Genetic Mining und Heuristic Mining

Jannik Arndt

Deparment für Informatik
Carl von Ossietzky Universität Oldenburg
26111 Oldenburg
jannik.arndt@uni-oldenburg.de

Abstract: Das *Process Discovery* ist ein Teilbereich des *Process Mining* und hat das Ziel, Prozessmodelle aus gegebenen *Event Logs* zu erstellen. Einfache Algorithmen haben jedoch Probleme mit Noise, Infrequent Activities, Loops, Splits, Joins und Long Distance Dependencies. Diese Herausforderungen werden von den beiden Discovery-Ansätze *Heuristic Mining* und *Genetic Mining* größtenteils gelöst: Das Heuristic Mining erstellt einen Abhängigkeitsgraphen und entdeckt Parallelitäten, kann von sich aus aber keine Long Distance Dependencies lösen. Das Genetic Mining nutzt die Konzepte der Rekombination, Mutation und Selektion, um über eine globale Suche das passende Prozessmodell zu finden, stellt allerdings neue Herausforderungen, wie eine aussagekräftige Fitnessfunktion und eine geeignete Startpopulation. Letztenendes können die Schwächen des Genetic Mining durch Ansätze aus dem Heuristic Mining ausgeglichen werden.

1 Einleitung

Process Mining unterteilt sich in die drei Themenfelder *Process Discovery*, *Conformance Checking* und *Model Enhancement*. Diese Arbeit ist im Bereich des *Discovery* eingeordnet, der sich damit beschäftigt, aus gegebenen Daten in Form von *Event Logs* ein Prozessmodell zu erstellen, das die Abläufe in diesen Daten wiedergibt. Ein Ablauf wird *Trace* genannt und besteht aus mehreren *Events*, die alle zu demselben *Case* gehören. Ein gutes Modell zeichnet sich dadurch aus, dass es alle vorkommenden Traces darstellen kann, aber keine zusätzlichen zulässt. [vdAI12]

Zunächst werden die Herausforderungen betrachtet, die Algorithmen bewältigen müssen, um ein gutes Modell zu erstellen. Danach werden in Abschnitt 3 und 4 der heuristische Algorithmus HeuristicsMiner von [WvdAdM06] und das auf evolutionären Algorithmen basierende Genetic Mining nach [dMWvdA06] vorgestellt. In Abschnitt 5 werden die beiden Ansätze miteinander verglichen.

2 Herausforderungen bei Process Discovery-Algorithmen

Das Erstellen eines Prozessmodells aus einem gegebenen Event Log ist, solange dieser einfach genug (siehe 2.6), vollständig (d. h. dass alle Traces vorhanden sind, siehe 2.2) und fehlerfrei (siehe 2.1) ist, für einfache Algorithmen wie den Alpha-Algorithmus [dMvDvdAW, vdAdMM04] kein Problem. Auf realen Daten kann es dabei jedoch zu Schwierigkeiten kommen, die von einfachen Algorithmen nicht oder nicht zufriedenstellend gelöst werden können:

2.1 Noise

Das größte Problem, das Event Logs einem Algorithmus bereiten, sind fehlerhafte Einträge, sogenannter *Noise*. Bei realen Daten kann nicht davon ausgegangen werden, dass Daten *vollständig*, *minimal* und in der richtigen *Reihenfolge* sind. Es können also Traces, die nur sehr selten auftreten, in dem betrachteten Event Log gar nicht vorhanden sein, oder aber es sind Traces vorhanden, die nicht auftreten *sollten*, zum Beispiel durch Eingabefehler aber im Log auftauchen. Auch die Reihenfolge ist für das Prozessmodell wichtig, kann aber ebenfalls durch Eingabefehler verfälscht sein. Algorithmen müssen also eine gewisse Fehlertoleranz bieten. Der Alpha-Algorithmus hingegen definiert klare Relationen: Um zum Beispiel $a \rightarrow_L b$ zu widerlegen, genügt *ein* (fehlerhaftes) Gegenbeispiel, in dem $b >_L a$ gilt.

2.2 Infrequent Activities

Sobald Algorithmen beginnen, mit *Noise* tolaranter umzugehen, taucht hier das zweite Problem auf: Nicht alle Events treten mit der gleichen Häufigkeit auf, es kann durchaus Ausnahmen geben, also *infrequent Activities*, die im Prozessmodell jedoch vorgesehen sind und enthalten sein sollten. Diese korrekten aber seltenen Traces von fehlerhaften Traces zu unterscheiden ist jedoch extrem schwierig, da sie leicht mit derselben Häufigkeit auftreten können.

Anmerkung: In der Literatur findet man diese Eigenschaft auch unter dem Namen *low frequent activities*.

2.3 Verbotenen Traces sind unbekannt

Im Machine Learning kann es von großem Vorteil sein, ein Modell mit positiven *und* negativen Daten zu traineren, sprich Traces, die verboten sind. Diese könnten fehlerhafte positive Fälle widerlegen. Im Fall des Process Mining besteht die Datengrundlage jedoch ausschließlich aus Traces, die irgendwann tatsächlich vorgekommen sind. Daraus folgen

oft Modelle, die nicht *minimal* sind, und also auch Traces zulassen, die nicht vorkommen sollten.

2.4 Loops

In Prozessen können verschiedene Arten der Schleifen auftreten: Loops der Länge eins (auch *Short Loops*) wiederholen einzelne Events beliebig oft, wodurch Traces wie ABC, $ABBC$, $ABBBC$, $\ldots$ entstehen. Abbildung 1 zeigt ein Petrinetz, das solche Traces ausgeben kann, wenn nach B die *unsichtbare* Transition $\#1$ genommen wird. Loops der Länge zwei haben Traces im Format $ABCD$, $ABCBCD$, $ABCBCBCD$, $\ldots$, in der Abbildung wird dies durch die Transition $\#2$ realisiert.

Die Schwierigkeit, Loops zu erkennen, liegt darin, dass in Petrinetzen eine zusätzliche, *unsichtbare* Transition benötigt wird (in der Abbildung $\#1$ und $\#2$). Unsichtbar ist sie deshalb, weil sie im Event Log nicht auftritt, und genau das sorgt dafür, dass der Algorithmus eine spezielle Behandlung für Loops benötigt.

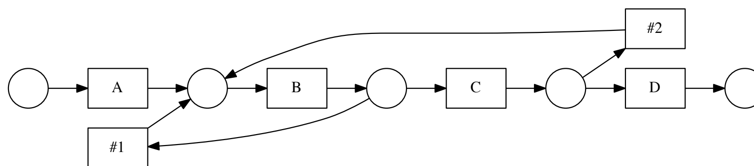

Abbildung 1: Ein Petrinetz mit einer Schleife der Länge 1 (B) und einer Schleife der Länge 2 (BC).

2.5 Splits und Joins

Prozessmodelle kennen zwei Konstrukte für Parallelität: AND und XOR. Dabei kann jeweils eins oder mehrere Events parallel zu einem oder mehreren anderen auftreten. Abbildung 2 zeigt die parallelen Events B, C und D. Es können dabei *entweder* B *und* C *oder* D ausgeführt werden. Genaugenommen handelt es sich jedoch nur um Pseudo-Parallelität, da die Events im Event Log geordnet vorkommen.

In der Darstellung von Events in Petrinetzen unterscheiden sich die beiden Konstrukte wie folgt: Ein XOR-Split wie er nach A stattfindet wird durch mehrere ausgehende Kanten von einer Stelle (rund) dargestellt. Ein Token, der auf dieser Stelle liegt, kann also nur genau einen der weiterführenden Wege nehmen. AND-Splits hingegen werden durch mehrere ausgehende Kanten von einer Transition (eckig) dargestellt, wie vor B und C zu sehen ist. Transitionen machen also aus einem Token mehrere. AND-Splits fügen dem Modell *unsichtbare* Transitionen (siehe Loops) hinzu, die nicht im Event Log auftauchen.

Joins funktionieren ähnlich: Bei AND-Joins (in der Abbildung 2 nach B und C) laufen mehrere Kanten in eine Transition ein. Die Transition ist erst aktiviert (kann also erst

genommen werden), wenn auf *allen* Stellen davor mindestens ein Token liegt. XOR-Joins hingegen sind mehrere Kanten, die in eine Stelle laufen, hier reicht es, wenn über *eine* Kante ein Token kommt. Genaugenommen wird beim Join also nicht XOR sondern OR modelliert, der Symmetrie halber spricht man aber von XOR-Joins.

Es ist zwar möglich, in einem Petrinetz nach einem AND-Split einen XOR-Join zu setzen, dann würden aber am Ende *zwei* Token ankommen, was in dem Kontext, für den das Process Mining die Netze benutzt, nicht erwünscht ist. Ein AND-Join nach einem XOR-Split führt zu einer Verklemmung, da zwei Token benötigt werden, aber nur einer verfügbar ist.

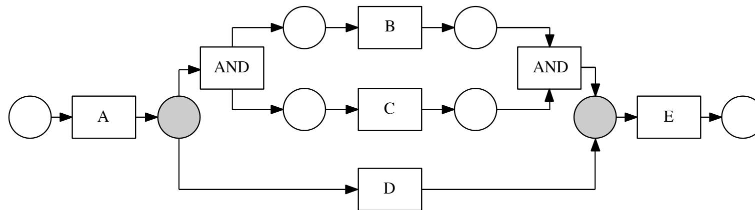

Abbildung 2: Ein Petrinetz mit AND-Splits und -Joins und XOR-Splits und -Joins (grau).

2.6 Long Distance Dependencies

Long Distance Dependencies besagt, dass in einem Teilbereich des Netzes zwar eine Wahl besteht (in Abbildung 3 ob nach D nun E oder F gewählt wird), durch die größere Struktur des Netzes diese Wahl aber schon vorbestimmt wurde: Jenachdem, ob vorher B oder C gewählt wurde, liegt nun entweder auf der Stelle vor E *oder* auf der Stelle vor F ein Token. Diese Entscheidung hängt von einem anderen Teil des Netzes ab, das u. U. 'weit entfernt' liegt.

Solche Abhängigkeiten können in der Praxis sehr leicht auftreten, sind beim *Discovery*-Prozess jedoch nur mit viel Aufwand erkennbar.

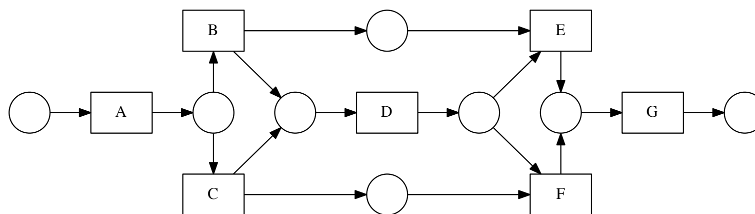

Abbildung 3: Ein Petrinetz mit Long Distance Dependencies (die Wahl nach D hängt davon ab, ob früher B oder C gewählt wurde).

3 Heuristic Mining

Heuristische Methoden machen sich das Wissen über die in Abschnitt 2 aufgeführten Schwierigkeiten zu nutze. Durch dedizierte Teile im Algorithmus und Erfahrungswerte können viele der Herausforderungen so umgangen bzw. gelöst werden.

Im Folgenden wird der *HeuristicsMiner* nach [WvdAdM06] als Beispiel genommen und genauer beschrieben. Er besteht aus drei Schritten, die in den Abschnitten 3.1, 3.2 und 3.3 erklärt werden:

1. Einen *Abhängigkeitsgraphen* (dependency graph) erstellen
2. Für parallele Events die Art der *Parallelität* bestimmen
3. Nach *Long Distance Dependencies* suchen

3.1 Abhängigkeitsgraph

Als erstes Problem werden *Noise* und *Infrequent Activities* aufgegriffen. Hierfür wird eine neue Funktion $\Rightarrow_{\mathcal{W}} \rightarrow (-1, 1)$ eingeführt, welche die Abhängigkeit zweier Events ausdrückt:

$$a \Rightarrow_{\mathcal{W}} b = \frac{|a >_{\mathcal{W}} b| - |b >_{\mathcal{W}} a|}{|a >_{\mathcal{W}} b| + |b >_{\mathcal{W}} a| + 1}$$

mit

$\mathcal{W}$ als Event Log über der Menge der Traces $\mathcal{T} \subseteq \mathcal{E}^*$
$a, b \in \mathcal{E}$ als Events und
$|a >_{\mathcal{W}} b|$ die Häufigkeit, mit der b auf a in $\mathcal{W}$ folgt.

Der berechnete Wert zwischen -1 und 1 gibt für zwei Events an, wie stark sie zusammenhängen. Werte nahe der Null bedeuten, dass zwei Events unabhängig sind, Werte nahe 1 deuten auf einen Zusammenhang hin. Negative Werte zeigen eine Abhängigkeit in umgekehrter Reihenfolge, es gilt $a \Rightarrow_{\mathcal{W}} b = -1 \cdot (b \Rightarrow_{\mathcal{W}} a)$ (siehe Tabelle 1).

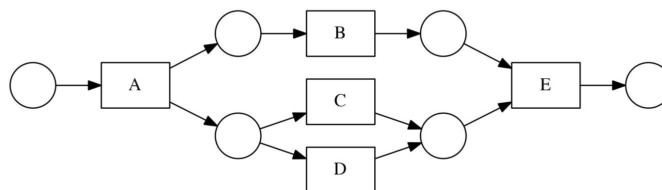

Abbildung 4: Ein Petrinetz mit AND-Split bei A, XOR-Split bei C und D und AND-Join bei E.

$\Rightarrow_\mathcal{W}$	**A**	**B**	**C**	**D**	**E**
A	0	0.941	0.889	0.857	0
B	-0.941	0	0.063	0.071	0.813
C	-0.889	-0.063	0	0.5	0.889
D	-0.857	-0.071	-0.5	0	0.889
E	0	-0.813	-0.889	-0.889	0

Tabelle 1: Abhängigkeitsmatrix für Events A–E.

Als Beispiel wird das Netz in Abbildung 4 betrachtet, für das folgender Event Log auftreten kann:

$$\mathcal{W} = [ABCE^7, ABDE^7, ACBE^7, ADBE^6, ABE, ABCDE, ACEB]$$

Die vier ersten Traces sind korrekt und kommen mehrfach vor, die letzten drei sind fehlerhaft und Einzelfälle. Der Noise-Anteil beträgt damit 10%. Hieraus lässt sich die Abhängigkeitsmatrix in Tabelle 1 für alle Events berechnen.

Den Startpunkt des Netzes (hier A) kann man sehr einfach an der Spalte erkennen, die keinen positiven Wert besitzt, also nur Nachfolger hat.

Der heuristische Anteil des Algorithmus besteht nun darin, Schwellenwerte (*thresholds*) dafür festzulegen, wann zwei Events direkt aufeinander folgen. Betrachtet man die Nachfolger von A, so zeigt die erste Zeile der Tabelle eine hohe Abhängigkeit (> 0.8) zu B, C und D.

In der zweiten Zeile erkennt man für das Event B den Vorgänger A am negativen Wert und C, D und E als mögliche Nachfolger, C und D jedoch mit einem sehr niedrigen Wert (< 0.1). Hier sorgt ein ein der Erfahrung gewählter Schwellenwert (zwischen 0.1 und 0.8) dafür, dass nur E als Nachfolger akzeptiert wird.

In der dritten Zeile sieht man eine Abhängigkeit von C zu D vom Grad 0.5. Dieser sollte eigentlich 0 betragen, weil die beiden Events durch ein XOR verbunden sind, der fehlerhafte vorletzte Eintrag im Event Log sorgt jedoch dafür, dass der Algorithmus eine Abhängigkeit sieht. Um diesen Fehler zu beheben, muss der Schwellenwert also über 0.5 liegen. Dieses Wissen stammt allerdings *ausschließlich* aus dem Originalnetz in Abbildung 4.

Erstellt man den Abhängigkeitsgraphen in Abbildung 5, so ist dieser mit Ausnahme der gestrichelten Kanten (die unterhalb des Schwellenwertes $|0.501|$ liegen) isomorph zu den Transitionen des Ausgangsnetzes.

3.2 Parallelität

Der in Abschnitt 3.1 erstellte Abhängigkeitsgraph ist noch kein Petrinetz, da er keine Stellen (rund) besitzt, und kann somit auch noch kein Prozessmodell darstellen. Insbesondere hat er keine AND- und XOR-Splits und -Joins. Diese tauchen im Event Log auch nicht explizit

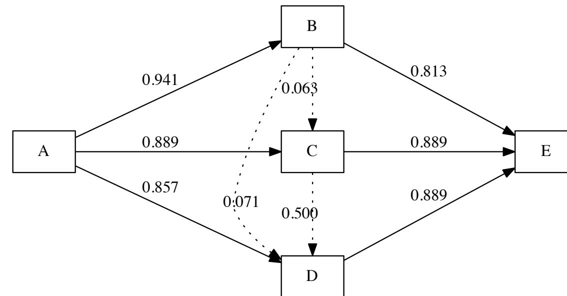

Abbildung 5: Der Abhängigkeitsgraph zur Matrix aus Tabelle 1.

auf (siehe Abschnitt 2.5). Den richtigen Split zu finden, ist jedoch nicht schwierig: für je zwei Folgetransitionen einer Transition gilt entweder:

- im weiteren Verlauf tauchen *beide* auf (⇒ AND-Split) oder
- es taucht nur *einer* der beiden auf (⇒ XOR-Split).

Am Beispiel in Abbildung 4 sieht man nach A einen AND-Split, daher tauchen im Log die Teil-Traces AB und AC bzw. AD auf. Vor C und D hingegen ist ein XOR-Split, es kann nur eins der beiden Events gewählt werden. Daher tauchen die Teil-Traces CD und DC *nie* auf (mit Ausnahme des fehlerhaften Trace).

Formal bedeutet das:

$$a \leadsto_{\mathcal{W}} (b \wedge c) = \frac{|b >_{\mathcal{W}} c| + |c >_{\mathcal{W}} b|}{|a >_{\mathcal{W}} b| + |a >_{\mathcal{W}} c| + 1}$$

Es werden also alle Vorkommnisse, in denen c auf b oder b auf c folgt, ins Verhältnis zur Anzahl Vorkommnisse von b und c gesetzt. Dabei werden statt b und c die Kombinationen ab und ac betrachtet, damit Loops das Ergebnis nicht verfälschen. Die Funktion bildet ab auf $[0, 1)$, wobei Werte < 0.1 eine XOR-Beziehung von b und c bedeuten, während Werte > 0.1 auf eine AND-Beziehung schließen lassen. Der empirische Schwellenwert von 0.1 stammt aus [WvdAdM06].

Es ist zu beachten, dass die Formel *immer* eine Entscheidung für XOR oder AND ausgibt, daher ist es nicht sinnvoll, sie auf sequentielle Transitionen anzuwenden. Aus diesem Grund muss dieser Schritt *nach* dem Abhängigkeitsgraphen durchgeführt werden.

Am Beispiel wird diese Formel nun auf die Transition A angewandt, weil diese als einzige mehrere Folgetransitionen besitzt:

$$A \leadsto_{\mathcal{W}} (B \wedge C) = \frac{8 + 7}{16 + 8 + 1} = 0.6$$

$$A \leadsto_{\mathcal{W}} (B \wedge D) = \frac{7 + 6}{16 + 6 + 1} \approx 0.565$$

$$A \leadsto_{\mathcal{W}} (C \wedge D) = \frac{0 + 1}{8 + 6 + 1} \approx 0.067$$

Die Werte zeigen, dass B sowohl zu C ($0.6 > 0.1$) als auch zu D ($0.565 > 0.1$) in einer AND-Beziehung steht, C und D untereinander hingegen in einer XOR-Beziehung stehen ($0.067 < 0.1$). Mit diesem Wissen kann das Petrinetz in Abbildung 4 vollständig wiederhergestellt werden.

3.3 Long Distance Dependencies

Das Beispielnetz besitzt keine Long Distance Dependencies wie sie in Abschnitt 2.6 vorgestellt wurden. Tatsächlich ist der *HeuristicsMiner* auch noch nicht in der Lage, diese korrekt umzusetzen. In [WWS06] gibt es einen Ansatz, der auf dem α^+-Algorithmus aufbaut und die Abhängigkeiten (dort *Implicit Dependencies* genannt) mit Hilfe von drei Theoremen nachträglich in den Modellen korrigieren kann. Da sich die Vorraussetzungen allerdings sehr stark von den hier gegebenen unterscheiden, wird auf diesen Ansatz nicht weiter eingegangen. Stattdessen wird im nächsten Abschnitt 4 das *Genetic Mining* vorgestellt werden, das Long Distance Dependencies ebenfalls auffinden kann.

4 Genetic Mining

Neben neuronalen Netzen und Fuzzy Logik sind evolutionäre bzw. genetische Algorithmen eine der drei Grundsäulen der Computational Intelligence. Sie arbeiten auf einer Grundmenge von Lösungskandidaten, der sogenannten *Population*. Aus dieser wird in jeder Iteration des Algorithmus durch die Operatoren *Rekombination* (auch als *crossover* bekannt) und *Mutation* eine neue Menge von Kandidaten generiert. Man spricht hierbei auch von Eltern- und Kindgeneration. Aus dieser neuen Menge werden durch *Selektion* nun die fittesten Individuen ausgewählt. Diese bilden (in einigen Fällen zusammen mit den fittesten Eltern) die neue Population. [Kra09, Kra12]

Der entscheidende Vorteil dieser Vorgehensweise ist, dass sich damit auch *Black-Box-Probleme* lösen lassen: Hierbei ist die Funktion, die das eigentliche Problem beschreibt, *nicht* bekannt und kann daher auch nicht algebraisch gelöst werden. Auch das Optimum ist in der Regel *nicht* bekannt. Durch die stochastische Vorgehensweise kann die Funktion trotzdem optimiert werden, solange es eine Fitnessfunktion gibt, mit der Individuen bewertet und verglichen werden können.

4.1 Evolution im Process Mining

Will man nun die Prinzipien der Evolutionären Algorithmen auf das Process Mining anwenden, so stellen sich zunächst die Fragen, was eigentlich eine Population ist, wie Rekombinations- und Mutationsoperatoren darauf arbeiten und vor allem wie die Fitness berechnet wird. In [dMWvdA06] finden sich alle drei Zuordnungen, die hier kurz vorgestellt werden.

4.1.1 Population

Die Population des Evolutionären Algorithmus besteht aus Modellen, die Traces darstellen können. Idealerweise sollten alle im Log auftretenden Events auch im Modell als Transitionen vorhanden sein. Um eine Startpopulation zu erzeugen, werden die Transitionen im Grundalgorithmus durch Stellen, AND- und XOR-Splits und -Joins *zufällig* verbunden. Dieser Algorithmus kann jedoch verbessert werden, da der Event Log bekannt ist und Methoden aus dem heuristischen Mining, wie zum Beispiel die Abhängigkeitsmatrix, genutzt werden können, um Startkandidaten mit einer hohen Ausgangsfitness zu erzeugen.

4.1.2 Selektion, Rekombination und Mutation

Um sich einer optimalen Lösung anzunähern, stellt [dMWvdA06] folgende drei Operatoren vor:

Selektion Als Selektionsoperator wird *Elitismus* verwand, auch bekannt als *Plus-Selektion*. Hierbei werden aus der Elterngeneration die besten Individuen mit in die nächste Generation übernommen. Dies kann außerdem durch sogenannte *Tournament-Selektion* geschehen, wobei immer zufällig eine Gruppe Eltern ausgewählt wird und aus dieser das beste Individuum übernommen wird.

Rekombination Rekombination zielt darauf ab, Lösungskandidaten an sogenannten *crossover points* in Teilstücke zu trennen und dann zu neuen Kandidaten zu kombinieren. Dieser Operator würde also besonders dann sehr hilfreich sein, wenn eine Fitness auf Teilstücken feststellbar wäre. Abbildung 6 zeigt ein einfaches Beispiel für eine Rekombination zweier Modelle.

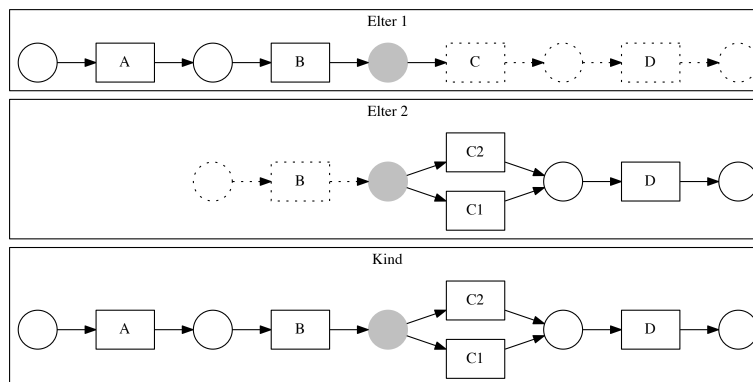

Abbildung 6: Ein Beispiel für eine Rekombination zwischen zwei Eltern. Die graue Stelle markiert den *crossover point*, die nicht-gestrichelten Teilstücke werden an das Kind weitervererbt.

Mutation Mutation verändert einen Lösungskandidaten, in kontinuierlichen Problemräumen meist durch Multiplikation mit einer gaussverteilten Zufallszahl. In diskreten Räumen wie dem *travelling salesperson problem* oder Petrinetzen gibt es die Inversionsmutation, die Elemente vertauscht. Für jede Transition in einem Modell wird also mit einer gegebenen Wahrscheinlichkeit (der *Mutationsrate*) entschieden, ob sie mit einer anderen vertauscht wird. Dies funktioniert besonders dann, wenn im Modell alle notwendigen Transitionen bereits vorhanden sind.

Für das Process Mining können im Zuge der Mutation außerdem Transitionen an beliebigen Stellen hinzugefügt oder entfernt werden. Diese relativ drastische Änderung erlaubt es, in einem Schritt zu 'weit entfernten' Lösungskandidaten zu wandern, was auch *globale Suche* (im Vergleich zur *lokalen Suche* um einen guten Lösungskandidaten herum) genannt wird.

4.1.3 Fitnessfunktion

Die Fitnessfunktion muss nicht nur bewerten, ob ein Trace im Modell dargestellt wird oder nicht, sondern sollte auch Abstufungen abbilden können, damit der Algorithmus das Modell in die richtige Richtung entwickelt.

In [dMWvdA06] wird vorgeschlagen, bei Fehlern nicht aufzuhören, sondern sie zu ignorieren und den Trace weiter auszuführen, dabei aber zu zählen, wie oft solche Fehler auftreten. Praktisch bedeutet das, dass wenn ein Token fehlt um eine Transition zu aktivieren, dieser Token bereitgestellt wird. Fehlt eine ganze Transition, so wird das Event übersprungen. Am Ende des Traces werden die zusätzlich benötigten sowie alle noch im Netz verbleibenden Token und die Anzahl hinzugefügter Transitionen gezählt.

Fehlende Token deuten dabei meist darauf hin, dass AND-Splits *fehlen* oder statt eines AND-Joins ein XOR-Join besser passen würde. Übrige Token zeigen genau das Gegenteil, nämlich *zu viele* AND-Splits und *zu viele* XOR-Joins.

Formal lässt sich die Fitnessfunktion $\mathcal{F} \rightarrow (-\infty, 1]$ (mit optimaler Fitness 1) definieren als:

$$\mathcal{F}(\mathcal{W}, \mathcal{M}) = \frac{|\sigma_{parsed} \in \mathcal{W}| - G(\mathcal{W}, \mathcal{M})}{|\sigma \in \mathcal{W}|}$$

mit

- $\mathcal{W}$ als Event Log über der Menge der Traces $\mathcal{T} \subseteq \mathcal{E}^*$
- Modell $\mathcal{M}$
- Anzahl Traces im Event Log $|\sigma \in \mathcal{W}|$,
- Anzahl Traces im Event Log die vom Modell dargestellt werden $|\sigma_{parsed} \in \mathcal{W}|$
- und der Straffunktion $G(\mathcal{W}, \mathcal{M}) = \frac{|m_{missing}|}{|\sigma \in \mathcal{W}| - |\sigma_{missing}| + 1} + \frac{|m_{spare}|}{|\sigma \in \mathcal{W}| - |\sigma_{spare}| + 1}$ mit
  - Anzahl fehlender Token für den Event Log im Modell $|m_{missing}|$
  - Anzahl Traces im Event Log mit fehlenden Token im Modell $|\sigma_{missing}|$

– Anzahl übriger Token für den Event Log im Modell $|m_{spare}|$
– Anzahl Traces im Event Log mit übrigen Token im Modell $|\sigma_{spare}|$

4.2 Abbruchkriterium und Parameter

Im Gegensatz zu heuristischen Algorithmen, die nach einer festen Anzahl von Berechnungen den bestmöglichen Wert erreichen, sind Evolutionäre Algorithmen stochastische Verfahren, die ein Abbruchkriterium benötigen, um zu terminieren. Für gewöhnlich ist die optimale Fitness nicht bekannt, in diesem Fall jedoch schon: Sind *alle* Traces im Modell enthalten, so beträgt die Fitness $= 1$.

Da allerdings aufgrund des im Event Log vorhandenen Noise nicht immer ein Modell, das *alle* Traces akzeptiert, das Ziel ist, sondern in der Regel ein Modell, das *die meisten* akzeptiert, ist es sinnvoll, andere Abbruchbedingungen zu definieren. Beliebte Beispiele sind zeitliche Begrenzungen, Anzahl der Generationen oder Stagnation der Fitness für eine bestimmte Anzahl an Generationen. Die genauen Parameter sind problemabhängig.

In [dMWvdA06] finden sich Tests auf Noise-freien Event Logs mit 1000 Traces mit 5, 7, 8, 12 und 22 Events. Hierfür haben die Autoren eine Populationsgröße von 500 und eine Generationenzahl von maximal 100 000 gewählt. Im Durchschnitt wurde immer eine Fitness sehr nah am Maximum 1 erreicht. Die Variante des Algorithmus *ohne* heuristische Erstellung der Anfangspopulation hat allerdings *nie* das Originalnetz gefunden, während die Variante *mit* Heuristik dies in fast der Hälfte aller Fälle erreicht hat.

5 Zusammenfassung

Das *Process Discovery* ist der größte Bereich im *Process Mining*, und daher sind effiziente Algorithmen sehr wichtig. Der Alpha-Algorithmus hat bei einigen komplexeren Strukturen, die in Abschnitt 2 erläutert wurden, Probleme. In diesem Paper wurden zwei Algorithmen betrachtet, welche diese Herausfoderungen teilweise lösen.

Der heuristische Algorithmus (*Heuristic Mining*) erstellt zuerst einen Abhängigkeitsgraphen, dem im zweiten Schritt parallele Strukturen in AND- und XOR-Konstrukten hinzugefügt werden. So werden einfache Prozessmodelle entdeckt, Long Distance Dependencies werden jedoch nicht gefunden.

Der Evolutionäre Algorithmus (*Genetic Mining*) hingegen eignet sich gut für eine globale Suche und ist einer der wenigen, die Long Distance Dependencies finden, allerdings wird das Endergebnis nur mit Hilfe einer heuristisch erstellten Anfangspopulation wirklich gut. Außerdem neigen Evolutionäre Algorithmen dazu, sehr rechenaufwändig zu sein, da für jedes Individuum in jeder Generation eine Fitness berechnet werden muss.

Zusammenfassend spricht alles für eine Kombination der beiden Algorithmen: Heuristische Verfahren sind nicht nur für Anfangspopulationen von Evolutionären Algorithmen sinnvoll einzusetzen, sondern können auch innerhalb der Mutation für eine stärkere Zielausrichtung

sorgen. Gleichzeitig würde der Evolutionäre Algorithmus so weit eingeschränkt werden, dass er nicht das komplette Modell variiert, sondern nur die Teile, die in der Heuristik nicht erreicht werden, wie zum Beispiel weit voneinander entfernte Splits und Joins, die einander beeinflussen. Durch diese Konzerntration würde auch der hohe Rechenaufwand reduziert werden und das Discovery-Verfahren als ganzes beschleunigt werden.

Literatur

[dMvDvdAW] A.K. Alves de Medeiros, B.F. van Dongen, W.M.P. van der Aalst und A.J.M.M. Weijters. Process Mining: Extending the α-algorithm to Mine Short Loops.

[dMWvdA04] A.K. Alves de Medeiros, A.J.M.M. Weijters und W.M.P. van der Aalst. Using genetic algorithms to mine process models: representation, operators and results. 2004.

[dMWvdA06] A.K. Alves de Medeiros, A.J.M.M. Weijters und W.M.P. van der Aalst. Genetic Process Mining: A Basic Approach and Its Challanges. In *Business Process Management Workshops*, number task C, Seiten 203–215. Springer Berlin Heidelberg, 2006.

[dMWvdA07] A.K. Alves de Medeiros, A.J.M.M. Weijters und W.M.P. van der Aalst. Genetic process mining: an experimental evaluation. *Data Mining and Knowledge Discovery*, 14(2):245–304, Januar 2007.

[Kra09] Oliver Kramer. *Computational Intelligence*. Informatik im Fokus. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[Kra12] Oliver Kramer. Evolution Strategies, 2012.

[vdAdMM04] W.M.P. van der Aalst, A.K. Alves de Medeiros und L Maruster. Workflow Mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(9):1128–1142, 2004.

[vdAI12] W.M.P. van der Aalst und IEEE Task Force on Process Mining. Process mining manifesto. 2012.

[WvdAdM06] A.J.M.M. Weijters, W.M.P. van der Aalst und A.K. Alves de Medeiros. Process mining with the heuristics miner-algorithm. 2006.

[WWS06] Lijie Wen, Jianmin Wang und Jiaguang Sun. Detecting Implicit Dependencies Between Tasks from Event Logs. In Xiaofang Zhou, Jianzhong Li, HengTao Shen, Masaru Kitsuregawa und Yanchun Zhang, Hrsg., *Frontiers of WWW Research and Development - APWeb 2006*, Jgg. LNCS 3841, Seiten 591–603, Berlin Heidelberg, 2006. Springer.