Masaryk University
Faculty of Informatics

Algorithmic Analysis of Code-Breaking Games

Master's Thesis

Miroslav Klimoš

Brno, 2014

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institution of tertiary education. Information derived from the published or unpublished work of others has been acknowledged in the text and a list of references is given.

Advisor: prof. RNDr. Antonín Kučera, Ph.D.

Acknowledgement

I would like to express my deepest appreciation and thanks to my advisor, prof. RNDr. Antonín Kučera, Ph.D., for inspiring and valuable discussions and for coming up with this fascinating topic in the first place.

I would also like to thank friends of mine who expressed their interest in this work and provided constructive comments. All my friends deserve a special thanks for their patience with me during the time I have been working on this thesis.

Keywords

code-breaking games,
deductive games,
strategy synthesis,
one-step look-ahead strategies,
SAT solving,
model counting,
Mastermind,
counterfeit coin

Abstract

Code-breaking games are two-player games in which the first player selects a code from a given set and the second player strives to reveal it using a minimal number of experiments. A prominent example of a code-breaking game is the board game Mastermind, where the codebreaker tries to guess a combination of coloured pegs.

There are many natural questions regarding code-breaking games. What strategy for experiment selection should the codebreaker use? Is it possible to compute a lower or upper bound for the number of experiments needed? Can we compute the optimal strategy? What is the performance of a given heuristic? Much research on the topic has been done but it usually focuses on one particular game and little has been written about code-breaking games in general.

In this work, we create a general model of code-breaking games based on propositional logic and design a computer language for game specification. Further, we suggest general algorithms for game analysis and strategy synthesis and implement them in a computer program. Using the tool, we can reproduce existing results for Mastermind, analyse new code-breaking games and easily evaluate new heuristics for experiment selection.

Contents

1 **Introduction** . . . 3
2 **Examples of code-breaking games and existing results** . . . 7
2.1 *The counterfeit coin* . . . 7
2.2 *Mastermind* . . . 10
2.3 *Other games* . . . 13
3 **Code-breaking game model** . . . 17
3.1 *Notation and terminology* . . . 17
3.2 *Basic definitions* . . . 17
3.3 *Strategies in general* . . . 23
3.4 *One-step look-ahead strategies* . . . 28
4 **Experiment equivalence and algorithms** . . . 31
4.1 *Experiment equivalence* . . . 31
4.2 *Well-formed check* . . . 36
4.3 *Analysis of one-step look-ahead strategies* . . . 37
4.4 *Optimal strategy synthesis* . . . 38
5 **COBRA tool** . . . 43
5.1 *Input language* . . . 43
5.2 *Compilation and basic usage* . . . 47
5.3 *Modes of operation* . . . 47
5.4 *Modularity and extensibility* . . . 49
5.5 *SAT solving* . . . 51
5.6 *Graph isomorphism* . . . 55
5.7 *Implementation details* . . . 55
6 **Experimental results** . . . 57
6.1 *Performance* . . . 57
6.2 *One-step look-ahead strategies* . . . 59
7 **Conclusions** . . . 63
Bibliography . . . 65
Contents of the electronic attachment . . . 69

1 Introduction

Code-breaking games (sometimes also called *deductive games* or *searching games*) are games of two players in which the first player, usually referred to as *the codemaker*, chooses a secret code from a given set, and the second player, usually referred to as *the codebreaker*, strives to reveal the code through a series of experiments that give him partial information about the code.

One prominent example of a code-breaking game is famous board game *Mastermind*. In this game, the codemaker creates a puzzle for the codebreaker by choosing a combination of four coloured pegs (with colour repetitions allowed). The codebreaker makes guesses about the colours, which are evaluated by the codemaker with black and white markers. A black marker corresponds to a position where the code and the guess match. A white marker means that some colour is present both in the code and in the guess but at different positions.

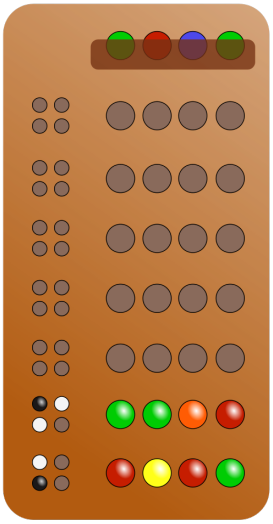

Figure 1.1: Mastermind game (illustrative image)[1].

Another example of a code-breaking game is the *counterfeit coin problem*, the problem of identifying an odd-weight coin among a collection of genuine coins using only a balance scale. The codemaker is not a real player here; the balance scale takes his function and evaluates the weighings performed by the codebreaker. Numerous other examples can be found among various board games and logic puzzles; we present some of them in the next chapter.

Code-breaking games offer many interesting research problems.

- *How should the codebreaker play in order to minimize the number of experiments needed to undoubtedly determine the code?*
- *Is there a strategy that would guarantee revealing the code in at most k steps?*
- *What strategy is optimal with respect to the average-case number of experiments, given that the code is selected from the given set with uniform distribution?*

Synthesis of an optimal strategy is a computationally intensive task. In some games, the optimal strategy might have a simple structure and can be described easily, such as in the counterfeit coin problem (see section Section 2.1 for details).

1. Image adopted from http://commons.wikimedia.org/wiki/File:Mastermind_beispiel.svg, by Thomas Steiner under GFDL.

In general, however, the strategy may have complicated structure and there may be no other way to discover an optimal strategy than to consider all possible experiments in a given state and analyse the subproblems.

Therefore, one may prefer a suboptimal strategy or heuristic for experiment selection, which is easier to compute. This brings about another kind of questions. *Given a strategy, how can we compute the worst-case and the average-case number of experiments the strategy needs to reveal the code?*

Some particular code-breaking games, such as Mastermind and the counterfeit coin problem, have been intensively studied in the last decades and most of these questions are at least partially answered. A detailed summary of the existing results is presented in Chapter 2. Nevertheless, little has been written about code-breaking games in general. Some authors have suggested general methods (and applied them in one particular game, e.g. [1, 2]), some have vaguely stated that their approach can be applied to other games of the same kind but, to the best of our knowledge, no one has tried to create a general framework and provide results for code-breaking games in general.

This work proposes to bridge the gap and provides a general framework for code-breaking games based on propositional logic. The secret code is encoded as a valuation of a set of propositional variables and the codebreaker's goal is to discover the valuation through a series of experiments. Each experiment can result in several outcomes, which are given in the form of a propositional formula.

We address the following challenges in the suggested framework.

- Formally define code-breaking games and strategies.
- Propose general strategies or heuristics for experiment selection.
- Suggest efficient methods for state-space reduction based on symmetry detection.
- Propose algorithms for strategy evaluation and optimal strategy synthesis.
- Design a computer language for game specification.
- Develop a computer program that parses a game description from the designed language and implements suggested algorithms.

Some of the proposed methods for code-breaking game analysis depend on algorithms for related problems. To analyse propositional formulas emerging during the course of the game, we need to decide satisfiability or count the number of models of a formula, which can be done using a modern SAT solver. Further, our symmetry detection approach is based on the reduction of experiment equivalence to graph isomorphism. For this purpose, we need a tool that computes the canonical labelling of a given graph.

We created a computer program for code-breaking game analysis and named it COBRA, the code-breaking game analyser. Using this tool, we can reproduce some of the existing results for Mastermind, analyse new code-breaking games

and and easily evaluate new heuristics for experiment selection.

The thesis is structured as follows. Chapter 2 introduces several examples of code-breaking games and discusses existing results, variants of the games and related research. The general code-breaking game model is described in Chapter 3. Chapter 4 is dedicated to our method for symmetry detection and other algorithms. Our computer program, COBRA, with descriptions of its usage and abilities is introduced in Chapter 5. Experimental results with comparisons of analysed strategies are presented in Chapter 6. Finally, Chapter 7 concludes the work with suggestions for future work and possible extensions of the tool.

2 Game examples and existing results

We introduce a few examples of code-breaking games in this chapter. The counterfeit coin problem and Mastermind are quite well known, the other examples are based on various board games or less known logic puzzles. We briefly summarize related research for each game, discuss its variations and applications and give a list of references.

2.1 The counterfeit coin

The problem of finding a counterfeit coin among a collection of genuine coins in the fewest number of weighings on a balance scale is a folklore of recreational mathematics.
In all problems of this kind, you can only use the scale to weigh the coins. You put some coins on the left pan, the same number of coins on the right pan and get one of the three possible outcomes. Either both the sides weigh the same (denoted "=") or the left side is lighter ("<"), or the right side is lighter (">"). The standard, easiest version can be formulated as follows.

Figure 2.1: Balance scale (illustrative image)[1].

Problem 2.2 (The nine coin problem). *You are given $n \geq 3$ (typically 9) coins, all except one having the same weight. The counterfeit coin is known to be lighter. Identify the counterfeit coin in minimal number of weighings.*

This problem is very easy as one can use *ternary search* algorithm. In short, we divide the coins into thirds and put one third on the left pan and another on the right pan of the balance scale. If both pans weigh the same, the counterfeit coin must be in the last third, otherwise it must be on the lighter pan. In this way, the size of the search space is reduced by a factor of three in each step, which is optimal.
In 1940s, more complicated version was introduced by Grossman[3].

Problem 2.3 (The twelve coin problem). *You are given $n \geq 3$ (typically 12) coins, all except one having the same weight. It is not known whether the counterfeit coin is heavier or lighter. Identify the counterfeit coin and its weight relative to others in minimal number of weighings.*

The optimal solution for $n = 12$ requires 3 weighings. One of the optimal strategies is shown in Figure 2.4 as a decision tree.

---

1. Image adopted from http://pixabay.com/en/justice-silhouette-scales-law-147214, under CC0 1.0 License.

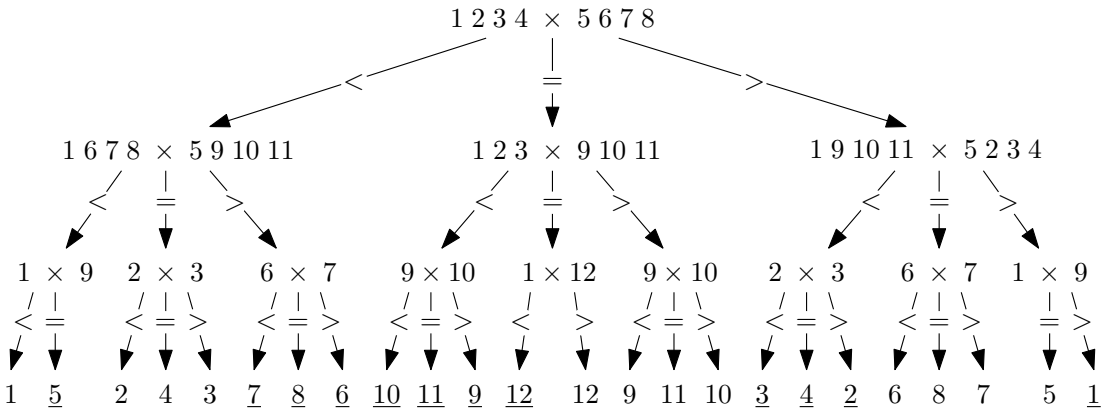

Figure 2.4: Decision tree for The Twelve Coin Problem.
Leaf x means that the coin number x is lighter, $\underline{x}$ means that the coin number x is heavier.

Known results

The research usually focuses on bounds on the maximal value of n for which the problem can be solved in w weighings, for a given w. Thus a solution of a problem is usually formulated as a theorem like the following one.

Theorem 2.5 (Dyson, [4]). *There exists a strategy that identifies the counterfeit coin and its type as described in Problem 2.3 with w weighings, if and only if*

$$3 \leq n \leq \frac{3^w - 3}{2}.$$

Proof. We show the main part of the original Dyson's proof[4] here because of its elegant combinatorial idea. We show a strategy for $n = \frac{1}{2}(3^w - 3)$.
Let us number the coins from 1 to n. To a coin number i, we assign two labels from $\{0, 1, 2\}^w$ – those corresponding to the numbers i and $3^w - 1 - i$ in ternary form. Notice that all possible labels are used exactly once, except for $0^w, 1^w$ and 2^w, which were not assigned to any coin. The labelling has the property that you can get one label of a coin from another by substituting 0 by 2 and 2 by 0.
A label is called "clockwise" if the first change of digit in it is the change from 0 to 1, from 1 to 2, or from 2 to 0. Otherwise, it is called "anticlockwise". Thanks to the property we mentioned, one of the labels of a coin is always clockwise and the other is anticlockwise.
Let $C(i, d)$ be a set of coins such that i-th symbol in its clockwise label is d. Since a permutation changing 0 to 1, 1 to 2 and 2 to 0 transfers coins from $C(i, 0)$ to $C(i, 1)$, from $C(i, 1)$ to $C(i, 2)$ and from $C(i, 2)$ to $C(i, 0)$, all the sets $C(i, d)$ contain exactly $n/3$ coins. Now, let i-th experiment be the weighing of the

coins $C(i, 0)$ against $C(i, 2)$. It remains to show that the experiments uniquely determine the counterfeit coin. Let a_i be 0, 1, or 2 if the result of i-th experiment is left side is lighter, both are the same, or right side is lighter, respectively.
If the counterfeit code is overweight, the i-th symbol of its clockwise label must be a_i. On the other hand, if it is underweight, the i-th symbol of its anticlockwise label must be a_i. The solution of the problem is therefore the coin with the label $a_1 a_2 \dots a_w$ and is heavier than others if and only if this label is clockwise. Figure 2.6 shows an example of the construction for $n = 12 = \frac{1}{2}(3^3 - 3)$, clockwise labels printed in bold.

coin	label 1	label 2
1	**001**	221
2	002	**220**
3	**010**	212
4	**011**	211
5	**012**	210
6	020	**202**
7	021	**201**
8	022	**200**
9	100	**122**
10	101	**121**
11	102	**120**
12	110	**112**

Experiments:

1) 1, 3, 4, 5 × 2, 6, 7, 8
2) 1, 6, 7, 8 × 2, 9, 10, 11
3) 2, 3, 8, 11 × 5, 6, 9, 12

Solution:

the coin labelled $a_1 a_2 a_3$, where a_i is the outcome of i-th experiment.

Figure 2.6: Demonstration of the ternary label construction for $n = 12$.

The case $n < \frac{1}{2}(3^w - 3)$ can be solved similarly with some modifications to the labelling. However, the scheme makes use of a genuine coin that was discovered in the first weighing and, therefore, the following experiments depend on the outcome of the first. Finally, the proof that the coin cannot be identified for $n > \frac{1}{2}(3^w - 3)$ can be carried out using information theory. ■

Generalizations and related research

Naturally, the problem has been generalized in various ways and studied by many authors. In "Coin-Weighing Problems"[5], Guy and Nowakovski gave a great overview of the research in the area until 1990s with an extensive list of references. We list the most interesting variations and generalizations below.

Weight of counterfeit coin. Either it is known whether the counterfeit coin is lighter or heavier, or it is not. The first one allows for more generalizations due to its simpler nature but both problems have been heavily researched.

Number of counterfeit coins. In the most common case, there is exactly one counterfeit coin, which allows for natural generalizations. First, a variation

of Problem 2.2 with 2 or 3 counterfeit coins was studied[6][7], then with m counterfeit coins in general[8]. Some authors studied the problem for unknown number of counterfeit coins[9], or for *at most* m counterfeit coins[10].

Additional regular coin(s). In some cases, it may help if you are given an additional coin (or more coins), which is guaranteed not to be counterfeit. For example, for $n = 13$ in Problem 2.3, you need 4 weighings. However, if you are given this one extra coin, you can determine the solution in just 3 weighings[4].

Non-adaptive strategies. In this popular variation of the problem you have to announce all experiments in advance and then just collect the result. In other words, later weighings must not depend on the outcomes of the earlier weighings. Notice that the scheme constructed in the proof of Theorem 2.5 for $n = \frac{1}{2}(3^w - w)$ is indeed non-adaptive. However, the original proof uses an adaptive scheme for a smaller n. This was later updated, showing that there always exists an optimal scheme for Problem 2.3 which is non-adaptive[11].

Unreliable balance. This generalization introduces the possibility that one (or more) answers may be erroneous. The problem of errors/lies in general deductive games is well studied, see [12]. It was applied on the counterfeit coin problem (Problem 2.2 variant) in [13] with at most one erroneous outcome or in [14] with two.

Multi-pan balance scale. In this variation, your balance scale has k pans. You put the same number of coins on every pan and you get either the information that all weigh the same or which arm is lighter or heavier than others[15].

Parallel weighing. In this generalization, you have 2 (or k, in general) balance scales, you can weigh different coins on the two scales simultaneously and it counts as one experiment only[16]. The motivation here is that weighing takes significant time, you have more scales and strive to minimize the time the whole process takes.

2.2 Mastermind

Mastermind is a classic code-breaking board game for 2 players, invented by Mordecai Meirowitz in 1970. One player has the role of a *codemaker* and the other of a *codebreaker*. First, the codemaker chooses a secret code of n coloured pegs. Then a codebreaker tries to reveal the code by making guesses. The codemaker evaluates the guesses using black and white markers. Black markers correspond

to positions at which the code and the guess matches, a white marker means that some colour appears both in the code and in the guess, but at different positions. The markers in the answer are not ordered, so the codebreaker does not know, which marker correspond to which peg in the guess. Codebreaker's aim is to find out the code in minimal number of guesses.

More formally, let C be a set of colours of size c. Define a distance $d : C^n \times C^n \to \mathbb{N}_0 \times \mathbb{N}_0$ of two colour sequences by $d(u, v) = (b, w)$, where

$$b = |\{i \in \mathbb{N} \mid u[i] = v[i]\}|$$

$$w = \sum_{j \in C} \min\left(\left|\{i \mid u[i] = j\}\right|, \left|\{i \mid v[i] = j\}\right|\right) - b.$$

If the codemaker's secret code is h and the codebreaker's guess is g, the guess should be evaluated with b black pegs and w white pegs, where $(b, w) = d(h, g)$. Therefore, if the codebreaker have guessed $g_1, g_2, \ldots, g_k$ and the results were $(b_1, w_1), \ldots, (b_k, w_k)$, the search space is reduced to codes

$$\{u \in C^n \mid \forall i \leq k.\ d(u, g_i) = (b_i, w_i)\}.$$

Another way of looking at the guess evaluation is using *maximal matching* of the pegs in the code h and the guess g. A matching is a set of pair-wise non-adjacent edges between the pegs in the guess (represented by $i_\bullet$ for $1 \leq i \leq n$) and the pegs in the code (represented by $i^\bullet$ for $1 \leq i \leq n$). Let M be a maximal matching such that

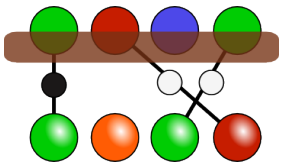

Figure 2.7: Guess evaluation by maximal matching.

1. an edge connects only pegs of the same colour, i.e. if $(i_\bullet, j^\bullet) \in M$, then $h[i] = g[j]$, and
2. if $h[i] = g[i]$ then $(i_\bullet, i^\bullet) \in M$.

Maximal means that no edge can be added without breaking one of the conditions. The edges in M correspond to the markers in the response, a marker being black if and only if the corresponding edge connects $i_\bullet$ with $i^\bullet$ for some i.

Known results and related research

Much research has been done on this game, authors focusing on *exact values*, *asymptotics* (e.g. [17]), or computer generated strategies. One of the fundamental theoretical results is that *Mastermind satisfiability problem*, asking whether there exists at least one valid solution, given a set of guesses and their scores, is NP-complete[18].

When focusing on strategy synthesis, the goal is either to minimize *the maximal number of guesses* or *the expected number of guesses*, given that the code is selected from the set of possible codes with uniform distribution. These two problems are quite different and strategies performing well in one case may perform poorly in the other.

Knuth[19] proposes a strategy that chooses a guess that minimizes the maximal number of remaining possibilities over all possible responses by the codemaker. In the following, we call this strategy "max-models". In the worst-case, it requires 5 guesses in the standard $n = 4$, $c = 6$ variant, which can be shown optimal. In the average case, the strategy makes 4.48 guesses.

Other authors proposed other *one-step look-ahead* strategies. Irving[20] suggested minimizing the expected number of remaining possibilities (exp-models), Neuwirth[21] maximized the entropy of the number of possibilities in the next round (ent-models). Much later, Kooi[22] came up with a simple strategy that maximizes the number of possible responses by the codemaker (parts), which is computationally easier and performs better that the previous two.

Using a backtracking algorithm, Koyama and Lai[23] found the optimal strategy for the expected case, which performs 4.34 guesses on average. The comparison of the described strategies is shown in Table 2.8.

Strategy	First guess	Average-case	Worst-case
Max-models	AABB	4.476	5
Exp-models	AABC	4.395/4.626[2]	6
Ent-models	ABCD	4.416/4.643[3]	6
Parts	AABC	4.373	6
Avg-case optimal	AABC	4.340	6

Table 2.8: Comparison of one-step look-ahead strategies. Data from [24] and [22].

Apart from *one-step look-ahead* strategies, which do not scale very well for bigger n or c, other approaches has been suggested. Many authors tried to apply genetic algorithms (see [25] for an exhaustive overview and references therein), other analysed various heuristic methods (e.g. [26]).

2. Irwing's paper reports 4.395 as the expected number of experiments of this strategy. However, he states that his strategy selects the first two experiments on the basis of the expected number of models and the rest is done by exhaustive search. We were not able to reproduce this result and the paper contains several other irreproducible results, which has been already pointed out in [22]. The number reported by our tool when using this strategy only is 4.626.

3. We were not able to reproduce the result of Neuwirth as he does not always follow the choice of the strategy as described. The number reported by our tool is 4.643.

Variations and applications

Bulls and Cows is an old game with a principle very similar to Mastermind. The only difference is that it uses digits instead of colours and does not allow repetitions. Slovesnov wrote an exhaustive analysis of the problem, see [27].

Static Mastermind is a variation of the game in which all guesses must be made in one go. The codebreaker prepares a set of guesses, then the codemakers evaluates all of them as usual and the codebreaker must determine the code from the outcomes. This variation was introduced by Chvátal[17] and partially solved (for $n \leq 4$) by Goddard[28], proving that for 4 pegs and k colours, the optimal strategy uses $k-1$ guesses. Note that this corresponds to so-called *non-adaptive* strategies for the Counterfeit Coin problem.

String matching, also called *Mastermind with black-markers only* is a variation without white markers, i.e. you make guesses and the only information you get is the number of positions at which your guess is correct. This problem was already studied by Erdős[29], who gave some asymptotic results about the worst-case number of guesses. Later, this problem found an application in genetics with a need of methods to select a subset of genotyped individuals for phenotyping [30][31].

Extended Mastermind was introduced by Focardi and Luccio, who showed that it is strictly related to cracking bank PINs for accessing ATMs by so-called *decimalization attacks*[32]. In this variation, a guess is not just a sequence of colours, but a sequence of sets of colours. For example, if we have six colours $\{A, B, C, D, E, F\}$ and the code is $AECA$, you can make a guess $\{A\}, \{C, D, E\}, \{A, B\}, \{F\}$, which will be awarded two black markers (for the first two positions) and one white marker (for A guessed at position 3).

2.3 Other games

Black Box

Black Box is a code-breaking board game in which one player creates a puzzle by placing four marbles on a 8×8 grid. The other player's goal is to discover their positions by the use of "rays". The codebreaker chooses a side of the grid and an exact row/column, in which the ray enters the grid (thus having 32 choices). For each ray, the codemaker announces the position, where the ray emerged from the grid, or says "hit", if the ray directly hit a marble[33].
The marbles interact with rays in three ways:

Hit. If a ray fired into the grid directly strikes a marble, the result is "hit" and the ray does not emerge from the box.

Deflection. If a ray does not directly strike a marble, but it should pass to one side of a marble, the ray is "deflected" and changes its direction by 90 degrees.

Reflection. If a ray should enter a cell with marbles on both sides, than it is "reflected" and returns back the same way it came. The same happens if a marble is at the edge of the grid and a ray is fired from a position next to it (so that it should be deflected even before entering the box according to the second rule).

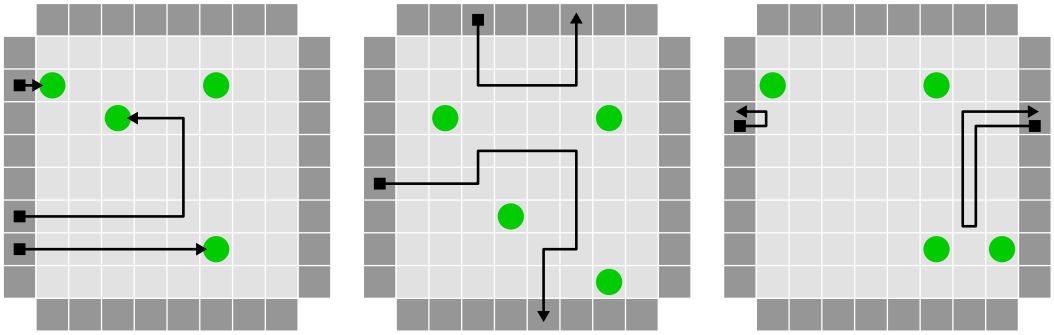

Figure 2.9: Illustration of the rules of Black Box game[4].

A few examples are shown in Figure 2.9. The first image shows cases in which the ray hits the marble, the second shows rays deflected multiple times, emerging from the box at a different place, and the third demonstrates the two cases in which reflection happens.

Note that if the game is played with 5 or more marbles, they can be placed in the grid so that their position can not be uniquely determined. Figure 2.10 shows an example of such problematic configuration.

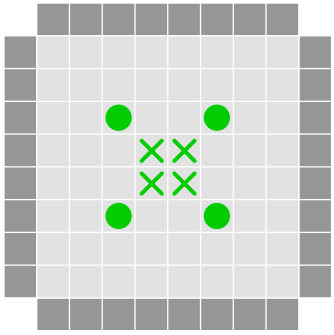

Figure 2.10: An example of ambigous configuration[4].

Although Black Box is an interesting example of a code breaking game, there are configurations for which the codebreaker has to fire a ray from all positions to discover the marbles

4. Images adopted from http://en.wikipedia.org/wiki/Black_Box_(game) under GFDL 1.2. with minor modifications.

(and, for 5 or more marbles, it may even be impossible), which makes the game uninteresting from a research point of view.
However, the game has become a popular puzzle for children and its principle has been used in other board games such as *Laser maze*[34].

Code 777

During the board game named *Code 777*, players sit in a circle, each drawing three cards at the beginning. Players must not look at their own cards but they put them to a rack in front of them so that other players can see them. Each card has one of seven colours and contains a number from one to seven. The goal of the game is to determine which cards you have, using questions like "Do you see more yellow sevens or blue fives?", which the others answer[35].
We can reformulate this as a code-breaking game, in which a player receives some cards, each having several attributes, each of which can have multiple values. A player's goal is to determine his cards using questions like "Do I have more [A] or [B]", where [A] and [B] are conditions on any subset of attributes. For example, if the attributes are number, colour, and shape, one can ask "Do I have more triangles or green twos?".

Bags of gold

Imagine you have 10 bags of gold coins and you know that all coins in one bag are the same. You were tipped off that some of the bags may contain counterfeit coins, which weigh 9 grams instead of 10 but are indistinguishable otherwise. Suppose all coins in one bag are the same. You have a digital scale that can show you exact weight of a set of coins. How to find out which bags contain counterfeit coin in the minimal number of weighings? Suppose there is a sufficient number of coins in each bag.
In the original version of this riddle, the scale has unlimited capacity and there is only one bag of counterfeit coins. In that case, the secret can be determined in a single experiment. You take one coin from the first bag, two coins from the second and so on up to 10 coins from the last. You put all those 55 coins on the scale and, if they are all good, they weigh 550 grams. If the weight is by x grams lower, you know that there are x counterfeit coins in your set and, therefore, the x-th bag is the one with counterfeit coins.
The game gets more interesting if the capacity of the scale is limited, or if we have more bags and the number of coins in them is limited. A special case, in which each bag contains a single coin is studied in [29], and is shown to be similar to the string matching problem (Mastermind with black-markers only). Otherwise, the game lives only in a form of a logic puzzle and, to the best of our knowledge, no general results have been found.

3 Code-breaking game model

In this chapter, we formally define code-breaking games within the framework of propositional logic, where we represent a secret code as a valuation of propositional variables. We define strategies in general, study several strategy classes and introduce *one-step look-ahead strategies*.

3.1 Notation and terminology

Symbols $\mathbb{N}_0$ and $\mathbb{N}$ denote the set of natural numbers with and without zero; the set of real numbers is denoted by $\mathbb{R}$. The number of elements of a set X is denoted by $|X|$. Notation $X^* = \cup_{i \in \mathbb{N}_0} X^i$ is used to denote the set of finite sequences of elements of X. The k-th element of a sequence $s \in X^*$ is denoted by $s[k]$.

The set of all permutations of a set X (bijections $X \to X$) is denoted by PERM_X and ID_X is the identity permutation. A *partition* P of a set X is a set of disjoint subsets of X, union of which is equal to X. Members of P are called *cells*. Let $P(x)$ be the cell containing x, i.e. $P(x) = A$, where $A \in P$ and $x \in A$. For a function $f : X \to Y$ and a set $Z \subseteq X$, the *restriction of f to Z* is denoted by $f|_Z : Z \to Y$.

Let FORM_X denote the set of *propositional formulas* over the set of variables X and let VAL_X be the set of *valuations* (boolean interpretations) of variables X. Apart from standard logical operators, we allow n-ary *numerical operators* EXACTLY_k, ATLEAST_k, ATMOST_k. For a valuation $v \in \text{VAL}_X$ and propositional formulas $\varphi_1, \ldots, \varphi_n \in \text{FORM}_X$, the operator EXACTLY_k has the semantics $v(\text{EXACTLY}_k(\varphi_1, \ldots, \varphi_n)) = 1$ if and only if $|\{i \mid v(\varphi_i) = 1\}| = k$. The semantics of ATMOST and ATLEAST is defined analogically.

Formulas $\varphi_0, \varphi_1 \in \text{FORM}_X$ are *equivalent*, written $\varphi_0 \equiv \varphi_1$, if $v(\varphi_0) = v(\varphi_1)$ for all $v \in \text{VAL}_X$. We say that v *is a model of* φ or that v *satisfies* φ if $v(\varphi) = 1$. For a formula $\varphi \in \text{FORM}_X$, let $\#_X\varphi = |\{v \in \text{VAL}_X \mid v(\varphi) = 1\}|$ be the number of models of φ. We often omit the index X if it is clear from the context. A *fixed variable* x of a formula φ is a variable that is assigned the same value in all models of φ. If $v(x) = 1$ for all $v \in \text{VAL}$ such that $v(\varphi) = 1$, we that that x is fixed to 1 (or *true*). Similarly, if $v(x) = 0$, we say that x is fixed to 0 (or *false*).

3.2 Basic definitions

A code-breaking game can be represented by a *set of variables*, *initial constraint* (a formula that is guaranteed to be satisfied), and a set of *allowed experiments*. An experiment is defined by the set of possible outcomes in which it can result. The outcomes are specified in the form of a propositional formula that represents

the partial information that the codebreaker gains if the experiment results in the outcome.

The number of experiments in a code-breaking game is typically very large. For example, in the counterfeit coin problem defined in Section 2.1, experiments correspond to combinations of coins you put on the pans of the balance scale. It can be calculated that there are 36,894 combinations for 12 coins. However, most of them have the same structure, so it would be inefficient to specify them one by one. Therefore we choose a compact representation with *parametrized experiments*, where parametrization is a fixed-length string over a defined alphabet. This whole idea is formalized below.

Definition 3.1 (Code-breaking game). A *code-breaking game* is a quintuple $\mathcal{G} = (X, \varphi_0, \Sigma, F, T)$, where

- X is a finite set of propositional variables,
- $\varphi_0 \in \text{FORM}_X$ is a satisfiable propositional formula,
- Σ is a finite alphabet,
- F is a collection of mappings of type $\Sigma \to X$ with pairwise disjoint images,
- T is a set of *parametrized experiments*, defined below.

Definition 3.2 (Parametrized experiment). A *parametrized experiment* for a game $\mathcal{G} = (X, \varphi_0, \Sigma, F, T)$ is a triple $t = (n, P, \Phi)$, where

- n is the number of parameters of the experiment,
- P is a partition of the set $\{1, \ldots, n\}$,
- Φ is a set of parametrized formulas, defined below.

If k and l are in the same cell of the partition P, then the k-th and the l-th parameter must be different. We denote the components of a parametrized experiment $t \in T$ by n_t, P_t, and Φ_t.

Definition 3.3 (Parametrized formula). A *parametrized formula* for a parametrized experiment t of a game $\mathcal{G} = (X, \varphi_0, \Sigma, F, T)$ is a string ψ generated by the following grammar, specified in Backus–Naur Form.

$$\langle\text{form}\rangle ::= x \mid f(\$k) \mid \langle\text{form}\rangle \circ \langle\text{form}\rangle \mid O(\langle\text{form-list}\rangle) \mid \neg\, \langle\text{form}\rangle,$$
$$\langle\text{form-list}\rangle ::= \langle\text{form-list}\rangle\,,\ \langle\text{form}\rangle \mid \langle\text{form}\rangle$$

where $x \in X$ is a propositional variable, $f \in F$ is a mapping, $1 \le k \le n_t$ is a parameter index, $\circ \in \{\wedge, \vee, \Rightarrow\}$ is a standard logical operator, and $O \in \{\text{EXACTLY}_k\,, \text{ATMOST}_k\,, \text{ATLEAST}_k \mid k \in \mathbb{N}\}$ is a numerical operator. The special notation $\$k$ in $f(\$k)$ is used to denote the k-th parameter.

The set E of all experiments in the game $\mathcal{G}$ is given by

$$E = \left\{(t,p) \mid t \in T,\ p \in \Sigma^{n_t},\ \forall x,y \leq n_t :\ P_t(x) = P_t(y) \Rightarrow p[x] \neq p[y]\right\}$$

An experiment $e \in E$ is thus a pair (t,p), where t is referred to as the *type of the experiment*, and p is referred to as its *parametrization*.
Let $e = (t,p) \in E$ be an experiment, and $\psi \in \Phi_t$ a parametrized formula. By $\psi(p)$ we denote the application of the parametrization p on ψ, which is defined recursively on the structure of ψ in the following way:

$$\begin{aligned}(x)(p) &= x,\\ (f(\$k))(p) &= f(p[k]),\\ (\psi_1 \circ \psi_2)(p) &= \psi_1(p) \circ \psi_2(p),\\ O(\psi_1,\ldots,\psi_m)(p) &= O(\psi_1(p),\ldots,\psi_m(p)),\\ (\neg\psi)(p) &= \neg(\psi(p)).\end{aligned}$$

To simplify the notation, let us denote the set of possible outcomes for an experiment $e = (t,p) \in E$ by $\Phi(e) = \{\psi(p) \mid \psi \in \Phi_t\}$.

Example 3.4. Consider the counterfeit coin problem for 4 coins. We use this game as a running example throughout this chapter.
The counterfeit coin and its relative weight to others can be encoded as a valuation of variables x_1, x_2, x_3, x_4 and y, $v(x_i)$ being 1 if and only if the i-th coin is counterfeit and y determining its relative weight ($v(y) = 0$ meaning lighter, $v(y) = 1$ meaning heavier). The initial constraint φ_0 should capture the restriction that exactly one coin if counterfeit. Therefore, let φ_0 be $\textsc{exactly}_1(x_1, x_2, x_3, x_4)$.
The experiments are parametrized by the coins on the pans of the balance scale. Let $\Sigma = \{1,2,3,4\}$ and $F = \{f_x\}$ where f_x maps i to the corresponding variable x_i.
The first parametrized experiment t is weighing one coin against one. We need two parameters ($n_t = 2$), the first determining the coin on the left pan, the second determining the coin on the right pan that must be different from the first. P_t is therefore the trivial partition $\{\{1,2\}\}$.
If the left pan is lighter, it is either the case that the coin on the left is underweight ($f_x(\$1) \wedge \neg y$) or the coin on the right is overweight ($f_x(\$2) \wedge y$). If the right pan is lighter, we get symmetrical knowledge $(f_x(\$1) \wedge y) \vee (f_x(\$2) \wedge \neg y)$. If both sides weigh the same, the counterfeit coins is not present on either pan and we can conclude $\neg f_x(\$1) \wedge \neg f_x(\$2)$. To sum it up,

$$\begin{aligned}t = \big(2,\ \big\{\{1,2\}\big\},\ \big\{&(f_x(\$1) \wedge \neg y) \vee (f_x(\$2) \wedge y),\\ &(f_x(\$1) \wedge y) \vee (f_x(\$2) \wedge \neg y),\\ &\neg f_x(\$1) \wedge \neg f_x(\$2)\ \big\}\big).\end{aligned}$$

The second parametrized experiment is weighing two coins against two. There are 4 parameters, they must be pairwise distinct and the outcome formulas can be constructed analogically. ⧫

Note that the compact representation with parametrized experiments does not restrict the class of games that can fit Definition 3.1, compared to a possible definition with direct experiment enumeration. The thing is that there can always be a parametrized experiment with no parameters for each actual experiment.

Definition 3.5 (Solving process). An *evaluated experiment* is a pair (e, φ), where $e \in E$ and $\varphi \in \Phi(e)$. Let us denote the set of evaluated experiments by Ω. A *solving process* is a finite or infinite sequence of evaluated experiments.

Let λ be a solving process. For simplicity, we omit parentheses around the pairs and write $\lambda = e_1, \varphi_1, e_2, \varphi_2, \ldots$. Let

- $|\lambda|$ denote the length of the sequence,
- $\lambda(k) = e_k$ denote the k-th experiment,
- $\lambda[k] = \varphi_k$ denote the k-th outcome,
- $\lambda[1:k] = e_1, \varphi_1, \ldots, e_k, \varphi_k$ denote the prefix of length k, and
- $\lambda\langle k \rangle = \varphi_0 \wedge \varphi_1 \wedge \ldots \wedge \varphi_k$ denote the accumulated knowledge after the first k experiments (including the initial constraint φ_0). For finite λ, let $\lambda\langle\rangle = \lambda\langle|\lambda|\rangle$ be the overall accumulated knowledge.

We denote the set of valuations that satisfy φ_0 by $\textsc{val}' = \{v \in \textsc{val}_X \mid v(\varphi_0) = 1\}$ and the set of *reachable formulas* by $\textsc{form}' = \{\lambda\langle\rangle \mid \lambda \in \Omega^*\}$.

Course of the game

Let us now describe the course of the game in the defined terms.

1. The codemaker chooses a valuation v from $\textsc{val}'$.
2. The codebreaker chooses an experiment e from E.
3. The codemaker gives the codebreaker a formula $\varphi \in \Phi(e)$ that is satisfied by valuation v. In order for the codemaker to always be able to do so, there must be a formula $\varphi \in \Phi(e)$ satisfied by any valuation in $\textsc{val}'$. This is defined below as *well-formed* property of the game.
4. The evaluated experiment (e, φ) is appended to the (initially empty) solving process λ.
5. If $\#\lambda\langle\rangle = 1$, the codebreaker can uniquely determine the valuation v and the game ends. Otherwise, it continues with step 2.

Definition 3.6 (Well-formed game). A code-breaking game is *well-formed* if for all $e \in E$,

$$\forall v \in \text{VAL}'.\ \exists \text{ exactly one } \varphi \in \Phi(e)\ .\ v(\varphi) = 1$$

In the sequel, we focus only on well-formed games and, by default, we assume a game is well-formed unless otherwise stated.

Examples

In the rest of this section, we show two ways of defining the counterfeit coin problem and a formal definition of Mastermind. We do not provide formal definitions of other code-breaking games presented in Chapter 2, however, a computer language for game specification that is based on this formalism is introduced in Chapter 5, and specifications of all the code-breaking games in this language can be found in the electronic attachment to the thesis.

Example 3.7 (The counterfeit coin problem). A formal definition of the counterfeit coin problem with 4 coins has already been introduced in Example 3.4. This is a straightforward generalization for n coins. We define a game $\mathcal{F}_n = (X, \varphi_0, \Sigma, F, T)$ with the following components.

- $X = \{x_1, x_2, \ldots, x_n, y\}$. Variable x_i tells whether coin i is counterfeit, variable y tells whether it is lighter or heavier.
- $\varphi_0 = \text{EXACTLY}_1\,(x_1, \ldots, x_n)$, saying that exactly one coin is counterfeit.
- $\Sigma = \{1, 2, \ldots, n\}$, $F = \{f_x\}$, where $f_x(i) = x_i$. The experiments are parametrized with coins that are represented by numbers from 1 to n.
- $T = \big\{(2 \cdot m,\ \{\{1, \ldots, 2m\}\},\ \Phi_m) \mid 1 \le m \le n/2\big\}$, where

$$\begin{aligned}\Phi_m = \big\{&((f_x(\$1) \vee \ldots \vee f_x(\$m)) \wedge \neg y) \vee ((f_x(\$m+1) \vee \ldots \vee f_x(\$2m)) \wedge y),\\ &((f_x(\$1) \vee \ldots \vee f_x(\$m)) \wedge y) \vee ((f_x(\$m+1) \vee \ldots \vee f_x(\$2m)) \wedge \neg y),\\ &\neg(f_x(\$1) \vee \ldots \vee f_x(\$2m))\big\}.\end{aligned}$$

  For every $m \in \mathbb{N}$, $m \le n/2$, we have a parametrized experiment of weighing m coins against m coins. It has $2m$ parameters, the first m are put on the left pan, the last m are put on the right pan.
  There are 3 possible outcomes. First, the left pan is lighter. This happens if the counterfeit coin is lighter and it appears among the first m parameters, or if it is heavier and it appears among the last m parameters. Second, analogically, the right pan is lighter. Third, both pans weigh the same if the counterfeit coin does not participate in the experiment.

For demonstration purposes, we show another possible formalization of the same problem. Let $\mathcal{F'}_n = (X, \varphi_0, \Sigma, F, t)$ be a game with the following components.

- $X = \{x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n\}$. Variable x_i tells that coin i is lighter, variable y_i tells that coin i is heavier.
- $\varphi_0 = \text{EXACTLY}_1(x_1, \ldots, x_n, y_1, \ldots, y_n)$, saying that exactly one coin is odd-weight.
- $\Sigma = \{1, 2, \ldots, n\}$, $F = \{f_x, f_y\}$, where $f_x(i) = x_i$, $f_x(i) = y_i$.
- $T = \big\{(2 \cdot m,\ \{\{1, \ldots, 2m\}\},\ \Phi_m) \mid 1 \le m \le n/2\big\}$, where

  $$\Phi(w_m) = \big\{f_x(\$1) \vee \ldots \vee f_x(\$m) \vee f_y(\$m+1) \vee \ldots \vee f_y(\$2m),$$
  $$f_y(\$1) \vee \ldots \vee f_y(\$m) \vee f_x(\$m+1) \vee \ldots \vee f_x(\$2m),$$
  $$\neg\,(f_x(\$1) \vee \ldots \vee f_x(\$2m) \vee f_y(\$1) \vee \ldots \vee f_y(\$2m))\big\}.$$

  In this formalization, the variables correspond one-to-one to possible codes, so the outcome formulas effectively list all possibilities. ⧫

Example 3.8 (Mastermind). Mastermind game with n pegs and m colours can be formalized as a code-breaking game $\mathcal{M}_{n,m} = (X, \varphi_0, \Sigma, F, T)$ with the following components.

- $X = \{x_{i,j} \mid 1 \le i \le n, 1 \le j \le m\}$. Variable $x_{i,j}$ tells whether there is colour j at position i.
- $\varphi_0 = \bigwedge \{\text{EXACTLY}_1 \{x_{i,j} \mid 1 \le j \le m\} \mid 1 \le i \le n\}$, saying that there is exactly one colour at each position.
- $\Sigma = \{1, \ldots, m\}$,
  $F = \{f_1, \ldots, f_n\}$, where $f_i(c) = x_{i,c}$ for $1 \le i \le n$,
  $T = \{(n, P, \Phi)\}$.
  There is only one parametrized experiment with n parameters corresponding to the colours. All parameters can be the same, so the partition P is the discrete partition $\{\{1\}, \ldots, \{n\}\}$.
- $\Phi = \{\texttt{Outcome}(b, w) \mid 0 \le b \le n, 0 \le w \le n, b+w \le n\}$, where `Outcome` function is computed by the algorithm described below.

As described in Section 2.2, the outcome of an experiment corresponds to some maximal matching between the pegs in the code and the pegs in the guess. The idea here is to generate a formula that asserts existence of such maximal matching with b edges corresponding to black markers and w edges corresponding to white markers.

The computation of `Outcome` (b, w) is performed as follows. First, we generate all admissible matchings. Let $P = \{1, 2, \ldots, n\}$ be the set of positions.

- Select $B \subseteq P$ such that $|B| = b$. These are the positions at which the colour in the code matches the colour in the guess. They correspond to the black markers.

- Select $W \subseteq P \times P$ such that $|W| = w$, $p_1(W) \cap B = \varnothing$, and $p_2(W) \cap B = \varnothing$, where p_1, p_2 are projections. These correspond to the white markers; $(i, j) \in W$ means that the colour at position i in the guess is at position j in the code.

Recall that $i_\bullet$ denotes position i in the guess and $i^\bullet$ denotes position i in the code. For a fixed combination (B, W), we define matchin M by $M = \{(i_\bullet, i^\bullet) \mid i \in B\} \cup \{(i_\bullet, j^\bullet) \mid (i, j) \in W\}$. We construct a parametrized formula that asserts that M is the maximal matching satisfying conditions in Section 2.2 for a guess $\$1, \$2, \ldots, \$n$ and the code given by a valuation of the variables. The formula has a form of a conjunction constructed in the following way.

- For $i \in B$, we add $f_i(\$i)$. This asserts that $(i_\bullet, i^\bullet)$ is an edge in the matching.
- For $(i, j) \in W$, we add $f_j(\$i) \wedge \neg f_i(\$i) \wedge \neg f_j(\$j)$. This asserts that the colour $\$i$ is at position j in the code and that $(i_\bullet, i^\bullet)$, $(j_\bullet, j^\bullet)$ cannot be edges in the matching.
- For $(i, j) \in (P \setminus B \setminus p_1(W)) \times (P \setminus B \setminus p_2(W))$, we add $\neg f_j(\$i)$. This asserts that no edge can be added and the matching is maximal.

The result of $\texttt{Outcome}(b, w)$ is a disjunction of all the conjunctions constructed in this way for all combinations of B and W. For example, for $n = 4$, $B = \{1\}$ and $W = \{2, 3\}$, the generated formula is

$$f_1(\$1) \wedge f_3(\$2) \wedge \neg f_2(\$2) \wedge \neg f_3(\$3) \wedge \neg f_2(\$3) \wedge \neg f_2(\$4) \wedge \neg f_4(\$3) \wedge \neg f_4(\$4).$$

The number of combinations for B and W grows exponentially with n and so does the size of generated formulas. For $n = 4$, the result of $\texttt{Outcome}(1, 1)$ contains 24 clauses at the top level with 192 literals in total. ⧫

3.3 Strategies in general

This section introduces the concept of a strategy for experiment selection. We define worst-case and average-case number of experiments of a strategy and optimal strategies. Further, we examine several strategy classes.

Definition 3.9 (Strategy). A *strategy* is a function $\sigma : \Omega^* \to E$, determining the next experiment for a given finite solving process.

A strategy σ together with a valuation $v \in \text{VAL}'$ induce an infinite solving process

$$\lambda_v^\sigma = e_1, \varphi_1, e_2, \varphi_2, \ldots,$$

where $e_{i+1} = \sigma(e_1, \varphi_1, \ldots, e_i, \varphi_i)$ and φ_{i+1} is the formula from $\Phi(e_{i+1})$ satisfied by v, for all $i \in \mathbb{N}$. Note that thanks to the well-formed property, φ_{i+1} is uniquely defined.

We define *length* of a strategy σ on a valuation v, denoted $|\sigma|_v$, as the smallest $k \in \mathbb{N}_0$ such that $\lambda_v^\sigma\langle k\rangle$ uniquely determines the code, i.e.

$$|\sigma|_v = \min\{k \in \mathbb{N}_0 \mid \#\lambda_v^\sigma\langle k\rangle = 1\}$$

The *worst-case number of experiments* Λ^σ of a strategy σ is the maximal length of the strategy on a valuation v, over all $v \in \text{VAL}'$, i.e.

$$\Lambda^\sigma = \max_{v \in \text{VAL}'} |\sigma|_v.$$

The *average-case number of experiments* $\Lambda^\sigma_{\text{exp}}$ of a strategy σ is the expected number of experiments if the code is selected from models of φ_0 with uniform distribution, i.e.

$$\Lambda^\sigma_{\text{exp}} = \frac{\sum_{v \in \text{VAL}'} |\sigma|_v}{\#\varphi_0}.$$

We say that a strategy σ *solves the game* if Λ^σ is finite. Note that Λ^σ is finite if and only if $\Lambda^\sigma_{\text{exp}}$ is finite. The game is *solvable* if there exists a strategy that solves the game.

Definition 3.10 (Optimal strategy). A strategy σ is *worst-case optimal* if $\Lambda^\sigma \leq \Lambda^{\sigma'}$ for any strategy σ'. A strategy σ is *average-case optimal* if $\Lambda^\sigma_{\text{exp}} \leq \Lambda^{\sigma'}_{\text{exp}}$ for any strategy σ'.

The following lemma provides us with a lower bound on the number of experiments of a worst-case optimal strategy.

Lemma 3.11. *Let $b = \max_{t \in T} |\Phi(t)|$ be the maximal number of possible outcomes of an experiment. Then for every strategy σ,*

$$\Lambda^\sigma \geq \lceil \log_b(\#\varphi_0) \rceil.$$

Proof. Let us fix a strategy σ and $k = \Lambda^\sigma$. For an unknown model v of φ_0, $\lambda_v^\sigma\langle k\rangle$ can take up to b^k different values. By pigeon-hole principle, if $\#\varphi_0 > b^k$, there must be a valuation v such that $\#\lambda_v^\sigma\langle k\rangle > 1$. This would be a contradiction with $k = \Lambda^\sigma$ and, therefore, $\#\varphi_0 \leq b^k$, which is equivalent with the statement of the lemma. ∎

Lemma 3.12. *Let σ be a strategy and let $v_1, v_2 \in \text{VAL}'$. If v_1 is a model of $\lambda_{v_2}^\sigma\langle k\rangle$, then $\lambda_{v_1}^\sigma[1:k] = \lambda_{v_2}^\sigma[1:k]$.*

Proof. Let $\lambda_1 = \lambda^\sigma_{v_1}$, $\lambda_2 = \lambda^\sigma_{v_2}$ and consider the first place where λ_1 and λ_2 differs. It cannot be an experiment $\lambda_1(i) \neq \lambda_2(i)$ as they are both values of the same strategy on the same process: $\lambda_1(i) = \sigma(\lambda_1[1:i-1]) = \sigma(\lambda_2[1:i-1]) = \lambda_2(i)$.
Suppose it is an outcome of the i-th experiment, $\lambda_1[i] \neq \lambda_2[i]$ and $i \leq k$. Since v_1 satisfies $\lambda_2\langle k\rangle$ and $i \leq k$, it satisfies $\lambda_2[i]$ as well. However, v_1 always satisfies $\lambda_1[i]$ and both $\lambda_1[i]$ and $\lambda_2[i]$ are from the set $\Phi(\lambda_1(i)) = \Phi(\lambda_2(i))$. Since there is exactly one satisfied experiment for each valuation in the set, $\lambda_1[i]$ and $\lambda_2[i]$ must be the same. Contradiction. ■

Example 3.13. Recall our running example of the counterfeit coin problem with 4 coins, as defined in 3.13.
Consider a strategy σ defined as follows. For simplicity, we denote experiments by their parametrizations only and the outcomes by a symbol <, > and =, instead of the corresponding formula.

$$\sigma(\lambda) = \begin{cases} 13 & \text{if } \lambda = (12,<), \\ 23 & \text{if } \lambda = (12,>), \\ 14 & \text{if } \lambda = (12,=),(12,=), \\ 34 & \text{if } \lambda = (12,=),(12,=),(14,=), \\ 12 & \text{otherwise.} \end{cases}$$

Let $v \in \textsc{val}'$ be a valuation such that $v(x_3) = v(y) = 1$. The induced solving process is

$$\lambda^\sigma_v = (12,=),(12,=),(14,=),(34,>),(12,=),(12,=),\ldots$$

The length of σ on v is 4, because v is the only model of the accumulated knowledge after 4 experiments,

$$\textsc{exactly}_1(x_1,x_2,x_3,x_4) \wedge \neg(x_1 \vee x_2) \wedge \neg(x_1 \vee x_2) \wedge \neg(x_1 \vee x_4) \wedge ((x_3 \wedge y) \vee (x_4 \wedge \neg y)).$$

The strategy is intentionally inefficient and repeats the experiment 12 if the outcome in the first step is '='. In fact, every valuation is discovered by σ in at most 4 experiments, so $\Lambda^\sigma = 4$.
Lemma 3.11 gives us a lower bound $\lceil \log_3(8) \rceil = 2$ on the worst-case number of experiments of an optimal strategy. However, we already know from Theorem 2.5 that the minimal number of experiments needed to reveal the code is 3.

Non-adaptive strategies

Non-adaptive strategies correspond to the well-studied problems of static Mastermind and non-adaptive strategies for the counterfeit coin problem[28][11]. We define them here only to show the possibility of formulating the corresponding problems in our framework but we do not study them any further.

Definition 3.14 (Non-adaptive strategy). A strategy σ is *non-adaptive* if it decides the next experiment based on the length of the solving process only, i.e. whenever λ_1 and λ_2 are processes such that $|\lambda_1| = |\lambda_2|$, then $\sigma(\lambda_1) = \sigma(\lambda_2)$.
Non-adaptive strategies can be considered functions $\tau : \mathbb{N}_0 \to E$, where $\tau(|\lambda|) = \sigma(\lambda)$.

Memory-less strategies

The general definition of a strategy allows for the next experiment to depend on the exact history of the solving process, not only on the accumulated knowledge. This is in a sense unintuitive, as the nature of code-breaking games is memory-less and the course of a game depends only on the accumulated knowledge.

Definition 3.15 (Memory-less strategy). A strategy σ is *memory-less* if it decides the next experiment based on the accumulated knowledge only, i.e. whenever λ_1 and λ_2 are processes such that if $\lambda_1\langle\rangle \equiv \lambda_2\langle\rangle$ then $\sigma(\lambda_1) = \sigma(\lambda_2)$.
Memory-less strategies can be considered functions $\tau : \textsc{form}' \to E$ such that $\varphi_1 \equiv \varphi_2 \Rightarrow \tau(\varphi_1) = \tau(\varphi_2)$. Then $\sigma(\lambda) = \tau(\lambda\langle\rangle)$.

Note that the number of non-equivalent formulas over variable X is finite and, therefore, the number of memory-less strategies for a fixed code-breaking game is finite as well.
Now we prove some basic properties of memory-less strategies. The following lemma says that once we do not get any new information from the experiment selected by a experiment, we never get any new information with the strategy. Then, the theorem below proves that there exists an optimal memory-less strategy.

Lemma 3.16. *Let σ be a memory-less strategy and $v \in \textsc{val}'$. If there exists $k \in \mathbb{N}$ such that $\#\lambda_v^\sigma\langle k\rangle = \#\lambda_v^\sigma\langle k+1\rangle$, then $\#\lambda_v^\sigma\langle k\rangle = \#\lambda_v^\sigma\langle k+l\rangle$ for any $l \in \mathbb{N}$.*

Proof. To simplify the notation, let $\alpha^k = \lambda_v^\sigma\langle k\rangle$. There is a formula $\varphi \in \Phi(\alpha^k)$, such that $\alpha^{k+1} \equiv \alpha^k \wedge \varphi$. Therefore, if α^{k+1} is satisfied by valuation v, so must be α^k. Since $\#\alpha^k = \#\alpha^{k+1}$, the sets of valuations satisfying α^k and α^{k+1} are exactly the same and the formulas are thus equivalent. This implies $\sigma(\alpha^k) = \sigma(\alpha^{k+1})$ and $\alpha^{k+2} \equiv \alpha^{k+1} \wedge \varphi \equiv \alpha^{k+1}$.
By induction, $\sigma(\alpha^{k+l}) = \sigma(\alpha^k)$ and $\alpha^{k+l} \equiv \alpha^k$ for any $l \in \mathbb{N}$. ∎

Theorem 3.17. *Let σ be a strategy. Then there exists a memory-less strategy τ such that $|\sigma|_v \geq |\tau|_v$ for all $v \in \text{VAL}'$.*

Proof. Let us choose any total order $\varphi_1, \varphi_2, \ldots$ of FORM' such that if φ_i implies φ_j, then $i \leq j$. We build a sequence of strategies $\sigma_0, \sigma_1, \sigma_2, \ldots$ inductively in the following way. Let $\sigma_0 = \sigma$.

- If there is no $v \in \text{VAL}', k \in \mathbb{N}_0$ such that $\lambda_v^{\sigma_{i-1}}\langle k \rangle \equiv \varphi_i$, select any $e \in E$ and define σ_i by
$$\sigma_i(\lambda) = \begin{cases} \sigma_{i-1}(\lambda) & \text{if } \lambda\langle\rangle \not\equiv \varphi_i, \\ e & \text{if } \lambda\langle\rangle \equiv \varphi_i. \end{cases}$$
Clearly, all induced solving processes for σ_i and σ_{i-1} are the same and $|\sigma_i|_v = |\sigma_{i-1}|_v$.
- If there exists $v \in \text{VAL}', k \in \mathbb{N}_0$ such that $\lambda_v^{\sigma_{i-1}}\langle k \rangle \equiv \varphi_i$, choose the largest l such that $\lambda_v^{\sigma_{i-1}}\langle l \rangle \equiv \varphi_i$ and define
$$\sigma_i(\lambda) = \begin{cases} \sigma_{i-1}(\lambda) & \text{if } \lambda\langle\rangle \not\equiv \varphi_i, \\ \lambda_v^{\sigma_{i-1}}(l) & \text{if } \lambda\langle\rangle \equiv \varphi_i. \end{cases}$$
First we prove that this definition is correct. Let v_1, v_2, k_1, k_2 be such that $\lambda_{v_1}^{\sigma_{i-1}}\langle k_1 \rangle \equiv \varphi_i \equiv \lambda_{v_2}^{\sigma_{i-1}}\langle k_2 \rangle$. Take l_1, l_2 as the largest numbers such that $\lambda_{v_1}^{\sigma_{i-1}}\langle l_1 \rangle \equiv \varphi_i \equiv \lambda_{v_2}^{\sigma_{i-1}}\langle l_2 \rangle$. Since v_1 satisfies $\lambda_{v_2}^{\sigma_{i-1}}\langle l_2 \rangle \equiv \varphi_i$, then $\lambda_{v_2}^{\sigma_{i-1}}[1 : l_2] = \lambda_{v_1}^{\sigma_{i-1}}[1 : l_2]$ by Lemma 3.12. The same holds for l_1 which means that $l_1 = l_2$ and $\lambda_{v_1}^{\sigma_{i-1}}(l_1) = \lambda_{v_1}^{\sigma_{i-1}}(l_2)$, which proves that the definition of σ_i is independent of the exact choices of v and k.
Now $|\sigma_i|_v = |\sigma_{i-1}|_v - (l - k)$, where k and l is the smallest and the largest number such that $\lambda_v^{\sigma_{i-1}}\langle k \rangle \equiv \varphi_i$ and $\lambda_v^{\sigma_{i-1}}\langle l \rangle \equiv \varphi_i$, respectively, because $\lambda_v^{\sigma_{i-1}}(l) = \lambda_v^{\sigma_i}(k)$ and due to the ordering, the rest of the process is independent of the beginning.

The last strategy of the sequence is clearly memory-less and satisfies the condition in the lemma. ∎

Corollary 3.18. *There exists a worst-case optimal strategy that is memory-less and there exists an average-case optimal strategy that is memory-less.*

Example 3.19. Recall the game and the strategy σ from Example 3.13. The strategy is clearly not non-adaptive, as $\sigma((12, <)) \neq \sigma((12, >))$. It is neither memory-less as $\sigma((12, =)) \neq \sigma((12, =), (12, =))$ but the accumulated knowledge of the solving processes is the same.
Consider a non-adaptive strategy $\tau: 1 \mapsto 12,\ 2 \mapsto 13,\ 3 \mapsto 14$. If the counterfeit coin is among the first three, it is discovered by the strategy in two experiments.

If the counterfeit coin is coin 4, it requires three experiments. Hence $\Lambda^\tau = 3$ and the value of τ on greater numbers is irrelevant.
If we apply the construction in Theorem 3.17 on σ, we get a memory-less strategy σ', given by

$$\sigma'(\varphi) = \begin{cases} 13 & \text{if } \varphi \equiv (x_1 \wedge \neg y) \vee (x_2 \wedge y), \\ 23 & \text{if } \varphi \equiv (x_1 \wedge y) \vee (x_2 \wedge \neg y), \\ 14 & \text{if } \varphi \equiv \neg x_1 \wedge \neg x_2, \\ 34 & \text{if } \varphi \equiv \neg x_1 \wedge \neg x_2 \wedge \neg x_4, \\ 12 & \text{otherwise.} \end{cases}$$

Notice that the valuation v with $v(x_3) = v(y) = 1$ is discovered in 3 experiments as the strategy does not repeat the experiment 12 now. Therefore, $\Lambda^{\sigma'} = 3$.
Both strategies τ and σ' are worst-case optimal. ♦

3.4 One-step look-ahead strategies

Specification of a strategy in general can be very complicated. In this section, we study a subclass of memory-less strategies that we call *one-step look-ahead*. These strategies select an experiment that minimizes the value of a given function on the set of possible knowledge in the next step.

Definition 3.20 (One-step look-ahead strategy). Let f be a function of type $2^{\text{FORM}'} \to \mathbb{R}$. A one-step look-ahead strategy with respect to f is a memory-less strategy such that for every $\varphi \in \text{FORM}_X$ and $e' \in E$,

$$f(\{\, \varphi \wedge \psi \mid \psi \in \Phi(e) \,\}) \leq f(\{\, \varphi \wedge \psi \mid \psi \in \Phi(e') \,\}).$$

Note that one-step look-ahead strategy with respect to f is not unique. For some formulas, there can be more experiments with the same value of f. To uniquely specify a strategy, we must provide the function f and a resolution method for these ambiguous states. Typically, we specify a total order on experiments and select the least experiment in the order satisfying the condition of Definition 3.20.
A few one-step look-ahead strategies for Mastermind have been already introduced in Section 2.2. We now define them formally in the general code-breaking games. In the Mastermind case, the experiments are ordered lexicographically by the colour combination.

Max-models. This strategy minimizes the worst-case number of remaining codes. For Mastermind, this was suggested by Knuth[19].

$$f(\Psi) = \max_{\varphi \in \Psi} \#\varphi.$$

Exp-models. This strategy minimizes the expected number of remaining codes. For Mastermind, this was suggested by Irwing[20].

$$f(\Psi) = \frac{\sum_{\varphi\in\Psi}(\#\varphi)^2}{\sum_{\varphi\in\Psi}\#\varphi}.$$

Ent-models. This strategy maximizes the entropy of the numbers of remaining codes, For Mastermind, this was suggested by Neuwirth[21].

$$f(\Psi) = \sum_{\varphi\in\Psi}\frac{\#\varphi}{N}\cdot\log\frac{\#\varphi}{N}, \text{ where } N = \sum_{\varphi\in\Psi}\#\varphi.$$

Parts. This strategy maximizes the number of satisfiable outcomes. For Mastermind, this was suggested by Kooi[22].

$$f(\Psi) = -|\{\varphi \mid \varphi\in\Psi,\ SAT(\varphi)\}|.$$

We suggest and analyse one-step look ahead strategies based on fixed variables. Let

$$\#_{\text{fixed}}\,\varphi = |\{x\in X \mid \forall v.v(\varphi)=1\Rightarrow v(x)=1\}\cup\{x\in X \mid \forall v.v(\varphi)=1\Rightarrow v(x)=0\}|$$

be the number of variables that have same value in all models of φ. Note that while the aforementioned strategies does not depend on the exact formalization of a problem, the number of fixed variables may differ for different encodings. For example, the choice of the following strategies in Example 3.7 differs for the two possible formalisations.

Min-fixed. Maximize the worst-case number of fixed variables, i.e.

$$f(\Psi) = -\min_{\varphi\in\Psi}\#_{\text{fixed}}\,\varphi.$$

Exp-fixed. Maximize the expected number of fixed variables, i.e.

$$f(\Psi) = -\frac{\sum_{\varphi\in\Psi}\#\varphi\cdot\#_{\text{fixed}}\,\varphi}{\sum_{\varphi\in\Psi}\#\varphi}.$$

Example 3.21. Recall Example 3.4 and consider two experiments in the first step. First, consider an experiment of weighing coin 1 against coin 2. All the 3 outcomes are satisfiable, the number of models is 2, 2 and 4 for outcome <, >, and =, respectively. If the experiment results in < or >, we know that the counterfeit coin is coin 1 or coin 2. If it results in =, the counterfeit coin is coin 3 or coin 4. Therefore, every outcome fixes two variables.

	12	1234
Max-models	4	4
Exp-models	3	4
Ent-models	-1.04	-0.69
Parts	-3	-2
Min-fixed	-2	0
Exp-fixed	-2	0

Table 3.22: Values of various one-step look-ahead strategies in the counterfeit coin problem with four coins on experiments 12 and 1234.

Second, consider an experiment of weighing coins 1 and 2 agains coins 3 and 4. As exactly one coin must be counterfeit, outcome = is not possible. Outcomes < and > are symmetrical, both have 4 models and fix no variables.

Table 3.22 shows the values of the defined strategies on these two experiments. The experiment 12 wins with all strategies except for Maximal number of models, where the values are the same. ⧫

4 Experiment equivalence and algorithms

What makes the analysis of code-breaking games difficult is typically the large number of experiments. For example, during the evaluation a one-step look-ahead strategy with respect to function f, we need to compute the value of f for all experiments. Even more importantly, during optimal strategy synthesis, we have to try all experiments in every state, finish the game with the gained knowledge, and determine the optimal choice.

Fortunately, some experiments are usually equivalent to some others in the sense that the knowledge they can give us is either exactly the same or symmetrical. In the counterfeit-coin problem, for example, the parametrized experiment of weighting 4 coins against 4 coins has $\frac{1}{2} \cdot \binom{12}{4} \cdot \binom{8}{4} = 17325$ possible parametrizations but, in the initial state, all of them are equivalent as they give us symmetrical knowledge.

This chapter formally introduces the concept of experiment equivalence. We prove that in various situations, it is sufficient to consider one experiment from each equivalence class. This fact is used in presented algorithms for well-formed checking, evaluation of one-step look-ahead strategies and optimal strategy synthesis.

4.1 Experiment equivalence

We start with a formal definition of equivalence of two experiments. The section continues with our suggestion on a method for equivalence testing based on isomorphism of labelled graphs. This method is crucial for the algorithms presented in the following sections.

Definition 4.1 (Experiment equivalence). Let $e \in E$ be an experiment and $\pi \in \text{PERM}_X$ a variable permutation. A π-symmetrical experiment to e is an experiment $e^\pi \in E$ such that $\{\varphi^\pi \in \Phi(e)\} = \{\varphi \in \Phi(e^\pi)\}$. Clearly, no π-symmetrical experiment to e may exists.

A *symmetry group* Π of the given game is the maximal subset of PERM_X such that for every $\pi \in \Pi$ and for every experiment $e \in E$, there exists a π-symmetrical experiment to e.

An experiment $e_1 \in E$ is equivalent to $e_2 \in E$ with respect to φ, written $e_1 \cong_\varphi e_2$, if and only if there exists a permutation $\pi \in \Pi$ such that

$$\{\varphi \wedge \psi \mid \psi \in \Phi(e_1)\} \equiv \{(\varphi \wedge \psi)^\pi \mid \psi \in \Phi(e_2)\}.$$

Example 4.2. Recall the running example from the previous chapter, introduced in Example 3.4. Experiment 23 is a (x_1x_3)-symmetrical experiment to 12, because (for $\pi = (x_1x_3)$)

$$\left\{ ((x_1 \wedge \neg y) \vee (x_2 \wedge y))^\pi,\ ((x_1 \wedge y) \vee (x_2 \wedge \neg y))^\pi,\ (\neg(x_1 \vee x_2))^\pi \right\} =$$
$$\left\{ (x_3 \wedge \neg y) \vee (x_2 \wedge y),\quad (x_3 \wedge y) \vee (x_2 \wedge \neg y),\quad \neg(x_3 \vee x_2) \right\}.$$

In fact, for every experiment $e = (t, p)$ and every permutation π stabilizing y, we can permute the parameters of t accordingly and get a π-symmetrical experiment to e. Therefore, the symmetry group of the game is $\Pi = \{\pi \in \mathrm{PERM}_X \mid \pi(y) = y\}$. Since Π is also the symmetry group of φ_0, all experiments of the same type are equivalent, and the quotient set of E by $\cong_{\varphi_0}$ has only two equivalence classes. For a more complex example, let $\varphi = \varphi_0 \wedge \neg(x_1 \vee x_2)$. Experiment 3124 is now equivalent to 43, with $\pi = \mathrm{ID}_X$. The corresponding formulas are equivalent even though they are syntactically different.

In the rest of the section, we suggest a method for testing whether two given experiments are equivalent with respect to a given formula.
First, we show a construction of *base graph* for the given game, automorphisms of which are a subset of symmetry group Π. Then we describe the construction of *experiment graph*, which is build on top of the base graph. We prove that if the experiment graphs are isomorphic, the corresponding experiments are equivalent.
Recall that a *labelled graph* is a triple (V, E, l), where (V, E) is a graph and $l : V \to L$ (L being the set of labels) is a labelling function. Isomorphism of two labelled graphs is a bijection between their sets of vertices that preserves edges and labels.

Base graph construction

Base graph of a game $\mathcal{G} = (X, \varphi_0, \Sigma, F, T)$ is a labelled graph $B = (V, E, l)$ described below.

- There is a vertex for every proposition variable and every mapping, i.e. $V = X \cup F$.
- A mapping is connected by edges with all variables in its value range, i.e. $(f, x) \in E$ if there is a symbol $a \in \Sigma$ such that $f(a) = x$,
- Two variables are connected if the variables are values of different mappings on the same symbol from the alphabet, and these mappings appear in outcome formulas of the same experiment. Formally, $(x_1, x_2) \in E$ if there is a symbol $a \in \Sigma$ and two mappings $f_1, f_2 \in F$ such that $f_1(a) = x_1$, $f_2(a) = x_2$, and there is a parametrized experiment $t \in T$ and a number $k \leq n_t$ such that both $f_1(\$k)$ and $f_2(\$k)$ appear in the outcome formulas of t.

- Vertices corresponding to mappings have their own labels. Vertices corresponding to variables are labelled "variable", expect for variables that appear directly in some outcome formula of a parametrized experiment, which has their own labels.

Example 4.3. Base graph for the counterfeit coin problem with 4 coins is shown in Figure 4.4 on the left. Note that vertices y and f_x have separate labels while other vertices are labelled "variable".

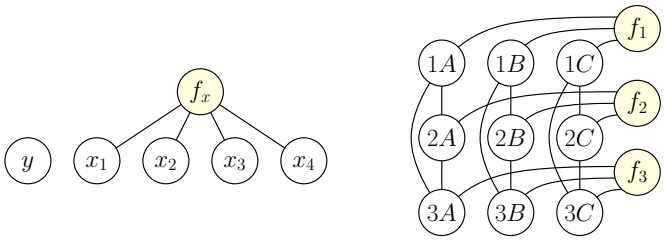

Figure 4.4: Base graph for the counterfeit coin problem with 4 coins (left) and for Mastermind with 3 pegs and 3 colours (right).

A more complicated example is the base graph of Mastermind with 3 pegs and 3 colours, shown on the right-hand side. Vertices f_1, f_2, f_3 have separate labels, all other vertices are labelled "variable". For simplicity, we leave out symbol x in the figure, e.g. write $1A$ instead of x_{1A}. ♦

Lemma 4.5. *Let π be an automorphism of B. Then $\pi|_X \in \Pi$.*

Proof. Let π be an automorphism of B and (t, p), $p = p_1, p_2, \ldots, p_n$ an experiment. We show that there exists a π-symmetrical experiment to (t, p).

Let $F_i \subseteq F$ be a set of mapping that are present in an outcome formula of t with parameter $\$i$. The vertices $f(p_i)$ for $f \in F_i$ form a clique in B and so must vertices $\pi(f(p_i))$, $f \in F_i$.

Since mappings F have pairwise disjoint images, two variables x_1, x_2 can be connected by an edge only if there is a symbol $k \in \Sigma$ and mappings $f, g \in F$ such that $f(k) = x_1$, $g(k) = x_2$.

Define r_i as a symbol in Σ satisfying $f(r_i) = \pi(f(p_i))$ for some $f \in F_i$. Note that such r_i always exists because $\pi(f) = f$ for every $f \in F$. Thanks to the aforementioned property, if $f(r_i) = \pi(f(p_i))$ holds for *some* $f \in F_i$, it holds for *all* $f \in F_i$ and the definition is thus correct.

Now, consider the experiment (t, r), $r = r_1, r_2, \ldots, r_n$. All variables appearing directly in the parametrized formula are stabilized by π and, for all expressions $f(\$i)$ it holds $f(r_i) = \pi(f(p_i))$ by the construction of r_i, which means that (t, r) is π-symmetrical to (t, p). ■

Experiment graph

Let $\varphi \in \text{FORM}_X$ be a formula. An *x-rooted tree of φ* is a graph created from the syntax tree of φ by unification of leaves that correspond to the same variables and adding a special vertex with label x that is connected to the root of the syntax tree, i.e. to the top-level operator of φ. Other vertices of the graph are labelled by their type (e.g. "variable", "and-operator", etc.)
In this construction, we need the trees of two formulas be isomorphic if and only if the formulas are syntactically equivalent. This clearly holds if all the operators are commutative. As the only non-commutative operator is implication, we substitute subformulas of a form $\varphi \to \psi$ with $\psi \vee \neg\varphi$.
Let B be the base graph for the given game, $\varphi \in \text{FORM}'$ some partial knowledge and e an experiment. The experiment graph $B_{\varphi,e}$ is constructed as follows.

- Begin with graph B.
- Add a "knowledge"-rooted tree of φ.
- For each outcome $\psi \in \Phi(e)$, add an "outcome"-rooted tree of ψ.

Theorem 4.6. *If B_{φ,e_1} is isomorphic to B_{φ,e_2}, then $e_1 \cong_\varphi e_2$.*

Proof. Let ρ be the graph isomorphism of B_{φ,e_1} and B_{φ,e_2} and let $\pi = \rho|_X$, considered as a permutation of X. Since B is the vertex-induced subgraph of both B_{φ,e_1} and B_{φ,e_2} by the set of vertices $X \cup F$, π is a member of Π by Lemma 4.5.
The isomorphism ρ maps the only "knowledge"-labelled vertex in the first graph to the only "knowledge"-labelled vertex in the second graph, which implies the equivalence of the formulas, $\varphi^\pi \equiv \varphi$. Similarly, "outcome"-labelled vertices are mapped to "outcome"-labelled vertices, which means that $\{\psi^\pi \mid \psi \in \Phi(e_1)\} = \Phi(e_2)$. This is sufficient for the experiments to be equivalent with respect to φ. ■

Example 4.7. Recall the running example of the counterfeit coin problem with 4 coins. Base graph for the game was shown in Example 4.3. Let $\varphi = \varphi_0 \wedge \neg(x_1 \vee x_2)$ be the accumulated knowledge of the solving process $(12, =)$ and let e be experiment 3124. Experiment graph $B_{\varphi,e}$ is shown in Figure 4.8; Ex_1 denotes the EXACTLY_1 operator.
Unfortunately, the graph for experiment 43 is clearly not isomorphic to this graph, although the experiments are equivalent with respect to φ. We address this problem in the following. ⧫

Improvement by fixed variables

The previous example shows that the method explained above does not detect some basic equivalences. To address the problem, we suggest the following improvement to the construction of $B_{\varphi,e}$.

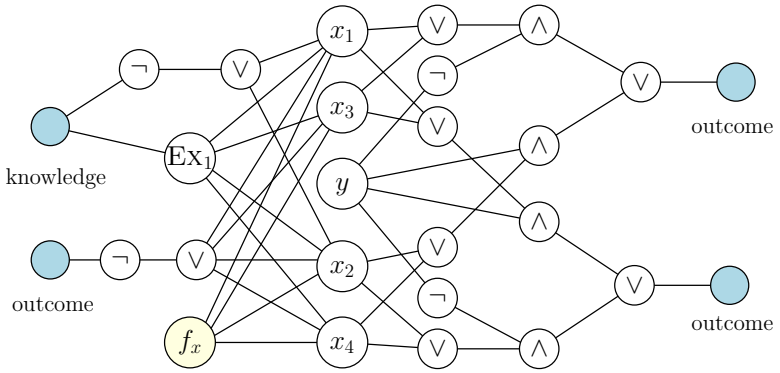

Figure 4.8: Experiment graph for 3124 with knowledge $\varphi_0 \wedge \neg(x_1 \vee x_2)$.

1. Compute fixed variables of formula φ using a SAT solver.

2. Simplify formula φ with the knowledge of its fixed variable.

3. Simplify the outcomes of e, formulas $\psi \in \Phi(e)$, with the knowledge of fixed variables in φ.

4. Construct the graph as described above.

5. Label the vertices corresponding to fixed variables with "false" or "true" label, according to their fixed value.

As the simplified formulas are equivalent to the original formulas, Theorem 4.6 also holds if the graphs B_{φ,e_1}, B_{φ,e_2} are constructed with this approach.

Example 4.9. Let us apply the suggested improvement on the previous example. The formula $\varphi = \varphi_0 \wedge \neg(x_1 \vee x_2)$ fixed variables x_1 and x_2 to 0. Figure 4.10 shows the constructed experiment graph after the simplification of the formulas.

Vertices x_1 and x_2 are now labelled "false" and are connected only to vertex f_x. Compare the structure with the graph in Figure 4.8. Note that the graph is now isomorphic to the graph of experiment 43. ⧫

Algorithm 4.11 describes the elimination of equivalent experiments with respect to a formula φ, which is a straightforward application of the method described in this section. We assume we have a method to construct canonical labelling of a graph, which is used to decide the graph isomorphism.

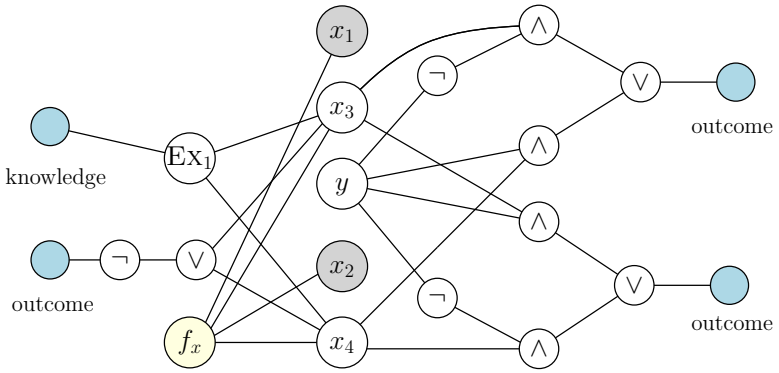

Figure 4.10: Simplified experiment graph for 3124 with knowledge $\varphi_0 \wedge \neg(x_1 \vee x_2)$.

Algorithm 4.11: Elimination of equivalent experiments

Input: formula φ
Output: set $S \subseteq E$, such that $\forall e \in E\ \exists s \in S.\ e \cong_\varphi s$

$B \leftarrow$ construct the base graph for the game
$fixed \leftarrow$ compute fixed variables of φ using a SAT solver
$\varphi' \leftarrow$ substitude values for fixed variables in φ and simplify
Label vertices in B corresponding to fixed variables with their fixed value
Add the "knowledge"-rooted tree of φ' to B
$S \leftarrow \varnothing$
$hash \leftarrow$ an empty hash table for graphs
for $e \in E$ **do**
    $B_e \leftarrow$ clone B
    for $\psi \in \Phi(e)$ **do**
        $\psi' \leftarrow$ substitude values for $fixed$ in ψ and simplify
        Add "outcome"-rooted tree of ψ' to B_e
    $B_e \leftarrow$ canonize B_e
    if B_e *is not present in* $hash$ **then**
        $hash$.insert(B_e)
        $S \leftarrow S \cup \{e\}$
return S

4.2 Well-formed check

Experiment equivalence can be used during the verification that a given game is well-formed, as stated by the following lemma.

Lemma 4.12. *Let $S \subseteq E$ be a subset of experiments such that for every $e \in E$,*

there exists $s \in S$ such that $e \cong_{\varphi_0} s$. If the formula $\varphi_0 \Rightarrow \textsc{exactly}_1(\Phi(e'))$ is a tautology for all $s \in S$, then the game is well-formed.

Proof. Assume by contradiction that the game is not well formed, i.e. there is $e \in E$ and $v \in \textsc{val}'$ such that the number of formulas in $\Phi(e)$ satisfied by v is not equal to one.
If $e \in S$, the formula $\varphi_0 \Rightarrow \textsc{exactly}_1(\Phi(e'))$ is not satisfied by v. Contradiction.
Otherwise, there exists $s \in S$ such that $e \cong_{\varphi_0} s$, i.e. there exists $\pi \in \textsc{perm}_X$ such that $\{\varphi_0 \wedge \psi \mid \psi \in \Phi(e)\} = \{(\varphi_0 \wedge \psi)^\pi \mid \psi \in \Phi(s)\}$. Since $\varphi_0 \Rightarrow \textsc{exactly}_1(\Phi(s))$ is a tautology, the permuted formula $\varphi_0^\pi \Rightarrow \textsc{exactly}_1(\psi^\pi \mid \psi \in \Phi(s))$ is a tautology as well. Therefore, exactly one formula from the set $\{(\varphi_0 \wedge \psi)^\pi \mid \psi \in \Phi(s)\}$ is satisfiable and the same holds for $\{\varphi_0 \wedge \psi \mid \psi \in \Phi(e)\}$, which implies that $\varphi_0 \Rightarrow \textsc{exactly}_1(\Phi(e))$ is a tautology. ∎

4.3 Analysis of one-step look-ahead strategies

The following lemma gives us a right to disregard equivalent experiments during analysis of some one-step look-ahead strategies.

Lemma 4.13. *Let $f : 2^{\textsc{form}'} \to \mathbb{R}$ be a function such that $f(\Psi) = f(\{\varphi^\pi \mid \varphi \in \Psi\})$ for any $\Psi \subseteq \textsc{form}'$ and $\pi \in \textsc{perm}_X$. Let $<$ be a total order of experiments E. Let σ be a one-step look-ahead strategy with respect to f and $<$, and let φ be a formula. Suppose there are experiment e_1, e_2 such that $e_1 \cong_\varphi e_2$ and $e_1 \leq e_2$. Then $\sigma(\varphi) \neq e_2$.*

Proof. If follows directly from Definition 4.1 and the property of f that

$$f(\{\varphi \wedge \psi \mid \psi \in \Phi(e_1)\}) = f(\{\varphi \wedge \psi \mid \psi \in \Phi(e_2)\}).$$

Since $e_1 < e_2$, the strategy always prefers e_1 to e_2. ∎

Note that all one-step look-ahead strategies discussed in Section 3.4 satisfy the condition of the lemma. In general, any function based on satisfiability, the number of models and/or the number of fixed variables of the formulas will satisfy this condition as these function are permutation independent.
A recursive approach for the analysis of one-step look-ahead strategies is shown in Algorithm 4.14. There are two options on line 5 of Analyse function. The first is to use the algorithm to eliminate equivalent formulas and thus evaluate the strategy only on a subset of experiments. The second is to go through all possible experiments.
In general, it cannot be said which variant is faster. This depends on the ratio between the time needed for graph canonization and the time needed for strategy evaluation.

Algorithm 4.14: Analysis of a one-step look-ahead strategy

Input: function $f : 2^{\text{FORM}'} \to \mathbb{R}$
Output: (w, a), where w and a is the worst-case and the average-case number of experiments performed by the strategy

```
globalsum ← 0
globalmax ← 0
Analyse(φ0, 1)
return (globalmax, globalsum / #φ0)

Function Analyse(φ, depth)
    choice ←None
    bestvalue ← ∞
    S ← eliminate equivalent experiments by running Algorithm 4.11 on φ
    for e ∈ S (variant 1) or e ∈ E (variant 2) do
        value ← f(e) if value < bestvalue then
            choice ← e
            bestvalue ← value
    for ψ ∈ Φ(e) do
        if not SAT(φ ∧ ψ) then continue
        if #(φ ∧ ψ) = 1 then
            globalsum ← globalsum + depth
            globalmax ← max(globalmax, depth)
        else
            Analyse(φ ∧ ψ, depth + 1)
```

4.4 Optimal strategy synthesis

Backtracking is the basic method for worst-case and average-case optimal strategy synthesis. Our goal in this section is to prove that we can disregard equivalent experiments and thus significantly reduce branching of the algorithm in every step.

First, let us define $\kappa(\varphi)$ and $\kappa_{\text{exp}}(\varphi)$ as the optimal number of experiments needed to reveal the secret code when starting with knowledge φ in the worst-case and in the average-case, respectively. We can say that $\kappa(\varphi)$, resp. $\kappa_{\text{exp}}(\varphi)$ is the worst-case, resp. average-case number of experiments of a worst-case, resp. average-case optimal strategy if we change the initial constraint of the game to φ.

Similarly, we define $\kappa(\varphi, e)$ and $\kappa_{\text{exp}}(\varphi, e)$ as the optimal number of experiment needed to reveal the secret code when starting with knowledge φ and with e as the first experiment.

There is an obvious relationship between $\kappa(\varphi)$ and $\kappa(\varphi, e)$ and between $\kappa_{\exp}(\varphi)$ and $\kappa_{\exp}(\varphi, e)$. For any $\varphi \in \text{FORM}'$,

$$\kappa(\varphi) = \min_{e \in E} \kappa(\varphi, e), \text{ and} \qquad \kappa_{\exp}(\varphi) = \min_{e \in E} \kappa_{\exp}(\varphi, e). \tag{4.1}$$

Further, we can compute $\kappa(\varphi, e)$ and $\kappa_{\exp}(\varphi, e)$ from the optimal values for the subproblems after one experiment. These relationships are based on the definitions of the worst-case and average-case number of experiments of a strategy (Λ^σ and $\Lambda^\sigma_{\exp}$). For any $\varphi \in \text{FORM}'$ and $e \in E$,

$$\kappa(\varphi, e) = \begin{cases} 0 & \text{if } \#\varphi = 1, \\ \infty & \text{if } \exists \psi \in \Phi(e).\ \varphi \wedge \psi \equiv \varphi, \\ 1 + \max_{\psi \in \Phi(e)} \kappa(\varphi \wedge \psi) & \text{otherwise.} \end{cases} \tag{4.2}$$

$$\kappa_{\exp}(\varphi, e) = \begin{cases} 0 & \text{if } \#\varphi = 1, \\ \infty & \text{if } \exists \psi \in \Phi(e).\ \varphi \wedge \psi \equiv \varphi, \\ 1 + \frac{\sum_{\psi \in \Phi(e)} \#(\varphi \wedge \psi) \cdot \kappa_{\exp}(\varphi \wedge \psi)}{\#\varphi} & \text{otherwise.} \end{cases} \tag{4.3}$$

Let us now define a set of optimal choices in a state. For a $\varphi \in \text{FORM}'$, we define

$$\varepsilon(\varphi) = \{e \in E \mid \forall e' \in E.\ \kappa(\varphi, e) \leq \kappa(\varphi, e')\}, \text{ and}$$
$$\varepsilon_{\exp}(\varphi) = \{e \in E \mid \forall e' \in E.\ \kappa_{\exp}(\varphi, e) \leq \kappa_{\exp}(\varphi, e')\}.$$

The following lemma is a straightforward consequence of the definitions of κ and ε.

Lemma 4.15. *Let σ be a strategy such that $\sigma(\varphi) \in \varepsilon(\varphi)$ for every $\varphi \in \text{FORM}'$. Then σ is worst-case optimal. Similarly, let σ' be a strategy such that $\sigma'(\varphi) \in \varepsilon_{exp}(\varphi)$ for every $\varphi \in \text{FORM}'$. Then σ' is average-case optimal.*

Now we are ready for the main theorem, which gives as a right to consider only one experiment from each equivalence class of $E/\cong_\varphi$ in the state where the accumulated knowledge is φ. The exact algorithm for optimal strategy synthesis with further optimizations is described in Section 5.3.

Theorem 4.16. *For every $\varphi \in \text{FORM}'$,*

1. *$\kappa(\varphi) = \kappa(\varphi^\pi)$ and $\kappa_{exp}(\varphi) = \kappa_{exp}(\varphi^\pi)$ for all $\pi \in \Pi$, and*
2. *if $e_1 \cong_\varphi e_2$, then $e_1 \in \varepsilon(\varphi) \Leftrightarrow e_2 \in \varepsilon(\varphi)$ and $e_1 \in \varepsilon_{exp}(\varphi) \Leftrightarrow e_2 \in \varepsilon_{exp}(\varphi)$.*

Proof. The proof for the worst case (κ, ε) and for the average-case $(\kappa_{\text{exp}}, \varepsilon_{\text{exp}})$ is exactly the same, so we prove it only for worst-case.

Since $\pi \in \Pi$, there exists a π-symmetrical experiment e^π for every $e \in E$. Recall that $\Phi(e^\pi) = \{\psi^\pi \mid \psi \in \Phi(e)\}$. We show by induction on the number of models of φ that $\kappa(\varphi, e) = \kappa(\varphi^\pi, e^\pi)$, which is sufficient for the first part.

As $\#\varphi = \#\varphi^\pi$, the statements follows directly from (4.1) and (4.2) for formulas with one model. For the induction step, observe that $\#(\varphi^\pi \wedge \psi^\pi) = \#(\varphi \wedge \psi)$ and $\kappa(\varphi^\pi \wedge \psi^\pi) = \kappa(\varphi \wedge \psi)$ if $\varphi \not\equiv \varphi \wedge \psi$, by the induction hypothesis. The statement now follows from (4.2) as the right sides are equal.

For the second part, it suffices to prove that $\kappa(\varphi, e_1) = \kappa(\varphi, e_2)$. As the experiments are equivalent, there exists a permutation $\pi \in \Pi$, such that $\{\varphi \wedge \psi \mid \psi \in \Phi(e_1)\} = \{(\varphi \wedge \psi)^\pi \mid \psi \in \Phi(e_2)\}$. The equation now follows from (4.2) and the facts that $\#\varphi = \#\varphi^\pi$ and $\kappa(\varphi) = \kappa(\varphi^\pi)$ (proven in the first part). ■

A recursive algorithm for the computation of the value of an average-case optimal strategy, $\kappa_{\text{exp}}(\varphi)$ is shown in Algorithm 4.17. The algorithm is based on equation (4.3) and the previous theorem.

Apart from formula φ, the recursive function takes another argument, *opt*, which is used for branch pruning. The value of *opt* is the best value of $\kappa_{\text{exp}}(\varphi, e)$ that was found so far. Clearly, if we can be sure that $\kappa_{\text{exp}}(\varphi, e') > opt$, we are not interested in the exact value as the experiment will not be selected anyway. A lower bound on $\kappa_{\text{exp}}(\varphi, e')$ can be computed using Lemma 3.11.

The inital value of *opt* should be ∞ or an upper bound on $\kappa_{\text{exp}}(\varphi_0, e)$.

Algorithm 4.17: Synthesis of average-case optimal strategy.

```
1  Function Optimum(φ, opt)
2      if #φ = 1 then return 0
3      lb ← LowerBound(φ)
4      if lb > opt then return ∞
5      compute subset of experiments S such that e ∈ E ⇒ ∃e' ∈ S. e ≅_φ e'
6      for s ∈ S, ordered by max_{ψ∈Φ(s)} #(φ ∧ ψ) do
7          if only one of φ ∧ ψ, ψ ∈ Φ(s) is satisfiable then continue
8          val ← 0
9          for ψ ∈ Φ(s) do
10             if SAT(φ ∧ ψ) then
                   val ← val + #(φ ∧ ψ) · (1 + Optimum(φ ∧ ψ, opt))
11         val ← val / #(φ)
12         if val < opt then opt ← val
13     return opt
```

Note that the order of experiments on line 7 is not necessary for correctness of the algorithm. Our goal is to find a good experiment as soon as possible, because this allows us to prune many branches in the beginning.

5 COBRA tool

Development of a general tool for code-breaking games analysis is the main part of this work. We named the tool COBRA, the **co**de-**br**eaking game **a**nalyzer. Input of the tool is a game specification in a special language, which we describe first. Basic usage is explained afterwards with a description of various tasks that the tool can perform with a loaded game. Notes on dependencies and requirements on external tools, on extensibility of COBRA and some more implementation details are described in later sections.

Source codes of the tool, together with a detailed documentation and specifications of the games described in Chapter 2 can be found in the electronic attachment of the thesis. A git repository on GitHub[1] was used during the development, so another way of obtaining the source codes is by cloning the repository at `https://github.com/myreg/cobra`. This website also serves as a homepage of the project, and contains all related documents.

COBRA is available under *BSD 3-Clause License*[2], text of which is a part of source codes. The tool is dependent on external tools, namely Picosat, Minisat and Bliss. Picosat and Minisat are available under *MIT License*; Bliss is available under *GNU GPL v3*.

5.1 Input language

First we describe the low-level language that is the input format of COBRA. Next, the language is equipped with a preprocessor that allows parametrized generation of the low-lever format.

Low-level language

The low-level language is directly based on Definition 3.1, the formal definition of code-breaking games. It is case-sensitive and whitespace is not significant at any position.

From a lexical point of view, there are three atoms. Identifier (<ident>), is a string starting with a letter or underscore, which may contain letters, digits and underscores. Integer (<int>) is a sequence of digits. String (<string>) is sequence of arbitrary characters enclosed in quotes. Further, list of X (<x-list>) is a comma-separated list of atoms of type X, generated by grammar <x-list> ::= <x> | <x-list> , <x>.

1. `http://www.github.com`
2. `http://opensource.org/licenses/BSD-3-Clause`

VARIABLE <ident>	Declares a variable with a given identifier.
VARIABLES <ident-list>	Declares variables with given identifiers.
CONSTRAINT <formula>	Defines the initial constraint φ_0.
ALPHABET <string-list>	Defines the parametrization alphabet Σ.
MAPPING <ident> <ident-list>	Defines a mapping with a given identifier. The seconds argument is a list of variable identifiers defining the values of the mapping for all elements of the alphabet.
EXPERIMENT <string> <int>	Opens a section defining a new experiment named by the first argument and having the number of parameter given by the second argument. The section closes automatically with a definition of a new experiment.
PARAMS-DISTINCT <int-list>	Defines a restriction on the parameters of the experiment, requiring that parameters at specified positions are different. This is the only type of allowed restriction.
PARAMS-SORTED <int-list>	Declares that the outcomes of the experiment does not depend on the order of the parameters at specified positions and, therefore, only parametrizations where parameters at these positions are sorted can be considered. This is not necessary for the game specification but can significantly simplify game analysis.
OUTCOME <string> <formula>	Defines an outcome of the experiment named by the first argument.

Table 5.1: Statements in the low-level language.

The input file is parsed line by line, each line must have one of the forms listed in Table 5.1. We specify what "formula" is by the following grammar:

<formula> ::= <ident$_1$> | (<formula>) | ! <formula>
| <formula> ∘ <formula> | X-<int$_1$> (<formula-list>),
| <ident$_2$> ($ <int$_2$>),

where <ident$_1$> is an identifier of a variable and ∘ ∈ {and, &, or, |, –>, <–, <–>} is a standard logical operator with its usual meaning.

X is a *numerical operator* ATLEAST, ATMOST or EXACTLY, explained in Section 3.1. These operators are non-standard and could be cut out, however, they are quite common and useful in specification of code-breaking games and their naive ex-

pansion to standard operators causes exponential expansion of the formula (with respect to k). Hence we support these operators in the language and we handle them specifically during the transformation to CNF, avoiding the exponential expansion by introduction of new variables. The conversion is described in detail in Section 5.5.

Finally, the last rule of the grammar allows for formula parametrization. This can appear only in formulas defining an outcome of an experiment. The first part, <ident$_2$>, must be an identifier of a defined mapping, and <int$_2$> must be in the range from 1 to the number of parameters of the currently defined experiment.

Example 5.2. Recall the running example introduced in Example 3.4. The counterfeit-coin problem with four coins can be specified in the low-level language as follows.

```
VARIABLES y, x1, x2, x3, x4
CONSTRAINT Exactly-1(x1, x2, x3, x4)
ALPHABET '1', '2', '3', '4'
MAPPING X x1, x2, x3, x4

EXPERIMENT 'weighing2x2' 4
  PARAMS_DISTINCT 1, 2, 3, 4
  OUTCOME 'lighter' ((X$1 | X$2) & !y) | ((X$3 | X$4) & y)
  OUTCOME 'heavier' ((X$1 | X$2) & y) | ((X$3 | X$4) & !y)
  OUTCOME 'same' !(X$1 | X$2 | X$3 | X$4)
```

♦

To parse this language, we use a standard combination of *GNU Flex*[3] for lexical analysis and *GNU Bison*[4] for parser generation. The exact LALR grammar used can be found in `cobra.ypp` file in the source codes.

Python preprocessing

Although the low-level language is sufficient for our purposes, it is not very user-friendly and simple changes in a game may require extensive changes in the input file. For example, if you want to change the number of coins in the Counterfeit Coin problem, it would be nice to change only one number in the input but now you have to change many lines and create or delete some experiment sections. The situation is even worse in Mastermind, in which the outcome formulas are generated by the algorithm described in 3.8. We would need to write a script or a computer program to generate the input file.

This is the point where preprocessor comes into the picture. As the demands may significantly differ for different games, we decided not to create our own

3. http://flex.sourceforge.net/
4. http://www.gnu.org/software/bison/

preprocessing engine and use Python[5], a popular and intuitive scripting language, instead.

The input can now be an arbitrary Python file with calls to extra functions Variable, Variables, Constraint, Alphabet, Mapping, Experiment, Params-distinct and Outcome, which map directly to the constructs in the low-level language.

The generation of the low-level input is done by execution of the Python file with those special function ingested. All the functions do is printing the corresponding low-level language constructs to the output file. Types of their parameters are listed below.

Function	Type of x	Type of y
Variable(x)	string	-
Variables(x)	list of strings	-
Constraint(x)	formula (as a string)	-
Alphabet(x)	list of strings	-
Mapping(x, y)	string	list of strings
Experiment(x, y)	string	integer
Params-distinct(x)	list of integers	-
Params-sorted(x)	list of integers	-
Outcome(x, y)	string	formula (as a string)

Example 5.3. An example specification of the counterfeit coin problem, based on Example 3.7, follows.

```
N = 12
x_vars = ["x" + str(i) for i in range(N)]
VARIABLES(["y"] + x_vars)
CONSTRAINT("Exactly-1(%s)" % ",".join(x_vars))
ALPHABET([str(i) for i in range(N)])
MAPPING("X", x_vars)

# Helper function for disjunction of parameters
# For example, params(2,4) = "X$2 | X$3 | X$4"
params = lambda n0, n1: "|".join("X$" + str(i)
                                 for i in range(n0, n1 + 1))

for m in range(1, N//2 + 1):
  EXPERIMENT("weighing" + str(m), 2*m)
  PARAMS_DISTINCT(range(1, 2*m + 1))
  OUTCOME("lighter", "((%s) & !y) | ((%s) & y)" %
                     (params(1, m), params(m+1, 2*m)))
  OUTCOME("heavier", "((%s) & y) | ((%s) & !y)" %
                     (params(1, m), params(m+1, 2*m)))
  OUTCOME("same", "!(%s)" % params(1, 2*m))
```

5. https://www.python.org

5.2 Compilation and basic usage

To compile COBRA, run `make` in the program folder. This automatically compiles external tools and builds the necessary libraries. If everything finishes successfully, the binary executable `cobra-backend` is created and ready for being used. In case of a problem during the compilation, please refer to the *Requirements* paragraph of Section 5.7.

The basic syntax to launch the tool is the following.

```
./cobra [-m <mode>] [-b <backend>] [other options] <input file>
```

Mode of operation, specified by `-m` switch, specifies what the tool will do with the game. The four possible modes are described in Section 5.3, together with the description of other options that depend on the mode. Backend, specified by `-b` switch, specifies which backend should be used for SAT solving and model counting. Details can be found in Section 5.5.

The main executable, `cobra`, is a Python script that preprocesses the input file and writes the low-level game specification to `.cobra.in`. Then it executes `cobra-backend` and passes on all the options given by the user. Therefore, you can run COBRA on a low-level input format by launching `cobra-backend` directly with the same syntax.

Before `cobra-backend` finishes, it always outputs a *time overview*, with information on how much time was spend on which operations and how many calls to the SAT solver and to the symmetry breaker has been made.

```
===== TIME OVERVIEW =====
Total time: 74.68s
Bliss (calls/time): 1984 / 0.10s
SAT solvers          sat              fixed            models
* PicoSolver       59 / 0.09s      197  / 0.26s      5635 / 73.23s
```

Figure 5.4: An example of the time overview after a time demanding task.

5.3 Modes of operation

Overview mode [o, overview] (default)

```
./cobra -m overview <input file>
```

Overview mode serves as a basic check that the input file is syntactically correct and that the specified game is sensible. In this mode, the tool prints basic information about the loaded game, such as number of variables, number of experiments,

size of the search space, size of the preprocessed input file, trivial bounds on the worst-case and expected-case number of experiments, etc.

It also performs a *well-formed check*, i.e. verifies that the specified game is well-formed according to Definition 3.6. Algorithm for the verification follows directly from Lemma 4.12. For each experiment, we verify that $\varphi_0 \to \text{EXACTLY}_1(\psi_1, \ldots, \psi_k)$, where $\psi_1, \ldots \psi_k$ are the outcomes of the experiment, is a tautology. This is be done by negating the formula, passing it to a SAT solver, asking for satisfiability and expecting a negative result.

If a problem is found, the tool outputs an assignment and an experiment for which no outcome, or more that one outcome, is satisfied.

```
Well-formed check... failed!
EXPERIMENT: weighing1 1 2
PROBLEMATIC ASSIGNMENT:
  TRUE: y x3
  FALSE: x1 x2 x4 x5 x6 x7 x8 x9 x10 x11 x12
```

Figure 5.5: An example of a failed well-formed check in the counterfeit coin problem with no "=" outcome.

Simulation mode [s, simulation]

```
./cobra -m simulation -e <strategy> -o <strategy> <input file>
```

In the simulation mode, you specify a strategy for the codebreaker (for experiment selection) and for the codemaker (for outcome selection). This can be done using the `-e` or the `--codebreaker` switch, and the `-o` or the `--codemaker` switch, respectively.

Here, the codemaker is not considered a player who chooses the secret code in the beginning and then only evaluates experiments, but a player who chooses the outcomes of the experiments as they come according to his will. The only condition is that the outcomes are consistent.

For the codebreaker, COBRA supports all one-step look-ahead strategies described in Section 3.4: `max-models`, `exp-models`, `ent-models`, `parts`, `min-fixed` and `exp-fixed`. The codemaker can either use strategy `models`, which selects an outcome with maximal number of models, or `fixed`, which select an outcome with minimal number of fixed variables.

Apart from these, the tool supports two extra options for both players, `interactive` and `random`, which are not strategies in the sense of Definition 3.9.

If "interactive" is specified as the codebreaker's strategy, the tool prints a list of all non-equivalent experiments in each round (together with the number of possible outcomes, number of fixed variables and number of remaining possibilities) and

the user is asked to choose the experiment that will be performed. This effectively allows the user to play the game against a codemaker's strategy. Similarly, if "interactive" is the codemaker's strategy, possible outcomes are printed after each experiment and the user is asked to choose one. Unsatisfiable outcomes are printed as well but are marked accordingly and the user cannot select them.

In the random mode, an experiment, or an outcome of an experiment, is chosen by random from the list.

The default values for both players are interactive, so if you run the simulation mode without any strategy specification, you will be first asked to select an experiment and then to select its outcome.

Strategy analysis mode [a, analysis]

```
./cobra -m analysis -e <strategy> <input file>
```

In the analysis mode the tool computes the worst-case number of experiments and the average-case number of experiments needed by a given codebreaker's strategy to reveal the valution of the variables. Supported strategy are the same as in the simulation mode.

The algorithm for this task has been described in Algorithm 4.14. Two variants on the algorithm have been presented, one with and one without the symmetry detection. By default, the symmetry detection in turned on but can be turned off with `--no-symmetry` switch.

Optimal strategy mode [ow, optimal-worst, oa, optimal-average]

```
./cobra -m optimal-worst [--opt-bound <double>] <input file>
./cobra -m optimal-average [--opt-bound <double>] <input file>
```

In the optimal strategy mode, the tool computes either the worst-case optimal, or the average-case optimal strategy for a given code-breaking game.

The algorithm for this purpose has been described in Algorithm 4.17. An upper bound can be specified with `--opt-bound` switch. In some cases, this can significantly speed up the process.

Note that the optimal strategy synthesis is currently very slow even for small instances of code-breaking games. Further optimizations would be necessary for the tool to synthesise the optimal strategy for Mastermind with 4 pegs and 6 colors in a reasonable time.

5.4 Modularity and extensibility

COBRA uses external tools for SAT solving and graph canonization. Nowadays, many high-performance SAT solvers are available and multiple tools for graph

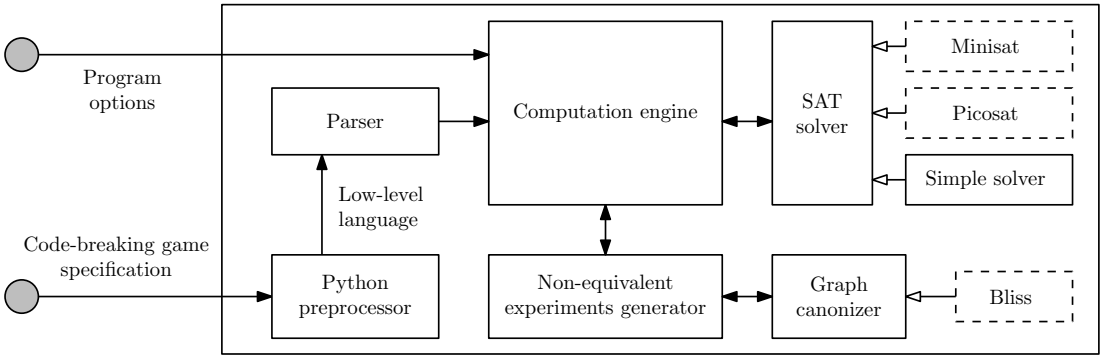

Figure 5.6: Component diagram of COBRA.

canonization exists. COBRA was developed so that other external tools with the same functionality can be easily integrated. Figure 5.6 shows a component diagram of the modular design of COBRA.

Detailed requirements on a SAT solver and on a graph canonization tool are listed in the next sections.

If you want to try another SAT solver, alter the algorithm for model counting, or test any other modification, you can implement your own solver class that inherits from `Solver` and implements all the necessary methods.

Check the `solver.h` file and the documentation therein for the information about the exact functions required. Further, you need to to include the file with your class in `main.cpp` and add a new case into the `get_solver` function in this file.

COBRA can be easily extended with a new strategy or heuristics to selects experiments. All available strategies are implemented in `strategy.h/strategy.cpp` file. If you want to add a new strategy, create a corresponding function in this file and add an entry about the strategy to the `breaker_strategies` table in `strategy.h`. A strategy function takes a list of experiments as the only argument and returns the index of the selected one.

If the strategy is one-step look-ahead, you can use a provided template with a corresponding lambda function. We demonstrate this possibility with a code snippet of the exact implementation of the *exp-models* strategy below. For exact details, see the documentation in the file.

```
uint breaker::exp_num(vec<Option>& options) {
  return minimize([](Option& o){
    auto models = o.GetNumOfModels();
    int sumsq = 0;
    for (uint i = 0; i < o.type().outcomes().size(); i++) {
      sumsq += models[i] * models[i];
    }
    return (double)sumsq / o.GetTotalNumOfModels();
  }, options);
```

```
}
```

5.5 SAT solving

COBRA uses a SAT solver for the following tasks.

- Compute the total number of possible codes.
- Verify that an experiment is well-formed (see Section 5.3 for details).
- Identify satisfiable outcomes of an experiment and disregard the others.
- Decide whether the game is finished – whether the accumulated knowledge as a formula has only one model.
- Evaluate the strategies – count models, fixed variable, etc.

Most of these tasks require an *incremental SAT solver*, i.e. a sat solver to which you can add constraints and take them back later. Without this feature, we would have to call the solver from a clean state many times on the whole formula, which would ruin the computation time.

COBRA uses a SAT solver as an abstract class, which can have multiple implementations. This allows a simple extension with another SAT solver. The solver to be used can be specified with the `-b` or `--backend` switch.

Solver must implement the following methods:

- AddConstraint(formula). Adds a constraint to the SAT solver.
- Satisfiable() → Bool. Decides whether the current formula is satisfiable.
- GetAssignment() → Assignment. After Satisfiable call, this function retrieves a satisfying assignment from the solver.
- OpenContext(), CloseContext(). This is our understanding of incremental SAT solving. OpenContext adds a context to a stack. Every call to AddConstraint adds the constraint to the current context. CloseContext removes all the constraint in the current context and removes it from the stack. It must be possible to nest contexts arbitrarily. The two methods are sometimes called just Push and Pop.
- HasOnlyOneModel() → Bool. Decides whether to formula has only one model. This can be implemented by asking whether the formula is satisfiable, if yes, retrieving the satisfying assignment, adding a clause with the assignment negated and asking for satisfiability again. Adding the new clause should be done in a new context in order not to pollute the solver state. The pseudocode is shown in Algorithm 5.7.
- CountModels() → Int. SAT solvers do not typically include support for model counting, the problem commonly referred to as #SAT. One solution

is to use special tools designed for this purpose, such as SharpSAT[6] [36]. However, these tools do not support incremental solving and must be run from a clean state for each formula models of which we want to count.
Second option is to use a SAT solver, repeatedly ask for satisfiability and add clauses that forbids the current assignment until we get an unsatisfiable formula. The pseudocode is shown in Algorithm 5.8.
Third option is to use a SAT solver and a simple backtracking approach, progressively assuming a variable to be true or false and cut the non-perspective branches. The pseudocode is shown as a recursive function in Algorithm 5.9.

Algorithm 5.7: Decision whether a formula has exactly one model.

```
1 if not Satisfiable() then return false
2 v ← GetAssignment()
3 OpenContext()
4 AddConstraint(x̄_1 | ... | x̄_n), where x̄_i is ¬x_i if v(x_i) = 1 and x_i otherwise
5 sat ←Satisfiable()
6 CloseContext()
7 return ¬sat
```

Algorithm 5.8: Model counting, second option.

```
1 models ← 0
2 OpenContext()
3 while Satisfiable() do
4 |   v ← GetAssignment()
5 |   AddConstraint(x̄_1 | ... | x̄_n), where x̄_i is ¬x_i if v(x_i) = 1 and x_i
  |   otherwise
6 |   models ← models + 1
7 CloseContext()
8 return models
```

COBRA includes three solver implementations which we describe next.

PicoSat

Picosat[7] [37] is a simple, extensible SAT solver, which supports incremental SAT solving exactly in the way we need.

6. https://sites.google.com/site/marcthurley/sharpsat
7. http://fmv.jku.at/picosat/

Algorithm 5.9: Model counting, third option.

```
1  models ← 0
2  x ← any variable from vars
3  OpenContext()
4  AddConstraint(x)
5  if Satisfiable() then models ← models + count(vars ∖ {x})
6  CloseContext()
7  OpenContext()
8  AddConstraint(¬x)
9  if Satisfiable() then models ← models + count(vars ∖ {x})
10 CloseContext()
11 return models
```

Bindings to Picosat are implemented in `PicoSolver` class. This class also implements model counting algorithm 5.9 as Picosat does not support model counting itself.

MiniSat

Minisat[8] [38] is a minimalistic, extensible SAT solver that won several SAT competitions in the past.

Minisat does not support incremental SAT solving in the manner we described, but it supports assumptions. You can assume arbitrary number of unit clauses (i.e. that a variable is true of false) and ask for satisfiability under those assumptions.

The behaviour we want can be simulated by assumptions in the following way. For each context, we create a new variable, say a. Then, instead of adding clauses $C_1, C_2, \ldots, C_n$ to the context, we add clauses $\{\neg a, C_1\}$, $\{\neg a, C_2\}$, ... $\{\neg a, C_n\}$ and ask for satisfiability under the assumption a (in general, assumption that all variables of open contexts are true). Afterwards, when a context is closed, we add a unit clause $\{\neg a\}$, which effectively removes all the clauses added in the context. The only minor issue with this approach is that the variable a is wasted, the solver will remember it somewhere and may consume more memory.

Bindings to Minisat are implemented in `MiniSolver` class. This class implements the context opening and closing in the way described above and also model counting algorithm 5.9.

8. http://minisat.se/

Simple solver

We include a special SAT solver, called `SimpleSolver` to show that the usage of a proper SAT solver in this application is justifiable. Simple solver uses another SAT backend (Picosat) to generate all models of the initial constraint φ_0 (or of the first constraint, in general). Later satisfiability questions with additional constraint are resolved by going though all possible codes (assignments) and checking that the constraints are satisfied. Model counting and the other functions are implemented similarly.

Transformation to CNF

The input formula for a SAT solver must be typically specified in the conjunctive normal form(CNF). As we do not have such requirement for formulas in the input format, and allow non-standard numerical operators, we need to transform a formula to CNF first.

The standard transformation works as follows. First, we express the formula in a form that uses only negations, conjunctions and disjunctions as operators. Then, we transform it to *negation normal form* using De Morgan's laws and, finally, we use distributivity of conjunction and disjunction to move all the conjunction to the top level. However, this may lead to an exponential explosion of the formula, so another solution, called *Tseitin Transformation*, is commonly used when converting a formula for a SAT solver[39].

Imagine the formula as a circuit with gates corresponding to the logical operators. Input vectors correspond to variable assignments and the circuit output is true if and only if the input assignment satisfies the formula.

For each gate, a new variable representing its output is created. The resulting formula is a conjunction of sub-formulas that enforce the proper operation of the gates.

For example, consider an AND gate, inputs of which corresponds to variables x, y and output corresponds to a variable w. We need to ensure that w is true if and only if both x and y are true, which is done by adding a sub-formula $w \leftrightarrow (x \wedge y)$, which can be expressed in CNF as

$$(\neg x \vee \neg y \vee w) \wedge (x \vee \neg w) \wedge (y \vee \neg w).$$

Other gates types are handled similarly and this is done for all gates in the circuit. Finally, the variables corresponding to the result of the top level operator is added to the resulting formula as a unit clause.

It remains to explain how we handle the numerical operators ATLEAST, ATMOST and EXACTLY. We show it on the $\text{EXACTLY}_k\ (f_1, \ldots, f_n)$ operator, others are transformed similarly. For simplicity, assume f_i are variables; if not, we take the variable corresponding the the sub-formulas.

For each $l \in \{0, 1, \ldots, k\}$ and $m \in \{1, \ldots, n\}$, $l \leq m$, we create a new variable $z_{l,m}$ which will be true if and only if the formula $\text{EXACTLY}_l(f_1, \ldots, f_m)$ is satisfied. To enforce this assignment, we add sub-formulas of the form

$$z_{l,m} \leftrightarrow (f_m \wedge z_{l-1,m-1}) \vee (\neg f_m \wedge z_{l,m-1})$$

for each $l > 1$, $m > 1$ (in CNF form, of course). Special cases $l = m$ and $l = 0$ are equivalent to AND and OR formulas, respectively, and are handled accordingly. The size of the resulting sub-formula is linear in $n \cdot k$. Although this is not polynomial in the size of the input (supposing k is encoded in binary form) but it is much better than the naive solution to express the formula as a conjunction of the $\binom{n}{k}$ possibilities, which would be double exponential.

5.6 Graph isomorphism

To implement the suggested method for detection of equivalent experiments, we need to solve the graph isomorphism problem, i.e. decide whether two given graphs are isomorphic. This problem is famous for not being proven either P-complete or NP-complete, so no polynomial algorithm for the problem is known. However software tools are available for graph canonization, which are quite efficient for sparse graphs and can be used to decide graph isomorphism by comparison of the canonical forms of the graphs.
Nauty[9][40] and Bliss[10][41] are the most well-known tools for this purpose. These programs are primarily designed to compute automorphism groups of graphs but they can produce a canonical labelling of the graph as well. For various reasons including simple integration, we decided to integrate Bliss. For comparison of Nauty and Bliss, read an overview of the algorithms used by these tools in [42] and see benchmarks on the Nauty's website.

5.7 Implementation details

Programming Language and Style

Since the problems we solve are computationally very demanding, we had to choose a high-performing programming language. The external tools we use, especially SAT solvers, are typically written in C/C++, so C++ was a natural choice for our tool. COBRA is written in the latest standard of ISO C++, C++11, which contains significant changes both in the language and in the standard libraries

9. http://pallini.di.uniroma1.it
10. http://www.tcs.hut.fi/software/bliss

and, in our opinion, improves readability and programmer's efficiency compared to previous versions.

We wanted the style of our code to be consistent and to use the language in the best manner possible according to industrial practice. From the wide range of style guides available online we chose *Google C++ Style Guide*[43] and made the code compliant with all its rules except for a few exception. The only significant violation are lambda functions, which are forbidden due to various reasons, but we think they are more beneficial than harmful in this project.

Requirements

Usage of a modern language requires a modern compiler that supports all the C++11 features we use. We have successfully tested compilation with `gcc` version 4.8.2 and `clang` version 3.2.

The tool is platform independent. We have been able to successfully compile and test the functionality of the tool on Linux (Ubuntu 14.04) and Mac OS X (10.9).

Testing

Correctness is automatically a top priority for a tool of this kind so we implemented two automatic testing methods to capture potential programmer's error as soon as possible.

Unit testing has became a popular part of software development process in the last decades and is very effective for testing the functionality of particular modules and functions. From the large amount of unit tests frameworks available for C++, we have chosen *Google Test*[11], because of its simplicity, minimal amount of work needed to add new tests and very good assertion support. The unit tests are automatically compiled and executed if you run `make utest` in the tool directory.

Functional tests provide a great method to test end-to-end functionality of the software. These tests execute the program on sample inputs and compare the output with their expectations. We have implemented several functional tests for the most common operations with a small instance of Mastermind. They can be run by `make ftest` in the tool directory.

11. https://code.google.com/p/googletest/

6 Experimental results

In this chapter, we present experimental results of our tool on several code-breaking games. We compare running times of the tool on various tasks with different SAT solvers and evaluate presented one-step look-ahead strategies.

In the tables below, MM(n,m) refers to Mastermind with n pegs and m colours, CC(n) refers to the counterfeit coin problem with n coins, BG(n,m) refers to Bags of gold problem with n bags and balance scale capacity m and SM(n,m) refers to the Mastermind variant with black markers only, i.e. for each guess, the codebreaker gets the number of positions at which the code and the guess match.

6.1 Performance

All experiments have been run on Intel Core i7-3770 3.40GHz and COBRA was compiled with `gcc` 4.8.2. The symmetry breaking engine has been turned off for this section, so that the differences between SAT solvers become more apparent.

The numbers in the following tables are running times in seconds with respective SAT solvers. The last column ("# calls") states the total number of calls to the SAT solver during the task. Model counting and counting the number of fixed variables is considered one call.

Table 6.1 lists running times of the well-formed check of several code-breaking games. Naturally, proper SAT solvers are orders of magnitude faster than simple solver as well-formed check is based on verification of unsatisfiability of a (relatively) large formula.

Game	Simple	Minisat	Picosat	# calls
MM(4,6)	174	**2.7**	3.2	1,296
MM(5,4)	217	**14.3**	34.8	1,024
BG(12,12)	3,539	**0.5**	1.1	271

Table 6.1: Running times (in seconds) of the well-formed check.

The first two lines of Table 6.2 show execution times of the first experiment selection in Mastermind with 4 pegs and 6 colours. The last two lines of the table list running times of the simulation of respective strategies on EDEE code.

As can be seen from the numbers, evaluation of "parts" strategy is slightly faster with Minisat than with simple solver. However, since our model counting algorithm implemented on top of Minisat and Picosat is naive and unoptimized, "max-models" strategy evaluation is significantly faster with simple solver.

Task	Simple	Minisat	Picosat	# calls
Select first exp. (parts)	3.9	**2.6**	523	17,108
Select first exp. (max-models)	**9.3**	45.3	> 5,000	3,145
Simulate (parts on EDEE)	10.5	**7.9**	749	92,644
Simulate (max-models on EDEE)	**15.1**	32.3	3,024	31,061

Table 6.2: Running times (in seconds) of simulation and strategy evaluation on MM(4,6).

Game	Strategy	Simple	Minisat	Picosat	# calls
MM(3,4)	parts	0.1	0.4	13.5	13,769
MM(3,4)	max-models	0.1	2.1	179	9,349
MM(3,4)	exp-fixed	0.3	17.4	1,974	15,002
MM(4,4)	parts	6.9	237	> 5,000	260,144
MM(4,4)	max-models	5.0	1,126	> 5,000	188,828
MM(4,4)	exp-fixed	24	> 5,000	> 5,000	329,820
CC(20)	parts	0.1	0.3	12.8	7,718
CC(20)	max-models	0.2	28	3,161	12,117
CC(20)	exp-fixed	0.3	73	> 5,000	14,273

Table 6.3: Running times (in seconds) of strategy analysis.

Table 6.3 shows running times of strategy analysis. A clear winner in this case is the simple solver based on model enumeration, which is not unexpected. On lower levels of the backtracking algorithm, the analysed formulas have only very few models and the overhead with calling a proper SAT solver is greater than the naïve approach of a simple solver.

The results also show that model counting is harder than satisfiability questions. The difference is not that significant for simple solver but, as we have already mentioned, our model counting algorithm is much slower. That is the reason why the analysis of "max-models" strategy takes Minisat and Picosat much more time than analysis of "parts" strategy.

To summarize the results of this section, Picosat turned out to be very slow on instances of this size. Minisat proved to be useful for strategy evaluation in the first rounds, when the number of possible models of the formula is relatively large. Simple solver is a clear winner for strategy analysis.

A question arises whether we can benefit from a hybrid approach of using a SAT solver in the first steps and switching to the simple solver when the number of possibilities shrinks. This was, however, beyond the scope of this thesis.

6.2 One-step look-ahead strategies

In this section, we compare performance of one-step look-ahead strategies defined in Section 3.4 on the counterfeit coin problem, Mastermind, and Mastermind with black markers only.

The counterfeit coin problem

The average-case number of experiments performed by one-step look-ahead strategies in the counterfeit coin problem for the number of coins from 3 to 40 is shown in Figure 6.4.

Notice that larger number of coins does not necessarily mean that identifying the counterfeit coin is more difficult. For example, the "max-models" strategy needs 3.44 experiments on average to identify the counterfeit coin among 16 coins but only 3.41 if the number of coins is 17. That is because the additional coin changes the number of models of some outcomes, which may lead to better experiment selection.

Interestingly, "exp-fixed" strategy outperforms all others on 20, 21, 23, 25 and 27 coins, while it seems to be generally worse. Strategies "parts" and "min-fixed" are clearly unsuitable for a problem of this kind.

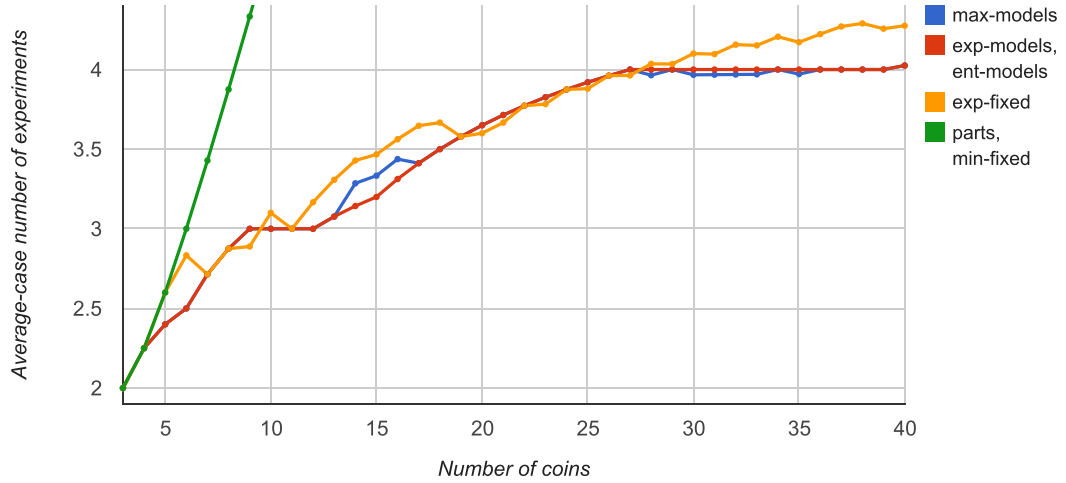

Figure 6.4: Average-case number of experiments in the counterfeit coin problem.

Mastermind

Results for Mastermind are shown in Table 6.5, rounded to three decimal places. A clear winner in the average case is "parts" strategy, closely followed by "max-models".

In the worst case, "max-models" outperforms other strategies in all cases except MM(3,2). In this case, "max-models" is unlucky in the initial state and selects guess AAA, which has the same maximal number of models over the outcomes as AAB but is lexicographically smaller. All other strategies select AAB, which turns out to be a better choice in this case.

Again, notice that larger size of the problem does not necessarily mean that revealing the secret is more difficult, as can be seen on the values for MM(5,2) and MM(5,3).

Game	max-mod		parts		exp-mod		ent-mod		min-fix		exp-fix	
MM(2,3)	2.667	4	2.333	3	2.444	3	2.444	3	2.667	4	2.444	3
MM(2,6)	3.667	5	3.667	5	3.861	5	3.861	5	4.611	7	4.167	6
MM(3,2)	2.625	4	2.250	3	2.250	3	2.250	3	2.625	4	2.25	3
MM(3,6)	4.046	5	3.977	5	4.227	5	4.218	5	5.259	8	4.546	6
MM(3,8)	4.787	6	4.701	6	4.879	6	4.844	6	6.688	10	5.631	8
MM(4,2)	2.750	4	2.750	4	3.063	4	3.063	4	3.250	5	3.063	4
MM(4,6)	4.476	5	4.374	6	4.626	6	4.643	6	5.765	9	5.231	7
MM(4,7)	4.837	6	4.743	6	4.962	6	4.947	6	6.476	10	5.945	8
MM(4,8)	5.183	6	5.102	7	5.293	7	5.272	7	7.213	11	6.410	9
MM(5,2)	3.500	5	3.313	5	3.938	5	3.625	5	3.875	6	3.781	5
MM(5,3)	3.407	4	3.379	4	3.634	4	3.609	4	4.444	7	3.942	5
MM(5,4)	3.991	5	3.880	5	4.092	5	4.083	5	5.014	9	4.617	6

Table 6.5: Average-case and worst-case number of experiments of one-step look-ahead strategies in Mastermind.

Mastermind with black markers only (string matching)

Mastermind with black markers only is an example of a code-breaking game, where "max-models" does not perform so well. Exact values rounded to two decimal points are shown in Table 6.6.

The best one-step look-ahead startegy for this game is "ent-models", the strategy based on the entropy of the number of models, closely followed by "parts" strategy.

Game	max-mod		parts		exp-mod		ent-mod		min-fix		exp-fix	
SM(3,3)	3.15	4	2.89	4	2.89	4	2.89	4	3.52	4	3.52	4
SM(3,6)	6.1	8	5.58	7	5.74	8	5.53	7	8.3	13	8.28	13
SM(3,12)	10.76	14	10.28	13	10.5	14	10.23	13	17.41	31	13.3	22
SM(6,3)	4.94	6	4.5	7	4.5	6	4.47	6	6.5	7	6.16	7
SM(6,6)	8.61	12	8.2	12	7.99	11	7.93	11	15.8	25	15.75	25

Table 6.6: Average-case and worst-case number of experiments of one-step look-ahead strategies in Mastermind with black markers only.

7 Conclusions

We presented a general model of code-breaking games based on propositional logic, which can fit Mastermind, the counterfeit coin problem and many others. Experiment equivalence was introduced and we proved that equivalent experiments can be disregarded during strategy analysis and optimal strategy synthesis. We suggested an algorithm for equivalence testing based on graph isomorphism.

A computer language for code-breaking game specification was introduced and we developed a computer program that can perform various tasks with a given code-breaking game. Using the tool, we reproduced some of the existing results for Mastermind, analysed other code-breaking games and evaluated strategies for experiment selection based on the number of fixed variables.

There are many more interesting things to try in this framework. We present a few suggestions for future work in the next paragraphs.

First, our code-breaking game model can be further generalized in many ways. Numerous possibilities arise if we allow experiments to have different costs.

Imagine Mastermind with another type of experiment that directly tells you a colour at a specified position. What price must the new experiment have so that it is worth using it, given that the standard guess has unit cost?

Second, one-step look-ahead strategies provide us with a simple heuristics to select experiments. However, if some experiments are assigned the same value, the lexicographically smaller experiment is chosen, which is not very reasonable. *Randomized strategies*, where the experiment is selected from the experiments with the best value with uniform distribution may lead to many interesting results.

Third, look-ahead strategies can be naturally extended to more than one step. This would lead to the *minimax algorithm* applied to the tree of possible outcomes and experiments in the next rounds. Evaluation of such strategies would be much more computationally demanding. Would their performance be significantly better?

Finally, several completely different approaches for strategy synthesis were suggested for Mastermind. In particular, genetic algorithms proved to be very useful for problems of larger sizes as they scale much better than the backtracking approach. Can we apply these methods in our model?

For now, these interesting questions remain open. We hope they will lead to further research in this area.

Bibliography

[1] Shan-Tai Chen, Shun-Shii Lin, and Li-Te Huang. “A two-phase optimization algorithm for Mastermind”. In: *The Computer Journal* 50.4 (2007), pp. 435–443.

[2] JJ Merelo-Guervós, P Castillo, and VM Rivas. “Finding a needle in a haystack using hints and evolutionary computation: the case of evolutionary MasterMind”. In: *Applied Soft Computing* 6.2 (2006), pp. 170–179.

[3] Howard D Grossman. “The twelve-coin problem”. In: *Scripta Mathematica* 11 (1945), pp. 360–363.

[4] Freeman J Dyson. “1931. The Problem of the Pennies”. In: *The Mathematical Gazette* (1946), pp. 231–234.

[5] Richard K Guy and Richard J Nowakowski. “Coin-weighing problems”. In: *American Mathematical Monthly* (1995), pp. 164–167.

[6] Ratko Tošić. “Two counterfeit coins”. In: *Discrete Mathematics* 46.3 (1983), pp. 295–298.

[7] Anping Li. “Three counterfeit coins problem”. In: *Journal of Combinatorial Theory, Series A* 66.1 (1994), pp. 93–101.

[8] László Pyber. “How to find many counterfeit coins?” In: *Graphs and Combinatorics* 2.1 (1986), pp. 173–177.

[9] Xiao-Dong Hu, PD Chen, and Frank K. Hwang. “A new competitive algorithm for the counterfeit coin problem”. In: *Information Processing Letters* 51.4 (1994), pp. 213–218.

[10] Martin Aigner and Anping Li. “Searching for counterfeit coins”. In: *Graphs and Combinatorics* 13.1 (1997), pp. 9–20.

[11] Axel Born, Cor AJ Hurkens, and Gerhard J Woeginger. “How to detect a counterfeit coin: Adaptive versus non-adaptive solutions”. In: *Information processing letters* 86.3 (2003), pp. 137–141.

[12] Andrzej Pelc. “Searching games with errors—fifty years of coping with liars”. In: *Theoretical Computer Science* 270.1 (2002), pp. 71–109.

[13] A Pelc. “Detecting a counterfeit coin with unreliable weighings”. In: *Ars Combinatoria* 27 (1989), pp. 181–192.

[14] Wen-An Liu, Qi-Min Zhang, and Zan-Kan Nie. “Searching for a counterfeit coin with two unreliable weighings”. In: *Discrete applied mathematics* 150.1 (2005), pp. 160–181.

[15] Annalisa De Bonis, Luisa Gargano, and Ugo Vaccaro. “Optimal detection of a counterfeit coin with multi-arms balances”. In: *Discrete applied mathematics* 61.2 (1995), pp. 121–131.

[16] Tanya Khovanova. “Parallel Weighings”. In: *arXiv preprint arXiv:1310.7268* (2013).

[17] Vasek Chvátal. “Mastermind”. In: *Combinatorica* 3.3-4 (1983), pp. 325–329.

[18] Jeff Stuckman and Guo-Qiang Zhang. “Mastermind is NP-complete”. In: *arXiv preprint cs/0512049* (2005).

[19] Donald E Knuth. “The computer as master mind”. In: *Journal of Recreational Mathematics* 9.1 (1976), pp. 1–6.

[20] Robert W Irving. “Towards an optimum Mastermind strategy”. In: *Journal of Recreational Mathematics* 11.2 (1978), pp. 81–87.

[21] E Neuwirth. “Some strategies for Mastermind”. In: *Zeitschrift für Operations Research* 26.1 (1982), B257–B278.

[22] Barteld P Kooi. “Yet Another Mastermind Strategy.” In: *ICGA Journal* 28.1 (2005), pp. 13–20.

[23] Kenji Koyama and Tony W Lai. “An optimal Mastermind strategy”. In: *Journal of Recreational Mathematics* 25.4 (1993), pp. 251–256.

[24] Geoffroy Ville. “An Optimal Mastermind (4, 7) Strategy and More Results in the Expected Case”. In: *arXiv preprint arXiv:1305.1010* (2013).

[25] Lotte Berghman, Dries Goossens, and Roel Leus. “Efficient solutions for Mastermind using genetic algorithms”. In: *Computers & operations research* 36.6 (2009), pp. 1880–1885.

[26] Alexandre Temporel and Tim Kovacs. “A heuristic hill climbing algorithm for Mastermind”. In: *UKCI’03: Proceedings of the 2003 UK Workshop on Computational Intelligence, Bristol, United Kingdom*. 2003, pp. 189–196.

[27] Alexey Slovesnov. *Search of optimal algorithms for bulls and cows game*. 2013. URL: http://slovesnov.users.sourceforge.net/bullscows/bullscows.pdf (visited on 04/20/2014).

[28] Wayne Goddard. “Static mastermind”. In: *Journal of Combinatorial Mathematics and Combinatorial Computing* 47 (2003), pp. 225–236.

[29] Paul Erdős and Alfréd Rénei. *On two problems of information theory*. 1963. URL: http://193.224.79.10/~p_erdos/1963-12.pdf (visited on 04/20/2014).

[30] Michael T Goodrich. “The mastermind attack on genomic data”. In: *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE. 2009, pp. 204–218.

[31] Julien Gagneur, Markus C Elze, and Achim Tresch. “Selective phenotyping, entropy reduction, and the mastermind game”. In: *BMC bioinformatics* 12.1 (2011), p. 406.

[32] Riccardo Focardi and Flaminia L Luccio. “Guessing bank pins by winning a mastermind game”. In: *Theory of Computing Systems* 50.1 (2012), pp. 52–71.

[33] Wikipedia. *Black Box (game) — Wikipedia, The Free Encyclopedia*. 2014. URL: http://en.wikipedia.org/wiki/Black_Box_(game) (visited on 04/20/2014).

[34] Jonathan H. Liu. *Laser Maze: A Delightful Puzzle Game*. 2013. URL: http://geekdad.com/2013/06/laser-maze (visited on 04/20/2014).

[35] Board game geek. *Code 777 (1985)*. URL: http://boardgamegeek.com/boardgame/443/code-777 (visited on 04/20/2014).

[36] Marc Thurley. “sharpSAT–counting models with advanced component caching and implicit BCP”. In: *Theory and Applications of Satisfiability Testing-SAT 2006*. Springer, 2006, pp. 424–429.

[37] Armin Biere. “PicoSAT Essentials.” In: *JSAT* 4.2-4 (2008), pp. 75–97.

[38] Niklas Eén and Niklas Sörensson. “An extensible SAT-solver”. In: *Theory and applications of satisfiability testing*. Springer. 2004, pp. 502–518.

[39] Wikipedia. *Tseitin transformation — Wikipedia, The Free Encyclopedia*. 2014. URL: http://en.wikipedia.org/wiki/Tseitin_transformation (visited on 04/27/2014).

[40] Brendan D. McKay and Adolfo Piperno. “Practical graph isomorphism, {II}”. In: *Journal of Symbolic Computation* 60 (2014), pp. 94–112. ISSN: 0747-7171. DOI: http://dx.doi.org/10.1016/j.jsc.2013.09.003. URL: http://www.sciencedirect.com/science/article/pii/S0747717113001193.

[41] Tommi A Junttila and Petteri Kaski. “Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs.” In: *ALENEX*. Vol. 7. SIAM. 2007, pp. 135–149.

[42] Hadi Katebi, Karem A Sakallah, and Igor L Markov. “Graph symmetry detection and canonical labeling: Differences and synergies”. In: *arXiv preprint arXiv:1208.6271* (2012).

[43] Google. *Google C++ Style Guide*. 2013. URL: http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml (visited on 04/20/2014).

Contents of the electronic attachment

An archive with source codes of COBRA is available in the thesis repository in IS MU, available online at https://is.muni.cz/th/359972/fi_m/.
The archive contains the following files and directories.

`examples/`	Directory with sample code-breaking game specifications, including Mastermind, the counterfeit coin problem and others.
`src/`	Soures codes of COBRA.
`test/`	Unit tests and functional tests.
`tools/`	Directory with source codes of external tools, Picosat release 957, Minisat version 2.2 and Bliss version 0.72.
`cobra`	The main executable.
`makefile`	Makefile with compilation rules and dependencies.
`LICENSE`	Full test of BSD License.
`README`	Short description of COBRA with basic information.