Process Mining: The ideas behind the alpha algorithm

A Process Engineering Lecture

Mining process from logs

Given: Set of traces

Find: Process that generates these

Process mining algorithms: create a process from set of traces. The entire set is to be considered. For example the *alpha* algorithm and many successors.

We shall discuss ideas about the alpha algorithm in this lecture.

For example: {abc, acb, ae} is one such trace.

Multiple traces are obtained by running the process multiple times over several instances.

Applications of Process mining algorithms

Discovery: Discover the process if documentation is incomplete or not available

Conformance: Cross verify whether the discovered process is the same as the actual one that is claimed to have been implemented

Monitoring: Continuous monitoring of traces to detect anomalies

Other applications of mining in general: process optimization, simplification, strengthening, performance improvement

What all can we discover from traces

Sequences

XOR Branching

XOR Join

AND Brancing

AND Join

Loops also may get factored in

Idea 1: AND Branching

Trace fragments   bc, cb

This means AND branching

Idea 1: AND Branching

Trace fragments   bc, cb

This means AND branching

Example: set {ae, abd, adb}

Has one and branching. Can you find it?

Idea 2: XOR Split

Trace fragments ae, ab

This means XOR branching

Example: set {ae, abd, adb}

Has an XOR branching. Can you find it?

How to find out these connections between transitions?

Direct before relation a>b

If ab is in trace

Apply direct before relation to find **direct before set** in trace set

{t1t2t3t4t5, t1t19} ?

How to find out these connections between transitions?

Direct before relation a>b

If ab is in trace

Apply direct before relation to find **direct before set** in trace set

{t1t2t3t4t5, t1t19} ?

⇒ { t1>t2, t2>t3, t3>t4, t4>t5, t1>t19 }

How to find out these connections between transitions?

causality relation a->b

If ab is in trace set, but ba is not!

Which is to say, a>b but not(b>a) are satisfied

Apply causality relation to find **the causality set** in trace set

{abc, acb} ?

How to find out these connections between transitions?

causality relation a->b

If ab is in trace set, but ba is not!

Which is to say, a>b but not(b>a) are satisfied

Apply causality relation to find **the causality set** in trace set

{abc, acb} ?

⇒ { a->b, a->c }

How to find out these connections between transitions?

Independence relation a#b

neither ab is in trace set, not ba!

Which is to say, not(a>b) and not(b>a) are satisfied

Apply independence relation to find **the independence set** in trace set

{abc, acb} ?

How to find out these connections between transitions?

Parallel relation a#b

Both ab and ba are in trace set!

Which is to say, (a>b) and (b>a) are both satisfied

Apply parallel relation to find **the parallel set** in trace set

{abc, acb} ?

How to find out these connections between transitions?

Independence relation a#b

neither ab is in trace set, not ba!

Which is to say, not(a>b) and not(b>a) are satisfied

Apply independence relation to find **the independence set** in trace set

{abc, acb} ?

⇒ {a#c}??? Is incorrect because we have a>c which violates the above condition

⇒ {a#a,b#b,c#c} ?

is correct because we do not have x>y,y>x for every x#y in this set

How to find out these connections between transitions?

Parallel relation a||b

a>b, b>a are in trace set

Apply parallel relation to find **the parallel set** in trace set

{abc, acb} ?

How to find out these connections between transitions?

Parallel relation a||b

a>b, b>a are both in trace set

Apply parallel relation to find **the parallel set** in trace set

{abc, acb} ?

⇒ { a||b, a||c, c||a } ? not correct since x>y and y>x are not both satisfied for all cases in the set

=> {b||c}

FOOTprint matrix

The matrix of relationships across transitions

> need not be considered, since it is used to infer the following:

->

#

||

Exercise: construct the footprint matrix

Given trace set

(abcd, acbd, aed}

Construct > set, -> set, # set and || set

And then represent these sets in the form a matrix over the transitions a,b,c,d,e

The matrix for set (abcd, acbd, aed}

<table>
<tr><th></th><th>a</th><th>b</th><th>c</th><th>d</th><th>e</th></tr>
<tr><td>a</td><td>#</td><td>-></td><td>-></td><td>#</td><td>-></td></tr>
<tr><td>b</td><td><-</td><td>#</td><td>||</td><td>-></td><td>#</td></tr>
<tr><td>c</td><td><-</td><td>||</td><td>#</td><td>-></td><td>||</td></tr>
<tr><td>d</td><td>#</td><td><-</td><td><-</td><td>#</td><td><-</td></tr>
<tr><td>e</td><td><-</td><td>#</td><td>#</td><td>-></td><td>#</td></tr>
</table>

Constructing the net

```
a->b  : sequence
a->b, a->c, b#c : XOR split
a->c, b->c, a#b : XOR join
a->b, a->c, b||c : AND split
a->c, b->c, a||b : AND join

But the alpha algorithm is defective
Find flaws in it
```