UNIVERSITY OF AMSTERDAM

Authors: Robin Visser & Tristan Laan
Based on the work by:
Bas van den Heuvel
Daan de Graaf

Automaten en Formele Talen
Teacher: R. Reiffenhäuser
Supervisor: Tristan Laan
9th April 2024

Assignment 3

Reverse engineering a Turing machine

Due: 23rd May 2024

Introduction

In Assignment 2.3 you were asked to determine the output produced by a Turing machine given an input and the corresponding execution trace. The possibility of such a computation highlights an interesting fact about the 'closed-source' Turing machine systems that we have analyzed in the previous assignments:

- A Turing machine execution trace can be interpreted as a complete list of program instructions **for a corresponding input**.

A corollary of this statement is that, given a finite list of inputs and the corresponding execution traces produced by a single unknown Turing machine, one has enough information to create a second Turing machine whose behaviour is identical to the first (for the given inputs). In other words, we can partially *reverse engineer* a Turing machine given a sample of its traces.

Assignment 3.1: Input extraction (20 points)

In Assignment 2.3 you were given the Turing machine's input along with the execution trace. If you did attempt the computation of the output you may have noticed something:

- (Part of) the input that makes a Turing machine produce the corresponding execution trace is embedded within that trace.

Indeed, at the beginning of every step a Turing machine reads the character that is stored within the current cell (and thus the character that was read is 'recorded' in the trace). As an example, consider the following trace produced by some unknown Turing machine:

$$
\begin{aligned}
&- \vdash + \vdash > \\
&- \ \mathtt{a} + \sqcup > \\
&- \ \mathtt{b} + \sqcup > \\
&- \ \mathtt{c} + \sqcup > \\
&- \sqcup + \sqcup >
\end{aligned}
$$

event repr.	event descr.
$<$	Move left
$>$	Move right
$-$	Read
$+$	Write
<symbol literal>	symbol $\in \Gamma$

Because it moves to the right at end of every step, the input 'abc' in neatly embedded within the trace. And since the fifth cell initially contains a BLANK ($\sqcup$) character, which is not part of the

input alphabet, we know that the entire input has been read. Note that even though not every trace may behave as nice as this example, the following should always hold:

- Whenever a Turing machine lands on a cell *for the first time*, and if that cell does not contain a left endmarker (⊢) or BLANK (␣), then its contents are part of the input.

The goal of the assignment is to derive an input extraction algorithm from this statement and implement it in our framework. To that end, implement the **extract_input()** function in **reverse.py**, which returns the input for any given trace.

- You may use regular Python 3 for the implementation (i.e. you are **not** limited to some automaton).
- You may assume that a Turing machine uses its entire input, that is, every piece of input is read at least once.
- Write the function in such a way that it works on any execution trace and any length input.

The **extract_input()** function takes a Turing machine execution trace and returns a string containing the input, like the following examples:

```
extract_input("- ⊢ + ⊢ > - 0 + 1 > - 0 + 1 < - 1 + ␣ > - 1 + ␣ > - ␣ + a >")
"00"
extract_input("- ⊢ + ⊢ > - a + ⊢ < - ⊢ + ⊢ > - ⊢ + ␣ > - b + ␣ < - ␣ + ⊢ > "
              "- ␣ + ␣ > - ␣ + ⊢ <")
"ab"
```

Assignment 3.2: Output extraction (20 points)

After a Turing machine has halted one may look at the resulting tape and determine an output from its contents. For our framework we use the following definition:

- A Turing machine's output is the longest possible string after the first left endmarker that does not end in a BLANK (␣) character.

Like the input, the output is also embedded in the execution trace. That is, a trace contains a record of every character that is written to the tape during execution. The goal of the assignment is to derive an output extraction algorithm and implement it in our framework. To that end, implement the extract_output**()** function in **reverse.py**, which returns the output for any given trace.

- You may use regular Python 3 for the implementation (i.e. you are **not** limited to some automaton).
- You may assume that the entire input and output are embedded within any given trace.
- Write the function in such a way that it works on any execution trace and any length output.

The **extract_output()** function takes a Turing machine execution trace and returns a string containing the output, like the following examples:

```
extract_output("- ⊢ + ⊢ > - 0 + 1 > - 0 + 1 < - 1 + ␣ > - 1 + ␣ > - ␣ + a >")
"␣␣a"
extract_output("- ⊢ + ⊢ > - a + ⊢ < - ⊢ + ⊢ > - ⊢ + ␣ > - b + ␣ < - ␣ + ⊢ "
               "> - ␣ + ␣ > - ␣ + ⊢ <")
"⊢␣⊢"
```

Assignment 3.3: Reverse engineering, manually (35 points)

With input/output extraction readily available, it is now time to reverse engineer our first Turing machine. You will find a sample of 12 execution traces in **traces.txt** (with the tokenized counterparts in **traces_tokenized.txt**). All the traces were produced by the same unknown Turing machine. The **main()** function already contains the code to import the traces from the files, simply run: `python3 reverse.py traces.txt traces_tokenized.txt`.

1. Our first goal is to try to determine what the Turing machine was designed to do. We can use our newly developed extraction functions here. By extracting the input and output of each trace in the sample, we might be able to find a pattern. In addition, the framework provides a static method (**TM.visualize(input, trace)**) that can visualize the computation that occurs in any valid execution trace. Of course you can also write your own methods. The **main()** function has a cratchpad area where you can easily run tests on the traces. **Once you have figured out what the Turing machine was designed to do, report your findings in the docstring of the reverse_manually() function.**

2. Now that we know what the Turing machine does, the next step is to find out how exactly it does it. A straightforward way could be to use the **TM.visualize(input, trace)** method to follow along, step by step, with a few of the provided traces. **Once you have a good idea of how the algorithm works (and how you would design a Turing machine that implements it), report your findings in the same docstring as before.**

3. The last step is to actually build a TM object (from **TM.py**, see the appendix at the bottom of this document) that implements the algorithm in a way that is indistinguishable from the original.

   - It is not enough for your Turing machine to only produce the same outputs, **it must produce the exact same execution traces as well**.
   - Build the new Turing machine by making sure that it does exactly the same thing as the traces every step of the way.
   - The only additional constraint on your TM is that it may not have more than 20 *states*.

   Implement your Turing machine (and an explanation of how you built it) in the **reverse_manually()** function.

Assignment 3.4: Reverse engineering (25 points)

In Assignment 3.3 our first step towards reverse engineering was to identify what the unknown Turing machine was designed to do. In general we can not assume that it is possible to figure out the function of a Turing machine (it could just be a pseudo-random amalgamation of states and transitions). Luckily, this is not a hard requirement for reverse engineering. The goal here is to write a method, **reverse_generic()**, that is capable of generating a reverse-engineered Turing machine for any given sample of coherent valid execution traces. The exact approach is up to you, though we will discuss the problems that must be solved:

1. **Extracting Sigma and Gamma.** In Assignment 3.3, the input alphabet (Σ) and the tape alphabet (Γ) were both implicitly assumed to be known. However, in the generic case, we have to somehow extract (the relevant part of) these from the given traces. Doing so is somewhat similar to how we did input/output extraction earlier, but with one important caveat: there can be elements that are only visible in an intermediate stage. For example, consider the following trace:

"- ⊢ + ⊢ > - a + b > - c + d < - b + a > - d + e > - a + c > - d + a >"

The element 'b' is not present in either the input or the output, but it is still part of Γ (and possibly Σ) and must be extracted. Additionally, you may/must assume that all elements of Σ and Γ are at most one character in length.

2. **Automated reverse engineering.** One of the guarantees that we have is that the set of traces came from a single valid Turing machine. As such, their behaviour must be consistent. For example: if one trace has input "0000" and another has input "0001" then we can say for certain that, for these two traces, the behaviour of the Turing machine (and thus the traces themselves) is/are *identical* right up until the point where the Turing machine visits the differing fourth cell of the inputs. As such a valid reverse engineering strategy could be:

   (a) Reverse engineer a single trace.

   (b) Reverse engineer another trace and combine/merge the results.

   (c) Repeat step 2b until there are no traces left.

3. **Additional constraints & Testing.** Contrary to Assignment 3.3, there is no limit on the amount of states a Turing machine may have here. Generating Turing machines with a large amount of states is expected behaviour. As for testing, initially you can use the same set of 12 traces from earlier. Further tests requires generating sets of traces from Turing machines you create yourself. For grading, you will get partial points for producing partial traces.

Submission & Grading

Submit your work to Canvas before 23rd May 2024 by submitting the tarball **PO3.tar.gz** created by running `./create_submission.sh`.

If your code is not written in Python 3 you will receive no points. In addition, make sure your code:

- Complies with the **PEP8** style guide[1] (e.g. by using flake8).
- Is properly commented.

If your program fails to run (or crashes) your work will be graded by judging the code alone, but it will no longer be possible to score all the points. Your code will be tested using **Python 3.11**.

On the other hand, fully functional code does not guarantee that you will receive a perfect score. You can see the rubric in CodeGrade for a complete overview of how your submission is graded.

After submitting, make sure the **'Verify submission'** and **'Check code style'** AutoTests have **passed** in CodeGrade. **If we have to modify your code to run the AutoTests, you will not be able to score all the points!**

Assignment due date: 23rd May 2024

[1] https://www.python.org/dev/peps/pep-0008/

Appendix – Turing machines and TM.py

The TM class in the assignment (**TM.py**) is identical to the 'deterministic one-tape model' defined in Kozen (the one discussed in Assignment 1). It is recommended to fully read **TM.py** before attempting Assignments 3.3 and 3.4 (as you might/will need most functions defined there). To ease the burden of digesting the TM class, a basic example is provided here; a Turing machine which erases its input by writing BLANKs (␣):

```
Q = ['Q1', 't', 'r']
Sigma = ['0', '1']
Gamma = ['0', '1', '␣', '⊢']
delta = [
         (('Q1', '0'),('Q1', '␣', 'R')),
         (('Q1', '1'),('Q1', '␣', 'R')),
         (('Q1', '⊢'),('Q1', '⊢', 'R')),
         (('Q1', '␣'),('t',  '␣', 'R'))
        ]
s = 'Q1'
t = 't'
r = 'r'
my_tm = TM(Q, Sigma, Gamma, delta, s, t, r, verbose=True)
my_tm.set_input("101010")
my_tm.transition_all()
```

The **'verbose=True'** flag makes a TM print a visualisation of the tape and computation. Unicode characters for the left endmarker (⊢) and BLANK (␣) are provided in **reverse.py**.