School of Computing and Information Systems

comp10002 Foundations of Algorithms
Semester 2, 2020
Assignment 2

Learning Outcomes

In this project, you will demonstrate your understanding of dynamic memory and linked data structures (Chapter 10), and extend your skills in terms of program design, testing, and debugging. You will also learn about Robotic Process Automation (RPA) for executing routine clerical tasks in organizations, and implement simple mechanisms for checking candidate routines in a given trace of executed activities.

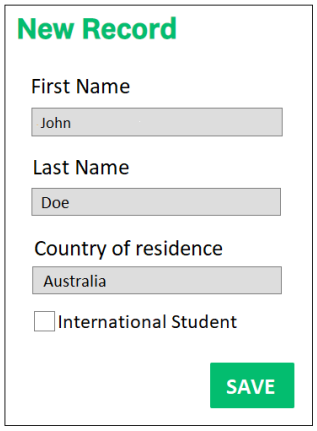
Robotic Process Automation

RPA is an emerging technology that allows organizations to automate repetitive clerical work by executing software scripts (RPA scripts) that support executions of sequences of fine-grained interactions with Web and desktop applications. An example of such repetitive clerical work is transferring data across information systems, for instance, from a spreadsheet into a Web form (refer to Figure 1).

A common approach to eliciting RPA scripts nowadays is through observing workers or interviews, which is time-consuming. Alistair and Artem would like to start a business, called A&A RPA, which will offer a new product to the market. This product will aim to automate the process of identifying routines, i.e., sequences of interactions with User Interfaces of information systems that lead to the same effect. For example, in the context of Figure 1, a routine is a procedure for transferring data from a row in sub-figure (a) to the Web form in sub-figure (b) and clicking the "Save" button. As the first step, we developed a tool that records a *trace* of consecutive *actions*, where an action captures information about user interaction with an information system. Examples of actions include selecting a cell in a spreadsheet, copying the content of a cell, and clicking a button in a Web form. An action is characterized by a *precondition* and *effect*, each captured as a collection of Boolean variables.

	A	B	C
1	Name	Surname	Country of residence
2	John	Doe	Australia
3	Albert	Rauf	Germany
4	Steven	Richards	Australia
5	Gerard	Dubois	France
6	Audrey	Backer	USA
7	Carl	Gustafsson	Sweden
8	Sarah	Johnson	Australia
9	Andrea	Bolzano	Italy
10	Hannah	Dietmeier	Germany
11	Igor	Honchar	Ukraine
12	Oliver	Dunkan	Ireland
13	Terry	Lee	Australia
14	Volodymyr	Leno	Ukraine
15	William	Macdonald	Canada
16	Jorge	Canales	Spain
17	Thomas	Taylor	Australia
18	Jack	Brown	Australia
19	Christina	Esposito	Italy
20	Amelia	Wilson	Australia

(a) Student records spreadsheet



(b) New Record creation form

Figure 1: Extract of spreadsheet with student data that needs to be transferred to a Web form; `https://bit.ly/2SPet3u`.

For example, the precondition of clicking the "Save" button action in the Web form in Figure 1(b) may be that variables `FirstNameEntered`, `LastNameEntered`, and `CountryOfRresidenceEntered` are all set to true, and variable `InternationalStudentSelected` is set to false. The effect of this example action may be that variable `NewStudentEntryCreated` is set to true, and all the precondition variables are set to false.

As the next step, we would like to develop a tool that given a trace and a *candidate routine*, both specified as sequences of actions with preconditions and effects, identifies all sub-sequences of actions in the trace that generate the same cumulative effect as the candidate routine. Candidate routines that lead to identifying many such sub-sequences can be recommended for implementing in RPA scripts. Note that the identified sub-sequences may, in general, be different as, for example, a user may be transferring content from the spreadsheet cells to the form in different orders for different students (first name then last name, or vice versa).

Your task is to implement a tool for checking candidate routines.

Input Data

Your program should read input from `stdin`. The input starts by listing names of Boolean variables, one per line, that must be set to true in the *initial state*, where the initial state is defined by values of all the Boolean variables before a trace gets executed. All the other variables must be set to false in the initial state. An input line

with a single character '#' denotes the end of input of the initial state. The lines that follow define actions, one action per line. Each *action* is defined as a quintuple. The first two elements of an action list its *precondition*, i.e., variables that must be set to true and false to enable the action. The third element is the name of the action. The last two elements of an action specify its *effect*, i.e., variables that are set to be true and false after the action is executed. Another input line with a single character '#' denotes the end of the action definition section of the input. The subsequent input characters encode a trace, each referring to the name of the action executed in the corresponding position of the trace. All the lines of the input, except the last line, are not empty. The following file `test0.txt` defines the initial state, three actions, and a trace executed from the initial state (the numbers in italics show line numbers and are not part of the input file).

```
1 a
2 b
3 x
4 y
5 z
6 #
7 ab:c:X:cf:b
8 f::V:pq:
9 fpq:b:U::pq
10 #
11 XVUVU
12
```

Boolean variables are denoted by characters 'a', 'b', ..., 'z' (lower-case Latin alphabet letters), while action names are referred to as 'A', 'B', ..., 'Z' (upper-case Latin alphabet letters). Hence, the above input specifies the initial state in which variables `a`, `b`, `x`, `y`, and `z` are set to true, while variables `c`, `d`, ..., `w` are set to false. The input trace consists of five actions, see line 11. From the initial state, the trace commences by executing action `X`, followed by an execution of action `V`, followed by action `U`, followed by another occurrence of `V`, and concludes by another occurrence of action `U`. Actions `U`, `V`, and `X` are defined at lines 7–9 of the input. The five components of each action are delimited using the colon character ':'. For example, the action defined at line 7 of the input has name `X`. This action can occur in a state where variables `a` and `b` are set to true, and variable `c` is set to false. Once this action is executed, variables `c` and `f` are set to true, while variable `b` is set to false.

Stage 0 – Reading and Analyzing Input Data (8/20 marks)

The first version of your program should read the input from `stdin`, ensure that the input trace is valid, and print out some basic information so that you can be sure you have read the inputs correctly. A *valid* trace is composed of actions that occur in states that satisfy their preconditions; otherwise, the trace is *invalid*. The required output from this stage generated for `test0.txt` file shown above is the summary shown below on the left.

```
mac: ass2-soln < test0.txt
==STAGE 0==============================
Number of distinct actions: 3
Length of the input trace: 5
Trace status: valid
---------------------------------------
  abcdefghijklmnopqrstuvwxyz
> 11000000000000000000000111
X 10100100000000000000000111
V 10100100000000011000000111
U 10100100000000000000000111
V 10100100000000011000000111
U 10100100000000000000000111
```

```
mac: ass2-soln < test1.txt
==STAGE 0==============================
Number of distinct actions: 3
Length of the input trace: 5
Trace status: invalid
---------------------------------------
  abcdefghijklmnopqrstuvwxyz
> 11000000000000000000000111
X 10100100000000000000000111
V 10100100000000011000000111
U 11100100000000000000000111
V 11100100000000011000000111
```

The basic information printed by your program should specify the number of distinct actions, length of the trace, and the status of the trace read from the input, which are `3`, `5`, and `valid`, respectively, for input file `test0.txt`. This information should be followed by a delimiter line, followed by the detailed specification of the trace given as a table of ASCII characters. The columns in this table refer to variables, while rows specify states and executed actions. A state is encoded as a sequence of '0' and '1' characters, where '0' indicates that the corresponding variable (specified in the column header) is set to false and '1' indicates that the variable is set to true. The first printed state, denoted by character '>' in the left-most column, specifies the initial state. Each subsequent line prints the next executed action, followed by the sequence of zeros and ones that encodes the state one achieves after executing the action. For instance, the execution of action `X` in the example input trace flips the values of variables `b` and `f` in the initial state.

If the input trace is invalid, your program should report the corresponding status of the trace and print its valid prefix. For example, if line 9 in file `test0.txt` is changed from "`fpq:b:U::pq`" to "`fpq:b:U:b:pq`" to obtain file `test1.txt`, then your program should produce the output shown above on the right. In this case, the second occurrence of action `U` cannot take place as its precondition requires that variable `b` is set to false. However, `b` is set to true after the execution of the first four actions of the trace.

All inputs will follow the format specified in the Input Data section. No input action will specify that a variable is equal (or should be set) to true and false simultaneously, e.g., "`a:a:X:b:b`". Also, no input action will set a value of a variable as in the precondition, e.g., "`a:b:Y:a:b`". Your program will not be tested on erroneous inputs. Refer to the input files distributed with this specification for the exact formatting of inputs.

Stage 1 – Check Basic Routines (16/20 marks)

Extend your program from Stage 0 to perform a basic check of a candidate routine by identifying all trace sub-sequences that produce the same effect. All of the Stage 0 output should be retained. If the input proceeds with a line with a single character '`#`', your program should generate Stage 1 output, which starts with this line:

```
==STAGE 1==============================
```

For each subsequent input line containing a candidate routine specified as a sequence of action names, discover and print non-overlapping sub-sequences of consecutive actions of the trace that produce the same cumulative effect as the candidate routine. Proceed from left to right in the trace. Once the *shortest* sub-sequence with the desired effect is identified, record it and restart searching from the next position in the trace.

The cumulative effect of a routine or sequence of actions is determined as all the values (true and false) set to the Boolean variables as effects of the actions executed in the order they appear in the routine/sequence. Considering input file `test0.txt`, candidate routine `VU` generates the cumulative effect of setting variables `p` and `q` to false. First, action `V` sets `p` and `q` to true. Next, action `U` updates them to be false. Hence, would input file `test0.txt` be modified to contain a single character '`#`' at line 12 and string "`VU`" at line 13 to encode a candidate routine followed by the new line character, Stage 1 output should proceed by printing these lines:

```
Candidate routine: VU
    1: VU
    3: VU
```

Here, the first line specifies the candidate routine and the two subsequent lines refer to two sub-sequences that start at positions 1 and 3 of the input trace (counting positions from zero); trace positions are printed using the formatting string "`%5d`". These two sub-sequences generate the same cumulative effect as the candidate routine.

Each subsequent input line with a candidate routine should be processed in the same way, and its output should be separated from the previous output by the delimiter line composed of '`-`' characters, refer to Stage 0 specification above for examples of using the delimiter line. Consider input file `test2.txt` listed below.

```
1 a
2 #
3 a:b:A:x:
4 x:b:B:yz:
5 x:b:C:y:
6 ::D:xyz:
7 ::E:z:
8 x::F:u:xyz
9 a:ij:G:ij:
10 xy:b:H::ij
11 #
12 DFABFACGHE
13 #
14 AB
15 D
16 #
17 D
18 AB
19
```

The Stage 1 output for input file `test2.txt` is shown below on the left. Indeed, both sequences of actions `D` and `AB` have the cumulative effect of setting variables `x`, `y`, and `z` to true, and are therefore identified. Sub-sequences with smaller start positions should be printed first.

```
==STAGE 1==============================
Candidate routine: AB
    0: D
    2: AB
---------------------------------------
Candidate routine: D
    0: D
    2: AB
```

```
==STAGE 2==============================
Candidate routine: D
    0: D
    2: AB
    5: ACGHE
---------------------------------------
Candidate routine: AB
    0: D
    2: AB
    5: ACGHE
==THE END==============================
```

Stage 2 – Check Advanced Routines (20/20 marks)

Extend your program from Stage 1 to perform further checks for the input candidate routines. All of the output from Stages 0 and 1 should be retained. If your input after Stage 1 continues with a single character line of '`#`', process subsequent input lines with candidate routines (one routine per line). This time, an identified sub-sequence of actions in the input trace should be allowed to modify the values of variables not set by the candidate routine. However, in this case, once all the actions of the identified sub-sequence are executed, the

values of such variables must be set to the values they had before executing the sub-sequence. Note that this knowledge of variable values can be obtained from the preconditions of the trace actions.

For example, the sub-sequence of actions `ACGHE` in the trace of input file `test2.txt` sets values of variables `i`, `j`, `x`, `y`, and `z`. Note that, to execute, action `G` requires that `i` and `j` are both set to false, and sets them to true. However, next, action `H` sets them to false, which are their original values at the start of the sub-sequence. Hence, the cumulative effect of executing `ACGHE` is the same as that one of executing `D` or `AB`, i.e., setting `x`, `y`, and `z` to true. The complete output of Stage 2 for input file `test2.txt` is proposed on the right of the listings presented in the specification of Stage 1. If your program concludes with Stage 2, at the end, it should output this line:

```
==THE END==============================
```

Important...

The outputs generated by your program should be exactly the same as the sample outputs for the corresponding inputs. You should not assume the maximal allowed lengths for the input trace and candidate routines and, thus, use `malloc` and *linked lists* to store them. Regardless of the algorithmic techniques you will employ in this assignment, your program should run under 1 second on each sample input.

Boring But Important...

This project is worth 20% of your final mark, and is due at **11:00pm on Friday 30 October**.

Submissions that are made after that deadline will incur penalty marks at the rate of two marks per day or part day late. Students seeking extensions for medical or other "outside my control" reasons should email `ammoffat@unimelb.edu.au` as soon as possible after those circumstances arise. If you attend a GP or other health care service as a result of illness, be sure to take a Health Professional Report (HPR) form with you (get it from the Special Consideration section of the Student Portal), you will need this form to be filled out if your illness develops in to something that later requires a Special Consideration application to be lodged. You should scan the HPR form and send it with any non-Special Consideration assignment extension requests.

A rubric explaining the marking expectations is provided on the FAQ page. You need to submit your program for assessment; detailed instructions on how to do that will be posted on the FAQ page once submissions are opened. Submission will *not* be done via the LMS or `grok`, and you will instead use a software system known as `submit`. You can (and should) use `submit` **both early and often** – to get used to the way it works, and also to check that your program compiles correctly on our test system, which has some different characteristics to the lab machines. *Failure to follow this simple advice is highly likely to result in tears.* Only the last submission that you make before the deadline will be marked. Marks and a sample solution will be available on the LMS two weeks after submissions close.

Academic Honesty: You may discuss your work during your workshop, and with others in the class, but what gets typed into your program must be individual work, not copied from anyone else. So, do **not** give hard copy or soft copy of your work to anyone else; do **not** "lend" your "Uni backup" memory stick to others for any reason at all; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "**no**" if they ask to see your program, pointing out that your "**no**", and their acceptance of that decision, are the only way to preserve your friendship. See `https://academicintegrity.unimelb.edu.au` for more information. Note also that solicitation of solutions via posts to online forums, whether or not there is payment involved, is also Academic Misconduct. In the past students have had their enrolment terminated for such behavior.

The FAQ page contains a link to a program skeleton that includes an Authorship Declaration that you must "sign" and include at the top of your submitted program. Marks will be deducted (see the rubric linked from the FAQ page) if you do not include the declaration, or do not sign it, or do not comply with its expectations. A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions. Students whose programs are identified as containing significant overlaps will have substantial mark penalties applied, or be referred to the Student Center for possible disciplinary action, without further warning.

Nor should you post your code to any public location (`github`, `codeshare.io`, etc) while the assignment is active or prior to the release of the assignment marks.