



AGH

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

**Wydział Elektrotechniki, Automatyki,
Informatyki i Inżynierii Biomedycznej**

Projekt dyplomowy

*Automatyczne odkrywanie procesów biznesowych przy użyciu
programowania genetycznego*

Automated Business Process Discovery using Genetic Programming

Autor:

Piotr Seemann

Kierunek studiów:

Informatyka

Opiekun pracy:

dr inż. Krzysztof Kluza

Kraków, 2021

Spis treści

1. Wprowadzenie	5
1.1. Zarys tematyki pracy	5
1.2. Cele pracy	5
1.3. Zawartość pracy	6
2. Wstęp teoretyczny	7
2.1. Procesy biznesowe	7
2.2. Metody modelowania i eksploracji procesów	10
2.2.1. Modelowanie procesów biznesowych	10
2.2.2. Eksploracja procesów	12
2.2.3. Dzienniki zdarzeń	13
2.2.4. Automatyczne odkrywanie procesów biznesowych	14
2.3. Ewolucja gramatyczna	15
2.3.1. Algorytmy ewolucyjne	15
2.3.2. Operatory i działanie algorytmów ewolucyjnych	16
2.3.3. Ewolucja gramatyczna	17
2.3.4. Notacja Backusa-Naura	18
2.3.5. Tworzenie gramatyki pod kątem ewolucji	19
2.3.6. Omówienia działania algorytmu ewolucji gramatycznej	19
2.4. Metryki	21
2.4.1. Metryki a funkcja dopasowania	21
2.4.2. Dodatkowa metryka – złożoność	21
2.4.3. Opis i wzory użytych metryk	22
2.4.4. Obliczanie metryk	23
3. Projekt i implementacja	27
3.1. Wykorzystane technologie	27
3.1.1. Python 3.8.1	27
3.1.2. PonyGE2	27
3.2. Tworzenie gramatyki procesu biznesowego	28

3.3.	Projekt systemu	30
3.3.1.	Podział na moduły	30
3.3.2.	Model	31
3.4.	Implementacja	33
3.4.1.	Parsowanie gramatyki	33
3.4.2.	Obliczanie metryk	34
3.4.3.	Obliczanie odwzorowania dla pojedynczego wariantu	35
3.4.4.	Wyszukiwanie w modelu ścieżek o określonej długości	37
3.4.5.	Obliczanie odwzorowania	39
3.4.6.	Znajdowanie najlepiej dopasowanych aktywności w modelu	41
3.4.7.	Pozostałe wnioski dotyczące implementacji	42
3.5.	Wybór parametrów algorytmu	43
4.	Dyskusja rezultatów	45
4.1.	Przykładowe wyniki	45
4.1.1.	Wynik dla przykładu ze wstępu	45
4.1.2.	Inne przykłady działania	47
4.2.	Wyniki w zależności od przyjętych wag poszczególnych metryk	55
4.2.1.	Brak poszczególnych metryk	55
4.2.2.	Wpływ złożoności na wynik	58
4.3.	Wnioski dotyczące ewolucji	61
5.	Podsumowanie	63

1. Wprowadzenie

1.1. Zarys tematyki pracy

Zdefiniowanie kroków potrzebnych do osiągnięcia danego efektu jest konieczne do zrozumienia podejmowanych działań i wprowadzenia ewentualnych udoskonaleń. Z czasem biznes zdał sobie z tego sprawę i kierując się zasadą: „*Jeżeli nie jesteś w stanie opisać czegoś jako proces, nie masz pojęcia, co robisz*”, firmy zaczęły podejmować próby uporządkowania i zamknięcia swoich działań w ramy, co doprowadziło do wzrostu popularności procesów biznesowych.

Identyfikacja i opis procesów biznesowych sprawia, że wszystkie operacje w firmie stają się przejrzyste i łatwiejsze do zrozumienia. Analiza procesów biznesowych może pozwolić na zwiększenie produktywności oraz redukcję kosztów. Procesy biznesowe mogą pozwolić na przewidywanie przyszłych zdarzeń na podstawie danych, znajdowanie wąskich gardeł, a także zmniejszającą zależność firm od poszczególnych ludzi.

W związku z możliwością gromadzenia coraz większej ilości danych, a także chęcią ich wykorzystania oraz rosnącą popularnością analizy danych (ang. *data science*), biznes zdał sobie sprawę z możliwości wykorzystania technologii informatycznych w kontekście procesów biznesowych. Zapoczątkowało to powstanie na pograniczu zarządzania procesami biznesowymi i metod informatycznych używanych do analizy danych dziedziny zwanej eksploracją procesów (ang. *process mining*).

1.2. Cele pracy

Celem pracy jest projekt i implementacja metody odkrywania procesów biznesowych przy użyciu programowania genetycznego, a dokładniej ewolucji gramatycznej. W ramach pracy zaimplementowano program realizujący to zadanie oraz zbadano, jak wybór metryk, metod ewolucji, gramatyki, a także parametrów programu wpływa na jakość rozwiązania. Działanie zweryfikowano poprzez użycie stworzonego algorytmu do okrycia modeli procesów biznesowych dla dzienników zdarzeń różnej wielkości. Przykłady czego zamieszczono i omówiono w pracy. Ponadto została zbadana hipoteza, czy kontrola złożoności modelu w trakcie ewolucji ma korzystny wpływ na działanie algorytmu i ostateczne rozwiązanie.

1.3. Zawartość pracy

Praca została podzielona na cztery części. We wstępie teoretycznym zostały przybliżone zagadnienia potrzebne do zrozumienia pracy, takie jak procesy biznesowe, eksploracja procesów oraz ewolucja gramatyczna. Omówiono też wybór i działanie metryk dla modeli procesów biznesowych. W kolejnej części została przedstawiona gramatyka stworzona na potrzeby odkrywania procesów oraz projekt i implementacja służącego do tego algorytmu opartego o ewolucję gramatyczną. Następnie zaprezentowane zostały wyniki działania algorytmu dla przykładowych dzienników zdarzeń. Sprawdzono zostało jak na czas znajdowania rozwiązania oraz jego jakość wpływają przyjęte parametry algorytmu w szczególności wybór metryk, a także wagi, z jakimi każda metryka powinna być brana pod uwagę. Zweryfikowana została hipoteza dotycząca wpływu kontroli złożoności modelu. Na koniec przedstawiono podsumowanie całości pracy.

2. Wstęp teoretyczny

W celu wprowadzenia do tematyki odkrywania procesów biznesowych oraz programowania genetycznego w tej części zostały przybliżone niezbędne zagadnienia teoretyczne. Zaprezentowana tematyka obejmuje procesy biznesowe, zarządzanie nimi oraz ich eksplorację. Przedstawiono również notacje używaną do reprezentowania procesów w dalszej części pracy. Następnie, przybliżono programowanie genetyczne oraz ewolucję gramatyczną, a także omówiono, jakiej gramatyki używać w tej technice. Do oceny jakości modelu konieczne są metryki, dlatego opisano używane metryki i pokazano, jak je obliczać.

2.1. Procesy biznesowe

W każdym dużym przedsiębiorstwie codziennie wykonywana jest ogromna ilość czynności koniecznych do funkcjonowania tej organizacji. Ludzie oraz systemy realizują rozmaite działania związane z różnymi, często niemającymi wiele wspólnego zadaniami, jak np. przetwarzanie płatności, składanie zamówień, wytwarzanie produktów czy ich transport. Przykłady te można mnożyć w zależności od sektora, do jakiego należy dana firma. Im jest ona większa, tym trudniej jest osobom nią zarządzającym zrozumieć i opisać poszczególne czynności. W pewnym momencie, kiedy ilość różnych zadań rośnie do setek czy tysięcy, staje się to niemożliwe i potrzebny jest sposób na zebranie wiedzy o pojedynczych operacjach i zamknięcie ich w uporządkowaną strukturę. Stąd narodził się pomysł na wykorzystanie procesów biznesowych.

Procesy biznesowe opisują zbiór aktywności, które podejmuje grupa podmiotów w celu osiągnięcia celu biznesowego. W literaturze brakuje jednej ogólnie przyjętej definicji procesu biznesowego. W latach 90. XX wieku orędownicy BPR, czyli przeprojektowania procesów biznesowych (ang. *Business Process Re-engineering*) starali się sprecyzować pojęcie procesu biznesowego. W książce „Process Innovation: Reengineering Work through Information Technology” [1] określono termin ten jako: „*Ustrukturyzowany, mierzalny zbiór działań, których celem jest wytworzenie określonego produktu dla określonego klienta lub rynku*”. Autor położył nacisk na zbiór kroków prowadzących do celu, raczej niż na końcowy efekt. W dalszej części podsumowano: „*Proces jest zatem określonym uporządkowaniem czynności roboczych w czasie i przestrzeni, z początkiem i końcem oraz jasno określonymi wejściami i wyjściami: strukturą działania*”. Inni pionierzy BPR Michael Hammer i James Champy zdefiniowali proces biznesowy jako: „*zbiór działań, który pobiera jeden lub więcej rodzajów danych wejściowych i tworzy wynik, który ma wartość dla klienta*” [2]. Autorzy dają większą dowolność, co do definicji procesu, nie wspominając o konieczności jego logicznej organizacji czy mierzalności. Z kolei Ivar Jacobson zupełnie

pomija konieczność zamknięcia procesu w jakiegokolwiek ramy, określając go jako: „Zestaw czynności wewnętrznych wykonywanych w celu obsługi klienta” [3]. Nacisk na konieczność odniesienia procesów do wymiernych środków firmy widzimy w definicji zaproponowanej przez Hans-Erika Erikssona: „Procesy biznesowe są aktywną częścią biznesu. Opisują funkcje firmy i obejmują zasoby, które są używane, przekształcane lub wytwarzane. Proces biznesowy to abstrakcja, która pokazuje współpracę między zasobami i transformację zasobów w biznesie. Podkreśla, w jaki sposób wykonywana jest praca, zamiast opisywać produkty lub usługi wynikające z tego procesu.” [4]. Szczególnie ważny jest tutaj fragment o transformacji zasobów, gdyż każe on rozumieć poszczególne aktywności w procesie jako powiązane ze sobą i kończące się namacalnymi rezultatami. Definicja „Proces biznesowy to seria kroków mających na celu wytworzenie produktu lub usługi. W wyniku niektórych procesów produkt lub usługa jest odbierana przez zewnętrznego klienta organizacji. Nazywamy je podstawowymi procesami. Inne procesy wytwarzają produkty, które są niewidoczne dla klienta zewnętrznego, ale są niezbędne do efektywnego zarządzania firmą. Nazywamy je procesami wsparcia” [5] wprowadza rozgraniczenie na podtypy procesów. Ważnym jest jednak, że nie jest koniecznością, aby rezultaty procesu były widoczne na zewnątrz organizacji. Warto też zaznaczyć, że procesy biznesowe nie dotyczą jednej osoby czy nawet działu, a raczej udział w nich bierze wiele ludzi, maszyn czy systemów z różnych działów połączonych celem dostarczenia wspólnej wartości biznesowej.

Powyższe definicje skupiają się na odmiennych aspektach procesów biznesowych, nie zawsze szczegółowo wspominając o innych. Starając się usystematyzować powyższe sformułowania, chcąc zbudować bazę do dalszej analizy tematu, można przyjąć, że procesy biznesowe charakteryzują:

- Określony cel, którym jest wytworzenie wartości dla klienta zewnętrznego lub pośrednio firmy – klienta wewnętrznego. Jednak warto jeszcze raz zaznaczyć, że proces biznesowy skupia się na sposobie osiągnięcia celu, a nie opisie celu samego w sobie.
- Dyskretny, jasno zdefiniowany i identyfikowalny zbiór aktywności.
- Jasno określony początek – wejście i koniec – wyjście.
- Zależność przyczynowo-skutkowa pomiędzy kolejnymi aktywnościami.

Aby lepiej zilustrować, czym jest proces biznesowy, na rys. 2.1 znajduje się prosty przykład często spotykanego procesu.



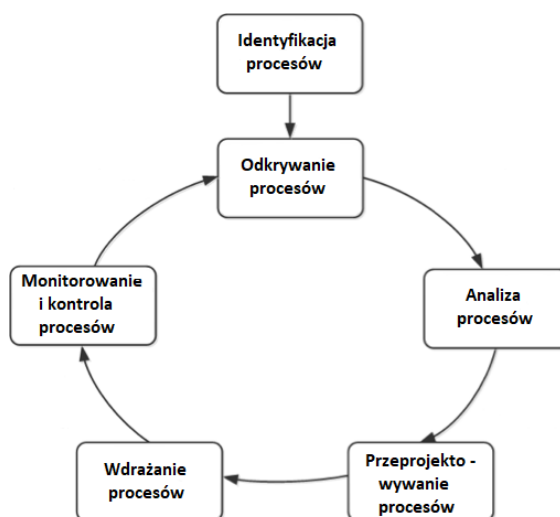
Rys. 2.1. Przykład prostego procesu

Zauważmy, że mamy jasno zdefiniowany wejście – otrzymanie zamówienia od klienta oraz wyjście, kiedy dostarczamy oczekiwaną wartość dla klienta, a całość składa się z serii tworzących logiczną całość aktywności. Są one konkretnie zdefiniowane.

Zdefiniowanie procesu biznesowego otwiera wiele możliwości analizy działań przedsiębiorstwa i wskutek tego wprowadzanie usprawnień. Dziedziną, która się tym zajmuje, jest zarządzanie procesami biznesowymi (ang. *Business Process Management*) zwane w skrócie BPM. Sercem jest proces, a samo BPM jest dyscypliną używającą różnych metod, technik i sposobów w celu projektowania, wprowadzania w życie, zarządzania i analizy procesów biznesowych [6].

Celem stosowania metod zarządzania procesami biznesowym jest udoskonalanie procesów w danej organizacji biznesowej. Udoskonalanie może być rozumiane w różnoraki sposób w zależności od kierunku rozwoju firmy. Może to być na przykład redukcja czasu, kosztów, czy dostarczanie lepszego produktu końcowego. Ważne jest, aby było to podejście całościowe i odnosiło się do całego zbioru aktywności w ramach danego procesu. Usprawnianie pojedynczej aktywności to nie jest BPM. Patrząc na przykład powyżej, jeśli wprowadzono by usprawnienia w ramach wysyłania faktury, robiąc to elektronicznie zamiast tradycyjną pocztą, mimo że taka zmiana przyniosłaby poprawę wydajności, nie byłoby to zarządzaniem procesami biznesowymi. O BPM można by mówić, gdyby znaleziono sposób, żeby przeprojektować cały proces tak, aby wysyłanie faktury nie było potrzebne lub odwrotnie, jeśli dodano by nowe aktywności, co usprawniłoby proces jako całość czy nawet zmieniono kolejności zdarzeń w procesie, gdyż zmiana ilości poszczególnych, jednostkowych aktywności nie są konieczna, żeby ulepszyć proces jako całość [7].

Zarządzanie procesami biznesowymi jest zbiorem praktyk mających na celu udoskonalanie procesów. Trzeba więc rozumieć BPM jako pojęcie abstrakcyjne, jednak szczególnie w dzisiejszym świecie, zarządzanie procesami biznesowymi nie może się obyć bez wsparcia ze strony oprogramowania czy technik znanych z różnych dziedzin informatyki [8].



Rys. 2.2. Cykl życia procesu biznesowego [9]

Cykl życia procesu biznesowego (ang. *Business Process Lifecycle*) przedstawiono na rys. 2.3. Jest to zbiór kroków niezbędnych do skutecznego zarządzania procesami biznesowymi. W celu dostosowania do zmieniającej się rzeczywistości poszczególne kroki powinny być co pewien czas powtarzane.

Konieczność powtarzania elementów cyklu życia procesu biznesowego sygnalizuje przewagę komputerów i algorytmów nad wykonywaniem tych operacji przez człowieka. Metody informatyczne są stosowane na każdym z wymienionych etapów. W szczególności dane zebrane w wyniku monitorowania procesów dają możliwość zastosowania metod z zakresu eksploracji procesów (sekcja 2.2). Praca skupia się w głównej mierze na odkrywaniu procesów, czyli znajdowaniu istniejących już procesów na podstawie realnych danych. Należy zaznaczyć, że identyfikacja polega na ogólnym rozpoznaniu i nazwaniu zachodzących procesów, podczas gdy odkrywanie jest bardziej szczegółowe, a w jego wyniku otrzymujemy dokładny model.

2.2. Metody modelowania i eksploracji procesów

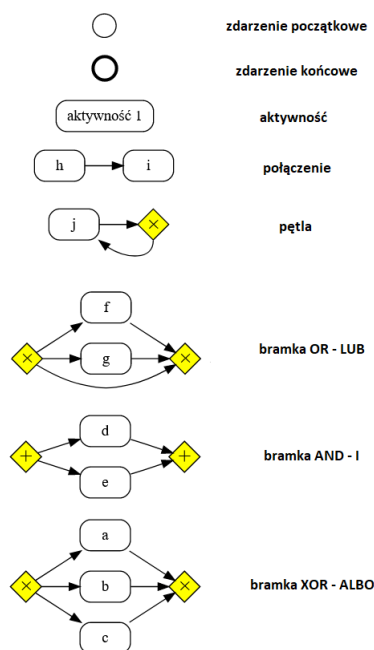
W tej sekcji rozwinięto tematykę procesów biznesowych o metody ich modelowania oraz eksploracji na podstawie danych zawartych w dziennikach zdarzeń.

2.2.1. Modelowanie procesów biznesowych

Na rys. 2.1 przedstawiono przykład uproszczonego procesu biznesowego. Łatwo sobie wyobrazić, że proces ten w rzeczywistości może być znacznie bardziej skomplikowany. Część aktywności może być wykonywana równolegle, niektóre zdarzenia w ogóle nie zaistnieją lub będą występować kilkakrotnie w ramach jednego procesu.

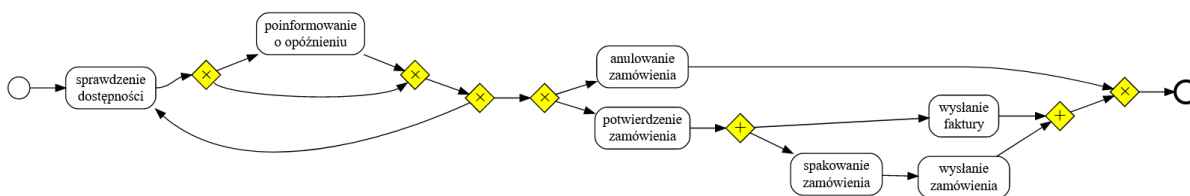
W sytuacji, w której zamówiony przez klienta towar będzie niedostępny, logiczne wydaje się poinformowanie go o opóźnieniu oraz danie mu możliwości anulowania zamówienia lub jego kontynuacja i ponowne sprawdzenie dostępności. Ponadto, czynności takie jak wysłanie faktury oraz spakowanie i wysłanie zamówienia mogą być wykonane w dowolnej kolejności czy nawet jednocześnie przez dwie różne osoby. Proces staje się bardziej skomplikowany i konieczna do stworzenia jego modelu jest bardziej złożona notacja niż użyta do przedstawienia najprostszego procesu. Istnieje wiele notacji do modelowania procesów biznesowych, wśród nich można wymienić schematy blokowe, diagramy aktywności UML, łańcuchy procesu sterowanego zdarzeniami (ang. *Event-driven Process Chains*), sieci Petriego [10]. Obecnie najpopularniejszą notacją używaną do opisu procesów biznesowych jest Business Process Model and Notation, w skrócie BPMN [11]. Daje ona możliwość opisanie w jednoznaczny sposób skomplikowanych procesów czy stworzenia diagramów współdziałania procesów, jednocześnie pozostając łatwą do zrozumienia.

Na grafice na rys. 2.3 przedstawiono notację opartą o elementy BPMN, używaną w dalszej części pracy. Składają się na nią zdarzenia początkowe i końcowe, połączenia, bramki logiczne oraz aktywności. Bramki LUB oraz ALBO mogą zostać pominięte, co pokazano dla bramki LUB. Podobnie pętle mogą zostać wykonane zero razy.



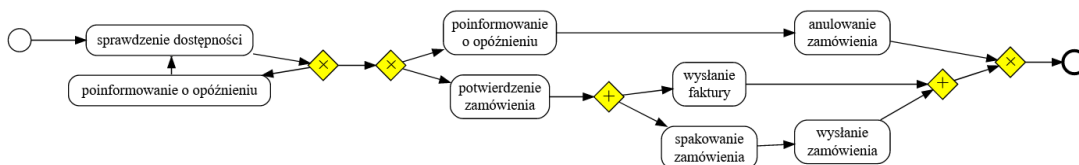
Rys. 2.3. Elementy składające się na notację

Korzystając z tej notacji, można przedstawić opisany wcześniej proces. Na rys. 2.4 widać model po modyfikacjach.



Rys. 2.4. Rozbudowany model procesu – przykład 1

Możliwe jest teraz poinformowanie klienta o opóźnieniu, a następnie anulowanie zamówienia lub powtórne sprawdzenie dostępności. Model ten jednak nie jest wystarczająco precyzyjny i pozwala na potwierdzenie zamówienia po informacji o jego opóźnieniu, a bez uprzedniego ponownego sprawdzenia dostępności. Można zaproponować inny model (rys. 2.5), który rozwiązuje powyższe problemy, jednak aktywność – poinformowanie o opóźnieniu – występuje w nim dwukrotnie, co jest niepożądane i pogarsza jego czytelność.



Rys. 2.5. Rozbudowany model procesu – przykład 2

Niewykluczona jest również sytuacja, że w pewnych przypadkach możliwe jest anulowanie zamówienia bez uprzedniego poinformowania klienta o opóźnieniu, a z czego nie zdawano sobie sprawy na etapie tworzenia modelu. Wtedy konieczne będzie przemodelowanie procesu. Aby radzić sobie z tymi problemami i tworzyć lepsze modele, powstał szereg zestawów wytycznych, którymi warto się kierować, modelując procesy biznesowe. Wśród takich zasad można wymienić: zminimalizuj liczbę elementów w modelu, zminimalizuj liczbę ścieżek w modelu, używaj jednego zdarzenia początkowego i jednego końcowego, unikaj bramek LUB, zdekomponuj model zawierający więcej niż 50 elementów [12].

Modelowanie procesów biznesowych jest próbą stworzenia uproszczonej wersji rzeczywistości na podstawie przewidywań i założeń. Modele dają użyteczne przybliżenie rzeczywistości, jednak należy pamiętać, że „wszystkie modele są błędne” i rzeczywisty proces najprawdopodobniej będzie różnił się od nawet najlepszego modelu.

2.2.2. Eksploracja procesów

W dzisiejszych czasach standardem jest, że organizacje biznesowe korzystają z systemów informatycznych, takich jak systemy ERP czy CRM, wspierających ich działalność. Systemy te rejestrują dane o procesach, które wspierają. Dane te mogą być później analizowane i wykorzystane do wprowadzenia usprawnień w działaniu firmy.

Tradycyjne, ręczne modelowanie procesów jest wolne, kosztowne i narażone na błędy ludzkie, a konieczność ciągłego powtarzania tej czynności, połączona z wszechobecnym w biznesie trendem automatyzacji sprawiają, że eksploracja procesów zyskuje na znaczeniu [13]. Ważna jest możliwość szybkiej adaptacji do zmian, a automatyzacja odkrywania procesów biznesowych pozwala na wykonywanie powtarzalnych zadań, eliminując przy tym błędy, co idealnie wpisuje się w ten trend.

Eksploracja procesów jest to szeroko pojęta dziedzina, która zawiera różne aplikacje inteligencji obliczeniowej, uczenia maszynowego i eksploracji danych do modelowania i analizy procesów. Jest wartościowym dodatkiem do innych metod eksploracji danych, gdyż zamiast skupiać się pojedynczym rezultacie i tworzyć dotyczące jego predykcje, celem jest poznanie akcji, które prowadzą do końcowego wyniku. Jest to trudniejsze, ale wyjątkowo cenne z punktu widzenia biznesowego, gdyż jakakolwiek zmiana w trakcie procesu może sprawić, że przewidywania będą kompletnie nietrafione, a zrozumienie całego procesu pozwala na pełniejszy obraz i łatwiejsze dostosowywanie się do zmian.

Procesy biznesowe są zazwyczaj domeną analityków i menadżerów, którzy nie zawsze podchodzą do tematu ich analizy w sposób ścisły i mający odniesie w faktach, często opierając się na własnych przeczuciach czy doświadczeniach, wprowadzając czynnik ludzki, który może być przyczyną błędów. Stworzenie pomostu między metodami informatycznymi a biznesem i stworzenie możliwości na ścisłe, powtarzalne i sprawdzalne analizowanie procesów jest więc nad wyraz cenne. Eksploracja procesów biznesowych oparta jest bowiem na danych i nie ma w niej wiele miejsca na przypuszczenia i domysły.

Na eksplorację procesów składają się techniki, narzędzia oraz metody odkrywania, monitorowania i usprawniania rzeczywistych procesów poprzez wiedzę wyodrębnioną z dzienników zdarzeń powszechnie dostępnych w systemach informatycznych [14, 15]. Wyróżnia się trzy podkategorie:

- automatyczne odkrywanie procesów (ang. *process discovery*),

- sprawdzanie zgodności (ang. *conformance checking*),
- udoskonalanie procesu (ang. *performance mining*).

2.2.3. Dzienniki zdarzeń

Danymi wejściowymi dla algorytmów z dziedziny eksploracji procesów są dzienniki zdarzeń, zwane często logami. Przykład przedstawiono na rys. 2.6.

nr przypadku	aktywność	data	osoba wykonująca
1	zgłoszenie problemu - a	2021.02.03 20:29:38	tester
1	programowanie (development) - b	2021.02.04 12:31:25	programista 1
2	zgłoszenie problemu - a	2021.02.05 19:13:32	klient
2	analiza - c	2021.02.06 02:43:09	analityk
2	programowanie (development) - b	2021.02.07 01:37:13	programista 2
2	testowanie - d	2021.02.08 12:43:45	tester
3	zgłoszenie problemu - a	2021.02.09 15:39:42	tester
3	development - b	2021.02.10 15:36:21	programista 1
1	analiza - c	2021.02.11 12:31:43	analityk
1	programowanie (development) - b	2021.02.12 00:01:54	programista 2
1	testowanie - d	2021.02.13 21:35:39	tester
4	zgłoszenie problemu - a	2021.02.14 09:23:59	tester
5	zgłoszenie problemu - a	2021.02.15 16:37:13	analityk
3	analiza - c	2021.02.16 02:29:56	analityk
3	programowanie (development) - b	2021.02.17 09:48:51	programista 1
3	testowanie - d	2021.02.18 20:50:28	tester
4	analiza - c	2021.02.19 15:48:37	analityk
4	programowanie (development) - b	2021.02.20 21:29:16	programista 1
4	sprawdzenie kodu (review) - e	2021.02.21 04:22:30	programista 2
5	uznanie problemu za rozwiązany - f	2021.02.22 06:28:29	programista 2
5	testowanie - d	2021.02.23 08:36:07	tester
4	testowanie - d	2021.02.24 21:17:54	tester

Rys. 2.6. Przykład dziennika zdarzeń

Przyjmuje się, że aby mówić o dzienniku zdarzeń powinien on zawierać trzy informacje niezbędne w kontekście odkrywania procesów biznesowych: numer przypadku, czyli unikalny identyfikator zbioru aktywności, nazwę aktywności oraz datę jej wykonania – ważną tylko w kontekście kolejności wykonywania pojedynczych aktywności. Ponadto może on zawierać inne dodatkowe informacje, takie jak: podmiot wykonujący daną aktywność, miejsce, koszt czy aktualny postęp wykonania. Te pozostałe dane mogą być wykorzystywane w kolejnych etapach analizy i usprawniania procesu.

Mając do dyspozycji te trzy informacje – poszczególne przypadki, aktywności na nie się składające oraz ich kolejność, zliczane jest, jak często poszczególne aktywności występują w danej kolejności. Każdy taki uporządkowany zbiór aktywności zwany jest wariantem procesu. Przykład odpowiadający dziennikowi zdarzeń pokazanemu w tej sekcji przedstawiono na rys. 2.7.

nr wariantu	ilość wystąpień	kolejność aktywności
1		2 a,b,c,b,d
2		1 a,c,b,d
3		1 a,c,b,e,d
4		1 a,f,d

Rys. 2.7. Warianty procesu odpowiadające przykładowemu dziennikowi zdarzeń

Dla poprawy czytelności aktywności często reprezentowane są jako symbole, np. kolejne litery alfabety, zamiast pełnej nazwy.

2.2.4. Automatyczne odkrywanie procesów biznesowych

Automatyczne odkrywanie procesów biznesowych jest poddziedziną eksploracji procesów i obejmuje techniki przekształcania danych w procesy. Wejściem jest dziennik zdarzeń, a wyjściem mapa lub model procesu. Ważne jest, żeby proces był oparty na analizie prawdziwych danych, a nie spekulacjach i założeniach. Automatyczne odkrywanie procesów pozwala na znajdowanie procesu takim, jaki jest, a nie takim, jakim chciano by, żeby był.

Celem automatycznego odkrywania procesów biznesowych jest zaprojektowanie funkcji – algorytmu, która przekształci dane z dziennika zdarzeń w model procesu [16]. Istnieje wiele algorytmów do odkrywania procesów biznesowych. Wśród najpopularniejszych można wymienić:

- Alpha algorithm [17],
- The ILP Miner [18],
- Heuristic Miner [19],
- Multi-phase Miner [20],
- Inductive Miner [21].

Istnieją cztery powszechnie używane kryteria do określania jakości otrzymanego modelu. Konieczne jest znalezienie balansu między nimi, gdyż często starając się poprawiać model pod kątem jednego kryterium, pogorszy się on pod względem innych. Powstało wiele metryk przedstawiających je za pomocą wzorów matematycznych [22, 23]. Bardziej szczegółowo wybór metryk omówiono w sekcji 2.4, a poniżej przedstawiono jedynie krótki opis poszczególnych kryteriów:

- odwzorowanie (ang. *fitness*) – zgodność modelu z dziennikiem zdarzeń,
- prostota (ang. *simplicity*) – złożoność i łatwość zrozumienia modelu,
- precyzja (ang. *precision*) – brak zachowań niezwiązanych z logiem, a możliwych w modelu,
- generalizacja (ang. *generalization*) – odzwierciedlenie w modelu brakujących aktywności z powodu niekompletnego logu.

Wśród problemów dotyczących istniejące algorytmy można wymienić problem ze znajdowaniem aktywności zachodzących równolegle, brak możliwości pomijania aktywności czy reprezentowania duplikatów, nieradzenie sobie z zakłóceniami w logu, tworzenie zbyt skomplikowanych modeli, czy trudność z odwzorowaniem niektórych zachowań. Modele stworzone mogą nie być spójne strukturalnie [24, 25], przez co rozumie się modele, w których istnieje aktywność, z której nie możemy osiągnąć zdarzenia końcowego lub nie może być ona w żaden sposób osiągnięta ze zdarzenia początkowego. Metody te zazwyczaj oparte są na grafach bezpośrednich następstw (ang. *directly follows graphs*), przez co problemem może być sytuacja, kiedy log jest niekompletny.

Zastosowanie algorytmów genetycznych do automatycznego odkrywania procesów biznesowych może być odpowiedzią na te kłopoty. Takie podejście pozwala na wyeliminowanie części problemów często dotyczących innych metod. Najważniejszą jednak przewagą algorytmów genetycznych jest pełna dowolność w kwestii generowania modelu pod kątem metryk zdefiniowanych przez użytkownika. Często znalezienie dobrze dopasowanego do logu modelu może być okupione stworzeniem bardzo skomplikowanego modelu – niewystarczająca prostota, pozwalającego na wiele zachowań nieobecnych w logu – niewystarczająca precyzja. Algorytm ewolucyjny pozwala na nieograniczoną możliwość manipulacji parametrami, żeby znaleźć oczekiwany balans między wszystkimi metrykami. Możliwe jest też stworzenie nowych, własnych metryk, gdyż są one niezależne od samego algorytmu ewolucyjnego. Klasyczne algorytmy mają problem z uzyskaniem dobrych rezultatów dla wszystkich metryk i nie mają możliwości zmiany parametrów startowych.

2.3. Ewolucja gramatyczna

2.3.1. Algorytmy ewolucyjne

Algorytmy ewolucyjne [26] są inspirowaną selekcją naturalną metaheurystyką, która używa znanych z ewolucji biologicznej operacji jak selekcja, krzyżowanie czy mutacja do rozwiązywania problemów wyszukiwania i optymalizacji. Są rodziną metod przeszukiwania przestrzeni losowych rozwiązań w celu wyszukania najlepszych z nich.

Algorytmy ewolucyjne znajdują zastosowanie w problemach, dla których nie jest konieczna gwarancja znalezienia najlepszego rozwiązania. Cechą wyróżniającą je na tle innych algorytmów uczenia maszynowego jest istnienie puli zamiast jednego rozwiązania, co umożliwia szersze przeszukiwanie przestrzeni rozwiązań oraz nieograniczone i łatwe w zaimplementowaniu zrównoleglenie. Algorytmy te są znacznie bardziej nastawione na globalne eksplorowanie nowych rozwiązań, zamiast na jak najszybsze osiągnięcie celu. Z tego względu dobrze nadają się do problemów, gdzie istnieje dużo ekstremów, a przestrzeń poszukiwań jest duża. Jako przeciwieństwo, najprostsze metody oparte na gradiencie znajdują tylko lokalne ekstrema, nawet po modyfikacjach takich, jak na przykład symulowane wyżarzanie (ang. *simulated annealing*) wciąż nie mają populacji i możliwości tak szerokiego przeszukiwania przestrzeni rozwiązań.

Sposób działania algorytmów genetycznych polega na stworzeniu populacji losowych rozwiązań zwanych genotypami lub chromosomami, które kodowane są za pomocą genów reprezentowanych przez bity, liczby lub znaki i zapisywanych w strukturze łatwo przetwarzalnej przez komputer. Najczęściej jest to lista jednowymiarowa liczb całkowitych. Fenotyp natomiast jest reprezentacją utożsamianą z docelowym programem lub modelem. Genotyp może być równoznaczny z fenotypem, jednak poza prostymi przykładami, zazwyczaj są to oddzielne reprezentacje i geny mapowane są na odpowiadające wartości w fenotypie, zwane allelami. Fenotyp jest postacią, dla której możliwe jest obliczanie funkcji dopasowania (ang. *fitness function*), co pozwala ocenić, jak dobre jest wygenerowane rozwiązanie. Następnie, istniejąca populacja jest modyfikowana poprzez krzyżowanie i mutacje. Warto zauważyć, że krzyżowanie przeszukuje przestrzeń rozwiązań globalnie, podczas gdy mutacja odpowiada za lokalne wyszukiwanie.

Po zastosowaniu tych operatorów i sklasyfikowaniu rozwiązań, spośród głównie(choć nie tylko) najlepszych osobników, utworzona zostaje nowa populacja, dla której cały proces jest powtarzany do momentu znalezienia satysfakcjonującego rezultatu.

2.3.2. Operatory i działanie algorytmów ewolucyjnych

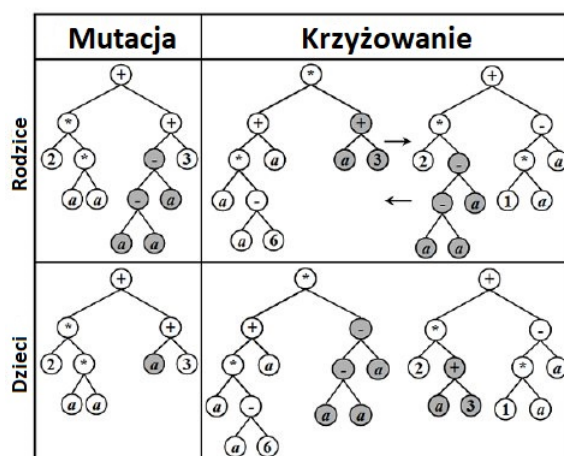
Tworzenie populacji jest pierwszym krokiem algorytmu ewolucyjnego. Może ono odbywać się kompletnie losowo lub do tworzenia nowych osobników może zostać użyta odpowiednia heurystyka.

Kolejny krok – selekcja – pozwala na zachowanie w populacji części osobników promując najlepszych z nich, co sprawia, że przeszukiwanie przestrzeni rozwiązań nie jest kompletnie losowe. Można skorzystać z metod takich jak:

- Selekcja proporcjonalna – wybierane są losowo osobniki z puli wszystkich w populacji z warunkiem, że rozwiązania z największą wartością metryk mają większą szansę na bycie w niej zachowanymi.
- Selekcja turniejowa – wybierany jest podzbiór ze zbioru rozwiązań i zachowywane w przyszłej populacji są najlepsze osobniki z tego podzbioru. Rozwiązanie to pozwala na wpływ na presję genetyczną – zwiększając wielkość podzbioru, ograniczamy szansę na wybór z niską wartością metryk. Jest to metoda prosta w implementacji, która umożliwia łatwe zrównoleglenie.

Żeby uniknąć sytuacji, w której najlepsze rozwiązania zostaną zmodyfikowane, możliwe jest zastosowanie elityzmu, który pozwala na zachowanie w kolejnej generacji części najlepszych osobników w populacji niezależnie od wyniku selekcji.

Następnie stosowane są operatory krzyżowania i mutacji. Zostały one zilustrowane na rys. 2.8. Krzyżowanie jest zamianą materiału genetycznego, czyli części genotypu pomiędzy dwoma osobnikami w populacji tworząc również dwa zmienione osobniki. Mutacja natomiast zachodzi w obrębie wyłącznie jednego osobnika.



Rys. 2.8. Mutacja i krzyżowanie [27]

Operatory te nie muszą i zazwyczaj nie są stosowane do każdego osobnika w populacji, a to jak często powinny być stosowane, ustalane jest za pomocą odpowiedniego parametru. Zdarza się, że mutacja jest stosowana więcej niż dla danego genu w danym genotypie.

Najczęściej używane operatory krzyżowania to:

- Krzyżowanie punktowe – spośród dwóch genotypów losowo wybierany jest jeden punkt, następnie tworzone są dwa nowe genotypy pierwszy z genów na prawo od punktu w pierwszym genotypie i na lewo w genotypie drugim oraz drugi z dwóch pozostałych.
- Krzyżowanie dwupunktowe – spośród dwóch genotypów losowo wybierane są dwa punkty, następnie część pomiędzy tymi punktami jest zamieniana pomiędzy genotypami.
- Krzyżowanie n-punktowe – uogólnienie powyższych krzyżowań dla n punktów.
- Krzyżowanie zamiana w drzewie – metoda opiera się na modyfikacjach w fenotypie, który jest reprezentowany jako drzewo, w tej metodzie zamieniane są ze sobą dwa poddrzewa. W tym przypadku tworzone są tylko prawidłowe rozwiązania, jednak jest to metoda wymagająca większej ilości obliczeń.

Ponadto krzyżowania możemy podzielić na stało-punktowe, czyli takie, w którym wybierany jest ten sam punkt lub punkty w obu osobnikach, przez co nowe genotypy mieć tę samą długość jak rodzice i zmiennie-punktowe, gdzie mogą to być różne punkty, co skutkuje dużą zmiennością w rozmiarach utworzonych genotypów.

Natomiast operatory mutacji to:

- Mutacja punktowa – dowolny gen w genotypie zostaje zmieniony na inną losową wartość.
- Mutacja zamiana w drzewie – metoda opiera się na modyfikacjach w fenotypie, który jest reprezentowany jako drzewo, w tej metodzie tworzone jest nowe poddrzewo. Analogicznie jak w przypadku krzyżowania tworzone są tylko prawidłowe rozwiązania, jednak konieczna jest większa ilość obliczeń.

Ostatnim krokiem jest zastąpienie osobników w poprzedniej populacji przez nowych powstałych na skutek zastosowania wcześniej wspomnianych operatorów. Istnieją dwa podejścia: sugerujące zastępowanie całej populacji lub tylko części osobników [28].

2.3.3. Ewolucja gramatyczna

Optymalny model procesu biznesowego może mieć różną wielkość, dlatego chcąc go znaleźć, potrzebna jest metoda, która pozwoli na generowanie rozwiązań o zmiennym rozmiarze. Tradycyjne algorytmy genetyczne operują na stałej strukturze i mogą być użyte na przykład, żeby dobrać odpowiednie parametry do istniejącego modelu. W wielu problemach jednak pożądanym jest rozwiązanie o nieznanym rozmiarze, dlatego obecnie najpopularniejsza metoda z tej dziedziny to programowanie genetyczne [29, 30], które pozwala na generowanie rozwiązań o różnej długości, dzięki ewolucji całej struktury

fenotypu, najczęściej reprezentowanego jako drzewo. Otrzymane rozwiązania są najczęściej gotowymi wykonywalnymi programami, jednak mogą być też równaniami czy modelami.

Na programowaniu genetycznym oparta jest ewolucja gramatyczna [31]. Korzysta ona ze standardowych metod ewolucji genetycznej, jednak ewoluuje gramatykę w celu znalezienia programu, który najlepiej rozwiązuje problem. Gramatyka najczęściej zapisana jest w notacji BNF (sekcja 2.3.4). Dzięki zastosowaniu rekursywnych produkcji możliwe jest tworzenie rozwiązań o różnej długości.

2.3.4. Notacja Backusa-Naura

Gramatyka jest zbiorem zasad opisujących budowę języka. Opisuje syntaktykę języka, czyli sposób łączenia poszczególnych symboli, nie mówiąc nic o znaczeniu poszczególnych słów języka. Odnosi się to także do języków formalnych np. języków programowania. Do opisu takich języków służą gramatyki formalne. Na gramatykę formalną składa się uporządkowana czwórka $G = (T, N, P, S)$, gdzie:

- T – skończony zbiór symboli terminalnych, czyli stałych, symboli, które nie mogą być zastąpione innym ani podzielone na mniejsze
- N – skończony zbiór symboli nieterminalnych, czyli zmiennych, symboli, które są modyfikowane przy tworzeniu języka
- P – skończony zbiór produkcji, czyli zasad, przekształceń postaci $(NUT)^*N(NUT)^* \rightarrow (NUT)^*$, czyli przynajmniej jednego symbolu nieterminalnego w dowolny zbiór symboli terminalnych i nieterminalnych. Gwiazdka oznacza wszystkie możliwe słowa nad alfabetem łącznie ze słowem pustym.
- S – symbol startowy, gdzie $S \in N$

W informatyce szczególnie szeroko stosowane są gramatyki bezkontekstowe. Pozwalają one na pokazanie, w jaki sposób rekurencyjnie tworzony jest język, co jest potrzebne, żeby zrozumieć znaczenie programu. Zaletą jest też ich stosunkowo łatwe parsowanie. Gramatyka bezkontekstowa jest to gramatyka, w której wszystkie produkcje mają postać:

$$A \rightarrow \alpha,$$

gdzie A jest to pojedynczy symbol nieterminalny – $A \in N$, a α to dowolny (także pusty) zbiór symboli terminalnych i nieterminalnych – $\alpha \in (NUT)^*$.

Notacja Backusa-Naura (ang. *Backus-Naur Form*) [32, 33, 34] jest najpopularniejszą notacją używaną do kodowania gramatyk bezkontekstowych. Symbole, które składają się na BNF to:

- $::=$ – produkcja,
- $|$ – lub,
- $\langle \rangle$ – symbole nieterminalne.

Używając tych symboli, możemy opisywać składnię języka w następujący sposób:

$$\langle \text{Nieterminalny1} \rangle ::= \langle \text{Nieterminalny2} \rangle \text{Terminalny1} \mid \langle \text{Nieterminalny3} \rangle \mid \text{Terminalny1}$$

Oczywiście możliwe jest też definiowanie rekurencyjnie, co pozwala na tworzenie skomplikowanego języka za pomocą prostych zasad:

$$\langle \text{Nieterminalny1} \rangle ::= \langle \text{Nieterminalny2} \rangle \mid \langle \text{Nieterminalny2} \rangle \langle \text{Nieterminalny1} \rangle$$

2.3.5. Tworzenie gramatyki pod kątem ewolucji

Dla każdego problemu można stworzyć nieskończoną ilość poprawnych gramatyk, jednak nie wszystkie z nich będą odpowiednie do zastosowania w algorytmie ewolucji gramatycznej. Chcąc stworzyć gramatykę, która umożliwi najwydajniejsze rozwiązanie problemu można przyjąć kilka wytycznych [35]:

- Każda rekursywna produkcja powinna mieć co najmniej tyle samo produkcji nierekursywnych, inaczej biorąc pod uwagę, że genotyp jest mapowany na fenotyp metodą pseudolosową, prawdopodobieństwo otrzymania rozwiązania, które musi się składać tylko symboli terminalnych, będzie zbyt niskie.
- Tworząc gramatykę pod kątem wykorzystania jej w procesie ewolucji, ważne jest, żeby ilość produkcji jak najlepiej odzwierciedlała, jak często chcemy uzyskać dane rozwiązanie.
- Stosując mutacje oraz krzyżowanie punktowe lub n-punktowe, które podmienia geny w genotypie, może dojść do sytuacji, w której dany gen po podmianie zostanie zmapowany na produkcje o innej ilości symboli nieterminalnych, przez co kolejne geny, będą mapowane niezgodnie z pierwotnym sensem, gdyż liczba genów w genomie pozostaje stała. Żeby temu zapobiec, należy ograniczyć liczbę symboli nieterminalnych do minimum, a każdy symbol nieterminalny powinien mieć produkcje tworzące tyle samo symboli nieterminalnych oraz możliwie tyle samo produkcji, dzięki czemu po zmapowaniu symbole nadal będą odpowiadały pierwotnym.
- Stworzona gramatyka powinna umożliwić tworzenie jak najmniejszej ilości rozwiązań, które nie należą do języka, co prawda nie uniemożliwi to działania algorytmu, gdyż można odrzucić te rozwiązania na etapie parsowania, ale warto oszczędzić niepotrzebnych obliczeń i rozwiązać ten problem już na etapie tworzenia gramatyki.

2.3.6. Omówienia działania algorytmu ewolucji gramatycznej

Mechanizmem, który wyróżnia ewolucję gramatyczną na tle innych algorytmów ewolucyjnych, jest mapowanie genotypu zapisywanego jako jednowymiarowa tablica liczb całkowitych na fenotyp – język, zgodnie z zasadami stworzonej gramatyki. Przykładem może być genotyp:

[6, 71, 92, 59, 52, 95, 23, 45, 12, 2]

i gramatyka w notacji BNF:

$$\begin{aligned}
\langle e \rangle &::= \langle var \rangle = \langle math \rangle \langle var \rangle \\
\langle math \rangle &::= \langle math \rangle \langle math \rangle \mid \langle var \rangle \langle op \rangle \\
\langle op \rangle &::= + \mid - \\
\langle var \rangle &::= a \mid b \mid c \mid d,
\end{aligned}$$

której symbolem startowym jest $\langle e \rangle$. Wybór kolejnych produkcji odbywa się według zasady:

$$produckja = gen_jako_liczba_cakowita \% ilosc_dostepnych_produkcji$$

Używając wyprowadzenia lewostronnego:

- Aktualne wyrażenie: $\langle e \rangle$;
Aktualny symbol nieterminalny: $\langle e \rangle$;
Ilość możliwych produkcji: 1;
Wartość genu: 6;
 $6 \bmod 1 = 0$, więc $\langle e \rangle$ zostaje zastąpiony przez $\langle var \rangle = \langle math \rangle \langle var \rangle$
- Aktualne wyrażenie: $\langle var \rangle = \langle math \rangle \langle var \rangle$;
Aktualny symbol nieterminalny: $\langle var \rangle$;
Ilość możliwych produkcji: 4;
Wartość genu: 71;
 $71 \bmod 4 = 3$, więc $\langle var \rangle$ zostaje zastąpiony przez d
- Aktualne wyrażenie: $d = \langle math \rangle \langle var \rangle$;
Aktualny symbol nieterminalny: $\langle math \rangle$;
Ilość możliwych produkcji: 2;
Wartość genu: 92;
 $92 \bmod 2 = 0$, więc $\langle math \rangle$ zostaje zastąpiony $\langle math \rangle \langle math \rangle$
- Aktualne wyrażenie: $d = \langle math \rangle \langle math \rangle \langle var \rangle$;
Aktualny symbol nieterminalny: $\langle math \rangle$;
Ilość możliwych produkcji: 2;
Wartość genu: 59;
 $59 \bmod 2 = 1$, więc $\langle math \rangle$ zostaje zastąpiony $\langle var \rangle \langle op \rangle$

Kontynuując analogicznie, po zastąpieniu wszystkich symboli nieterminalnych, ostatecznie otrzymany fenotyp to:

$$d = a - b + c$$

Używając tego prostego mechanizmu, w przypadku bardziej skomplikowanej gramatyki, możliwe jest generowanie znacznie bardziej złożonych rozwiązań. W połączeniu z metodami z algorytmów ewolucyjnych możliwe jest teoretycznie ewoluowanie programów w dowolnym języku programowania, a następnie za pomocą odpowiedniej funkcji dopasowania znalezienie takiego, który rozwiąże dany problem. W kolejnej sekcji omówiono jak dobrać metryki, aby użyć tego mechanizmu do odkrywania procesów biznesowych.

2.4. Metryki

2.4.1. Metryki a funkcja dopasowania

Funkcja dopasowania jest obliczana w każdej iteracji, dla każdego osobnika w populacji, dlatego powinna być możliwie najmniej kosztowna obliczeniowo. Nie warto więc nadmiernie jej komplikować, gdyż nawet jeśli taka wersja lepiej określi, jak dobre jest dane rozwiązanie, to jej obliczenie może stać się wąskim gardłem całego algorytmu, co zwiększy czas potrzebny na znalezienie rozwiązania i pogorszy jego działanie.

Dobierając funkcję dopasowania, ważne jest, żeby była ona regularna, gdyż duża ilość lokalnych ekstremów może spowolnić ewolucję, a nawet całkowicie uniemożliwić znalezienie globalnie najlepszego rozwiązania. Nie może więc ona jedynie bezwzględnie mierzyć, jak dobre jest rozwiązanie. Dobrym przykładem jest tutaj problem układanie planów (ang. *timetabling problem*), gdzie większość otrzymanych rozwiązań będzie nieprawidłowa, a wartość funkcji dopasowania będzie wynosiła zero, nawet jeśli mała zmiana może sprawić, że znalezione zostanie właściwe rozwiązanie, dlatego konieczne jest, jak najlepsze uchwycenie jak blisko algorytm jest prawidłowego rozwiązania [36, 37].

Ostatecznie funkcja dopasowania w naszym algorytmie jest średnią ważoną metryk wymienionych w sekcji 2.2.4, jednak oprócz tego dodano kolejną metrykę złożoność, która ma poprawiać funkcje dopasowania zgodnie z powyższymi zasadami. Użytkownik może określić, z jakimi wagami wziąć pod uwagę poszczególne metryki.

2.4.2. Dodatkowa metryka – złożoność

Dodatkowa metryka zawiera informacje niedotyczące bezpośrednio jakości odkrytego modelu. Istnieją jednak teoretyczne przesłanki, że powinna wpłynąć pozytywnie na działanie algorytmu. W odróżnieniu od innych metryk celem jej używania jest poprawa wydajności algorytmu ewolucji i próba zminimalizowania czasu potrzebnego na znalezienie rozwiązania.

Zamysłem jest promowanie rozwiązywania prostych problemów w prosty sposób i unikanie lokalnych ekstremów – w tym wypadku sytuacji, w której populacja zostanie zdominowana przez zbyt skomplikowane osobniki na wczesnym etapie ewolucji, zamiast tego ewolucja jest bardziej nakierowana na znajdowanie coraz lepszego rozwiązania wraz ze wzrostem skomplikowania modelu. Analogię można odnaleźć w naturze, w pewnym momencie na Ziemi dinozaury były najlepiej przystosowanymi do życia na tej planecie organizmami – lokalne maksimum i blokowały powstawanie mniej skomplikowanych form życia, jednak jak pokazuje historia, po wyginięciu dinozaurów, te były w stanie stać się jeszcze lepiej przystosowanymi istotami.

Metryka korzysta z istniejących już obliczeń, przez co nie zwiększa w istotny sposób czasu każdej iteracji algorytmu. Większa złożoność każdego modelu przekłada się także na dłuższe obliczanie pozostałych metryk dla niego, przez co znalezienie ostatecznego rozwiązania zajmuje więcej czasu.

2.4.3. Opis i wzory użytych metryk

2.4.3.1. Prostota

Celem używania tej metryki jest zmniejszenie skomplikowania modelu. Głównym czynnikiem na to wpływającym jest liczba aktywności w modelu. Idealna byłaby sytuacja, w której model ma tyle samo aktywności, ile unikalnych aktywności jest w dzienniku zdarzeń. To nie zawsze jest możliwe, jednak chcąc otrzymać maksymalnie czytelny model, powinniśmy do tego dążyć. Stąd też wybrana metryka skupia się na dwóch czynnikach, czyli liczbie duplikatów w modelu i liczbie brakujących wartości w modelu. Znaczenie ma wielkość modelu, dlatego wartości te muszą zostać odniesione do ilości wszystkich zdarzeń w logu i modelu. Metrykę zapożyczono z [38] i wyrażono wzorem:

$$M_{pro} = 1 - \frac{\text{ilosc duplikatow w modelu} + \text{ilosc brakujacych zdarzen w modelu}}{\text{ilosc unikalnych zdarzen w logu} + \text{ilosc zdarzen w modelu}}$$

2.4.3.2. Odwzorowanie

Jest to najbardziej kosztowna obliczeniowo metryka. Pozostałe metryki obliczane są na podstawie informacji uzyskanych podczas jej obliczania. Ważne jest, żeby obliczać odwzorowanie częściowo dla każdej aktywności, a nie całego wariantu procesu, co sprawi, że metryka będzie bardziej wrażliwa na zmiany. Możliwe jest zastosowanie prostej zero-jedynkowej metryki do obliczania, która weryfikowałaby czy model całkowicie zgadza się z wariantem logu, jednak co jest szczególnie niekorzystne w przypadku algorytmu genetycznego, nie byłoby sprawdzane, o ile bliżej jesteśmy do celu, dopóki losowo nie zostałby znaleziony model idealnie pasujący do wariantu. Metrykę oparto o [39], jednak przy procesie z wieloma aktywnościami, najczęściej duża część aktywności może zostać dopasowana, co sprawi, że łączny błąd będzie niewielki, dlatego wprowadzono modyfikację i podniesiono otrzymany wynik do potęgi 4, żeby zrobić metrykę bardziej wrażliwą na zmiany. Ostatecznie metrykę wyrażono wzorem:

$$M_o = (1 - \text{blad_odwzorowania})^4$$

$$\text{blad_odwzorowania} = \frac{\sum \text{ilosc procesow w logu} \cdot \frac{\text{blad odwzorowania logu w modelu}}{\text{minimalna dugosc najlepszej sciezki w modelu} + \text{dugosc sciezki w logu}}}{\text{ilosc zdarzen w logu}}$$

2.4.3.3. Precyzja

Pozwala na unikanie niewystarczającego dopasowania (ang. *underfitting*). Celem jest uniknięcie tworzenia modeli, według których możliwe są praktycznie dowolne zachowania. Możliwe jest osiągnięcie maksymalnej wartości pozostałych metryk, tworząc bramkę LUB – OR zawierającą wszystkie aktywności, jednak nie jest to pożądany model. We wzorze skupiono się na ilości osiągalnych zdarzeń następujących po danej aktywności, możliwych w modelu. Chcemy, żeby poszczególne aktywności mogły poprzedzać tylko te, które rzeczywiście poprzedzają w logu. Metrykę oparto o [40], jednak otrzymany wynik podniesiono do potęgi $\frac{1}{3}$, bo oryginalna metryka jest zbyt wrażliwa na zmiany, przez co jest nieproporcjonalna do innych, co zwiększa prawdopodobieństwo utknięcia w lokalnym maksimum, gdzie każda mała zmiana w modelu będzie powodować ogromną zmianę w metryce. Ostatecznie wyrażono ją wzorem:

$$M_{pre} = (1 - \sum_{\text{zdarzenia w modelu}} \frac{\text{ilosc osiagalnych zdarzen w modelu} - \text{ilosc osiagalnych zdarzen w logu}}{\text{ilosc osiagalnych zdarzen w modelu}})^{\frac{1}{3}}$$

2.4.3.4. Generalizacja

Pozwala na unikanie nadmiernego dopasowania (ang. *overfitting*). W pierwszej chwili może wydawać się przeciwieństwem precyzji, jednak chodzi tutaj o uwzględnienie brakujących w dzienniku zdarzeń wariantów, a nie o rozszerzenie modelu o nowe, zupełnie niezwiązane z istniejącymi. Dobrą wizualizacją jej działania jest operacja domknięcia znana z dziedziny przetwarzania obrazów cyfrowych. Generalizację można zmierzyć poprzez obliczenie średniej ważonej liczby przejść w modelu przez dane zdarzenie, dzięki czemu wciąż uwzględniane są ścieżki, które są często odwiedzane w innych wariantach, mimo że pozwalają na zachowanie niewidoczne w dzienniku zdarzeń, a jednocześnie karane jest istnienie ścieżek, które zupełnie nie pokrywają się z żadnymi wariantami. Wraz ze zwiększaniem ilości wykonań danej aktywności wzrasta pewność, że ścieżki ją uwzględniające faktycznie istnieją i ta informacja staje się mniej cenna, dlatego wzięto pierwiastek z liczby wystąpień danego zdarzenia. Metrykę zapożyczono z [38] i ostatecznie wyrażono wzorem:

$$M_g = 1 - \frac{\sum_{\text{zdarzenia w modelu}} \frac{1}{\sqrt{\text{ilosc wystapien zdarzenia}}}}{\text{ilosc zdarzen w logu}}$$

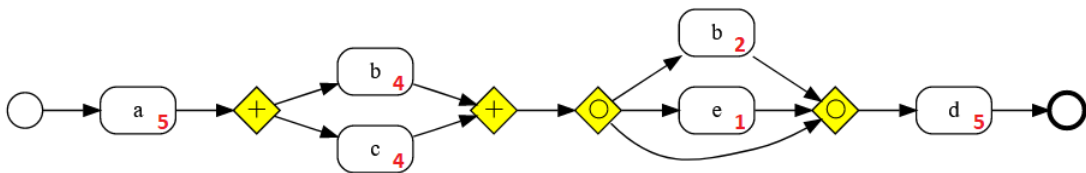
2.4.3.5. Złożoność

Metryka jest powiązana z odwzorowaniem. Jeśli ono rośnie, pozwala się na tworzenie bardziej złożonego modelu. Złożoność jest wyrażana jako ilość wszystkich możliwych ścieżek w modelu. W przypadku pętli jej wartość byłaby nieskończona, dlatego przyjęto, że pętla oznacza maksymalnie dwukrotne jej wykonanie. Jako że np. dla bramki AND jest to $n!$, zauważamy, że złożoność rośnie niewspółmiernie szybciej do odwzorowania, dlatego we wzorze wzięto pierwiastek z jej wartości. Ostatecznie metrykę wyrażono wzorem:

$$M_z = 1 - \frac{1}{\sqrt{1 + \text{blad_odwzorowania} * \sqrt{\text{zlozonosc modelu}}}}$$

2.4.4. Obliczanie metryk

W sekcji 2.2.3 przedstawiono przykład dziennika zdarzeń i warianty procesu uzyskane dla niego. Na rys. 2.9 przedstawiono, jak mógłby wyglądać model tego procesu – na czerwono zaznaczono ilość wykonań danej aktywności – oraz zademonstrowano obliczanie metryk:



Rys. 2.9. Model, dla którego obliczane są metryki

2.4.4.1. Obliczanie prostoty

Model posiada jeden duplikat – **b** oraz jedno brakujące zdarzenie – **f**, więc wartość metryki wynosi:

$$M_{pro} = 1 - \frac{1}{6 + 6} = 0.8333$$

2.4.4.2. Obliczanie odwzorowania

Są trzy możliwości na niezgodności modelu z logiem – brak zdarzenia w modelu, brak zdarzenia w logu lub zdarzenie w logu różne od zdarzenia w modelu, co może być utożsamiane z brakiem zdarzenia w logu i modelu, dlatego w takim przypadku błąd wynosi 2, a w pozostałych 1. W powyższym przykładzie nieprawidłowe odwzorowanie zachodzi tylko dla jednego wariantu, co można przedstawić na kilka sposobów, jednak nie ma to znaczenia i błąd zawsze wynosi 3 (rys. 2.10).

Ścieżka w modelu	a	b	e	d
Ścieżka w logu	a	f		d

Ścieżka w modelu	a		b	e	d
Ścieżka w logu	a	f			d

Rys. 2.10. Nieprawidłowe odwzorowanie

Dla pozostałych wariantów odwzorowanie jest pełne i błąd wynosi 0 (rys. 2.11).

Ścieżka w modelu	a	b	c	e	d
Ścieżka w logu	a	b	c	e	d

Rys. 2.11. Prawidłowe odwzorowanie

Podstawiając wszystkie odwzorowania do wzoru:

$$blad_odwzorowania = \frac{2 * \frac{0}{5+5} + \frac{0}{4+4} + \frac{0}{5+5} + \frac{3}{4+3}}{22} = 0.0195$$

$$M_o = (1 - 0.0195)^4 = 0.9243$$

2.4.4.3. Obliczanie precyzji

Potrzebna jest ilość osiągalnych, kolejnych zdarzeń dla danej aktywności. Zależy ona od kontekstu, w jakim dana aktywność jest wykonywana np. dla bramki I - AND można jako kolejne wykonać tyle aktywności, ile nie zostało do tej pory wykonanych w tej bramce, z wyjątkiem ostatniej, gdzie przechodzimy do kolejnych aktywności spoza bramki. Dla demonstrowanego przykładu kolejne zdarzenia zaprezentowano na rys. 2.12.

Poprzedzające zdarzenia	Zdarzenia w logu	Ilość zdarzeń w logu	Zdarzenia w modelu	Ilość zdarzeń w modelu
[]	['a']	1	['a']	1
['a']	['b', 'c']	2	['b', 'c']	2
['a', 'b']	['c']	1	['c']	1
['a', 'b', 'c']	['b']	1	['b', 'e', 'd']	3
['a', 'b', 'c', 'b']	['d']	1	['e', 'd']	2
['a', 'b', 'c', 'b', 'd']	['koniec']	1	['koniec']	1
['a', 'c']	['b']	1	['b']	1
['a', 'c', 'b']	['d', 'e']	2	['b', 'd', 'e']	3
['a', 'c', 'b', 'd']	['koniec']	1	['koniec']	1
['a', 'c', 'b', 'e']	['d']	1	['b', 'd']	2
['a', 'c', 'b', 'e', 'd']	['koniec']	1	['koniec']	1

Rys. 2.12. Kolejne zdarzenia

Podstawiając te dane do wzoru:

$$M_{pre} = (1 - (\frac{1-1}{1} + \frac{2-2}{2} + \frac{1-1}{1} + \frac{3-1}{3} + \frac{2-1}{2} + \frac{1-1}{1} + \frac{1-1}{1} + \frac{3-2}{3} + \frac{1-1}{1} + \frac{2-1}{2} + \frac{1-1}{1}))^{\frac{1}{3}} = 0.9465$$

2.4.4.4. Obliczanie generalizacji

Jeśli ilość przypadków jest mała, jak w powyższym przykładzie, to generalizacja będzie słaba. W tym przypadku otrzymano:

$$M_g = 1 - \frac{(\frac{1}{\sqrt{5}} + \frac{1}{\sqrt{4}} + \frac{1}{\sqrt{4}} + \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{1}} + \frac{1}{\sqrt{5}})}{6} = 0.3997$$

2.4.4.5. Obliczanie złożoności

Złożoność jest obliczana dla poszczególnych bramek logicznych w modelu, co jest konieczne na wcześniejszym etapie działania algorytmu, a następnie te wartości są podstawiane do wzoru wraz z błędem dopasowania obliczonym w sekcji 2.4.4.2:

$$zlozonosc_dla_bramki_AND = 2! = 2$$

$$zlozonosc_dla_bramki_OR = 2! * \frac{2!}{(2!*(2-2)!)} + 1! * \frac{2!}{(1!*(2-1)!)} + 0! * \frac{2!}{(0!*(2-0)!)} = 2 + 2 + 1 = 5$$

$$zlozonosc = 1 * 2 * 5 * 1 = 10$$

$$M_z = 1 - \frac{1}{\sqrt{1+0.0195 * \sqrt{10}}} = 0.9706$$

2.4.4.6. Znaczenie otrzymanych metryk

Najniższą wartość otrzymano dla generalizacji. Jest to naturalne, gdyż przykładowy dziennik zdarzeń składał się tylko z pięciu przypadków, więc należy spodziewać się nadmiernego dopasowania modelu do logu. Przedstawiony model nie opisuje wszystkich przypadków w modelu stąd niska wartość odwzorowania. Jest on jednak mało złożony i precyzyjny. Możliwe jest znalezienie lepszego modelu, co pokazano w dalszej części pracy.

3. Projekt i implementacja

W tej części zaprezentowana została stworzona gramatyka procesu biznesowego. Zaprezentowano również listingi kodu i omówiono najważniejsze elementy stworzonego programu. Na koniec zaproponowano, jakie parametry startowe wybrać z niego korzystając.

3.1. Wykorzystane technologie

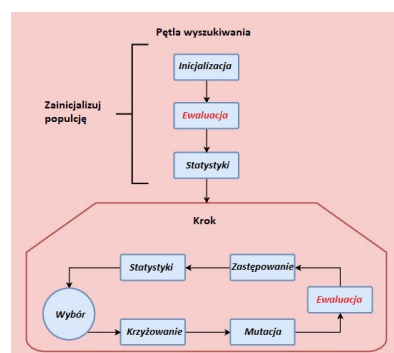
Do implementacji algorytmu został użyty język Python oraz PonyGE2.

3.1.1. Python 3.8.1

Python jest to najpopularniejszy język programowania w domenie uczenia maszynowego. Wymagana jest wersja 3.8 lub wyższa ze względu na użycie w implementacji metod dostępnych od tej wersji.

3.1.2. PonyGE2

PonyGE2 [41], którego ogólny schemat działania pokazano na rys. 3.1 jest implementacją ewolucji genetycznej w języku Python. Pozwala na łatwą konfigurację parametrów ewolucji genetycznej oraz możliwość dodania własnych problemów, a także sposobów ewaluacji rozwiązań. Niestety, PonyGE2 nie jest przystosowane do bycia dołączaną jako niezależna biblioteka i nie umożliwia dostępu poprzez wygodny interfejs.



Rys. 3.1. Ogólny schemat działania PonyGE2 [41]

3.2. Tworzenie gramatyki procesu biznesowego

Projektując gramatykę procesu biznesowego, przyjęto dwa początkowe założenia. Uznano, że generowane modele muszą być łatwe do przełożenia na notację BPMN oraz nie powinny być tworzone modele niespójne strukturalnie, co pozwala na zredukowanie przestrzeni rozwiązań, jednocześnie gwarantując tworzenie niegenerujących błędów modeli.

Przy tworzeniu gramatyki procesu biznesowego ważne jest, żeby znaleźć balans, jeśli chodzi o poziom skomplikowania modelu, jaki będzie możliwy do wygenerowania używając zaproponowanej gramatyki. W pracy [42] przeanalizowano składniki języka BPMN pod kątem częstotliwości ich stosowania. Najczęściej używanymi elementami, jeśli chodzi o bramki, są: XOR – ALBO kodowane w proponowanej gramatyce jako *xor*, AND – I jako *and* oraz pętle jako *lo<0_n>*. Do przedstawionej dalej gramatyki dodano także bramki OR – LUB reprezentowane jako *opt*. Ponadto konieczne jest użycie symbolu *seq*, która oznacza, że aktywności następują kolejno po sobie.

Zgodnie z zaleceniami z sekcji 2.2.1 przyjmuje się, że dobrą praktyką jest, żeby model zawierał tylko jedno zdarzenie początkowe i końcowe. Z tego powodu przyjęto, że zdarzenia te są domyślnie odpowiednio na początku i końcu wygenerowanego słowa i nie są one jawnie reprezentowane w gramatyce.

W sekcji 2.3.5 opisano problemem ewolucji gramatyki dla metod opartych o krzyżowanie i mutację punktową lub n-punktową, dlatego zdecydowano się na stworzenie gramatyki pod kątem wersji tych operatorów używających fenotypu – drzewa. Lokalne przeszukiwanie często staje się słabym punktem algorytmów ewolucyjnych. Stosując wspomniane operatory, prawdopodobna jest sytuacja, że mała modyfikacja blisko korzenia może poprawić rozwiązanie, jednak jej zaistnienie wymaga wygenerowania identycznego z istniejącym poddrzewa na nowo, przez co prawdopodobieństwo wystąpienia takiej sytuacji jest niskie. Do rozwiązania tego problemu mogą służyć metody inspirowane algorytmami memetycznymi, a działające na drzewach [27]. Dają one możliwość aplikowania lokalnych zmian bez konieczności powtórnego generowania całego poddrzewa, co pozwala na usprawnienie procesu ewolucji. Niestety, użyta biblioteka nie posiada podobnych metod lokalnej optymalizacji. Żeby w pewnym stopniu zaradzić temu problemowi, zmniejszono głębokość potrzebnego do reprezentacji modelu drzewa, jednocześnie zwiększając szanse na lokalne mutacje poprzez wprowadzenie symbolu nieterminalnego *<slots>*. Sprawia to, że drzewo rośnie bardziej wszędy i oprócz jednego symbolu *<anygate>*, którego użycie ma zapewnić tworzenie poprawnych, niepustych rozwiązań, generowane są symbole *<slot>*, które mogą pozostać puste lub wygenerować symbol *<anygate>* z 10% prawdopodobieństwem. Z tego powodu produkcja dla symbolu *<slot>* zawiera 9 zapisów ' ' oznaczających pusty znak. Przekłada się to na większą ilość lokalnych zmian na późniejszych etapach ewolucji. Lokalne przeszukiwanie jest także wspierane poprzez reprezentowanie bramki jako dwa odrębne symbole – *<name>*(*<slots>*). Dzięki temu w wyprawadzeniu nazwa bramki *<name>* jest oddzielona od jej zawartości, co sprawia, że możliwa jest zmiana nazwy bramki bez modyfikacji jej wnętrza.

Wzięto również pod uwagę odwzorowania w gramatyce częstotliwości występowania poszczególnych bramek logicznych. Zgodnie z [42] połączenia i bramki XOR – ALBO, AND – I są tworzone przez większą ilość produkcji niż rzadziej występujące pętle i bramki OR – LUB.

Zapis `GE_RANGE:n` jest rozszerzeniem notacji zapewnianym przez PonyGE2, które umożliwia wygodne dodanie n zmiennych, czyli `GE_RANGE:2` w BNF oznacza `0|1|2`. Podobny jest zapis `GE_RANGE:dataset_vars`, który umożliwia dodanie ilości zmiennych odpowiadającej ich ilości w zbiorze danych w tym wypadku liczbie aktywności w dzienniku zdarzeń. Został on dodany, dzięki rozszerzeniu standardowego, zapewnianego przez PonyGE2, parsera gramatyki.

Wszystkie bramki mają nazwy tej samej długości – 3 znaki, co ułatwia parsowanie gramatyki. Symbol startowy to `<e>`.

```

<e> ::= <slot><slot><anygate><slot><slot>

<anygate> ::= <anygate><anygate> | <name>(<slots>) | {<event>}

<slot> ::= <anygate> | ' ' | ' ' | ' ' | ' ' | ' ' | ' ' | ' ' | ' ' | ' '

<slots> ::= <slot><slot><anygate><slot><slot>

<name> ::= and | xor | seq | and | xor | seq | and | xor | seq |
          and | xor | seq | and | xor | seq | lo<0_n> | lo<0_n> | opt

<event> ::= GE_RANGE:dataset_vars

<0_n> ::= GE_RANGE:5

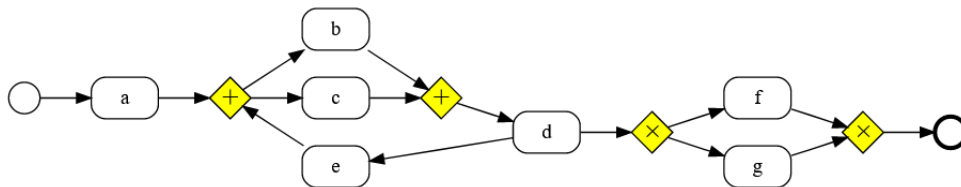
```

Listing 3.1. Proponowana gramatyka procesu biznesowego

Model, dla którego zaprezentowano obliczanie metryk (rys. 2.9) byłby za pomocą powyższej notacji zakodowany jako:

$$(\{a\} \text{and} (\{b\}\{c\}) \text{opt}(\{b\}\{e\}))\{d\}$$

Zapis `lo<0_n>` jest nieoczywisty, jednak konieczny do reprezentacji pętli, które mogą być przerywane na innej aktywności, niż kończy się ich pojedyncza iteracja. Poniższy przykład pokazuje model, który ciężko opisać przy pomocy podstawowych bramek logicznych:



Rys. 3.2. Przykład problemu z pętlą

Jest to możliwe za pomocą słowa – `lop` oznacza pętlę:

$$\{a\} \text{and} (\{b\}\{c\})\{d\} \text{lop}(\{e\} \text{and} (\{b\}\{c\})\{d\}) \text{xor}(\{f\}\{g\})$$

Użycie powyższego zapisu jest poprawne, jednak kodowanie pętli w ten sposób sprawia, że powstałe słowo jest skomplikowane, a jego wyewoluowanie mało prawdopodobne. Problem ten rozwiązano, używając zapisu $lo<0_n>$, gdzie $<0_n>$ oznacza, ile znaków ma być pominięte w pierwszej iteracji pętli, dzięki czemu możliwe jest zakodowanie takiej pętli przy użyciu znacznie mniejszej liczby symboli, co ułatwia wyewoluowanie takiego modelu. Ten sam model opisany za pomocą stworzonej gramatyki wygląda następująco:

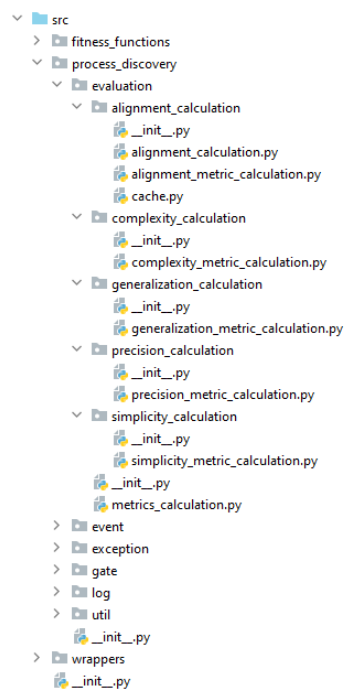
$$\{a\}lo1(\{e\}and(\{b\}\{c\})\{d\})xor(\{f\}\{g\})$$

3.3. Projekt systemu

W tej sekcji pokazano, w jaki sposób podzielono program na moduły oraz przedstawiono modele najważniejszych klas w nim używanych.

3.3.1. Podział na moduły

Sposób podziału programu przedstawiono na rys. 3.3.



Rys. 3.3. Podział na moduły

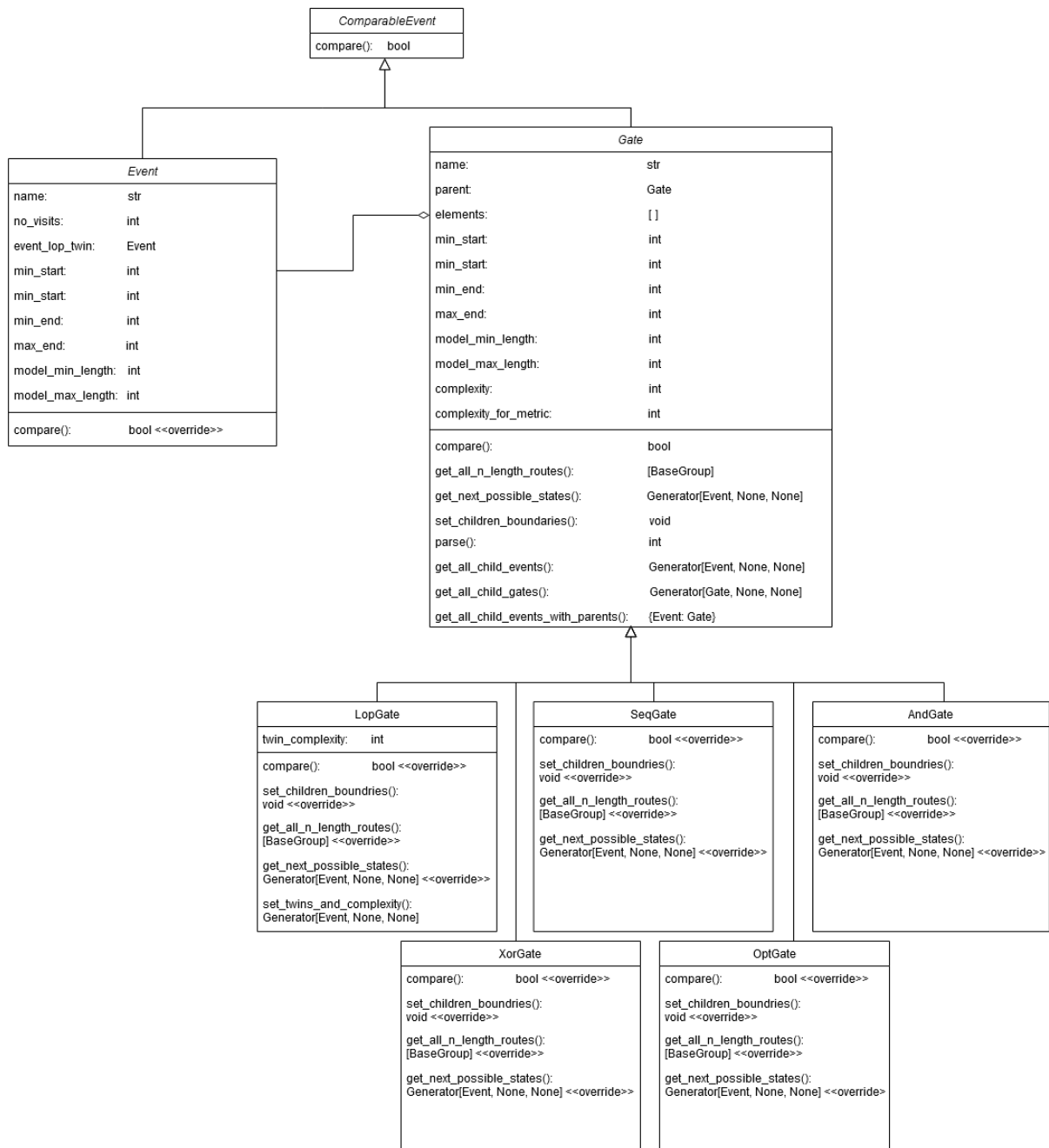
Główne moduły to:

- wrappers – PonyGE2 nie zostało przystosowane do zaimportowania go jako bibliotekę, dlatego, żeby oddzielić kod PonyGE2 od logiki odkrywania procesów biznesowych, w tym module rozszerzono lub nadpisano część z modułów tej biblioteki. Dodano także rozszerzenia do PonyGE2 dodające nowe, brakujące funkcjonalności.

- `fitness_functions` – moduł, w którym znajduje się klasa do obliczania funkcji dopasowania, która korzysta z metod w module `process_discovery`.
- `process_discovery` – moduł zawiera całą logikę parsowania modelu i obliczenia metryk.

3.3.2. Model

Klasa *Gate* i klasy po niej dziedziczące przedstawione na rys. 3.4 są reprezentacją bliższą realnemu modelowi.



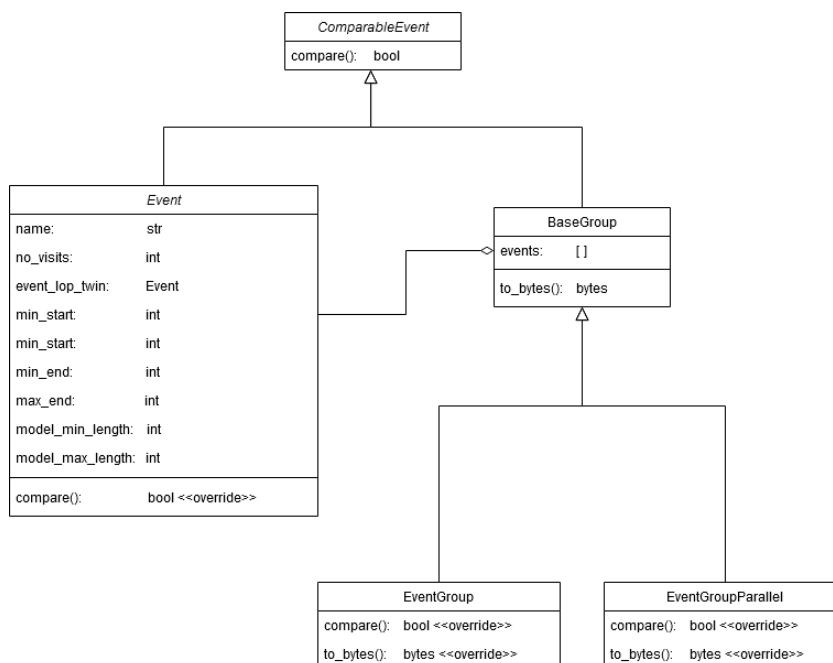
Rys. 3.4. Klasa *Gate* i klasy po niej dziedziczące reprezentowane jako UML

Zdecydowano się na podział na dwie reprezentacje modelu procesu biznesowego wykorzystywane na różnym etapie przetwarzania. Wszystkie klasy implementują interfejs *ComparableEvent* pozwalający na ich porównywanie definiowanych przez nie obiektów. Aktywności są reprezentowane przez obiekty *Event*, które przechowują także informacje o ilości przejść w modelu przez dane zdarzenie, potrzebną do obliczenia generalizacji.

Model w formie ciągu znaków musi być zamieniony na formę, na której łatwiej będzie operować. Zakodowane zgodnie z zasadami gramatyki słowo zostaje sparsowane w metodzie *parse()* klasy *Gate* na obiekty klas po niej dziedziczących odpowiadające poszczególnym bramkom logicznym. Obiekty posiadają wskaźnik na swojego rodzica, czyli bramkę – obiekt *Gate*, w której się znajdują oraz na bramki lub aktywności – obiekty *Event*, które zawierają. Przechowują też leniwie obliczaną informację o złożoności. Najważniejsze metody, które klasy dziedziczące po *Gate* muszą nadpisać to:

- *get_next_possible_states()* – zwraca jako generator możliwe kolejne aktywności, co jest potrzebne przy liczeniu precyzji.
- *get_all_n_length_routes()* – zwraca możliwe ścieżki w modelu o danej długości jako tablicę obiektów *BaseGroup*, co jest potrzebne przy liczeniu odwzorowania.

Obliczanie metryk dla klasy *Gate* byłoby utrudnione z uwagi na dużą ilość bramek logicznych, dlatego konieczne jest przerobienie tych obiektów na uproszczoną formę pośrednią. Są nią obiekty klas dziedziczących po *BaseGroup* (rys. 3.5), które dzielą się pod względem tego, czy aktywności w nich zgrupowane mogą być wykonywane w dowolnej kolejności – *EventGroupParallel* czy muszą być wykonywane kolejno po sobie – *EventGroup*. Są to wystarczające informacje do obliczenia odwzorowania, a dzięki temu algorytm ten jest prostszy. Takie rozgraniczenie pozwala również na dodawanie nowych bramek logicznych bez konieczności zmieniania metody obliczania odwzorowania.



Rys. 3.5. Klasa *BaseGroup* i klasy po niej dziedziczące reprezentowane jako UML

Pojedyncze aktywności lub ich grupy są przechowywane jako tablica. Jedyna metoda, która musi zostać nadpisana w klasach rozszerzających *BaseGroup* to *to_bytes()* potrzebna do przechowywania obiektów w pamięci podręcznej (ang. *cache*).

3.4. Implementacja

W tej części przedstawiono listingi z pseudokodem opartym na języku Python. Tam, gdzie to konieczne pozostawiono słowa kluczowe oraz operatory tego języka.

3.4.1. Parsowanie gramatyki

Parser pozwala na przetworzenie wyników uzyskanych na drodze ewolucji gramatycznej na postać, na której łatwiej będzie operować. Rezultaty uzyskane na drodze ewolucji gramatycznej w PonyGE2 są w formie tekstowej, z którą praca byłaby niewygodna, dlatego używamy parsera, żeby otrzymać wynik w postaci zagnieżdżonych obiektów *Gate*, które zawierają obiekty *Event*.

Metoda *parsuj*, której pseudokod znajduje się na listingu 3.2, należy do obiektu *Gate* i bezpośrednio modyfikuje obiekt, na którym jest wywoływana. Argumentem metody jest wyrażenie wygenerowane w procesie ewolucji. Zwracana jest natomiast ilość przeparsowanych znaków. To na tym etapie odrzucamy też procesy, które, mimo że gramatyka pozwala na ich stworzenie, nie mają sensu z punktu widzenia biznesowego. Pozwala to na ograniczenie zbędnego wykorzystania zasobów i niekontynuowanie obliczeń dla modeli, które są bezwartościowe. Na przykład takie, które pozwalają na posiadanie w jednej bramce dwóch takich samych aktywności. Parsując, korzystamy z faktu, że przy projektowaniu gramatyki wszystkie bramki logiczne zostały oznaczone trzyliterowymi symbolami, a wszystkie aktywności otoczone są nawiasami klamrowymi. Pasowanie bramek można podzielić na trzy przypadki:

- Bramki *seq* wewnątrz bramek *lop* lub *seq* są redundantne i mogą zostać pominięte.
- Bramki *lop* ze względu na konieczność specjalnego parsowania z powodu problemu opisanego w sekcji 3.2.
- Pozostałe przypadki.

```
def parsuj(wyrazenie: str) -> int:

    for i in range(długość_wyrazenia):
        if wyrażenie[i] == "{":
            zdarzenie := Event(wyrażenia[i + 1])
            dodaj_zdarzenie_do_aktualnie_parsowanej_bramki
            i += 2
        elif wyrażenie[i] == ")":
            return i+1
        elif i+4 < długość_wyrażenia:
            if wyrażenie[i:i + 3] == "seq" and
                (self.name == "seq" or self.name == "lop"):
```

```

        # pomiń zbędne bramki
        i += 3
        przeparsowane_znaki = bramka.parsuj(wyrazenie[i+4:])
        i += ilość_przeparsowanych_znaków
    else:
        if wyrażenie[i:i+2] == 'lo' and wyrażenie[i:i+3] != 'lop':
            bramka := stwórz_nową_bramkę_Gate_typu_zgodnego_z_wyrazeniem
            i += 3
            przeparsowane_znaki = bramka.parsuj(wyrazenie[i+4:])
            if self.name == "seq" or self.name == "lop":
                if int(wyrazenie[i+2]) <= długość(bramka.elementy):
                    for x in bramka.elementy[int(wyrazenie[i+2]):]:
                        self.dodaj_element(x)
            dodaj_zdarzenie_do_aktualnie_parsowanej_bramki
            i += ilość_przeparsowanych_znaków
        else:
            bramka := stwórz_nową_bramkę_Gate_typu_zgodnego_z_wyrazeniem
            i += 3
            przeparsowane_znaki = bramka.parsuj(wyrazenie[i+4:])
            dodaj_zdarzenie_do_aktualnie_parsowanej_bramki
            i += ilość_przeparsowanych_znaków
    else:
        wyrzucić_wyjątek

```

Listing 3.2. Pseudokod parsera gramatyki

3.4.2. Obliczanie metryk

Argumentami metody *oblicz_metryki*, której pseudokod znajduje się na listingu 3.3, są obiekt *LogInfo* zawierający dane i metody dotyczące wariantów, model, czyli obiekt *Gate*, najkrótsza i najdłuższa dozwolona długość modelu obliczane na podstawie parametru podanego w konfiguracji programu, dzięki czemu możliwe jest zmniejszenie ilości obliczeń oraz *cache*. Zwracana jest natomiast średnia ważona metryk, czyli wartość funkcji dopasowania. Metryką, która nie wymaga czasochłonnego obliczenia odwzorowania, jest prostota, dlatego możemy ją obliczyć wcześniej, co przy niskim wyniku pozwala na wstępne odrzucenie części rezultatów. Łatwo można zauważyć, że jeżeli zdarzenie znajduje się w logu, a nie znajduje się w modelu, odwzorowanie nie będzie dobre. Pozwala to przerwać obliczenia, jeżeli stosunek wspólnych zdarzeń w logu i modelu jest mniejszy niż skonfigurowany parametr. Pozostałe metryki wymagają już obliczenia odwzorowania i są obliczane dla najlepiej odwzorowanej ścieżki.

Odwzorowanie obliczane jest osobno dla każdego wariantu w logu, który jest reprezentowany jako tablica znaków. Jeśli błąd odwzorowania wynosi 0, to ilość wystąpień danego wariantu konieczna do obliczenia precyzji jest zapisywana w słowniku. Dodawana do każdej aktywności jest też ilość jej dotychczasowych wystąpień potrzebna we wzorze na generalizację.

Po uzyskaniu tych informacji dla wszystkich wariantów obliczamy średnią ważoną metryk zgodnie ze wzorami w sekcji 2.4.3 i uzyskujemy w ten sposób wartość funkcji dopasowania.

```

def oblicz_metryki(log_info, model, najkrótsza_dozwolona_długość,
                  najdłuższa_dozwolona_długość, cache) -> int:

    lista_zdarzeń_w_modelu = model.zwróć_listę_zdarzeń_w_modelu()
    metryki['PROSTOTA'] := oblicz_metrykę_prostota(lista_zdarzeń_w_modelu,
                                                  unikalne_zdarzenia_w_logu)

    if metryki['PROSTOTA'] < MINIMALNY_PRÓG_PROSOTY:
        return 0
    stosunek_wspólnych_zdarzeń_w_logu_i_modelu :=
        oblicz_stosunek_wspólnych_zdarzeń_w_logu_i_modelu(lista_zdarzeń_w_modelu,
                                                         unikalne_zdarzenia_w_logu)

    if stosunek_wspólnych_zdarzeń_w_logu_i_modelu <
        MINIMALNY_STOSUNEK_WSPÓLNYCH_ZDARZEŃ_W_LOGU_I_MODELU:
        return stosunek_wspólnych_zdarzeń_w_logu_i_modelu/10

    idealnie_dopasowane_logi := pusty_słownik
    skumulowany_błąd := 0

    for wariant in log:
        minimalny_błąd_odwzorowania, najlepiej_dopasowane_zdarzenia,
        najlepsza_ścieżka := oblicz_odwzorowanie_dla_jednego_wariantu(wariant, model,
                                                                    najkrótsza_dozwolona_długość,
                                                                    najdłuższa_dozwolona_długość, cache)
        if minimalny_błąd_odwzorowania == 0:
            idealnie_dopasowane_logi[najlepiej_dopasowane_zdarzenia] :=
                log_info[wariant].ilość_wystąpień
            dodaj_wystąpienia(lista_zdarzeń_w_modelu, najlepiej_dopasowane_zdarzenia,
                             log_info[wariant].ilość_wystąpień)

    metryki := oblicz_metryki
    fitness := oblicz_średnią_ważoną_metryk
    return fitness

```

Listing 3.3. Pseudokod metody oblicz metryki

3.4.3. Obliczanie odwzorowania dla pojedynczego wariantu

Procedurę obliczenia odwzorowania można podzielić następująco:

- Znalezienie ścieżek o długości **n** w modelu.
- Przerobienie ścieżek na postać *BaseGroup*.
- Obliczenie odwzorowania.

Argumentami metody *oblicz_odwzorowanie_dla_jednego_wariantu*, której pseudokod znajduje się na listingu 3.4, są wariant, model, najkrótsza i najdłuższa dozwolona długość modelu oraz *cache*. Zwracane są natomiast minimalny błąd odwzorowania jako liczba całkowita niedodatnia, najlepiej dopasowane zdarzenia jako tablica obiektów *Event* oraz najlepsza ścieżka jako obiekt *BaseGroup*.

Algorytm obliczania odwzorowania wymaga ścieżek o stałej, określonej długości. Ważne jest, żeby jak najbardziej ograniczyć czas potrzebny na znalezienie najlepszej ścieżki, dlatego obliczanie odwzorowania rozpoczynamy od *n* równego długości wariantu. Jeśli *n* jest różne od długości ścieżki, błąd odwzorowania zawsze będzie równy przynajmniej różnicy tych wartości. Jednak wciąż może być lepszy niż aktualnie najmniejszy, więc obliczamy odwzorowanie kolejno dla ścieżek o długości *n*-1, *n*+1, *n*-2, *n*+2... aż do momentu, dopóki jest możliwe uzyskanie mniejszego błędu lub zostanie osiągnięty limit, do którego w konfiguracji zezwolono na szukanie.

Następnie obliczane jest najwcześniejsze i najpóźniejsze wystąpienie danej aktywności w modelu, co ułatwi dalsze obliczenia. W kolejnym kroku znajdowane są wszystkie ścieżki o długości *n* w modelu i są one sortowane względem procentu wspólnych zdarzeń w modelu i logu. Można z tego wywnioskować, jakie najlepsze odwzorowanie można otrzymać dla danej ścieżki i ewentualnie jeśli przekroczona jest dopuszczalna tolerancja błędu odwzorowania lub nie jest możliwe już zmniejszenie błędu przerwanie obliczeń dla niej. W końcu zgodnie z kolejnością po sortowaniu obliczane jest odwzorowanie i jeśli błąd jest mniejszy niż aktualnie najmniejszy, zamieniane są wartości minimalnego błędu odwzorowania, najlepiej dopasowane zdarzenia i najlepsza ścieżka, a jeśli błąd wynosi 0, algorytm jest przerywany i te wartości są zwracane.

```
def oblicz_odwzorowanie_dla_jednego_wariantu(wariant, model,
                                             najkrótsza_dozwolona_długość,
                                             najdłuższa_dozwolona_długość, cache):
    długość_wariantu := oblicz_długość(wariantu)
    n := długość_wariantu
    i := 1
    minimalny_błąd_odwzorowania := -(długość_wariantu + model.minimalna_długość)
    najlepiej_dopasowane_zdarzenia := []
    najlepsza_ścieżka := []
    dolny_limit_osiągnięty := False
    górny_limit_osiągnięty := False

    while not (dolny_limit_osiągnięty and górny_limit_osiągnięty):
        if n >= min(oblicz_maksymalna_dozwolona_długość(długość_procesu),
                  długość_procesu - minimalny_błąd_odwzorowania):
            górny_limit_osiągnięty := True
            n += (-i if i % 2 == 1 else i); i += 1
            continue
        if n <= max(oblicz_minimalna_dozwolona_długość(długość_wariantu),
                  długość_wariantu + minimalny_błąd_odwzorowania):
            dolny_limit_osiągnięty := True
            n += (-i if i % 2 == 1 else i); i += 1
            continue
```

```

if najkrótsza_dozwolona_długość <= n <= najdłuższa_dozwolona_długość:
    ustaw_najwcześniejsze_i_najpóźniejsze_wystąpienie_zdarzenia(model, n)
    ścieżki = model.znajdź_wszystkie_ścieżki_długości_n(n, wariant)
    if ścieżki istnieją:
        for ścieżka in ścieżki:
            procent_wspólnych_zdarzeń := oblicz_procent_wspólnych_zdarzeń_
                w_modelu_i_logu(ścieżka, wariant)
            if procent_wspólnych_zdarzeń >= 1 - TOLERANCJA_BŁĘDU_ODWZOROWANIA:
                dodaj_ścieżkę do lista_ścieżek_do_obliczenia
        posortowane_ścieżki := posortuj lista_ścieżek_do_obliczenia
        for ścieżka in posortowane_ścieżki:
            if procent_wspólnych_zdarzeń <= 1 + minimalny_błąd_odwzorowania /
                długość_wariantu:
                break
            błąd_odwzorowania, najlepiej_dopasowane_zdarzenia :=
                oblicz_odwzorowanie(ścieżka, wariant, cache)
            if błąd_odwzorowania > minimalny_błąd_odwzorowania:
                minimalny_błąd_odwzorowania := błąd_odwzorowania
                najlepiej_dopasowane_zdarzenia := dopasowane_zdarzenia
                najlepsza_ścieżka := ścieżka
            if błąd_odwzorowania == 0:
                return minimalny_błąd_odwzorowania, najlepiej_dopasowane_
                    zdarzenia, najlepsza_ścieżka
    n += (-i if i % 2 == 1 else i); i += 1
return minimalny_błąd_odwzorowania, najlepiej_dopasowane_zdarzenia,
    najlepsza_ścieżka

```

Listing 3.4. Pseudokod obliczania odwzorowania dla jednego wariantu

3.4.4. Wyszukiwanie w modelu ścieżek o określonej długości

Łatwiejszym niż obliczenie odwzorowania dla modelu składającego się z bramek logicznych jest znalezienie najpierw w modelu wszystkich ścieżek o określonej długości i następnie jego obliczenie dla nich. Algorytm służący do tego jest kolejno wywoływany dla wszystkich bramek – podmodeli, a następnie na podstawie ścieżek znalezionych w podmodelach są tworzone ścieżki dla całego modelu. Implementacja różni się w zależności od przeszukiwanej bramki logicznej. Poniżej zaprezentowano przykład dla bramki *and*.

Argumentami metody *znajdź_wszystkie_ścieżki_długości_n*, której pseudokod znajduje się na listingu 3.5, są długość szukanej ścieżki oraz wariant jako tablica znaków potrzebny wyszukiwania ścieżek dla obiektu *LopGate*. Zwracana jest natomiast lista wszystkich ścieżek o określonej długości jako obiekty *BaseGroup*, a w przypadku błędu *None*.

Bramki mogą zawierać różną ilość elementów, dlatego należy obliczyć minimalne i maksymalne długości, czyli ilość zdarzeń dla wszystkich dzieci. Jeśli dziecko jest obiektem *Event*, wtedy dodawane jest bezpośrednio do listy rezultatów. W innym wypadku, kiedy jest obiektem *Gate*, na podstawie długości dzieci obliczany jest dolny i górny limit długości, dla jakich zostanie dla danego elementu wywołana

metoda *znajdź_wszystkie_ścieżki_długości_n*. Dzięki temu może znacznie ograniczyć ilość długości, dla jakich trzeba przeszukiwać podmodele. Wszystkie znalezione ścieżki są dodawane do listy, a całość do globalnej listy.

Rezultat nie może być jednak zagnieżdżoną listą, więc musi ona zostać przerobiona na listę jednowymiarową rozwiązań. Każda lista składa się 2-wymiarowej listy ścieżek dla każdego z elementów modelu. Ścieżki każdego podmodelu są łączone ze ścieżkami kolejnych pod modeli każda z każdą, żeby utworzyć możliwe przejścia dla całego modelu. Celem jest znalezienie tylko tych o długości **n**, więc pozostałe odrzucamy. Na końcu, jako że jest to bramka „and wszystkie ścieżki o długości większej niż 1 są opakowywane w *EventGorupParallel*, żeby zachować informacje o tym, że zdarzenia mogą być wykonywane w dowolnej kolejności.

```
def znajdź_wszystkie_ścieżki_długości_n(n, wariant) -> [BaseGroup]:
    if n == 0:
        return []
    if self.minimalna_długość_modelu < n or n < self.maksymalna_długość_modelu:
        return None

    minimalne_długości := self.oblicz_minimalne_długości_dzieci()
    maksymalne_długości := self.oblicz_maksymalne_długości_dzieci()
    globalna_lista := []

    for element in self.elementy:
        lokalna_lista := []
        if isinstance(element, Event):
            lokalna_lista.dodaj(elem)
            minimalne_długości.usuń_na_pozycji(0)
            maksymalne_długości.usuń_na_pozycji(0)
        else:
            dolny_limit, górny_limit :=
                self.oblicz_docelowy_zakres(n, globalna_lista, minimalne_długości,
                                            maksymalne_długości)
            for i in range(dolny_limit, górny_limit + 1):
                try:
                    wszystkie_ścieżki_dziecka_o_długości_n :=
                        element.znajdź_wszystkie_ścieżki_długości_n(i, wariant)
                except ValueError:
                    return None
                if wszystkie_ścieżki_dziecka_o_długości_n is not None:
                    lokalna_lista.dodaj(wszystkie_ścieżki_dziecka_o_długości_n)

            if lokalna_lista:
                globalna_lista.dodaj(lokalna_lista)

    ścieżki = []
    if globalna_lista:
        for element in spłaszczyć_listę(globalna_lista):
            if self.sprawdź_długość(n, elem):
```

```

        if n == 1:
            ścieżki.dodaj(element[0])
        else:
            ścieżki.dodaj(EventGroupParallel(element))
    if ścieżki:
        return ścieżki
    else:
        return None

```

Listing 3.5. Pseudokod wyszukiwania procesów o długości n

3.4.5. Obliczanie odwzorowania

Pomysł został zaczerpnięty z algorytmu Needleman-Wunsch [43], który jest uogólnieniem odległości Levenshteina dla dowolnych wartości błędów. Stworzona zostaje macierz o wymiarach długość modelu i długość logu, w której obliczana jest najmniejsza suma błędów. Algorytm został rozwinięty o możliwość przeszukiwania modelu rekurencyjnie oraz o możliwość podawania listy równoległych zdarzeń.

Podstawowy algorytm można opisać za pomocą czterech kroków dla każdego zdarzenia:

1. Oblicz wartość w poprzednim wierszu i kolumnie zsumowaną z błędem „odwzorowanie” lub „brak odwzorowania”.
2. Oblicz wartość w poprzednim wierszu zsumowaną z błędem „przerwa”.
3. Oblicz wartość w poprzedniej kolumnie zsumowaną z błędem „przerwa”.
4. Wybierz najmniejszą wartość.

Jego działanie dla przykładu z sekcji 2.4.4.2 przedstawiono na rys. 3.6. Na osi x znajduje się wariant z logu, a na osi y jest ścieżka w modelu.

		A	F	D
	0	-1	-2	-3
A	-1	0	← -1	← -2
B	-2	↑ -1	← -2	← -3
E	-3	↑ -2	← -3	← -4
D	-4	↑ -3	← -4	← -3

Rys. 3.6. Klasyczny algorytm Needleman-Wunsch [44]

Argumentami metody *oblicz_odwzorowanie*, której pseudokod znajduje się na listingu 3.6, są model jako obiekt *Gate* oraz wariant jako tablica znaków. Zwracane są natomiast ostatni wiersz, który zawiera błąd odwzorowania modelu wraz z pośrednimi błędami dla wszystkich, zawsze zaczynając od pierwszego zdarzenia, możliwych długości wariantu, co jest potrzebne, jeśli model zawiera podmodele oraz najlepiej dopasowana ścieżka. Metoda opakowana jest w metodę, która jeśli dla danego modelu i wariantu zostało już obliczone odwzorowanie, zwraca rozwiązanie z pamięci podręcznej (ang. *cache*) bez powtarzania obliczeń.

Zgodnie z 2.4.4.2 dla brakującego zdarzenia błąd wynosi 1, a jeśli zdarzenia się nie zgadzają, wynosi 2. Stworzona macierz jest inicjalizowana zerami, a następnie pierwsza kolumna jest wypełniana stosownymi wartościami błędu. Są trzy opcje – normalne obliczenie zgodne z klasycznym algorytmem, a także sytuacja, w której model zawiera podmodele, gdzie algorytm jest powtórnie wywoływany dla podmodelu i dla wszystkich zawsze zaczynając od pierwszego zdarzenia możliwych długości wariantu lub gdy zawiera zdarzenia równoległe – *EventGroupParallel* wtedy stosujemy inny algorytm, który bezpośrednio porównuje wszystkie zdarzenia w wariantcie ze zdarzeniami w obiekcie *EventGroupParallel*.

```
def oblicz_odwzorowanie(model, wariant):
    bład := {'ODWZOROWANIE': 0, 'BRAK_ODWZOROWANIA': -2, 'PRZERWA': -1}
    ilość_wierszy = długość(model) + 1
    ilość_kolumn = długość(wariant) + 1
    najlepiej_dopasowane_ścieżki_podmodeli := [None] * ilość_wierszy
    macierz_rozwiązań := zainicjalizuj_macierz_zerami()

    for j in range(ilość_kolumn):
        macierz_rozwiązań[0][j] := bład['PRZERWA'] * j

    for i in range(1, ilość_wierszy):
        if jest_podmodelem(model[i-1]):
            macierz_rozwiązań[i], najlepiej_dopasowane_ścieżki_podmodeli[i] :=
                odwzorowanie_podmodeli(macierz_rozwiązań[i - 1], model[i - 1],
                                         [x for x in odwrócone_substringi(wariant)], i)
        elif długość(model[i-1]) > 1:
            macierz_rozwiązań[i], najlepiej_dopasowane_ścieżki_podmodelu[i] :=
                odwzorowanie_równoległe(macierz_rozwiązań[i - 1], model[i - 1],
                                         [x for x in odwrócone_substringi(wariant)], kara, i)
        else:
            macierz_rozwiązań[i][0] := macierz_rozwiązań[i-1][0] + kara['PRZERWA']
            odwzorowanie(macierz_rozwiązań, model[i - 1], wariant, kara, i, ilość_kolumn)

    najlepiej_dopasowana_ścieżka :=
        znajdź_ścieżkę(macierz_rozwiązań, bład['PRZERWA'], model, wariant,
                       najlepiej_dopasowana_ścieżka_podmodelu)

    return macierz_rozwiązań[ilość_wierszy-1], najlepiej_dopasowana_ścieżka
```

Listing 3.6. Pseudokod obliczania odwzorowania

3.4.6. Znajdowanie najlepiej dopasowanych aktywności w modelu

Znalezienie najlepiej dopasowanych aktywności w modelu jest potrzebne do obliczenia precyzji oraz generalizacji. Algorytm obliczający odwzorowanie zwraca najmniejszy błąd, ale nie daje informacji o tym, dla jakiej ścieżki otrzymano ten błąd. Dodatkowo znajdowanie najlepiej dopasowanych aktywności w modelu jest utrudnione przez fakt, że modele może składać się z podmodeli – *BaseGroup*.

Argumentami metody *znajdź_ścieżkę*, której pseudokod znajduje się na listingu 3.7, są kopia macierzy rozwiązań, model, kopia wariantu oraz rozwiązania podmodeli. Zwracana jest natomiast najlepiej dopasowana ścieżka.

Algorytm zaczyna znajdowanie ścieżki od ostatniego elementu i cofa się do początku. W komórkach, dla których znaleziono dopasowanie, wpisywane jest 0. Tak jak poprzednio, są trzy możliwości: brak zdarzenia w modelu, w logu, oraz zupełna niezgodność. Ścieżki szuka się poprzez porównywanie wartości w komórce z sumą błędów z wartościami odpowiednich kolumn. Znaną ścieżkę pokazano na rysunku 3.6. Uwzględnione muszą być dwie sytuacje – taką, w której dana pozycja została obliczona dla podmodelu lub nie.

W tym drugim przypadku, jeśli zdarzenie zostało pominięte w modelu, wpisywane jest do ścieżki *None*, natomiast jeżeli zostało znalezione dopasowanie, to zdarzenie jest dodawane do ścieżki i usuwane z wariantu.

Jeśli dana pozycja została obliczona dla podmodelu, algorytm działa podobnie, największą różnicą jest to, że w takiej sytuacji nie ma jednego zdarzenia tylko kilka i tylko część może się zgadzać. Dlatego znajdowane jest ostatnie *k* zdarzenie w podmodelu dla każdego podwariantu i kolejno porównywane są ich ilości i różnica w błędzie, dzięki czemu można stwierdzić, dla którego podwariantu znaleziono najlepsze odwzorowanie.

```
def znajdź_ścieżkę(macierz_rozwiązań, model, wariant,
                  rozwiązania_podmodeli) -> [Event]:
    ścieżka = []
    i = długość(model)
    j = długość(wariant)

    while i != 0:
        długość_podmodelu = długość(model[i - 1])
        if rozwiązania_podmodeli[i] is not None:
            znaleziono_dopasowanie := False
            if macierz_rozwiązań[i][j] ==
                macierz_rozwiązań[i - 1][j] + długość_podmodelu * błąd['PRZERWA']:
                [ścieżka.dodaj(None) for _ in range(długość_podmodelu)]
                macierz_rozwiązań[i][j] := 0
                i -= 1
            else:
                for k in range(j):
                    zdarzenia := znajdź_nie_none(rozwiazania_podmodeli[i][k])
                    [długość(rozwiazania_podmodeli[i][k]) - (j-k)], wariant)
                if macierz_rozwiązań[i][j] == macierz_rozwiązań[i - 1][k] +
                    (długość_podmodelu + (j-k) - 2*długość(zdarzenia)) * błąd['PRZERWA']:
```

```

        [ścieżka.dodaj(x) for x in odwróć(zdarzenia)]
    for x in zdarzenia:
        wariant = wariant.usuń(x.nazwa)
    [ścieżka.dodaj(None)
     for _ in range(długość_podmodelu - długość(zdarzenia))]
    macierz_rozwiązań[i][j] := 0
    i -= 1
    j = k
    znaleziono_dopasowanie = True
    break
if not znaleziono_dopasowanie:
    if macierz_rozwiązań[i][j] == macierz_rozwiązań[i][j - 1] +
        błąd['PRZERWA']:
        macierz_rozwiązań[i][j] := 0
        j -= 1
else:
    if macierz_rozwiązań[i][j] == macierz_rozwiązań[i - 1][j] + kara:
        ścieżka.dodaj(None)
        macierz_rozwiązań[i][j] := 0
        i -= 1
    elif macierz_rozwiązań[i][j] == macierz_rozwiązań[i][j - 1] + kara:
        macierz_rozwiązań[i][j] := 0
        j -= 1
    elif macierz_rozwiązań[i][j] == macierz_rozwiązań[i - 1][j - 1]:
        ścieżka.dodaj(model[i-1])
        wariant = wariant.usuń(model[i-1].nazwa)
        macierz_rozwiązań[i][j] := 0
        i -= 1
        j -= 1
return ścieżka

```

Listing 3.7. Pseudokod znajdowania ścieżki w modelu

3.4.7. Pozostałe wnioski dotyczące implementacji

Używając algorytmów genetycznych, konieczne jest wielokrotne obliczenie metryk, żeby znaleźć rozwiązanie. Z tego powodu, duży nacisk położono na ograniczenie czasu obliczeń. W wiele miejscach zaimplementowano mechanizm przerywający obliczenia, jeżeli nie dają one perspektyw na znalezienie lepszego niż aktualnie najlepsze rozwiązanie. Użytkownik może też zdefiniować maksymalną złożoność modelu, co przełoży się także na czas jego znajdowania. Duże znaczenie ma też fakt, że algorytm pozwala na równoległe przetwarzanie.

W sytuacji, kiedy wiele obliczeń się powtarza, można znacząco przyspieszyć czas działania aplikacji poprzez przechowywanie danych w pamięci podręcznej (ang. *cache*). W przypadku naszego algorytmu można zauważyć dwa miejsca, w których często dochodzi to powtórzeń:

- Poprzednio obliczone rozwiązanie – model może się powtórzyć. W tym wypadku możemy skorzystać z przechowywania genotypów w pamięci podręcznej, które jest dostarczane przez bibliotekę PonyGE2.
- Podczas obliczania odwzorowania, które jest najbardziej czasochłonne. Zaimplementowano tutaj własną metodę przechowywania wyników w pamięci podręcznej.

Żeby ograniczyć czas pojedynczej iteracji, można wprowadzić ograniczenie czasowe na obliczanie metryk dla danego osobnika. Czas obliczania jest powiązany ze złożonością modelu. Przy właściwym ustawieniu ograniczenia czasowego będzie ono oddziaływało tylko na zbyt złożone rozwiązania i zostanie dla nich zwrócona wartość funkcji dopasowania równa 0.

Tworząc program, nacisk położono na możliwość łatwego rozszerzania i oddzielenie od biblioteki PonyGE2. Dzięki temu zwiększono niezależność od biblioteki i zmian w niej. Program może być też łatwo modyfikowany i ewentualnie usprawniany.

Rozszerzono także możliwość konfiguracji o nowe parametry, jednak aby umożliwić użytkownikowi niski próg wejścia w korzystanie z programu, starano się ograniczyć ilość parametrów potrzebnych do skonfigurowania i tam, gdzie to możliwe postarano się wstawić domyślne wartości, jeśli są one wystarczająco dobre.

3.5. Wybór parametrów algorytmu

Wybór parametrów algorytmu ma ogromny wpływ na jakość i szybkość znalezienia rozwiązania. Jest kilka zasad, którymi należy się kierować przy tym wyborze właśnie. Ilość parametrów wymagana przez ponyGE2 jest duża [45]. Mimo że starano się ograniczyć możliwość konfiguracji, która nie daje dużo korzyści do minimum, tworząc aplikacje, konieczne było dodanie kilku innych niezbędnych parametrów. Poniżej przedstawiono i krótko omówiono niezbędne do działania aplikacji parametry dodatkowe parametry:

*ALIGNMENT_CACHE_SIZE: 32*1024*

Określa wielkość cache przy liczeniu odwzorowania.

DATASET: discovered-processes.csv

Nazwa pliku z wariantami. Potrzebna przy tworzeniu nazwy pliku wynikowego.

MAX_ALLOWED_COMPLEXITY_FACTOR: 300

Maksymalne skomplikowanie modelu. Obliczane jako iloczyn ilości unikalnych aktywności w modelu i powyższego parametru.

MIN_SIMPLICITY_THRESHOLD: 2/3

Minimalna wartość prostoty, powyżej której metryki będą dalej obliczane.

MINIMIZE_SOLUTION_LENGTH: True

Dodaje małą karę za długość rozwiązania, co pozwala usunąć zbędne bramki, nawet jeśli wartość metryk jest taka sama.

RESULT_TOLERANCE_PERCENT: 5

Używany w kilku miejscach w programie. Określa jak złe pod względem wartości metryk modele będą tolerowane i dalej przetwarzane. Zaleca się nie przekraczać wartości 10.

TIMEOUT: 5

Określa czas, po jakim program przestaje obliczać metryki, bo najprawdopodobniej model i tak jest zbyt skomplikowany.

Rekomendowane wagi poszczególnych metryk. Dla małych modeli, kiedy łatwo znaleźć model z odwzorowaniem = 1 warto zwiększyć wagę precyzji, żeby sprawdzić, czy możliwe jest znalezienie precyzyjniejszego modelu, wciąż zachowując perfekcyjne odwzorowanie:

WEIGHT_ALIGNMENT: 8

WEIGHT_COMPLEXITY: 2

WEIGHT_GENERALIZATION: 2

WEIGHT_PRECISION: 2

WEIGHT_SIMPLICITY: 2

4. Dyskusja rezultatów

We wszystkich zaprezentowanych przykładach kierowano się wytycznymi z sekcji 3.5 odnośnie wyboru parametrów startowych algorytmu i wag poszczególnych metryk. Ewolucje przeprowadzano na różnych komputerach stąd różnice w ilości użytych wątków.

4.1. Przykładowe wyniki

4.1.1. Wynik dla przykładu ze wstępu

Na rys. 4.1 znajdują się warianty procesu dla dziennika zdarzeń przedstawionego w sekcji 2.2.3, a dla którego stworzono przykładowy model i obliczono metryki w sekcji 2.4.4.2.

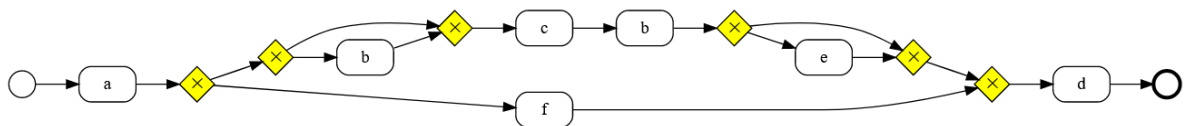
2, a, b, c, b, d
1, a, c, b, d
1, a, c, b, e, d
1, a, f, d

Rys. 4.1. Warianty procesu – przykład ze wstępu

Przy odkrywaniu modelu dla tego wariantu użyto następujących wag poszczególnych metryk: odwzorowanie = 8, złożoność = 2, generalizacja = 2, precyzja = 8, prostota = 2. Model dla tego dziennika zdarzeń znaleziony przy pomocy algorytmu to:

$$\{a\} \text{ xor } (\text{seq}(\text{opt}(\{b\})\{c\}\{b\}\text{opt}(\{e\}))\{f\})\{d\}$$

Model został przedstawiony graficznie na rys. 4.2.



Rys. 4.2. Znaleziony model – przykład ze wstępu

Do znalezienia modelu potrzebne było 179 generacji, podczas których przeszukano 47615 unikalnych osobników. Zajęło to 11.3 minuty, używając 1 wątku procesora. Natomiast, metryki mają następujące wartości:

Średnia ważona: 0.9287

Odwzorowanie: 1.0

Złożoność: 1.0

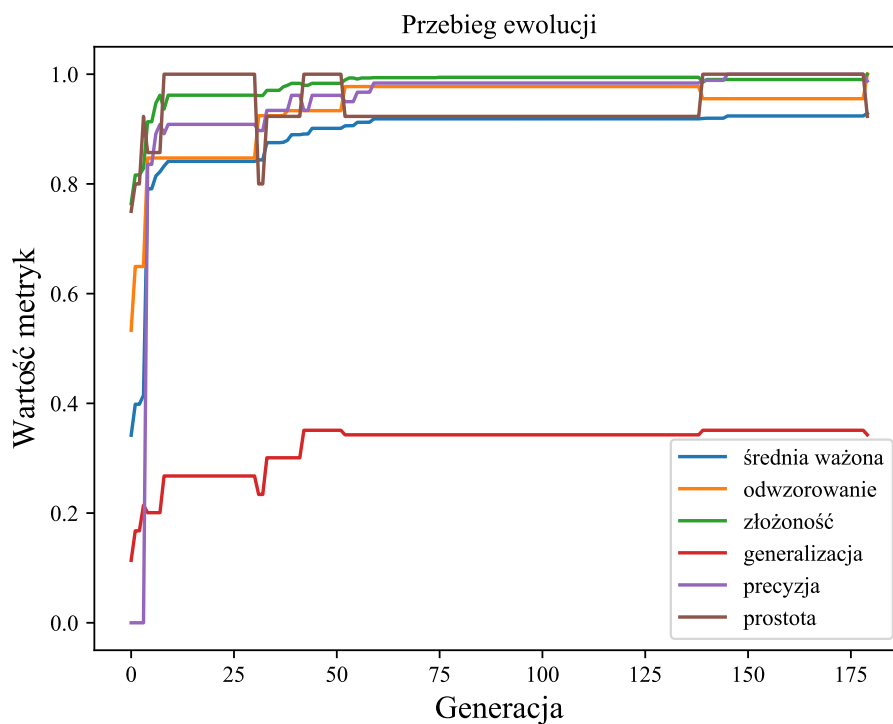
Generalizacja: 0.3426

Precyzja: 0.9875

Prostota: 0.9231

Dla porównania poszczególne metryki obliczone w sekcji 2.4.4.2 wynosiły odwzorowanie = 0.9243, złożoność = 0.9706, generalizacja = 0.3997, precyzja = 0.9465, prostota = 0.8333, a średnia ważona używając przyjętych wag, wyniosłaby 0.8743. Używając algorytmu ewolucyjnego, otrzymano więc znacznie lepszy model.

Na rys. 4.3 znajduje się wykres, na który zaprezentowano zmianę wartości metryk w kolejnych generacjach. Jest on generowany podczas działania algorytmu i kończy się w momencie przerwania działania algorytmu lub po określonej ilości iteracji, a nie znalezienia rozwiązania, gdyż przy algorytmach ewolucyjnych nie można być pewnym, czy znaleziono najlepsze rozwiązanie.



Rys. 4.3. Przebieg ewolucji – przykład ze wstępu

4.1.2. Inne przykłady działania

Przykładowe dziennik zdarzeń wzięto z [16]. Część z nich została wygenerowana sztucznie, a część zawiera realne dane. Następnie przerobiono je na warianty procesu.

4.1.2.1. Przykład #Sztucznie wygenerowany

Prosty, sztucznie wygenerowany przykład. Zawiera 5 unikalnych aktywności, 40 przypadków, 8 wariantów, z których najdłuższy ma 5 zdarzeń. Na rys. 4.4 znajdują się warianty procesu dla tego przykładu.

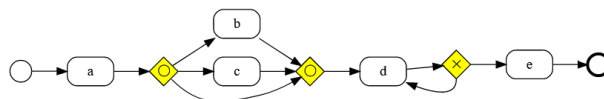
```
5, a, e
10, a, b, c, e
10, a, c, b, e
1, a, b, e
1, a, c, e
10, a, d, e
2, a, d, d, e
1, a, d, d, d, e
```

Rys. 4.4. Warianty procesu – przykład #Sztucznie wygenerowany

Przy odkrywaniu modelu dla tego wariantu użyto następujących wag poszczególnych metryk: odwzorowanie = 8, złożoność = 2, generalizacja = 2, precyzja = 2, prostota = 2. Model dla tego dziennika zdarzeń znaleziony przy pomocy algorytmu to:

$$\{a\}lo2(\{d\})opt(\{b\}\{c\})\{e\}$$

Model został przedstawiony graficznie na rys. 4.5, natomiast przebieg ewolucji na rys. 4.6.



Rys. 4.5. Warianty procesu – przykład #Sztucznie wygenerowany

Do znalezienia modelu potrzebne było 5 generacji, podczas których przeszukano 2369 unikalnych osobników. Zajęło to 66.5 sekund, używając 4 wątków procesora. Natomiast, metryki mają następujące wartości:

Średnia ważona: 0.9722

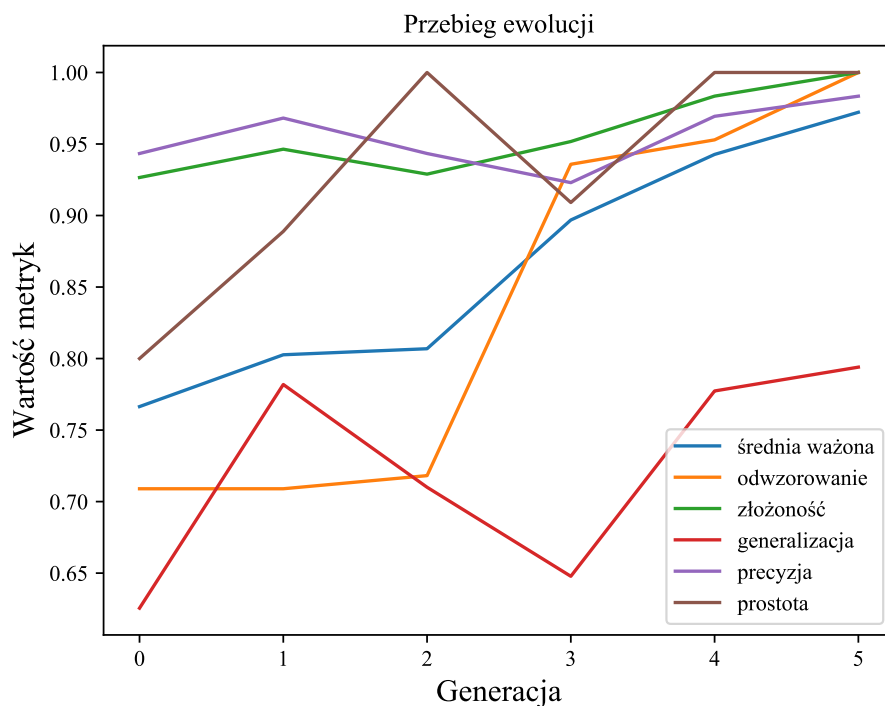
Odwzorowanie: 1.0

Złożoność: 1.0

Generalizacja: 0.7940

Precyzja: 0.9834

Prostota: 1.0



Rys. 4.6. Warianty procesu – przykład #Sztucznie wygenerowany

4.1.2.2. Przykład #Obsługa roszczeń w firmie ubezpieczeniowej

Dziennik danych zawiera dane opisujące obsługę roszczeń w firmie ubezpieczeniowej. Proces może być obsługiwany przez dwie różne działy w Brisbane i Sydney. Możliwe jest połączenie danych dla tych działów, ale nie zrobiono tego, żeby utrudnić odkrywanie modelu. Przykład składa się z 11 unikalnych aktywności, 3512 przypadków, 12 wariantów, z których najdłuższy ma 9 zdarzeń. Na rys. 4.7 znajdują się warianty procesu dla tego przykładu.

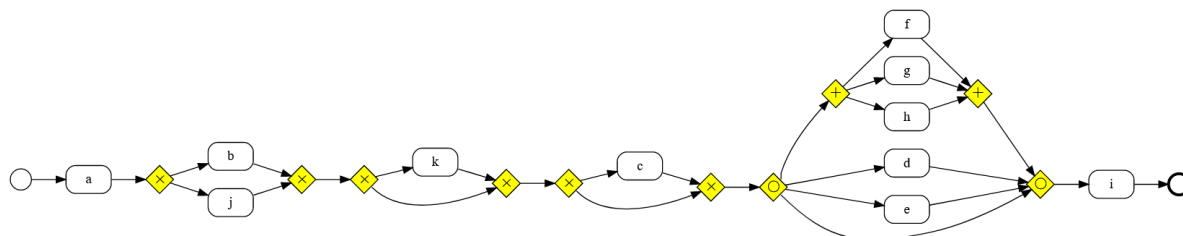
525,a,b,c,d,e,f,g,h,i	a	nadejście zgłoszenia
469,a,j,k,d,e,f,g,h,i	b	sprawdzenie, czy dostępne są wystarczające informacje (Brisbane)
367,a,j,i	c	zarejestrowanie zgłoszenia (Brisbane)
262,a,j,k,d,e,g,h,f,i	d	określenie prawdopodobieństwa roszczenia
260,a,b,c,d,e,i	e	ocena roszczenia
257,a,b,c,d,i	f	doradzenie wnioskodawcy w sprawie zwrotu kosztów
248,a,b,c,d,e,g,h,f,i	g	rozpoczęcie zapłaty
237,a,j,k,d,e,g,f,h,i	h	zamknięcie zgłoszenia
235,a,b,c,d,e,g,f,h,i	i	zarchiwizowanie zgłoszenia
232,a,j,k,d,i	j	sprawdzenie, czy dostępne są wystarczające informacje (Sydney)
225,a,j,k,d,e,i	k	zarejestrowanie zgłoszenia (Brisbane)
195,a,b,i		

Rys. 4.7. Warianty procesu – przykład #Obsługa roszczeń w firmie ubezpieczeniowej

Przy odkrywaniu modelu dla tego wariantu użyto następujących wag poszczególnych metryk: odwzorowanie = 8, złożoność = 2, generalizacja = 2, precyzja = 2, prostota = 2. Model dla tego dziennika zdarzeń znaleziony przy pomocy algorytmu to:

```
{a}opt(xor({j}{b}){k})opt(opt({d}{e}){c})opt(and({f}{h}{g}))i
```

Model został przedstawiony graficznie na rys. 4.8, natomiast przebieg ewolucji na rys. 4.9.



Rys. 4.8. Znalezionej model – przykład #Obsługa roszczeń w firmie ubezpieczeniowej

Do znalezienia modelu potrzebne było 4110 generacji. Zajął to 4.1 godziny, używając 32 wątków procesora. Natomiast, metryki mają następujące wartości:

Średnia ważona: 0.9748

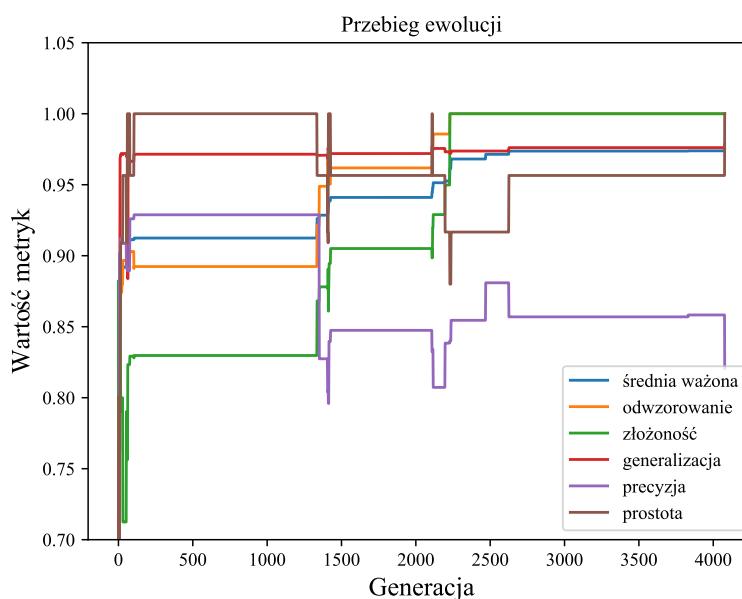
Odzworowanie: 1.0

Złożoność: 1.0

Generalizacja: 0.9782

Precyzja: 0.8204

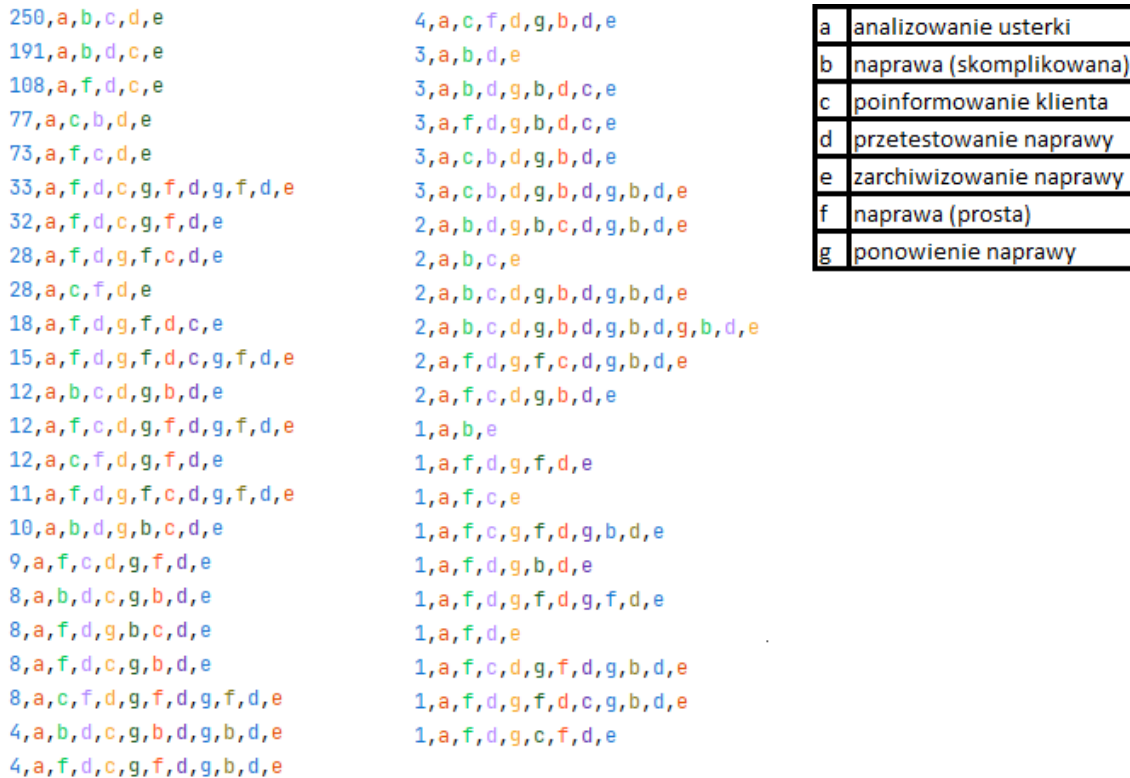
Prostota: 1.0



Rys. 4.9. Przebieg ewolucji – przykład #Obsługa roszczeń w firmie ubezpieczeniowej

4.1.2.3. Przykład #Naprawa telefonu

Dziennik zdarzeń zawiera dane dotyczące procesu naprawy telefonu. Przykład składa się z 7 unikalnych aktywności, 1000 przypadków, 45 wariantów, z których najdłuższy ma 14 zdarzeń. Na rys. 4.10 znajdują się warianty procesu dla tego przykładu.

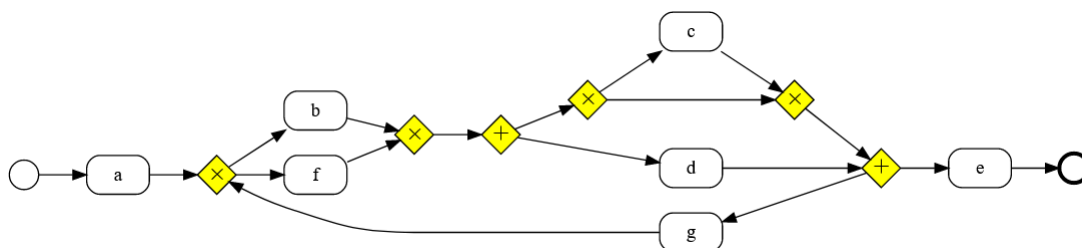


Rys. 4.10. Warianty procesu – przykład #Naprawa telefonu

Przy odkrywaniu modelu dla tego wariantu użyto następujących wag poszczególnych metryk: odwzorowanie = 8, złożoność = 2, generalizacja = 2, precyzja = 4, prostota = 1. Model dla tego dziennika zdarzeń znaleziony przy pomocy algorytmu to:

$$\{a\} \text{ xor } (\{f\} \{b\}) \text{ and } (\{d\} \text{ opt } (\{c\})) \text{ lo } 3(\{g\} \text{ xor } (\{f\} \{b\}) \text{ and } (\{d\} \text{ opt } (\{c\}))) \{e\}$$

Model został przedstawiony graficznie na rys. 4.11, natomiast przebieg ewolucji na rys. 4.12.



Rys. 4.11. Znaleziony model – przykład #Naprawa telefonu

Do znalezienia modelu potrzebne było 436 generacji. Zajęło to 78 minut, używając 32 wątków procesora. Natomiast metryki mają następujące wartości:

Średnia ważona: 0.9859

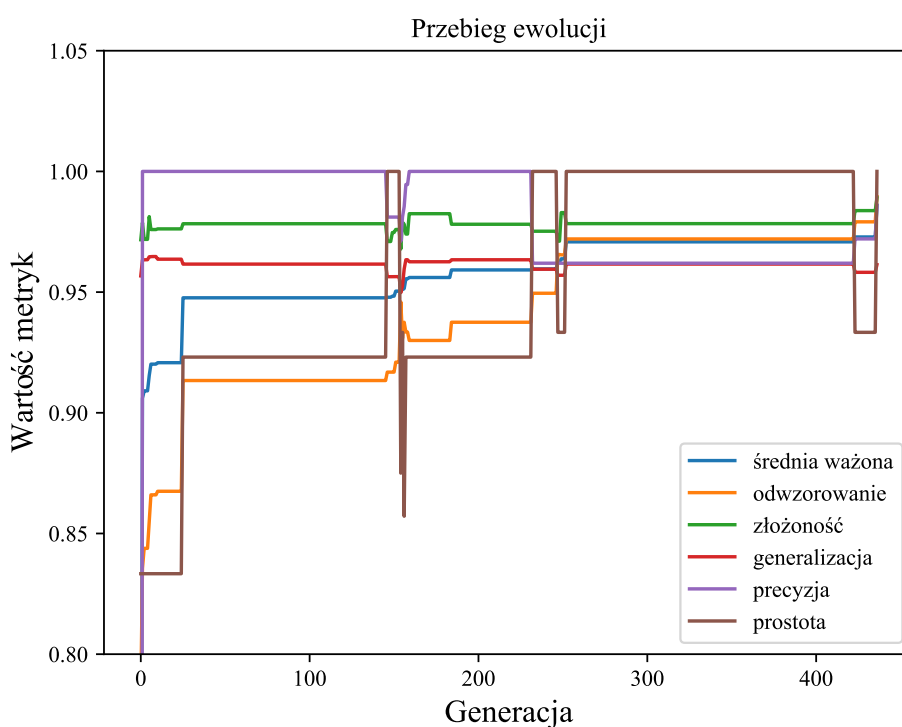
Odwzorowanie: 0.9896

Złożoność: 0.9891

Generalizacja: 0.9614

Precyzja: 0.9859

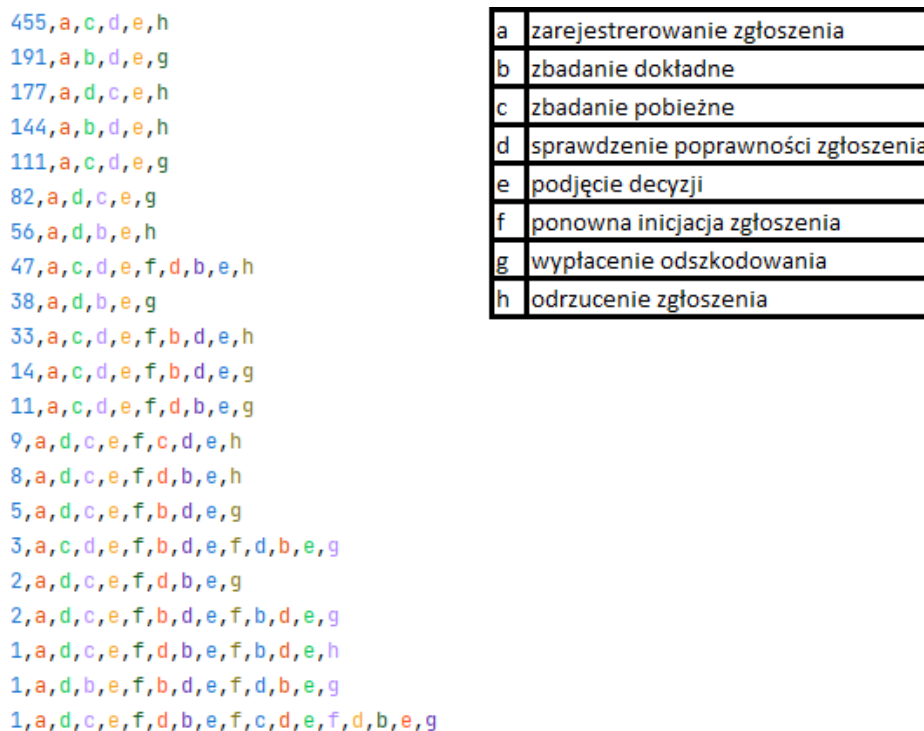
Prostota: 1.0



Rys. 4.12. Przebieg ewolucji dla przykładu #Naprawa telefonu

4.1.2.4. Przykład #Obsługa wniosku o odszkodowanie

Dziennik zdarzeń zawiera dane obsługi wniosku o odszkodowanie. Przykład składa się z 8 unikalnych aktywności, 1391 przypadków, 21 wariantów, z których najdłuższy ma 17 zdarzeń. Na rys. 4.13 znajdują się warianty procesu dla tego przykładu.

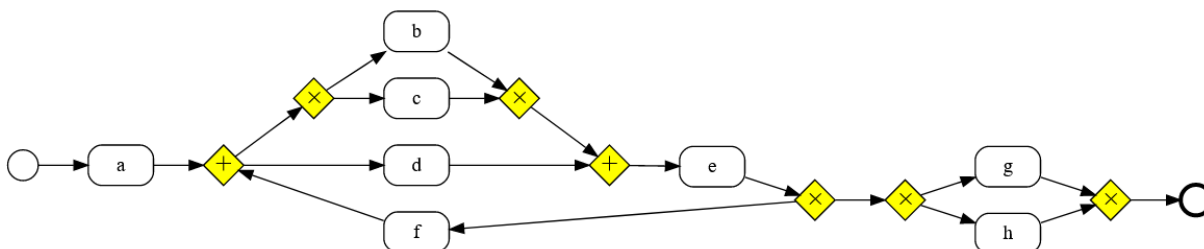


Rys. 4.13. Warianty procesu – przykład #Obsługa wniosku o odszkodowanie

Przy odkrywaniu modelu użyto następujących wag poszczególnych metryk: odwzorowanie = 8, złożoność = 2, generalizacja = 2, precyzja = 2, prostota = 2. Model dla tego dziennika zdarzeń znaleziony przy pomocy algorytmu to:

$$\{a\} \text{lo1}(\{f\} \text{and}(\{d\} \text{xor}(\{b\} \{c\})))\{e\} \text{xor}(\{g\} \{h\})$$

Model został przedstawiony graficznie na rys. 4.14, natomiast przebieg ewolucji na rys. 4.15



Rys. 4.14. Znaleziony model – przykład #Obsługa wniosku o odszkodowanie

Do znalezienia modelu potrzebne było 685 generacji. Zajął to 4.1 godziny, używając 1 wątku procesora. Natomiast, metryki mają następujące wartości:

Średnia ważona: 0.9940

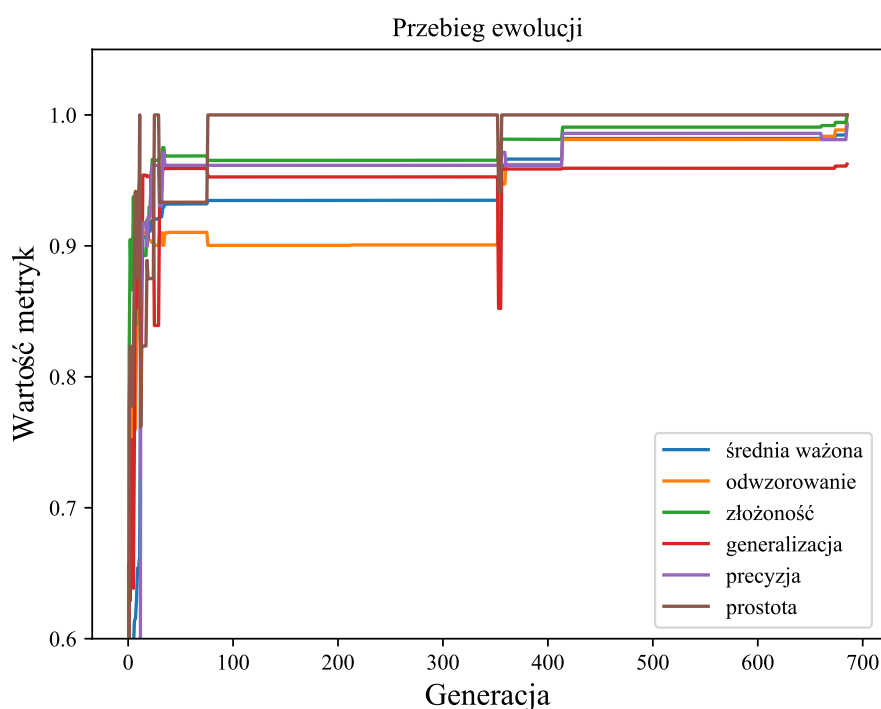
Odwzorowanie: 1.0

Złożoność: 1.0

Generalizacja: 0.9624

Precyzja: 0.9899

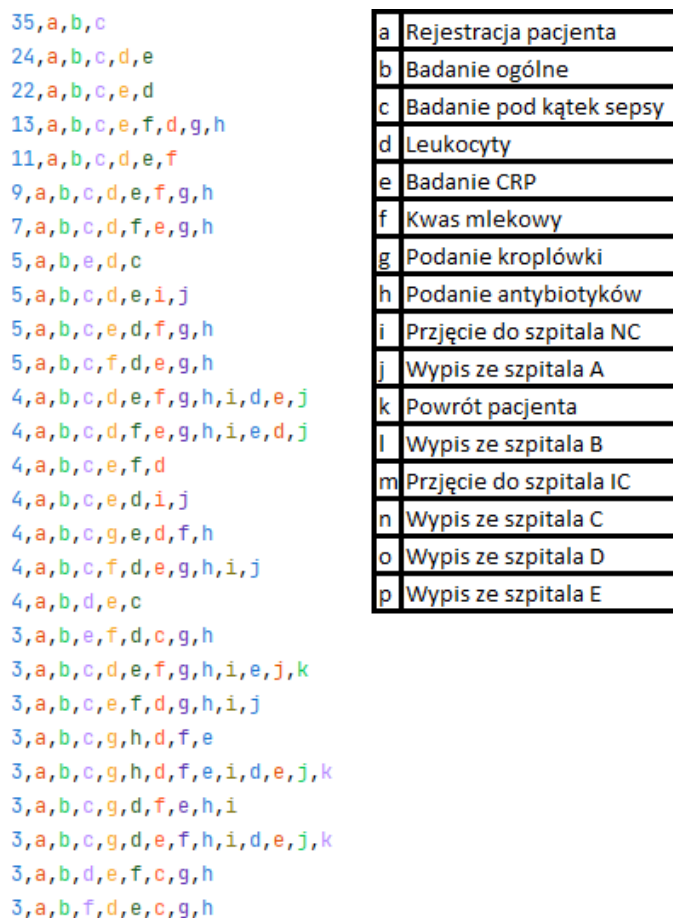
Prostota: 1.0



Rys. 4.15. Przebieg ewolucji – przykład #Obsługa wniosku o odszkodowanie

4.1.2.5. Przykład #Przypadki sepsy

Dziennik zdarzeń zawiera dane dotyczące przypadków sepsy zarejestrowane przez system CRM w anonimowym, holenderskim szpitalu [46]. Przykład składa się z 16 unikalnych aktywności, 1050 przypadków, 846 wariantów, z których najdłuższy ma 185 zdarzeń. Na rys. 4.16 znajdują się warianty procesu dla tego przykładu. Z powodu dużej ilości wariantów przedstawiono jedynie te, które występują więcej niż 2 razy.

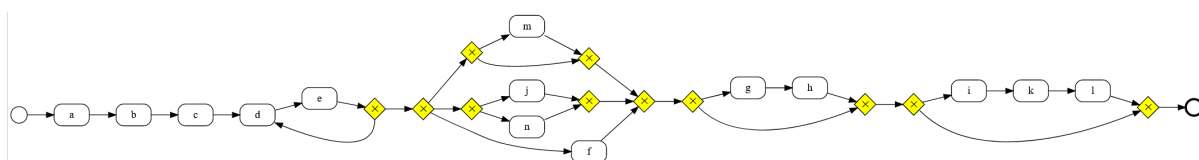


Rys. 4.16. Warianty procesu – przykład #Przypadki sepsy

Przy odkrywaniu modelu użyto następujących wag poszczególnych metryk: odwzorowanie = 8, złożoność = 2, generalizacja = 2, precyzja = 2, prostota = 2. Model dla tego dziennika zdarzeń znaleziony przy pomocy algorytmu to:

$$\{a\}\{b\}\{c\}lo0(\{d\}\{e\})xor(opt(\{m\})\{f\}xor(\{n\}\{j\}))opt(seq(\{g\}\{h\}))opt(seq(\{i\}\{k\}\{l\}))$$

Model został przedstawiony graficznie na rys. 4.17, natomiast przebieg ewolucji na rys. 4.18



Rys. 4.17. Znaleziony model – przykład #Przypadki sepsy

Do znalezienia modelu potrzebne było 60 generacji. Zajął to 4.03 godziny, używając 32 wątków procesora. Znaczne zwiększenie ilości wariantów i ich długości znacząco zwiększyło czas iteracji, natomiast wpłynęło negatywnie na otrzymany model. Pominięte zostały w nim dwie najrzadziej występujące

aktywności. W modelu widać jednak znaczną zgodność z najczęściej występującymi wariantami. Znalezienie lepszego rozwiązania byłoby możliwe, używając wydajniejszego komputera lub zwiększając czas ewolucji. Metryki mają następujące wartości:

Średnia ważona: 0.9094

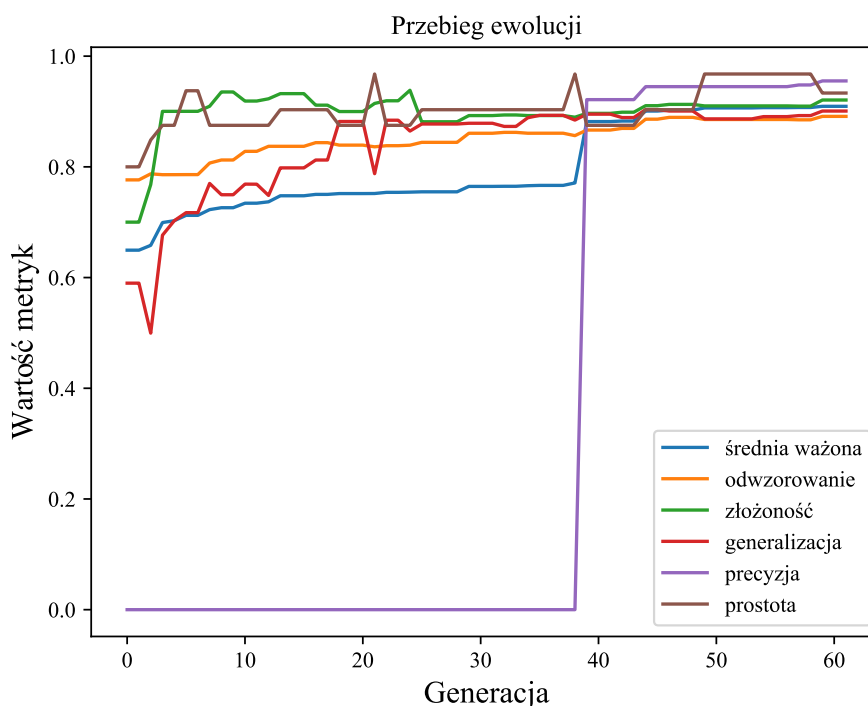
Odwzorowanie: 0.8912

Złożoność: 0.9207

Generalizacja: 0.9008

Precyzja: 0.9552

Prostota: 0.9333



Rys. 4.18. Przebieg ewolucji – przykład #Przypadki sepsy

4.2. Wyniki w zależności od przyjętych wag poszczególnych metryk

4.2.1. Brak poszczególnych metryk

Odwzorowanie jest kluczowe, gdyż jest jedyną metryką, która sprawdza zgodność modelu z dziennikiem zdarzeń i bez tej metryki model byłby pozbawiony wartości. Pozostałe metryki wpływają na jego jakość. Warto więc sprawdzić, jak ich brak wpłynąłby na odkryty model.

4.2.1.1. Poprawny model

Wpływ braku poszczególnych metryk został sprawdzony dla podzbioru dziennika zdarzeń z sekcji 4.1.2.5 uproszczonego poprzez wybranie tylko przypadków składających się z 5 zdarzeń. Składa się on z 7 unikalnych aktywności, 1254 przypadków, 8 wariantów, które mają po 5 zdarzeń. Na rysunku 4.19 znajdują się warianty procesu dla tego przykładu.

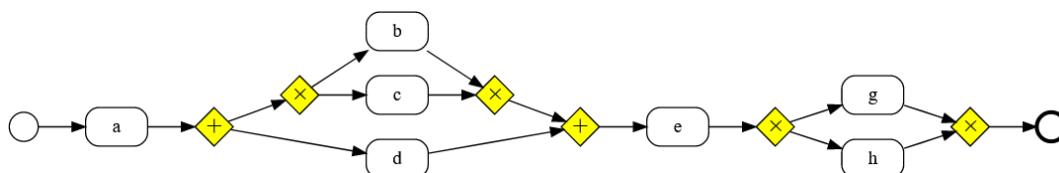
455, a, c, d, e, h
 191, a, b, d, e, g
 177, a, d, c, e, h
 144, a, b, d, e, h
 111, a, c, d, e, g
 82, a, d, c, e, g
 56, a, d, b, e, h
 38, a, d, b, e, g

Rys. 4.19. Warianty procesu dla przykładu, w którym sprawdzany jest wpływ braku poszczególnych metryk

W tej sekcji dla porównania przedstawiono poprawny model, odkryty używając wszystkich metryk. Przy jego odkrywaniu użyto następujących wag poszczególnych metryk: odwzorowanie = 8, złożoność = 2, generalizacja = 2, precyzja = 4, prostota = 2. Model dla tego dziennika zdarzeń znaleziony przy pomocy algorytmu to:

$$\{a\} \text{and} (\text{xor}(\{c\}\{b\})\{d\})\{e\} \text{xor}(\{h\}\{g\})$$

Model został przedstawiony graficznie na rys. 4.20.



Rys. 4.20. Poprawny znaleziony model dla przykładu, w którym sprawdzany jest wpływ braku poszczególnych metryk

Do znalezienia modelu potrzebne były 233 generacje, podczas których przeszukano 41537 unikalnych osobników. Zajęło to 8.4 minuty, używając 4 wątków procesora. Natomiast, metryki mają następujące wartości:

Średnia ważona: 0.9960

Odwzorowanie: 1.0

Złożoność: 1.0

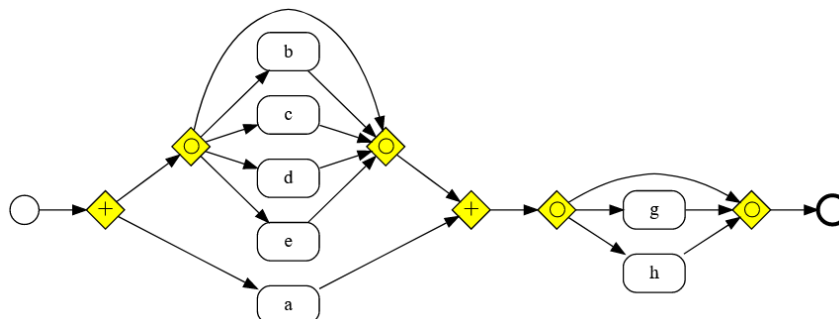
Generalizacja: 0.9641

Precyzja: 1.0

Prostota: 1.0

4.2.1.2. Brak precyzji

W kolejnym przykładzie użyto tych samych wag dla poszczególnych metryk, z wyjątkiem precyzji, dla której przyjęto wagę równą 0. Model został przedstawiony graficznie na rys. 4.21.



Rys. 4.21. Model znaleziony bez użycia precyzji

Cechą charakterystyczną modelu ze słabą precyzją będzie zastąpienie bramek *xor* bardziej skomplikowanymi bramkami *opt* i *and* oraz niewystarczające „dopasowanie się” do logu, przez co model pozwala na nie mające biznesowego sensu zachowania. Ze względu na mniejszą ilość bramek zazwyczaj potrzebną w modelu ze słabą precyzją oraz z uwagi na większą ilość przypadków uchwyczonych w takim modelu i większą szansę na opisanie także tych w logu, wygenerowanie takiego modelu jest łatwiejsze.

Poniżej zaprezentowano średnią ważoną, gdyby przyjąć wagi metryk z poprawnego modelu, a także wartości poszczególnych metryk.

Średnia ważona: 0.9289

Odwzorowanie: 1.0

Złożoność: 1.0

Generalizacja: 0.9641

Precyzja: 0.6982

Prostota: 1.0

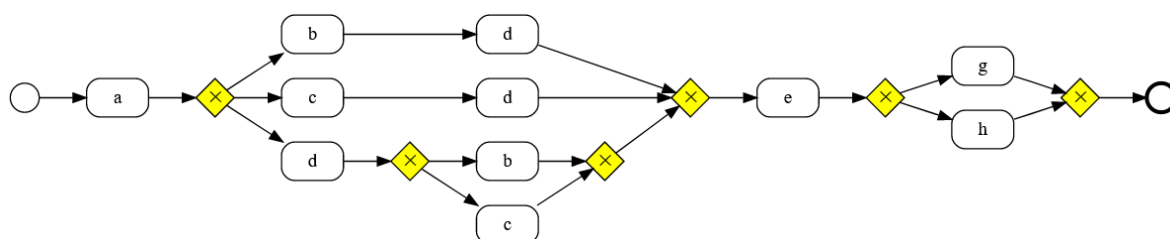
4.2.1.3. Brak generalizacji i prostoty

Brak użycia generalizacji miał niezauważalny wpływ na otrzymany model. Ciężko było znaleźć model, który miałby słabą generalizację. Powodem takiej sytuacji może być duża zależność generalizacji od prostoty, gdyż ciężko o niską wartość generalizacji bez obecności dodatkowych elementów w modelu, co obniży wartość jego prostoty. Z tego powodu w tym przykładzie dla wag obu tych metryk przyjęto wartość 0.

Mimo tego wciąż ciężkie okazało się znalezienie modelu ze słabą generalizacją i precyzją. Wynika to z natury ewolucji gramatycznej, gdyż bardziej prawdopodobne jest wygenerowanie prostszego ciągu znaków, dlatego częściej będziemy otrzymywać proste modele z dobrą generalizacją. Szansa na otrzymanie modelu z wysoką wartością funkcji dopasowania spada ze wzrostem ilości elementów. Niemniej, nadal jest możliwe otrzymanie takiego modelu, więc obecność tych metryk nie jest zupełnie pozbawiona znaczenia.

Cechą charakterystyczną modelu ze słabą generalizacją będzie zastąpienia bramek *and* i *opt* prostszymi bramkami *xor* oraz nadmierne „dopasowanie się” do jak największej ilości pojedynczych wariantów w logu, przez co nieuchwycone mogą zostać brakujące w logu zachowania.

Model został przedstawiony graficznie na rys. 4.22.



Rys. 4.22. Model znaleziony bez użycia generalizacji i prostoty

Poniżej zaprezentowano średnią ważoną, gdyby przyjąć wagi metryk z poprawnego modelu, a także wartości poszczególnych metryk. Co ciekawe, wartość generalizacji wciąż nie jest dużo niższa niż w poprawnym modelu.

Średnia ważona: 0.9697

Odwzorowanie: 1.0

Złożoność: 1.0

Generalizacja: 0.9498

Precyzja: 1.0

Prostota: 0.7778

4.2.2. Wpływ złożoności na wynik

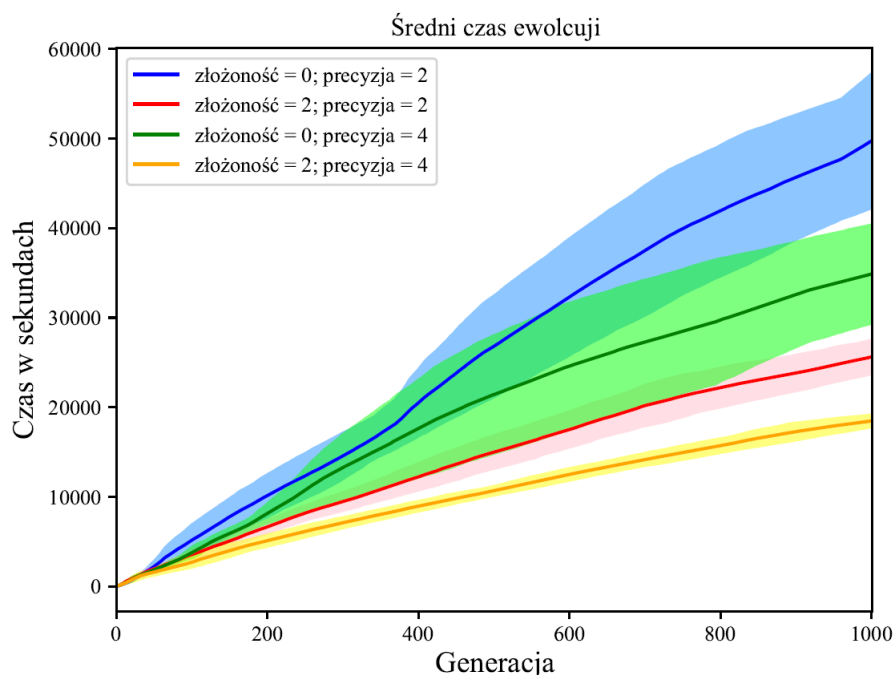
W sekcji 2.4.2 przewidywano dwa aspekty ewolucji, na które powinna wpłynąć złożoność. Były to czas trwania obliczania pojedynczej generacji oraz poprawa ogólnej jakości modelu poprzez unikanie lokalnych ekstremów na wczesnych etapach ewolucji.

Celem sprawdzenia tej hipotezy przetestowano algorytm dla przykładu z sekcji 4.1.2.5. Został on uruchomiony 35 razy ze złożonością równą 0 i 36 ze złożonością równą 2. Wszystkie modele ewoluowano przez 1000 pokoleń, gdyż po takiej ilości generacji jest małe prawdopodobieństwo, że nastąpią kolejne zmiany w modelu.

Ostatecznie użyto 4 różnych konfiguracji i oprócz złożoności zmieniano też wagę precyzji, z uwagi na to, że jest to najbliższa złożoności metryka i pozwoliło to na zbadanie, jak bardzo zależne od siebie są te dwie metryki. W sekcji 4.2.1.3 pokazano, że generalizacja i prostota mają mały wpływ na przebieg ewolucji, dlatego pozostały one stałe podczas tego eksperymentu.

Ostatecznie przy odkrywaniu modelu dla tego wariantu użyto następujących wag poszczególnych metryk: odwzorowanie = 8, generalizacja = 2, prostota = 2 i dwie zmienne precyzja równa 2 lub 4 oraz złożoność równa 0 lub 2.

Pierwszy aspektem, który zbadano, był czas trwania ewolucji. Na rysunku 4.23 przedstawiono, jak na czas ewolucji wpływają poszczególne konfiguracje parametrów.



Rys. 4.23. Średni czas i odchylenie standardowe trwania ewolucji

Czas dla złożoności = 0 i precyzja = 2: 50600 ± 6670 ,

Czas dla złożoności = 0 i precyzja = 4: 32800 ± 5960

Czas dla złożoności = 2 i precyzja = 2: 24700 ± 2300

Czas dla złożoności = 2 i precyzja = 4: 18300 ± 657

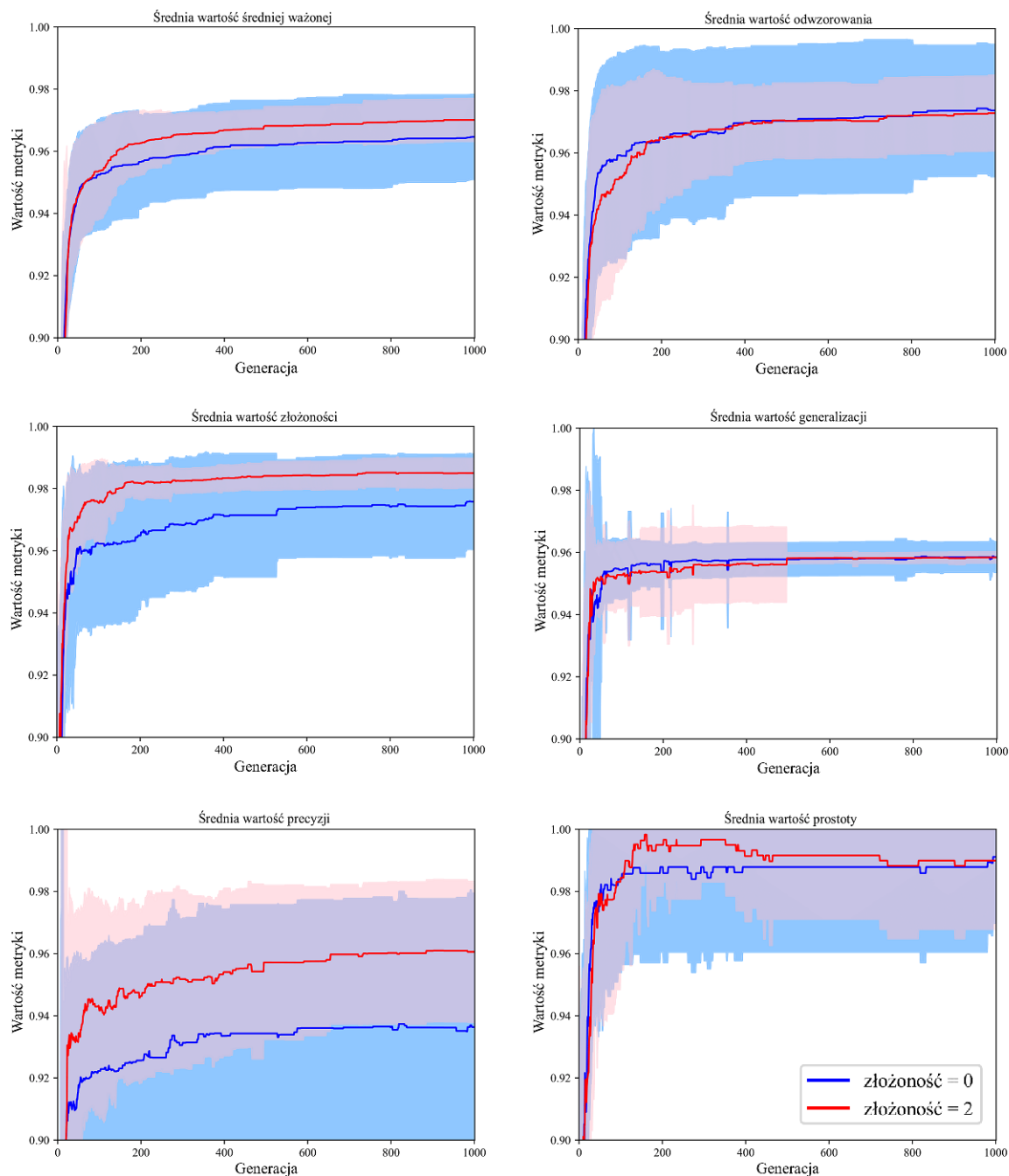
Można zauważyć, że zarówno złożoność, jak i precyzja wpływają na czas ewolucji, a im wyższa waga tych metryk, tym krócej trwa ewolucja. Metryki te mogą być używane razem, a używanie złożoności umożliwia dodatkowe przyspieszenie algorytmu.

Przeanalizowano również, jak zmieniały się wartości poszczególnych metryk i ich średnia ważona. Na rysunku 4.24 dla poprawy czytelności i lepszego zrozumienia rezultatów pominięto rolę precyzji i skupiono się jedynie na wpływie złożoności.

Warto zaznaczyć, że licząc średnią, nie wzięto pod uwagę złożoności, gdyż jest to parametr, który w założeniu ma służyć jedynie usprawnieniu procesu ewolucji, a nie służyć do oceny końcowego modelu. Użyto wagi precyzji równej 4.

Nie udało się stwierdzić zdecydowanego wpływu złożoności na jakość modelu, jednak można zauważyć kilka interesujących różnic, które zgadzają się z intuicją. Otrzymany model jest średnio lepszy, kiedy używa się złożoności, a wpływ na to ma głównie fakt, że jest on bardziej precyzyjny. Warto też zauważyć, że w takim przypadku we wczesnych generacjach model jest prostszy, ale ma gorsze odwzorowanie, a dopiero z czasem staje się bardziej skomplikowany. Niestety, złożoność zdaje się nie mieć wpływu na najważniejszą z metryk, czyli odwzorowanie, jak i na generalizację.

W obu przypadkach algorytm znalazł model tożsamy z modelem w przykładzie z sekcji 4.1.2.5, który wydaje się optymalny. Najlepsze modele otrzymywano używając konfiguracji bez złożoności.



Rys. 4.24. Średnie wartości i odchylenia standardowe średniej ważonej metryk, odwzorowania, złożoności, generalizacji, prostoty, precyzji

	Złożoność	Brak złożoności
Średnia ważona:	0.970 ± 0.0069	0.965 ± 0.0136
Odwzorowanie:	0.973 ± 0.0122	0.974 ± 0.0211
Złożoność:	0.985 ± 0.0048	0.976 ± 0.0152
Generalizacja:	0.958 ± 0.0018	0.959 ± 0.0049
Precyzja:	0.961 ± 0.0224	0.935 ± 0.0432
Prostota:	0.989 ± 0.0233	0.991 ± 0.0212

4.3. Wnioski dotyczące ewolucji

Analizując czasy ewolucji dla przedstawionych przykładów, można zauważyć, że na wzrost czasu oraz ilości generacji potrzebnych do znalezienia rozwiązania, wpływa bardziej ilość unikalnych aktywności oraz długość zdarzeń w pojedynczym wariancie niż ich ilość. Algorytmy ewolucyjne źle radzą sobie z dużą ilością zmiennych. Można częściowo jednak rozwiązać ten problem poprzez zrównoleglenie obliczeń, na którą w łatwy sposób pozwalają.

Dobór wag metryk oddziałuje nie tylko na końcowy model, ale także na każdy model – osobnika w populacji na każdym etapie ewolucji, dlatego wagi metryk powinny być dobierane nie tylko pod kątem końcowego rozwiązania, ale także z myślą o tym, czy wybrany zbiór wag pozwoli na otrzymanie tego rezultatu. Na wykresach pokazujących przebieg ewolucji można zaobserwować wiele przykładów, gdzie aby model mógł stać się ogólnie lepszy, konieczne było tymczasowe pogorszenie się, którejs z metryk. Z tego powodu ważne jest, żeby zachować balans pomiędzy nimi i nawet jeśli pożądane jest zoptymalizowanie końcowego modelu pod kątem konkretnej metryki, należy ustalić ją tak, żeby nie zdominowała całego procesu ewolucji.

Po eksperymentach można dojść do wniosku, że wybierając metryki, należy kierować się ich wpływem na proces ewolucji, a nie końcowy rezultat. Powszechnie używane w eksploracji procesów metryki, jak generalizacja nie miały w tym przypadku dużego wpływu na działanie algorytmu ewolucyjnego, podczas gdy nowe, nieznane metryki, jak złożoność, niemające zastosowania do oceny końcowego modelu, mogą mieć korzystny wpływ na przebieg ewolucji poprzez ukierunkowanie doboru pośrednich osobników we właściwą stronę, co pozwoli znaleźć lepszy finalny model.

5. Podsumowanie

W pracy przygotowano program, przy którego użyciu możliwe jest odkrywanie najlepszych lub będących blisko optymalnych modeli procesów biznesowych [47]. Może on być łatwo konfigurowany i pozwala na swobodnie eksperymentowanie podczas znajdowania modeli celem znalezienia tego, który najlepiej spełnia założenia i potrzeby. Przy jego pomocy możliwe jest znalezienie dowolnie dobrego modelu i nie dotyczą go ograniczenia znane z klasycznych algorytmów. Ze sposobu działania algorytmów ewolucyjnych wynika jednak problem – program nie daje gwarancji znalezienia najlepszego rozwiązania, a w niektórych przypadkach konieczne było kilkanaście jego uruchomień, żeby znaleźć rozwiązanie z wartością odwzorowania równą jeden, czyli bezbłędnie opisujące wszystkie warianty procesu. Uruchomienie kilku instancji programu równolegle nie jest jednak problemem, dlatego sugerowany jest podział dostępnej mocy obliczeniowej i wybór najlepszego modelu spośród otrzymanych rozwiązań.

Główną przeszkodą podczas tworzenia programu było połączenie wiedzy z dziedzin eksploracji procesów, algorytmów ewolucyjnych oraz lingwistyki we wspólną całość. Zmiany dotyczące jednego zagadnienia nie mogą być traktowane oddzielnie, a wprowadzając do programu rozwiązania dotyczące jednej z jego części konieczna była modyfikacja innych, w celu ich jak najlepszego współdziałania.

Przykładem przenikania się wiedzy z dwóch dziedzin jest dodatkowa zaproponowana metryka – złożoność. Jej użycie daje obiecujące rezultaty i wpływa korzystnie na czas potrzebny do znalezienia rozwiązania. Należy zaznaczyć, że jej użycie powinno przede wszystkim sygnalizować możliwość ewoluowania modelu uwzględniając fakt, że odkrywanie procesu odbywa się właśnie metodą ewolucyjną. Sama metryka jest tylko jedną z możliwych propozycji, ale patrząc szerzej, można zaproponować inne sposoby jej obliczania, jak i inne techniki poprawiające działanie podobnych programów poprzez metody wychodzące poza dziedzinę eksploracji procesów. Pokazano też, że niektóre metryki, takie jak generalizacja i prostota nie są konieczne podczas ewolucji.

Podczas implementacji duży nacisk został położony na minimalizację czasu obliczeń. Algorytmy ewolucyjne rozwiązują problem metodą prób i błędów bazując na przeszukaniu jak największej ilości możliwych rozwiązań, dlatego tak ważne jest, żeby pojedyncza iteracja zajmowała jak najmniej czasu. Konieczne było więc nie tylko stworzenie działającego programu, ale też ciągłe wprowadzenie poprawek zmniejszających czas potrzebny na znalezienie modelu.

W algorytmach ewolucyjnych najważniejsza jest eksploracja, więc czerpiąc z tego podejścia i z obserwacji podczas tworzenia pracy tematyka w niej poruszana może być rozwijana, w szczególności poprzez wprowadzenie bardziej zaawansowanych metod ewolucji oraz eksperymentowanie z innymi gramatykami opisującymi procesy biznesowe.

Algorytmy ewolucyjne są ciekawym sposobem rozwiązywania problemów, bo mniejsze znaczenie ma przy nich ekspercka wiedza z danej dziedziny, a ważniejsze jest zoptymalizowanie sposobu rozwiązania problemu pod kątem takich algorytmów, do czego może być konieczne wyjście poza przyjęte w danej dziedzinie schematy. Rozwiązania znalezione przez takie algorytmy mogą być innowacyjne, a same algorytmy świetnie sprawdzają się tam, gdzie nie ma powszechnie przyjętych, klasycznych metod.

Zarówno dla eksploracji procesów, jak i algorytmów ewolucyjnych istnieje pole do rozwoju, gdyż głównym problemem, z którym obecnie się borykają, są możliwości obliczeniowe współczesnych komputerów. Wraz z ich wzrostem w przyszłości można się więc spodziewać rozkwitu obu dziedzin.

Bibliografia

- [1] Thomas H Davenport. „*Process innovation: reengineering work through information technology*”. Harvard Business Press, 1993.
- [2] Michael Hammer i James Champy. „*Reengineering the corporation: A manifesto for business revolution*”. W: *Business Horizons* 36.5 (1993), s. 90–91. ISSN: 0007-6813. DOI: [https://doi.org/10.1016/S0007-6813\(05\)80064-3](https://doi.org/10.1016/S0007-6813(05)80064-3).
- [3] Ivar Jacobson, Maria Ericsson i Agneta Jacobson. „*The Object Advantage: Business Process Re-engineering with Object Technology*”. USA: ACM Press/Addison-Wesley Publishing Co., 1994. ISBN: 0201422891.
- [4] Hans-Erik Eriksson i Magnus Penker. „*Business Modeling With UML: Business Patterns at Work*”. 1st. USA: John Wiley i Sons, Inc., 1998. ISBN: 0471295515.
- [5] Geary A. Rummler i Alan P. Brache. „*Improving performance: how to manage the white space on the organization chart*”. Jossey-Bass, 1995.
- [6] Wil M. P. van der Aalst. „*Business process management demystified: A tutorial on models, systems and standards for workflow management*”. W: *Lectures on Concurrency and Petri Nets*. Red. J. Desel, W. Reisig i G. Rozenberg. Springer, 2003, s. 1–65.
- [7] Nathaniel Palmer. „<https://bpm.com/what-is-bpm>”. dostęp: 12 lutego 2020.
- [8] Wil M. P. van der Aalst. „*Business process management: a comprehensive survey. ISRN Softw. Eng. 1-37*”. W: *ISRN Software Engineering* (sty. 2012). DOI: [10.1155/2013/507984](https://doi.org/10.1155/2013/507984).
- [9] M. Dumas i in. „*Fundamentals of Business Process Management*”. Springer Berlin Heidelberg, 2013. ISBN: 9783642331435.
- [10] Jan Recker i in. „*Business Process Modeling- A Comparative Analysis*”. W: *Journal of the Association of Information Systems* 10 (kw. 2009). DOI: [10.17705/1jais.00193](https://doi.org/10.17705/1jais.00193).
- [11] OMG. „*Business Process Model and Notation (BPMN), Version 2.0*”. Object Management Group, 2011.
- [12] Jan Mendling, H.A. Reijers i Wil M. P. van der Aalst. „*Seven Process Modeling Guidelines (7PMG)*”. W: *Information and Software Technology* 52 (lut. 2010), s. 127–136. DOI: [10.1016/j.infsof.2009.08.004](https://doi.org/10.1016/j.infsof.2009.08.004).
- [13] Marc Kerremans. „*Gartner Market Guide for Process Mining*”. kw. 2018.

- [14] Wil M. P. van der Aalst i in. „*Process Mining Manifesto*”. W: *Lecture Notes in Business Information Processing* 99 (sierp. 2011), s. 169–194. DOI: 10.1007/978-3-642-28108-2_19.
- [15] Wil M. P. van der Aalst. „*Process Mining: Overview and Opportunities*”. W: *ACM Transactions on Management Information Systems* 3 (lip. 2012), s. 7.1–7.17. DOI: 10.1145/2229156.2229157.
- [16] Wil M. P. van der Aalst. „*Process Mining - Data Science in Action, Second Edition*”. Springer, 2016, s. 163–240. ISBN: 978-3-662-49850-7. DOI: 10.1007/978-3-662-49851-4.
- [17] Wil M. P. van der Aalst, A. Weijters i Laura Mărușter. „*Workflow Mining: Discovering Process Models from Event Logs*”. W: *Knowledge and Data Engineering, IEEE Transactions on* 16 (paź. 2004), s. 1128–1142. DOI: 10.1109/TKDE.2004.47.
- [18] Jan Martijn Van der Werf i in. „*Process Discovery Using Integer Linear Programming*”. W: *Fundamenta Informaticae* 94 (czer. 2008), s. 368–387. DOI: 10.1007/978-3-540-68746-7_24.
- [19] A. Weijters, Wil M. P. van der Aalst i Alves Medeiros. „*Process Mining with the Heuristics Miner-algorithm*”. T. 166. Sty. 2006.
- [20] B Dongen i Wil M. P. van der Aalst. „*Multi-phase process mining: Aggregating instance graphs into EPCs and Petri nets*”. W: *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management* (sty. 2005).
- [21] Sander J. J. Leemans, Dirk Fahland i Wil M. P. van der Aalst. „*Discovering Block-Structured Process Models from Event Logs - A Constructive Approach*”. W: *Application and Theory of Petri Nets and Concurrency*. Red. José-Manuel Colom i Jörg Desel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, s. 311–329. ISBN: 978-3-642-38697-8.
- [22] Wil M. P. van der Aalst. „*Relating Process Models and Event Logs - 21 Conformance Propositions*”. W: *Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data 2018 Satellite event of the conferences: 39th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2018 and 18th International Conference on Application of Concurrency to System Design ACSD 2018, Bratislava, Slovakia, June 25, 2018*. Red. Wil M. P. van der Aalst, Robin Bergenthum i Josep Carmona. T. 2115. CEUR Workshop Proceedings. CEUR-WS.org, 2018, s. 56–74.
- [23] F. Blum. „*Metrics in process discovery*”. Spraw. tech. TR/DCC-2015-6. 2015.
- [24] B. F. van Dongen, J. Mendling i W. M. P. van der Aalst. „*Structural Patterns for Soundness of Business Process Models*”. W: *2006 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)*. 2006, s. 116–128. DOI: 10.1109/EDOC.2006.56.
- [25] Ahmed Awad i Frank Puhlmann. „*Structural detection of deadlocks in business process models*”. W: *International Conference on Business Information Systems* (grud. 2008), s. 239–250. DOI: 10.1007/978-3-540-79396-0_21.
- [26] P. A. Vikhar. „*Evolutionary algorithms: A critical review and its future prospects*”. W: *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*. 2016, s. 261–265. DOI: 10.1109/ICGTSPICC.2016.7955308.

- [27] Emad Mabrouk, Abdel-Rahman Hedar i Masao Fukushima. „*Memetic Programming Algorithm with Automatically Defined Functions*”. Grud. 2010.
- [28] D. Noever i Subbiah Baskaran. „*Steady-state vs. generational genetic algorithms: A comparison of time complexity and convergence properties*”. Spraw. tech. 1992-07-032. Lip. 1992.
- [29] John R. Koza. „*Non-Linear Genetic Algorithms for Solving Problems*”. United States Patent 4935877. filed may 20, 1988, issued june 19, 1990, 4,935,877. Australian patent 611,350 issued september 21, 1991. Canadian patent 1,311,561 issued december 15, 1992. 1990.
- [30] John R. Koza. „*Genetic Programming: On the Programming of Computers by Means of Natural Selection*”. Cambridge, MA, USA: MIT Press, 1992. ISBN: 0262111705.
- [31] Conor Ryan, Jj Collins i Michael O Neill. „*Grammatical evolution: Evolving programs for an arbitrary language*”. W: *Lecture Notes in Computer Science Genetic Programming* (1998), 83–96. DOI: 10.1007/bfb0055930.
- [32] J. Backus. „*The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference*”. W: *IFIP Congress*. 1959.
- [33] Peter Naur. „*A Course of Algol 60 Programming*”. W: *ALGOL Bull.* Sup 9 (sty. 1961), 1–38. ISSN: 0084-6198.
- [34] Donald E. Knuth. „*Backus Normal Form vs. Backus Naur Form*”. W: *Communications of the ACM* 7.12 (1964), s. 735–736. ISSN: 0001-0782. DOI: <http://doi.acm.org/10.1145/355588.365140>.
- [35] Miguel Nicolau i Alexandros Agapitos. „*Understanding grammatical evolution: Grammar design*”. W: *Handbook of Grammatical Evolution* (sty. 2018), s. 23–53. DOI: 10.1007/978-3-319-78717-6_2.
- [36] Michael Lynn Cramer. „*A representation for the Adaptive Generation of Simple Sequential Programs*”. W: *Proceedings of an International Conference on Genetic Algorithms and the Applications*. Red. John J. Grefenstette. Carnegie-Mellon University Pittsburgh PA USA, 1985, s. 183–187.
- [37] David Beasley, David R. Bull i Ralph R. Martin. „*An Overview of Genetic Algorithms: Part 1, Fundamentals*”. W: *University Computing* 15.2 (1993), s. 58–69.
- [38] J. C. A. M. Buijs, B. F. van Dongen i W. M. P. van der Aalst. „*Quality Dimensions in Process Discovery: The Importance of Fitness, Precision, Generalization and Simplicity*”. W: *International Journal of Cooperative Information Systems* 23.01 (2014), s. 1440001. DOI: 10.1142/S0218843014400012. eprint: <https://doi.org/10.1142/S0218843014400012>.
- [39] Wil M. P. van der Aalst, Arya Adriansyah i Boudewijn van Dongen. „*Replaying history on process models for conformance checking and performance analysis*”. W: *WIREs Data Mining and Knowledge Discovery* 2.2 (2012), s. 182–192. DOI: <https://doi.org/10.1002/widm.1045>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1045>.

- [40] J. Munoz-Gama i J. Carmona. „*Enhancing precision in Process Conformance: Stability, confidence and severity*”. W: *2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*. 2011, s. 184–191. DOI: 10.1109/CIDM.2011.5949451.
- [41] Michael Fenton i in. „*PonyGE2*”. W: *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (2017). DOI: 10.1145/3067695.3082469.
- [42] Michael zur Muehlen i Jan Recker. „*How Much Language Is Enough? Theoretical and Practical Use of the Business Process Modeling Notation*”. W: *Advanced Information Systems Engineering*. Red. Zohra Bellahsene i Michel Léonard. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, s. 465–479. ISBN: 978-3-540-69534-9.
- [43] Saul B. Needleman i Christian D. Wunsch. „*A general method applicable to the search for similarities in the amino acid sequence of two proteins*”. English (US). W: *Journal of Molecular Biology* 48.3 (mar. 1970), s. 443–453. ISSN: 0022-2836. DOI: 10.1016/0022-2836(70)90057-4.
- [44] „<http://experiments.mostafa.io/public/needleman-wunsch/>”. dostęp: 20 marca 2020.
- [45] „<https://github.com/PonyGE/PonyGE2/wiki>”. dostęp: 15 lutego 2020.
- [46] Felix Mannhardt. „*Sepsis Cases - Event Log*”. 2016. DOI: 10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460.
- [47] „<https://github.com/psm8/automated-business-process-discovery>”. dostęp: 25 kwietnia 2020.