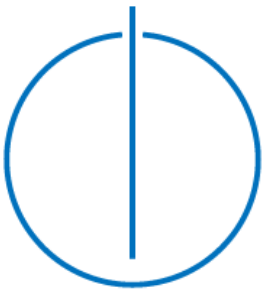
Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Automatic generation of exercises on Turing machines



Christian Backs

Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Automatic generation of exercises on Turing machines

Automatische Aufgabenerzeugung für Turingmaschinen

Author:
Christian Backs

Supervisor:
Prof. Jan Křetínský
Advisors:
M. Sc. Maxi Weininger
M. Sc. Julia Krämer

Submission date:
August 16, 2018

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Contents

Abstract

This thesis presents methods for generating and grading exercises on the topic of Turing machines automatically. Exercises are generated by constructing the abstract version of an existing exercise and then computing its concretizations. This yields a set of new exercises of similar difficulty. The exercise types WhileToTM, WordsInTM and InputOutputMapping are explored. The focus lies on the first type, in which a While-program has to be converted to an equivalent Turing machine. The concepts of extended While-programs and TM-programs are introduced as variants of While-programs and Turing machines that are more suited for problem generation.

1 Introduction

1.1 Motivation

Teaching theoretical computer science to bachelor students heavily relies on exercises to train students in formal language description, generating automata and Turing machines to accept certain languages of finite automata and proving various statements on languages. In this thesis, we explore how to automatically generate exercises on Turing machines for students to practice formal language description and creating Turing machines to accept given languages. The goal is to enable students to train their skills with no additional work for lecturers and tutors. To reach this goal, problem generation and grading are automated. In addition, a web-based implementation allows students to access exercises easily since they are not tied to a fixed exercise schedule or reliant on material resources. Furthermore, different difficulty settings enable them to practice at their own pace.

A special focus in this thesis lies on guaranteeing the quality of the exercises. They should contribute to the students' understanding of the topic and be appropriate for the chosen difficulty level. Heuristic methods have to be employed to maximise the percentage of good exercises while keeping the total number of generated problems high enough to provide a meaningful practice resource.

1.2 Related work

The framework used for implementing the problem generation and grading is "Automata Tutor" [DWW+15], an online tool whose purpose is

> *to help students learn basic concepts in automata theory and to help teachers preparing and grading exercises and problem sets. Automata Tutor currently supports DFA, NFA, NFA to DFA, and regular expression constructions.*[1]

Exercises on product and power set construction as well as formal grammars have also been implemented [Wag17, Hel17]. Problem generation for DFAs has been researched and implemented using Automata Tutor in [Wei14]. That paper inspired the concepts of abstraction and concretization and the use of an SMT-solver in this thesis (see chapter 3.1). The graphical user interface for constructing Turing machines was designed and implemented in [Ajd18].

1.3 Outline

This thesis presents both the theoretical and the practical parts of problem generation for Turing machines. Chapter 2 gives a brief overview of formal grammars and Turing machines and introduces extended WHILE-programs and TM-programs. In chapter 3, the different exercise types and general design concepts of problem generation are presented. There is a section for each exercise type describing theoretical considerations. For the type WHILETOTM, a specific implementation is also presented. The implementation of WHILETOTM exercises has been tested by a group of students. Their feedback is evaluated in chapter 4. That chapter also contains a section on certain trade-offs of the implementation.

2 Preliminaries

This chapter offers a brief overview of some well-known concept from theoretical computer science. A more detailed description can be found in many introductory books, for example [Sch09].

[1] `http://automatatutor.com/`

2.1 General

Convention 2.1. All sets are considered to be finite unless specifically denoted otherwise.

Notation 2.2. For any tuple t with n elements, t_i denotes the ith element of t ($i \in \{1, \ldots, n\}$).

Definition 2.3. The binary operator $\dot{-}$ denotes the *modified difference*, which is defined as

$$\dot{-} : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0, \quad n \dot{-} m := \max(0, n - m)$$

2.2 Directed rooted trees

Directed rooted trees are the basis for several data structures used in this thesis. They are introduced here briefly; a more detailed description can be found in [Die17], for example.

Definition 2.4. A *directed graph* G is a tuple (V, E) where V is the set of nodes and $E \subseteq V \times V$ is the set of edges.

Definition 2.5. A *path* is a non-empty directed graph $P = (V, E)$, where $V = \{v_0, \ldots, v_k\}$, $E = \{(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)\}$ and all the v_i are distinct, except for possibly v_0 and v_k. P is called a path from v_0 to v_k and its *length* is k.

A directed graph $G = (V, E)$ is called *connected* iff for every pair of nodes $v_1, v_2 \in V$, G contains a path from v_1 to v_2. G is called *acyclic* iff for all nodes $v \in V$ there exists no path from v to v of length $l \geqslant 1$.

Definition 2.6. A *directed rooted tree* is a connected acyclic directed graph, in which exactly one node is designated as the *root*.

All directed rooted trees in this thesis are *arborescent*, i.e. their edges are directed away from the root. From now on, directed rooted trees are referred to as trees.

Definition 2.7. Let $T = (V, E)$ be a tree and let $p, c \in V$ be two nodes of T. c is a *child* of p iff $(p, c) \in E$.

2.3 Formal grammars

Definition 2.8.

- An *alphabet* is a finite set, e.g. $\{0, 1\}$.
- A *word* over an alphabet Σ is a finite sequence of symbols in Σ, e.g. 010.
- Σ^* is the set of all words over an alphabet Σ, including the empty word ϵ.
- A *formal language* L over an alphabet Σ is a subset of Σ^*. Languages may be infinite.

Notation 2.9. For a word w of length n, w_i is the ith letter in w ($i \in \{1, \ldots, n\}$).

Definition 2.10. A *formal grammar* is a tuple $G = (V, \Sigma, P, S)$ where

- V is the set of nonterminal symbols.
- Σ is the set of terminal symbols with $\Sigma \cap V = \varnothing$.
- $P \subseteq (V \cup \Sigma)^* V (V \cup \Sigma)^* \times (V \cup \Sigma)^*$ is the set of production rules.
- $S \in V$ is the start symbol.

Convention 2.11. The following conventions are used for formal grammars:

- Upper case letters are nonterminals.
- Lower case letters and symbols are terminals (except for [,] , $\to$, | , ?).
- $\alpha \to \beta$ is written instead of $(\alpha, \beta) \in P$.
- $\alpha \to \beta \,|\, \gamma$ is shorthand for $\alpha \to \beta$, $\alpha \to \gamma$.
- $\alpha \to \beta\gamma?\delta$ is shorthand for $\alpha \to \beta\delta \,|\, \beta\gamma\delta$.
- '[' and ']' are used for grouping in combination with '|' and ' ?'.

A formal grammar G defined as above generates a language over the alphabet Σ by means of its production rules P. A word w over Σ is generated by G iff there exists a sequence of production rules starting with the symbol S, replacing a substring of symbols in each step, and ending at w.

Example 2.12. Let $G = (V, \Sigma, P, S)$ be a grammar with the following production rules:

$$\begin{aligned} S &\rightarrow 0S1 \mid A \\ A &\rightarrow 0 \mid 1 \end{aligned}$$

G generates the language $\{0, 1, 001, 011, 00011, 00111, \dots\}$. For example, the word 00011 is generated by the following sequence of production rules:
$S \rightarrow 0S1 \rightarrow 00S11 \rightarrow 00A11 \rightarrow 00011$.

2.4 Turing machines

Definition 2.13. A *deterministic k-tape Turing machine* is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ where

- $k \in \mathbb{N}$
- Q is a set of *states*.
- Σ is the *input alphabet*.
- $\Gamma \supsetneq \Sigma$ is the *tape alphabet*.
- $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R, N\}^k$ is a partial function called the *transition function*.
- $q_0 \in Q$ is the *initial state*.
- $\square \in \Gamma \setminus \Sigma$ is the *blank symbol*.
- $F \subseteq Q$ is the set of *accepting states*.

All Turing machines in this thesis are deterministic.

A Turing machine is an automaton with a set of tapes that contain a sequence of symbols in Γ. The tapes are infinite in both directions, by default containing the sequence $\square^\omega$.[2] On each tape, there is a *head* at the position of some symbol.

[2] a^ω denotes an infinite sequence of 'a's for some $a \in \Gamma$.

The heads allow the Turing machine to read and write symbols at their current positions. They can also be moved to adjacent positions in both directions.

The transition function δ determines what actions the Turing machine takes in each step. If δ is defined for the current state and the symbols read by the heads, the image of δ defines the target state, which symbol of the tape alphabet Γ is written on each tape, and the directions in which the heads move afterwards. δ may not be defined for some configurations. In this case, the Turing machine terminates. A run of the Turing machine may or may not terminate, and if it terminates it may or may not be accepting, depending on whether the last state is accepting or not.

A Turing machine M can be *run* on an *input* $t = (t_1, \ldots, t_k) \in (\Sigma^*)^k$, which is a tuple of words over Σ. For every $i \in \{1, \ldots, k\}$, the initial content of tape i is $\square^\omega t_i \square^\omega$. The initial position of the ith head is on the first symbol t_i. If M terminates on t, there exists a unique *output* $M(t) = (u_1, \ldots, u_k) \in (\Gamma^*)^k$, such that for every $i \in \{1, \ldots, k\}$, the content of tape i after the run is $\square^\omega u_i \square^\omega$.

This means that M defines a partial function mapping inputs to outputs. Note that the definition of this function is completely independent of M's accepting states. M also defines a language $L(M) \in \Sigma^k$, which is the set of all inputs accepted by M.

Example 2.14. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ be a single-tape Turing machine with $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, \square\}$ and $F = \{q_2\}$. Let δ be the following mapping:

$$
\begin{array}{lcl}
(q_0, 0) & \to & (q_1, 1, R) \\
(q_0, 1) & \to & (q_0, 1, R) \\
(q_1, 0) & \to & (q_1, 1, R) \\
(q_1, 1) & \to & (q_1, 1, R) \\
(q_1, \square) & \to & (q_2, \square, N)
\end{array}
$$

Since M has only 1 tape, inputs are words over Σ. M is a simple Turing machine that scans the input word from left to right, replacing all '0's with '1's. Once the first $\square$ is reached, M terminates. The run is accepting iff at least one '0' was replaced.

Thus, the function defined by M is $f : \{0, 1\}^* \to \{0, 1\}^*, f(w) := 1^{|w|}$. The lan-

guage of M is $L(M) = \{w \mid w \in \{0,1\}^* \wedge \exists i : w_i = 0\} = \{0, 00, 01, 10, 000, 001, 010...\}$

The languages described by Turing machines are the type-0 languages in the Chomsky-hierarchy, which are undecidable in general [Sch09, Chapter 1.1.2]. This means that given a word w and a Turing machine M, there is no algorithm that determines whether $w \in L(M)$ in a finite amount of time. Therefore, there is also no algorithm to determine the language of a Turing machine, or whether the languages of two Turing machines are equal [Sch09, Chapter 1.5].

Finally, it is worth noting that every k-tape Turing machine can be simulated by a single-tape Turing machine [Sch09, Chapter 2.2].

Theorem 2.15. For every Turing machine M, there exists a single-tape Turing machine M' such that $L(M') = L(M)$ and that M and M' define the same function.

2.5 WHILE-programs

WHILE-programs are a class of simple programs, containing only variables, constants, arithmetic expressions and while-loops. Regardless of their simple structure, WHILE-programs are Turing-complete [Sch09, Chapter 2.3].

Definition 2.16. The language of WHILE-programs is generated by the grammar with the following production rules:

$$
\begin{array}{lcl}
\mathrm{S} & \to & \mathrm{S}\ ;\ \mathrm{S} \\
 & | & \mathrm{X} := \mathrm{X} + \mathrm{C} \\
 & | & \mathrm{X} := \mathrm{X} \dot{-} \mathrm{C} \\
 & | & \texttt{while}\ \mathrm{X} \neq 0\ \texttt{do}\ \mathrm{S}\ \texttt{end} \\
\mathrm{X} & \to & \mathtt{x}_1 \mid \mathtt{x}_2 \mid \ldots \\
\mathrm{C} & \to & \mathtt{0} \mid \mathtt{1} \mid \ldots
\end{array}
$$

WHILE-programs use a set of variables $x_{i_1}, \ldots, x_{i_n}$ with $i_1 < i_2 < \cdots < i_n$. A WHILE-program can be *run* on an *input* $t = (t_1, \ldots, t_n)$, which is a tuple containing an initial value for each variable: For all $j \in \{1, \ldots, n\}$, t_j is the initial value of x_{i_j}. If the run of a WHILE-program W on an input t terminates, t can be associated with a unique *output*, $W(t)$. The output is a tuple of the values of the variables after the run. It is unique because WHILE-programs are deterministic.

Example 2.17. The following WHILE-program W uses the variables x_1 and x_2 and computes their sum.

```
while x_2 ≠ 0 do
  x_1 := x_1 + 1
  x_2 := x_2 ∸ 1
end
```

For every input $t = (t_1, t_2)$ the output is $W(t) = (t_1 + t_2, 0)$.

3 Problem generation

In this chapter, I will describe my approach to generating exercises on Turing machines. In chapter 3.1, I will explain my choice of problem types and the general concept used for generating problems. The data structures used for problem generation are defined in chapter 3.2. In chapters 3.3 and 3.4, the specific approaches for the different problem types are presented.

3.1 Design concepts

As described in chapter 2.4, every Turing machine defines both a function and a language. This is reflected by the different problem types considered in this thesis, as each type focuses on either the function or the language of a Turing machine.

Since not only the generation of problems should be automated but also the grading of exercises, the problem types need to be designed such that the solution attempts can be tested for correctness algorithmically. This cannot be done in a mathematically rigorous way, as almost all decision problems are undecidable for Turing machines [Sch09, Chaper 1.5]. Therefore, heuristic methods are needed to determine whether a solution attempt is correct. These methods may fail and judge an attempt incorrectly.

The problem types that I chose are WHILETOTM, WORDSINTM and INPUTOUTPUTMAPPING. They are described here briefly and more precisely in the next chapters.

WHILETOTM is concerned with the interpretation of Turing machines as functions. In an instance of this problem type, a function is given in the form of a

WHILE-program, which has to be converted to a Turing machine. The given program uses a set of variables and contains statements that modify some of them. This has to be realized in the construction of the Turing machine by modifying the tapes corresponding to the variables. To evaluate the solution attempt, a Turing machine equivalent to the given program is constructed algorithmically. Then, both Turing machines are run on a set of inputs and their outputs are compared. The set of inputs is discussed in chapter 4.2.

The problem type WORDSINTM is concerned with the interpretation of Turing machines as languages. As it is impossible to determine the language of a Turing machine in general (see chapter 2.4), the goal in this exercise is to give a subset of the language. The solution attempt is considered correct if all given words are accepted by the Turing machine.

INPUTOUTPUTMAPPING is a variant of WORDSINTM that views the given Turing machine as a function. Rather than determining a set of accepted words, the goal here is to give a set of tuples, each containing an input and the corresponding output of the Turing machine. For evaluation, the Turing machine is run on all given inputs, and the given outputs are compared to the actual outputs.

3.1.1 Abstraction and concretization

The methods for generating problems of all types use the concept of abstraction and concretization, which is based on [Wei14]. This approach takes an existing problem P, called the *base problem*, and creates a set of new problems of similar difficulty. This is realized by the algorithms

1. ABSTRACT, which constructs the *abstraction* A_P of P, and
2. CONCRETIZE, which non-deterministically constructs a new problem from A_P.

The base problem P is represented by a tree structure called an *object tree* (see chapter 3.2). Its abstraction A_P is a unique object tree storing some of the information contained in P. Every possible output of CONCRETIZE is called a *concretization* of A_P.

For each of the different problem types, CONCRETIZE is designed in such a way that the set of concretizations is finite. Therefore, that set can be computed by enumerating all possible combinations of the random choices made by

CONCRETIZE. One way to achieve this is by constructing SMT-constraints[3] that correspond to the random choices and enumerating all models of the constraints with an SMT-solver. For every choice made by CONCRETIZE where a value v is picked from a set S of values, a pair of SMT-constraints is generated for a new, corresponding variable: $1 \leqslant x_v$ and $x_v \leqslant |S|$. Every value that can be assigned to x_v corresponds to one value in S. Therefore, every model that satisfies all of the constraints generated in this way corresponds to one of the possible combinations of random choices, and thus to one of the concretizations.

3.2 Extended WHILE-programs and TM-programs

Standard WHILE-programs and Turing machines are not well suited for automatic problem generation because they only contain relatively low-level information about their functionality. For instance, WHILE-programs do not contain syntactic elements for comparisons other than $\neq$, and Turing machines are overall difficult to parse by humans. In addition, modifying these structures in a controlled way is difficult, especially for Turing machines. Removing a state or changing a transition, for example, can change the automaton drastically.

For these reasons, *extended* WHILE-programs (EWPs) and TM-programs are introduced. They have the same expressive power as the standard versions, but address the issues mentioned above. Both structures are defined as classes of *object trees*. Object trees are tree-like structures that contain additional information like attributes and keys to access children. They make it possible to argue about EWPs and TM-programs in a way similar to an object-oriented programming language. This facilitates the definition of the abstraction and concretization algorithms.

Definition 3.1. An *object node* $v = (t, A)$ is a tuple, where t is its *type* and $A = \{a_1, \ldots, a_n\}$ (for some n) is a set of attributes. Furthermore, every attribute, as well as the node itself, has a *key*.

Definition 3.2. An *object tree* $T = (V, E)$ is a tree of object nodes, where V are its nodes, E are its edges, and for every node $v \in V$ the keys of its children and its attributes are pairwise distinct.

[3]SMT stands for *satisfiability modulo theories*.

Notation 3.3. In an object tree T, the dot operator '.' is a shorthand notation to access a child or an attribute of a node via its key. Given a node v in T and a key k of one of its attributes or children, $v.k$ denotes that attribute or child. The requirement for object trees that the children and attributes of each node have pairwise distinct keys makes this notation safe to use. Furthermore, type(v) denotes the type of v.

3.2.1 Extended WHILE-programs

Extended WHILE-programs (EWPs) are used for the exercise type WHILETOTM described in chapter 3.3. The following grammar gives a concise formulation and intuition of EWPs.

$$
\begin{array}{lcl}
\mathrm{S} & \rightarrow & \mathrm{S}\ ;\ \mathrm{S} \\
 & | & \mathrm{X} := \mathrm{V}\ [\ + \mid \dot{-}\ ]\ \mathrm{V} \\
 & | & \texttt{if}\ \mathrm{C}\ \texttt{then}\ \mathrm{S}\ [\ \texttt{else}\ \mathrm{S}\ ]\ ?\ \texttt{end} \\
 & | & \texttt{while}\ \mathrm{C}\ \texttt{do}\ \mathrm{S}\ \texttt{end} \\
\mathrm{C} & \rightarrow & (\mathrm{X}\ [\ = \mid < \mid \leqslant \mid \neq \mid \geqslant \mid >]\ \mathrm{V}) \\
 & | & (\mathrm{C}\ [\ \wedge \mid \vee\ ]\ \mathrm{C}) \\
\mathrm{X} & \rightarrow & \mathtt{x}_1 \mid \mathtt{x}_2 \mid \ldots \\
\mathrm{V} & \rightarrow & \mathrm{X} \\
 & | & \mathtt{0} \mid \mathtt{1} \mid \ldots
\end{array}
$$

Definition 3.4. An EWP is an object tree $W = (V, E)$ with additional properties. Let $\mathcal{T} = \mathcal{P} \cup \mathcal{C} \cup \mathcal{V}$ be the set of types, where

- $\mathcal{P} = \{$ `Concat`, `Arith`, `If`, `While`$\}$
- $\mathcal{C} = \{$ `Compare`, `Compound`$\}$
- $\mathcal{V} = \{$ `Variable`, `Constant`$\}$

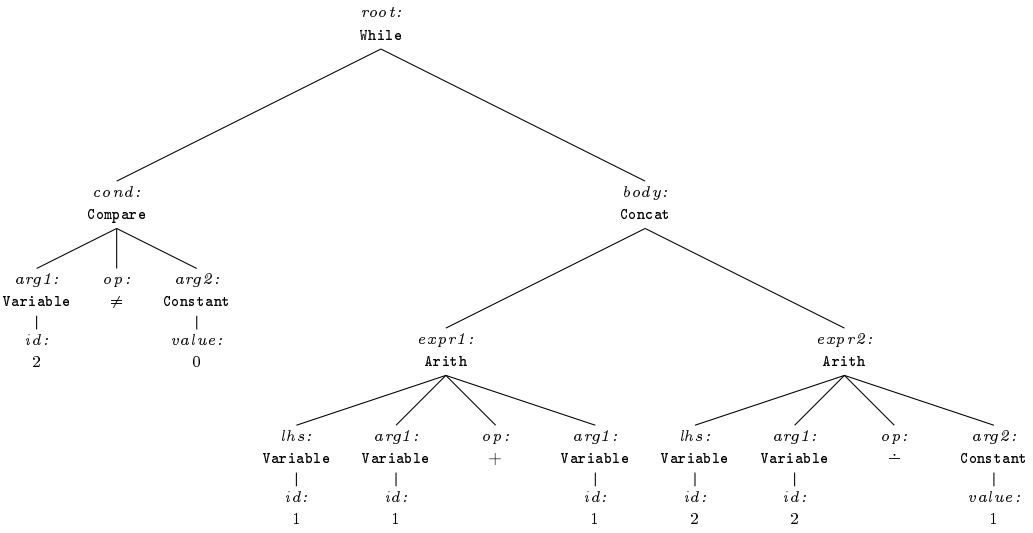
W is an EWP iff all of the following properties hold:

1. $\forall v \in V : \text{type}(v) \in \mathcal{T}$.
2. The root r of W has type(r) $\in \mathcal{P}$.

3. Depending on its type, every node $v \in V$ has exactly those attributes and children that are specified by the following table. Each attribute is denoted by a tuple (k, S), where k is its key and S is the set of possible values. Each child is denoted by a tuple (k, S), where k is its key and S is the set of possible types.

Type	**Attributes**	**Children**
`Concat`:	-	(expr1, $\mathcal{P}$), (expr2, $\mathcal{P}$)
`If`:	-	(cond, $\mathcal{C}$), (then, $\mathcal{P}$), optionally (else, $\mathcal{P}$)
`While`:	-	(cond, $\mathcal{C}$), (body, $\mathcal{P}$)
`Arith`:	(op, $\{+, \dot{-}\}$)	(lhs, {`Variable`}), (arg1, $\mathcal{V}$), (arg2, $\mathcal{V}$)
`Compare`:	(op, $\{= \mid < \mid \leqslant \mid \neq \mid \geqslant \mid >\}$)	(arg1, {`Variable`}), (arg2, $\mathcal{V}$)
`Compound`:	(op, $\{\wedge, \vee\}$)	(cond1, $\mathcal{C}$), (cond1, $\mathcal{C}$)
`Variable`:	(id, $\mathbb{N}$)	-
`Constant`:	(value, $\mathbb{N}_0$)	-

Example 3.5. The following EWP uses the variables x_1 and x_2 and computes $2^{x_2}x_1$.



The semantics of EWPs are as expected and therefore not explicitly defined here. A few notions are introduced informally.

- Like WHILE-programs, EWPs use a set of *variables* $x_{i_1}, \ldots, x_{i_n}$ with $i_1 < i_2 < \cdots < i_n$ that are accessed by the nodes of type Variable. The distinct id's contained in these nodes correspond to the variables.
- The notions of *runs*, *inputs* and *outputs* are the same as for WHILE-programs.
- Every subtree $U \subseteq W$ of an EWP W whose root r has type$(v) \in \mathcal{P} \cup \mathcal{C}$, is called an *expression* of W. Expressions are often referred to in text-form, based on the grammar described earlier. For example, an expression of type `If` may be referred to as `if` C `then` P_1 `else` P_2 `end`.
- Expressions whose type is contained in $\mathcal{P}$ are called *statements*. Expressions whose type is contained in $\mathcal{C}$ are called *conditions*.

Converting EWPs to WHILE-programs

Every EWP with n variables can be converted to an equivalent WHILE-program with $m \geqslant n$ variables. The additional variables are auxiliary and start with the value 0 in all runs.

Definition 3.6. Let W be an EWP using the variables $x_{i_1}, \ldots, x_{i_n}$ for some n with $i_1 < \cdots < i_n$. Let U be a WHILE-program using the variables $x_{i_1}, \ldots, x_{i_n} x_{j_1}, \ldots, x_{j_m}$ for some $m \in \mathbb{N}_0$ with $i_n < j_1 < \cdots < j_m$. U is equivalent to W iff for all inputs $t \in \mathbb{N}_0^n$:

- W terminates on $t \iff U$ terminates on $t' = (t_1, \ldots, t_n, 0, \ldots, 0) \in \mathbb{N}_0^m$.
- W terminates on $t \implies \forall i \in \{1, \ldots, n\} : W(t)_i = U(t')_i$.

Lemma 3.7. The following table introduces some syntactic sugar for standard WHILE-programs. Let $i, j, k \in \mathbb{N}$ be natural numbers that are not necessarily distinct, let $n \in \mathbb{N}_0$ be some constant and let y, z be some variables that are never used elsewhere and hence have the initial value 0.

sugar	**stands for**
$x_i := x_j$	$x_i := x_j + 0$
$x_i := n$	$x_i := y + n$

$x_i := x_j + x_k$	$x_i := x_j$; $y := x_k$;
	`while` $y \neq 0$ `do`
	  $x_i := x_i + 1$; $y := y - 1$
	`end`

$x_i := x_j - x_k$	$x_i := x_j$; $y := x_k$;
	`while` $y \neq 0$ `do`
	  $x_i := x_i - 1$; $y := y - 1$
	`end`

`if` $x = 0$ `then` P_1 `else` P_2 `end`	$y := x$; $z := 1$
	`while` $y \neq 0$ `do`
	  P_2 ; $z := 0$; $y := 0$
	`end`
	`while` $z \neq 0$ `do`
	  P_1 ; $z := 0$
	`end`

Lemma 3.8. Certain expressions of EWPs are syntactic sugar, i.e. they can be replaced by different expressions, as shown in the table below.

original expression	**replacement**

`if` $C_1 \wedge C_2$ `then` P_1 `else` P_2 `end`	`if` C_1 `then`
	  `if` C_2 `then` P_1 `else` P_2 `end`
	`else` P_2 `end`

`if` $C_1 \vee C_2$ `then` P_1 `else` P_2 `end`	`if` C_1 `then` P_1 `else`
	  `if` C_2 `then` P_1 `else` P_2 `end`
	`end`

`while` C `do` P `end`	`if` C `then` $y := 1 + 0$ `else` $y := 0 + 0$;
	`while` $y \neq 0$ `do`
	  P;
	  `if` C `then` $y := 1 + 0$ `else` $y := 0 + 0$
	`end`
	y is some variable that is not used in the original EWP.

`if` C `then` P `end`	`if` C `then` P `else` $x_1 := x_1$ `end`

If an EWP does not contain any syntactic sugar, then in particular it does not contain any expressions of type `Compound`, and all expressions of type `While` have a condition of the form $y \neq 0$.

Theorem 3.9. Let W be an EWP using the variables $x_{i_1}, \ldots, x_{i_n}$ for some n with $i_1 < \cdots < i_n$. There exists a WHILE-program U using the variables $x_{i_1}, \ldots, x_{i_n}$, $x_{j_1}, \ldots, x_{j_m}$ for some $m \in \mathbb{N}_0$ with $i_n < j_1 < \cdots < j_m$, such that U is equivalent to W.

Proof. The first step in the conversion is to replace all expressions in W that are syntactic sugar according to Lemma 3.8. The proof defines a recursive function ϕ that maps every object node that is contained in some EWP (without syntactic sugar) to a substring of some WHILE-program. For every EWP P with root r, $\phi(r)$ is a WHILE-program equivalent to P.

$\phi(v)$ is defined by the table below, where v is an object node in some EWP. Let y be some variable that is used by neither the original EWP nor elsewhere in

the resulting WHILE-program. (y refers to a different variable in each recursive call.)

If type(v) =	**then** $\phi(v) :=$	
`Concat`	$\phi(v.\text{expr1})$; $\phi(v.\text{expr2})$	
`Arith`	$\phi(v.\text{lhs}) := \phi(v.\text{arg1})\ v.\text{op}\ \phi(v.\text{arg2})$	if type(v.arg1) = Variable
	$\phi(v.\text{lhs}) := \phi(v.\text{arg1})$; $\phi(v.\text{lhs}) := \phi(v.\text{lhs})\ v.\text{op}\ \phi(v.\text{arg2})$	if type(v.arg1) = Constant

`If`	$y := \phi(c.\text{arg1}) \dot{-} \phi(c.\text{arg2})$;
	`if` $y = 0$ `then`
	$y := \phi(c.\text{arg2}) \dot{-} \phi(c.\text{arg1})$;
	`if` $y = 0$ `then` $\phi(v.\text{then})$ `else` $\phi(v.\text{else})$ `end`
	`else`
	$\phi(v.\text{else})$
	`end`
		if c.op = "=", where $c = v$.cond

	$y := \phi(c.\text{arg1}) \dot{-} \phi(c.\text{arg2})$;
	`if` $y = 0$ `then`
	$y := \phi(c.\text{arg2}) \dot{-} \phi(c.\text{arg1})$;
	`if` $y = 0$ `then` $\phi(v.\text{else})$ `else` $\phi(v.\text{then})$ `end`
	`else`
	$\phi(v.\text{then})$
	`end`
		if c.op = "$\neq$", where $c = v$.cond

	$y := \phi(c.\text{arg1}) \dot{-} \phi(c.\text{arg2})$; `if` $y = 0$ `then` $\phi(v.\text{then})$ `else` $\phi(v.\text{else})$ `end`	if c.op $\in \{<, \leqslant\}$, where $c = v$.cond
	The cases v.cond.op $\in \{>, \geqslant\}$ are analogue.	
`While`	`while` $\phi(v.\text{cond.arg1}) \neq 0$ `do` $\phi(v.\text{body})$ `end`	
`Variable`	$x_{v.\text{id}}$	
`Constant`	v.value	

Note that the table gives a complete definition of ϕ due to the limitations imposed on EWPs by Lemma 3.8: The conditions of `If` and `While` expressions are never of type `Compound`, and `While` expressions always have conditions of the form $y \neq 0$.

In the case type(v) = `If` with c.op $\in \{=, \neq\}$, the variable y is used to determine

whether $\phi(c.\text{arg1}) = \phi(c.\text{arg2})$. This is equivalent to

$$\phi(c.\text{arg1}) \dot{-} \phi(c.\text{arg2}) = 0 \quad \wedge \quad \phi(c.\text{arg2}) \dot{-} \phi(c.\text{arg1}) = 0,$$

which is checked by two nested `if`-statements.

When ϕ is used to convert W to U, the variables denoted by y must be chosen from the set $\{x_{j_1}, \ldots, x_{j_m}\}$. Furthermore, the table uses expressions that are syntactic sugar defined in Lemmas 3.7 and 3.8, which also make use of variables that do not occur elsewhere in the WHILE-program. These variables must be chosen from the set $\{x_{j_1}, \ldots, x_{j_m}\}$ as well. This is necessary to meet the requirements set by Definition 3.6. m can be chosen as large as necessary without violating the definition of equivalence. To summarize, the conversion of W to U consists of the following steps:

1. Replace expressions in W according to Lemma 3.8.
2. Convert W to a WHILE-program by computing $\phi(r)$ where r is the root of W. $\phi(r)$ may contain syntactic sugar.
3. Replace the syntactic sugar in $\phi(r)$ according to Lemma 3.7. The result is the WHILE-program U.

□

3.2.2 TM-programs

TM-programs are used for the exercise types WORDSINTM and INPUTOUTPUTMAPPING described in chapter 3.4. They can contain expressions similar to basic Turing machine operations (like `Move` or `Write`) as well as more high level expressions like `While` or `If`.

Definition 3.10. Let P be a tuple $(T, \Sigma, \Gamma, \square)$ where $T = (V, E)$ is an object tree, Σ is the input alphabet, $\Gamma \supset \Sigma$ is the tape alphabet and $\square \in \Gamma$ is the blank symbol. Let $\mathcal{T}_{TM} = \mathcal{P}_{TM} \cup \mathcal{C}_{TM} \cup \mathcal{V}_{TM}$ be the set of types, where

- $\mathcal{P}_{TM} = \{$ `Concat`, `If`, `While`, `Terminate`, `Move`, `Write`, `Find`, `End`$\}$
- $\mathcal{C}_{TM} = \{$ `True`, `ReadString`, `ReadChars`, `Find`$\}$

- $\mathcal{V}_{TM} = \{$ `Direction`, `Integer`, `String`, `CharSet`$\}$

P is a TM-program iff all of the following properties hold:

1. $\forall v \in V : \text{type}(v) \in \mathcal{T}_{TM}$.
2. The root r of P has $\text{type}(r) \in \mathcal{P}_{TM}$.
3. Depending on its type, every node $v \in V$ has exactly those attributes and children that are specified by the following table. Each attribute is denoted by a tuple (k, S), where k is its key and S is the set of possible values. Each child is denoted by a tuple (k, S), where k is its key and S is the set of possible types.

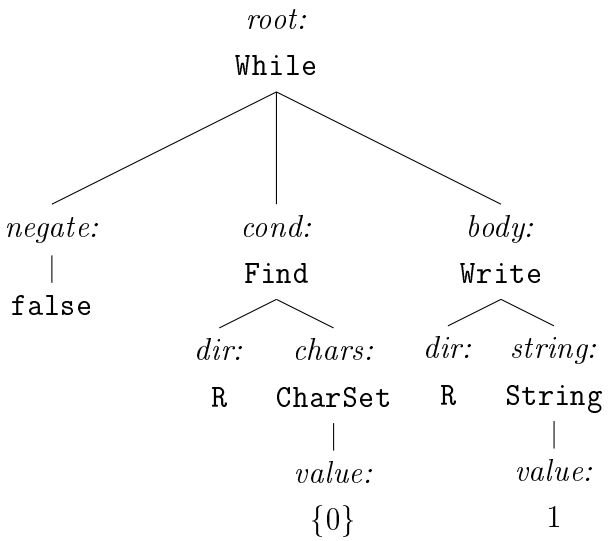
Type	**Attributes**	**Children**
`Concat`:	-	(expr1, $\mathcal{P}_{TM}$), (expr2, $\mathcal{P}_{TM}$)
`If`:	(negate, {`true`, `false`})	(cond, $\mathcal{C}_{TM}$), (then, $\mathcal{P}_{TM}$), optionally (else, $\mathcal{P}_{TM}$)
`While`:	(negate, {`true`, `false`})	(cond, $\mathcal{C}_{TM}$), (body, $\mathcal{P}_{TM}$)
`Terminate`:	(accept, {`true`, `false`})	-
`Move`:	-	(dir, `Direction`), (dist, `Integer`)
`Write`:	-	(dir, `Direction`), (string, `String`)
`Find`:	-	(dir, `Direction`), (chars, `CharSet`)
`True`:	-	-
`ReadString`:	-	(dir, `Direction`), (string, `String`)
`ReadChars`:	-	(chars, `CharSet`)
`Direction`:	(value, {`L`, `R`})	-
`Integer`:	(value, $\mathbb{N}$)	-
`String`:	(value, Γ^+)	-
`CharSet`:	(value, $2^\Gamma \setminus \varnothing$)	-

TM-programs always have a single tape, which functions like the tape of a Turing machine. *Inputs*, *outputs* and runs are defined analogously to Turing machines.

The notions of *expressions*, *statements* and *conditions* are the same as for EWPs. The semantics of the different object node types are described below.

Type of a node v	**Semantics of v**
`Concat`	Concatenation of two TM-programs.
`If`	Conditional statement. If v.negate = `true`, the condition is negated.
`While`	Loop statement. If v.negate = `true`, the condition is negated.
`Terminate`	Terminates the TM-program in an accepting or non-accepting state, indicated by v.accept.
`Move`	Moves the head v.dist positions in the direction v.dir.
`Write`	Writes v.string to the tape in the direction v.dir. Afterwards, the head is at the first position after the string, in the specified direction.
`Find`	Moves the head to the first occurrence of a symbol in v.chars $\cup$ $\{\square\}$ in the direction v.dir. When used as a condition, this returns `true` iff the head is moved to a symbol in v.chars.
`True`	A condition that always returns `true`.
`ReadString`	Returns whether v.string is currently written on the tape in the direction v.dir, starting at the current position of the head. Afterwards, the head is at the first position after the string, in the specified direction, or at the position of the first mismatched character.
`ReadChars`	Returns whether the symbol at the current position of the head is contained in v.chars.

Example 3.11. The following TM-program $P = (T, \Sigma, \Gamma, \square)$ with $\Sigma = \{0, 1\}$ and $\Gamma = \{0, 1, \square\}$ replaces all '0's in the input with '1's.



Every TM-program can be converted to an equivalent Turing machine. The table above indicates how single expressions can be converted. Expressions of type If or While can be converted by concatenating the Turing machines equivalent to their child expressions. Converting the top-level expression of a TM-program according to these rules results in a Turing machine equivalent to the TM-program.

3.3 WHILE-program to Turing machine

In a problem of type WHILETOTM, an EWP W is given and an equivalent Turing machine M has to be constructed. The variables of W have to correspond to the tapes of M. Figure 1 shows the problem description of an example exercise on Automata Tutor. Appendix A contains additional screen shots of the user interface.

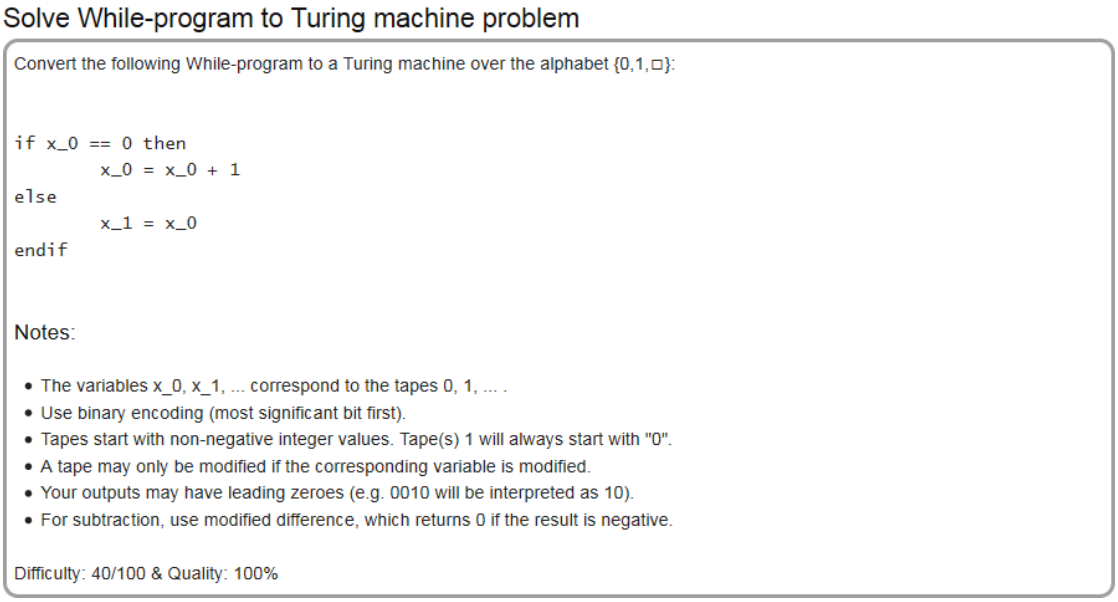


Figure 1: Example WhileToTM exercise on Automata Tutor

3.3.1 Converting Ewps to Turing machines

Every Ewp can be converted to an equivalent Turing machine by first constructing an equivalent While-program and then converting it to a Turing machine. Since the While-program uses additional auxiliary variables, the resulting Turing machine uses additional tapes that correspond to these variables.

Definition 3.12. Let W be an Ewp, n the number of variables used by W, M a Turing machine and Σ its input alphabet. M is equivalent to W iff

- M has $m \geqslant n$ tapes.
- There exists an injective function $e : \mathbb{N}_0 \to \Sigma^*$, such that for all inputs $t \in \mathbb{N}_0^n$:
  - W terminates on $t \iff M$ terminates on t'
  - W terminates on $t \implies \forall i \in \{1, \ldots, n\} : e(T(t))_i = M(t')_i$

  where $t' = (e(t_1), \ldots, e(t_n), e(0), \ldots, e(0)) \in (\Sigma^*)^m$.

Intuitively, e is the encoding M uses to convert natural numbers to words over its input alphabet.

Theorem 3.13. For every EWP there exists an equivalent Turing machine.

Proof. According to Theorem 3.9, for every EWP using n variables there exists an equivalent standard WHILE-program using $m \geqslant n$ variables. For every WHILE-program using m variables there exists an equivalent m-tape Turing machine [Sch09, Chapter 2.3]. □

As a part of this thesis, an algorithm has been implemented that converts EWPs with n variables to n-tape Turing machines, showing that the increase in tapes can be avoided. The algorithm does not convert EWPs to WHILE-programs first, but rather constructs an equivalent Turing machine directly. The EWP is explored recursively in depth-first-search and for each explored node, a Turing machine is constructed that simulates the expression:

- For expressions of type `Arith` and `Compare`, Turing machines are constructed that perform the operations $+, \dot{-}, =, <, \ldots$ bitwise on the tapes corresponding to the variables.
- For expressions of type `Concat`, `If`, `While` and `Compound`, the Turing machines corresponding to their child expressions are concatenated.

3.3.2 Abstraction

The algorithm ABSTRACT_{while} takes an EWP and computes its abstraction, which has the form of an *abstract* EWP. Like EWPs, abstract EWPs are defined as object trees.

Definition 3.14. An abstract EWP is an object tree $W = (V, E)$ with additional properties. Let $\mathcal{T}_\mathcal{A} = \mathcal{P}_\mathcal{A} \cup \mathcal{C}_\mathcal{A} \cup \mathcal{V}$ be the set of types, where

- $\mathcal{P}_\mathcal{A} = \{$ `AConcat`, `AAssign`, `AArith`, `AIncDec`, `AIf`, `AWhile`$\}$
- $\mathcal{C}_\mathcal{A} = \{$ `CSimple`, `CComplex`, `CCompound`$\}$
- $\mathcal{V} = \{$ `Variable`, `Constant`$\}$

W is an abstract EWP iff all of the following properties hold:

1. $\forall v \in V : \text{type}(v) \in \mathcal{T}_\mathcal{A}$
2. The root r of W has $\text{type}(r) \in \mathcal{P}_\mathcal{A}$
3. Depending on its type, every node $v \in V$ has exactly those attributes and children that are specified by the following table. Each attribute is denoted by a tuple (k, S), where k is its key and S is the set of possible values. Each child is denoted by a tuple (k, S), where k is its key and S is the set of possible types.

Type	**Attributes**	**Children**
`AConcat`:	-	(expr1, $\mathcal{P}_\mathcal{A}$), (expr2, $\mathcal{P}_\mathcal{A}$)
`AIf`:	-	(cond, $\mathcal{C}_\mathcal{A}$), (then, $\mathcal{P}_\mathcal{A}$), optionally (else, $\mathcal{P}_\mathcal{A}$)
`AWhile`:	-	(cond, $\mathcal{C}_\mathcal{A}$), (body, $\mathcal{P}_\mathcal{A}$)
`AArith`:	-	(lhs, {`Variable`}), (arg1, $\mathcal{V}$), (arg2, $\mathcal{V}$)
`AAssign`:	-	(lhs, {`Variable`}), (arg1, $\mathcal{V}$), (arg2, $\mathcal{V}$)
`AIncDec`:	-	(lhs, {`Variable`}), (arg1, $\mathcal{V}$), (arg2, $\mathcal{V}$)
`CSimple`:	-	(arg1, {`Variable`}), (arg2, $\mathcal{V}$)
`CComplex`:	-	(arg1, {`Variable`}), (arg2, $\mathcal{V}$)
`CCompound`:	-	(cond1, $\mathcal{C}_\mathcal{A}$), (cond1, $\mathcal{C}_\mathcal{A}$)
`Variable`:	(id, $\mathbb{N}$)	-
`Constant`:	(value, $\mathbb{N}_0$)	-

The algorithm ABSTRACT$_{while}$ takes an EWP W and constructs an abstract EWP U. The core of the algorithm is the recursive function ABSTRACTREC$_{while}$, which creates a copy of W while exploring it via depth-first-search. The types of the nodes in the copy are modified in dependence of the types, attributes and children of the explored nodes. Furthermore, the copy does not contain all attributes of the original.

The algorithm uses the functions ADDATTRIBUTE, ADDCHILD and CHILDREN. ADDATTRIBUTE adds an attribute to a node, with the first argument as its key and the second as its value. ADDCHILD adds a node as a child of another node. CHILDREN returns a list of all children of a node.

Algorithm 1 ABSTRACT$_{while}$

Input: EWP W

```
1:  u := new Node
2:  U := new ObjectTree
3:  U.root := u
4:  abstractRec_while(W.root, u)
5:  return U
6:  function abstractRec_while(Node w, Node u)
7:      u.key := w.key
8:      l := new Type
9:      switch type(w) do
10:         case Concat
11:             l := AConcat
12:         case Arith
13:             if type(w.arg1) = type(w.arg2) then
14:                 l := AArith
15:             else
16:                 if type(w.arg1) = Constant then
17:                     c := w.arg1
18:                     x := w.arg2
19:                 else
20:                     c := w.arg2
21:                     x := w.arg1
22:                 if c.value = 0 then
23:                     l := AAssign
24:                 else if w.lhs.id = x.id ∧ c.value = 1 then
25:                     l := AIncDec
26:                 else
27:                     l := AArith
```

```
28:         case If
29:             l := AIf
30:         case While
31:             l := AWhile
32:         case Compare
33:             if type(w.arg2) = Constant ∧ w.op ∈ {=, ≠} then
34:                 if w.arg2.value = 0 then
35:                     l := CSimple
36:                 else
37:                     l := CComplex
38:             else
39:                 l := CComplex
40:         case Compound
41:             l := CCompound
42:         case Variable
43:             l := Variable
44:             u.AddAttribute(id, w.id)
45:         case Constant
46:             l := Constant
47:             u.AddAttribute(value, w.value)
48:     u.type := l
49:     for each w′ in Children(w) do
50:         u′ := new Node
51:         u.AddChild(u′)
52:         abstractRec_while(w′, u′)
```

3.3.3 Concretization

The counterpart to $\textsc{Abstract}_{while}$ is $\textsc{Concretize}_{while}$. Given an abstract Ewp U, it computes one of its concretizations. It explores U in a pattern similar to depth-first-search: It starts at the root of U and for each explored node, it is called recursively on a subset of its children. This subset is stored in the list L. The algorithm makes non-deterministic choices when determining the values of certain attributes, resulting in a random concretization each time it is executed. For the randomisation of constants, the parameter $p \in \mathbb{N}$ is required. If a constant has the value c in the original Ewp, it has a value $c' \in \{\lfloor \frac{c}{p} \rfloor, \ldots, c\,p\}$ in the concretization.

The algorithm uses the following functions. NumberOfVariables returns the number of distinct variable id's contained in the nodes of an Ewp or abstract Ewp. Random takes a number of arguments as parameters, chooses one of them uniformly at random and returns it. AddConstant adds a child of type `Constant` with the specified key to a node and sets the child's "value"-attribute. AddVariable is the pendant for `Variable` nodes.

Algorithm 2 CONCRETIZE$_{while}$

Input: abstract EWP U, p

```
1:  n := NUMBEROFVARIABLES(U) - 1
2:  W := new ObjectTree
3:  w := new Node
4:  W.root := w
5:  CONCRETIZEREC_while(U.root, w)
6:  return W
7:  function CONCRETIZEREC_while(Node u, Node w)
8:      w.key := u.key
9:      l := new Type
10:     L := new List
11:     switch type(u) do
12:         case AConcat
13:             l := Concat
14:             L := CHILDREN(u)
15:         case AArith
16:             l := Arith
17:             w.ADDATTRIBUTE(op, RANDOM(+, ÷))
18:             L := CHILDREN(u)
19:         case AAssign
20:             l := Arith
21:             w.ADDATTRIBUTE(op, +)
22:             w.ADDCONSTANT(arg2, 0)
23:             L.ADD(u.lhs, u.arg1)
24:         case AIncDec
25:             l := Arith
26:             w.ADDATTRIBUTE(op, RANDOM(+, ÷))
27:             r := RANDOM(0, ..., n)
28:             w.ADDVARIABLE(lhs, r)
29:             w.ADDVARIABLE(arg1, r)
30:             w.ADDCONSTANT(arg2, 1)
```

```
31:         case AIf
32:             l := If
33:             L := Children(u)
34:         case AWhile
35:             l := While
36:             L := Children(u)
37:         case CSimple
38:             l := Compare
39:             w.AddAttribute(op, Random(=, ≠))
40:             w.AddConstant(arg2, 0)
41:             L.Add(u.arg1)
42:         case CComplex
43:             l := Compare
44:             w.AddAttribute(op, Random(=, ≠, <, >, ⩽, ⩾))
45:             L := Children(u)
46:         case CCompound
47:             l := Compound
48:             w.AddAttribute(op, Random(∧, ∨))
49:             L := Children(u)
50:         case Variable
51:             l := Variable
52:             w.AddAttribute(id, Random(1, ..., n))
53:         case Constant
54:             l := Constant
55:             w.value := Random(⌊c/p⌋, ..., cp)
56:     w.type := l
57:     for each u' in L do
58:         w' := new Node
59:         w.AddChild(w')
60:         concretizeRec_while(u', w')
```

3.3.4 Filtering

Given an EWP W, the algorithms ABSTRACT$_{while}$ and CONCRETIZE$_{while}$ provide a way to obtain a set S of EWPs that have a similar structure to W. However, S may contain programs that are ill suited for posing an exercise. This subsection presents heuristic methods for filtering some of these unwanted programs out of the set of concretizations.

Trivial expressions

Some programs in S, when posed as an exercise, may be significantly less difficult than the base program. This can be caused by so called *trivial* expressions. These expressions can be divided into the following categories:

1. $x_i := x_i$. These statements are equivalent to the empty statement.
2. $x_i := x_j - x_j$. These statements are a special case of $x_i := x_j - x_k$ that is far simpler than the general case.
3. Comparisons that are unsatisfiable or tautologies (e.g. $x_i < 0$, $x_i = x_i$).
4. Statements of the form `if` C `then` P_i `else` P_i `end`.
5. Conditions of the form C_i `[` $\wedge$ | $\vee$ `]` C_i.

Trivial expressions can be detected and removed during the execution of CONCRETIZE$_{while}$.

Infinite loops

Even after filtering out trivial statements of category 3, S may still contain programs with infinite loops. An EWP contains an infinite loop if there exists an input on which the program does not terminate. Determining whether a program always terminates is an undecidable problem (known as the halting problem [Sch09, Chapter 2.6]). A heuristic method of determining whether an EWP W contains an infinite loop is to convert it to a Turing machine M and run M on a set of inputs. If any of the runs of M take a number of steps that is higher than a chosen threshold n without terminating, W is considered to contain an infinite

loop. This method does not detect all infinite loops as only a finite set of inputs can be tested. This fault can be compensated by choosing a sufficiently large set of test inputs, which, of course, leads to a longer running time. The method may also falsely identify W as containing an infinite loop if one of the runs of M terminates in $m > n$ steps.

3.3.5 Implementation

The problem generation for WhileToTM problems has been implemented as a part of this thesis. The practical realization is close to the theoretical definitions of Ewps, abstract Ewps and the algorithms Abstract_{while} and $\text{Concretize}_{while}$. The notable differences are described next.

The implementation of Ewps contains the additional type `Not` that allows the negation of conditions. This additional functionality does not increase the expressive power of the implemented structure, as all negations of conditions can be pushed inwards by swapping conjunctions with disjunctions and vice versa as well as relational operators (e.g. changing $=$ to $\neq$).

The set of concretizations is computed with the help of the SMT-solver Z3 [DMB08], as described in chapter 3.1.1. The implementation normalizes the indices of the variables used by the concretizations, which leads to duplicates in the set of concretizations. The normalization works as follows: The first variable that appears in an Ewp is always x_1, the next variable that is distinct to x_1 is always x_2, and so on.

The algorithm Concretize uses the parameter p (see chapter 3.3.3). $p = 2$ was chosen for the implementation in order to allow for some variation in the values of constants while not changing them too drastically.

To evaluate the Turing machine M that is submitted by the user as a solution attempt, the generated Ewp is converted to a Turing machine N and compared to M. The heuristic grading method runs both Turing machines on a set of inputs. If their outputs are equal for each input, the solution attempt is considered correct.

Since M may not terminate for all inputs, a limit is set for the number of steps it can take. This may cause some solution attempts to be rejected, but only if they contain a run that takes an impractically high number of steps since the step limit can be set relatively high (see chapter 4.2).

In the implementation, Turing machines run on inputs that are tuples of binary-encoded non-negative integers. The users' solution attempts also have to use this encoding. The set of test inputs that is used for the evaluation and the "infinite loop"-filter is described and justified in chapter 4.2.

3.4 Words accepted by Turing machine

In a problem of type WordsInTM, a Turing machine M is given and a set of distinct words that are accepted by M has to be determined. This problem type, as well as InputOutputMapping, was not implemented as a part of this thesis. The following sections describe a possible way of designing the necessary algorithms.

3.4.1 Abstraction

For the purpose of generating problems of type WordsInTM, Turing machines are represented by TM-programs (see chapter 3.2.2). The abstraction step can be skipped for this problem type; the concretization algorithm simply takes a TM-program as input. Alternatively, one could design the algorithm similarly to Abstract_{while}, so that it distinguishes between simple and complex expressions of certain types.

3.4.2 Concretization

The algorithm Concretize_{TM} takes a TM-program Q and constructs one of its conretizations P at random. It explores Q entirely and creates a copy P of Q, whose nodes have modified attributes. The following table defines which attributes are modified for the different types of nodes, and how these attributes are modified. Let v be the original value of the attribute.

type	**attribute**	**new value chosen randomly from:**
`If`:	negate	{`true`, `false`}
`While`:	negate	{`true`, `false`}
`Terminate`:	accept	{`true`, `false`}
`Direction`:	value	{`L`, `R`}
`Integer`:	value	$\{\lfloor \frac{v}{p} \rfloor, \ldots, vp\}$
`String`:	value	$\Gamma^{\max(1,\lfloor \frac{l}{q} \rfloor)} \cup \cdots \cup \Gamma^{lq}$, where l is the length of v
`CharSet`:	value	$\{S \in 2^\Gamma \setminus \varnothing \mid \lvert S \rvert \in \{\max(1, \lfloor \frac{\lvert v \rvert}{r} \rfloor), \ldots, \lvert v \rvert r\}$

p, q and r are parameters that have to be chosen for the specific implementation of the algorithm. Since this problem type was not implemented, no concrete choice is given here.

3.4.3 Filtering

Some of the TM-programs generated by CONCRETIZE$_{TM}$ may have undesirable qualities for posing exercises. Heuristic methods can be used to reduce the number of bad exercises by filtering the set of concretizations.

TM-programs may contain infinite loops. They can be detected with a method similar to the one used for WHILETOTM problems (see chapter 3.3.4).

Some of the Turing machines represented by the concretizations may accept very few words, or even none at all. These Turing machines result in bad exercises because the users should be able to submit a certain number of accepted words. One way to check if a Turing machine M accepts a sufficient number of words is to run M on a set of inputs and counting how many of them are accepted. Naturally, there may be words that are accepted by M that are not contained in the set of inputs, so the choice of these inputs is particularly important. One approach that seems sensible is to take the n shortest words over the input alphabet Σ of M as the user can be expected to find these words for his solution attempt.

3.4.4 Evaluation

Solution attempts to WordsInTM-problems are checked by running the given Turing machine M (which is constructed from the generated TM-program) on all of the words submitted by the user. If all of them are accepted by M, the solution is considered correct. Since M may not always terminate, a step limit n has to be chosen. If some run of M takes more than n steps, that run is aborted and considered non-accepting.

3.4.5 Input-Output-Mapping

The problem type InputOutputMapping is a variant of WordsInTM, where a Turing machine M is given and pairs of corresponding inputs and outputs of M have to be determined. The problem generation works just like for WordsInTM.

The correctness of solution attempts is determined by running M on all of the specified inputs: Let $S = \{(I_1, O_1), \ldots, (I_n, O_n)\}$ be the set of submitted tuples. The solution is considered correct if $\forall i \in \{1, \ldots, n\} : M(I_i) = O_n$. Again, the runs of M must have a step limit.

The filter used for detecting infinite loops in WordsInTM exercises can also be used for this problem type.

4 Experimental evaluation

4.1 Survey

As a part of this thesis, a survey about the exercise type WhileToTM and the problem generation for this type was conducted among students taking an introductory course in theoretical computer science. The survey had the following topics:

- Is the difficulty of the generated exercises appropriate for the difficulty selected by the students?
- Do the exercises help students to understand Turing machines?
- Is the user interface usable?

The form for taking part in the survey is included in appendix B. Unfortunately, participation was too low to make any statistically representative statements. However, the results are summarized here qualitatively.

Overall, the difficulty of the exercises was experienced as appropriate by the students, with the exception that some exercises were trivially simple in spite of a high difficulty setting. This was caused by a mistake in the implementation of the filter for infinite loops. All participants stated that they learned something about Turing machines, specifically about the conversion of EWPs to Turing machines. Some parts of the user interface were rated usable or intuitive by most participants, such as the interface for problem generation and the problem description. Other parts, most notably the interface for constructing Turing machines was criticized by most participants for being difficult to use. There were also problems with the grading method, as some correct solutions did not receive maximal points.

The feedback suggests that problem generation and grading are working as intended, save for a few technical problems that can be easily solved. The main point of criticism was a part of the user interface whose design was not a part of this thesis.

4.2 Input sets for WhileToTM exercises

The methods used for grading WhileToTM exercises and filtering out EWPs with infinite loops require a set of inputs for Turing machines. These inputs need to be tuples of non-negative integers in binary encoding. In the following, a possible choice for this set of inputs, as well as the step limit used by the grading method, is given and justified.

For the grading method, an input set is needed that detects errors in solution attempts. It seems unlikely that errors can only be detected on inputs consisting of long words. Therefore, a reasonable set of inputs would be the combinations of all words up to a certain length. Some combinations should be omitted: EWPs can contain variables that never appear on the right-hand side of any expression. Therefore, the input for the tape corresponding to such a variable is irrelevant and should be set to "0" for all inputs. Such a tape is called *fixed*. This ensures that in the user's solution attempt, the content of the tape does not have to be wiped first, before it is written to.

The running time of the grading method depends on the size of the input set, which in turn depends on the maximal length of the words l that make up the inputs, as well as the number of non-fixed tapes k'. Let I_l be the set of all inputs that consist of words up to length l. Since all words are binary encoded non-negative integers, the first letter is always '1', except for the word "0". Therefore it holds

$$|I_l| = (\sum_{i=1}^{l} |\{\text{words of length } i\}|)^{k'} = (2 + \sum_{i=2}^{l} 2^{i-1})^{k'} = (2 + \sum_{i=1}^{l-1} 2^i)^{k'}.$$

An experiment was performed to examine which input set $|I_i|$ is feasible for the grading method. An EWP was picked and converted to a Turing machine M that was the correct solution. n distinct Turing machines were generated randomly to form a set A of solution attempts. For each $i \in \{1, \ldots, i_{max}\}$ and for each Turing machine $N \in A$, the time t_i required to run M and N on all inputs in I_i and compare their outputs was measured. Additionally, for each $N \in A$, the value i_N was determined, which is the smallest i such that I_i contains an input that detects an error in N.

The solution attempts $N \in A$ were generated according to the following rules.

1. N has k tapes, where k is the number of tapes of M.
2. The number of states in N is picked uniformly at random from $\{1, \ldots, 2|Q_M|\}$, where Q_M is the number of states in M.
3. One of N's states is marked initial.
4. For each state q, its number of outgoing transitions is picked uniformly at random from $\{0, \ldots, |\Gamma|^k = 3^k\}$. That number of tuples is picked from Γ^k. For each picked tuple r, a transition is added from q to a randomly picked target state, with r as its read condition. The transition performs randomly picked move and write actions.

The experiment was executed three times with different EWPs as the correct solution. In each experiment, 100 distinct solution attempts were randomly generated. Figures 2 - 4 show the times t_i averaged over all 100 Turing machines in A. The correct solutions and the value i_{max} were chosen as follows:

(a) $i_{max} = 17$, `if` $x_0 = 0$ `then` $x_0 := x_0 + 1$ `else` $x_1 := x_0$ `end`

(b) $i_{max} = 9$, `if` $x_0 = 0 \wedge x_1 = 0$ `then` $x_0 := x_0 + 1$ `else` $x_1 := x_1 + 1$

(c) $i_{max} = 9$, $x_0 := x_0 + x_1$

In every experiment, for every attempt in A, an error was detected on some input in I_1. This may not be the case for real solution attempts submitted by students, so a larger set of inputs should be used. The graphs indicate that, in terms of running time, I_{16} may be feasible for exercise (a), but for more complex exercises, I_8 is likely the largest viable set. For exercises (b) the grading method took about 800 ms on average, and about 1900 ms for (c). If even more complex exercises are to be supported, especially with a higher number of non-fixed tapes, a significantly smaller input set may have to be selected.

The method used for filtering out EWPs containing infinite loops also requires an input set. This method may have to be executed multiple times when generating an exercise because some of the generated EWPs may not meet the requirements set by the active filters. Therefore, a smaller set than for the grading method should be chosen, so that the running time is short enough. I_7 is probably the largest viable set.

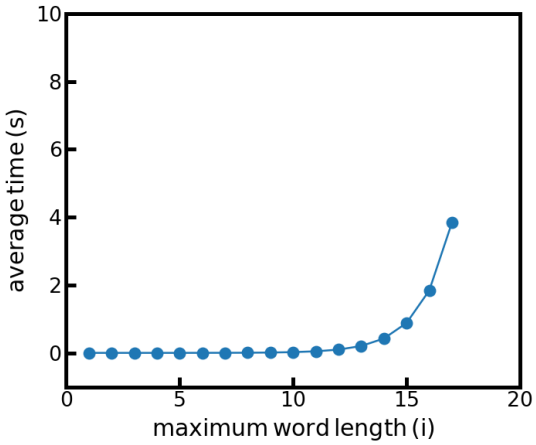


Figure 2: Running time graph for exercise (a)

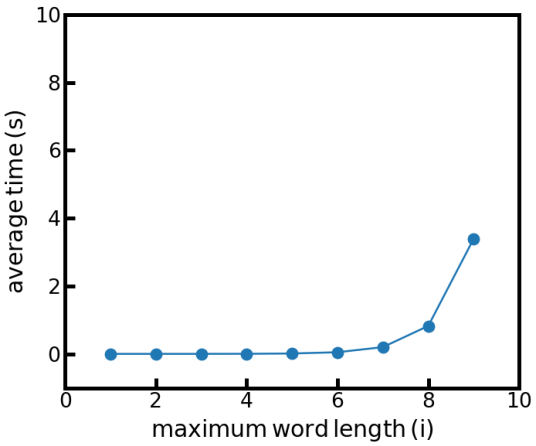


Figure 3: Running time graph for exercise (b)

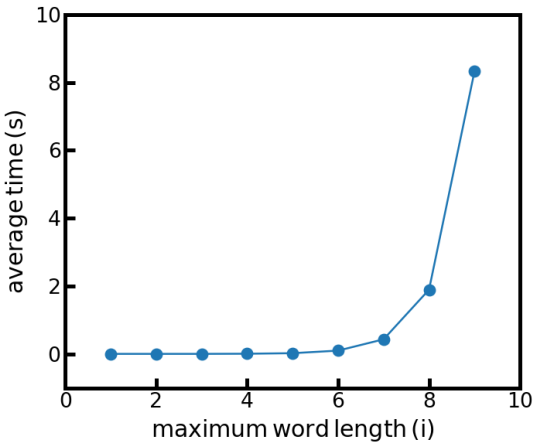


Figure 4: Running time graph for exercise (c)

Step limit

In order to find a suitable step limit for the grading method, the number of steps taken by the correct Turing machines used in the experiments was measured. With the input set I_9, the maximal number of steps taken for any word was 30, 56 and 58, respective to the programs (a), (b) and (c). Correct solution attempts should not take significantly more steps than this, so a step limit of 1000 should be sufficient in order not to falsely reject any of them. This value is also feasible in terms of running time, because firstly, performing 1000 steps of a Turing machine is not particularly expensive, and secondly, the number of runs that reach the step limit should be very low in most cases.

5 Conclusion and outlook

This thesis has established three exercise types on the topic of Turing machines and has presented methods for generating and grading problems of these types automatically. For this purpose, EWPs and TM-programs have been defined as a more suited way to describe WHILE-programs and Turing machines for the algorithms of problem generation. The approach for these algorithms, which consists

of abstraction, concretization and the use of an SMT-solver, has been adapted from DFAs [Wei14] to suit Turing machines. A set of filters has been designed for each exercise type in order to increase the quality of the generated exercises. Finally, problem generation has been implemented for one of the exercise types, WhileToTM, in the tool "Automata Tutor".

The remaining two exercise types, WordsInTM and InputOutputMapping, have yet to be implemented. This will include testing the viability of the design of the algorithms and filters suggested in this thesis. One type of filter, that was used for DFAs in [Wei14], but was not explored here, is that of *constraint modes*. They provide additional control over the randomization of variables in the generated Ewps and TM-programs. It is worth investigating, if this can be used to increase the quality of the exercises.

Appendices

A Automata Tutor screen shots

This appendix contains some screen shots of a WhileToTM exercise on Automata Tutor. The graphical user interface for creating Turing machines was designed in [Ajd18].

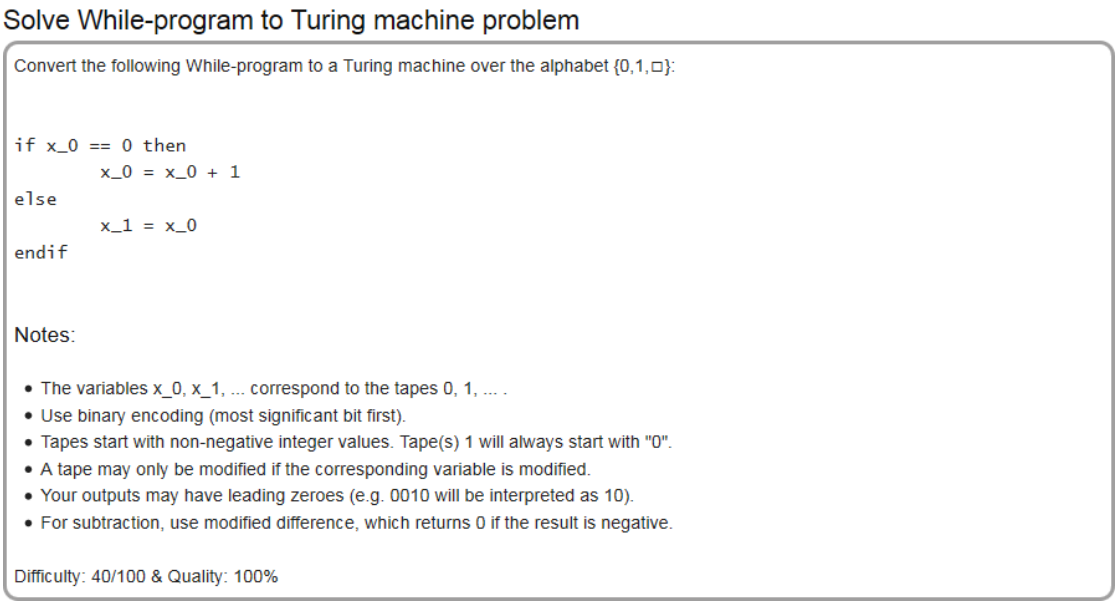


Figure 5: Problem description

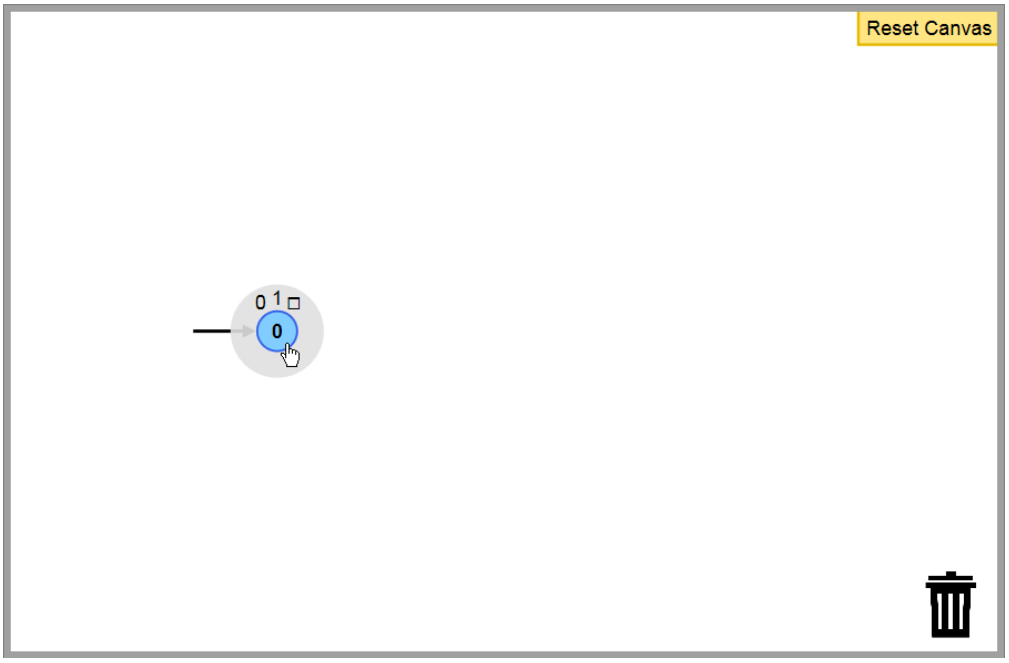


Figure 6: Hover menu for creating transitions

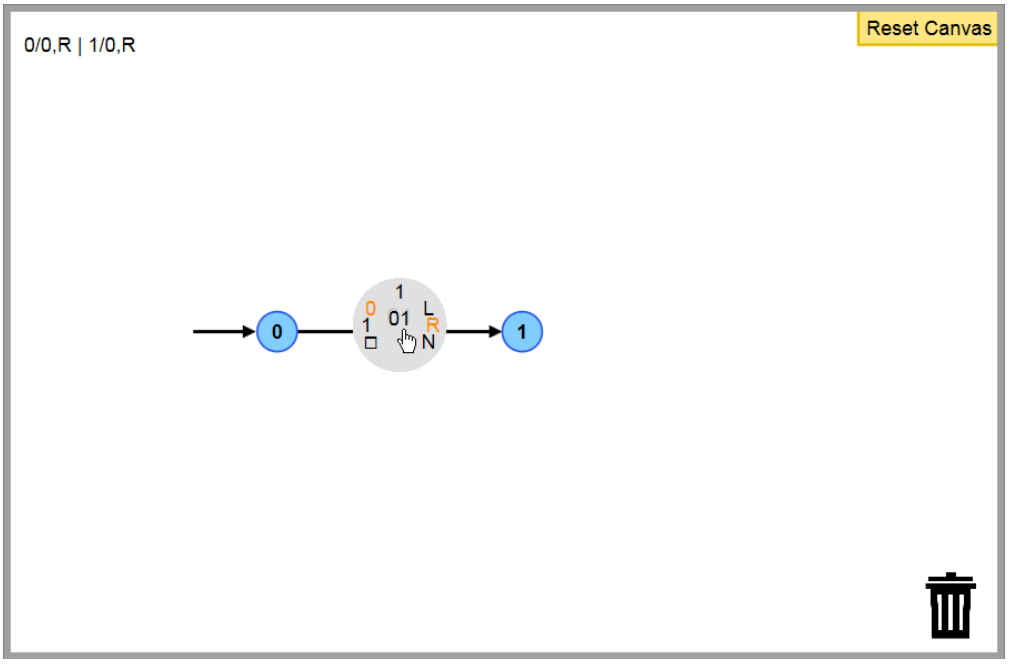


Figure 7: Hover menu for modifying transitions

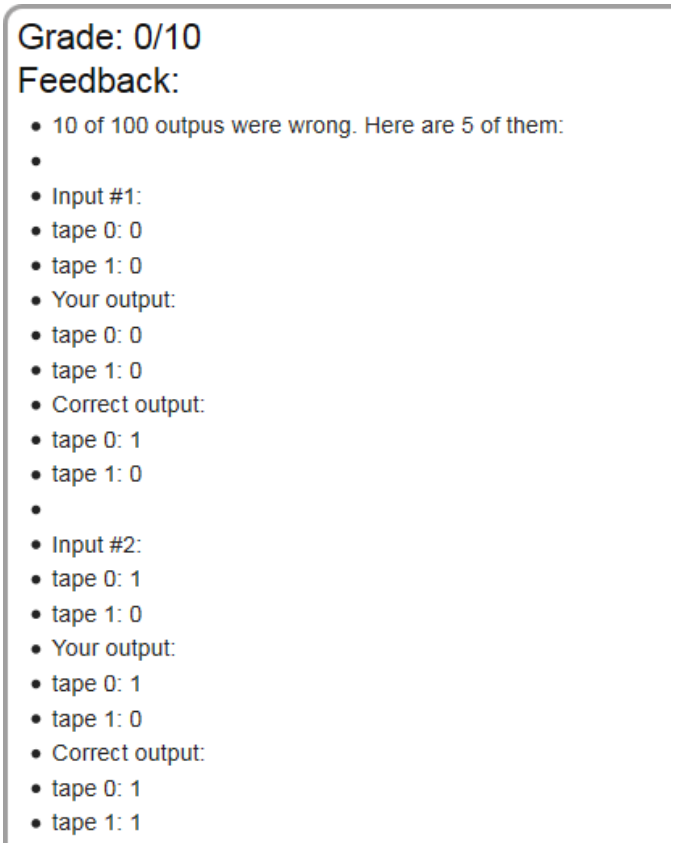


Figure 8: Part of the feedback on an incorrect solution

B Survey

This appendix contains the form for taking part in the survey discussed in 4.1.

Prof. J. Křetínský
Christian Backs

Technische Universität München

Chair for Foundations of Software Reliability and Theoretical Computer Science

Bachelor's Thesis:
Automatic generation of exercises on Turing machines

Turing machine problem generation

This is a survey about the problem type "While to TM" on Automata Tutor. To take part, please solve one premade and one randomly generated exercise.

Premade exercise

You can find the premade exercise under "Practice Problem Sets". Please keep track of how many attempts and how much time it takes you to solve the exercise.

Afterwards, fill out this form:

Number of attempts:	_____
Time in minutes:	_____
Final grade:	_____
Difficulty denoted in the problem description:	_____

Please describe if you found the difficulty of the exercise appropriate for the difficulty denoted in the problem decription.

	too easy	easy	just right	difficult	too difficult
The exercise was...	☐	☐	☐	☐	☐

Generated exercise

To generate an exercise, go to "Exercise", select the problem type "While to TM" and click "generate". Optionally, you can specify a difficulty range by selecting "best in difficulty range" and entering a lower and an upper bound.
Please keep track of how many attempts and how much time it takes you to solve the exercise.

Afterwards, fill out this form:

Number of attempts:	_____
Time in minutes:	_____
Final grade:	_____
Difficulty denoted in the problem description:	_____

Please describe if you found the difficulty of the exercise appropriate for the difficulty denoted in the problem decription.

	too easy	easy	just right	difficult	too difficult
The exercise was...	☐	☐	☐	☐	☐

Understanding

Please describe your understanding of the topic by checking one option for each of the following statements.

	Before the exercises	After the exercises	Not yet
I understand how the variables of a while-program can be encoded in a Turing machine.	☐	☐	☐
I know how to perform arithmetic operations on the tapes of a Turing machine.	☐	☐	☐
I know how to realize expressions of the form x == 0 in a Turing machine.	☐	☐	☐
I know how to realize expressions of the form x < y in a Turing machine.	☐	☐	☐
I know how to realize if and while statements in a Turing machine.	☐	☐	☐

	Yes	No
I have learned something from the exercises.	☐	☐

User interface

Please rate the usabilty of the user interface and its components.

	Unusable	Hindering	Usable	Intuitive
Problem generation	☐	☐	☐	☐
Problem description	☐	☐	☐	☐
Turing machine construction	☐	☐	☐	☐
Tape simulator	☐	☐	☐	☐
Feedback	☐	☐	☐	☐
Overall	☐	☐	☐	☐

Comments

Here you can give any feedback that was not covered by the questions above. (what you liked / disliked, what could be improved etc.)

Thank you for taking part in this survey.

References

[Ajd18] Tohid Ebrahim Ajdari. Extending the tool AutomataTutor with Turing machines. Bachelor's thesis, Technical University of Munich, 2018.

[Die17] Reinhard Diestel. *Graph theory (Graduate texts in mathematics)*. Springer Berlin, 2017.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DWW+15] Loris D'Antoni, Matthew Weavery, Alexander Weinert, Rajeev Alur, et al. Automata Tutor and what we learned from building an online teaching tool. *Bulletin of EATCS*, 3(117), 2015.

[Hel17] Martin Helfrich. Kontextfreie Grammatiken in AutomataTutor. Bachelor's thesis, Technical University of Munich, 2017.

[Sch09] Uwe Schöning. *Theoretische Informatik - kurz gefasst*. Spektrum Akademischer Verlag Heidelberg, 2009.

[Wag17] Jan Wagener. Lehren von Algorithmen für reguläre Sprachen mithilfe von AutomataTutor. Bachelor's thesis, Technical University of Munich, 2017.

[Wei14] Alexander Weinert. Problem generation for DFA construction. 2014.