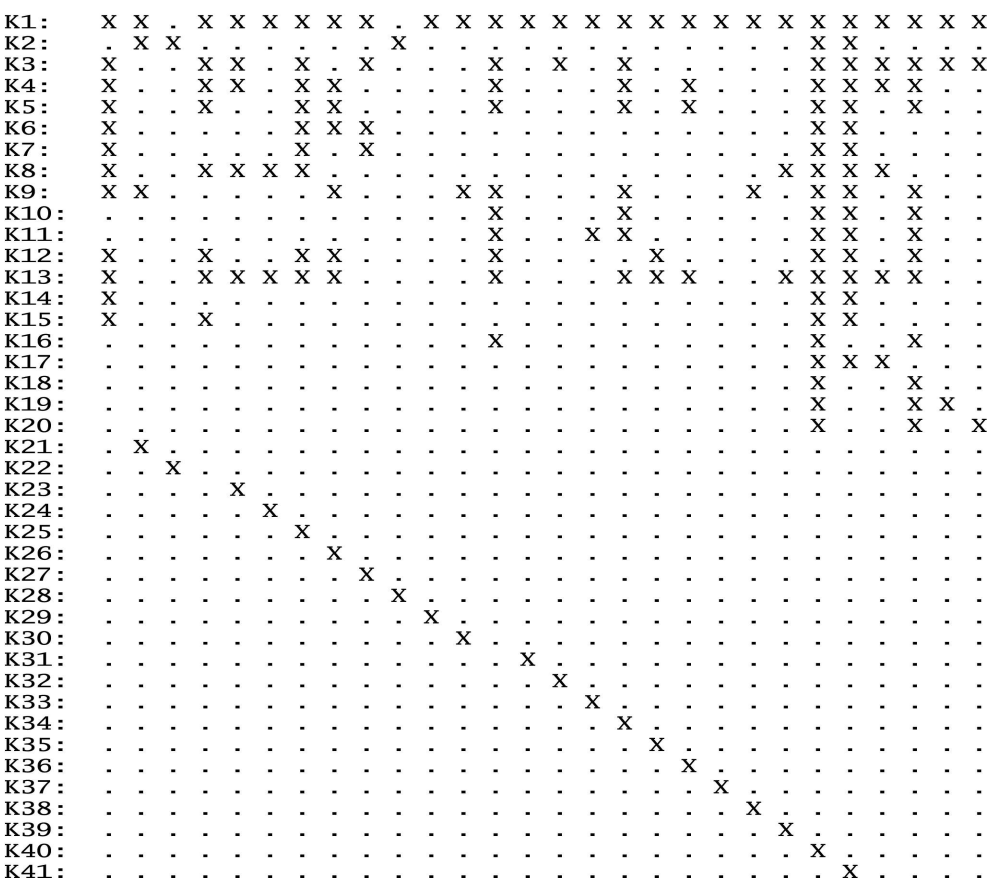
M206


K1: X X X X X X X X X X X X X X X X X X X X X X X X X X
K2: X . X X . X X X . . . . X X X X . . . . X . X .
K3: X X X X . X . X X X . X . . . . X X .
K4: X X X X X X . X . . . . X . . X . X . . X . . . . X X
K5: X . X X . X . X . . . . X . . X X . . . . X .
K6: X . X X . X . X . . . . X X X X . . . . X .
K7: X . X X . X . X . . . . X . . X X . . . X .
K8: X . X X . X . X . . . . X . . X X
K9: . . X X X X X
K10: X . . . .
K11: . X . . . .
K12: . . X . . . .
K13: . . . X . . . .
K14: . . . . X . . . .
K15: . . . . . X . . . .
K16: . . . . . . X . . . .
K17: . . . . X . . . .
K18: . . . . X . . . .
K19: . . . . X . . . .
K20: . . . . X . . . .
K21: . . . . X . . . .
K22: . . . . X . . . .
K23: . . . . X . . . .
K24: . . . . X . . . .
K25: . . . . X . . . .
K26: . . . . X . . . .
K27: . . . . X . . . .
K28: . . . . X . . . .
K29: . . . . X . . . .
K30: . . . . X . . . . . .
K31: . . . . X . . . . .
K32: . . . . X . . . .
K33: . . . . X . . .
K34: . . . . X . .
K35: . . . . X .
K36: X

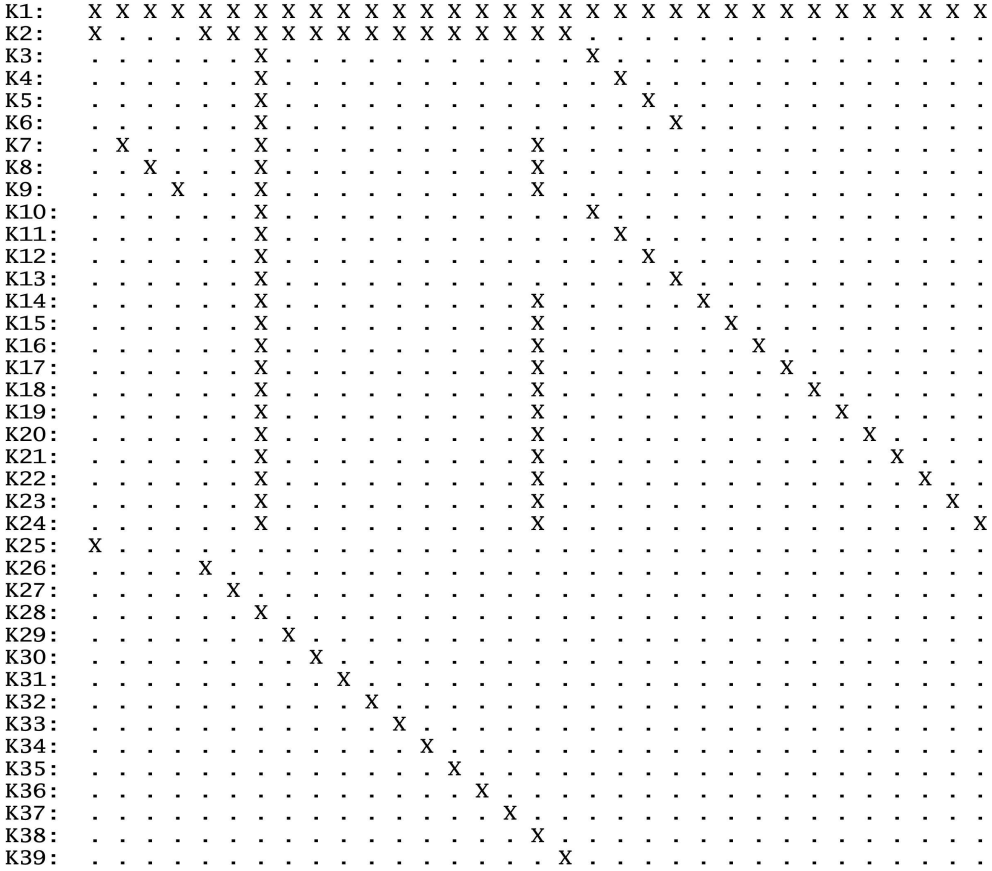
.3 Lock Charts Hard Problems

Cylinders are represented by vertical columns and keys by horizontal lines in the lock charts in this Appendix.

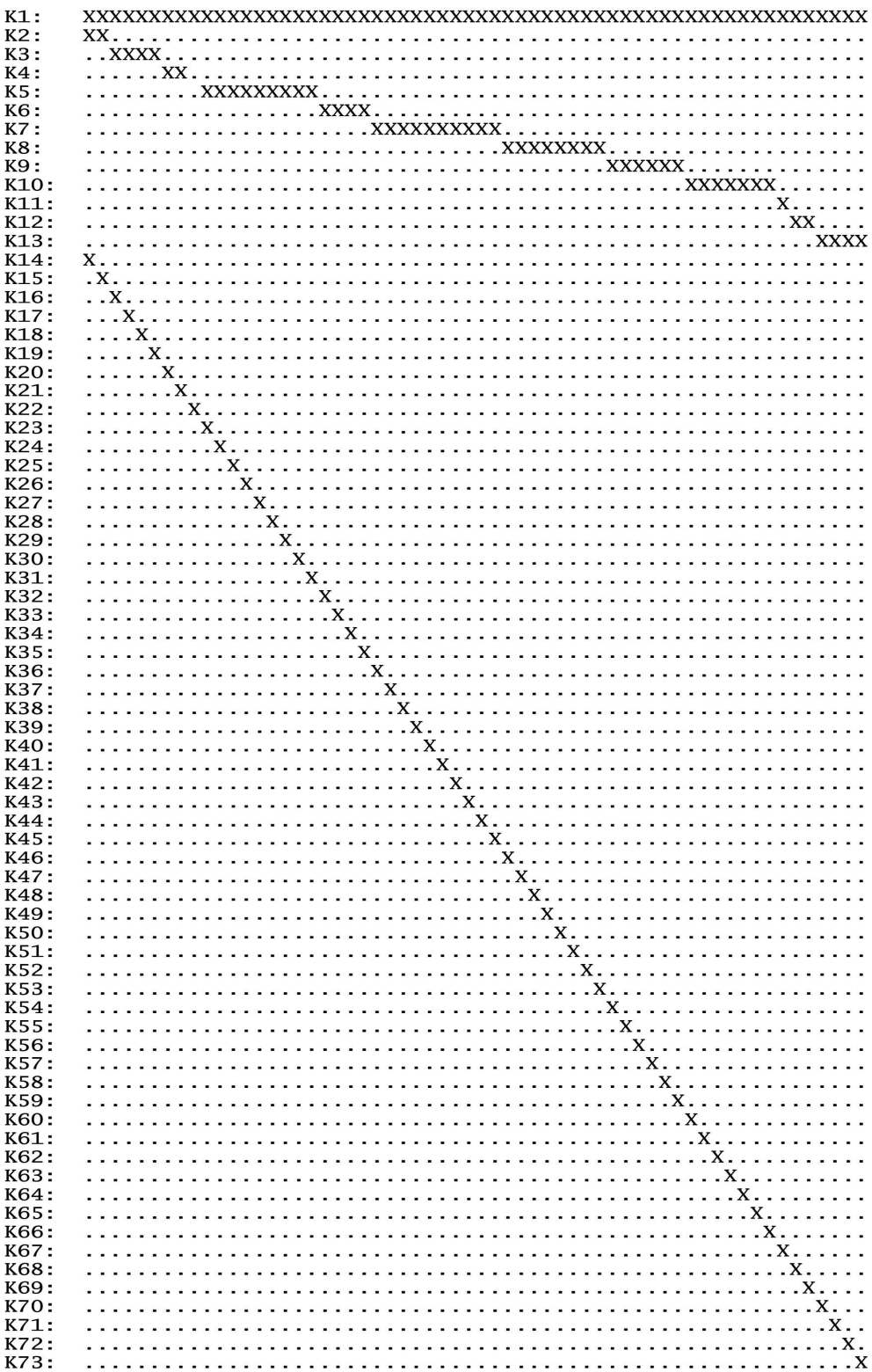
M100

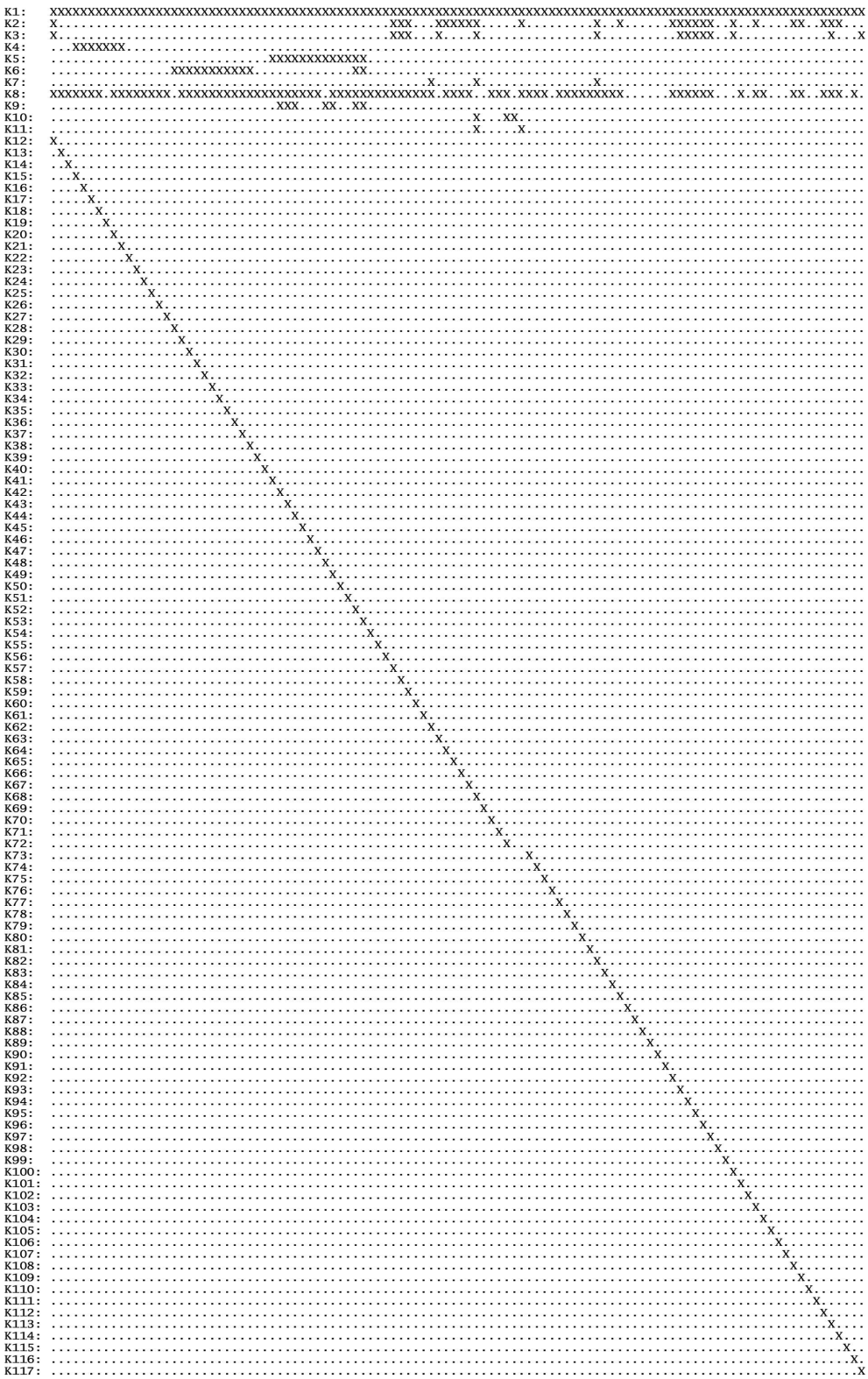


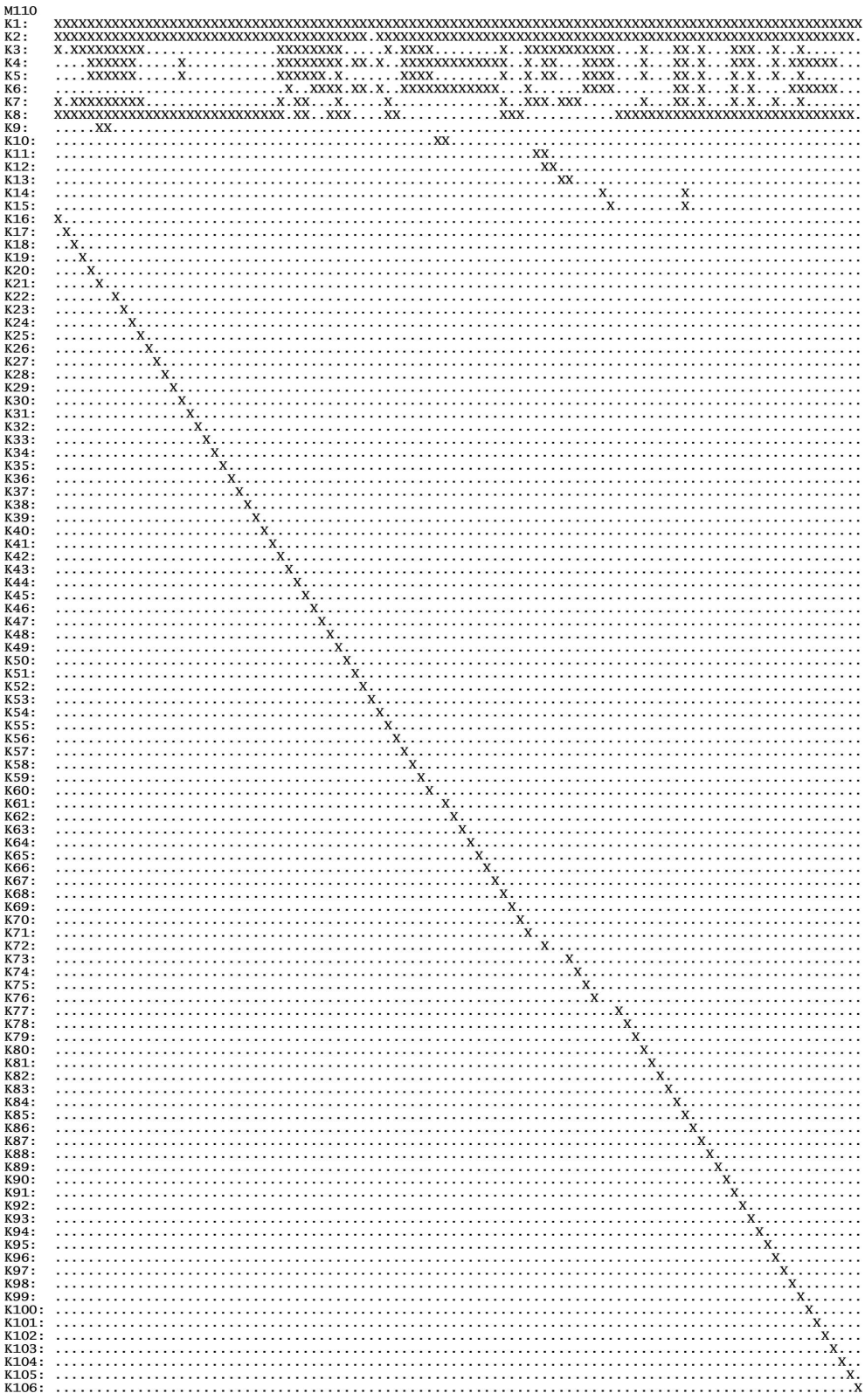
M205


K1: X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X
K2: X . . . X X X X X X X X X X X X X X X . . . .
K3: . . . . . . X . . . . X . . . .
K4: . . . . . . X . . . . X . . . .
K5: . . . . . . X . . . . X . . . .
K6: . . . . . . X . . . . X . . . .
K7: . X . . . . X . . . . X . . . .
K8: . . X . . . X . . . . X . . . .
K9: . . . X . . X . . . . X . . . .
K10: . . . . . X . . . . X . . . .
K11: . . . . . X . . . . X . . . .
K12: . . . . . X . . . . X . . . .
K13: . . . . . X . . . . X . . . .
K14: . . . . . X . . . . X . . . . . X . . . .
K15: . . . . . X . . . . X . . . . . . X . . . .
K16: . . . . . X . . . . X . . . . X . . . .
K17: . . . . . X . . . . X . . . . X . . . .
K18: . . . . . X . . . . X . . . . X . . . .
K19: . . . . . X . . . . X . . . . X . . . .
K20: . . . . . X . . . . X . . . . X . . . . .
K21: . . . . . X . . . . X . . . . X . . .
K22: . . . . . X . . . . X . . . . X . .
K23: . . . . . X . . . . X . . . . X .
K24: . . . . . X . . . . X X
K25: X . . . .
K26: . . . . X . . . .
K27: . . . . . X . . . .
K28: . . . . . X . . . .
K29: . . . . . X . . . .
K30: . . . . . X . . . .
K31: . . . . . X . . . .
K32: . . . . . X . . . .
K33: . . . . . X . . . .
K34: . . . . . X . . . .
K35: . . . . . X . . . .
K36: . . . . . X . . . .
K37: . . . . . X . . . .
K38: . . . . . X . . . .
K39: . . . . . X . . . .

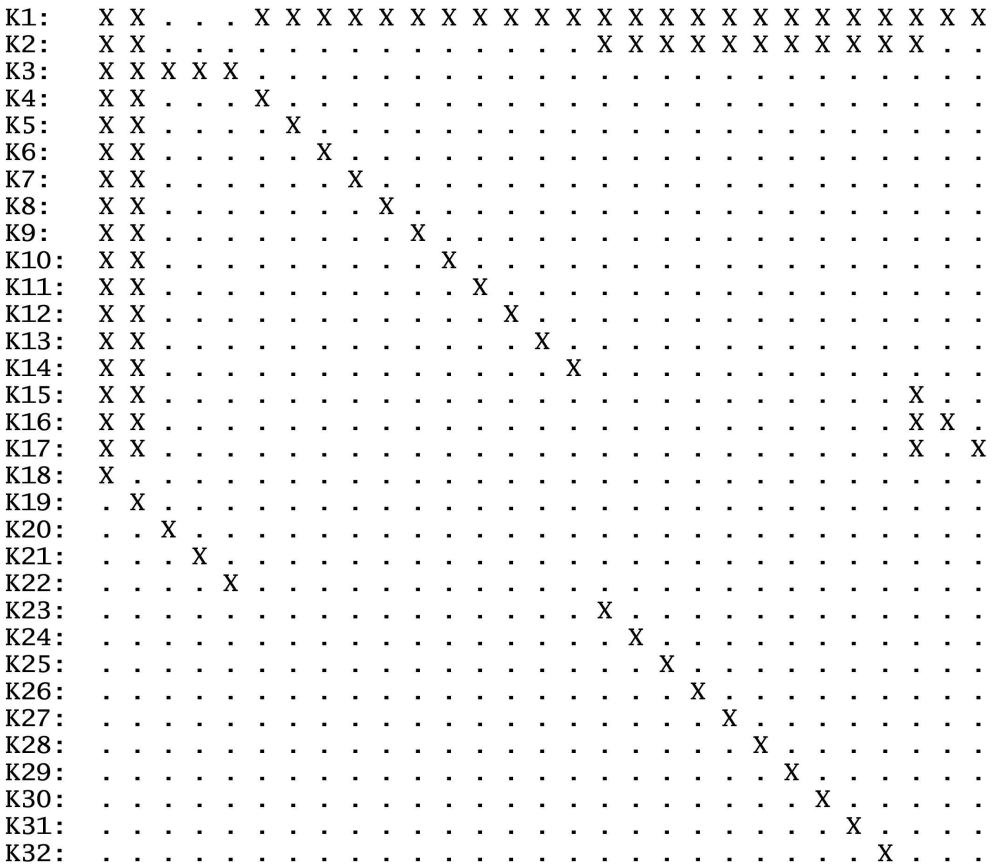
M102




M105
K1:
K2:
K3:
K4:
K5:
K6:
K7:
K8:
K9:
K10:
K11:
K12:
K13:
K14:
K15:
K16:
K17:
K18:
K19:
K20:
K21:
K22:
K23:
K24:
K25:
K26:
K27:
K28:
K29:
K30:
K31:
K32:
K33:
K34:
K35:
K36:
K37:
K38:
K39:
K40:
K41:
K42:
K43:
K44:
K45:
K46:
K47:
K48:
K49:
K50:
K51:
K52:
K53:
K54:
K55:
K56:
K57:
K58:
K59:
K60:
K61:
K62:
K63:
K64:
K65:
K66:
K67:
K68:
K69:
K70:
K71:
K72:
K73:
K74:
K75:
K76:
K77:
K78:
K79:
K80:
K81:
K82:
K83:
K84:
K85:
K86:
K87:
K88:
K89:
K90:
K91:
K92:
K93:
K94:
K95:
K96:
K97:
K98:
K99:
K100:
K101:
K102:
K103:
K104:
K105:
K106:
K107:
K108:
K109:
K110:
K111:
K112:
K113:
K114:
K115:
K116:
K117:


M110
K1:
K2:
K3:
K4:
K5:
K6:
K7:
K8:
K9:
K10:
K11:
K12:
K13:
K14:
K15:
K16:
K17:
K18:
K19:
K20:
K21:
K22:
K23:
K24:
K25:
K26:
K27:
K28:
K29:
K30:
K31:
K32:
K33:
K34:
K35:
K36:
K37:
K38:
K39:
K40:
K41:
K42:
K43:
K44:
K45:
K46:
K47:
K48:
K49:
K50:
K51:
K52:
K53:
K54:
K55:
K56:
K57:
K58:
K59:
K60:
K61:
K62:
K63:
K64:
K65:
K66:
K67:
K68:
K69:
K70:
K71:
K72:
K73:
K74:
K75:
K76:
K77:
K78:
K79:
K80:
K81:
K82:
K83:
K84:
K85:
K86:
K87:
K88:
K89:
K90:
K91:
K92:
K93:
K94:
K95:
K96:
K97:
K98:
K99:
K100:
K101:
K102:
K103:
K104:
K105:
K106:

M200


K1: X X . . . X X X X X X X X X X X X X X X X X X X X X X X X
K2: X X . . . . X X X X X X X X X X X . .
K3: X X X X X . . . .
K4: X X . . . X . . . .
K5: X X . . . . X . . . .
K6: X X . . . . . X . . . .
K7: X X . . . . . . X . . . .
K8: X X . . . . X . . . .
K9: X X . . . . X . . . .
K10: X X . . . . X . . . .
K11: X X . . . . X . . . .
K12: X X . . . . X . . . .
K13: X X . . . . X . . . .
K14: X X . . . . X . . . .
K15: X X . . . . X . .
K16: X X . . . . X X .
K17: X X . . . . X . X
K18: X . . . .
K19: . X . . . .
K20: . . X . . . .
K21: . . . X . . . .
K22: . . . . X . . . .
K23: . . . . X . . . .
K24: . . . . X . . . .
K25: . . . . X . . . .
K26: . . . . X . . . .
K27: . . . . X . . . .
K28: . . . . X . . . . . .
K29: . . . . X . . . . .
K30: . . . . X . . . .
K31: . . . . X . . .
K32: . . . . X . .