

INFSCI 2950 - INDEPENDENT STUDY

Automatic Cognitive Tutor Generator for Python Code

Snippets

Keyang Zheng

kez20@pitt.edu

School of Computing and Information

1. Overview

Inspired by Professor Peter Brusilovsky and Yun Huang, my topic for this independent study is focused on developing an automatic generator to generate a CTAT Example Trace Cognitive Tutor from a provided Python code snippets.

The system is currently available at:

<http://tutor-converter.herokuapp.com/converter/>

The overall idea of automatically generating the tutor is that building a CTAT Example Trace Tutor requires user to manually record all the steps in the tutor. This process is generally fine when only a small number of tutor is required to be built. But when a large number of tutors is required, the process can be very tiring. Because all the tutors only require the correct code processing path. If we can obtain the correct step by step code processing record, generating the tutor is possible. In this system, I used the debugger module from Python, to obtain the step by step debugging information, which includes the line number and the value of each variables. These information is enough for building the tutor automatically.

2.Objectives:

- Automatically generate the Behavior Graph file for cognitive tutor from the provided Python code
- Automatically generate the HTML Interface for the cognitive tutor
- Provide the generator as a standalone web app, for general use.

3.Used Techniques:

Python, Django, PostgreSQL, HTML, CSS, JavaScript, jQuery, Bootstrap.

4.System Design

In order to generate the tutor from provided Python code, the system takes the following steps:

1. Obtain the step by step debugging trace, which represent the machine's knowledge of how the code is executed.
2. Create an cognitive model from the trace, which represent the human's knowledge of how the code is executed.

3. Create the Behavior Graph file from the cognitive model.
4. Combine the Behavior Graph File and the generated HTML Interface, and create the tutor.

I - Obtaining the debugging trace

The idea of filling out the value of each variable line by line while executing the code is very similar to setting breakpoint on every line and debug step by step. And this is the reason the system is using the debug trace for the code rather than analyse the code directly.

The [PythonTutor](#) project developed by [Philip Guo](#) also uses the similar approach, and the backend he developed already contains a recorder for the debug trace.

However, the recorder is developed in a way which is closely integrated with whole PythonTutor system. So I developed a warp method for the recorder to integrate into my system, which takes the String of the code and return with a Dict type trace to be used in the next step.

II - Generate Cognitive Model

After obtaining the debug trace, the next step is to interpret the trace. The debug trace is the computer version of the cognitive model on how the code is executed, which is different from how a human in a learning process would understand about how the code is executed.

The main difference lies in the line number and the corresponding values of every variable. In human's cognitive model, we stop the execution on every line after that line's code is executed, so the values of every variables is updated with the result of this line's execution. But the breakpoint in debug stop the code execution before the line where the breakpoint exist, so the values of every variables is the result of the last line's execution. In order to get the result of this line's execution, we have to stop at the next line and get the value of those variables.

In my system, when interpret the debug trace, I added additional context information to record which line is executed a step earlier than the current line. So the variables' value is the result of that earlier line.

The function that fulfill this process is

```
get_cognitive_model(variables: list, db_trace: dict):
```

```
<4.1 trace_analysis.py>
```

```
def get_cognitive_model(variables: list, db_trace: dict):  
    trace = db_trace['trace']
```

```
cognitive_model = []

# initialization
snapshot = {"line": 1, "previous_line": 1}
for variables in variables:
    snapshot[variables] = 0

cognitive_model.append({"changed": "line", "value": snapshot['line'], "line":
snapshot['line']})

# running the code and update the snapshot
for step in trace:
    has_error, msg = check_trace_result(step)
    if not has_error:
        g_vars = step['globals']
        for var_name in g_vars:
            if g_vars[var_name] != snapshot[var_name]:
                cognitive_model.append({"changed": var_name, "value": g_vars[var_name],
"line": snapshot['line']})
                snapshot = update_snapshot(var_name, g_vars[var_name], snapshot)

        line = step['line']
        if line != snapshot['line']:
            cognitive_model.append({"changed": "line", "value": line, "line": line})
            snapshot = update_snapshot('previous_line', snapshot['line'], snapshot)
            snapshot = update_snapshot('line', line, snapshot)

    if step['stdout'] != '':
```

```
        cognitive_model.append({"changed": 'stdout', "value": step['stdout'],
"line": line})
        variables.append('stdout')

    else:
        print(msg)
        break

    return cognitive_model, variables
```

III - Generate Behavior Graph File (*.brd)

The most important part of the system to generate the Behavior Graph File, which store the information of every steps that is required to do in order to successfully execute the code.

The brd file is basically an XML file with different file name. Because we have already obtain the cognitive model of the provided code, following every steps of the cognitive model is enough to create the brd file. The brd file mainly consists of 2 major parts: nodes and edges. Edges corresponding to each step in the cognitive model, whereas nodes corresponding to the state between every step, since every step is basically an action the code snippet took from human's perspective.

I used a Python package called Yattag to create the XML style brd file. The package uses the Python context manager to create the XML structure in a readable fashion. The following

<4.2 behavior_graph.py>

```
def edge(prop: dict, var_pos: dict):
    doc, tag, text, line = Doc().ttl()

    with tag('edge'):
        with tag('actionLabel', preferPathMark="true", minTraversals="1",
maxTraversals="1"):
            with tag('studentHintRequest'):
                pass

            with tag('stepSuccessfulCompletion'):
                pass

            with tag('stepStudentError'):
                pass

            line('uniqueID', str(prop['id']))

            doc.asis(message_section(get_edge_action_msg(prop['step_value'], prop['target'])
if prop['target'] != 'done' else get_edge_ending_action_msg()))

            line('buggyMessage', 'No, this is not correct.')

            with tag('successMessage'):
                pass

            if prop['target'] != 'done':
                line('hintMessage', "Please enter '" + str(prop['step_value']) + "' in the
highlighted field.")
            else:
                line('hintMessage', "Please click on the highlighted button.")

            with tag('callbackFn'):
                pass

            line("actionType", "Correct Action")
            line("oldActionType", "Correct Action")
            line("checkedStatus", "Never Checked")

            with tag('matchers', Concatenation="true"):
                with tag('Selection'):
                    with tag('matcher'):
                        line('matcherType', 'ExactMatcher')
                        line('matcherParameter', prop['target'], name="single")

                with tag('Action'):
                    with tag('matcher'):
```



```
        line('matcherType', 'ExactMatcher')
        line('matcherParameter', prop['action'], name='single')

    with tag('Input'):
        with tag('matcher'):
            line('matcherType', 'ExactMatcher')
            line('matcherParameter', str(prop['step_value']), name='single')

        line('Actor', 'Student', linkTriggered="false")

    line("preCheckedStatus", "No-Applicable")

    with tag('rule'):
        line('text', 'unnamed')
        line('indicator', '-1')

    line('sourceID', str(prop['id']))
    line('destID', str(prop['id'] + 1))
    line('traversalCount', '0')

return doc.getvalue()
```

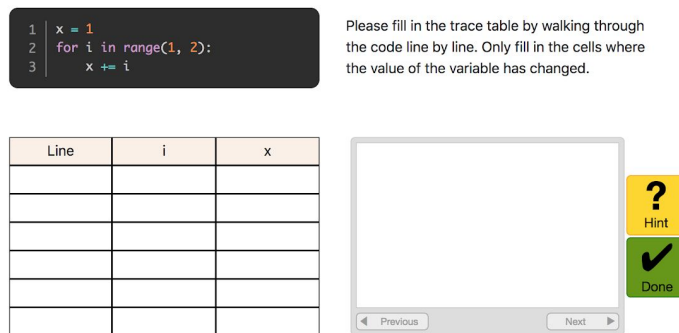
After the Behavior Graph being created, it is not stored in a file on disk, but stored in the database as JSON-coded content. When the server receive a request for the specific behavior graph file, the behavior graph will be fetched from the database, decoded and generated for download as a BRD file.

IV - Combine the Behavior Graph File and the generated HTML Interface, and create the tutor

This is the step processed by the Django server. The system provides a quick way to preview the generated Behavior Graph file in action with generated interface, as well as download the generated Behavior Graph File as shown in Figure 4.1.

The tutor interface consists of 4 different parts: the code snippet, instruction, user input table and the hints. Asides from the instruction, other three parts are all generated dynamically. The code snippet is the Python code provided by the user with highlighted syntax and line number. The number of rows and columns the user input table has is determined by the number of variables and the number of steps in order to correctly execute the code. The hints field shows the hints for user if they ever face difficulties when trying to finish the tutor.

CTAT Tutor Generator



The interface displays a code snippet in a dark box:

```
1 x = 1
2 for i in range(1, 2):
3     x += i
```

Below the code is a trace table with the following structure:

Line	i	x

To the right of the table is a large empty box for user input. Below this box are two buttons: a yellow button with a question mark labeled "Hint" and a green button with a checkmark labeled "Done". At the bottom of the input box are "Previous" and "Next" navigation buttons.

Please fill in the trace table by walking through the code line by line. Only fill in the cells where the value of the variable has changed.

Figure 4.1 Interface for Preview the Tutor

The interface utilize the JavaScript phraser provided by CMU CTAT group, which requires the a BRD file as input. Since all the Behavior Graph Files are not stored on the server disks but in the database, the web page will issue another HTTP GET

request to the server and the server will dynamically generate the file and return it as response. The same procedure will repeat when user try to download the BRD file.

Following code snippets is for the dynamically creating the BRD file.

<4.3 view.py>

```
def brd_download(request, behavior_model_id):
    behavior_model = get_object_or_404(BehaviorGraph, pk=behavior_model_id)
    json_dec = json.decoder.JSONDecoder()
    response = HttpResponse(json_dec.decode(behavior_model.brd),
content_type='application/brd')
    response['Content-Disposition'] = 'attachment; filename=' + behavior_model.problem_name
+ '_brd.brd'
    return response
```

5.Future Improvement

Support for complex data type and code flow

The system currently does not support more complex data types which requires the debugger to access the heap data. This limited the options for what kind of code snippets the system can accept. But for basic ideas like loop, if else blocks, the system can handle candidly. Recursions, class definitions, decorators and other

advanced topic is not supported in this version of the system, as they also needs access to the heap data.

Variable selection for the tutor

In the meantime, when code snippets become more and more complex, the number of variables involved in the code snippets also become larger. System should be able to let the user choose which variables to focus on rather than displaying all of them.

Also with added complexity of code execution order, local variables are becoming more important in understanding the execution order of the code. With more clear picture, I would be able to phrase out the local variables, and track them as the code executed.

Improvement of the Tutor Interface

Adding more complex data type also poist interface design challenges as how to display these data. I have yet to find a clear design to display the all the required information on screen without compromise the simple and easily user interaction with the system and tutor. For instance, how to represent a list object or a dict object in a way that is clear for user to see the total value of the object, as well as for user to enter the changed value of an individual cell or <key, value> pair.

6.Limitation and Constraints:

Due to the nature of the tutor, the Python snippets, which the specific tutor is built upon, is required to contain no user inputs. As the tutor is designed to test learner's understanding of basic programming knowledges. Also, technical constraints require the code snippets to not include function definition or class definitions, as well as non-primitive data types.

7.Codes:

<https://github.com/albuszheng/autoTutorGen>