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 Information derived from the published or unpublished work of others has been acknowledged in the text and a list of references is given.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.

Keywords

code braking games,
deductive games,
strategy synthesis,
greedy strategy,
SAT solving,
model counting,
mastermind,
counterfeit coin

Abstract

Contents

1 Introduction

Code breaking games (also known as Deductive games or Searching games) are games for two players, in which the first, usually reffered to as *the codemaker*, chooses a secret code from a given set, and the second, reffered to as *the codebreaker*, strives to reveal the code by a series of experiments that give him partial information about the code.

The famous board game of Mastermind is a prominent example. ...

Anothe example is the Counterfeit coin problem, the problem of determining a counterfeit coin among authentic ones using just a balance scale. Here, the codemaker is not a real player. The balance scale takes his function and evaluates the experiments – weighings – permfomred by the codebreaker.

Numerous other examples can be found among various board games and logic puzzles, some of which are presented in Chapter 2.

Although Mastermind and the Conterfeit coin problem have been subjected to heavy research, few have been written about Code Breaking Games in general. Some authors suggested general methods (and applied them one of the games), some vaguely stated that their approach can be applied to other games of this kind but, to the best of our knowledge, no one have tried to formalize and give some general results for these games in general.

Here comes this thesis to fill the gap. We develop a formalism ...

2 Studied Games and Known Results

We introduce a few examples of code-breaking games in this chapter. The Counterfeit Coin problem and Mastermind game 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.

Our goal in this work is neither to answer the research questions nor to study possible generalizations. We aim to create a general formalism and a computer language which could be used to describe arbitrary code-breaking game, if possible. This chapter provides an overview of what we had in mind when we designed the framework and the language described in the rest of the thesis.

2.1 The Counterfeit Coin

The problem of finding a counterfeit coin among regular coins in the fewest number of weighings on a pair of scales balance is a folklore of recreational mathematics.

In all problems of this kind, you can use the scales only 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 3 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 scales (illustrative image)[1].

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

This problem is very easy as one can use *ternary search* algorithm. In short, we divide the coins into thirds, put one third against another on scales. If both sides weigh the same, the counterfeit coin must be in the last third, otherwise it must be in the lighter third. In this way, the size of the search space reduces by a factor of 3 in each step, which is clearly optimal.

In 1940s, a more complicated version was introduced by Grossman[1].

Problem 2.3 (The Twelve Coin Problem). *You are given $n \geq 3$ (typically 12) coins, exactly one of which is counterfeit, but it is not known if it is heavier or*

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

lighter. Determine the unique coin and its weight relative to others in the minimal number of weighings.

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

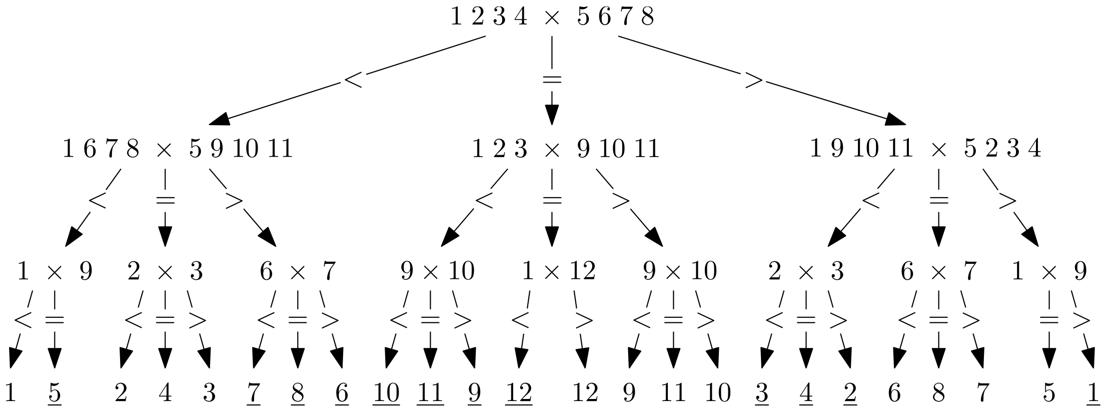

Figure 2.4: Decision tree for The Twelve Coin Problem.
Leaf x means that the coin number x is lighter, $\underline{x}$ means that the coin number x is heavier.

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, [2]). *There exists a scheme that determines the counterfeit coin and its type 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[2] here because of its elegant combinatorial idea. We show a scheme 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 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 labeling has the property that you can get one label of a coin from the other 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 done similarly with some modifications to the labeling. 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 detected for $n > \frac{1}{2}(3^w - 3)$ can be done using information theory. ■

Generalizations and related research

Naturally, the problem was generalized in various ways and studied by many authors. In “Coin-Weighing Problems”[3], 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[4][5], then with m counterfeit coins in general[6]. Some authors studied the problem for unknown number of counterfeit coins [7], or for *at most* m counterfeit coins[8].

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 solve in just 3 weighings[2].

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 fixed, showing that there always exists an optimal scheme for Problem 2.3 which is non-adaptive[9].

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 [10]. It was applied on the counterfeit coin problem (Problem 2.2 variant) in [11] with at most one erroneous outcome or in [12] with two.

Multi-arm balance. In this variation, your balance has k arms. You put the same number of coins on every arm and you get either the information that all weigh the same or which arm is lighter or heavier than others[13].

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[14]. 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 classical 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 colored 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 color 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 with minimal number of guesses.

More formally, let C be a set of colors of size c. Define a distance $d : C^n \times C^n \to \mathbb{N}_0 \times \mathbb{N}_0$ of two color 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

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

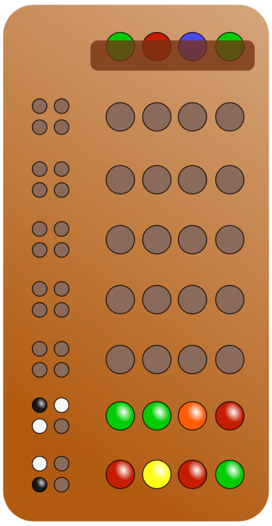

Figure 2.7: Mastermind game (illustrative image)[2].

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 pegs in the code (represented by $(0, i)$ for $1 \leq i \leq n$) and pegs in the guess (represented by $(1, i)$ for $1 \leq i \leq n$). Let M be a maximal matching such that

1. an edge connects only pegs of the same color, i.e. if $((0, i), (1, i)) \in M$, then $h[i] = g[j]$, and
2. if $h[i] = g[i]$ then $((0, i), (1, i)) \in M$.

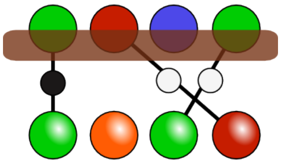

Figure 2.8: Guess evaluation by maximal matching.

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

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 $(0, i)$ with $(1, i)$ for some i.

Known results and related research

Much research has been done on this game, authors focusing on *exact values*, *asymptotics* (e.g. [15]), 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[16].

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[17] proposes a strategy that chooses a guess that minimizes the maximal number of remaining possibilities over all possible responses by the codemaker. This strategy requires at most 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[18] suggested minimizing the expected number of remaining possibilities, Neuwirth[19] maximized the entropy of the number of possibilities in the next round. Much later, Kooi[20] came up with a simple strategy that maximizes the number of possible responses by the codemaker, which is computationally easier and performs better that the previous two.

Strategy	First guess	Expected-case	Worst-case
Maximal num.	AABB	4.476	5
Expected num.	AABC	4.395	6
Entropy	ABCD	4.416	6
Most parts	AABC	4.373	6
Exp-case optimal	AABC	4.340	6

Table 2.9: Comparison of one-step look-ahead strategies. Data from [21] and [20].

Using a backtracking algorithm, Koyama and Lai [22] found the optimal strategy for the expected case, which requires 4.34 guesses on average. The comparison of the described strategies is shown in Table 2.9.

Apart from *one-step look-ahead* policies, which are, in general, computationally intensive and do not scale well for bigger n or c, other approaches were suggested.

Many authors tried to apply genetics algorithms (see [23] for an exhaustive overview and references therein), other analyzed various heuristic methods (e.g. [24]).

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 colors and does not allow repetitions. Slovesnov wrote an exhaustive analysis of the problem, see [25].

Static mastermind is a variation of the game in which all guesses must be made at 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[15] and partially solved (for $n \leq 4$) by Goddard [26], proving that for 4 pegs and k colors, 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 [27], 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 [28][29].

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*[30]. In this variation, a guess is not just a sequence of colors, but a sequence of sets of colors. For example, if we have six colors $\{A, B, C, D, E, F\}$ 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 Problems

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[31].
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).

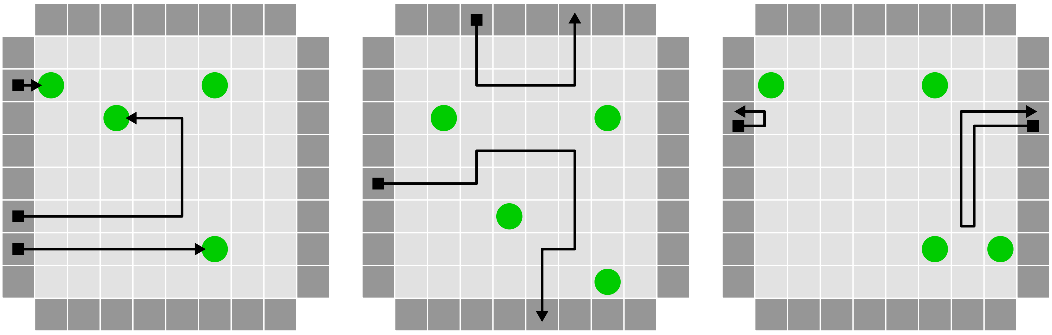

Figure 2.10: Illustration of the rules of Black Box game[3].

A few examples are shown in Figure 2.10. The first image shows cases in which the ray hits the marble, the second shows rays deflected multiple times, emerging from the box in 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 2.11 shows an example of such problematic configuration.

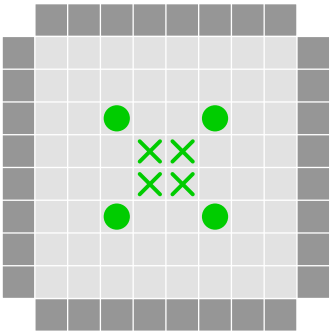

Figure 2.11: An example of ambigous configuration[3].

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

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 (and, for 5 or more marbles, it may be even 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 was used in other board games such as *Laser maze*[32].

Code 777

During the board game named *Code 777*, players sit in a circle, each drawing three cards in the beginning. Players must not look at their own cards but they put them in a rack in front of them so that all other players can see them. Each card has one of seven colors 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[33].

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 players goal is 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, color, 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. 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 old 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 only one 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, it is the x-th bag.

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 only one coin is studied in [27], 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 made.

3 Formal model

3.1 Notation and Terminology

Let $\texttt{Form}_X$ be the set of all prepositional formulas over the set of variables X; $\texttt{Val}_X$ be the set of all valuations (boolean interpretation) of variables X. Formulas $\varphi_0, \varphi_1 \in \texttt{Form}_X$ are (semantically) equivalent, written $\varphi_0 \equiv \varphi_1$, if $v(\varphi_0) = v(\varphi_1)$ for all $v \in \texttt{Val}_X$. We say that v *is a model of* φ or that v *satisfies* φ if $v(\varphi) = 1$.

For a formula $\varphi \in \texttt{Form}_X$, let $\#_X\varphi = |\{v \in \texttt{Val}_X \mid v(\varphi) = 1\}|$ be the number of models of φ (valuations satisfying φ). We often omit the index X if it is clear from the context.

The set of all permutations of a set X (bijections $X \to X$) is denoted by $\texttt{Perm}_X$ and $\texttt{id}_X$ is the identity permutation.

3.2 Formal definition

In this section, we formally define Code Breaking Games within the framework of propositional logic, where we represent the secret code as a valuation of propositional variables. The game is represented as a *set of variables*, *initial restriction* (a formula that is guaranteed to be satisfied), and a set of *possible experiments*. A finite set of possible *outcomes* is associated with each experiment. Outcome is a propositional formula that represents the partial information, which the codebreaker can gain from the experiment.

The number of experiments is typically very large (such as 36894 for the Counterfeit-coin Problem **??**) but most of them have same structure and yield similar outcomes. Therefore we opt for a compact representation of an experiment as a pair (type of experiment, parametrization), where parametrization is a string over a defined alphabet. This whole idea is formalized below.

Definition 3.1 (Code Breaking Game). A *Code Breaking Game* is a septuple $\mathcal{G} = (X, \varphi_0, T, \Sigma, E, F, \Phi)$, where

- X is a finite set of propositional variables,
- $\varphi_0 \in \texttt{Form}_X$ is a satisfiable propositional formula,
- T is a finite set of types of experiments,
- Σ is a finite alphabet,
- $E \subseteq T \times \Sigma^\star$ is an *experiment* relation, and
- F is a finite collection of functions of type $\Sigma \to X$,
- $\Phi : T \to 2^{\texttt{PForm}_{X,F,\Sigma}}$ is an *outcome function* such that $\Phi(t)$ is finite for any $t \in T$. Definition of $\texttt{PForm}$ follows (Definition 3.2).

Definition 3.2 (Parametrized formula). A set of *parametrized formulas* $\texttt{PForm}_{X,F,\Sigma}$ is a set of all strings ψ generated by the following grammar:

$$\psi ::= x \mid f(\$n) \mid \psi \circ \psi \mid \neg\psi,$$

where $x \in X$, $f \in F$, $n \in \mathbb{N}$, and $\circ \in \{\wedge, \vee, \Rightarrow\}$. By $\psi(p)$ we denote application of a parametrization $p \in \Sigma^\star$ on a formula ψ, which is defined recursively on the structure of ψ in the following way:

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

We use the special symbol \$ in $f(\$n)$ so that n cannot be mistaken for the argument of f, which is n-th symbol of the parametrization. Note that if $f(\$n)$ appears in ψ and $|p| < n$, then $\psi(p)$ is undefined.

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

The compact representation with parametrized formulas does not restrict the class of games that can fit this definition. If no two experiments can be united under the same type, every experiment can have its own type and allow only one possible parametrization.

Definition 3.3 (Solving process). An *evaluated experiment* is a pair (e, φ) such that $\varphi \in \Phi(e)$. Let us denote the set of evaluated experiments by Ω.

A *solving process* is a finite or infinite sequence of evaluated experiments.

For simplicity, we omit the brackets 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 accrued knowledge after the first k experiments (including the initial restriction φ_0). For finite λ, let $\lambda\langle\rangle = \lambda\langle|\lambda|\rangle$ be the overall accrued knowledge.

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

Let us now describe the course of the game in the defined terms. First, the codemaker choose a valuation v from $\texttt{Val}^*$. Second, the codebreaker chooses a type $t \in T$ and a parametrization $p \in \Sigma^*$ such that $(t,p) \in E$. Third, the codemaker gives the codebreaker a formula $\varphi \in \Phi((t,p))$, which is satisfied by the valuation v. Then the evaluated experiment $((t,p),\varphi)$ is appended to the (initially empty) solving process λ and they continue with the second step. The game continues until $\#\lambda\langle\rangle = 1$, which corresponds to the situation in which the codebreaker can uniquely determine the code.

So that the codemaker can always fulfill the third step, there must be a formula $\varphi \in \Phi(e)$ satisfied by any valuation. Although it might make sense to allow multiple satisfied formulas, we restrict ourselves to games where the outcome is uniquely defined for given valuation.

Definition 3.4 (Well-formed game). A code-breaking game is *well-formed* if for all $e \in E$,

$$\forall v \in \texttt{Val}^* .\ \exists \text{ exactly one } \varphi \in \Phi(e)\ .\ v(\varphi) = 1$$

As the semantics of non-well-formed games is unclear, we focus only on well-formed games and, be default, we suppose a game to be well-formed if not stated otherwise.

Example 3.5 (Fake-coin problem). Fake-coin problem with n coins, one of which is fake, can be formalized as a code breaking game $\mathcal{F}_n = (X, \varphi_0, T, \Sigma, E, F, \Phi)$.

- $X = \{x_1, x_2, \ldots, x_n, y\}$,
  $\varphi_0 = \texttt{Exactly}_1\ \{x_1, \ldots, x_n\}$.
  Intuitively, variable x_i tells weather the coin i is fake. Variable y tells weather it is lighter or heavier. Formula φ_0 says that exactly one coin is fake.
- $T = \{w_2, w_4, \ldots, w_n\}$,
  $\Sigma = \{1, 2, \ldots, n\}$,
  $E = \bigcup_{1 \le m \le n/2} \{(w_{2m}, p) \mid p \in \{1, \ldots, n\}^{2m}, \forall x \in X. \#_x(p) \le 1\}$.
  There are $n/2$ types of experiment – according to the number of coins we put on the weights. The alphabet contains natural numbers up to n and possible parametrizations for w_{2m} are strings of length $2m$ with no repetitions.
- $F = \{f_x\}$, where $f_x(i) = x_i$ for $1 \le i \le n$,
  $\Phi(w_m) =$

$$\big\{((f_x(\$1) \vee \ldots \vee f_x(\$m)) \wedge \neg y) \vee ((f_x(\$m+1) \vee \ldots \vee f_x(\$2m)) \wedge y),$$
$$((f_x(\$1) \vee \ldots \vee f_x(\$m)) \wedge y) \vee ((f_x(\$m+1) \vee \ldots \vee f_x(\$2m)) \wedge \neg y),$$
$$\neg(f_x(\$1) \vee \ldots \vee f_x(\$2m))\big\}.$$

There are 3 possible outcomes of every experiment. First, the right side is heavier. This happens if the fake coin is lighter and it appears in the first half of the parametrization, or if it is heavier and it appears in the second half. Second, analogously, the left side is heavier. Third, the weights are balanced if the fake coin do not participate in the experiment.

Example 3.6 (Fake-coin problem, alternative). For demonstration purposes, here is another formalization of the same problem. $\mathcal{F'}_n = (X, \varphi_0, T, \Sigma, E, F, \Phi)$.

- $X = \{x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n\}$,
  $\varphi_0 = \texttt{Exactly}_1\,\{x_1, \ldots, x_n, y_1, \ldots, y_n\}$.
  Variable x_i tells that the coin i is lighter, variable y_i tells that the coin i is heavier. Formule φ_0 says that exactly one coin is different.
- T, Σ, E is defined as in Example 3.5.
- $F = \{f_x, f_y\}$, where $f_x(i) = x_i$ and $f_y(i) = y_i$ for $1 \le i \le n$,

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

Example 3.7 (Mastermind). Mastermind puzzle with n pegs and m colors can be formalized as a code breaking game $\mathcal{M}_{n,m} = (X, \varphi_0, T, \Sigma, E, F, \Phi)$.

- $X = \{x_{i,j} \mid 1 \le i \le n, 1 \le j \le m\}$,
  $\varphi_0 = \bigwedge\{\texttt{Exactly}_1\,\{x_{i,j} \mid 1 \le j \le m\} \mid 1 \le i \le n\}$.
  Variable $x_{i,j}$ tells whether there is the color j at position i. Formula φ_0 says that there is exactly one color at each position.
- $T = \{g\}$,
  $\Sigma = C$,
  $E = \{(g, p) \mid p \in \Sigma^n\}$.
  There is only one type of experiment, parametrization of which is any sequence of colors of length n.
- $F = \{f_1, \ldots, f_n\}$, where $f_i(c) = x_{i,c}$ for $1 \le i \le n$,
  $\Phi(g) = \{\text{Outcome}(b, w) \mid 0 \le b \le n, 0 \le w \le n, b + w \le n\}$, where Outcome function is computed by the algorithm described below.

As described in the introduction of the Mastermind problem, the outcome corresponds to some maximal matching between the pegs in the code and in the guess.

The idea here is to generate all matchings corresponding to a given outcome, generate a formula that expresses validity of the matching for a given experiment and put them into a big disjunction.

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

- Select $B \subseteq P$ such that $|B| = b$. These are the positions at which the color in the code and in the guess matches and they correspond to the black markers.
- Select $W \subseteq P \times P$ such that $|W| = w$, $p_1(W) \cap B = \emptyset$, and $p_2(W) \cap B = \emptyset$, where p_1, p_2 are the projections. These correspond to the white markers – $(i, j) \in W$ means that the color at position i in the guess is at position j in the code.

Next, for each combination (B, W), we generate a conjunction in the following way:

- For $i \in B$, we add $f_i(\$i)$.
- For $(i, j) \in W$, we add $\neg f_i(\$i) \wedge f_j(\$i)$.
- For $(i, j) \in (P \smallsetminus B \smallsetminus p_1(W)) \times (P \smallsetminus B \smallsetminus p_2(W))$, we add $\neg f_j(\$i)$. This guarantees that the matching is maximal.

The result is a disjunction of all these clauses, which effectively enumerates all the cases. For example, for $n = 4$ the result of Outcome$(1, 1)$ starts with

$(\neg f_0(\$0) \wedge \neg f_1(\$1) \wedge \neg f_1(\$2) \wedge \neg f_2(\$1) \wedge \neg f_2(\$2) \wedge f_3(\$3)) \vee (\neg f_0(\$0) \wedge \neg f_0(\$1) \wedge \neg f_0(\$2) \wedge \neg f_1(\$0) \wedge \neg f_1(\$1) \wedge$
$\neg f_2(\$1) \wedge \neg f_2(\$2) \wedge f_3(\$3)) \vee (\neg f_0(\$0) \wedge \neg f_0(\$1) \wedge \neg f_0(\$2) \wedge \neg f_1(\$1) \wedge \neg f_1(\$2) \wedge \neg f_2(\$0) \wedge \neg f_2(\$2) \wedge f_3(\$3)) \vee$
$(\neg f_0(\$0) \wedge \neg f_0(\$2) \wedge \neg f_1(\$0) \wedge \neg f_1(\$1) \wedge \neg f_2(\$0) \wedge \neg f_2(\$1) \wedge \neg f_2(\$2) \wedge f_3(\$3)) \vee (\neg f_0(\$0) \wedge \neg f_0(\$1) \wedge \neg f_1(\$1) \wedge$
$\neg f_1(\$2) \wedge \neg f_2(\$0) \wedge \neg f_2(\$1) \wedge \neg f_2(\$2) \wedge f_3(\$3)) \vee (\neg f_0(\$0) \wedge \neg f_0(\$1) \wedge \neg f_1(\$0) \wedge \neg f_1(\$1) \wedge \neg f_2(\$2) \wedge f_3(\$3)) \vee$
$(\neg f_0(\$0) \wedge \neg f_0(\$1) \wedge \neg f_1(\$0) \wedge \neg f_1(\$1) \wedge \neg f_1(\$2) \wedge \neg f_2(\$0) \wedge \neg f_2(\$2) \wedge f_3(\$3)) \vee (\neg f_0(\$0) \wedge \neg f_0(\$2) \wedge \neg f_1(\$1) \wedge$
$\neg f_2(\$0) \wedge \neg f_2(\$2) \wedge f_3(\$3)) \vee (\neg f_0(\$0) \wedge \neg f_0(\$2) \wedge \neg f_1(\$0) \wedge \neg f_1(\$1) \wedge \neg f_1(\$2) \wedge \neg f_2(\$1) \wedge \neg f_2(\$2) \wedge f_3(\$3)) \vee \ldots,$

and contains 24 clauses at the top level with 144 literals in total.

Example 3.8 (Mastermind (alternative)). For completeness, we show another way to formalize the Mastermind problem, which does not need algorithmic generation of the formulas. Let $\mathcal{M}'_{n,m} = (X, \varphi_0, T, \Sigma, E, F, \Phi)$.

- X and φ_0 is defined as in Example 3.7.
- $T = \{g_{k_1,\ldots,k_m} \mid k_i \in \{1, \ldots, n\}, \sum_i k_i = n\}$,
  $\Sigma = C$,
  $E = \{(g_{k_1,\ldots,k_m}, p) \mid p \in \Sigma^n, |\{i \mid p[i] = j\}| = k_j\}$.
  The type $g_{k_1,\ldots,k_m}$ covers all the guesses in which the number of j-colored pegs is k_j. Therefore, two guesses for which we use the same pegs (pegs are just shuffled) are of the same type, but if we change a peg for one with different color, it is other type of experiment.
- $F = \{f_1, \ldots, f_n\}$, where $f_i(c) = x_{i,c}$ for $1 \leq i \leq n$,

$$\begin{aligned}\Phi(g_{k_1,\dots,k_n}) = \Big\{ & \\ & \texttt{Exactly}_b \{f_i(\$i) \mid 1 \le i \le n\} \wedge & (1) \\ & \texttt{Exactly}_t \bigcup \big\{\{\texttt{AtLeast}l(x_{1,j},\dots,x_{n,j}) \mid 1 \le l \le k_j\} \mid 1 \le j \le m\big\} & (2) \\ & \qquad\qquad \mid 0 \le b \le t, 0 \le t \le n\Big\}. \end{aligned}$$

Part (1) of the formula captures the number of the black markers. Part (2) captures the total number of markers. Indeed, we get k markers for color j if and only if $k < k_j$ and there are at least k pegs of color j in the code, i.e. all the formulas $\texttt{AtLeast}i(x_{1,j},\dots,x_{n,j})$ are satisfied for $i \le k$. Note that since the number of pegs of each color is fixed by the type and we do not care about the exact positions, this part of the formula is not parametrized.

We do not provide the formal definition 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 4, and definition of all the games in this language can be found in **??**.

3.3 Strategies

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 $v \in \texttt{Val}^*$ induce an infinite solving process

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

where $e_{i+1} = \sigma(e_1, \varphi_1, \dots, e_i, \varphi_i)$ and $\varphi_{i+1} \in \Phi(e_{i+1})$ is such that $v(\varphi_{i+1}) = 1$, for all $i \in \mathbb{N}$. Note that thanks to the well-formed property, there is always exactly one such φ_{i+1}.

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.

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

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 models v of φ_0, i.e.

$$\Lambda^\sigma = \max_{v \in \texttt{Val}^*} |\sigma|_v.$$

We say that a strategy σ *solves the game* if Λ^σ is finite. The game is *soluble* if there exists a strategy that solves the game.
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.

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

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'.

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, $\lambda^\sigma_v\langle k \rangle$ can take up to b^k different values. By pidgeon-hole principle, if $\#\varphi_0 > b^k$, there must be a valuation v such that $\#\lambda^\sigma_v\langle k \rangle > 1$. 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 $v_1, v_2 \in \mathtt{Val}^*$. If v_1 is a model of $\lambda^\sigma_{v_2}\langle k \rangle$, then $\lambda^\sigma_{v_1}[1..k] = \lambda^\sigma_{v_2}[1..k]$.*

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. TODO: Příklad jednoduché hry, strategie, odhadu pomocí lematu, optimální strategie.

Non-adaptive strategies

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 seen as functions $\tau : \mathbb{N}_0 \to E$. Then $\sigma(\lambda) = \tau(|\lambda|)$.

Non-adaptive strategies corresponds to the well studied problems of static mastermind and non-adaptive strategies for the coutnerfeit coin problem **??** **??**. We mention them here just to show the possibility of formulating these problems in our framework but we do not study them any further.

Memory-less strategies

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 as functions $\tau : \mathtt{Form}^* \to E$ such that $\varphi_1 \equiv \varphi_2 \Rightarrow \tau(\varphi_1) = \tau(\varphi_2)$. Then $\sigma(\lambda) = \tau(\lambda\langle\rangle)$.

Lemma 3.16. *Let σ be a memory-less strategy and $v \in \mathtt{Val}^*$. 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. For the sake of simplicity, 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 $l \in \mathbb{N}$. ■

Lemma 3.17. *Let σ be a strategy. Then there exists a memory-less strategy τ such that $|\sigma|_v \geq |\tau|_v$ for all $v \in \mathtt{Val}^*$.*

Proof. Let us choose any total order $\varphi_1, \varphi_2, \dots$ of $\mathtt{Form}^*$ such that if φ_i implies φ_j, then $i \leq j$. We build a sequence of strategies $\sigma_0, \sigma_1, \sigma_2, \dots$ inductively in the following way. Let $\sigma_0 = \sigma$.

- If there is no $v \in \mathtt{Val}^*, k \in \mathbb{N}_0$ 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 $v \in \mathtt{Val}^*, k \in \mathbb{N}_0$ 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. ■

Definition 3.18 (Greedy strategy). Let $f : \mathtt{Form}_X \to \mathbb{Z}$. A memory-less strategy σ is *f-greedy* if for every $\varphi \in \mathtt{Form}_X$ and $e' \in E$,

$$\max_{\substack{\psi\in\Phi(\sigma(\varphi))\\ SAT(\varphi\wedge\psi)}} f(\varphi\wedge\psi) \leq \max_{\substack{\psi\in\Phi(e)\\ SAT(\varphi\wedge\psi)}} f(\varphi\wedge\psi).$$

In words, a greedy strategy minimizes the value of f on the formula in the next step. We say σ is *greedy* if it is $\#_X$-greedy.

Lemma 3.19. *Let $b = \max_{t\in T}|\Phi(t)|$ be the maximal number of possible outcomes of an experiment. If for any $\varphi \in \mathtt{Form}^*$,*

$$\exists e. \max_{\psi\in\Phi(e)} \#(\varphi\wedge\psi) = \left\lceil \frac{\#\varphi}{b} \right\rceil,$$

then a greedy strategy σ is optimal and

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

Proof. TODO: Napsat důkaz.

Example 3.20. Greedy strategies are optimal in the fake-coin game $\mathcal{F}_n$. TODO: Napsat důkaz.

3.4 Symmetries in Code Braking Games

The number of possible parametrizations of a type of experiment is typically very large, which makes the analysis a game much harder. For example, consider the conterfeit-coin problem (**??**) and the experiment of weighting 4 coins against 4 coins. There is $\frac{1}{2} \cdot \binom{12}{4} \cdot \binom{8}{4} = 17325$ possible combinations for parametrization, but in the initial state (i.e. with no knowledge except for the initial restriction), all of them are equivelent – they will give us the same information regardless of symmetries.

In this section, we formally define equivalence of two experiments and show that we can neglect all but one experiment in each equivelence class. Further, we present several lemmas that will form the basis of our symmetry breaking algorithms in the Chapter **??**.

Definition 3.21 (Symmetric experiment). For an experiment $e = (t, p)$ and a permutation $\pi \in \texttt{Perm}_X$, a π-symmetric experiment $e^\pi = (t, p') \in E$ is an experiment of the same type such that $\{\varphi^\pi \in \Phi(e)\} = \{\varphi \in \Phi(e^\pi)\}$. Clearly, no such experiment may exists.

Definition 3.22 (Symmetry group). We define a *symmetry group* Π as the maximal subset of $\texttt{Perm}_X$ such that for every $\pi \in \Pi$ and for every experiment $e \in E$, there exists a π-symmetric experiment e^π.

Definition 3.23 (Consistent strategy). A memory-less strategy σ is *consistent* if and only if for every $\varphi \in \texttt{Form}_X$ and every $\pi \in \Pi$, there exists $\rho \in \Pi$ such that $\varphi^\pi \equiv \varphi^\rho$ and $\sigma(\varphi^\rho) = \sigma(\varphi)^\rho$.

TODO: Example, na kterém bude vidět, že jednoduchá definice nevyhovuje, protože můžu vzít symetrie φ a dostanu, že to má dávat různé věci.

Lemma 3.24. *Let σ be a memory-less strategy. There exists a consistent memory-less strategy τ such that $|\sigma|_v \geq |\tau|_v$ for all $v \in \texttt{Val}_X$ satisfying φ_0.*

Proof. Similar to the proof of Lemma **??**. Let us choose any total order $\varphi_1, \varphi_2, \ldots$ of $\texttt{Form}_X$ such that if φ_i implies φ_j, then $i \leq j$. We build a sequence of strategies $\sigma_0, \sigma_1, \sigma_2, \ldots$ in the following way: Let $\sigma_0 = \sigma$ and for $i > 0$,

$$\sigma_i(\varphi) = \begin{cases} \sigma_{i-1}(\varphi) & \text{if } \nexists\, \pi \in \Pi . \varphi^\pi \equiv \varphi_i \\ \lambda_{v_i}^{\sigma_{i-1}}(k_i + 1) & \text{if } \exists \pi \in \Pi . \varphi^\pi \equiv \varphi_i, \text{ where } (v_i, k_i) = \arg\min_K |\sigma_{i-1}|_v - k, \\ & \text{and } K = \{(v, k) \in \texttt{Val} \times \mathbb{N}_0 \mid \exists \pi . \lambda_v^{\sigma_{i-1}} \langle k \rangle^\pi \equiv \varphi_i\}. \end{cases}$$

We prove that $|\sigma_i|_v \leq |\sigma_{i-1}|_v$. If there is no k and π such that $\lambda_v^{\sigma_i}\langle k\rangle^\pi \equiv \varphi_i$ then the processes $\lambda_v^{\sigma_i}$ and $\lambda_v^{\sigma_{i-1}}$ are the same. If the is such k and TODO: ....

The last strategy of the sequence is consistent and satisfies the condition in the lemma. ■

Definition 3.25 (Experiment equivalence). 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 $\{\varphi \wedge \psi \mid \psi \in \Phi(e_1)\} \equiv \{(\varphi \wedge \psi)^\pi \mid \psi \in \Phi(e_2)\}$.

Theorem 3.26. *Let σ, τ be two consistent memory-less strategies, such that $\sigma(\varphi) \cong_\varphi \tau(\varphi)$ for any $\varphi \in \mathtt{Form}_X$. There is a bijection $f : \mathtt{Val}_X \to \mathtt{Val}_X$ such that $|\sigma|_v = |\tau|_{f(v)}$.*

Proof. First, we prove by induction for any $k \in \mathbb{N}_0$, there is a permutation $\pi \in \Pi$ such that $(\lambda_v^\sigma\langle i\rangle)^\pi = \lambda_{v^\pi}^\tau\langle i\rangle$ for all $i \in \mathbb{N}_0$, $i \leq k$. For better readability, let $\alpha_k = \lambda_v^\sigma\langle k\rangle$ and $\beta_{k,\pi} = \lambda_{v^\pi}^\tau\langle k\rangle$

For $k = 0$, take $\pi = \mathtt{id}_X$. Clearly, $\lambda_v^\sigma\langle 0\rangle = \varphi_0 = \lambda_{v^{\mathtt{id}}}^\tau\langle 0\rangle$.

For the induction step, suppose we have $\pi \in \Pi$ such that $\alpha_i^\pi = \beta_{i,\pi}$ for $i \leq k$. Further, suppose π is such that $\sigma(\alpha_k^\pi) = \sigma(\alpha_k)^\pi$. TODO: Víc zdůvodnit.

Let $e_1 = \sigma(\alpha_k)$, $e_2 = \tau(\beta_{k,\pi})$ be the $(k+1)$-th experiments of the strategies. It holds

$$e_2 = \tau(\beta_{k,\pi}) \cong_{\beta_{k,\pi}} \sigma(\beta_{k,\pi}) \overset{IH}{=} \sigma(\alpha_k^\pi) = \sigma(\alpha_k)^\pi = e_1^\pi$$

and, therefore, there exists $\rho \in \Pi$ such that

$$\begin{aligned} \{\beta_{k,\pi} \wedge \psi \mid \psi \in \Phi(e_2)\} &= \{(\beta_{k,\pi} \wedge \psi)^\rho \mid \psi \in \Phi(e_1^\pi)\} = \\ &= \{(\alpha_k^\pi \wedge \psi^\pi)^\rho \mid \psi \in \Phi(e_1)\} = \{(\alpha_k \wedge \psi)^{\rho\pi} \mid \psi \in \Phi(e_1)\} \end{aligned} \tag{*}$$

As $\rho \in \Pi$ and Π is a permutation group, $\rho\pi \in \Pi$.

Since the game is well-formed, v satisfies exactly one formula in $\{\alpha_k \wedge \psi | \psi \in \Phi(e_1)\}$. Therefore $v^{\rho\pi}$ satisfies exactly one formula in $\{(\alpha_k \wedge \psi)^{\rho\pi} | \psi \in \Phi(e_1)\} = \{\beta_{k,\pi} \wedge \psi | \psi \in \Phi(e_2)\}$, which means that $v^{\rho\pi}$ satisfies $\beta_{k,\pi}$. TODO: Tohle nefugujeee! From Lemma **??**, $\beta_{k,\pi} = \beta_{k,\rho\pi}$. Both $\alpha_{k+1}^{\rho\pi}$ and $\beta_{k+1,\rho\pi}$ is thus the only formula from (*) satisfied by $v^{\rho\pi}$ and, therefore, $\alpha_{k+1}^{\rho\pi} = \beta_{k+1,\rho\pi}$.

Now for a fixed v, take $k = |\sigma|_v$, take $\pi \in \Pi$ such that $(\lambda_v^\sigma\langle k\rangle)^\pi = \lambda_{v^\pi}^\tau\langle k\rangle$ and define $f(v) = v^\pi$. Since $(\lambda_v^\sigma\langle i\rangle)^\pi = \lambda_{f(v)}^\tau\langle i\rangle$ for $i \leq k$ and variable permutation preserves the number of models of a formula, i.e. $\#\varphi = \#\varphi^\pi$ for any $\varphi \in \mathtt{Form}_X$, $\pi \in \mathtt{Perm}_X$, we have $|\sigma|_v = |\tau|_{f(v)}$. ■

Corollary 3.27. *Let σ_1, σ_2 be two consistent memory-less strategies, such that $\sigma_1(\varphi) \cong_\varphi \sigma_2(\varphi)$ for any $\varphi \in \texttt{Form}_X$. Then $\Lambda^{\sigma_1} = \Lambda^{\sigma_2}$ and $\Lambda^{\sigma_1}_{exp} = \Lambda^{\sigma_2}_{exp}$.*

For any accrued knowledge φ, this lemma gives us the right to consider only one of the experiments e_1, e_2 if $e_1 \sim_\varphi e_2$.
Now, we would love an algorithm that would, for a given formula φ, generate a set of experiment, such that there is exaclty one expriment from every equivalence class in $E/\sim_\varphi$.

3.5 Symmetry Breaking

Phase 1 - Interchangeble symbols

Phase 2 - Canonical Form of parametrization

Phase 3 - Canonical Form of formula graph

Comparison

4 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. TODO: Jak je to s licenci Picosatu a tak? Zminit tady.

4.1 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>.

The input file is parsed by lines, each of which must have one of the forms listed in the following table.

[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.
RESTRICTION <formula>	Defines the initial restriction 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.
OUTCOME <string> <formula>	Defines an outcome of the experiment named by the first argument.

We specify what "formula" is by the following grammar:

$$\begin{array}{rl} \text{<formula>} ::= & \text{<ident}_1\text{>} \quad \mid \quad (\ \text{<formula>}\ ) \quad \mid \quad !\ \text{<formula>} \\ \mid & \text{<formula>} \circ \text{<formula>} \quad \mid \quad \texttt{X}\text{-<int}_1\text{>}\ (\ \text{<formula-list>}\ ), \\ \mid & \text{<ident}_2\text{>}\ (\text{\$}\ \text{<int}_2\text{>}\ ), \end{array}$$

where <ident$_1$> is an identifier of a variable and $\circ \in$ {and, &, or, |, ->, <-, <->} is a standard logical operator with its usual meaning.

`X` is one of `AtLeast`, `AtMost`, `Exactly` and we call it a *numerical operator*. Let $\varphi := \texttt{X-}k(\varphi_1, \dots, \varphi_n)$ be a formula, v a valuation of the variables and let s be the number of satisfied formulas among $\varphi_1, \dots \varphi_n$ by valuation v. Then the formula φ is satisfied by v if and only if $s \geq k$, $s \leq k$ and $s = k$, for X being `AtLeast`, `AtMost` and `Exactly`, respectively. 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 naïve expansion 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 4.5.

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 4.1. TODO: Running example.

```
VARIABLES y, x1, x2, x3, x4
RESTRICTION 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 exapmle, 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 3.7. 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 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, Restriction, 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.

[3] http://flex.sourceforge.net/

[4] http://www.gnu.org/software/bison/

[5] https://www.python.org

Function	Type of x	Type of y
Variable(x)	string	-
Variables(x)	list of strings	-
Restriction(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	-
Outcome(x, y)	string	formula (as a string)

Example 4.2. An example specification of the counterfeit coin problem, based on Example 3.5, follows.

```
N = 12
x_vars = ["x" + str(i) for i in range(N)]
VARIABLES(["y"] + x_vars)
RESTRICTION("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))
```

4.2 Compilation and basic usage

COBRA is written in C++ and uses some features of the modern C++11 standard so you need a modern C++ compiler to build the tool. We have tested and can recommend `gcc` version 4.8 or higher, or `clang` version 3.2 or higher. To compile the tool, run `make` in the program folder. It 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.

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 4.3, 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. In most cases, you should be fine with the default backend and can ignore this option. Details can be found in Section 4.5.

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. Thus, if you want to run COBRA on a low-level input format, you can run `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 4.3: An example of the time overview after a time demanding task.

4.3 Modes of operation

Overview mode [o, overview] (default)

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

Overview mode serves as a basic check that your input file is syntactically correct and that the game you specified 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 3.4.

Verification that an experiment with outcomes $\psi_1, \ldots \psi_k$ is well-formed can be done by verifying that $\varphi_0 \rightarrow \texttt{Exactly}_1\,(\psi_1, \ldots, \psi_k)$ is a tautology. We negate the formula, pass it to a SAT solver, ask for satisfiability and expect a negative result.

Verification that the game is well-formed is done by generating possible non-equivalent experiments for the first round and verifying that all of them are

```
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 4.4: An example of a failed well-formed check in the counterfeit coin problem with missing "=" outcome.

well-formed. This is enough thanks to ?? TODO: Lemma: pokud jsou experimenty ekviv a jeden je well-formed, je i druhy well-formed. If a problem is found, the tool outputs an assignment and an experiment for which no outcome, or more that one outcome, is satisfied.

Simulation mode [s, simulation]

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

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

We do not consider the codemaker a player who just chooses the secret code and evaluates experiment but a player who chooses the outcomes of experiments as they come according to his will. The only condition is that the outcomes are consistent.

Implemented strategies for both codebreaker and codemaker are described in the next section and mostly correspond to strategies that can be found in the literature on Mastermind. 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>
```

This makes the tool compute worst-case number of experiments and average-case number of experiments for a given codebreaker's strategy.

TODO: pseudocode?

Optimal strategy mode [o, optimal]

TODO: ...

4.4 Strategies

Codebreaker's strategy

We support the following one-step look-ahead strategies for the codebreaker. Let N denote the number of possible codes in the current state, and for a fixed experiment, let n_i denote the number of possible codes for which the experiment results in outcome i.

Min-num. `-e minnum`
: Minimizes the worst-case number of remaining codes in the next step, i.e. selects experiment with minimal $\max_i n_i$. For Mastermind, this was suggested by Knuth[17].

Min-exp. `-e minexp`
: Minimizes the expected number of remaining codes in the next step, i.e. minimize $\sum_i n_i^2/N$. For Mastermind, this was suggested by Irwing[18].

Entropy. `-e entropy`
: Maximizes the entropy of numbers of remaining codes, i.e. maximizes $-log(k_i) \cdot k_i$, where $k_i = n_i/N$. For Mastermind, this was suggested by Neuwirth[19].

Most-parts. `-e parts`
: Maximizes the number of possible outcomes, i.e. maximize $\#i.(n_i > 0)$. For Mastermind, this was suggested by Kooi[20].

Fixed. `-e fixed`
: Maximize the number of fixed variables in the next step. This is specific to propositional logic representation of a code-breaking game. Further, it

depends on the exact formalization. For example, though Examples 3.5 and 3.6 both describe the same problem, the choice of the experiment by the *fixed* strategy is different in the very first step.

Codemaker's strategy

The codemaker has the following strategy options. Note that none of them is guaranteed to maximize the number of experiments of any codebreaker's strategy.

Max-num. `-o maxnum`
Select an outcome with largest n_i. This corresponds to the *min-num* strategy for codebreaker.

Fixed. `-o fixed`
Minimizes the number of fixed variables in the next step.

Extensibility

If you want to analyze a new heuristics to selects experiments, all you need to do is to implement a new function in `strategy.h/strategy.cpp` file and add a corresponding entry to the ... table in `strategy.h`. A strategy function takes a list of sensible experiments as an argument and it returns the index of the selected one.

If your strategy only *maximizes* or *minimizes* some metrics on the experiments, 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 *Min-exp* 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);
}
```

4.5 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 4.3 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 backend. 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 4.5.
- 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] [34]. 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 4.6.

  Third option is to use a SAT solver and a simple backtracking approach,

[6] https://sites.google.com/site/marcthurley/sharpsat

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 4.7.

Algorithm 4.5: Decision whether a formula has exactly one model.

```
if not SATISFIABLE() then return false
$v \leftarrow$ GETASSIGNMENT()
OPENCONTEXT()
ADDCONSTRAINT($\overline{x_1} \mid \dots \mid \overline{x_n}$), where $\overline{x_i}$ is $\neg x_i$ if $v(x_i) = 1$ and $x_i$ otherwise
$sat \leftarrow$SATISFIABLE()
CLOSECONTEXT()
return $\neg sat$
```

Algorithm 4.6: Model counting, second option.

```
$models \leftarrow 0$
OPENCONTEXT()
while SATISFIABLE() do
    $v \leftarrow$ GETASSIGNMENT()
    ADDCONSTRAINT($\overline{x_1} \mid \dots \mid \overline{x_n}$), where $\overline{x_i}$ is $\neg x_i$ if $v(x_i) = 1$ and $x_i$ otherwise
    $models \leftarrow models + 1$
end
CLOSECONTEXT()
return $models$
```

Algorithm 4.7: Model counting, third option.

```
$models \leftarrow 0$
$x \leftarrow$ any variable from $vars$
OPENCONTEXT()
ADDCONSTRAINT($x$)
if SATISFIABLE() then $models \leftarrow models +$ COUNT($vars \setminus \{x\}$)
CLOSECONTEXT()
OPENCONTEXT()
ADDCONSTRAINT($\neg x$)
if SATISFIABLE() then $models \leftarrow models +$ COUNT($vars \setminus \{x\}$)
CLOSECONTEXT()
return $models$
```

COBRA includes three solver implementations which we describe next.

PicoSat

Picosat[7] [35] is a simple, extensible SAT solver, which supports incremental SAT solving exactly in the way we need.

Bindings to Picosat are implemented in `PicoSolver` class. This class also implements model counting algorithm 4.7 as Picosat does not support model counting itself.

MiniSat

Minisat[8] [36] 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. TODO: Model counting?.

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 restriction 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.

[7] http://fmv.jku.at/picosat/

[8] http://minisat.se/

Extensibility

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.

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[37].

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 $\texttt{Exactly}_k(f_1, \ldots, f_n)$ 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 $l \in \{0, 1, \dots, k\}$ and $m \in \{1, \dots, n\}$, $l \le m$, we create a new variable $z_{l,m}$ which will be true if and only if the formula $\texttt{Exactly}_l\,(f_1, \dots, f_m)$ 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 naïve solution to express the formula as a conjunction of the $\binom{n}{k}$ possibilities, which would be double exponential.

4.6 Symmetry breaking

TODO: Bliss, vs saucy vs nauty

4.7 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 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*[38] 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

As noted earlier, compilation of COBRA requires a compiler, which supports all the C++11 features we use. We recommend using standard `gcc` version 4.8 or higher, or `clang` version 3.2 or higher.

The tool is platform independent. We tested compilation and functionality on all three major operating systems, on Linux (Ubuntu 12.04), Mac OS X (10.9) and Windows (8.1).

Unit testing

Unit testing has became a common part of software development process in the recent years. Correctness is automatically a top priority for a tool of this kind and unit tests are a perfect way to capture potential programmer's error as soon as possible and avoid regression.

Lots of unit tests framework for C++ are available. We focused on simplicity, minimal amount of work needed to add new tests and good assertion support, and opted for *Google Test*[9].

All available tests are compiled and executed if you run `make test` in the tool directory. This should serve as a basic sanity test and we highly recommend doing this in case you make any changes in the code.

[9] https://code.google.com/p/googletest/

5 Experimental Results

6 Conclusions

Bibliography

Counterfeit Coin

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

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

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

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

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

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

[7] 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.

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

[9] 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.

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

[12] 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.

[13] 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.

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

Mastermind

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

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

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

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

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

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

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

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

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

[24] 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.

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

[28] Michael T Goodrich. “On the algorithmic complexity of the Mastermind game with black-peg results”. In: *arXiv preprint arXiv:0904.4911* (2009).

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

[30] 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.

Other

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

[25] 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).

[27] 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).

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

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

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

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

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

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

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