Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Complexity of solving puzzles

master's thesis

2018

Bc. Barbora Klembarová

Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Complexity of solving puzzles

master's thesis

Study programme:	Informatics
Study field:	Informatics
Department:	Department of Computer Science
Supervisor:	RNDr. Michal Forišek, PhD.

Bratislava, 2018

Bc. Barbora Klembarová

65872738

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Barbora Klembarová
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Complexity of solving puzzles
Zložitosť riešenia logických úloh

Cieľ: Cieľom práce je zvoliť si malú sadu logických úloh (ako napr. maľované krížovky, Sudoku, alebo Sokoban) a analyzovať rôzne aspekty zložitosti ich riešenia.

Primárna definícia zložitosti je samozrejme výpočtová zložitosť rozhodovacieho problému, pri ktorom chceme o danej inštancii úlohy povedať, či je riešiteľná.
Pre väčšinu logických úloh je tento problém NP-ťažký, ak nie ťažší.
Na druhej strane nás však môže zaujímať aj zložitosť riešenia konkrétnej inštancie človekom.

V tejto práci by autorka mala preskúmať tieto druhy zložitosti pre niekoľko vhodne zvolených logických úloh. Pri vyhodnocovaní zložitosti pre ľudí by mali byť použité reálne dáta. V závere práce by bolo vhodné uviesť diskusiu o vzťahu medzi týmito druhmi zložitosti.

Vedúci: RNDr. Michal Forišek, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 29.02.2016

Dátum schválenia: 13.04.2016 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Acknowledgement

Most of all, I would like to thank my supervisor Mišof who struggled with me through all these years, for all his help, for answering my endless questions and for psychological support.

I would also like to thank Faculty of Mathematics, Physics and Informatics for many great courses which covered every aspect needed to successfully finish this thesis.

Next, I would like to thank my family for their support and understanding; all my friends for their encouragement, especially Olívia for reading most of the pages, Tommy for being my unofficial consultant, Denis for neverending kitty joke and Trojsten organization for keeping me afloat.

Special thanks goes to Alastor and guys from Kickresume, for giving me opportunity to keep up my work ethic and avoid spending my time procrastinating.

Last, but not least, I would like to thank my boyfriend Vladko, for being my consultant, reviewer and emotional support through all this time.

Abstrakt

Ako súvisí zložitosť riešenia človekom a výpočtová zložitosť? V tejto práci sa zaoberáme dvoma hrami *Gulička* a *Sokoban*. Z výpočtového hľadiska patrí Gulička medzi problémy v triede P a Sokoban do triedy *PSPACE*-úplnych problémov. Pre každú hru navrhneme niekoľko rôznych syntaktických atribútov a budeme zisťovať, ktoré sú zodpovedné za obtiažnosť levelu pre ľudského hráča. Na konci práce rozoberáme ako tieto rôzne aspekty obtiažnosti navzájom súvisia, s ohľadom na špecifické atribúty nájdené počas analýzy riešenia problému človekom.

Kľúčové slová: puzzle, hra, riešenie človekom, výpočtová zložitosť

Abstract

How is the complexity of solving puzzles by a human connected to computational complexity? In this thesis we study two games *Tilt maze* and *Sokoban*. From a computational point of view Tilt maze belongs to P and Sokoban belongs to *PSPACE*-complete problems. We design different syntactic attributes of these puzzles and recognize those influencing difficulty in human problem solving. We conclude this thesis by discussing how are these different aspects of problem complexity related, regarding specific attributes identified during human problem solving analysis.

Keywords: puzzle, human problem solving, computational complexity

Contents

List of Figures

List of Tables

Introduction

Human problem solving has been studied for a long time. One of the first seminal works is written by Simon and Newell [29]. Relevant to this work are also Water jug puzzle [11], Tower of Hanoi puzzle [22], river crossing problems [17], Chinese ring puzzle [23], and Fifteen puzzle [32], all of these can be represented as state space traversal problems.

In our analysis we focus on the issue of puzzle difficulty, considering human problem solving but also computational complexity. First, we introduce a formalization of a puzzle, which is used to define games Tilt maze and Sokoban as decision problems. Computational complexity, for both of these problems, was already analyzed [12, 36]. But what makes a problem difficult for humans? We try to answer this question for these puzzles.

There are, of course, several factors that influence problem difficulty – context of problem solving, difficulty of individual steps in the solution, the overall structure of the problem state space. We focus mainly on the last one.

The goal of our thesis is to design different syntactic attributes of level instances and identify those responsible for the problem difficulty. For evaluation we use real data from two different websites, both collecting solution time for logged users. We extract attributes and train difficulty function on user data using machine learning models. We predict the average logarithm of solution time and inspect feature coefficients of trained models. These coefficients corresponds to the impact they have on these models.

At the end of the thesis we compare different attributes responsible for the complexity of solving puzzles for humans versus computers.

Chapter 1

Background

In the first part of this chapter we try to answer the question *What is a puzzle?*. Subsequently, we discuss problem difficulty, involving an introduction to the computational complexity. The last part includes information about our data sources.

1.1 Motivation

> If a game is in P, it becomes no fun once you learn "the trick" to perfect play, but hardness results imply that there is no such trick to learn: the game is inexhaustible.
>
> *David Eppstein*

Puzzles, although hard, are popular amongst people. Even though large part of possible instances would be unsolvable by human in reasonable time, instances can be chosen, so that humans are able to solve them. Moreover, people produce consistent results, corresponding to ones skill in the game – various people may perform very differently in solving the problems of this type.

In reality, there is a lot of bias hidden in many aspects, from players to the instances. For example, instances of the game Nonograms – only instances which result to pictures are chosen. Furthermore, the final drawing has objects, that are nicely structured and consist of large filled areas. Since this game is *NP*-complete, SAT can be reduced to it. This reduction yields images, which are clearly different from the images chosen as a solution for magazine Nonograms.

Problem solving is very different for humans and computers – they have complemen-

tary strengths [37]. For successful creation of tools for human-computer interaction, we need to understand what makes problem difficult for humans. Among another applications belongs intelligent tutoring systems [9], which serve for teaching and training. If confronted with adequate difficulty, people enjoy solving problems – if the problems are too hard it's deterring, if too easy it's boring.

In practice, we can observe that even though solving these puzzles in general is undoubtedly hard, on real data we can predict the time of the solution which would human solver need.

How humans solved problems is a complex question which consists of many different parts. We cannot answer fully how people solve the problem, so we will try to answer parts of this problematics.

1.2 Puzzle

The cornerstone of this thesis is a puzzle or a game. We will consider only games played by one player.

We propose the definition of a puzzle[1]:

Definition 1.1 Puzzle *is a game that satisfies the following conditions:*

(1) There is one player.

(2) For every instance, there is a set, usually finite, of possible positions of the game.

(3) The rules of the game specify for each position legal moves to other positions.

(4) The game ends, when a winning position is reached, declared by rules.

If the game never ends, it is declared to be lost.

(5) It is a game of perfect information with no random moves such as the rolling of dice.

We can consider puzzle to be a computational problem. In this way, puzzle can be viewed as an infinite collection of instances together with an answer, whether solution exists, for given instance. The input is referred to as a problem instance, and should

[1]Inspired by definition of the combinatorial game of two players in Game theory [14]

not be confused with the problem itself. For example, consider the 15-puzzle[2] in general version: $(n^2 - 1)$-puzzle. The instance is a grid 4×4 (e.g. 1.1) filled with numbers from one to fifteen (excluding one cell) and solvability answer "yes" or "no". The case below is solvable, hence the answer is "yes".

3	2	1	4
5	-	11	8
9	14	10	12
13	7	6	15

Table 1.1: 15-puzzle instance

In addition, instances of one puzzle share common "look" and rules. We propose a formal definition of a puzzle, where an instance is a string over a finite alphabet.

Definition 1.2 Puzzle P *is a 5-tuple* $(\Sigma, S, val, win, init)$*, where:*

- Σ *is a finite alphabet,*
- *position is a string* Σ^+*,*
- $val : \Sigma^+ \to \{true, false\}$ *is a recursive predicate determining, whether given position is valid,*
- *let* V *be auxiliary set of all valid positions* $V = \{p \mid p \in \Sigma^+ \wedge val(p)\}$*,*
- $S : V \to 2^V$ *is a recursive successor function determining set of valid following positions,*
- $win : V \to \{true, false\}$ *is a recursive predicate determining, whether given position is winning and*
- $init : V \to \{true, false\}$ *is a recursive predicate determining, whether given position is a valid starting position.*

Definition 1.3 Puzzle instance *of a game* $P = (\Sigma, S, val, win, init)$ *is a position* $I \in \Sigma^+$*, where* $init(I) = true$*.*

Based on the previous definitions, we introduce a definition of *state graph* and a *solution*, both of them needed in further parts.

[2]Rules can be found on Wikipedia [1]

Definition 1.4 *For instance I of puzzle $P = (\Sigma, S, val, win, init)$, the* state graph *is a directed graph $G = (V, E)$, where:*

$$V = \{v_p \mid \exists\ p_0, \ldots p_k : p_0 = I \ \wedge\ p_k = p \ \wedge\ \forall i < k : p_{i+1} \in S(p_i)\}$$

$$E = \{e \mid \exists\ p, q : v_p, v_q \in V \ \wedge\ e = (v_p, v_q) \ \wedge\ q \in S(p)\}$$

Definition 1.5 *A* solution *of an instance I of puzzle $P = (\Sigma, S, val, win, init)$ is a path in the state graph G starting in the vertex v_I and ending in some vertex v_p, such that* win(p) = true.

Example of usage is presented in the next section.

1.2.1 Sokoban

Sokoban is one of the well-known puzzles created by Hiroyuki Imabayashi.

Rules: The game board is divided into square cells, forming two-dimensional grid. Each cell is either a wall or a floor. There are k boxes and one man, occupying exactly $k + 1$ different floor cells. Moreover, exactly k floor cells are labeled as targets for the boxes. Man can move horizontally or vertically onto empty squares. He can also push one box before him, moving it to another empty cell in that direction. The game ends, when each box stands on one target cell.

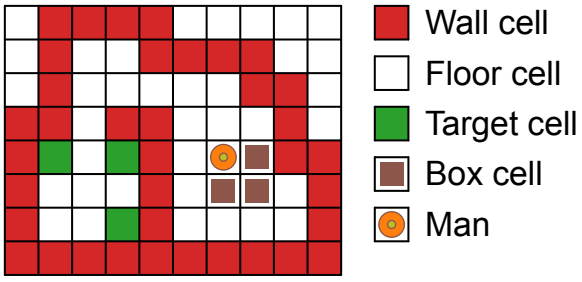


Figure 1.1: Sokoban instance

On the picture 1.1 we can see one instance of Sokoban.

Formally, Sokoban can be defined using definition 1.2. Let's first define the alphabet:

$$\Sigma = \{\#, \sqcup, \&, \sim, +, *, \times, \$\}$$

- # – a wall cell; $\sqcup$ – a floor cell

- & – the man; $\sim$ – the man on a target cell
- $+$ – the box; $*$ – the box on a target cell
- $\times$ – a target cell; $*$ – a target cell containing a box; $\sim$ – a target cell containing the man
- \$ – new line

For the instance 1.1, the initial position looks like:

␣ |#| |#| |#| |#| ␣ ␣ ␣ ␣ ␣ \$
␣ |#| ␣ ␣ |#| |#| |#| |#| ␣ ␣ \$
␣ |#| ␣ ␣ ␣ ␣ ␣ |#| |#| ␣ \$
|#| |#| ␣ |#| |#| ␣ ␣ ␣ |#| ␣ \$
|#| × ␣ × |#| ␣ & + |#| |#| \$
|#| ␣ ␣ ␣ |#| ␣ + + ␣ |#| \$
|#| ␣ ␣ × |#| ␣ ␣ ␣ ␣ |#| \$
|#| |#| |#| |#| |#| |#| |#| |#| |#| \$

The position is written in separate lines for better readability.
For the other parts of Sokoban definition:

- $win(p) = (\#_+(p) = 0)$, if there is no box not on a target cell
- $init(p) = (\#_+(p) > 0)$, if there is at least one box not on a target cell

The definition of *val* predicate:

$$\begin{aligned} val(p) = (r = \#_\$(p)) \ \wedge \ (p = p_0\$p_1\$ \ldots p_{r-1}\$) \wedge \\ c = |p_0| \ \wedge \ \forall i : p_i \in \Sigma^c \wedge \\ \#_\&(p) + \#_\sim(p) = 1 \wedge \\ \#_+(p) + \#_*(p) = \#_\times(p) + \#_*(p) + \#_\sim(p) \wedge \\ \#_+(p) + \#_*(p) \geq 1 \wedge \\ p_0 = p_{r-1} = |\#|^c \ \wedge \\ \forall i : p_{i,0} = p_{i,c-1} = |\#| \end{aligned}$$

The definition is strict, because it forces walls all around the perimeter. However, all real Sokoban levels have areas of floor cells enclosed inside wall perimeter, which

isn't necessarily in form of a rectangle. Nevertheless, these levels can be modified by changing all empty cells outside the perimeter to walls (image 1.2), hence satisfying the *val* predicate.

(a) Original (b) Walls added

Figure 1.2: Sokoban instance

Last part of the definition missing is the recursive successor function S, which we define with help of four auxiliary functions for different directions:

$$S(p) = \{move\_up(p), move\_right(p), move\_down(p), move\_left(p)\}$$

We show only the definition of *move_down* and leave the other three for the reader. First, let's define three helper functions:

$$man\_left(cell) = \begin{cases} \text{␣} & \text{if } cell = \& \\ \times & \text{if } cell = \sim \end{cases}$$

$$man\_came(cell) = \begin{cases} \& & \text{if } cell \in \{\text{␣}, +\} \\ \sim & \text{if } cell \in \{\times, *\} \end{cases}$$

$$box\_came(cell) = \begin{cases} + & \text{if } cell = \text{␣} \\ * & \text{if } cell = \times \end{cases}$$

Using these functions, we define the *move_down* function:

$$move\_down(p) = \begin{cases} p & \text{if } p_{mr+1,mc} = \text{▨} \\ p & \text{if } p_{mr+1,mc} \in \{+, *\} \ \wedge \ p_{mr+2,mc} \in \{+, *, \text{▨}\} \\ p' & \text{if } p_{mr+1,mc} \in \{\times, \text{␣}\} \\ p'' & \text{else} \end{cases} \quad \text{where :}$$

$$r = \#_{\$}(p) \ \wedge \ p = p_0\$p_1\$\ldots p_{r-1}\$ \ \wedge \ c = |p_0|$$

$$\exists!\ (mr, mc) : p_{mr,mc} \in \{\&, \sim\}$$

$$\wedge$$

$$p'_{mr} = p_{mr,0}\ldots p_{mr,mc-1}\ \mathbf{man\_left(p_{mr,mc})}\ p_{mr,mc+1}\ldots p_{mr,c-1}\ \wedge$$

$$p'_{mr+1} = p_{mr+1,0}\ldots p_{mr+1,mc-1}\ \mathbf{man\_came(p_{mr+1,mc})}\ p_{mr+1,mc+1}\ldots p_{mr+1,c-1}\ \wedge$$

$$p' = p_0\$\ldots p_{mr-1}\$\ p'_{mr}\$p'_{mr+1}\$\ p_{mr+2}\$\ldots p_{r-1}\$$$

$$\wedge$$

$$p''_{mr+2} = p_{mr+2,0}\ldots p_{mr+2,mc-1}\ \mathbf{box\_came(p_{mr+2,mc})}\ p_{mr+2,mc+1}\ldots p_{mr+2,c-1}\ \wedge$$

$$p'' = p_0\$\ldots p_{mr-1}\$\ p'_{mr}\$p'_{mr+1}\$p''_{mr+2}\$\ p_{mr+3}\$\ldots p_{r-1}\$$$

The first case describes the event when the man cannot move down, because there is a wall right below him. The second case is similar, there is a box below the man, but he cannot push it, since there is a wall or another box below the box.

The third case is simply the man moving one step down to an empty cell.

The fourth specify the case, when the man pushes one box below him to the next empty cell that's why three lines are changed.

1.2.2 Tilt maze

Tilt maze is a game with a lot of various designs. Most popular is a wooden maze, where you can tilt the board on four different sides. You need to get a ball from starting position to goal position without falling into one of the holes (example 1.3)

Figure 1.3: Wooden tilt maze [2]

Online games used this idea, but instead of avoiding holes, you need to collect checkpoints.

Rules: The game board is divided into square cells, forming two-dimensional grid. Each pair of cells can be separated by a wall. Moreover, there are walls on every perimeter edge. Exactly k cells are occupied by checkpoints and one other by a ball. The ball can move in one of the four basic directions, moving until it hits the nearest wall.

Every time the ball goes across the cell containing a checkpoint (or stops on it), the checkpoint will be collected (resulting in it's disappearing from the board). The game ends when all checkpoints are collected.

Formally, Tilt maze can be defined using definition 1.2. For better understanding of the definition, we'll show the *position* string for one small instance (image 1.4).

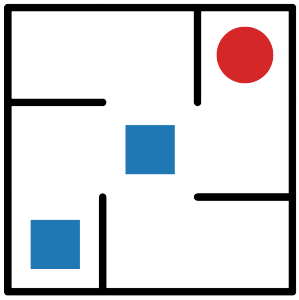


Figure 1.4: Small Tilt maze instance

Position is:

$$␣␣B\$␣C␣\$C␣␣\$\#WWW\$W␣␣\$␣␣W\$WWW\$\#WWW\$␣␣W\$W␣␣\$WWW\$\#$$

The position has three parts separated by #. Character \$ is used as a newline. In the first part, every ␣ means an empty cell, every **C** a checkpoint and **B** is the ball.

The second and third part describes horizontal (described from left to right) and vertical (from top to bottom) walls, the ␣ character representing missing wall.

The definition of the alphabet, *win* and *init* predicate:

- $\Sigma = \{␣, \mathbf{B}, \mathbf{C}, \mathbf{W}, \$, \#\}$; ␣ is an empty cell or no-wall, **B** is the ball, **C** is a checkpoint, **W** is a wall, \$ is end of line, and # is a separator for different parts.
- $win(p) = (\#_{\mathbf{C}}(p) = 0)$; if we collected all checkpoints
- $init(p) = (\#_{\mathbf{C}}(p) \geq 1)$; if there is at least one checkpoint to collect

Defining *val* predicate will be little more complicated. We need to check, that the input string follows the intended structure.

$$\begin{aligned} val(p) = (\#_{\#}(p) = 3) \ \wedge \ (p = g\#h\#v\#) \wedge \\ r = \#_{\$}(g) \ \wedge \ g = g_0\$g_1\$ \dots g_{r-1}\$ \wedge \\ c = |g_0| \ \wedge \ \forall i : g_i \in \{\text{␣}, \mathbf{B}, \mathbf{C}\}^c \wedge \\ \#_{\mathbf{B}}(g) = 1 \wedge \\ \#_{\$}(h) = r + 1 \ \wedge \ h = h_0\$h_1\$ \dots h_{r-1}\$, h_r\$ \wedge \\ \forall i : h_i \in \{\text{␣}, \mathbf{W}\}^c \wedge \\ h_0 = h_r = \mathbf{W}^c \wedge \\ \#_{\$}(v) = c + 1 \ \wedge \ v = v_0\$v_1\$ \dots v_{c-1}\$, v_c\$ \wedge \\ \forall i : v_i \in \{\text{␣}, \mathbf{W}\}^r \wedge \\ v_0 = v_c = \mathbf{W}^r \end{aligned}$$

The predicate checks three different parts: the first part must have r rows with c characters from the set $\{\text{␣}, \mathbf{B}, \mathbf{C}\}$, having $\mathbf{B}$ on exactly one place; the second part has $r + 1$ rows, where first and last row needs to be full of walls; and the third parts is similar to the second.

The last part of definition missing is the recursive successor function S. We define it using four auxiliary function for different directions:

$$S(p) = \{move\_up(p), move\_right(p), move\_down(p), move\_left(p)\}$$

We show the definition of *move_ up*, as the other three are fairly similar.

$$move\_up(p) = \begin{cases} p & \text{if } h_{br,bc} = \mathbf{W} \\ move\_up(p') & \text{if } h_{br,bc} \neq \mathbf{W} \end{cases} \quad \text{where}$$

$$p = g\#h\#v\# \ \wedge \ r = \#_{\$}(g) \ \wedge \ g = g_0\$g_1\$ \dots g_{r-1}\$ \wedge$$

$$\exists! \ (br, bc) : g_{br,bc} = \mathbf{B} \wedge$$

$$g'_{br} = g_{br,0} \dots g_{br,bc-1} \ \text{␣} \ g_{br,bc+1} \dots g_{br,c-1} \wedge$$

$$g'_{br-1} = g_{br-1,0} \dots g_{br-1,bc-1} \ \mathbf{B} \ g_{br-1,bc+1} \dots g_{br-1,c-1} \wedge$$

$$g' = g_0\$ \dots g_{br-2}\$ \ g'_{br-1}\$g'_{br}\$ \ g_{br+1}\$ \dots g_{r-1}\$ \wedge$$

$$p' = g'\#h\#v\#$$

In the function *move_up* we split the position into different parts and get row and column of the ball. If there is a wall above this cell ($h_{rb,rc} = \mathbf{W}$), we will return the same position p, since the ball cannot move in this direction.

If there is no wall, we can define next position p', where the ball is replaced by empty cell and the cell above it is replaced by the ball. However, the function doesn't return p' directly, but calls itself recursively on this position. The returned position will be the one, where the ball cannot go up anymore.

Conveniently, this definition will take care of collecting the checkpoints along the way, since the ball always replaces the next cell, and leaves empty one when moving.

1.3 Problem Difficulty

1.3.1 Computational complexity

Computational complexity describes asymptotic difficulty of a game, using O-notation or determining membership in a complexity class.

1.3.1.1 Definition of complexity classes

Complexity classes are defined using Turing machines.

Hopcroft and Ullman [19] formally define a (one-tape) Turing machine as a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

whose components have the following meanings:

Q: The finite set of *states* of the finite control.

Σ: The finite set of *input symbols*.

Γ: The complete set of *tape symbols*; Σ is always a subset of Γ.

δ: The *transition function*. The arguments of $\delta(q, X)$ are a state q and a tape symbol X. The value of $\delta(q, X)$, if it is defined, is a triple (p, Y, D), where:

1. p is the next state, in Q.
2. Y is the symbol, in Γ, written in the cell being scanned, replacing whatever symbol was there.

3. D is a *direction*, either L or R, standing for "left" or "right", respectively, and telling us the direction in which the head moves.

q_0**:** The *start state*, a member of Q, in which the finite control is found initially.

B**:** The *blank* symbol. This symbol is in Γ but not in Σ; i.e., it is not an input symbol. The blank appears initially in all but the finite number of initial cells that hold input symbols.

F**:** The set of *final* or *accepting* states, a subset of Q.

Turing machine model suitable for this section is TM with input and several working tapes – the input tape is read-only. Also, the input tape is bounded to the length of the input string and working tapes are infinite to the right.

Definitions of time and space-bounded machines [33]:

Definition 1.6 *Deterministic Turing machine A is* $S(n)$ space-bounded*, if every computation A on the string w of length n uses at most $S(n)$ cells on every working tape.*

Definition 1.7 *Non-deterministic Turing machine A is* $S(n)$ space-bounded*, if:*

- *(strong def.) Every computation on the input string w of length n uses at most $S(n)$ cells on every working tape.*
- *(medium def.) Every computation on the input string $w \in L(A)$ of length n uses at most $S(n)$ cells on every working tape.*
- *(weak def.) For every input string $w \in L(A)$ of length n exists computation that uses at most $S(n)$ cells on every working tape*

Definition 1.8 *Deterministic Turing machine A is* $T(n)$ time-bounded*, if TM makes at most $T(n)$ steps on every input string w of length n.*

Definition 1.9 *Non-deterministic Turing machine A is* $T(n)$ time-bounded*, if*

- *(strong def.) Every computation on every input string w of length n uses at most $T(n)$ steps.*
- *(medium def.) Every computation on every input string $w \in L(A)$ of length n uses at most $T(n)$ steps.*

- *(weak def.) For every string $w \in L(A)$ of length n exists computation that makes at most $T(n)$ steps.*

1.3.1.2 Complexity classes

Definition 1.10 *For every function, there is TM bounded by it, defining a class of languages. Classes are divided by determinism of TM and whether it is time or space bounded, using the* strong *definition of non-determinism (1.7, 1.9).*

Define classes:

$$DSPACE(S(n)) = \{L \mid \exists\ DTM\ A, \text{ that is } S(n) \text{ space-bounded } \wedge\ L = L(A)\}$$
$$NSPACE(S(n)) = \{L \mid \exists\ NTM\ A, \text{ that is } S(n) \text{ space-bounded } \wedge\ L = L(A)\}$$
$$DTIME(T(n)) = \{L \mid \exists\ DTM\ A, \text{ that is } T(n) \text{ time-bounded } \wedge\ L = L(A)\}$$
$$NTIME(T(n)) = \{L \mid \exists\ NTM\ A, \text{ that is } T(n) \text{ time-bounded } \wedge\ L = L(A)\}$$

Especially interesting are classes that are space or time bounded by polynomial, or other simple function (regarding only type, not coefficients). Most famous classes are P, NP and $PSPACE$

Definition 1.11

$$P = \bigcup_{f \text{ is polynomial}} DTIME(f(n)) = \bigcup_{k>0} DTIME(O(n^k)) = DTIME(n^{O(1)})$$
$$NP = \bigcup_{f \text{ is polynomial}} NTIME(f(n)) = \bigcup_{k>0} NTIME(O(n^k)) = NTIME(n^{O(1)})$$
$$PSPACE = \bigcup_{k>0} DSPACE(n^k) = DSPACE(n^{O(1)})$$

Less formally, we can describe each complexity class as:

- P – problem is in P, if there exists deterministic algorithm to find the solution in polynomial time with respect to the size of the input
- NP – problem is in NP, if there exists deterministic algorithm to verify the solution in polynomial time with respect to the size of the input
- NP-complete – problem H is NP-complete, if it is NP and every other problem in NP, can be transformed to H in polynomial time

- *PSPACE* – problem is in *PSPACE*, if there exists algorithm to find the solution in polynomial space with respect to the size of the input
- *PSPACE*-complete – problem H is *PSPACE*-complete, if it is in *PSPACE* and every other problem in *PSPACE*, can be transformed to H in polynomial time

The transformation used in the definition of *NP*-complete and *PSPACE*-complete problems is a deterministic many-one reduction running in polynomial time.

1.3.1.3 Puzzle

There is a lot of research, asking how hard are puzzles for the computer – described in terms of complexity classes.

To determine a membership of the puzzle to a complexity class, we define the puzzle as a language.

The puzzle instance is winnable if there exists a solution, corresponding to the definition1.5.

Definition 1.12 The language of winnable instances $L(P)$ *of the puzzle* P *is*

$$L(P) = \{I \mid I \in \Sigma^+ \ \wedge \ val(I) \ \wedge \ init(I) \ \wedge \ I \text{ has a solution}\}$$

where $P = (\Sigma, S, val, win, init)$.

Using these definitions, we can determine the membership of a puzzle P to a complexity class, by asking how hard it is to determine for arbitrary instance I of a puzzle P whether $I \in L(P)$.

Many of puzzles are proved to be *NP*-complete. Especially puzzles that require filling the grid – a few examples from Wikipedia [4]:

- Nurikabe – proofed by McPhail [26]
- Fillomino – proofed by Yato [40]
- general Sudoku, also called Number Place – proved by Yato and Seta [39]

Examples of sliding-block problems that are *PSPACE*-complete [5]: Sokoban [12] and Rush Hour [15].

There is also a small number of games belonging to P. For example Tilt Maze, which can be reduced to Quell that was proved to be polynomial by Tejada [36].

Puzzles are of various types. Sokoban and Sudoku are both puzzles, both satisfying the definition 1.2, but they have different difficulty. When we look onto the state graph of Sudoku, it is a directed acyclic graph – while playing, we always add numbers, so we cannot get to the same state twice. This means that this graph has polynomial depth, which concludes that Sudoku is in NP, because you can guess a solution and check it in polynomial time. This holds for every puzzle where the goal is filling a grid (with numbers, colors etc.).

1.3.2 Human problem solving

On the other hand, there is the question, how hard are puzzles for people. Still, it is not known how these two things relate. We have no ambition in understanding the whole process behind human solving. Hence we focus on interesting parts from the view of Computer Science, considering our insufficient background in psychology and human cognition.

This section gives a brief overview of research on difficulty assessment with a focus on automated methods for puzzle games.

Ashlock and Schonfeld [10] automatically grade the difficulty of Sokoban. The difficulty is measured by two different approaches: mean time-to-solution by an evolutionary algorithm and number of failures to solve a board. These measures are used to order the levels by difficulty.

Mantere and Koljonen [25] study the problems involved in solving, generating and rating Sudoku puzzles with genetic algorithms (GA). The last objective, whether GA can be used as a rating machine that evaluates the difficulty of a given Sudoku instance, is approached by testing whether puzzles that are considered difficult for humans are also difficult for the genetic algorithm.

Kreveld, Löffler and Mutser [38] introduce a method for automatically rating the difficulty of puzzle game levels. They study three games: Flow, Lazors and Move. Their method takes multiple attributes of these games, such as level size, and combines these into a difficulty function.

Komanová [21] introduces thirteen logic rules, which describes solving of the game

Nonograms. She uses this methods to solve various instances and predict the time of a human solver.

Other example is research done by Czech team in Brno. They have several papers, from which we chose articles by Jarušek and Pelánek *What Determines Difficulty of Transport Puzzles?* [20] and *Human Problem Solving: Sokoban Case Study* [31].

In first work, they study three games: Sokoban, Rush hour and Replacement puzzle. The second is focused only on Sokoban. They ask a question "What determines difficulty of solving a problem?". They found problems, where the difficulty isn't explained by previous research. They argue that metrics like state space size, or length of the shortest solution cannot capture the difficulty. They propose a computational model of human problem solving. They analyzed user data and observed that human players behave more randomly at the beginning, and later go more straightforwardly to the goal. The model is a combination of random and optimal walk, however it does not provide explanation of "how people think", it just simulates their behaviour.

In Sokoban Case Study they also formalize a metric based on state space bottleneck. Humans spent most of the time in states before the bottleneck, but once it's passed they quickly find a way to the winning state. The concept is not specific for Sokoban and can be used for any puzzle. Unfortunately, it didn't turned out to be good difficulty metric.

In our work, we use similar approach as Kreveld, extracting attributes from different games and trying to learn the difficulty function. However, for training we decided to use machine learning.

1.4 Data sources

In order to study difficulty of solving puzzles, we needed to collect the data. Since the goal is to determine how solving time depends on syntactic attributes of levels, we need to know how every level looks like.

Besides that, we have to be able to compute one metric, which we will use for predicting (e.g. average time or median of time)

The subject of our work is Tilt maze and Sokoban. The first puzzle is rather unknown. Every website contains the same set of several tens of instances. Furthermore,

they do not collect the data about individual users and their solutions.

Sokoban on the other hand is fairly popular and there are many websites offering playing Sokoban. Unfortunately, nor in this case, there isn't a lot of pages, where they collect time data. For example the website `sokoban.info` measure your solving time, but you cannot find the time of other players for specific level[3]. We successfully contacted the website `sokobanonline.com`, which also measures the time of solution. Unfortunately, they do not save this information.

The most useful page turned out to be `logic-games.spb.ru/`. It is a Russian website, where you can play different games including Sokoban. It saves all data about their players. Every level has difficulty rating, which is average time of playing. Per every player, you can only see his last five hundred plays. Sadly, we were not able to contact administrators of this website, so we didn't get the access to full data about users. Thus we decided to only use the last plays for every user, since there is less than two hundred users who played more levels. The data are from March 2018.

Finally, since we follow the research of Czech team, we have access to the data from their Problem Solving Tutor `tutor.fi.muni.cz`. This page offers wide range of puzzles and games. Every move of every game is saved for every user. They also save the time of every step. For every game one table is generated, with relation between users and all levels. Every cell contains either a number - time took by user to save the level, or nothing, if the user did not solve the level.

From this website we downloaded the data for Sokoban and Tilt Maze. Tilt maze user data are from January 2016 and Sokoban data from February 2018.

[3]We tried to contact them, but they never replied.

Chapter 2

Predicting average log solution time

In this chapter, we study two puzzle games: Tilt maze and Sokoban. Our goal is to explain *how difficulty of a puzzle depends on its syntactic attributes*.

In the first section, we give a brief explanation of syntactic attributes. In the second section, we provide a short introduction to machine learning, which is used for training and prediction of solution time. For each game, we use the same models and techniques that are described in parts 2.2.2 and 2.2.3.

The last two sections describe Tilt maze and Sokoban respectively. First we explain how we collect necessary data, followed by a section about feature engineering – where we break down every syntactic attribute computed.

The last part of every puzzle section consists of experiments and results using models introduced in the subsection 2.2.2.

2.1 Syntactic attributes

As we stated before, we want to predict the solution time based on syntactic features of level instances. We consider *syntactic features* anything that can be deterministically computed from the description of level instance.

For example, number of boxes in Sokoban, number of checkpoints in Tilt maze, width of grid etc. Great source of syntactic features is a state space (state graph, def. 1.4) of a level instance. We extract attributes such as number of reachable states, length of the shortest solution, number of viable states etc.

On the other hand, Sokoban level may consist of several similar parts that have to

be solved in very similar way. This fact has positive influence on the solution time by a human solver, since he can "learn" this pattern and use it multiple times. However, we cannot rigorously measure or quantify this "property". Hence, we do not consider this property to be a syntactic attribute. We could define some attribute which would relate to this property, but it would be only inaccurate approximation.

Previous example indicates that we are not able to fully explain the difficulty of the majority of puzzles using only syntactic features.

Syntactic features transform a description of level instance to a vector of numbers. In this setting, its clear we want to learn a difficulty function. We choose to learn it by using methods of machine learning, which are discussed in the next section.

2.2 Machine learning

"Machine Learning is the science of getting computers to learn and act like humans do, and improve their learning over time in autonomous fashion, by feeding them data and information in the form of observations and real-world interactions." [7]

Machine learning tasks can be categorized into a wide range of categories. The basic distinction is to *supervised* and *unsupervised* learning. We talk more about *supervised* learning in the next section.

2.2.1 Introduction

Supervised learning refers to any machine learning process that learns a function from an input type to an output type using data comprising examples that have both input and output values. Two typical examples of supervised learning are classification learning and regression. In these cases, the output types are respectively categorical (the classes) and numeric. [34]

Our goal is to predict solving time, which indicates towards the regression. We need to construct an input matrix X with features for every level and an output vector y with one number per every level – the value we want to predict (average time, median etc.).

There is a number of models solving regression tasks. However, we don't only want to predict solving time, but also understand which features have the largest impact.

This rules out for example regression based on *k-nearest neighbors*, since we cannot extract information about impact of individual features.

We talk about chosen models in the next section.

2.2.2 Models

We use three types of models: *Linear regression*, *Support vector machine* and *Random Forest*. All models are implemented in `Python` library – `scikit-learn` [30]. Specific models are:

- `LinearRegression` – Ordinary least squares Linear Regression.
- `Ridge` – Linear least squares with L2 regularization.
- `Lasso` – Linear Model trained with L1 prior as regularizer.
- `LassoLars` – Lasso model fit with Least Angle Regression.
- `ElasticNet` – Linear regression with combined L1 and L2 priors as regularizer.
- `LinearSVR` – Linear Support Vector Regression.
- `RandomForestRegressor`

The first six models contain an output attribute called `coef_` and the last one contains `feature_importances_` – all of them giving us a clue how each feature affects the model.

2.2.3 Training

Our goal is to train a model which generalizes well. Generalization can be measured by *testing error* – error on previously unseen data. When training a model on dataset with an input matrix X and an output vector y, we split the dataset to a *training* and *testing* dataset. While training, the model never sees any sample from the *testing* set. The split is done only once and randomly.

Many of the models assume that the individual features look like standard normally distributed data: Gaussian with zero mean and unit variance. For this, we use *standardization*, which is implemented as `StandardScaler` in `scikit-learn`. The mean and variance is determined only from the training data and then the same transformation is used to both training and testing set. Every one of our models except the `RandomForestRegressor` uses dataset with scaled features.

Many estimators have hyper-parameters. These are the parameters that are not directly learnt during the training. For example `Lasso` has `alpha`, or `LinearSVR` has hyper-parameter `C`. These parameters can be chosen by hand, but it is common to search a space of parameters to find the ones with best *cross-validation* score.

Cross-validation is a process, where the *training* set is partitioned into k folds – subsets. The learning algorithm runs k times, each time one fold acts as a *validation* set, and all others creates an *estimation* set. The algorithm learns on an *estimation* set, and computes score on *validation* set. The final score is average of the k validation scores.

Searching for the optimal hyper-parameters can be done using `GridSearchCV` – exhaustive search over specified parameter values for an estimator. Parameters for each model are:

- `LinearRegression` – no hyper-parameters
- `Ridge` – 49 values of `alpha` evenly spaced on a log scale from 10^{-3} to 10^{3}
- `Lasso` – 49 values of `alpha` evenly spaced on a log scale from 10^{-3} to 10^{3}
- `LassoLars` – 49 values of `alpha` evenly spaced on a log scale from 10^{-3} to 10^{3}
- `ElasticNet` – 49 values of `alpha` evenly spaced on a log scale from 10^{-3} to 10^{3}
- `LinearSVR` – 36 values of `C` evenly spaced on a log scale from 10^{-2} to 10^{3}
- `RandomForestRegressor` – `n_estimators` from list $[1, 5, 10, 15]$

`GridSearchCV` refits an estimator using the best found parameters on the whole dataset.

Feature selection [24] is an important and widely used approach to dimensionality reduction. For a dataset with N features and M dimensions (or features, attributes), feature selection aims to reduce M to M' and $M' \leq M$. It is an important and widely used approach to dimensionality reduction. It is important to note that the selected features are a subset of original features.

`RFE` from `Python` package `scikit-learn` selects features by recursively considering smaller and smaller sets of features. `RFECV` performs feature ranking with recursive feature elimination and cross-validated selection of the best number of features.

Because we don't have a large amount of data – many features and small number of samples, we use this approach to try to reduce the dimensionality. This also gives

us a clue, which features are important and which are not.

The output vector consists of an average logarithmic time. For one specific level, with user times in a vector $t = (t_0, \dots t_{n-1})$ considering only users that solved this level, the output variable is:

$$\frac{1}{n} \sum_i \log t_i$$

2.2.3.1 Score

The quality of prediction can be measured in different ways. We choose the *coefficient of determination*, which can be directly extracted from all trained models by calling the `score()` method.

Coefficient of determination R^2 is defined as:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} \qquad SS_{res} = \sum_i (y_i - \hat{y}_i)^2 \qquad SS_{tot} = \sum_i (y_i - \bar{y})^2$$

where $y = (y_0, \dots y_{n-1})$ is the output vector, $\bar{y}$ is the mean of y, and $\hat{y}$ is a vector of predicted values.

The best possible score is 1.0. The constant model, which would always predict the mean, would get a score 0. Since the model may perform worse than the constant model, the score can drop below zero.

The coefficient of determination shows how well a regression model fits the data. Its value represents the percentage of variation that can be explained by the trained model. If every point fits the data, the score would be 1.0; if the score is 0.5 – only half of the variation is explained.

2.3 Tilt maze

The first game we study is called Tilt maze. Rules of this game are described in subsection 1.2.2. In this part, we talk about data collection and extraction of features, ending in results.

2.3.1 Data collection

Before we can use supervised learning, we need to create a proper dataset. Every dataset consists of an input matrix X and an output vector y. In our case, the output

vector is average log time for every level and X consists of one vector per each level containing all features.

2.3.1.1 Time data

In section 1.4 we mentioned website Problem Solving Tutor. We obtained special user from administrators, which allows us to download all data for every puzzle. Specifically, we download a data table – relation between users and levels. A small fraction from this table can bee seen in the table 2.1, where the first line are level IDs and the first column are user IDs. Each cell contains number of seconds of solution.

Login	**221**	**223**	**225**	**227**	**228**	**229**	**230**	**231**	**232**	**233**	**234**	**235**	**236**	**294**	**295**	**296**
U1	11	12			64									42		
U2	2	28	45	59	41			27	100		107		83	21	78	
U4	25			205	12	29		32	116	55	89		96	49	38	
U5		5	9		45	160		53	41				39	32		
U6	10	11												89	46	
U7	5	11	10	41	16	49	204	28	39	89	85	283	123	18	27	
U8	5	11		43	54										60	
U9	8	11	18	45	20	82		53						44	392	
U10	44	15			26											
U13	9	12	19	37	49	28	128	42	66	50	37	273	136	47	40	460
U14		65	33	86	65	135									59	
U15		18	38		32	96									70	
U17	4															
U20	22	28	35	48	31	141	270	124	163	41	176	179	47	35	123	494
U21	35	42	129		63			37						83		
U22	11	29	32		31			51						98	161	
U24	10	20	19		194											
U25	26	16	23	96	23	57	514	15	99	120	268		156	22	48	251
U26	23			86	22	58	306	37	80	51	144	184	192	65	47	

Table 2.1: Tilt maze user time data

In the table 2.2 we provide a summary of collected data.

For our objective, we computed the average log time for every level.

#users	6265
#levels	110
avg #users per level	882
avg #levels per user	15
total time	90 days 5 hours

Table 2.2: Summary information about collected data

2.3.1.2 Levels

To compute any features, we need to know how every level looks like. We use `Python` [16] programming language, specifically library `Selenium` [28] and its module `webdriver` to scrape the data from the website.

First, we need to login into the website, since the games aren't accessible otherwise. From the time table, we have all level IDs. After analyzing the website, we learned that every level has its own unique URL, which can be constructed from its ID. Using `javascript` on this specific URL, we can extract all necessary variables:

- `hasTopWall` – 2D `bool` array – for each cell, whether it has a top wall
- `hasRightWall` – 2D `bool` array – for each cell, whether it has a right wall
- `hasBottomWall` – 2D `bool` array – for each cell, whether it has a bottom wall
- `hasLeftWall` – 2D `bool` array – for each cell, whether it has a left wall
- `isCheckpoint` – 2D `bool` array – for each cell, whether it contains a checkpoint
- `ball_row` and `ball_col` – integer coordinates of the starting ball position
- `width`, `height` – size of the game board

Using `webdriver` we open a browser instance and navigate through all the level URLs, where it executes scripts returning the all required variables. We put all the variables into a dictionary and save it using the `json` library to one file per level.

2.3.2 Feature engineering

In this subsection we describe every feature used for prediction. The basic features are `width` of a level[1], `tile count` and `checkpoint count`. All three can be easily extracted from level description.

[1] `height` is omitted, since all game boards are squares.

Other features are computed using one of two graphs – *state* graph and *segment* graph. An explanation of these graphs is given in following sections.

2.3.2.1 State graph

The definition 1.4 of state graph offers general description of state space over positions. However, when we talk about the state graph of one specific instance, we don't have to include the information about walls. We simplify every position (or state) to a pair:

$$[ball\ position,\ set\ of\ collected\ checkpoints]$$

There exists a homomorphism from this description of states, to a valid positions satisfying the definition 1.2.

To explain the *set of collected checkpoint*, we number checkpoints from zero, as displayed in image 2.1. With this numbering, the second part of the state is essentially a bitfield – one bit for each checkpoint.

For the game instance in image 2.1, the starting state is: $[(2,2),0]$, because no checkpoints are collected yet. In the winning state all checkpoints have to be collected, so the bitfield is 511, corresponding to binary number 11111111.

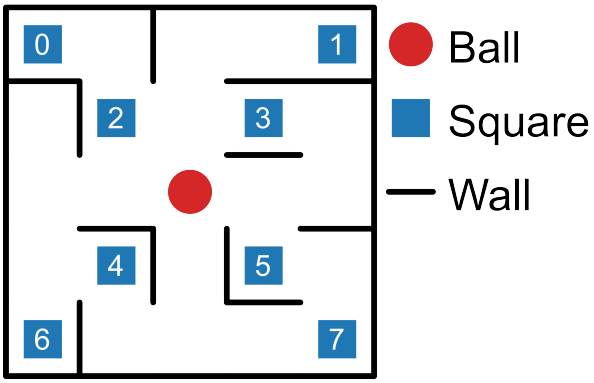


Figure 2.1: Tilt Maze with numbered checkpoints

We construct a state graph, where every state has at most four neighbors – the ball can only move in four directions, sometimes hitting a wall before it actually moves – this will not create any new vertex or edge.

In reality, we first compute neighbours for each cell and remember for each edge a bitfield – which checkpoints can be collected during this move. Later, during traversing of the state space, we can use this information directly: if we are in the state $[ball_0, ch]$

and the edge UP points to $ball_1$ with checkpoint change *change*, we can easily get the next state as $[ball_1, ch|change]$. The bitwise *or* will compute new collected checkpoints perfectly – if we already collected some of the checkpoints on this edge, it will not be counted twice and new checkpoints will be added to the state bitmap.

Using state graph traversal we can compute five features: `shortest_path`, `reachable_states_count`, `reachable_tile_count`, `viable_states_count` and `shortest_path_tiles`. First, we run a basic BFS algorithm [27] from the starting state to all the others. We intentionally traverse the whole state space in the interest of computing the number of reachable states. During the BFS we also remember the shortest path from starting state to all the others.

If we encounter final state – which we can detect by comparing its checkpoint bitfield to the number $(2^{\#checkpoints+1} - 1)$ – we do not continue traversing the state space from this vertex, as the game ends when all checkpoints are collected. We also remember the shortest path to first winning state visited.

To compute the feature *reachable_ tile_ count*, we initialize a set containing only first position of the ball. During the exploring of neighbours we add ball position to this set.

Lastly, we remember the `previous` map, which gives us previous states for every state visited and `final_states` array, which includes all winning states.

After the first BFS completes, we run the second – reversed BFS, which starts from all final states, and goes along the edges in the transposed graph (with help of the `previous` map), to visit all states reachable from winning states. This gives us the number of viable states – in which the game is still winnable.

Feature `shortest_path_tiles` should correspond to a minimal number of tiles that the ball has to traverse along any shortest path. We "create" a graph, where every node lies on some shortest path. In reality, we only create a set of vertices lying on shortest paths and use this set to filter vertices during the search.

From the first BFS we have the length L of the shortest path. Initially, the set of *on_ shortest* vertices contains only winning states with distance L. After that, we traverse the transposed graph, but for vertex v only considering previous neighbour u, if $dis(u) + 1 = dis(v)$ stands – so the vertex u is closer to the start.

After that, we run Dijkstra's algorithm [13] to find the minimal number of tiles along

any shortest path. We compute this on filtered graph – only vertices in *on_shortest* set are considered and every edge has weight, based on how many tiles the balls traverse during this move.

2.3.2.2 Segment graph

We already mentioned that Tilt maze belongs to complexity class P. The game Quell is essentially the same, however walls aren't between cells, but each cell can be a wall. The instance of Quell is displayed on image 2.2, where the blue rain droplet collects small golden pearls.

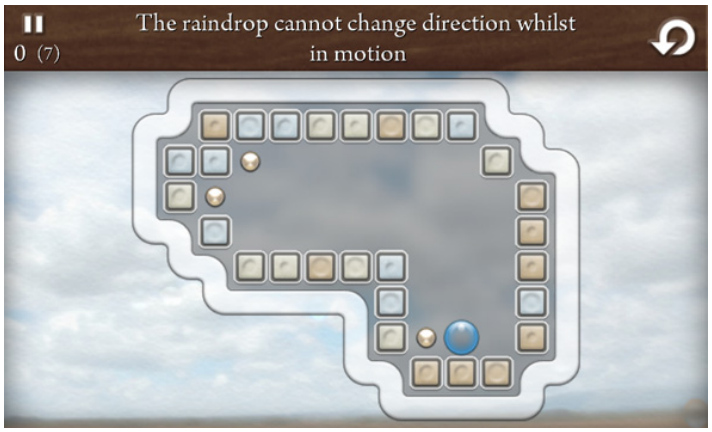
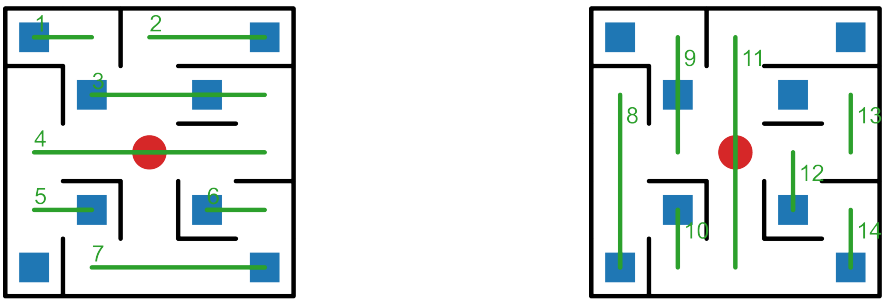


Figure 2.2: Quell level [6]

Quell was proved to be in P by Tejada [36]. Since each game of Tilt maze can be reduced to Quell, Tilt maze is in P too.

The proof is based on maximal segments – every row and column can be divided into these segments (discarding segments of length 1). The illustration can be found on images 2.3a and 2.3b.



(a) Horizontal (b) Vertical

Figure 2.3: Maximal segments

It's possible to construct directed graph G, where every segment is a vertex and containing one special vertex v_0 for starting location of the ball. G has directed edges from v_0 to the vertices corresponding to the maximal segments incident to the starting position of the ball. In our case, it would be v_4 and v_{11}. In graph G, every other edge from vertex u to v exists, only if one of the endpoints of the maximal segment for u is on the maximal segment for v.

Graph G is shown on image 2.4, we call is *Segment graph* The red vertex is special vertex for starting location of the ball. Vertices, which correspond to segments with checkpoints, are marked blue.

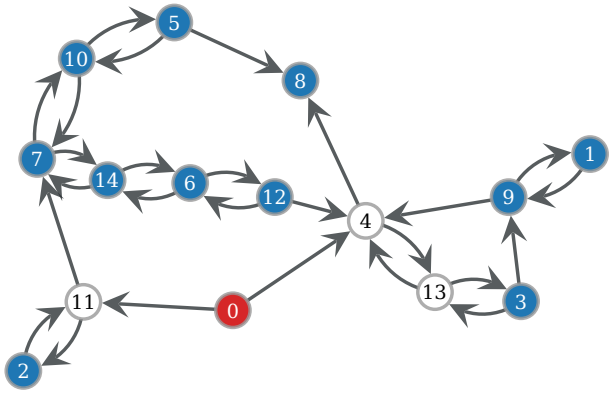


Figure 2.4: Tilt Maze – segment graph G

In the graph G we find the strongly connected components, constructing graph G', which is directed acyclic graph, where every checkpoint belongs to at most two vertices. The graph in the picture 2.5 is G, with coloured groups of vertices corresponding to strongly connected components.

The proof in Tejada's article is based on reduction to 2-SAT. We can construct clause, which is satisfiable only if all the checkpoints can be collected. Since we can solve 2-SAT in polynomial time, subsequently we can construct the path of solution from it in polynomial time.

We use the number of strongly connected components and the number of strongly connected components containing at least one checkpoint as two features: `scc_count` and `scc_checkpoint_count`.

In implementation, we start by creating special starting node with ID 0. Then we create nodes for all horizontal maximal segments and all vertical maximal segments.

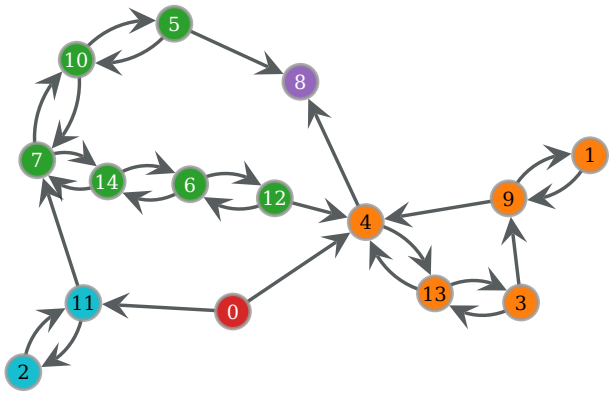


Figure 2.5: Graph G with coloured strongly connected components

Every vertex gets at most two children – nodes matching the segments at the endpoints. Also, every checkpoint remembers, which nodes it belongs to.

After that we run the Kosaraju's algorithm [8] for finding strongly connected components, which essentially consist of two separate runs of depth-first search algorithm – first to order vertices, the second to traverse the transposed graph and mark the components.

From this run, we can easily extract the number of strongly connected components to get the feature `scc_count`. With one pass over the checkpoints, we can detect, which components have checkpoints in them, and get their count for the feature `scc_checkpoint_count`.

2.3.3 Experiments and results

In this section, we describe the results of experiments. First, we adjust the set of features, afterwards we predict the average log time using different models and find their parameters using methods *GridSearch* and select the best features by running *RFECV* from the section 2.2.3.

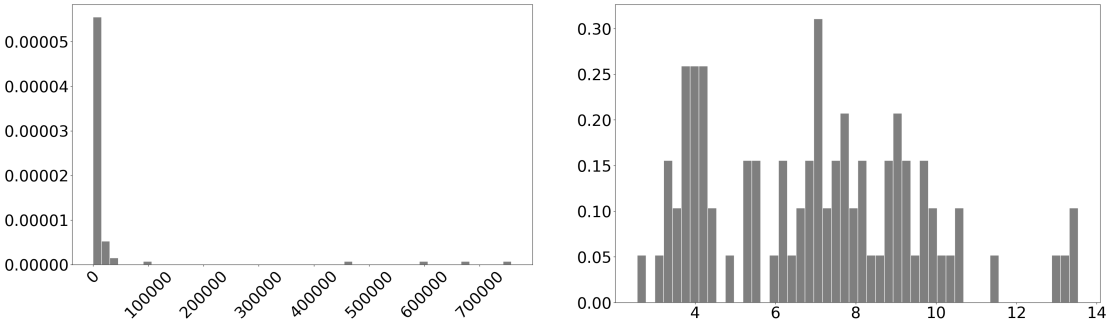
2.3.3.1 Final set of features

Many machine learning models expect that features are drawn from normal distribution. Hence, we decided to plot histograms of all features. They can be found in appendix A.1, images A.1a – A.10a. One of the basic transformations, when dealing with non-normal data, is *logarithmic* function. We applied natural logarithm to all

features, producing histograms A.1b – A.10b.

Captions of all histograms contain a *p-value* of Kolmogorov-Smirnov test for normality. If the *p-value* is less than 0.05, the feature is probably not normally distributed. All histograms are generated only from the training set.

Features `reachable_states_count`, `viable_states_count`, `shortest_path` and `scc_count` don't have normal distributions, but their *log* versions have distributions more similar to normal. Hence we use only the *log* features. One example is in the image 2.6, where we clearly see that the original feature has a lot of small values and only a few large values.

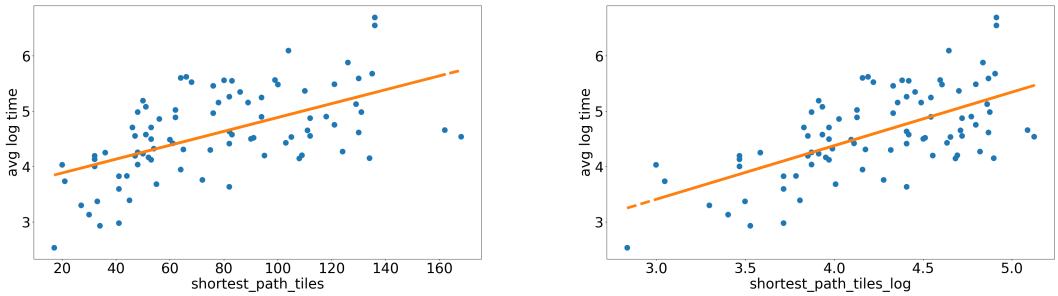


(a) Original (p-value: 8.882e-16) (b) Log (p-value: 0.229)

Figure 2.6: Reachable States Count

Not all other features have normal distribution, but neither their log versions have it, so we just use the original versions.

One special case is the feature `shortest_path_tiles`. We use the *log* version of this feature, because it has a higher *p-value* and intuitively, when looking on images 2.7a and 2.7b, the *log* feature seems to be more linearly dependent.



(a) Original (b) Log

Figure 2.7: Shortest Path Tiles Scatter Plot

Finally, we remove features `checkpoint_count` and `tile_count`, because they highly correlate with features `reachable_states_count` and `width` respectively (with use of Spearman's correlation coefficient). The correlation would skew the results and the feature coefficients would be affected by it.

The final set of features is: `width`, `shortest_path_log`, `reachable_tile_count`, `reachable_states_count_log`, `viable_states_count_log`, `scc_checkpoint_count`, `shortest_path_tiles_log`, `scc_count_log`. We believe that a level is more difficult with increase of any of these features.

2.3.3.2 Grid search

We explained *Grid search* in the section 2.2.3. It trains all models and finds the best parameters. Best parameters, train and test score of all models are in the table 2.3.

Model	Parameters	Train score	Test score
`LinearRegression`		0.72	**0.6887**
`Ridge`	`alpha`: 56.23	0.64	0.5360
`Lasso`	`alpha`: 0.10	0.64	0.5402
`LassoLars`	`alpha`: 0.10	0.64	0.5402
`ElasticNet`	`alpha`: 0.18	0.64	0.5433
`LinearSVR`	`C`: 0.37	0.68	0.5792
`RandomForestRegressor`	`n_estimators`: 15.00	**0.79**	0.4611

Table 2.3: Tilt maze – GridSearch parameters and scores

As we can see, the best train score has `RandomForestRegressor`. Random forests have a tendency to overfit, which is probably the case, since the test score is the lowest. The second best estimator is basic Linear Regression, which has the best test score 0.6887. It means, we can explain 69% of the data variance, as described in the subsection 2.2.3.1.

Now, let's look onto the individual coefficients. They are shown in the table 2.4. Note that the first six lines contain `coef_` field and the last row corresponds to `feature_importances_` field, which values are from range $[0, 1]$ summing up to

1.0[2].

Model	width	reachable tile count	scc checkpoint count	reachable states count log	shortest path tiles log	viable states count log	shortest path log	scc count log
`LinearRegression`	-0.26	0.39	-0.19	0.39	**0.79**	-0.35	-0.39	0.21
`Ridge`	0.10	0.10	0.10	0.11	0.14	-0.01	0.06	**0.15**
`Lasso`	0.0	0.13	0.0	0.06	**0.27**	0.0	0.0	0.24
`LassoLars`	0.0	0.13	0.0	0.06	**0.27**	0.0	0.0	0.24
`ElasticNet`	0.04	0.11	0.03	0.08	**0.22**	0.0	0.0	0.20
`LinearSVR`	0.03	0.17	-0.01	0.31	**0.39**	-0.29	-0.05	0.13
`RandomForestRegressor`	0.03	0.22	0.12	0.21	**0.33**	0.04	0.02	0.04

Table 2.4: Tilt maze – GridSearch coefficients

In the table 2.4, the highest coefficients in each row is marked bold. Feature *shortest path tile log* has the largest impact on all models, except for `Ridge`, where it's the second highest.

2.3.3.3 RFECV

RFECV is described in the section 2.2.3. It recursively eliminates features and finds the optimal number of features. Train and test scores are shown in the table 2.5.

[2]Values in the table are rounded to two decimal places, therefore the sum of the last row may be less or more than 1.0

Model	Train score	Test score
`LinearRegression`	0.46	0.4779
`Ridge`	0.64	**0.5373**
`Lasso`	0.61	0.5054
`LassoLars`	0.61	0.5054
`ElasticNet`	0.60	0.4981
`LinearSVR`	0.45	0.4613
`RandomForestRegressor`	**0.80**	0.4055

Table 2.5: Tilt maze – RFECV scores

More interesting are the features that this methods selected. In the table 2.6, we can see the coefficients of all models. If a feature wasn't selected during training the model, its cell contains *False*.

Model	width	reachable tile count	scc checkpoint count	reachable states count log	shortest path tiles log	viable states count log	shortest path log	scc count log
`LinearRegression`	False	False	False	**0.78**	False	-0.40	False	False
`Ridge`	0.10	0.10	0.10	0.10	0.14	False	0.06	**0.15**
`Lasso`	False	0.11	False	False	**0.28**	False	False	0.23
`LassoLars`	False	0.11	False	False	**0.28**	False	False	0.23
`ElasticNet`	False	0.12	False	False	**0.27**	False	False	0.22
`LinearSVR`	False	False	False	**0.75**	False	-0.40	False	False
`RandomForestRegressor`	0.01	0.19	0.07	0.24	**0.36**	0.01	False	0.11

Table 2.6: Tilt maze – RFECV coefficients

The highest value in each row is marked bold. As you can see, features *reachable tile count*, *reachable states count log*, *shortest path tiles log* and *scc count log* are chosen by at least half of the models. We may say, these are the most important ones. On the other hand, features *width*, *scc checkpoint count* and *shortest path log* are chosen by two or only one model. We assume, these do not impact the difficulty very much.

2.3.3.4 Conclusion

We conclude that the difficulty of Tilt maze levels depends on given syntactic attributes. The most impact has *shortest path tiles* features, followed by *reachable tile count*, *reachable states count log* and *scc count log*.

Using these attributes, we are able to explain **69**% of data variance. Since we had a rather small number of levels, it would be worth to get more data. Since we now know which features impact difficulty the most, it would be worth to generate levels where these features are fixed, but other things vary. This could help us to find new attributes effecting the difficulty.

2.4 Sokoban

The second game we study is Sokoban. Rules can be found in section 1.2.1. In this part we describe the collection of the data, extraction of all features and finally the experiments and their results.

The *Data collection* and *Experiments and results* sections are divided into two parts: Czech and Russian dataset. We use the same algorithms and the same features for both datasets, which are described in subsection *Feature engineering*.

2.4.1 Data collection

Before we can create features and train models, we need to get the output variable – average logarithmic solving time and the representation of all levels.

2.4.1.1 Czech data

We have the data table – relation between users and levels, where every cell corresponds to seconds the user took to solve a level (or empty, if not solved). The table looks the same as in the section about Tilt maze 2.3.1.1. In table 2.7 we can observe the summary for the Sokoban game.

Levels

For scraping all levels from the website, we use `Python` [16] programming language, specifically library `Selenium` [28] and its module `webdriver`.

#users	5081
#levels	81
avg #users per level	415
avg #levels per user	7
total time	60 days 23 hours

Table 2.7: Summary information about collected data

Every level lives on its own unique URL, with only one `javascript` variable `rows`. This variable represents a game board. It consist of characters:

- `␣` – a space, if a cell is empty
- `#` – a wall
- `@, +` – the man on and empty cell or on a target cell
- `$, *` – a box on an empty cell or on a target cell
- `., *, +` – a target cell with nothing, a box or the man standing on it

From this two-dimensional character array, we construct variables `width`, `height`, `isWall` (2D `bool` array), `boxes` (array of `int` pairs), `targets` (array of `int` pairs), `man_row, man_col`, which we save as `json` to one file per level.

We use the same structure of saved files in the Russian dataset, which gives us the ability to handle them in the same way.

2.4.1.2 Russian data

Scraping data from Russian Sokoban website is a bit trickier, since it is only one PHP site. But using asynchronous `javascript` calls, we are able to get information and navigate through various levels.

One `javascript` execution for variable `this.levelList` give us all level IDs. With execution of: `this.controller.requestGame(level_id);` we can navigate through all instances.

When on level page, we can retrieve the grid where every cell is a dictionary. From the values, we can detect whether it is a wall or an empty cell; whether there is a box or not; whether there is a man or not; whether it is a target cell. We process it to a dictionary, which is later saved as json to a text file. The structure is the same as in

the previous section.

The total number of levels is **3525**. However, since one of the features is number of states in the state space, some of the levels were too large to have the state space constructed and held in 16GB RAM of the computer. Discarding all large levels get us to final number of **762** levels, which is still almost ten times more than the Czech dataset.

In this case, we weren't able to contact the owners of the website, so we don't have full data about all users. However, we have access to last five hundred plays of every user. We get the data for every player from the rating table who had at least three plays, totaling in almost 13000 users. However, since we filter out most of the levels, there is only 1584 users that played at least one of the chosen level instances. In the table 2.8 we can see the final summary of the data.

#users	1584
#levels	762
avg #users per level	55
avg #levels per user	26
total time	76 days 12 hours

Table 2.8: Summary information about collected data

2.4.2 Feature engineering

In this section, we talk about every feature created for predicting the average log time. The basic features, extracted directly from a level, are `box_count`, `tile_count`, `width` and `height`.

For creation of other features, we always need to run an algorithm on a state graph. Because of that, we create the state graph only once and save it to a file.

For graph creation and all algorithms we use `C++` library called `Boost` [3].

2.4.2.1 State graph

The state graph is defined using the definition 1.4, where nodes are positions. When creating the state graph of a specific instance, we can consider every node (state) to

be a pair:

$$[man\_position, boxes\_positions]$$

We don't have to include information about walls or targets, because they don't change locations during playtime. There exists a simple homomorphism from these states to a positions satisfying the definition 1.2.

We represent the man position as an integer pair and positions of the boxes as a set of integer pairs. Set is more suitable than list, because it doesn't impose arbitrary order on the boxes.

There are at most four edges from every vertex v, because the man can move in four directions. However, sometimes he may immediately hit an obstacle. When the man moves in one direction, there are five possible cases:

- man moves to an empty cell;
- man tries to walk into a wall;
- man pushes the box into empty cell in the same direction, and takes its place;
- man tries to push a box into a wall;
- man tries to push a box into another box.

Only in two cases the state actually changes.

Furthermore, there are no outgoing edges from winning state, since the game ends there.

2.4.2.2 Graph creation

During the graph creation, we traverse the state space and produce new vertices. Every vertex gets new ID and a few other fields: `starting, winning, h, min_pushes`. First two are `bool` indicators, whether the state corresponding to this vertex is starting/winning. The `h` field is an `integer`, which is a value of heuristic function in this vertex. We talk about the heuristic function more in the section 2.4.2.7 about A*. The last `integer` field `min_pushes` is discussed in the section 2.4.2.6.

Furthermore, we save one information about every edge, called `pushed`, which is `true` for every edge (u, v), where the boxes positions in v are different from positions in u. This field is used during computation of `shortest_boxes_path` in 2.4.2.5.

In the interest of saving space, nodes do not remember which states they represent. Almost none of the algorithms need to know anything about the man position or boxes

positions. Everything needed is included in the vertex fields. However, if necessary, the mapping between the states and vertices is saved to a specific file using `boost` bidirectional map – `bimap`.

2.4.2.3 Features: `shortest_path` and `reachable_states_count`

Natural choice for first features is `shortest_path` and `reachable_states_count`. We expect that level is more difficult, if the shortest solution is long. The same goes for the number of reachable states, if there is a lot of states – it would take more time to find the right solution.

Reachable states count can be extracted directly from the state graph, which is loaded from a file created during graph creation phase. Reachable states count corresponds to the number of vertices.

Shortest path is computed via basic breadth-first search algorithm.

2.4.2.4 Features: `viable_states_count` and `good_boxes_tiles`

Level is simple, if no matter what we do, we can still win it. On the other hand, level is hard when every wrong step takes us to a dead state (the game is lost). The number of dead states is difference between reachable states and viable states. We believe that more viable states means level is more difficult.

We count viable states with BFS run on the transposed graph. Since `boost` library support only BFS with one starting point, we create artificial vertex which has incoming edges from all winning vertices. After the run, the count of visited vertices is giving us the number of viable states.

The trick with transposing the graph is also used when counting good boxes tiles. The tile is defined as good for a box, if there exists a viable state in which some box is positioned at this tile. Some cells are often bad – e.g. in corners, along walls. If the instance has small number of good tiles, the player must move boxes carefully. We assume that small `good_boxes_tiles` means level is more difficult.

On image 2.8, you can see good tiles marked as blue shaded squares.

After getting all viable vertices, we load the mapping of vertices to states. With one pass over viable states, we add all boxes positions to a set of good boxes tiles.

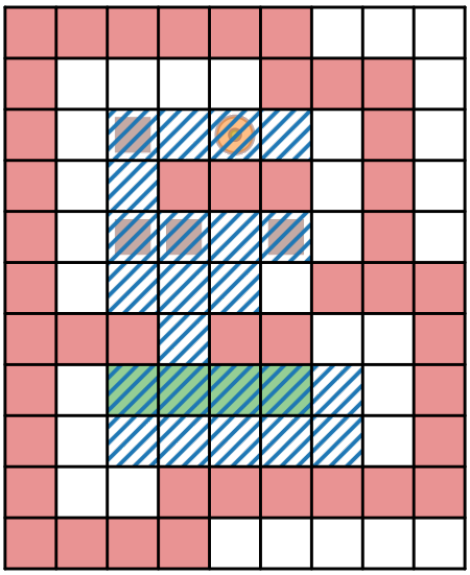


Figure 2.8: Good tiles

2.4.2.5 Feature: `shortest_boxes_path`

Other interesting feature is the length of solution using the smallest number of box pushes. E.g. on Russian Sokoban website, it's possible to control the man using mouse. We can even click to a cell, and if it's possible he will go there. So this feature should correlate with solution time[3].

Edge of the state graph can be annotated with zero or one – if one of the boxes moved or not. We get a graph, in which we want to find the shortest path based on the edge weight. In this case, we use *Dijkstra algorithm* [13], for finding the shortest path.

2.4.2.6 Feature: `counter_intuitive_steps`

In Sokoban, the move may be considered counter-intuitive, if it gets a box further from target cells. Humans while playing, wants to get boxes closer, not further from the target cells. Hence, we compute at least how many counter-intuitive moves you have to do along any shortest path. Formally, in every state we compute value `min_pushes`:

$$min\_pushes = \sum_i \min_j \ dis(b_i, t_j)$$

where b is a vector of boxes positions and t is a vector of target cells positions. This value is saved in every vertex during graph creation.

[3]However, we don't know how many users know about this feature and use it.

In practice, before creating the graph, we run BFS several times, once from every target cell, to get the shortest distance from each target cell to all other cells. We can use this information directly when creating new state to find the nearest target cell for every box and sum over it.

Every move – every edge from a vertex u to a vertex v in the graph – is considered counter-intuitive, if the difference $min\_pushes(v) - min\_pushes(u)$ is greater than zero.

To compute the minimal number of counter-intuitive moves, we need to create a graph, where every edge lies on some shortest path. We start by running BFS algorithm and computing distance from starting to all vertices. After that, we identify the length L of the shortest path.

We create a set of vertices on the shortest paths, starting with the winning vertices with distance L. Than we run a special search from this winning states to other states, but for vertex v only considering neighbours u, where $dis(u) + 1 = dis(v)$ stands – so the vertex u is closer to the start.

After that, we create a graph G' using `filtered_graph`, where we specify to only use vertices from the created set. We construct *weight map* which assign all edges 0 or 1 – based on positiveness of `min_pushes` difference.

Finally, one run of *Dijkstra algorithm* on G' will compute the shortest distance to winning states – giving us the minimal number of counter-intuitive steps along any shortest path.

2.4.2.7 Feature: `astar_states_count`

A* [18] is a heuristic search on a weighted directed graph guided by a heuristic function h. Intuitively, A* follows paths to the goal that are estimated by the heuristic function to be the best paths. The heuristic should be a lower-bound of real distance from each node to the goal.

In our case, the goal is every state, where all boxes are positioned on target cells. As a heuristic function we use a minimal number of pushes needed to get all boxes to target cells:

$$h(b) = \min_{b' \in S(b)} \sum_i dis(b'_i, t_i)$$

where $S(b)$ is a set of all permutations of vector b.

Vector b is a vector of all boxes positions and vector t is a vector of all targets positions. The *dis* function is computed using simple BFS search from each target cell to all other cells – computing minimum number of pushes to get a box from each cell to the specific target cell.

Since we find the minimum through all permutations of boxes, we find the matching which needs minimum number of pushes. The value of h function is computed during graph creation and saved inside each vertex.

During the A* search, we compute how many vertices where visited. We believe, that humans do something similar to A* search, so the higher number of visited nodes means level is more difficult.

2.4.2.8 Features: `scc_count` and `scc_shortest`

The next interesting feature is a number of strongly connected components. Level with one component (except the one for the winning component) is easy, we can never do any wrong move. These two features are similar to first two features `shortest_path` and `reachable_states_count`, only on graph of strongly connected components.

First, we run *Tarjan's algorithm* [35] to find the components. After that we create the graph, where every vertex corresponds to one strongly connected component. Every winning vertex will have its own component, since there are no outgoing edges. The component containing starting vertex is marked starting as well. On this graph we run simple BFS algorithm, to get the shortest path – feature `scc_shortest`. The `scc_count` feature is simply the number of vertices in this graph.

2.4.2.9 Feature: `reachable_tile_count`

The last feature is number of reachable tiles. Every floor cell may be reachable, if there exists a state in which the man stands on this tile. We believe that more reachable tiles means easier level, because the man isn't restricted by the space and can easily maneuver the boxes.

This feature is computed with the same algorithm used in the graph creation. We traverse the state graph without explicitly creating the edges, and create a set of all tiles on which the man ever stood.

2.4.2.10 Technical details

Computing the features was also a technical challenge. We started by implementing everything in `Python`. At the beginning, we only traversed the state space once, computing shortest path, reachable states count and reachable tiles count.

Russian dataset brought a lot of problems, because running `Python` on eight hundred levels wasn't fast enough. We switched to `C++` and started computing each feature with different file – giving us the possibility to rerun only one feature. Since every feature is based on a state graph, we decided to use `boost` to create these graphs and save them.

At first, we saved files in readable `.dot` format, which was over 2GB for the largest instances. Using `boost` archive, we could get the space used down to 50%, and at the same time we sped up the writing and reading of graphs.

2.4.3 Experiments and results

In this section we talk about the experiments and their results. This section is divided into three parts: Czech dataset, Russian dataset and conclusion.

2.4.3.1 Czech dataset

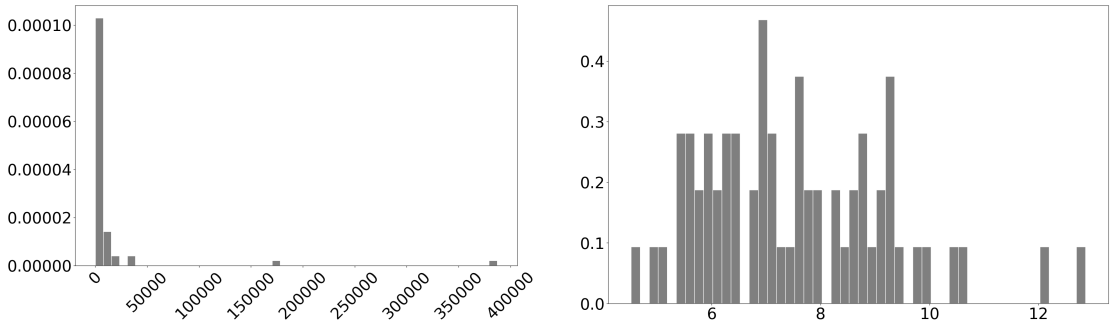
We first adjust the set of features, then run *Grid search* and *RFECV* from the section 2.2.3.

Final set of features

A lot of machine learning models expect that features are drawn from the normal distribution. Because of that, we plot histograms for all features (appendix A.2) – all *a*) images from A.11a to A.24a. The *b*) parts are logarithmic versions of all features (images from A.11b to A.24b).

Because not all features are normal, we try to apply *log* function on them, which is often used to normalize data. We also test each feature and its *log* version for normality using Kolmogorov-Smirnov test. The *p-value* is a part of every caption, when the value is larger than 0.05 we consider the feature to be normal.

One example is shown on image 2.9, where the original feature isn't considered normal, but the *log*-transformed one is.



(a) Original (p-value: 1.513e-10) (b) Log (p-value: 0.738)

Figure 2.9: Viable States Count

In the end, we apply *log* function to features: `tile_count`, `astar_states_count`, `scc_shortest`, `scc_count`, `viable_states_count`, `counter_intuitive_steps` and `reachable_states_count`. Thus, we only use the *log* of these features and remove the original. Some of the other features do not have normal distribution, but neither their *log* versions, so we just use the original features.

Furthermore, we actually remove `reachable_states_count` and `scc_count` features, because they both highly correlate with the feature `astar_states_count_` (we use Spearman's correlation coefficient), which would skew the coefficients.

The final set of features is: `box_count`, `width`, `height`, `reachable_tile_count`, `shortest_path`, `shortest_boxes_path`, `tile_count_log`, `astar_states_count_log`, `viable_states_count_log`, `counter_intuitive_steps_log`, `scc_shortest_log` and `good_boxes_tiles`. We expect that level is easier, if we increase number of *good boxes tiles* or *reachable tiles*, but harder if we increase any other feature.

Grid search

Grid Search finds the best parameters for a model (description in 2.2.3). The best parameters, train and test score for all models is shown in the table 2.9.

Model	Parameters	Train score	Test score
`LinearRegression`		**0.8149**	**0.7293**
`Ridge`	`alpha`: 1.33	0.8047	0.7164
`Lasso`	`alpha`: 0.01	0.8056	0.7154
`LassoLars`	`alpha`: 0.01	0.8056	0.7154
`ElasticNet`	`alpha`: 0.01	0.8037	0.7149
`LinearSVR`	`C`: 0.52	0.7768	0.6815
`RandomForestRegressor`	`n_estimators`: 15.00	0.8146	0.6039

Table 2.9: CZ Sokoban – GridSearch parameters and scores

`LinearRegression` has the best train and test score. The test score 0.7293 means that the model can explain 73% of the data variance (description of score in 2.2.3.1). The second best estimator (regarding train score), `RandomForestRegressor` has the lowest test score, probably because of overfitting.

Now, let's look on coefficients of individual features, which are shown in the table 2.10. We note that in first six models, the table contains `coef_` field, however, the last row is `feature_importances_` field[4].

Model	box count	width	height	reachable tile count	shortest path	shortest boxes path	good boxes tiles	viable states count log	counter intuitive steps log	astar states count log	scc shortest log	tile count log
`LinearRegression`	0.23	-0.19	-0.15	-0.61	**0.64**	-0.24	0.41	-0.34	0.11	0.21	-0.16	0.58
`Ridge`	0.25	0.04	0.05	-0.47	**0.57**	-0.15	0.21	-0.22	0.14	0.14	-0.15	0.19
`Lasso`	0.26	0.0	0.0	-0.53	**0.61**	-0.14	0.20	-0.15	0.12	0.10	-0.14	0.26
`LassoLars`	0.26	0.0	0.0	-0.53	**0.61**	-0.14	0.20	-0.15	0.12	0.10	-0.14	0.26
`ElasticNet`	0.26	0.0	0.0	-0.49	**0.59**	-0.13	0.18	-0.16	0.13	0.10	-0.14	0.25
`LinearSVR`	0.24	-0.06	0.01	-0.57	**0.63**	-0.15	0.40	-0.35	0.15	0.23	-0.21	0.27
`RandomForestRegressor`	0.01	0.01	0.03	0.03	**0.77**	0.04	0.01	0.01	0.03	0.01	0.04	0.01

Table 2.10: CZ Sokoban – GridSearch coefficients

The highest value in each row of the table 2.10 is marked bold. Clearly, the length of *shortest path* has the highest impact on every model. It has the largest coefficient, even

[4]Values are from the range $[0, 1]$, summing up to 1.

when we look on absolute values. The second interesting feature is *reachable tile count*, which has the lowest value for every model, except `RandomForestregressor`. The lowest number in the random forest would mean that this feature is least important.

This means that longer the shortest path is, the harder a level is. On the other hand, more reachable tiles means easier level, probably because the man has more space for moving and maneuvering boxes.

RFECV

Train and test scores for all models after *RFECV* are shown in the table 2.11.

Model	Train score	Test score
`LinearRegression`	**0.81**	0.6827
`Ridge`	0.74	0.6304
`Lasso`	0.74	0.6317
`LassoLars`	0.74	0.6317
`ElasticNet`	0.74	0.6318
`LinearSVR`	0.74	**0.7261**
`RandomForestRegressor`	0.79	0.6370

Table 2.11: CZ Sokoban – RFECV scores

We show features selected by *RFECV* in the table 2.12, where every cell is either *False*, if feature wasn't selected, or corresponds to the *RFECV* coefficient.

Model	box count	width	height	reachable tile count	shortest path	shortest boxes path	good boxes tiles	viable states count log	counter intuitive steps log	astar states count log	scc shortest log	tile count log
`LinearRegression`	0.21	-0.25	-0.19	-0.7002	0.6410	-0.15	0.45	-0.37	False	0.27	-0.18	**0.69**
`Ridge`	0.33	False	False	-0.5954	**0.5979**	False	False	False	False	False	False	0.33
`Lasso`	0.33	False	False	-0.6045	**0.6037**	False	False	False	False	False	False	0.34
`LassoLars`	0.33	False	False	-0.6046	**0.6036**	False	False	False	False	False	False	0.34
`ElasticNet`	0.31	False	False	-0.5241	**0.5829**	False	False	False	False	False	False	0.28
`LinearSVR`	0.37	False	False	-0.5484	**0.7336**	-0.37	0.23	-0.21	0.17	0.12	False	0.29
`RandomForestRegressor`	False	False	False	0.0571	**0.7433**	0.06	False	False	0.04	0.10	False	False

Table 2.12: CZ Sokoban – RFECV coefficients

As expected, two most important features: *shortest path* and *reachable tile count* were chosen by every model. Two other features: *box count* and *tile count* were chosen by six out of seven models, which means they probably impact the difficulty as well. All other features were chosen by less than half of models.

However, if we look onto `LinearSVR` and `LinearRegression` models, they are the only one choosing almost all features, but they also preserved high test score compared to other models. This could mean that even though those four features are the most important, they are not sufficient.

2.4.3.2 Russian dataset

We start by adjusting the dataset, then we run *Grid search* and *RFECV* described in the section 2.2.3.

Final dataset

We couldn't afford to remove outliers before, because of the small number of levels. Now, for this dataset, we have 762 levels, so the part of preprocessing is removing outliers.

An outlier is an observation that lies an abnormal distance from other values. We remove outliers based on interquartile range – IQR. We indetify lower an upper fences:

$$lower = q_1 - 1.5 \cdot IQR \qquad upper = q_3 + 1.5 \cdot IQR$$

where q_1 and q_3 are the first and third quartiles and $IQR = q_3 - q_1$. Every point lying outside the fences is considered to be an outlier.

First, as we already mentioned twice before, machine learning models assume that features are drawn from a normal distribution. Since not all our features are normal, we transform all features using logarithmic function, which may help them to become normal.

Histograms of all features and their *log* versions are in appendix A.3, images A.25 – A.38. We note that data on figures in only from the training set. Moreover, we remove the outliers, because they skew the information about normality. The *p-value* under all figures is from Kolmogorov-Smirnov test and if its value is larger than 0.05, the feature is normal.

Features which we apply *log* to are: `shortest_boxes_path`, `shortest_path` and `viable_states_count`. All others are used in their original form.

However, because some groups of features are highly correlated (regarding Spearman's correlation coefficient), we remove some of the features. For example `scc_count` and `reachable_states_count`, because they correlate with `astar_states_count`; feature `tile_count` correlates with `reachable_tile_count` and `viable_states_count` with `good_boxes_tiles`.

So the final set of features is: `box_count`, `width`, `height`, `astar_states_count`, `scc_shortest`, `reachable_tile_count`, `shortest_boxes_path`, `shortest_path_log`, `good_boxes_tiles` and `counter_intuitive_steps_log`, We expect that level is easier, if we increase number of `good_boxes_tiles` or `reachable_tiles`, but harder if we increase any other feature.

As we mentioned, we remove outliers from dataset. This means that final training set has 442 levels and final test set has 106 levels.

Grid search

First experiment is running *Grid search*, which is described in the section 2.2.3. It find the best parameters for every model. You can see the output in the table 2.13, including train and test scores.

Model	Parameters	Train score	Test score
`LinearRegression`		0.594123	0.5074
`Ridge`	`alpha`: 0.001	0.594123	0.5074
`Lasso`	`alpha`: 0.00133	0.593980	0.5095
`LassoLars`	`alpha`: 0.00133	0.593982	**0.5096**
`ElasticNet`	`alpha`: 0.001	0.594092	0.5086
`LinearSVR`	`C`: 10.00	0.576651	0.4847
`RandomForestRegressor`	`n_estimators`: 10.00	**0.783464**	0.3739

Table 2.13: RU Sokoban – GridSearch parameters and scores

The best estimator, with regard to train score, is `RandomForestRegressor`, however it has the lowest test score. This models probably overfit, since random forests have tendency to do it. All other models trained to almost identical train and test score.

Let's look onto individual coefficients in the table 2.14. We note that first six lines contain `coef_`, however the last contains `feature_importances_` field, which ranges from zero to one and sums up to one.

Model	box count	width	height	reachable tile count	scc shortest	counter intuitive steps	astar states count	good boxes tiles	shortest boxes path log	shortest path log
`LinearRegression`	0.42	-0.03	0.03	-0.15	-0.11	0.13	-0.01	-0.04	-0.06	**0.55**
`Ridge`	0.42	-0.03	0.03	-0.15	-0.11	0.13	-0.01	-0.04	-0.06	**0.55**
`Lasso`	0.41	-0.03	0.02	-0.13	-0.11	0.12	-0.01	-0.04	-0.04	**0.53**
`LassoLars`	0.41	-0.03	0.02	-0.13	-0.11	0.12	-0.01	-0.04	-0.04	**0.53**
`ElasticNet`	0.42	-0.03	0.02	-0.14	-0.11	0.13	-0.01	-0.04	-0.05	**0.54**
`LinearSVR`	0.43	0.11	0.04	-0.21	-0.13	0.12	0.01	-0.00	-0.14	**0.56**
`RandomForestRegressor`	0.05	0.03	0.02	0.04	0.04	0.03	**0.42**	0.04	0.06	0.27

Table 2.14: RU Sokoban – GridSearch coefficients

Clearly, the most important features are *shortest path log* and *reachable tile count*, first with the highest, second with the lowest values (everywhere, except for random forest, which has the worst test score). We believe that the longer the shortest solution is, the harder the level is. On the other hand, more reachable tiles means easier level, probably because the man isn't limited when manipulating boxes.

RFECV

RFECV recursively eliminates features, until optimal number is reached(description in 2.2.3). The test and train score is shown in the table 2.15.

Model	Train score	Test score
`LinearRegression`	0.593120	0.5123
`Ridge`	0.593158	0.5007
`Lasso`	0.588178	0.5223
`LassoLars`	0.588177	0.5223
`ElasticNet`	0.584241	**0.5239**
`LinearSVR`	0.386366	0.2610
`RandomForestRegressor`	**0.742122**	0.3398

Table 2.15: RU Sokoban – RFECV scores

You can see that random forest still overfits and linear SVR has low train and test score compared to other models. Let's look onto chosen features in the table 2.16.

Model	box count	width	height	reachable tile count	scc shortest	counter intuitive steps	astar states count	good boxes tiles	shortest boxes path log	shortest path log
`LinearRegression`	0.41	False	0.04	-0.20	-0.12	0.13	False	False	-0.09	**0.56**
`Ridge`	0.40	-0.04	False	-0.14	-0.11	0.13	False	False	-0.08	**0.55**
`Lasso`	0.37	-0.04	False	-0.09	-0.11	0.09	False	False	False	**0.46**
`LassoLars`	0.37	-0.04	False	-0.09	-0.11	0.09	False	False	False	**0.46**
`ElasticNet`	0.35	-0.04	False	-0.07	-0.09	0.09	False	-0.00	False	**0.44**
`LinearSVR`	0.30	-0.02	False	False	-0.12	0.08	-0.09	-0.28	False	**0.40**
`RandomForestRegressor`	0.07	False	False	False	False	False	**0.53**	False	0.09	0.32

Table 2.16: RU Sokoban – RFECV coefficients

Every cell in the table 2.16 contains *False* if a feature was not chosen by a model, otherwise contains corresponding coefficient.

Feature *shortest path log* has still the highest values and was chosen by all models. Interesting are also features *box count*, *width*, *scc shortest* and *counter intuitive steps*, which were chosen by almost all models. Moreover, the *box count* feature has the second highest values, suggesting that it has important impact on the difficulty. Also number of reachable tiles has still negative values, even though not always the lowest, but it was chose by all models with score more than 0.5.

2.4.3.3 Conclusion

We conclude that the difficulty of Sokoban levels depend on given syntactic attributes. In case of Czech dataset, we are able explain 73% of data variance; in the case of Russian dataset only 51% of data variation.

In both cases, features corresponding to the length of the shortest path and number of reachable tiles are the most important. When eliminating features, box count was chosen as an important feature in both cases as well.

In the future, it would be worth trying to get all data for Russian dataset, since missing data could be responsible for low scores. Furthermore, we could optimize the graph creation process, so it would be possible to compute features for larger levels, thus expanding number of samples.

In case of Czech dataset we have access to all user data, including individual steps user took to solve an instance. We could use this data for better state space analysis, visualization of the state space, eventually better understanding of human solving and creation of better features.

Conclusion

In our work, we studied how the structure of puzzle instances affects their difficulty, both from the viewpoint of computational complexity and the challenge it provides to human solvers. For our experiments, we chose puzzles Tilt maze and Sokoban, which provide a suitable challenge while remaining computationally tractable. We begin with formalizing these puzzles as decision problems and analyzing their computational complexity.

In the core of this work, we devise syntactic attributes and implement their extraction from level instances for the two chosen puzzles. We use these features to run experiments for prediction of the solution time of human players, thus assessing what makes these games difficult for humans. We are able to identify features which impact the difficulty the most.

In the case of Tilt maze, we identified attributes *shortest path tiles*, *reachable tile count*, *reachable states count* and *scc count* to be the most important – as their increase can help predict the growth in difficulty. From the computational point of view, finding solution of Tilt maze instances can be done in polynomial time with respect to the number of tiles. This means that the attribute *reachable states count* doesn't affect computational complexity at all – we can add one checkpoint to the level, which would almost double the number of reachable states, but doesn't change number of tiles. On the other hand, the number of strongly-connected components (*scc count*) in a segment graph affects computational complexity, since the polynomial algorithm is based on solving 2-SAT formula where every component gets one variable.

Number of reachable tiles in any instance corresponds to endpoints of segments, which means this attribute also affects computational complexity (every segment is a vertex in a segment graph). Contrarily, computational complexity doesn't directly depend on the feature *shortest path tiles*. We can create levels with millions of tiles

where the shortest path will only need one step to the right, or levels where the solution needs to visit all tiles at least once or some even twice. So the number of tiles on shortest path only gives us a lower bound of reachable tiles.

Sokoban game is *NP*-hard, more precisely it belongs to *PSPACE*-complete problems. Adding just one tile to a level instance may exponentially increase the difficulty for a computer. We identified three features which most affect complexity for human solvers: *length of the shortest path*, *reachable tile count* and *box count*. First of these features doesn't directly impact computational complexity. We can create a level with wide state space, but short solution or state space containing only one long solution path. So it only gives us a lower bound on states. The second feature *reachable tile count* is interesting. If a level contains a lot of reachable tiles, it's easier for a human than a level containing only a small number of reachable tiles. However, if we increase the number of reachable tiles, computer complexity increases.

When we look onto the third feature *box count* we find that the difficulty for both humans and computers grows as value of this feature increases. We can fix a level instance and just add one box, the number of states will grow exponentially, making it a lot harder for computers to find the solution.

When predicting the solution time of Sokoban, we couldn't get a better score (coefficient of determination) than 0.73. Hence, there is still a lot of space for improvement. There are two possible explanations, first, people solve this game in a fundamentally different way and we were not successful in capturing the essential attributes; alternatively, there may be other factors affecting the difficulty which we have yet to identify.

There are many directions for future research. We could find new well-defined problems, particularly other "move" puzzles and apply the same approach. We may design new attributes which would better explain the difficulty. Also, differences among individual solvers could be studied.

Bibliography

[1] 15 puzzle - wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/15_puzzle. (Accessed on 01/10/2016).

[2] Amazon.com: Brio labyrinth: Toys & games. https://www.amazon.com/Brio-34000-BRIO-Labyrinth/dp/B0001WGISK. (Accessed on 15/04/2018).

[3] Boost C++ Libraries. https://www.boost.org/. (Accessed on 24/04/2018).

[4] List of np-complete problems - Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/List_of_NP-complete_problems. (Accessed on 01/05/2016).

[5] List of pspace-complete problems - Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/List_of_PSPACE-complete_problems. (Accessed on 01/05/2016).

[6] Quell: The most relaxing free puzzle game for mobile devices [android, ios]. https://www.makeuseof.com/tag/quell-the-most-relaxing-free-puzzle-game-for-mobile-devices-android-ios/. (Accessed on 23/04/2018).

[7] What is machine learning? - an informed definition. https://www.techemergence.com/what-is-machine-learning/. (Accessed on 23/04/2018).

[8] Alfred V. Aho, John E. Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1983.

[9] J. R. Anderson, C. F. Boyle, and B. J. Reiser. Intelligent tutoring systems. *Science*, 228(4698):456–462, 1985.

[10] Daniel A. Ashlock and Justin Schonfeld. Evolution for automatic assessment of the difficulty of sokoban boards. *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.

[11] Michael E. Atwood, Michael E. J. Masson, and Peter G. Polson. Further explorations with a process model for water jug problems. *Memory & Cognition*, 8(2):182–192, Mar 1980.

[12] J Culberson. Sokoban is PSPACE-Complete. *International Conference on Fun with Algorithms*, (April):65–76, 1997.

[13] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, December 1959.

[14] Thomas S. Ferguson. Impartial Combinatorial Games. In *Game theory*, chapter I, pages I–1 to I–46. Mathematics Department, UCLA, second edition, 2014. `http://www.math.ucla.edu/~tom/Game_Theory/comb.pdf`.

[15] Gary William Flake and Eric B. Baum. Rush hour is pspace-complete, or "why you should generously tip parking lot attendants". *Theor. Comput. Sci.*, 270(1-2):895–911, 2002.

[16] Python Software Foundation. The Python Language Reference, version 3.6.1. `https://docs.python.org/3/reference/`.

[17] James G. Greeno. Hobbits and orcs: Acquisition of a sequential concept. *Cognitive Psychology*, 6(2):270 – 292, 1974.

[18] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[19] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation, 2nd edition. *SIGACT News*, 32(1):60–65, March 2001.

[20] Petr Jarušek and Radek Pelánek. What Determines Difficulty of Transport Puzzles? *Artificial Intelligence*, pages 428–433, 2011.

[21] Kristína Komanová. *Náročnosť riešenia maľovaných krížoviek*. Bachelor thesis, Comenius University in Bratislava, 2016.

[22] K Kotovsky, J.R Hayes, and H.A Simon. Why are some problems hard? evidence from tower of hanoi. *Cognitive Psychology*, 17(2):248 – 294, 1985.

[23] Kenneth Kotovsky and Herbert A Simon. What makes some problems really hard: Explorations in the problem space of difficulty. *Cognitive Psychology*, 22(2):143 – 183, 1990.

[24] Huan Liu. Feature selection. In Claude Sammut and Geoffrey I. Webb, editors, *Encyclopedia of Machine Learning*, pages 402–406. Springer US, Boston, MA, 2010.

[25] Timo Mantere and Janne Koljonen. Solving, rating and generating sudoku puzzles with ga. *2007 IEEE Congress on Evolutionary Computation*, pages 1382–1389, 2007.

[26] B.P. McPhail. *Complexity of Puzzles: NP-Completeness Results for Nurikabe and Minesweeper*. Reed College, 2003.

[27] E. F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[28] Baiju Muthukadan. Selenium with Python. `http://selenium-python.readthedocs.io/`.

[29] A. Newell and H.A. Simon. *Human problem solving*. Prentice-Hall, 1972.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[31] Radek Pelánek. Human Problem Solving: Sokoban Case Study, 2010.

[32] Zygmunt Pizlo and Zheng Li. Solving combinatorial problems: The 15-puzzle. *Memory & Cognition*, 33(6):1069–1084, Sep 2005.

[33] Branislav Rovan and Michal Forišek. Formálne Jazyky a Automaty, 2013. Skriptá.

[34] Claude Sammut and Geoffrey I. Webb. Supervised learning. In *Encyclopedia of Machine Learning*, pages 941–941. Springer US, Boston, MA, 2010.

[35] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[36] Pedro J Tejada. On the complexity of collecting items with a maximal sliding agent, 2014.

[37] Loren G. Terveen. Overview of human-computer collaboration. *Knowledge-Based Systems*, 8(2):67 – 81, 1995.

[38] Marc J. van Kreveld, Maarten Löffler, and Paul Mutser. Automated puzzle difficulty estimation. *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 415–422, 2015.

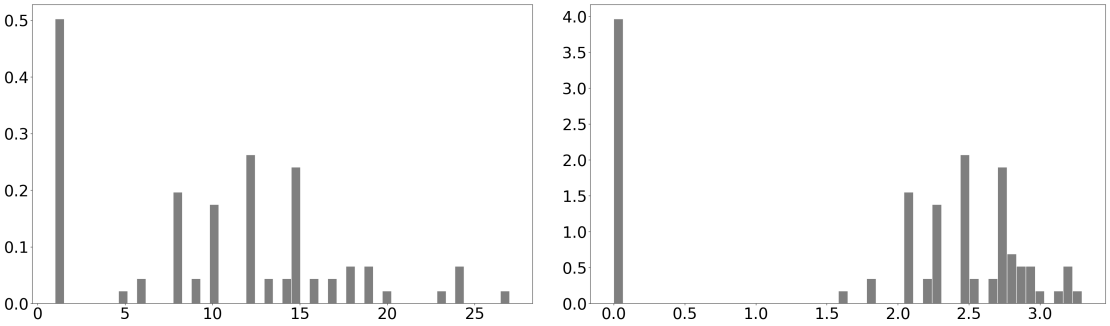
[39] T Yato and T SETA. Complexity and completeness of finding another solution and its application to puzzles, ipsg sig notes 2002-al-87-2, 2002.

[40] Takayuki YATO. *Complexity and completeness of finding another solution and its application to puzzles.* Master thesis, The University of Tokyo, 2003.

Appendix A

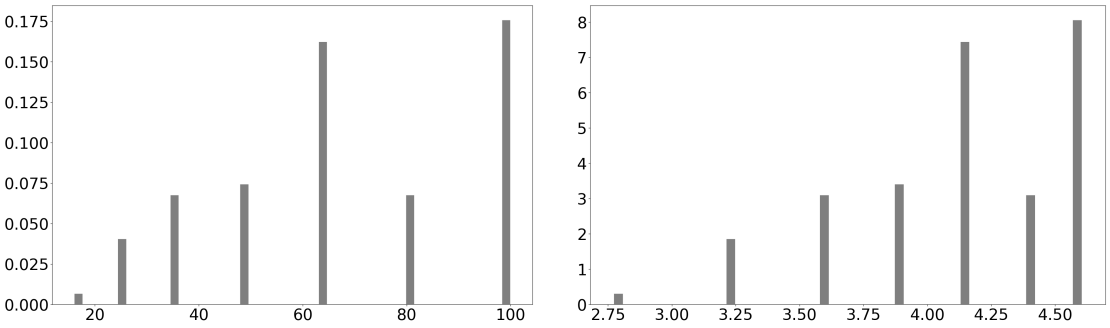
Feature histograms

A.1 Tilt Maze



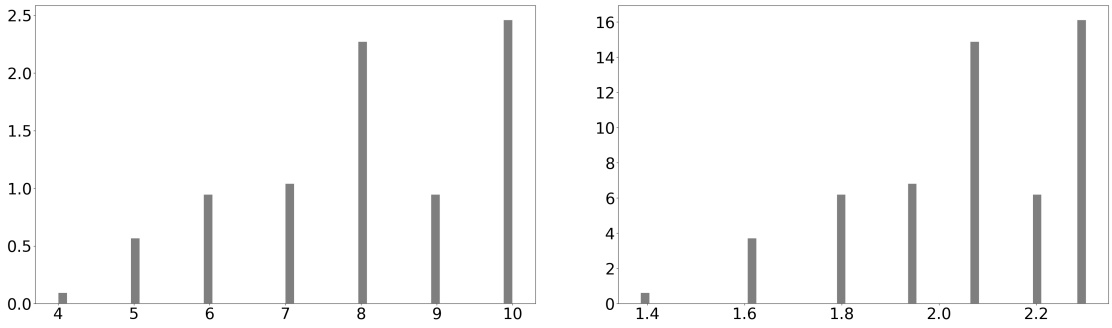
(a) Original (p-value: 0.010) (b) Log (p-value: 1.818e-06)

Figure A.1: Tilt Maze Feature – Checkpoint Count



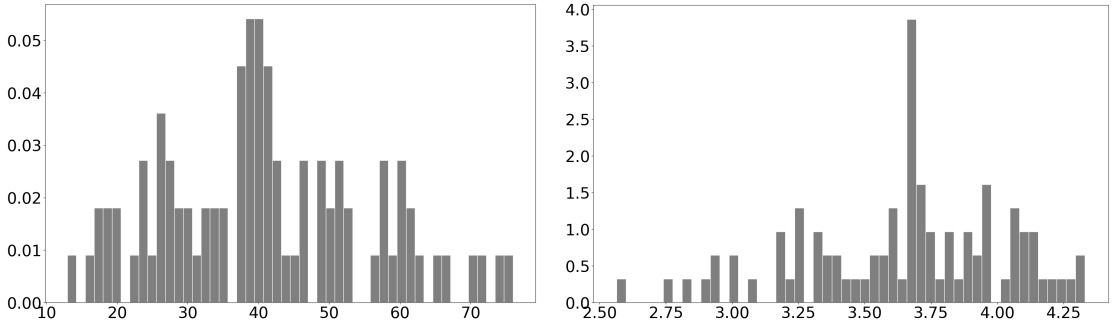
(a) Original (p-value: 0.003) (b) Log (p-value: 0.001)

Figure A.2: Tilt Maze Feature – Tile Count



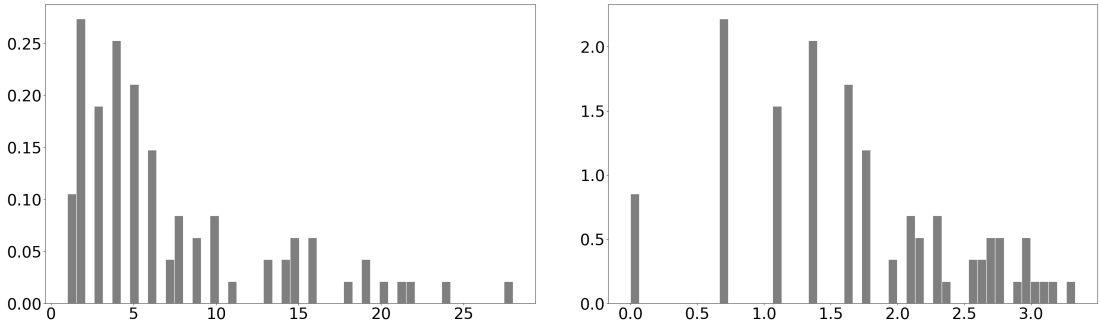
(a) Original (p-value: 0.009) (b) Log (p-value: 0.001)

Figure A.3: Tilt Maze Feature – Width



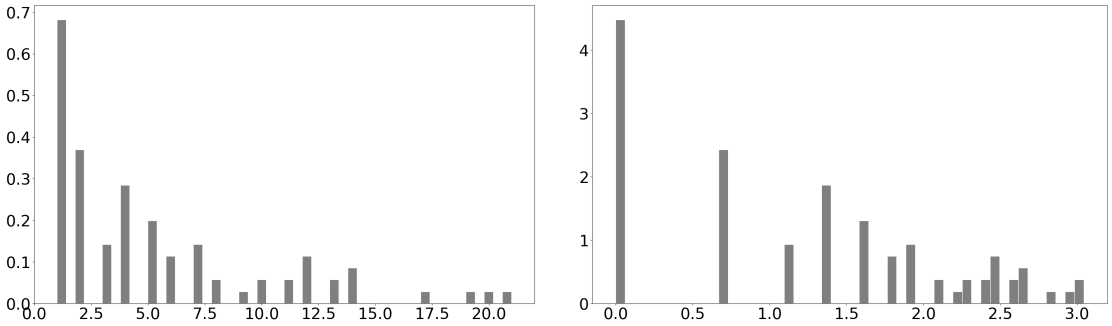
(a) Original (p-value: 0.290) (b) Log (p-value: 0.207)

Figure A.4: Tilt Maze Feature – Reachable Tile Count



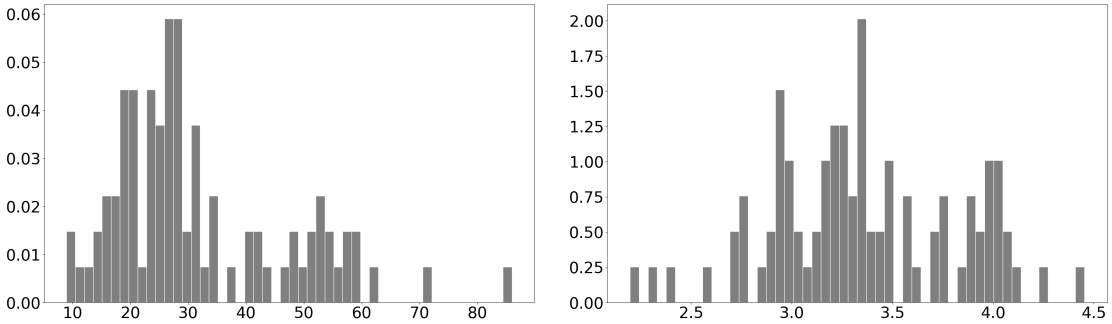
(a) Original (p-value: 3.217e-04) (b) Log (p-value: 0.561)

Figure A.5: Tilt Maze Feature – SCC Count



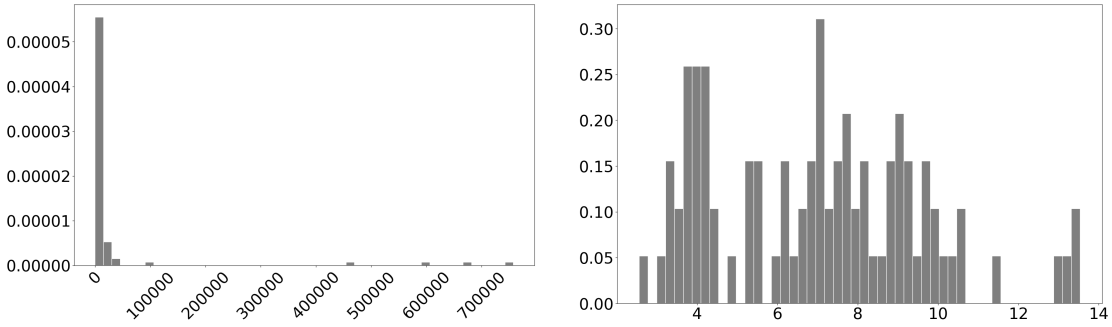
(a) Original (p-value: 0.002) (b) Log (p-value: 0.010)

Figure A.6: Tilt Maze Feature – SCC Checkpoint Count



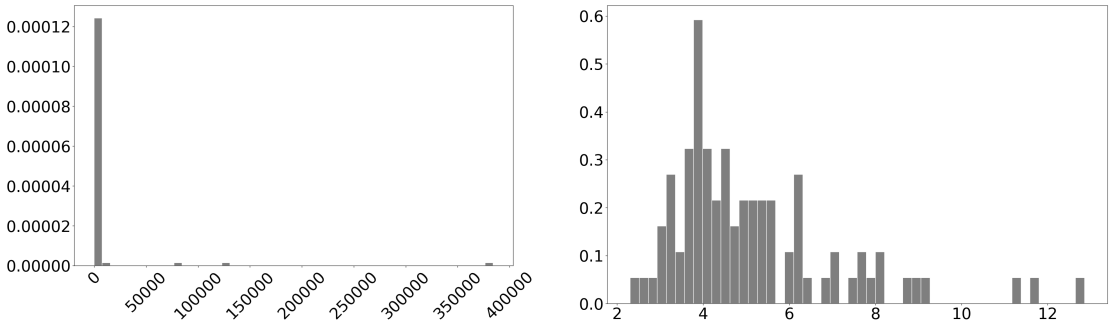
(a) Original (p-value: 0.012) (b) Log (p-value: 0.556)

Figure A.7: Tilt Maze Feature – Shortest Path



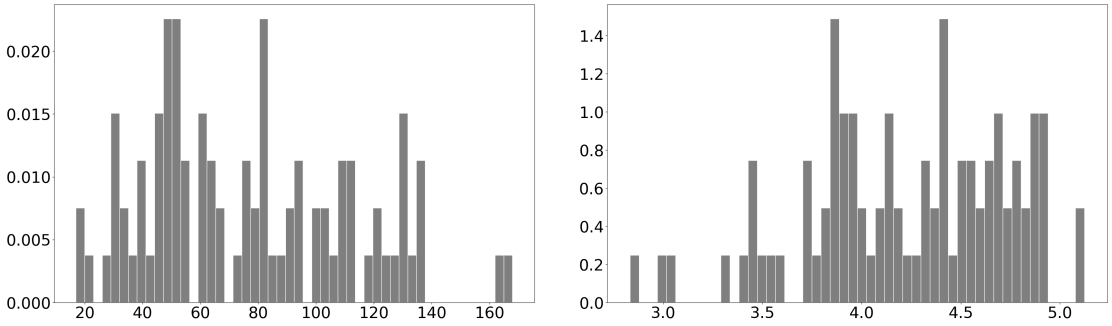
(a) Original (p-value: 8.882e-16) (b) Log (p-value: 0.229)

Figure A.8: Tilt Maze Feature – Reachable States Count



(a) Original (p-value: 0.000e+00) (b) Log (p-value: 0.051)

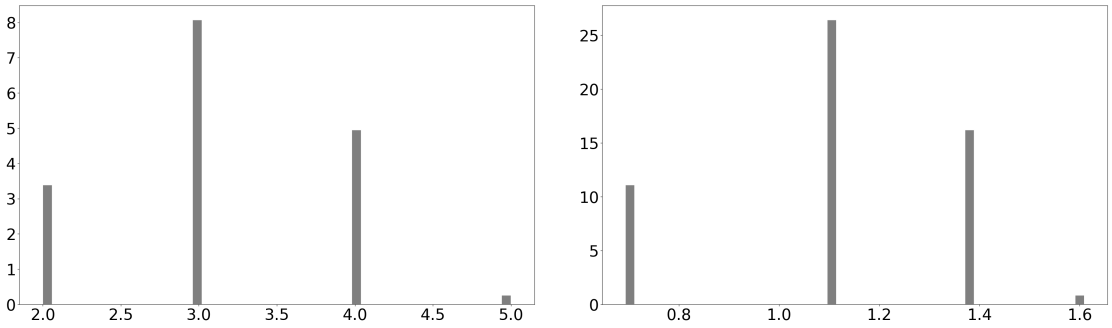
Figure A.9: Tilt Maze Feature – Viable States Count



(a) Original (p-value: 0.214) (b) Log (p-value: 0.570)

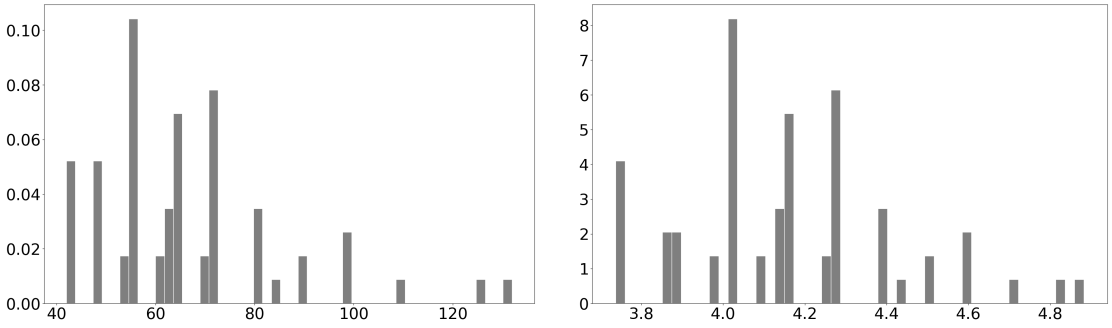
Figure A.10: Tilt Maze Feature – Shortest Path Tiles

A.2 Sokoban - Czech dataset



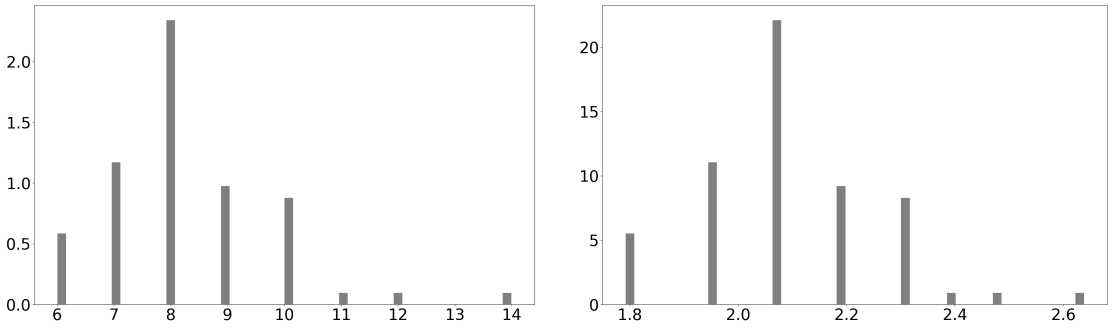
(a) Original (p-value: 3.814e-04) (b) Log (p-value: 6.642e-05)

Figure A.11: CZ Sokoban Feature – Box Count



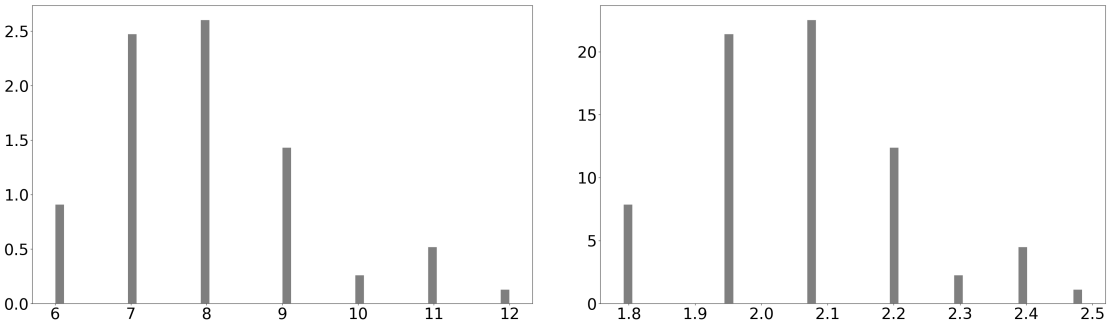
(a) Original (p-value: 0.030) (b) Log (p-value: 0.247)

Figure A.12: CZ Sokoban Feature – Tile Count



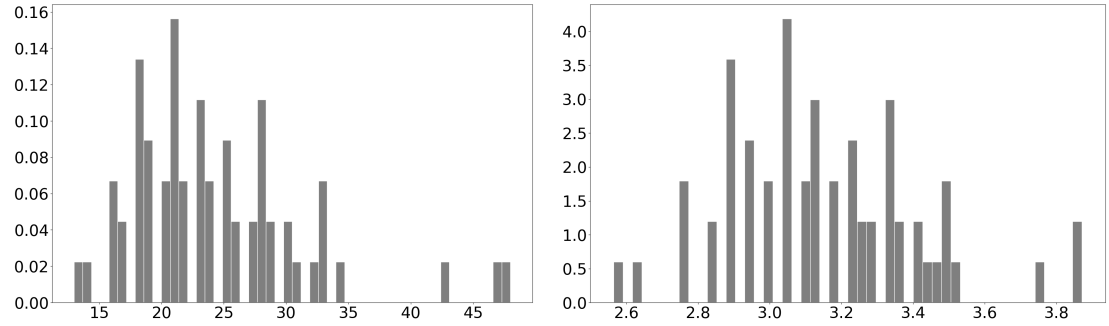
(a) Original (p-value: 0.002) (b) Log (p-value: 0.011)

Figure A.13: CZ Sokoban Feature – Width



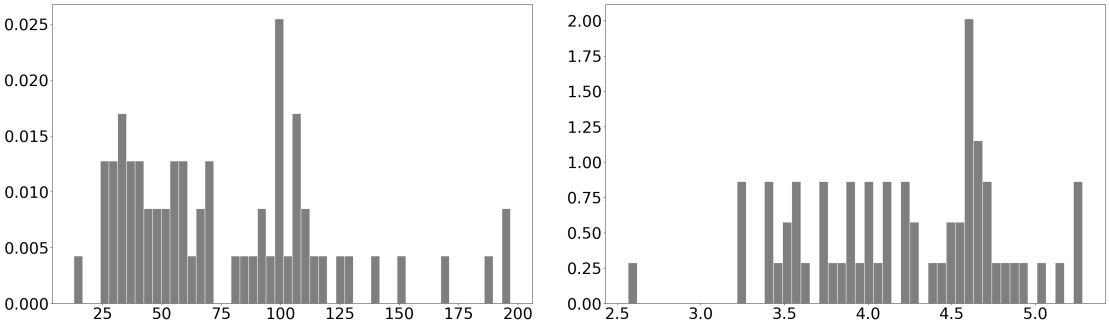
(a) Original (p-value: 0.006) (b) Log (p-value: 0.033)

Figure A.14: CZ Sokoban Feature – Height



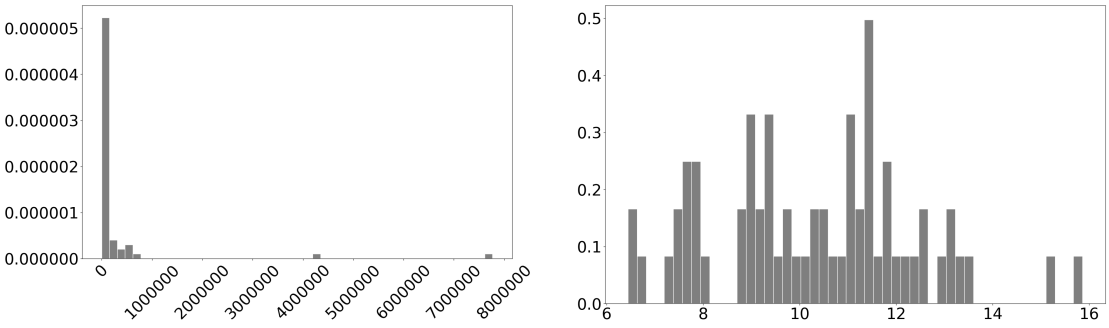
(a) Original (p-value: 0.319) (b) Log (p-value: 0.832)

Figure A.15: CZ Sokoban Feature – Reachable Tile Count



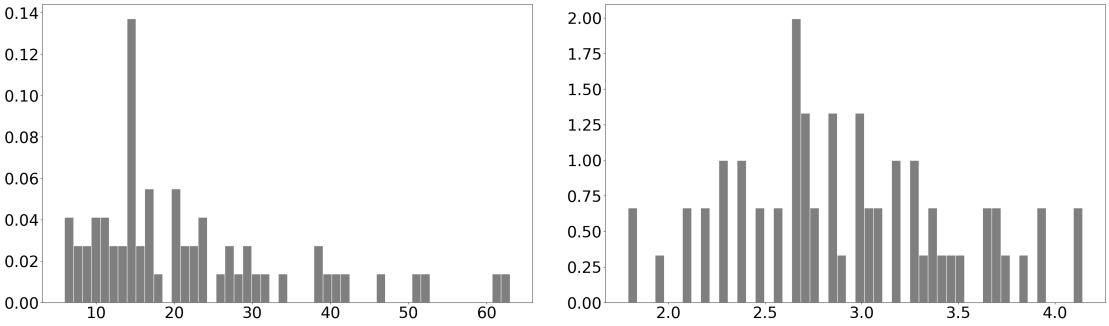
(a) Original (p-value: 0.276) (b) Log (p-value: 0.299)

Figure A.16: CZ Sokoban Feature – Shortest Path



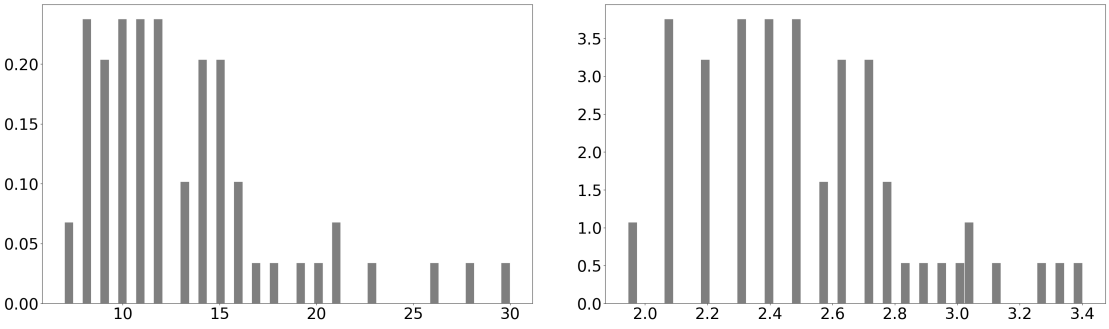
(a) Original (p-value: 1.120e-09) (b) Log (p-value: 0.853)

Figure A.17: CZ Sokoban Feature – Reachable States Count



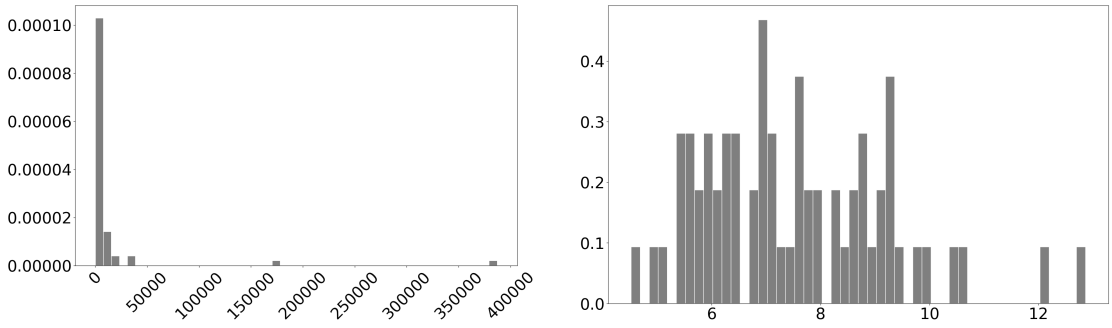
(a) Original (p-value: 0.064) (b) Log (p-value: 0.793)

Figure A.18: CZ Sokoban Feature – Shortest Boxes Path



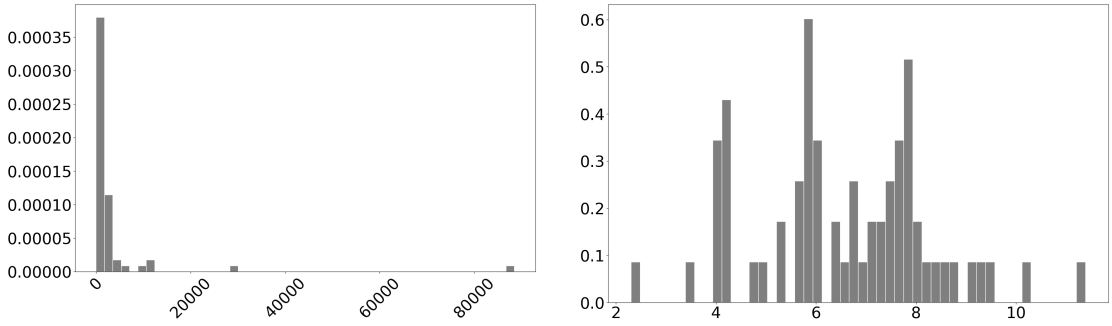
(a) Original (p-value: 0.093) (b) Log (p-value: 0.558)

Figure A.19: CZ Sokoban Feature – Good Boxes Tiles



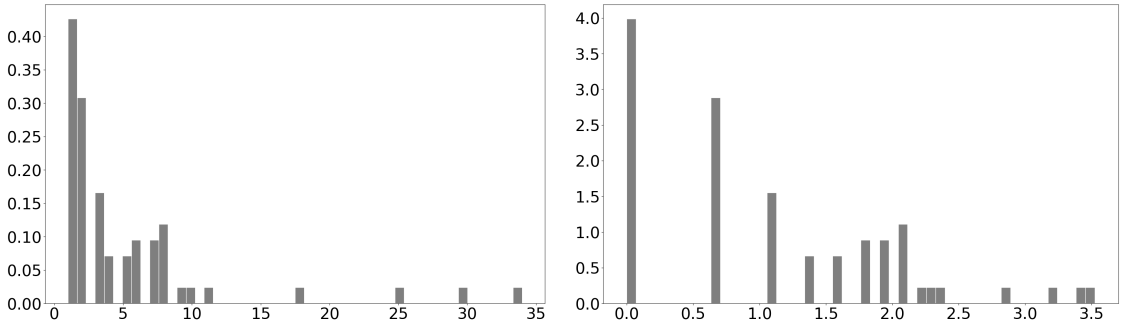
(a) Original (p-value: 1.513e-10) (b) Log (p-value: 0.738)

Figure A.20: CZ Sokoban Feature – Viable States Count



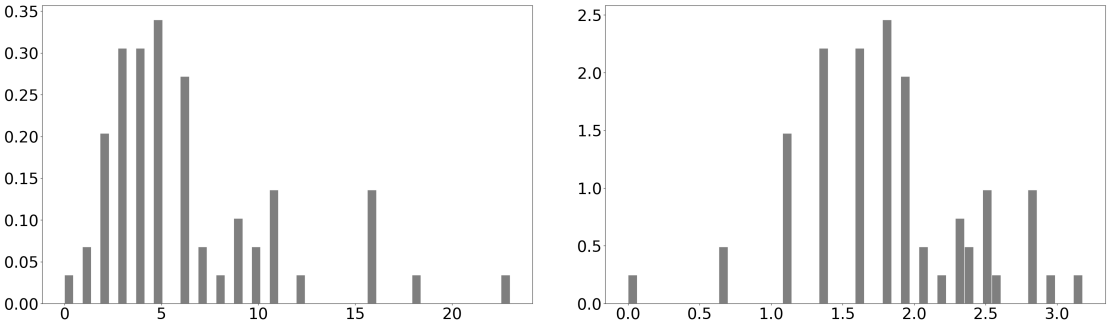
(a) Original (p-value: 7.067e-09) (b) Log (p-value: 0.780)

Figure A.21: CZ Sokoban Feature – SCC Count



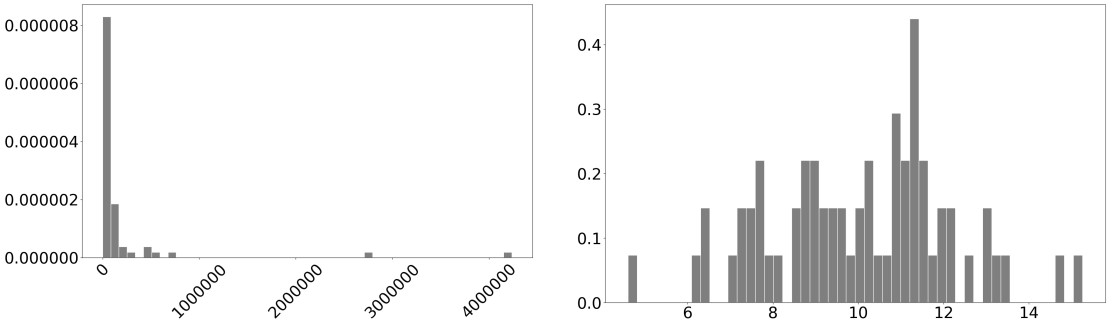
(a) Original (p-value: 1.880e-04) (b) Log (p-value: 0.068)

Figure A.22: CZ Sokoban Feature – SCC Shortest



(a) Original (p-value: 0.002) (b) Log (p-value: 0.307)

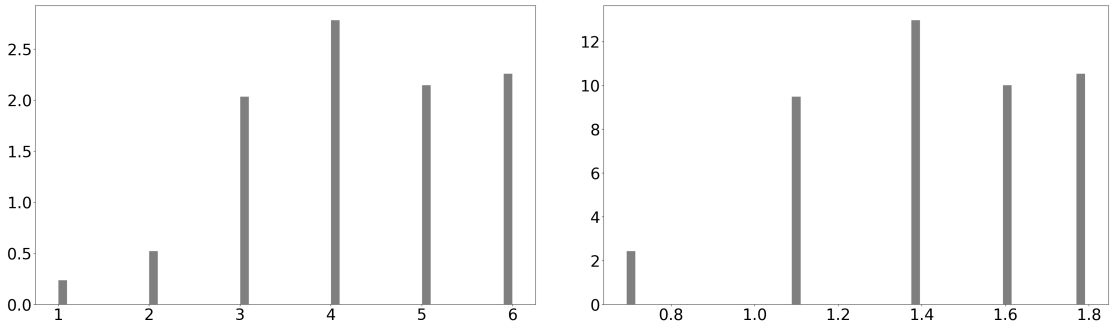
Figure A.23: CZ Sokoban Feature – Counter Intuitive Steps



(a) Original (p-value: 6.351e-09) (b) Log (p-value: 0.709)

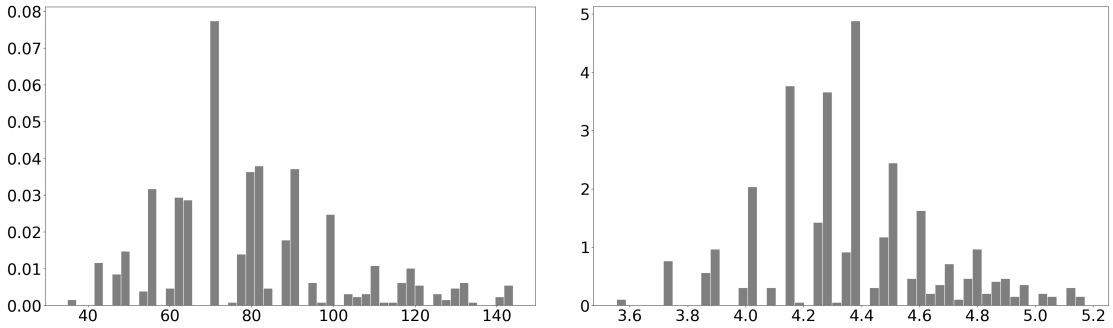
Figure A.24: CZ Sokoban Feature – A* States Count

A.3 Sokoban – Russian dataset



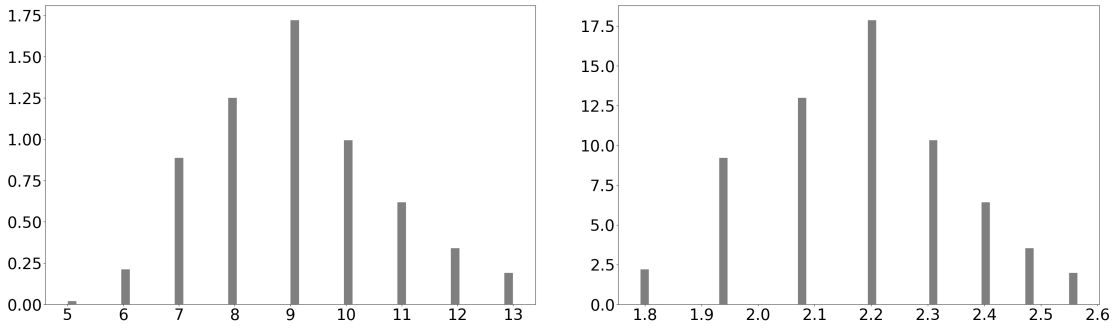
(a) Original (p-value: 6.146e-13) (b) Log (p-value: 0.000e+00)

Figure A.25: RU Sokoban Feature – Box Count



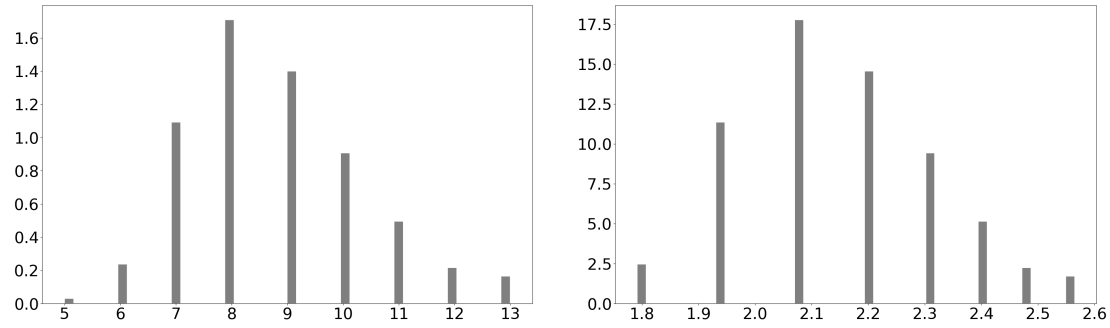
(a) Original (p-value: 1.352e-10) (b) Log (p-value: 8.875e-06)

Figure A.26: RU Sokoban Feature – Tile Count



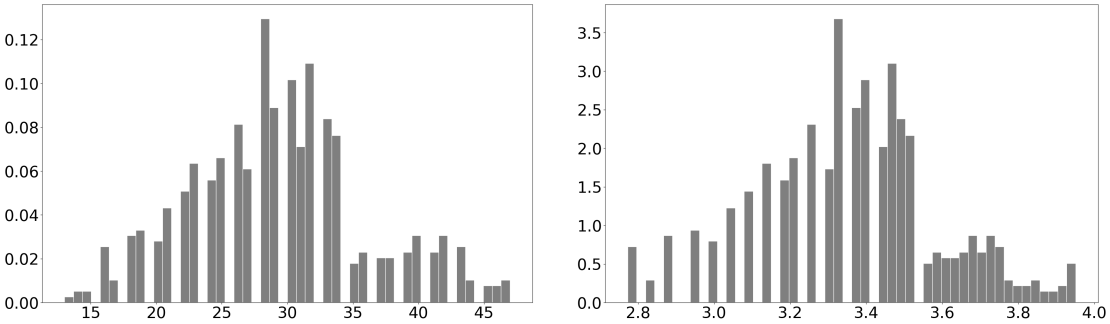
(a) Original (p-value: 1.199e-14) (b) Log (p-value: 7.437e-11)

Figure A.27: RU Sokoban Feature – Width



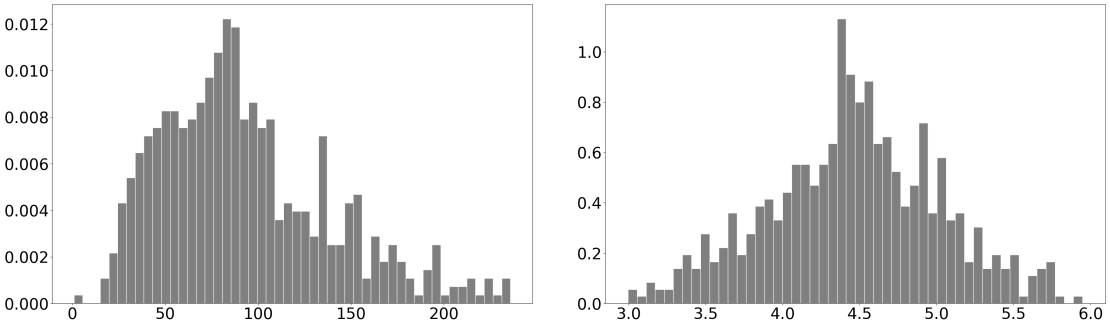
(a) Original (p-value: 2.220e-16) (b) Log (p-value: 1.279e-13)

Figure A.28: RU Sokoban Feature – Height



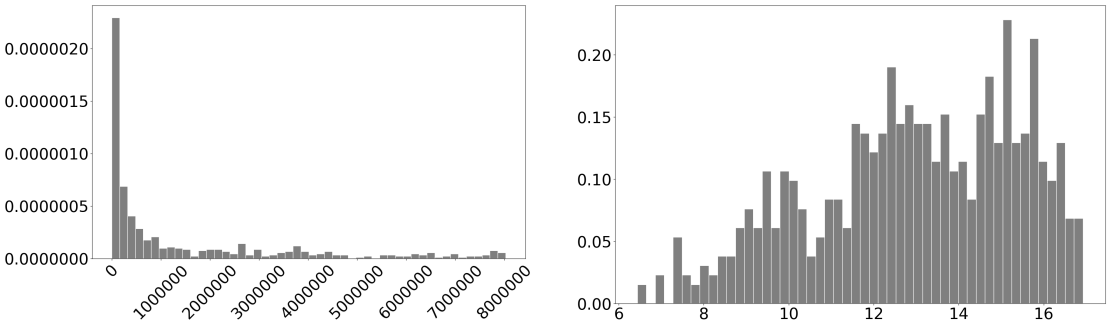
(a) Original (p-value: 0.004) (b) Log (p-value: 0.003)

Figure A.29: RU Sokoban Feature – Reachable Tile Count



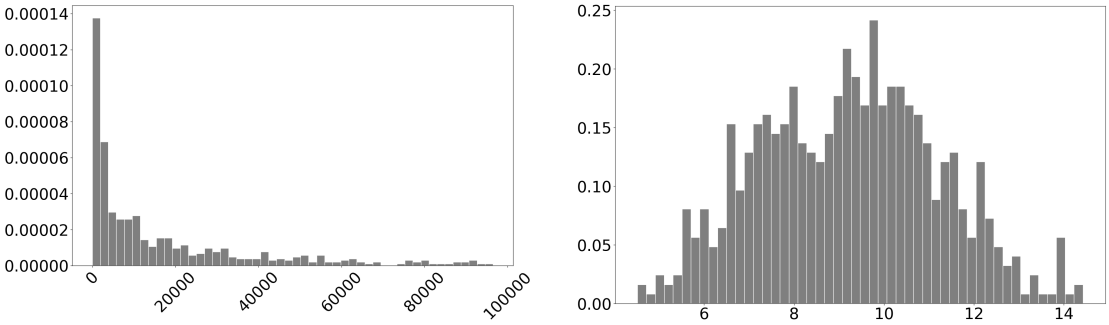
(a) Original (p-value: 1.154e-04) (b) Log (p-value: 0.316)

Figure A.30: RU Sokoban Feature – Shortest Path



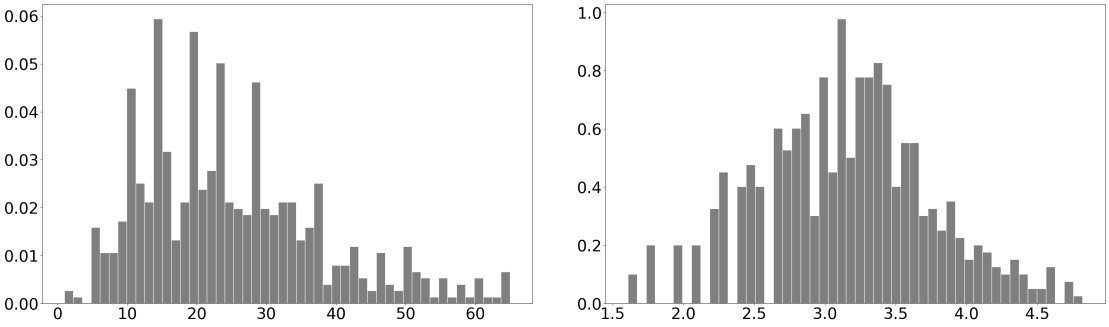
(a) Original (p-value: 0.000e+00) (b) Log (p-value: 0.001)

Figure A.31: RU Sokoban Feature – Reachable States Count



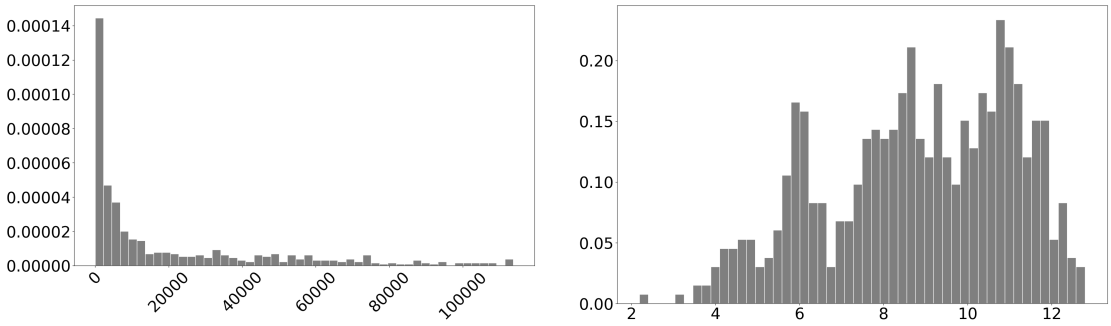
(a) Original (p-value: 0.000e+00) (b) Log (p-value: 0.424)

Figure A.32: RU Sokoban Feature – Viable States Count



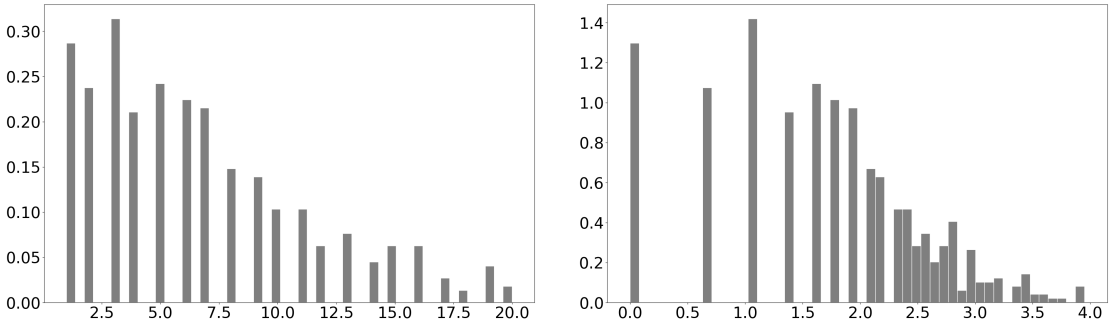
(a) Original (p-value: 3.291e-04) (b) Log (p-value: 0.169)

Figure A.33: RU Sokoban Feature – Shortest Boxes Path



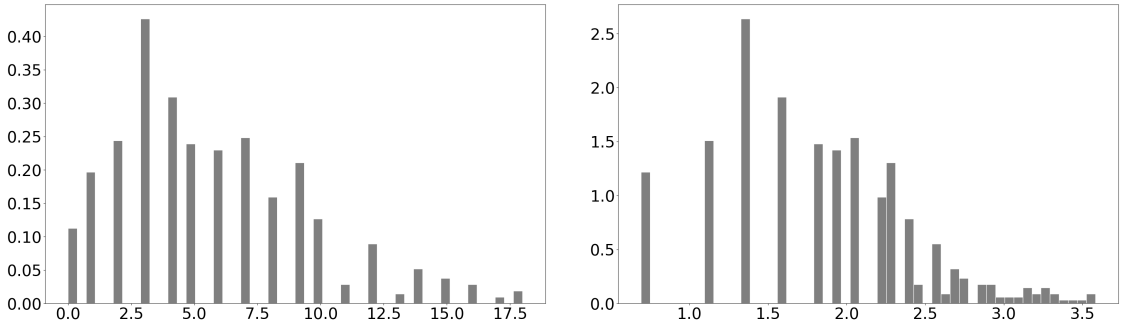
(a) Original (p-value: 0.000e+00) (b) Log (p-value: 0.001)

Figure A.34: RU Sokoban Feature – SCC Count



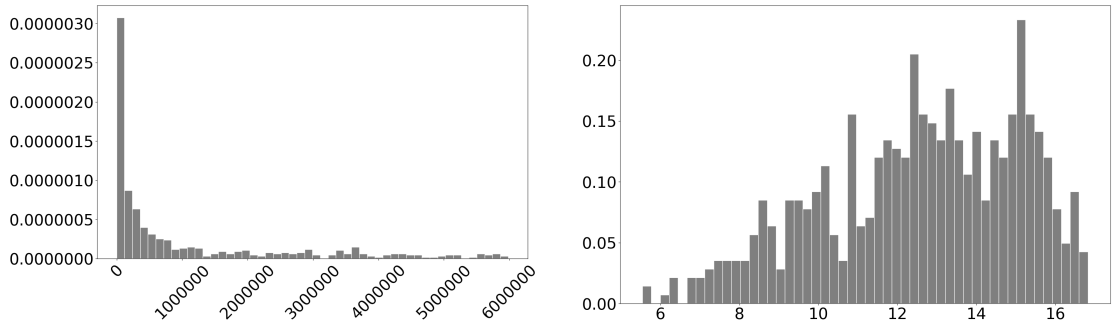
(a) Original (p-value: 1.051e-08) (b) Log (p-value: 3.749e-04)

Figure A.35: RU Sokoban Feature – SCC Shortest



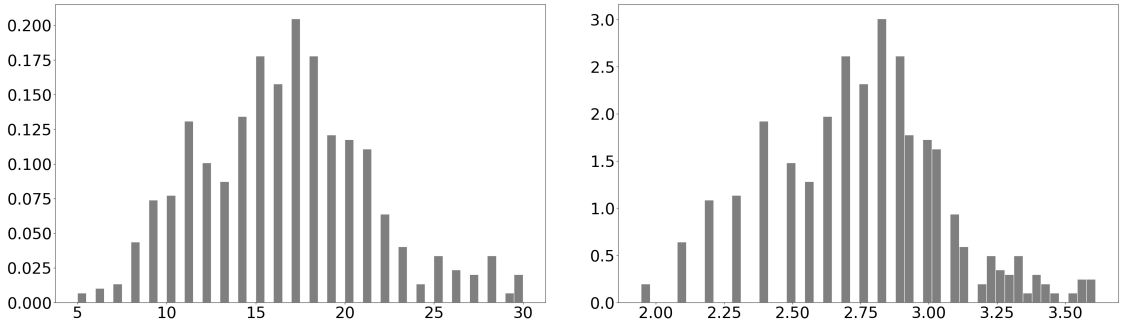
(a) Original (p-value: 1.400e-09) (b) Log (p-value: 3.688e-04)

Figure A.36: RU Sokoban Feature – Counter Intuitive Steps



(a) Original (p-value: 0.000e+00) (b) Log (p-value: 0.014)

Figure A.37: RU Sokoban Feature – A* States Count



(a) Original (p-value: 0.004) (b) Log (p-value: 3.721e-04)

Figure A.38: RU Sokoban Feature – Good Boxes Tiles

Appendix B

Source code

Source code for the diploma thesis, `README` file for instructions and all data are located on GitHub page (`http://github.com/Arasid/DiplomaThesisCode`) and on the attached CD.