School of Computing and Information Systems

comp10002 Foundations of Algorithms
Semester 2, 2022
Assignment 2

Learning Outcomes

In this project, you will demonstrate your understanding of dynamic memory and linked data structures (Chapter 10) and extend your program design, testing, and debugging skills. You will also learn about the problem of *process discovery* and implement a simple algorithm for discovering a process model from event data.

Process Discovery

Process mining is a relatively young research discipline that combines studies of inferences from data in data mining and machine learning with process modeling and analysis to discover, monitor, and improve real-world processes. The studied processes get formally captured in process models.

Figure 1 shows an example process model captured in Business Process Model and Notation (BPMN) language. In the figure, *actions* are drawn as rectangles with rounded corners. Diamonds represent *gateways*. An exclusive gateway has the "$\times$" marker, while a parallel gateway has the "$+$" marker inside the diamond shape. Directed arcs encode *control flow* dependencies. The circles with thin and thick borderlines denote the start and end of the process, respectively.

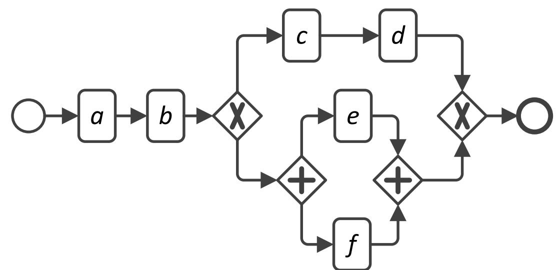

Figure 1: A process model.

A process model describes a collection of possible executions, or traces. A *trace* is a sequence of *events* that represent actions encountered when following control flow arcs and gateway routing decisions from the start to the end element of the model. If execution reaches an exclusive gateway, it continues along exactly one of its outgoing arcs. If execution reaches a parallel gateway with multiple outgoing arcs, all the outgoing arcs are followed simultaneously. A parallel gateway with multiple incoming arcs "waits" for all the incoming arcs to complete and then "passes" the execution to its outgoing arcs. The fact that several paths in a model are executed simultaneously means that the actions encountered on these paths are independent and can occur in any order. According to these rules, the model in Figure 1 describes exactly three traces: $\langle \texttt{a}, \texttt{b}, \texttt{c}, \texttt{d} \rangle$, $\langle \texttt{a}, \texttt{b}, \texttt{e}, \texttt{f} \rangle$, and $\langle \texttt{a}, \texttt{b}, \texttt{f}, \texttt{e} \rangle$. For simplicity, the model in Figure 1 uses abstract action labels. In practice, actions usually "wear" meaningful labels to describe, for example, a business process or a distributed algorithm. For instance, the process model from Figure 1 with labels $\texttt{a} =$"*receive loan application*", $\texttt{b} =$"*assess loan application*", $\texttt{c} =$"*reject loan application*", $\texttt{d} =$"*close loan application*", $\texttt{e} =$"*approve loan application*", and $\texttt{f} =$"*offer additional products*" describes some bank's loan application handling process.

In the real-world, different loan applications can be processed by following the same sequence of actions; thus, a trace can be observed multiple times. The problem of *process discovery* is a core problem in process mining that, given an *event log*, that is, a collection of observed traces, aims to automatically construct a process model that describes the traces. A process model constructed in this way aims to aggregate and generalize information on how processes were executed in the real-world. Process analysts can then use this constructed process model to understand real-world processes, identify interesting and problematic patterns of actions, and use the model as a starting point for redesigning and improving the processes.

In this assignment, you will implement a program that reads an event log from input, analyzes patterns of actions in the input traces, and discovers and reports model fragments that could have produced the traces.

Input Data

Your program should read input from `stdin` and write output to `stdout`. The input will list traces, one per input line, where each trace is provided as a comma-separated list of events that terminates either with "`\n`", that is, one newline character, or `EOF`, that is, the end of file constant defined in the `<stdio.h>` header file. An event is

encoded as a single alphabetic character that when provided as input to the `isalpha()` function defined in the `<ctype.h>` header file returns a non-zero integer.

The following file `test0.txt` uses fifteen lines to specify fifteen traces, each composed of four events.

```
1  a,b,c,d
2  a,b,f,e
3  a,b,f,e
4  a,b,f,e
5  a,b,c,d
6  a,b,c,d
7  a,b,c,d
8  a,b,c,d
9  a,b,c,d
10 a,b,e,f
11 a,b,e,f
12 a,b,e,f
13 a,b,f,e
14 a,b,c,d
15 a,b,e,f
```

In general, input data can contain an arbitrary number of traces, with each trace composed of at least one event.

Stage 0 – Reading, Analyzing, and Printing Input Data (10/20 marks)

The first version of your program should read an event log from input, analyze it, and print basic statistics about the event log to the output. The first thirteen lines from the listing below correspond to the output your program should generate for the `test0.txt` input file in Stage 0.

```
1  ==STAGE 0============================
2  Number of distinct events: 6
3  Number of distinct traces: 3
4  Total number of events: 60
5  Total number of traces: 15
6  Most frequent trace frequency: 7
7  abcd
8  a = 15
9  b = 15
10 c = 7
11 d = 7
12 e = 8
13 f = 8
14 ==STAGE 1============================
15          a    b    c    d    e    f
16     a    0   15    0    0    0    0
17     b    0    0    7    0    4    4
18     c    0    0    0    7    0    0
19     d    0    0    0    0    0    0
20     e    0    0    0    0    0    4
21     f    0    0    0    0    4    0
22 -------------------------------------
23 256 = SEQ(a,b)
24 Number of events removed: 15
25 c = 7
26 d = 7
27 e = 8
28 f = 8
29 256 = 15
30 =====================================
31          c    d    e    f  256
32     c    0    7    0    0    0
33     d    0    0    0    0    0
34     e    0    0    0    4    0
35     f    0    0    4    0    0
36   256    7    0    4    4    0
37 -------------------------------------
38 257 = SEQ(c,d)
39 Number of events removed: 7
40 e = 8
41 f = 8
42 256 = 15
43 257 = 7
44 ==STAGE 2============================
45          e    f  256  257
46     e    0    4    0    0
47     f    4    0    0    0
48   256    4    4    0    7
49   257    0    0    0    0
50 -------------------------------------
51 258 = CON(e,f)
52 Number of events removed: 8
53 256 = 15
54 257 = 7
55 258 = 8
56 =====================================
57        256  257  258
58   256    0    7    8
59   257    0    0    0
60   258    0    0    0
61 -------------------------------------
62 259 = CHC(257,258)
63 Number of events removed: 0
64 256 = 15
65 259 = 15
66 =====================================
67        256  259
68   256    0   15
69   259    0    0
70 -------------------------------------
71 260 = SEQ(256,259)
72 Number of events removed: 15
73 260 = 15
74 ==THE END============================
75
```

Lines 2 and 3 of the output report the number of distinct events and traces in the event log, respectively. The event log contains six distinct events, `a` through `f`, and three distinct traces, $\langle \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d} \rangle$, $\langle \mathtt{a}, \mathtt{b}, \mathtt{e}, \mathtt{f} \rangle$, and $\langle \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d} \rangle$. Lines 4 and 5 of the output report the total number of events and traces in the event log. As there are fifteen traces in the log, each composed of four events, in total, there are sixty events in the event log. Line 6 provides information about the number of times the most frequent trace appears in the event log. Trace $\langle \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d} \rangle$ is the most frequent trace; it appears seven times in the event log. The subsequent lines of the output should list all most frequent traces of the event log (one occurrence of each trace). If there are multiple traces with the same maximum frequency, they should all be listed, one per line, in lexicographical order, where uppercase characters precede all lowercase characters. Each trace should be printed as a string composed of characters that encode trace events in the order the events appear in that trace. As there is only one most frequent trace in the example input event log, it is printed on line 7; see the `abcd` string. In the following lines, see lines 8 to 13, your program should report all the events that appear in the event log, one per line, in ascending order of the corresponding ASCII codes, with information on how many times they appear in the event log.

You should not make assumptions about the maximum number of traces and events supplied. Use dynamic memory and data structures of your choice, for example, arrays or linked lists, to store the input event log.

The test input will always follow the proposed format.

Stage 1 – Identifying Sequences (16/20 marks)

Extend your program from Stage 0 to identify the most likely sequential (`SEQ`) patterns of actions based on the *directly follows* relationships between events in the event log. Two events are in the directly follows relationship if there is a trace in the log in which these events appear consecutively. First, your program should compute and print out all the directly follows relationships. Lines 15 to 21 of the output listing report the directly follows relationships between the events together with their *support*, that is, the number of times the events follow each other in the traces of the event log, given as a two dimensional matrix. For example, event `b` directly follows

event a once in each trace of the event log. Thus, its support is 15; see 15 in the third column of line 16 of the output; that is, $sup(\texttt{a}, \texttt{b}) = 15$. Each matrix column uses five characters, and each cell entry is right-justified.

Now, use the directly follows matrix to identify pairs of actions that most likely were arranged in sequences in the model that generated the event log. Given two events x and y, $x \neq y$, if $sup(x, y) > sup(y, x)$, then there is a chance that the model that generated the event log has actions x and y arranged in a sequence. To measure this chance, compute two values $pd(x, y) = (100 \times \texttt{abs}(sup(x, y) - sup(y, x)))/\texttt{max}(sup(x, y), sup(y, x))$ and $w(x, y) = \texttt{abs}(50 - pd(x, y)) \times \texttt{max}(sup(x, y), sup(y, x))$; use integer division to obtain the truncated result (for example, $15/4$ should result in 3). Here, $pd(x, y)$ is *percent difference* of $sup(x, y)$ and $sup(y, x)$ that reflects confidence that x is directly followed by y, while $w(x, y)$ is *weight* that further scales this confidence by the number of times y directly follows x; here, we assume that indeed $sup(x, y) > sup(y, x)$.

Each pair of events (u, v), $u \neq v$, for which $sup(u, v) > sup(v, u)$ and $pd(u, v) > 70$ is a candidate for defining a sequential pattern of actions. To select one most likely sequential pattern from the candidates, pick one with the highest weight and encountered first when traversing the directly follows matrix in *row-major order*.

Lines 23 to 29 of the output report the identified sequential pattern of actions and summarize information on the abstraction of the pattern in the event log. The most likely sequential pattern identified according to the proposed rules in the input event log is defined by the pair of events $(\texttt{a}, \texttt{b})$; note that $pd(\texttt{a}, \texttt{b}) = 100$ and $w(\texttt{a}, \texttt{b}) = 750$. Line 23 reports the identified pattern, assigning it the code of 256; note that the code of each subsequently identified pattern of actions should be incremented by one; see lines 38, 51, 62, and 71 of the output. Next, the identified pattern is abstracted in the event log to prepare it for discovering subsequent patterns. To abstract a pattern identified by a pair of events, rewrite each occurrence of one of these events in the traces of the event log with the abstract event identified by the code of the pattern. Then, in every trace, replace each subsequence composed of only new abstract events with one such abstract event without changing the order of the other events. For example, the abstraction of pattern 256 in the input event log leads to the below event log; keep the resulting event log in memory and do *not* write it to a file.

```
1  256,c,d
2  256,f,e
3  256,f,e
4  256,f,e
5  256,c,d
6  256,c,d
7  256,c,d
8  256,c,d
9  256,c,d
10 256,e,f
11 256,e,f
12 256,e,f
13 256,f,e
14 256,c,d
15 256,e,f
```

Line 24 of the output reports the number of events removed from the event log as the result of abstracting pattern 256. One event was removed from every trace to result in 15 removed events. Finally, lines 25 to 29 report all distinct events, both original and abstract, and their frequencies in the resulting event log, in the ascending order of their codes, where codes of the original events are ASCII codes of the corresponding characters. Lines 31 to 43 use the same template to report information on the next identified sequential pattern of actions.

The output of two consecutive patterns is separated by a sequence of "=" characters (see, for example, line 30), and the directly follows matrix is separated from the subsequent information on the identified pattern with a sequence of "-" characters (see, for instance, lines 22 and 37). The identification of patterns should continue from the resulting log until no pattern can be discovered based on the proposed rules. NB: In Stage 1, only patterns based on two events from the original input log, that is, *non-abstract* events, should be discovered.

Stage 2 – Identifying Concurrency and Choices (20/20 marks)

Extend your program from Stage 1 to allow discovering concurrency (`CON`) and choice (`CHC`) patterns of actions. Stage 2 of your program should continue from the state reached in Stage 1.

Two actions that can be executed concurrently are independent and, thus, can occur in a trace in any order. Given two events x and y, $x \neq y$, if $sup(x, y) > 0$ and $sup(y, x) > 0$, there is a chance that the model that generated the event log supports their concurrent execution. Each pair of events (u, v), $u \neq v$, $sup(u, v) > 0$, $sup(v, u) > 0$, for which $pd(u, v) < 30$, is a candidate for defining a concurrency pattern of actions. The weight of a concurrency pattern should be computed the same way as the weight of a sequential pattern.

Two events x and y, $x \neq y$, that do not, or rarely, occur together in traces of the log were likely triggered by actions that are in an exclusive choice pattern in the model. We accept events (u, v), $u \neq v$, as a candidate choice pattern if $\texttt{max}(sup(u, v), sup(v, u)) \leq N/100$, where N is the number of all events in all the traces of the event log; again, use integer division. The weight $w(u, v)$ of a choice pattern should be computed as $N \times 100$.

In Stage 2, to prioritize the discovery of concurrency and sequential patterns, the weight of each sequential pattern defined over two non-abstract events from the original input event log and the weight of any concurrency pattern should be additionally multiplied by 100.

In each iteration, your program should attempt to discover one pattern of actions. Each pair of distinct events should be checked for defining a sequential, concurrency, or choice pattern. Check a pair of events for being a choice pattern candidate first and, if confirmed, do not check it for defining other patterns. Among all identified candidate patterns, the one with the highest weight should be selected. If multiple candidates with the same weight exist, the one identified first when traversing the directly follows matrix in *row-major order* should be chosen. When proceeding to the next iteration, abstract the pattern in the log by following the rules from Stage 1.

Lines 45 to 73 of the output report three further discovered patterns using the same output template as in Stage 1. The patterns reported on lines 23, 38, 51, 62, and 71 of the output induce the hierarchy of patterns `SEQ(SEQ(a,b),CHC(SEQ(c,d),CON(e,f)))`, which rediscovers the model in Figure 1. Note that, in general, the rules proposed in this assignment do not guarantee the rediscoverability of the model. Your program should terminate once no further patterns can be discovered.

Important...

The output your program should generate for the `test0.txt` input file is provided in the `test0-out.txt` output file. Two further test inputs and outputs are provided. The outputs generated by your program should be *exactly the same* as the sample outputs for the corresponding inputs. Use `malloc` and dynamic data structures of your choice to read and store input logs. Before your program terminates, all the `malloc`'ed memory must be `free`'d.

Boring But Important...

This project is worth 20% of your final mark, and is due at **6:00pm on Friday 14 October**.

Submissions that are made after the 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 obtain a letter from them that describes your illness and their recommendation for treatment. Suitable documentation should be attached to **all** extension requests.

You need to submit your program for assessment via **Gradescope (Assignment 2)**. Submission is **not possible through Grok**. Multiple submissions may be made; only the last submission that you make before the deadline will be marked. If you make any late submission at all, your on-time submissions will be ignored, and if you have not been granted an extension, the late penalty will be applied.

As you can submit your program multiple times, test your program in the test environment early. The compilation in the Gradescope environment may differ from Grok and your laptop environment. Note that Gradescope may overload and even fail on the assignment due date when many students submit their programs, and if that does happen, it will not be a basis for extension requests. *Plan to start early and to finish early!!*

A rubric explaining the marking expectations is linked from the LMS. Marks will be available on the LMS approximately two weeks after submissions close, and feedback will be made available via Gradescope.

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** have any "accidents" that allow others to access your work; 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.

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.

© The University of Melbourne, 2022. The project was prepared by Artem Polyvyanyy, `artem.polyvyanyy@unimelb.edu.au`.