

POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Informatyka

SPECJALNOŚĆ: Inżynieria Systemów Informatycznych

**PRACA DYPLOMOWA
MAGISTERSKA**

Nowe mechanizmy uczenia w modelu sGCS

New learning algorithms in sGCS model

autor : inż. Michał Stempkowski

Opiekun pracy:
dr hab. inż. Olgierd Unold,
prof. nadzw. PWr

OCENA PRACY:

Wstęp.....	4
Cel pracy.....	5
Pojęcia podstawowe.....	6
Algorytm CYK	13
GCS	16
Główna pętla algorytmu.....	16
Indukcja gramatyki	16
Parsowanie zdań	17
Ścisk	18
Operatory pokrycia	18
Operator pokrycia terminalnego.....	18
Operator pokrycia uniwersalnego.....	19
Operator pokrycia startowego	19
Operator pokrycia agresywnego.....	19
Operator pokrycia pełnego.....	19
Algorytm genetyczny	20
Metody selekcji.....	20
Inwersja	20
Mutacja	21
Krzyżowanie	21
Korekcja gramatyki.....	21
Usuwanie produkcji redundantnych	21
Usuwanie produkcji nieproduktywnych	22
Usuwanie produkcji nieosiągalnych	22
sGCS.....	23
Algorytm CYK+	23
Traceback.....	24
Normalizacja.....	24
Przystosowanie w sGCS	25
Estymacja prawdopodobieństw	25
neg-sGCS.....	25
Implementacja.....	26
Użytkowanie aplikacji.....	26
Wykorzystanie gui_manager.py	27
Wykorzystanie console_fetcher.py.....	35
Artefakty	36
Biblioteka.....	40
Warstwowa budowa biblioteki.....	41
Moduły pomocnicze.....	42
GUI	43
Warstwa wykonawców	69
Właściwy algorytm	73
Właściwy algorytm – część indukcyjna	89
Warstwa rdzenia.....	101
Warstwa danych.....	104
Parametry algorytmu.....	106
Wyniki badań.....	107
Standardowa losowa populacja	107
GCS	107
sGCS.....	112
neg-sGCS.....	117

Reguły charakterystyczne	123
GCS	124
sGCS	127
neg-sGCS.....	130
Podsumowanie.....	134
Literatura	135
Spis ilustracji	136
Spis definicji	Błąd! Nie zdefiniowano zakładki.
Załączniki	140

Wstęp

Od zarania ludzkości gromadzenie i przetwarzanie informacji jest ważnym zadaniem. Z kolejnymi wiekami człowiek gromadził tych informacji coraz więcej i wykorzystuje coraz to bardziej wymyślne i automatyczne metody analizy tych informacji. Wraz z rozwojem komputerów stały się one istotną częścią mechanizmu poznawczego człowieka.

Jednym z ważniejszych problemów, jakie napotykamy przy próbie wykorzystania komputerów do wspomagania zdobywania wiedzy, jest ich sposób rozumowania i pozyskiwania wiedzy. Człowiek analizując problemy myśli w sposób wysoce abstrakcyjny, zauważa zależności, formułuje reguły pomiędzy obiektami, tworzy sieć połączeń „jeżeli A, to B”. Komputery posiadają coraz większą moc obliczeniową, jednakże nadal są dosyć ograniczone w kwestii samodzielnego zdobywania wiedzy.

Uczenie maszynowe (ang. Machine Learning) to jedna z gałęzi informatyki, która skupia się na rozszerzeniu umiejętności komputerów w rozpoznawaniu wzorców oraz samodzielnym uczeniu się. Zahacza one o inne dziedziny nauki, takie jak psychologia, czy statystyka. Obejmuje ono problematykę pisania programów, które potrafią pozyskiwać nową wiedzę i poprawiać istniejącą na podstawie zyskiwanych informacji.

Nie da się zanegować faktu, że uczenie maszyn jest zagadnieniem ważnym i ma wiele zastosowań. Umiejętność napisania aplikacji zdolnych do adaptowania się, znajdowania wzorców i symulujących proces ludzkiego wnioskowania jest ważna, zaś aplikacje takie znajdują wiele zastosowań we współczesnym świecie:

- Przetwarzanie języka naturalnego;
- Rozpoznawanie mowy – tworzenie aplikacji komputerowych wykorzystujących mowę ludzką jako interfejs sterowania;
- Systemy ekspertowe – systemy wspomagające proces podejmowania decyzji, wspomagające proces analizy złożonych problemów ekonomicznych, medycznych i nie tylko;
- Data mining – analiza i odnajdowanie zależności w dużych strukturach danych;
- Bioinformatyka – rozpoznawanie powtarzających się struktur łańcuchów RNA, czy DNA, rozpoznawanie oraz analiza zachowania chorób;
- Rozpoznawanie obrazów – wyszukiwanie wzorców z obrazów. Zamiana analogowej reprezentacji na cyfrową, odnajdywanie abstrakcyjnych kształtów;
- Robotyka – tworzenie automatów zdolnych do zdobywania wiedzy na temat środowiska, w którym zostały umieszczone, zdolność do autonomicznego funkcjonowania i wykonywania czynności;
- Wnioskowanie logiczne.

W poniższej pracy skupimy się na jednym z mechanizmów uczenia maszynowego, a konkretnie na systemie GCS (Grammar Classifier System). Jest to system uczenia nadzorowanego, skupiający się na problemie klasyfikacji. Został wprowadzony w publikacji „Ewolucyjne wnioskowanie gramatyczne” przez Unolda [1]. Wynikiem działania algorytmu jest zbiór reguł gramatyki, czyli zestaw reguł opisujący dane zjawisko.

Cel pracy

Celem poniższej pracy jest zbadanie konkretnych aspektów funkcjonowania algorytmów GCS oraz sGCS [2], ich wpływu na cały proces uczenia oraz zaproponowanie nowych mechanizmów systemu sGCS. Skupi się ona przede wszystkim na zbadaniu wpływu reguł początkowych na proces uczenia. Zbada również wpływ zdań negatywnych na proces uczenia sGCS i zaproponuje modyfikacje niezbędne do uruchomienia algorytmu sGCS w trybie uczenia z wykorzystaniem zdań negatywnych. Badania te powinny dać pole do rozwoju nowych mechanizmów uczenia systemu sGCS.

Rozdział „Pojęcia podstawowe” dostarczy niezbędną teoretyczną widzę na temat gramatyk oraz funkcjonowania samego algorytmu.

Sam algorytm ze wszystkimi wariantami zostanie omówiony w rozdziałach „Algorytm CYK”, „GCS” oraz „sGCS”.

Rozdział „neg-sGCS” jest poświęcony modyfikacjom, jakim należy poddać algorytm sGCS, aby uczył się też z wykorzystaniem zdań negatywnych.

Kolejny rozdział („Implementacja”) omawia szczegółowo budowę oraz użytkowanie dostarczonej wraz z pracą biblioteki sgcs.

W rozdziale „Wyniki badań” znajdują się wyniki badań wpływu zdań negatywnych oraz reguł charakterystycznych na proces uczenia.

Ostatni rozdział stanowi podsumowanie pracy.

Pojęcia podstawowe

Poniżej podano definicje pojęć, które występują w tej pracy. Źródła te zebrano z wielu źródeł oraz poprzednich prac [1] [2] [3] [4] [5] [6].

Definicja (Relacja)

Relacją na zbiorze nazywamy dowolny podzbiór iloczynu kartezjańskiego $X \times X$. Jeżeli $R \subseteq X \times X$ jest relacją na zbiorze Z , to zależność $(x, y) \in R$ możemy zapisać w postaci $x R y$.

Przykład

Przykładem relacji jest relacja mniejszości \leq na zbiorze liczb naturalnych.

Definicja (Zwrotność)

Relację R na zbiorze X nazywamy zwrotnią, jeżeli $\forall x \in X x R x$.

Przykład

Relacja $\leq \in \mathbb{N}^2$ jest zwrotna, gdyż $\forall x \in \mathbb{N} x \leq x$.

Definicja (Symetryczność)

Relację R na zbiorze X nazywamy przechodnią, jeżeli $\forall x, y \in X x R y \Rightarrow y R x$.

Definicja (Przechodniość)

Relację R na zbiorze X nazywamy przechodnią, jeżeli $\forall x, y, z \in X x R y \wedge y R z \Rightarrow x R z$.

Przykład

Relacja $\leq \in \mathbb{N}^2$ jest przechodnia, gdyż $\forall x, y, z \in \mathbb{N} x \leq y \wedge y \leq z \Rightarrow x \leq z$.

Definicja (Słaba antysymetryczność)

Relację R na zbiorze X nazywamy słabo antysymetryczną, jeżeli $\forall x, y \in X x R y \wedge y R x \Rightarrow x = y$.

Przykład

Relacja $\leq \in \mathbb{N}^2$ jest słabo antysymetryczna, gdyż $\forall x, y \in \mathbb{N} x \leq y \wedge y \leq x \Rightarrow x = y$.

Definicja (Częściowy porządek)

Częściowym porządkiem w zbiorze X nazywamy taką relację \leq (gdzie $\leq \subseteq X \times X$), która jest zwrotnia, przechodnia i słabo antysymetryczna.

Przykład

Relacja $\leq \in \mathbb{N}^2$ jest częściowym porządkiem w zbiorze \mathbb{N} .

Definicja (Porządek liniowy)

Porządek \leq jest liniowy w zbiorze X , jeżeli $\forall x, y \in X x \leq y \vee y \leq x$.

Przykład

Porządek $\leq \in \mathbb{N}^2$ jest liniowy, każde 2 elementy $x, y \in \mathbb{N}$ są porównywalne.

Definicja (Zbiór częściowo uporządkowany)

Zbiór częściowo uporządkowany to zbiór X z relacją częściowego porządku \leq . Zbiór częściowo uporządkowany oznaczamy $\langle X, \leq \rangle$.

Definicja (Ograniczenie górne)

Niech $\langle P, \leq \rangle$ będzie częściowym porządkiem i niech $X \subseteq P$. Element $a \in P$ nazywamy ograniczeniem górnym zbioru X , jeżeli $\forall x \in X x \leq a$.

Działalnie definiujemy ograniczenie dolne.

Przykład

Dla częściowego porządku $\langle [0, 1] \subseteq \mathbb{R}, \leq \rangle$ 0 jest ograniczeniem dolnym a 1 ograniczeniem górnym.

Definicja (Kres góry)

Kresem górnym zbioru X nazywamy najmniejszy element zbioru $\{a \mid a \text{ jest ograniczeniem górnym zbioru } X\}$. Kres góry oznaczamy $\sup X$. Dualnie definiujemy kres góry (i oznaczamy go $\inf X$).

Przykład

Dla częściowego porządku $\langle [0, 1] \subseteq \mathbb{R}, \leq \rangle$ 0 jest kresem dolnym i nie istnieje kres góry (zawsze jest jakaś inna liczba bliższa jedynki i ją nie będącą).

Definicja (Zbiór skierowany)

Niech $\langle X, \leq \rangle$ będzie zbiorem częściowo uporządkowanym. Wtedy zbiór $\emptyset \neq X \subseteq P$ jest zbiorem skierowanym, jeżeli $\forall x, y \in X \exists z \in X (x \leq z \wedge y \leq z)$.

Definicja (Porządek zupełny)

Zbiór $\langle P, \leq \rangle$ jest porządkiem zupełnym, jeżeli P ma element najmniejszy oraz każdy skierowany podzbiór X zbioru P ma kres góry.

Definicja (Porządek regularny)

Zbiór $\langle P, \leq \rangle$ jest porządkiem regularnym, jeżeli w każdym niepustym zbiorze $\forall X \subseteq P (X \neq \emptyset \Rightarrow \exists m \in X \forall x \in X x \leq m)$.

Definicja (Dobry porządek)

Porządek nazywamy dobrym, jeżeli jest liniowy i regularny.

Definicja (Alfabet)

Alfabetem nazywamy skończony, niepusty zbiór.

Definicja (Symbol)

Elementy należące do alfabetu nazywamy symbolami.

Definicja (Słowo, słowo nad alfabetem)

Słowem (wyrazem) W nad alfabetem V nazywamy każdy skończony ciąg symboli taki, że $\forall x \in W x \in V$.

Definicja (Słowo puste)

Symbolem ε oznaczamy słowo puste, czyli słowo niezawierające żadnego symbolu.

Definicja (Długość słowa)

Długość słowa w oznaczamy, jako $|w|$. Nazywamy w ten sposób liczbę symboli składających się na dane słowo.

Definicja (Język, język nad alfabetem)

Językiem L nad alfabetem A nazywamy dowolny zbiór słów nad alfabetem A . Będziemy oznaczać go, jako $L(A)$.

Definicja (Język nad dowolnym alfabetem)

Zbiór pusty jest językiem nad dowolnym alfabetem.

Definicja (Gramatyka formalna)

Gramatyką formalną nazywamy czwórkę $G = (N, T, P, S)$, w której:

N – skończony zbiór symboli pomocniczych (zwanych nieterminalnymi).

T – skończony zbiór symboli końcowych (zwanych terminalnymi).

P – relacja skończona zwana zbiorem produkcji (wzór).

S – wybrany symbol pomocniczy, zwany symbolem początkowym gramatyki.

Definicja (Bezpośrednia wyprowadzalność)

Mając gramatykę $G = (N, T, P, S)$, jeżeli $(A, B) \in P$, wówczas pisząc $A \rightarrow B$ stwierdzamy, że B można wyprowadzić z A .

Definicja (Wyprowadzalność)

Wyprowadzalność możemy zdefiniować, jako relację bezpośredniej wyprowadzalności z dodatkową właściwością przechodności.

Wyprowadzalność B z A oznaczamy, jako $A \xrightarrow{*} B$. Czyli $A \xrightarrow{*} B$, gdy istnieje skończony ciąg wyprowadzeń pośrednich taki, że $A \rightarrow C_0 \rightarrow C_1 \rightarrow \dots \rightarrow C_n \rightarrow B$.

Definicja (Osiągalność)

Mając gramatykę $G = (N, T, P, S)$, A nazywamy osiągalnym, jeżeli $S \xrightarrow{*} A$.

Definicja (Produktywność)

Mając gramatykę $G = (N, T, P, S)$, A nazywamy produktywnym, jeżeli $\exists B \in T^* A \xrightarrow{*} B$.

Definicja (Język generowany przez gramatykę)

Język $L(G)$ generowany przez gramatykę G jest ciągiem symboli takim, że $L(G) \equiv \{x \mid x \in T^* \wedge x \text{ jest osiągalny}\}$.

Definicja (Równoważność gramatyk)

Dwie gramatyki są równoważne, jeżeli generują taki sam język, czyli $L(G1) = L(G2)$.

Nakładając dodatkowe ograniczenia na gramatykę możemy zdefiniować 4 typy gramatyki. Typy te nazywamy hierarchią Chomsky'ego.

Definicja (Gramatyka kombinatoryczna)

$G0$ nazywamy gramatyką kombinatoryczną. Nie posiada ona żadnych ograniczeń. Języki generowane przez gramatykę nazываемy rekurencyjnie przeliczalnymi.

Definicja (Gramatyka kontekstowa)

Gramatyka $G1 = (N, T, P, S)$ to taka gramatyka, że jej reguły spełniają następującą zależność: $\forall r \in P \ p \equiv \alpha A \beta \rightarrow \alpha \gamma \beta$, gdzie $A \in T \wedge \alpha, \beta \in \{x \mid x \in (T \cup N)^*\vee x = \varepsilon\} \wedge \gamma \in (T \cup N)^*$. Nazywamy ją gramatyką typu 1 (kontekstową). Język $L(G1)$ nazywamy językiem kontekstowym.

Definicja (Gramatyka bezkontekstowa)

Gramatyką $G2 = (N, T, P, S)$ to taka gramatyka, że jej reguły spełniają zależność: $\forall r \in P \ p \equiv A \rightarrow \Gamma$, gdzie $A \in N \wedge \Gamma \in (T \cup N)^*$. $A \Rightarrow \text{Tau}$.

Definicja (Gramatyka lewostronne liniowa)

Gramatykę $G2 = (N, T, P, S)$ nazywamy lewostronne liniową, jeżeli każda jej produkcja jest postaci: $A \rightarrow a B$, gdzie $a \in T \wedge B \in N^* \cup \{\varepsilon\}$.

Definicja (Gramatyka prawostronnie liniowa)

Gramatykę $G2 = (N, T, P, S)$ nazywamy prawostronnie liniową, jeżeli każda jej produkcja jest postaci: $A \rightarrow B a$, gdzie $a \in T \wedge B \in N^* \cup \{\varepsilon\}$.

Definicja (Gramatyka regularna)

Gramatyki $G3$ prawostronnie liniowe i lewostronne liniowe nazywa się gramatykami regularnymi.

Definicja (Postać Chomsky'ego, PNC, CNF)

Gramatyka bezkontekstowa $G2 = (N, T, P, S)$ jest w postaci Chomsky'ego, jeżeli wszystkie jej reguły są zapisane w jednej z trzech postaci: $S \rightarrow \varepsilon; A \rightarrow a; A \rightarrow B C$, gdzie $A, B, C \in N \wedge a \in T$.

Twierdzenie (Równoważność CNF)

Dla każdej gramatyki bezkontekstowej istnieje równoważna gramatyka bezkontekstowa w postaci Chomsky'ego

Odniesień do dowodu należy szukać w pracy Skórzewskiego [5].

Definicja (klasyfikator GCS)

Klasyfikator GCS jest zbiorem postaci $cl = \{P_L, P_P, f, u_p, u_n, p, d\}$, gdzie:

P_L – lewa strona produkcji, $P_L \in N$.

P_P – prawa strona produkcji, $P_P | P_L \rightarrow P_P, P_L \in N, P_P : a \vee P_P : X Y, a \in T, X, Y \in N$.

u_p – liczba użyć zdania w rozkładzie zdań poprawnych.

u_n – liczba użyć zdania w parsowaniu zdań niepoprawnych.

p – liczba punktów zdobyta przez algorytm w wyniku parsowania zdań poprawnych (profit).

d – liczba punktów zdobyta przez algorytm w wyniku parsowania zdań niepoprawnych (debt).

f – przystosowanie klasyfikatora.

Symbolem $Par_{cl}(G)$ będziemy oznaczać zbiór wszystkich klasyfikatorów gramatyki G.

Definicja (Gramatyka GCS)

Gramatykę generowaną podczas działania algorytmu GCS będziemy nazywać gramatyką bezkontekstową złożoną z reguł postaci CNF.

W odniesieniu do tej gramatyki będzie się stosować alternatywną reprezentację następującej postaci: $G_{GCS} = (N, T, P^T, P^N, S, S_U)$, gdzie:

N – zbiór symboli nieterminalnych;

T – zbiór symboli terminalnych.;

P^T – produkcje terminalne, $P^T = \{A \rightarrow a \mid A \in N, a \in T\}$;

P^N – produkcje nieterminalne, $P^N = \{A \rightarrow B C \mid A, B, C \in N\}$;

S – symbol startowy $S \in N$;

S_U – symbol uniwersalny $S_U \in N$.

Definicja(Drzewo wyprowadzenia/derywacji/rozkładu)

Niech $G = (N, T, P, S)$ będzie gramatyką bezkontekstową. Drzewo Γ jest drzewem wyprowadzenia (lub derywacji, albo rozkładu) dla gramatyki G , jeżeli:

- $\forall v \in \Gamma v$ jest etykietowany symbolem x , gdzie $x \in N \cup T$;
- $root(\Gamma)$ jest etykietowany symbolem S ;
- Jeżeli wierzchołek wewnętrzny jest etykietowany symbolem A , to $A \in N$;
- Jeżeli synowie wierzchołka drzewa Γ etykietowanego symbolem A mają kolejno etykiety $X_1, X_2, \dots, X_n \in N \cup T$, to $A \rightarrow X_1 X_2 \dots X_n$ jest produkcją ze zbioru P.

Definicja (Stochastyczna gramatyka bezkontekstowa)

Stochastyczna gramatyka bezkontekstowa sGCS to taki zbiór, że $G = (N, T, P, S, Q)$, gdzie:

- (N, T, P, S) jest gramatyką bezkontekstową;

$Q: P \rightarrow [0, 1]$ jest funkcją prawdopodobieństwa przydzielającą każdej regule $r \in P$ taką wartość prawdopodobieństwa, że suma prawdopodobieństw dla reguł o lewej stronie A wynosi 1.

Definicja (klasyfikator sGCS)

Klasyfikator sGCS jest zbiorem postaci $cl = \{P_L, P_P, P, f, u_p, p, t_{tmp}\}$, gdzie:

P_L – lewa strona produkcji, $P_L \in N$.

P_P – prawa strona produkcji, $P_P | P_L \rightarrow P_P, P_L \in N, P_P : a \vee P_P : X Y, a \in T, X, Y \in N$.

u_p – liczba użyć zdania w rozkładzie zdań poprawnych.

t_{tmp} – liczba zastosowań produkcji w najbardziej prawdopodobnych drzewach rozborów zdań.

p – liczba punktów zdobyta przez algorytm w wyniku parsowania zdań poprawnych (profit).

P – prawdopodobieństwo produkcji.

f – funkcja przystosowania (ang. fitness).

Definicja (Znormalizowane prawdopodobieństwo)

Prawdopodobieństwo znormalizowane to funkcja o sygnaturze: $R: X \rightarrow [0, 1]$. Przyporządkowuje ona każdej regule $x \in P^T \cup P^N$ prawdopodobieństwo $P(R)$, zachowując przy tym zależność $\sum_{A \in N} P(A \rightarrow \omega) = 1 \mid \omega: a \vee \omega: B \ C, B, C \in N, a \in T$.

Definicja (Gramatyka sGCS)

Gramatykę sGCS przedstawiamy w następującej postaci: $G_{sGCS} = (N, T, P^T, P^N, S, S_U, R)$. Jedyny nowy element w stosunku do gramatyki GCS to R , które jest znormalizowaną funkcją prawdopodobieństwa.

Definicja (Zdanie)

Mając alfabety A, B (takie, że $A \subseteq B$) oraz gramatykę G nad alfabetem A, przez zdanie będziemy rozumieć dwójkę postaci (x, w) , gdzie:

- $w \in L(B)$;
- $x = \begin{cases} \text{True}, & w \in L(G) \\ \text{False}, & w \text{ p.p.} \end{cases}$

Zatem zdanie jest krótką zawierającą słowo nad pewnym alfabetem oraz informację, czy należy ono do danej gramatyki.

Definicja (Przystosowanie)

Po wykonaniu parsowania na całym zbiorze uczącym każdy klasyfikator otrzymuje miarę swojej użyteczności, zwaną przystosowaniem.

Wyliczamy je przy pomocy tak zwanej funkcji przystosowania, zgodnie z poniższymi wzorami:

$$f = \frac{w_c f_c + w_f f_f}{w_c + w_f}$$

$$f_c = \begin{cases} \frac{w_p u_p}{w_n u_n + w_p u_p}, & u_n + u_p \neq 0 \\ f_0, & u_n + u_p = 0 \end{cases}$$

$$f_f = \frac{p - d - f_{f_{min}}}{f_{f_{max}} - f_{f_{min}}}$$

$$f_{f_{max}} = \max_{cl \in G}(p - d)$$

$$f_{f_{min}} = \min_{cl \in G}(p - d)$$

f_0 – miara użyteczności klasyfikatora niebiorącego udziału w parsowaniu;

f_c – klasyczna funkcja przystosowania;

f_f – funkcja płodności;

u_p – liczba wykorzystań klasyfikatora w parsowaniu zdań pozytywnych;

u_n – liczba wykorzystań klasyfikatora w parsowaniu zdań negatywnych;

p – suma punktów zdobytych za parsowanie zdań poprawnych;

d – suma punktów zdobytych za parsowanie zdań niepoprawnych;

w_p – waga rozboru zdania poprawnego;

w_n – waga rozboru zdania niepoprawnego;

w_c – waga funkcji klasycznej;

w_f – waga funkcji płodności;

$f_{f_{max}}$ – maksymalna liczba punktów zdobytych przez różnicę $(p - d)$;

$f_{f_{min}}$ – minimalna liczba punktów zdobytych przez różnicę $(p - d)$.

Definicja (Pozytywna reprezentacja gramatyki)

Pozytywna reprezentacja R gramatyki G to pewien podzbiór zdań należących do gramatyki G.

Definicja (Pelna reprezentacja gramatyki)

Pozytywna reprezentacja R gramatyki G to pewien podzbiór wszystkich możliwych zdań (zarówno należących, jak i nie do gramatyki G).

Algorytm CYK

Podczas indukcji gramatyki ważne jest ustalenie, czy dane zdanie należy do gramatyki (*ang. membership query*). Najbardziej popularnymi algorytmami parsowania gramatyki są algorytm CYK oraz Earleya. Algorytm CYK [1] (nazwa pochodzi od nazwisk twórców, Cocke'a, Youngera oraz Kasamiego) to dynamiczny algorytm mający za zadanie sprawdzenie, czy dane zdanie należy do gramatyki (Rysunek 1). Jego złożoność to $O(gn^3)$, gdzie n – długość sprawdzanego zdania, a g – rozmiar gramatyki. Warunkiem niezbędnym do zastosowania tego algorytmu jest posiadanie gramatyki przedstawionej za pomocą zbioru reguł w postaci normalnej Chomsky'ego. Poniżej (Rysunek 1) przedstawiono pseudokod algorytmu:

```
procedure CYK
begin
    n := len(x)
    for I := 1 to n do
         $V_{1,i} = \{A \mid A \rightarrow a \text{ jest produkcją, } a \text{ jest } i - \text{tym symbolem łańcucha } x\};$ 
        for j := 2 to n do
            for i := 1 to n do
                begin
                     $V_{j,i} := 0;$ 
                    for k := 1 to j - 1 do
                         $V_{j,i} := V_{j,i} \cup \{A \mid A \rightarrow B \text{ i } C \text{ jest produkcją, } B \in V_{k,i} \wedge C \in V_{j-k,i+k}\};$ 
                end
            end
        end
    end
```

Rysunek 1: Algorytm CYK

Działanie algorytmu polega na sprawdzaniu, czy $\exists A \in N, i \in \mathbb{N}, j \in \mathbb{N} \ A \xrightarrow{*} x_{i,j}$, gdzie $x_{i,j}$ jest podłańcuchem zdania x , rozpoczynającym się na pozycji i , mającym długość j . Jeżeli istnieje takie wyprowadzenie, wówczas algorytm zapamiętuje ten fakt. Stosując indukcję po j wyprowadzamy coraz dłuższe łańcuchy, aż po osiągnięciu $j = n$ możemy rozstrzygnąć, czy $S \xrightarrow{*} x_{1,n}$. Ponieważ $x_{1,n} = x$, zatem $x \in L(G) \Leftrightarrow S \xrightarrow{*} x_{1,n}$.

Dane funkcjonowania algorytmu CYK są zapisywane w tablicy trójkątnej o wymiarach $n \times n$. Zawartość komórki $[j, i]$ tej tabeli odpowiada zbiorowi $V_{j,i}$.

- $S \rightarrow AB$
- $A \rightarrow a$
- $B \rightarrow b$
- $A \rightarrow AB$
- $B \rightarrow AA$
- $A \rightarrow BB$

Rysunek 2: Przykładowa gramatyka

a	b	a	b	a
A	B	A	B	A
S, A	-	S, A	-	
B	-	B		
B, A	A			
B, S, A				

Rysunek 3: Przykładowa tabela CYK

Powyżej przedstawiono przykładowe wypełnienie tabeli CYK dla zdania $a b a b a$ (Rysunek 3) oraz gramatyki G (Rysunek 2). Tabelę tą rozszerzono dodatkowo o zerowy wiersz zawierający zdanie w celu lepszej reprezentacji idei algorytmu.

Pierwszy wiersz tabeli CYK uzupełnia się poprzez wpisanie produkcji wyprowadzających pojedyncze symbole nieterminalne. Następnie należy przejść do uzupełniania wierszu drugiego. Uzupełnia się go takimi produkcjami, że wyprowadzają one symbole nieterminalne z wyższych wierszy. Przykładowo chcąc uzupełnić komórkę $[3,3]$ musimy najpierw uzupełnić wiersze 1 oraz 2. Następnie znajdujemy taki zestaw produkcji $X \rightarrow L R$, że $X, L, R \in N \wedge L \in CYK[1,3] \wedge CYK[2,4] \wedge (X \rightarrow L R) \in P$. Zapisujemy wszystkie odnalezione w ten sposób wartości X w komórce $[3, 3]$. Ponieważ komórka $CYK[2, 4]$ nie zawiera żadnych symboli, więc w efekcie nie znajdujemy żadnej produkcji. Następnie szukamy w ten sam sposób kolejnych wartości X , tym razem z wykorzystaniem komórek $CYK[2,3], CYK[1,5]$. Nie istnieje żadna produkcja postaci $X \rightarrow S A$, więc kombinacja tych dwóch symboli nic nam nie daje. Następnie próbujemy dopasowania $X \rightarrow A A$. Przyglądając się zestawowi reguł przedstawionemu na [...] widzimy, że pod wartość X możemy podstawić symbol B . Dodajemy go więc do zbioru symboli w komórce $CYK[3,3]$. Nie istnieje również żadna produkcja postaci $X \rightarrow S A$. Skończyły nam się kombinacje symboli w komórkach $CYK[2,3], CYK[1,5]$, nie mamy również kolejnych kombinacji komórek do wzięcia. Przechodzimy do uzupełniania kolejnej komórki – czyli $CYK[4, 1]$.

Postępując w ten sposób uzupełniamy wszystkie komórki tabeli CYK. Dysponując pełnym zestawem symboli w $CYK[5, 1]$ jesteśmy w stanie ustalić czy dane zdanie zostało sklasyfikowane jako należące do gramatyki, czy też nie. Zdanie należy do gramatyki, jeżeli w tej komórce znajduje się symbol startowy gramatyki. Wówczas zdanie jest wyprowadzalne z symbolu startowego, co możemy pokazać odtwarzając całe wyprowadzenie na podstawie tabeli CYK. Jak widzimy zdanie $ababa$ należy do gramatyki G .

Tablica CYK zawiera wszystkie możliwe rozkłady zdania.

Wariantem, jaki możemy wprowadzić do tego algorytmu, jest wprowadzenie powielania symboli nieterminalnych w tabeli CYK. Zakładając, że mamy zdanie postaci $aaaaaa$ oraz gramatykę, na którą składają się wyłącznie reguły $S \rightarrow a$ i $S \rightarrow S S$ otrzymujemy tabelę danych, którą ilustruje Rysunek 4.

a	a	a	a	a
S	S	S	S	S
S	S	S	S	
S, S	S, S	S, S		
S, S, S, S, S	S, S, S, S, S			
S, S, S, S, S, S, S, S, S, S, S, S				

Rysunek 4: Przykładowa tabela CYK z powtarzającymi się produkcjami

Widać zatem, że CYK może w przypadku takiej reprezentacji osiągnąć złożoność $\Theta(\exp n)$. Wynika to z faktu, że liczba wszystkich możliwych wyprowadzeń ciągu o długości n jest funkcją wykładniczą n . Reprezentacja taka bywa jednak przydatna gdy chcemy później odtworzyć drzewo wywodu, w szczególności jeżeli zapamiętujemy dodatkowo informacje jednoznacznie określające skąd wziął się dany symbol. W aplikacji opisanej w tej pracy magisterskiej zdecydowano się na taką właśnie reprezentację.

GCS

Algorytm GCS jest wieloetapowym algorytmem mającym na celu wygenerowanie gramatyki bezkontekstowej służącej do rozwiązywania problemu klasyfikacji. Został on zaproponowany w publikacji Unolda [1]. Na gramatykę składają się reguły terminalne oraz nieterminalne zapisane w postaci normalnej Chomsky'ego (CNF) [5]. Generowanie odbywa się z wykorzystaniem zarówno pozytywnych, jak i negatywnych zdań w procesie uczącym (zgodnie z twierdzeniem Golda niemożliwa jest indukcja jedynie na podstawie zdań poprawnych).

Główna pętla algorytmu

```
procedure GCS
begin
    wczytaj zestaw zdań uczących  $R$ 
    wczytaj parametry algorytmu
    inicjuj gramatykę  $G_i = (N, T, P^T, P^N, S, S_U)$ 
    while (not kryterium stopu) do
        begin
            indukuj gramatykę  $G_i$ 
            oblicz  $f_G$  dla  $G_i$ 
            if  $f_{GA}$  then uruchom algorytm genetyczny na gramatyce  $G_i$ .
        end
    end
```

Na samym początku algorytm (Rysunek 5) wczytuje zestaw danych uczących. Zbiór ten zawiera zarówno zdania pozytywne, jak i negatywne. Następnie wczytywane są również parametry modelu, czyli parametry konfiguracyjne algorytmu. Algorytm GCS ma ich całkiem sporo

Rysunek 5: Algorytm GCS

i zostaną one szczegółowo omówione później.

Kolejnym krokiem jest inicjacja gramatyki G . Zadaniem inicjacji jest wygenerowanie populacji początkowej zawierającej n_{start} reguł nieterminalnych. Są one generowane w sposób losowy z wykorzystaniem symboli nieterminalnych ze zbioru N . Część lub całość reguł początkowych może być dostarczonych przez użytkownika. Następnie zaczyna się główna pętla algorytmu, która będzie wykonywana do czasu spełnienia warunków stopu. Warunkiem stopu może być osiągnięcie pewnego pułapu funkcji przystosowania f_G (zazwyczaj oczekuje się wartości 100%) lub osiągnięcie pewnej ilości kroków. Implementacja towarzysząca tej pracy umożliwia dodatkowo zakończenie procesu po upływie określonego czasu t_{max} . W każdym kroku przeprowadzamy indukcję gramatyki, ocenę jej przystosowania oraz (o ile pozwala na to wartość f_{GA}) uruchomienie algorytmu genetycznego (obecna implementacja umożliwia wykorzystanie innego algorytmu ewolucyjnego).

```
procedure Indukuj gramatykę ( $G$ )
begin
    inicjuj parametry produkcji  $Par_{cl}(G)$ 
     $G^* \leftarrow G$ 
     $Par_{cl}(G^*) \leftarrow Par_{cl}(G)$ 
    if  $f_{kor}$  then korekcja gramatyki  $G^*$ 
    for each  $r \in R$  do
        begin
            parsuj zdanie  $r$  gramatyką  $G^*$ 
            aktualizuj parametry produkcji  $Par_{cl}(G^*)$ 
        end
     $Par_{cl}(G) \leftarrow Par_{cl}(G^*)$ 
    for each  $cl \in G$  do oblicz  $f$ 
end
```

Indukcja gramatyki

Rysunek 6: Procedura IndukujGramatykę

Indukcję gramatyki (Rysunek 6) przeprowadza się poprzez parsowanie zdanie po zdaniu wszystkich zdań ze zbioru uczącego z wykorzystaniem mechanizmów tworzenia nowych reguł. Dane dotyczące klasyfikatorów zbieramy w $Par_{cl}(G)$. Po zakończeniu parsowania wszystkich zdań obliczamy przystosowanie wszystkich klasyfikatorów.

Parsowanie zdań

```

procedure Parsuj zdanie (r, G)
begin
     $n \leftarrow |r = a_1 a_2 \dots a_n|$ ,  $a_i$  jest  $i$ -tym słowem zdania  $r$  o długości  $n$ 
    inicjuj tablicę  $CYK[n, n]$  reprezentującą otoczenie
    for  $i := 1$  to  $n$  do
    begin
        inicjuj zbiory  $M, A, D$ 
         $D \leftarrow a_i \in D$ ,  $a_i$  jest komunikatem zewnętrznym reprezentującym aktualny stan otoczenia
         $M \leftarrow \{X \in N \mid X \rightarrow a_i, a_i \in D\}$ 
        if  $|M| = 0$  then
            begin
                zastosuj operator pokrycia terminalnego dla  $a_i$ 
                if  $f_{cs} \wedge n = 1$  then zastosuj operator pokrycia startowego dla  $a_i$ 
            end
             $A \leftarrow M$ 
             $CYK[1, i] \leftarrow A$ , czyli uaktywnij akcje w otoczeniu
        end
        for j:= 2 to  $n$  do
            for  $i := 1$  to  $n$  do
            begin
                inicjuj zbiory  $M, A, D$ 
                 $r_j \leftarrow a_i a_{i+1} \dots a_{i+j-1}$ , podciąg zdania o długości  $j$ , rozpoczynający się od  $i$ -tej
                pozycji
                 $D \leftarrow \{B \mid B, C \text{ wyprowadzają odpowiednio pierwszą i drugą część } r_j\}$ ,
                czyli utwórz komunikat reprezentujący aktualny stan otoczenia
                 $M \leftarrow \{X \in N \mid X \rightarrow B C, BC \in D\}$ 
                if  $|M| = 0 \wedge r \in R^+$  then
                    begin
                        if liczba losowa z  $[0, 1] < p_a$  then zastosuj operator pokrycia
                        agresywnego dla  $D$ 
                        if  $f_{cp} \wedge a = 1 \wedge j = n$  then zastosuj operator pokrycia pełnego dla  $D$ 
                    end
                     $A \leftarrow M$ 
                     $CYK[j, i] \leftarrow A$ , czyli uaktywnij akcję w otoczeniu
            end
    end
end

```

Na samym początku zadaniem algorytmu (Rysunek 7) ma być uzupełnienie pierwszego wiersza tabeli CYK . Od zwykłego wypełniania pierwszego wiersza przy pomocy algorytmu CYK różni się to tylko tym, że w przypadku natrafienia na nieznany symbol uruchamiamy operator pokrycia terminalnego. Dodatkowo jeżeli zdanie składa się z pojedynczego słowa i należy przy tym do gramatyki wówczas możemy uruchomić dodatkowo operator pokrycia startowego. Zadaniem tych operatorów jest dodanie nowych reguł, z symbolem terminalnym a_i po prawej stronie, do zbioru reguł terminalnych oraz rozszerzenie zbioru M przez uwzględnienie tych dodatkowych produkcji.

Rysunek 7: Procedura parsowania zdania

Proces uzupełniania wierszy kolejnych również jest tak naprawdę algorymem CYK rozszerzonym o operatory pokrycia. Są to operatory pokrycia agresywnego oraz pełnego. Są one wykorzystywane do stworzenia nowych reguł nieterminalnych o prawej stronie D . Reguły te w klasycznym wydaniu algorytmu GCS są dodawane z tzw. ściskiem, załączona implementacja umożliwia zmianę tego zachowania.

Ścisk

```

procedure Dodaj ze ściskiem (nowy, P)
begin
     $K \leftarrow \emptyset$ 
    for  $i := 1$  to  $cf$  do
    begin
         $W^{cs} \subseteq P$ , losowa podpopulacja z  $P$  o wielkości  $cs$ 
         $X \leftarrow \text{najsłabszy osobnik z } W^{cs}$ 
         $K \leftarrow K \cup X$ 
    end
     $Y \in K \leftarrow \text{najbardziej podobny osobnik do nowy}$ 
     $P \leftarrow P \setminus Y$ 
     $P \leftarrow P \cup \{\text{nowy}\}$ 
end

```

Rysunek 8: Algorytm ścisku

Operatory pokrycia startowego, pełnego oraz startowego, podobnie jak algorytm ewolucyjny dodają reguły do populacji z tak zwanym ściskiem (Rysunek 8). Jego zadaniem jest dodanie nowej reguły do populacji w taki sposób, żeby jej rozmiar nie uległ zwiększeniu, poprzez zastąpienie nową regułą podobną, acz mało skutecznej metody już istniejącej w populacji. Parametrami są *nowy* osobnik, nazwany nowy oraz populacja reguł P . Na początku procedura ścisku wybiera z P podpopulację reguł W^{cs} o rozmiarze cs . Bierze z tej populacji najsłabszego osobnika i dodaje go do zbioru zbioru K . Procedurę tą kontynuujemy, dopóki nie dysponujemy zbiorem K o rozmiarze cf . Ponieważ dodawaliśmy tam tylko n najsłabszych osobników, zatem jest to zbiór reguł o potencjalnie niskich współczynnikach fitness, które nie wydają się być potrzebne w naszej populacji. Następnie w tej populacji jest szukany najbardziej podobny do nowego osobnik Y . Miarą podobieństwa jest tutaj liczba takich samych symboli występujących w obu regułach na tych samych pozycjach (na przykład podobieństwo reguł $S \rightarrow A B$ oraz $G \rightarrow B B$ wynosi 1, gdyż w przypadku obu reguł ich prawe dziecko to B). Reguła ta zostaje zastąpiona przez nowego osobnika.

Operatory pokrycia

Zadaniem operatorów pokrycia jest dodawanie nowych reguł do populacji. Zaimplementowana biblioteka pozwala zmienić domyślną metodę dodawania reguł przez operatory, aczkolwiek domyślnym zachowaniem jest dodawanie przez operatory pokrycia terminalnego oraz uniwersalnego reguł jako nowe pozycje, zaś startowego, agresywnego oraz pełnego ze ściskiem. Dodatkowo umożliwia zastosowanie prawdopodobieństwa uruchomienia również względem operatorów wymagających domyślnie zmiennych dyskretnych (wykorzystanie klasyczne również jest możliwe, 100% jest odpowiednikiem true, zaś 0% - false). Czyni to jednak cały model spójniejszym oraz daje nowe możliwości (np. symbol uniwersalny wyprowadzający tylko część symboli terminalnych. Możliwe jest też rozszerzenie o elityzm).

Operator pokrycia terminalnego

```

procedure Operator pokrycia terminalnego (a)
begin
    utwórz  $X \in N \rightarrow a$ 
     $P^T \leftarrow P^T \cup \{X \rightarrow a\}$ 
     $M \leftarrow M \cup \{X\}$ 
    if  $f_{cu}$  then zastosuj operator pokrycia uniwersalnego dla  $a$ 
end

```

Rysunek 9: Operator pokrycia terminalnego

Operator pokrycia terminalnego (Rysunek 9) jest uruchamiany, gdy w populacji brakuje reguły wyprowadzającej dany symbol terminalny. Zadaniem tego operatora jest wówczas dodanie takiej reguły do populacji (co powoduje zwiększenie rozmiaru populacji), a następnie uruchomienie operatora pokrycia uniwersalnego, jeżeli pozwala na to wartość f_{cu} .

Operator pokrycia uniwersalnego

```
procedure Operator pokrycia uniwersalnego (a)
begin
    utwórz  $S_u \rightarrow a$ 
     $P^T \leftarrow P^T \cup \{S_u \rightarrow a\}$ 
     $M \leftarrow M \cup \{S_u\}$ 
end
```

Rysunek 10: Operator pokrycia uniwersalnego

Operator pokrycia uniwersalnego (Rysunek 10) ma za zadanie wyprowadzić produkcje postaci $S_u \rightarrow a$. Ponieważ operator pokrycia uniwersalnego jest uruchamiany razem z operatorem pokrycia terminalnego, uruchamianego przy każdym nowym słowie, więc $\forall a \in T \exists S_u \rightarrow a \in P^T$, czyli $\forall a \in T S_u \xrightarrow{*} a$.

Operator pokrycia startowego

```
procedure Operator pokrycia startowego (a)
begin
    utwórz  $S \rightarrow a$ 
    dodaj ze ściskiem  $S \rightarrow a$  do  $P^T$ 
     $M \leftarrow M \cup \{S\}$ 
end
```

Rysunek 11: Operator pokrycia startowego

Uruchomienie operatora pokrycia startowego (Rysunek 11) jest uzależnione od zmiennej f_{cs} . Dodatkowo jest on stosowany wyłącznie w przypadku pozytywnych zdań długości 1. Zadaniem tego operatora jest dodanie do populacji produkcji wyprowadzających dane symbole terminalne bezpośrednio z symbolu startowego. Są one dodawane do populacji ze ściskiem.

Operator pokrycia agresywnego

```
procedure Operator pokrycia agresywnego (D)
begin
    utwórz  $X \rightarrow BC \mid X \in N, BC \in D$ 
    dodaj ze ściskiem  $X \rightarrow BC$  do  $P^N$ 
     $M \leftarrow M \cup \{X\}$ 
end
```

Rysunek 12: Operator pokrycia agresywnego

Uruchomienie operatora pokrycia agresywnego (Rysunek 12) jest wykonywane z prawdopodobieństwem p_a . Operator ten dodaje ze ściskiem nową regułę do populacji reguł nieterminalnych.

Operator pokrycia pełnego

```
procedure Operator pokrycia pełnego (D)
begin
    utwórz  $S \rightarrow BC \mid BC \in D$ 
    dodaj ze ściskiem  $S \rightarrow BC$  do  $P^N$ 
     $M \leftarrow M \cup \{S\}$ 
end
```

Operator pokrycia pełnego (Rysunek 13) tworzy produkcję, której lewa strona jest symbolem startowym, zaś prawa należy do zbioru D .

Rysunek 13: Operator pokrycia pełnego

Jest on uruchamiany tylko w ostatniej komórce tabeli CYK, wyłącznie w przypadku rozbioru zdania pozytywnego. Nowa reguła jest następnie dodawana ze ściskiem do pozostałych reguł.

Algorytm genetyczny

```

procedure Algorytm genetyczny (G)
begin
     $P_1 \in P^N \leftarrow$  dokonaj selekcji produkcji z  $P^N$ 
     $P_2 \in P^N \leftarrow$  dokonaj selekcji produkcji z  $P^N$ 
     $P_1^* \leftarrow P_1$ 
     $P_2^* \leftarrow P_2$ 
     $P_{n_{elit}}^{max} \in P^N \leftarrow$  wskaz  $n_{elit}$  najlepszych produkcji z  $P^N$ 
    if liczba losowa z  $[0, 1] < p_k$  then zastosuj krzyżowanie na  $P_1^*$  i  $P_2^*$ 
        zastosuj mutację na  $P_1^*$ 
        zastosuj mutację na  $P_2^*$ 
    if liczba losowa z  $[0, 1] < p_i$  then zastosuj inwersję na  $P_1^*$ 
    if liczba losowa z  $[0, 1] < p_i$  then zastosuj inwersję na  $P_2^*$ 
     $P^N \leftarrow P^N \setminus P_{n_{elit}}^{max}$ 
    dodaj ze ściskiem  $P_1^*$  do  $P^N$ 
    dodaj ze ściskiem  $P_2^*$  do  $P^N$ 
     $P^N \leftarrow P^N \cup P_{n_{elit}}^{max}$ 
end

```

Rysunek 14: Algorytm genetyczny

Algorytm genetyczny (Rysunek 14) wraz z jego działaniem został szczegółowo opisany w [7]. Na początku ma miejsce selekcja dwóch osobników z populacji reguł nieterminalnych. Rodzaj selekcji jest zależny od zmiennych decyzyjnych f_{GA}^1 oraz f_{GA}^2 . Dostępne rodzaje selekcji to selekcja metodą losową, ruletki lub turniejową. Następnie na kopiach wybranych produkcji są wykonywane operacje genetyczne (krzyżowanie, mutacja oraz inwersja). Wynik działania zostaje dodany do populacji ze ściskiem, chroniąc równocześnie przed usunięciem n_{elit} najlepszych produkcji przy pomocy mechanizmu elityzmu.

Metody selekcji

Istnieje wiele metod selekcji, tutaj jednak zostaną opisane trzy najpopularniejsze.

Selekcja ruletkowa

W selekcji ruletkowej szansa na wybranie klasyfikatora to wartość jego przystosowania f w stosunku do sumy f wszystkich klasyfikatorów. Bierzymy wartości f dla wszystkich klasyfikatorów i budujemy z nich ruletkę; tj. każdemu klasyfikatorowi przypisujemy wycinek koła proporcjonalny do prawdopodobieństwa jego wyboru. Selekcja jest dokonywana po losowym obrocie koła ruletki.

Selekcja turniejowa

W selekcji turniejowej losujemy populację o rozmiarze ts . Z tak wylosowanej podpopulacji wybieramy klasyfikator o najwyższej wartości funkcji przystosowania f .

Selekcja losowa

Zwraca losowy klasyfikator zgodnie z rozkładem normalnym, nie uwzględniając przy tym funkcji przystosowania.

Inwersja

```

procedure Inwersja(p)
begin
    utwórz  $p^* := A \rightarrow B C \mid A, B, C \in N$ 
     $p^* \leftarrow p$ 
     $p^* \leftarrow (A \rightarrow C B)$ 
     $p \rightarrow p^*$ 
end

```

Rysunek 15: Operator inwersji

Operator inwersji (Rysunek 15) jest uruchamiany z prawdopodobieństwem p_i dla każdej produkcji. Powoduje on odwrócenie kolejności symboli po prawej stronie produkcji.

Mutacja

```
procedure Mutacja(p)
begin
    utwórz  $p^* := A \rightarrow B C \mid A, B, C \in N$ 
     $p^* \leftarrow p$ 
    if liczba losowa z  $[0, 1] < p_m$  then  $p^* \leftarrow (X \rightarrow B C) \mid X \in N$ 
    if liczba losowa z  $[0, 1] < p_m$  then  $p^* \leftarrow (A \rightarrow X C) \mid X \in N$ 
    if liczba losowa z  $[0, 1] < p_m$  then  $p^* \leftarrow (A \rightarrow B X) \mid X \in N$ 
     $p \rightarrow p^*$ 
end
```

Rysunek 16: Operator mutacji

Mutacja (Rysunek 16) również jest uruchamiana dla każdej produkcji. Z prawdopodobieństwem p_m zmianie może ulec lewa strona produkcji, pierwszy symbol prawej strony produkcji oraz drugi symbol prawej strony produkcji. Test prawdopodobieństwa powtarzamy dla każdego z elementów, również symbol X jest losowy i przy każdym z tych testów może przyjąć inną wartość.

Krzyżowanie

```
procedure Krzyżowanie( $p_1, p_2$ )
begin
    utwórz  $p_1^* := A \rightarrow B C \mid A, B, C \in N$ 
    utwórz  $p_2^* := D \rightarrow E F \mid D, E, F \in N$ 
     $p_1^* \leftarrow p_1$ 
     $p_2^* \leftarrow p_2$ 
    if liczba losowa z  $\{1, 2\} = 1$  then
        begin
             $p_1^* \leftarrow (A \rightarrow E C)$ 
             $p_2^* \leftarrow (D \rightarrow B F)$ 
        end
    else
        begin
             $p_1^* \leftarrow (A \rightarrow B F)$ 
             $p_2^* \leftarrow (D \rightarrow E C)$ 
        end
    end
     $p_1^* \leftarrow (D \rightarrow B C)$ 
     $p_2^* \leftarrow (A \rightarrow E F)$ 
     $p_1 \leftarrow p_1^*$ 
     $p_2 \leftarrow p_2^*$ 
end
```

Rysunek 17: Operator krzyżowania

Krzyżowanie (Rysunek 17) dwóch produkcji powoduje zamianę pierwszego lub drugiego symbolu prawej strony dwóch reguł p_1, p_2 . Następnie reguły te zamieniają się lewymi stronami. Krzyżowanie jest wykonywane z prawdopodobieństwem p_k .

Korekcja gramatyki

```
procedure Korekcja gramatyki(G)
begin
    usuń produkcje redundantne z G
    usuń produkcje nieproduktywne z G
    usuń produkcje nieosiągalne z G
end
```

Rysunek 18: Algorytm korekcji gramatyki

Algorytm korekcji gramatyki (Rysunek 18) ma za zadanie usunąć z niej zbędne produkcje, które powstają w trakcie indukcji oraz ewolucji gramatyki. Wykonuje to zadanie stosując usuwanie produkcji redundantnych, nieproduktywnych oraz nieosiągalnych.

Usuwanie produkcji redundantnych

Usuwanie produkcji polega na przejrzeniu populacji i usunięciu powtarzających się reguł. Ze względu na konstrukcję populacji w dostarczonej implementacji duplikaty reguł nie są w ogóle dodawane do populacji, więc krok ten nie jest potrzebny.

Usuwanie produkcji nieproduktywnych

```
procedure Usuń produkcje nieproduktywne(G)
begin
     $N^* \leftarrow \{X \mid X \rightarrow a, a \in T\}$ 
    repeat
        for each  $X \rightarrow B C \in P^N$  do
            if  $\{B, C\} \subseteq N^*$  then  $N^* \leftarrow N^* \cup \{X\}$ 
    until  $N^*$  się nie zmienia
     $P^N \leftarrow \{X \rightarrow B C \in P^N \mid B, C \in N^*\}$ 
end
```

Rysunek 19: Algorytm usuwania produkcji nieproduktywnych

Kolejnym zadaniem jest usunięcie produkcji nieproduktywnych (Rysunek 19), czyli takich reguł z których nie da się wyprowadzić ciągu złożonego wyłącznie z symboli nieterminalnych.

Na samym początku w zbiorze N^* zapisujemy takie reguły, które bezpośrednio wyprowadzają symbole terminalne. Następnie przeglądamy populacje reguł, dodając do zbioru N^* takie produkcje, które bezpośrednio wyprowadzają jedynie symbole znajdujące się już w N^* . Postępując w ten sposób cały czas powiększamy zbiór N^* . Korzystając z indukcyjnej właściwości wyprowadzalności w zbiorze N^* zawsze znajdują się wyłącznie symbole wyprowadzające co najmniej jeden ciąg złożony wyłącznie z symboli nieterminalnych. Kiedy nie ulegnie on już zmianie oznacza to będzie, że w N^* mamy zbiór wszystkich symboli wyprowadzających co najmniej jeden ciąg złożony wyłącznie z symboli terminalnych. Wystarczy teraz pozostawić takie reguły, których prawe strony składają się jedynie z symboli w N^* .

Usuwanie produkcji nieosiągalnych

```
procedure Usuń produkcje nieosiągalne(G)
begin
     $N^* \leftarrow \{S\}$ 
     $P^* \leftarrow \emptyset$ 
    repeat
        for each  $\{X \rightarrow B C \in P^N \mid X \in N^*\}$  do
            begin
                 $N^* \leftarrow N^* \cup \{B, C\}$ 
                 $P^* \leftarrow P^* \cup \{X \rightarrow B C\}$ 
            end
    until  $N^*$  i  $P^*$  się nie zmienia
     $N \leftarrow N^*$ 
     $P^N \leftarrow P^*$ 
end
```

Rysunek 20: Algorytm usuwania produkcji nieosiągalnych

Produkcje nieosiągalne (Rysunek 20) to takie produkcje, których nie da się osiągnąć zaczynając wyprowadzenie z S .

Tworzymy dwa zbiory – N^* , czyli zbiór symboli osiągalnych z S oraz P^* - zbiór produkcji osiągalnych. W szczególności S jest symbolem osiągalnym z S , toteż od początku znajduje się w N^* . Następnie przeglądamy reguły, dodając do P^* produkcje o lewej stronie z N^* , dodając symbole z ich prawej strony do N^* . Tym sposobem coraz bardziej powiększamy zbiór symboli osiągalnych z S , powiększamy równocześnie zbiór produkcji osiągalnych. Kiedy żaden z tych zbiorów nie ulega już zmianie oznacza to, że w P^* jest zbiór wszystkich symboli osiągalnych z S , więc w celu usunięcia produkcji nieosiągalnych wystarczy jedynie przypisać go jako nowy zbiór P^N .

sGCS

Kolejnym algorytmem wykorzystywany w tej pracy magisterskiej jest algorytm sGCS (*ang. Stochastic Grammar-based Classifier System*). System ten stanowi rozszerzenie algorytmu GCS o obsługę omówionych wcześniej stochastycznych gramatyk bezkontekstowych. Został on wprowadzony i szczegółowo przedstawiony w pracy magisterskiej Kępy [8], a następnie zmodyfikowany i ulepszony w pracy Pasieki [2]. Poniższy rozdział skupi się na omówieniu różnic pomiędzy dwoma algorytmami. Pojawią się nowe elementy, niewystępujące w algorytmie GCS, niezbędne do indukowania struktury oraz parametrów gramatyki stochastycznej.

Algorytm CYK+

Pseudokod algorytmu CYK+ wydaje się znacznie różnić od pseudokodu algorytmu CYK. Jeżeli jednak zastanowić się nad samą funkcjonalnością to wprowadza on w zasadzie dwie rzeczy – po pierwsze pamięta w danych komórkach prawdopodobieństwa danych poddrzew rozbioru zdania, po drugie zapamiętuje informacje pomocnicze w odtworzeniu drzew lub drzева wywodu. Obie wersje pseudokodu możemy obejrzeć w pracy Kępy [8], tutaj jednak skupimy się wyłącznie na wyjaśnieniu wprowadzonych funkcjonalności.

- $S \rightarrow AB (1)$
- $A \rightarrow a (0,4)$
- $B \rightarrow b (0,5)$
- $A \rightarrow AB (0,3)$
- $B \rightarrow AA (0,5)$
- $A \rightarrow BB (0,3)$

Rysunek 21: Przykładowa gramatyka stochastyczna

a	b	a	b	a
A(0,4)	B(0,5)	A(0,4)	B(0,5)	A(0,4)
S(0,2), A(0,06)	-	S(0,2), A(0,06)	-	
B(0,012)	-	B(0,012)		
B(0,0018), A(0,0048)	A(0,0018)			
B(0,00096), S(0,00072), A(0,00072)				

Rysunek 22: Przykładowa tabela CYK+

Patrząc na Rysunek 21, przedstawiający reguły gramatyki stochastycznej oraz na Rysunek 22, przedstawiający przykładowy rozkład zdania *ababa* przy pomocy tych reguł widać, że komórki tabeli CYK+ uzupełnia się dokładnie w ten sam sposób jak w przypadku CYK. Dodatkowo w pierwszym wierszu wpisujemy prawdopodobieństwo dla reguły terminalnej wyprowadzającej dany symbol terminalny. W przypadku kolejnych wierszy sytuacja trochę się komplikuje – często symbol w danej komórce da się wyprowadzić na więcej niż jeden sposób. Istnieje wiele sposobów obliczania prawdopodobieństwa w takich sytuacjach (wprowadza je między innymi Pasieka [2]), poniżej przedstawiono dwa najpopularniejsze:

Podejście Viterbi

To podejście dąży do maksymalizacji prawdopodobieństwa wyprowadzenia danego ciągu. Jeżeli w komórce tabeli CYK+ znajduje się już dany symbol, wówczas zapamiętujemy większe z dwóch prawdopodobieństw (patrz Rysunek 23).

$$CYK[i][j][P_L] = R(P_L \rightarrow P_p) * P(A) * P(B)|P_p: A B$$

Rysunek 23: Prawdopodobieństwo przy podejściu Viterbi

Podejście Baum-Welch

W tym podejściu sumujemy prawdopodobieństwa wszystkich poddrzew wyprowadzenia danego ciągu (Rysunek 24).

$$CYK[i][j][P_L] = CYK[i][j][P_L] + R(P_L \rightarrow P_P) * P(A) * P(B) | P_P : A B$$

Rysunek 24: Prawdopodobieństwo przy podejściu Baum-Welch

Poza samym prawdopodobieństwem musimy również przechowywać w jakiś sposób dane umożliwiające łatwe odtworzenie drzewa derywacji przez algorytm Traceback. Najłatwiej tego dokonać dodając do każdego symbolu w tabeli CYK system koordynatów umożliwiający jednoznaczne określenie kto jest jego rodzicem. W obecnej implementacji wykonuje się przy pomocy listy krotek *Production*, które są następującej postaci (Rysunek 25):

$$Prod = (row, col, shift, left_{id}, right_{id}, rule, prob)$$

Rysunek 25: Przechowywanie śladu drzewa derywacji

row – wiersz obecnie rozpatrywanego symbolu.

col – kolumna obecnie rozpatrywanego symbolu.

shift – z której to z kolei kombinacji komórek uzyskano symbol. Wartość pozwala łatwo odtworzyć informację wg poniższego wzoru:

$$coord(A) = \{shift, col\}$$

$$coord(B) = \{row - shift, col + shift + 1\}$$

left_{id} – wartość umożliwiająca jednoznaczne określenie, który symbol z komórki o współrzędnych *coord(A)* jest rodzicem.

right_{id} – wartość umożliwiająca jednoznaczne określenie, który symbol z komórki o współrzędnych *coord(B)* jest rodzicem.

rule – reguła wyprowadzająca z obecnego symbolu *A* oraz *B*.

prob – prawdopodobieństwo policzone przy pomocy jednego z podejść.

Traceback

```
procedure Traceback(CYK)
begin
    root ← CYK[n][0][S]

    function aux(węzeł)
    begin
        aktualizuj fitness węzeł
        if ma rodziców(węzeł) then
            begin
                aux(Pp.A)
                aux(Pp.B)
            end
        end
        aux(root)
    end
end
```

Algorytm (Rysunek 26) umożliwiający odtwarzanie drzew derywacji. Jest on niezbędny do prawidłowego aktualizowania funkcji przystosowania algorytmu *SGCS*. Zazwyczaj porusza się on po najbardziej prawdopodobnym drzewie rozkładu, chociaż istnieją też inne implementacje [2].

Rysunek 26: Algorytm Traceback

Normalizacja

Dodając nowe i usuwając stare reguły należy pamiętać o tym, że zgodnie z definicją gramatyki stochastycznej prawdopodobieństwo musi być znormalizowane. Obecna implementacja dokonuje tego w locie, obliczając prawdopodobieństwo zgodnie z poniższym wzorem:

$$P_{NOR}(A \rightarrow \alpha) = \frac{P(A \rightarrow \alpha)}{\sum_{\omega \in P^T \cup P^N} P(A \rightarrow \omega)}$$

Rysunek 27: Wzór na normalizację prawdopodobieństwa

Wartości te są pamiętane, dzięki czemu nie jest konieczne liczenie ich za każdym razem. Zatem normalizacja odbywa się poprzez podzielenie prawdopodobieństwa interesującej nas reguły przez prawdopodobieństwo wszystkich reguł o takiej samej lewej stronie.

W przypadku dodawania nowych reguł ich prawdopodobieństwo jest losowane. Później wartość ta staje się zależna od wartości funkcji przystosowania.

Przystosowanie w sGCS

W przypadku funkcji przystosowania za Pasieką [2] stosujemy funkcję przystosowania opartą na liczbie wystąpień zdania w drzewach wyprowadzeń zdań pozytywnych (Rysunek 28).

$$f = \text{Count}(A \rightarrow \alpha)$$

Rysunek 28: Przystosowanie w sGCS

gdzie $\text{Count}(X)$ – liczbę wystąpień danej reguły w zbiorze zdań pozytywnych.

Estymacja prawdopodobieństw

Kolejnym ważnym elementem jest moduł estymacji. Zadaniem tego modułu jest obliczenie wartości prawdopodobieństw reguł. Jest ono przeliczane po każdym zakończonym etapie indukcyjnym.

Proces estymacji wykorzystuje obliczone wcześniej wartości parametru *fitness*, aktualizowany przez każde uruchomienie algorytmu Traceback. Poniżej (Rysunek 29) przedstawiono sposób obliczania prawdopodobieństwa na etapie estymacji:

$$P(A \rightarrow \alpha) = \frac{f}{\sum_{\omega \in P^T \cup P^N} \text{Count}(A \rightarrow \omega)}$$

Rysunek 29: Estymacja prawdopodobieństw w sGCS

neg-sGCS

Algorytm sGCS nie wykorzystuje w procesie uczenia zdań negatywnych. W dotychczasowych badaniach [8] [2] ten obiecujący algorytm często osiągał stosunkowo słabe rezultaty. Jest to spowodowane niepełnością wiedzy w przypadku podawaniu algorytmowi jedynie pozytywnych przykładów – o ile jest zawsze powinno być możliwe wygenerowanie dla pozytywnej reprezentacji języka $L(G)$ takiej gramatyki stochastycznej G_1 , że $L(G) = L(G_1)$, to dotychczasowe badania [8] pokazują, że znacznie bardziej prawdopodobne jest wygenerowanie nadgramatyki, czyli takiej gramatyki G_2 , że $L(G) \subseteq L(G_2)$. Prowadzi to do nieprawidłowego rozpoznawania zdań negatywnych jako pozytywne, czego chcielibyśmy uniknąć. Praca ta podejmuje próbę realizacji wariantu sGCS, który byłby w stanie uczyć się na zdaniach negatywnych.

Algorytm neg-sGCS nie różni się praktycznie niczym od algorytmu sGCS. Jedyne zmiany to:

- Wykorzystanie zdań negatywnych w procesie uczenia (tj. wysyłanie ich w ogóle do parsera CYK+);
- Stosowanie funkcji fitness GCS – stosujemy tutaj system nagrody i kary (profit oraz debt) za parsowanie zdań pozytywnych oraz negatywnych. Stosujemy tutaj wagę nagrody za parsowanie zdań pozytywnych oraz kary za parsowanie zdań negatywnych jak w przypadku GCS opisanego w publikacji Unolda [1], czyli odpowiednio 1 i 2. Początkowo istniała obawa, że zastosowanie takich wag spowoduje zmianę zachowania algorytmu w drugą stronę – tj. generowanie gramatyk u których rozpoznawanie zdań pozytywnych stoi na niskim poziomie. Jak pokazały jednak badania zjawisko to nie wystąpiło.

Implementacja

Aplikacja została zaimplementowana jako biblioteka języka Python w wersji 3.4. Może ona zatem być wykorzystywana w dowolnym środowisku posiadającym dostęp do interpretera języka Python w wersji 3.4 lub nowszej oraz niezbędne biblioteki.

Biblioteki niezbędne do wykorzystania aplikacji zostały wymienione w pliku „requirements.txt” (znajdującym się w głównym katalogu projektu). Są to:

- PyHamcrest – wymagany do uruchomienia testów. Jest to biblioteka dostarczająca zestaw rozszerzeń do standardowego modułu testów jednostkowych unittest, które czynią testy wysoce czytelnymi i łatwymi w utrzymaniu;
- Matplotlib – biblioteka umożliwiająca tworzenie wykresów dla języka Python, jest wykorzystywana do rysowania diagramów procesu uczenia gramatyki;
- Mock – wykorzystywany do tworzenia atrap obiektów (ang. Mock Object) na potrzeby testowania aplikacji. W bibliotece wykorzystany w testach jednostkowych, modułowych oraz komponentowych;
- Psutil – wykorzystywany do pobierania informacji o obecnym obciążeniu procesora i zajętej pamięci niezależnie od systemu operacyjnego;

Wszystkie te biblioteki możemy bardzo łatwo zainstalować przy pomocy narzędzia do instalowania modułów pip.

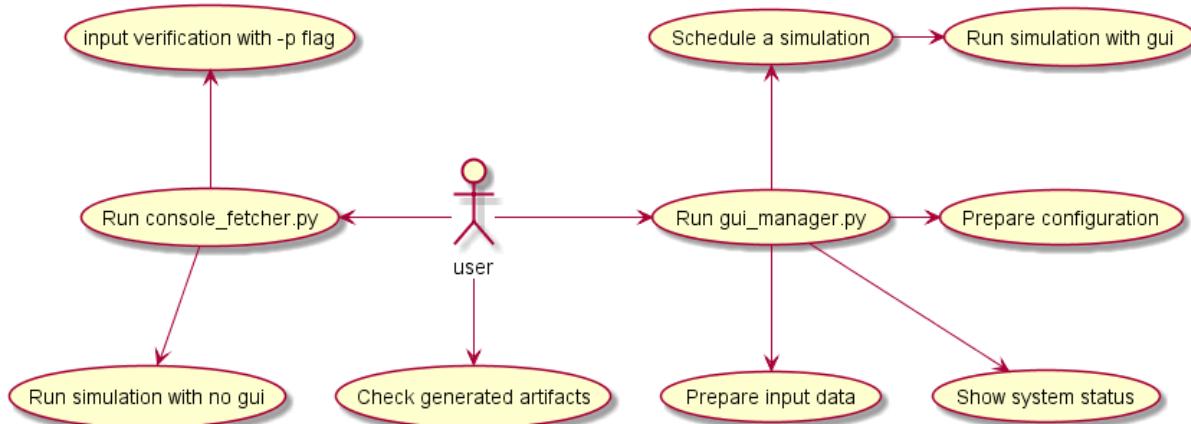
pip jest wygodnym menedżerem pakietów języka Python. Stanowi ważne narzędzie dla programistów rozwijających aplikacje w tym języku. Umożliwia łatwe instalowanie oraz usuwanie bibliotek języka Python. zazwyczaj wystarczy polecenie *sudo pip install nazwa – pakietu*, aby pip automatycznie pobrał pakiet wraz ze wszystkimi niezbędnymi zależnościami, zainstalował je i uczynił widocznymi z poziomu dowolnej aplikacji w języku Python uruchamianej na naszym komputerze. Oprócz ręcznego instalowania bibliotek jedna po drugiej istnieje możliwość zebrania ich nazw w pliku tekstowym (nazwijmy go na przykład requirements.txt), umieszczając każdą nazwę w osobnej linii, a następnie uruchomić tylko polecenie *sudo pip install – r requirements.txt*. Biblioteka wykorzystuje to drugie podejście – wywołanie *sudo pip install – r requirements.txt* na najwyższym poziomie projektu spowoduje zainstalowanie wszystkich zależności niezbędnych do uruchomienia algorytmu w trybie konsolowym.

W celu uruchomienia aplikacji w trybie okienkowym należy dodatkowo posiadać zainstalowany pakiet PyQt oraz wykorzystywane przez niego biblioteki Qt w odpowiedniej wersji. Ponieważ proces ten różni się znacznie na różnych systemach, a jest on niezbędny tylko do uruchomienia przykładowej otoczki graficznej dostarczonej biblioteki, więc nie zostanie on tutaj omówiony. Informacji na temat instalacji biblioteki PyQt oraz Qt na posiadany systemie należy szukać na stronie twórców PyQt, Qt oraz na forach poświęconych konkretnym systemom.

Podczas pisania aplikacji wykorzystywano metodologię wytwarzania oprogramowania TDD (*ang. Test Driven Development*). Stosowanie tej metodologii polega na pisaniu w pierwszej kolejności testu, a dopiero później kodu umożliwiającego zdanie go. Zwiększa to pokrycie kodu, gdyż zgodnie z wytycznymi tej metodologii każda linijka kodu właściwego powstaje w celu zdania jakiegoś testu, co przynajmniej w teorii powinno zapewnić 100% pokrycia kodu przez testy. Oznacza to, że każda linia kodu zostaje odwiedzona przez jakiś test w chwili uruchomienia ich wszystkich, co daje przyzwoicie przetestowany program wynikowy. W rzeczywistości osiągnięto niższy, aczkolwiek całkiem wysoki poziom pokrycia 88% linii kodu. Móglby być on wyższy gdyby nie ograniczenia czasowe uniemożliwiające prawidłowe pokrycie testami jednej z najwyższych warstwy aplikacji – warstwy logiki biznesowej. Poza testami jednostkowymi w aplikacji znajdują się również testy modułowe oraz komponentowe (za komponent uznajemy tutaj całą właściwą część algorytmu, pozbawioną dodatków typu części odpowiedzialne za rysowanie diagramów, obsługę GUI itp.). W przypadku testów modułowych oraz komponentowych również wykorzystano biblioteki PyHamcrest unittest oraz Mock.

Użytkowanie aplikacji

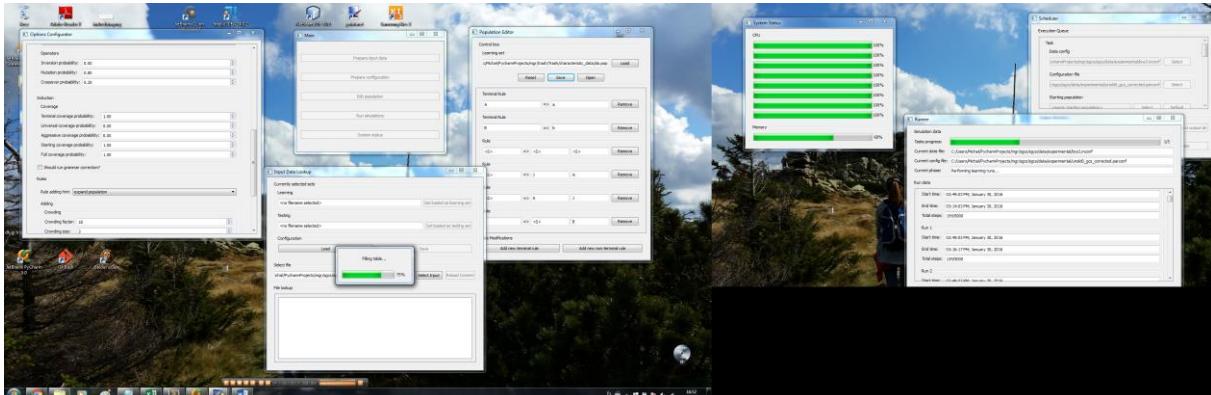
Dodatkowo do biblioteki zostały dołączone skrypty umożliwiające uruchomienie klasycznego modułu GCS/sGCS bez konieczności pisania własnego kodu. Aplikację możemy uruchomić na jeden z dwóch sposobów – z wykorzystaniem skryptu *gui_manager.py* albo *console_fetcher.py* (przypadki użycia - Rysunek 30).



Rysunek 30: Diagram przypadków użycia aplikacji

Wykorzystanie gui_manager.py

GCS oraz sGCS to algorytmy o wysokim poziomie złożoności. Posiadają kilkadziesiąt parametrów, mogą przyjmować różne dane wejściowe, zaś procesy generowania gramatyki mogą być żmudne i czasochłonne. Dlatego też w trakcie projektowania aplikacji zdecydowano się na zastosowanie wielookienkowego GUI, mając na celu ułatwienie wykorzystania wielu monitorów w pracy oraz łatwe przygotowywanie zbiorów danych do kolejnych testów, podczas gdy poprzednie są jeszcze w trakcie wykonywania. Przykładowe zdjęcie funkcjonowania aplikacji przedstawiono poniżej (Rysunek 31).



Rysunek 31: Aplikacja w trakcie użytkowania

Poniżej znajduje się omówienie kolejnych okien dostępnych w stworzonej aplikacji oraz prawidłowy sposób ich wykorzystania.

Menu główne

Po uruchomieniu skryptu pojawia się okno menu głównego. Przedstawia je Rysunek 32. Zawiera ono listę przycisków, które można uruchomić. Są to:

Prepare input data – podgląd plików ze zdaniami gramatyki napisanymi w formacie adabingo oraz przygotowywanie zestawów zbiorów uczących – testowych;

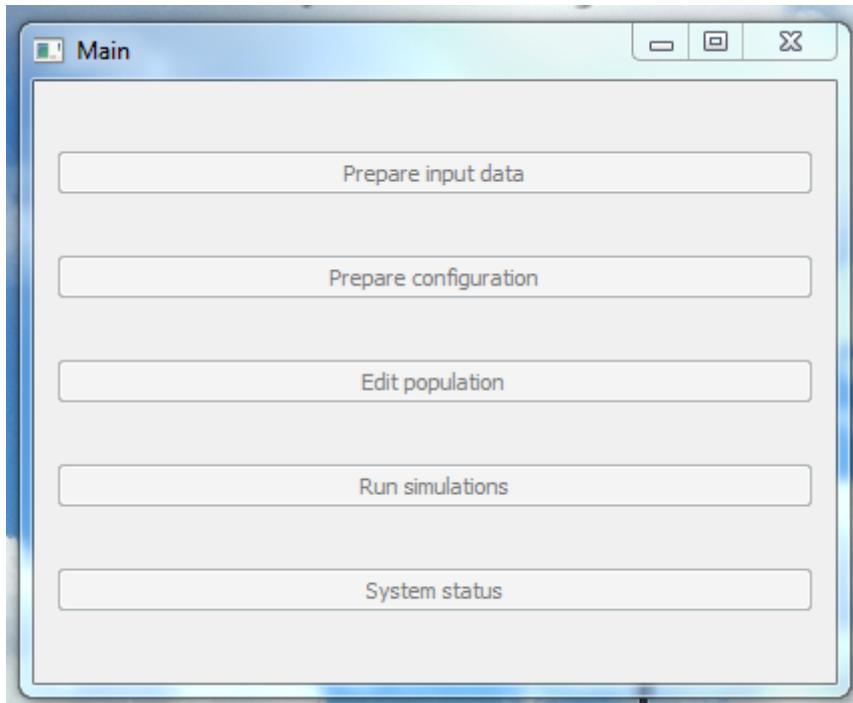
Prepare configuration – umożliwia ustalenie pożądanych parametrów algorytmu i zapisanie tych ustaleń w postaci pliku konfiguracyjnego. Dzięki temu jest możliwe późniejsze uruchamianie testów dla różnych kombinacji danych wejściowych i parametrów algorytmu bez konieczności ponownego ich wprowadzania;

Edit population – naciśnięcie tego przycisku spowoduje wyświetlenie edytora populacji. Z jego pomocą możliwy jest wygodny podgląd populacji wygenerowanej przez algorytm, czy też na przykład przygotowanie ręcznie całości lub fragmentu populacji początkowej;

Run simulations – Przycisk ten prowadzi do menu Schedulera, umożliwiającego przygotowanie zestawu testów do uruchomienia;

System status – wynikiem naciśnięcia tego przycisku jest otwarcie okna statusu systemu – okno to pokazuje obciążenie każdego z rdzeni procesora oraz ilość zajętej pamięci RAM. Narzędzie to umożliwia dokonanie oceny czy nasz algorytm w optymalny sposób wykorzystuje dostępne zasoby obliczeniowe.

Każde z wymienionych okien jest singletonem – jeżeli jedno gdzieś istnieje, to nie można w ramach tej samej aplikacji otworzyć drugiego takiego samego (co równocześnie nie przeszkadza w otwarciu okna innego typu).



Rysunek 32: Okno menu głównego

Prepare input data

Okno to (Rysunek 33) umożliwia wczytanie pliku wejściowego z gramatyką. Można tego dokonać przy pomocy przycisku Select Input. Wówczas program załadowuje wskazany plik do pamięci, podda go dokładnie takiemu samemu procesowi parsowania, jaki będzie miał miejsce przy uruchomieniu algorytmu, po czym wyświetli wszystkie występujące w nim zdania w formie czytelnej tabeli. Jeżeli plik jest duży, pojawi się okienko dialogowe informujące nas o postępie operacji oraz szacowanym postępem wypełniania tabeli. Aplikacja zakłada, że dane wejściowe są zawsze podane w formacie adabingo, aczkolwiek nie wykorzystuje wszystkich informacji oferowanych przez ten format, w wyniku czego zawartość pierwszego wiersza oraz długość zdania w kolejnych nie są weryfikowane ani wykorzystywane. Niemniej nie są to elementy opcjonalne i muszą w danym pliku wystąpić; w przeciwnym wypadku możliwe jest uzyskanie błędnie sparsowanego pliku. Można spowodować odświeżenie zawartości tabeli (na przykład w celu upewnienia się, że po dokonaniu modyfikacji zewnętrznym programem dane są nadal prawidłowo parsowane przez aplikację) naciskając przycisk Reload Content.

Po załadowaniu pliku możemy oznaczyć go jako zbiór uczący lub testowy (przyciski Set loaded as learning set oraz Set loaded as testing set). Po oznaczeniu pliku jego ścieżka bezwzględna pojawia się odpowiednio w polu Learning albo Testing. Kiedy wybierzemy oba zbiory, odblokowaniu ulega przycisk Save – możemy taką konfigurację zapisać jako plik konfiguracji *.inconf. Plik ten będzie zawierał ścieżki bezwzględne do obu podanych zbiorów (Learning oraz Testing).

The screenshot shows a software interface titled 'Form'. At the top, there's a section for 'Currently selected sets' with tabs for 'Learning' (containing '<no filename selected>'), 'Testing' (containing '<no filename selected>'), and 'Configuration' (with 'Load' and 'Save' buttons). Below this is a 'Select file' section with a text input field containing 'C:/Users/Michal/PycharmProjects/mgr/sgcs/sgcs/data/example grammatics/tomita 1.txt' and buttons for 'Select Input' and 'Reload Content'. The main area is titled 'File lookup' and contains a table with 15 rows and 8 columns. The columns are labeled 'is positive', '#0', '#1', '#2', '#3', '#4', '#5', '#6', and '#7'. The rows are numbered 3 to 17. The table entries are mostly 'a' or 'b', with row 7 having a blue highlight over the '#5' column entry 'a'.

Rysunek 33: Okno przygotowania danych wejściowych

Prepare configuration

Okno to (Rysunek 34) umożliwia przygotowanie zestawu parametrów dla algorytmu. W pierwszej kolejności warto wybrać najpierw rodzaj planowanego algorytmu. Pole Template jest odpowiedzialne za określenie czy konfiguracja jest planowana dla sGCS, czy GCS. Ze względu na fakt posiadania innych parametrów drzewo konfiguracji tych algorytmów nieco się różni i zmiana wartości tego pola może spowodować zresetowanie wartości pozostałych pól. Poniżej zostały opisane poszczególne pola.

Reset – przycisk powodujący zresetowanie pól do wartości domyślnych.

Save – przycisk umożliwiający zapisanie obecnego stanu konfiguracji.

Open – przycisk umożliwiający wczytanie wcześniej zapisanego pliku ustawień.

Template – rodzaj stosowanego algorytmu. Akceptowane wartości: sGCS i GCS (neg-sGCS to sGCS z zastosowaniem odpowiedniej funkcji przystosowania).

Should evolution be run? – odznaczenie tego pola spowoduje wyłączenie algorytmu genetycznego – gramatyki będą tworzone w oparciu wyłącznie na samym mechanizmie indukcji.

Max algorithm runs – maksymalna liczba uruchomień cyklu uczącego, jaką ma wykonać aplikacja.

Max algorithm execution time – ograniczenie umożliwiające zakończenie pojedynczego uruchomienia uczącego, jeżeli ten przekroczył podany czas. Jeżeli kroki uczenia zabraknie czasu, w ten sposób, to jest traktowany jakby przekroczył ograniczenie maksymalnej ilości kroków indukcyjnych.

Max evolution steps – maksymalna liczba kroków ewolucyjnych, jaką może wykonać aplikacja w pojedynczym uruchomieniu uczącym.

Satisfying fitness – przy jakiej wartości parametru fitness proces uczenia uzna gramatykę za wyuczoną (np. jeżeli ustawiemy tę wartość na 100%, to uruchomienie uczące zostanie uznane za sukces tylko w przypadku poprawnego sklasyfikowania wszystkich zdań).

Statistics configuration – Tutaj można wybrać rodzaj zastosowanej funkcji fitness. Obecnie w systemie są dostępne dwie funkcje fitness – Classical (przedstawiona w publikacji Unolda [1]) oraz Pasieka (przedstawiona w pracy Pasieki [2]). Wybór ten decyduje również o tym, czy będziemy dokonywali parsowania zdań negatywnych w procesie uczenia – funkcja Pasieka wykorzystuje jedynie informacje zdobyte w wyniku parsowania zdań pozytywnych, zatem w przypadku jej wybrania zdania negatywne będą ignorowane, klasyczna wykorzystuje je w procesie uczenia, więc będą one wykorzystane. W trybie GCS mamy możliwość wykorzystania wyłącznie funkcji klasycznej, w trybie sGCS obu.

Selectors – W bloku opisującym algorytm genetyczny istnieje możliwość wybrania do dziesięciu różnych selektorów. Liczba selektorów wiąże się z ilością reguł branych pod uwagę w procesie działania algorytmu genetycznego oraz wiąże się z arnością operatorów genetycznych (co zostanie dokładniej opisane w sekcji omawiającej moduł sgcs.evolution.evolution_operators). Z selektorów mamy do wyboru selekcję turniejową, losową oraz metodą ruletki.

Operators – Tutaj można nadać prawdopodobieństwo uruchomienia każdego z operatorów genetycznych. Dostępne operatory to operator krzyżowania, mutacji oraz inwersji

Induction – Ta część okna konfiguracyjnego daje możliwość ustalenia prawdopodobieństw operatorów pokrycia. Zaproponowano ujednolicony model parametrów operatorów pokrycia – wszystkie są wyrażone za pomocą prawdopodobieństwa. Więcej szczegółów na ten temat w omówieniu modułu sgcs.induction.coverage_operators.

Should run grammar corrections? – zaznaczenie tego pola spowoduje usunięcie produkcji nieosiągalnych oraz nieproduktywnych na końcu każdego etapu indukcji gramatyki (tj. tuż przed uruchomieniem algorytmu genetycznego).

Rules adding hint – W tym miejscu możemy sprecyzować ograniczenia na rozmiar zbioru – jeżeli to zrobimy, to wówczas liczba reguł nieterminalnych od chwili uruchomienia algorytmu nie ulegnie zmianie i będzie utrzymywana na stałym poziomie. Dostępne opcje to None (wówczas ta decyzja jest zależna od wartości domyślnej operatorów pokrycia), expand population (ścisł jest wówczas stosowany tylko podczas działania algorytmu ewolucyjnego, operatory pokrycia rozszerzają populację), control population size (elitism DISABLED) – operatory pokrycia dodają nowe reguły ze ściskiem, control population size (elitism ENABLED) dodatkowo uwzględnia mechanizm elityzmu. Słowo hint w przypadku tego pola jest ważne – mianowicie reguła ta zostanie nadpisana, jeżeli w jej wyniku miałoby nastąpić zwiększenie rozmiaru populacji powyżej wartości maksymalnej (patrz Max non terminal rules population size poniżej).

Crowding – w bloku tym mamy dwie kontrolki – Crowding factor oraz Crowding size. Pierwszy z nich decyduje o rozmiarze populacji ścisiku (współczynnik cf), drugi o rozmiarze podpopulacji, z której wybieramy każdego członka populacji ścisiku (współczynnik cs).

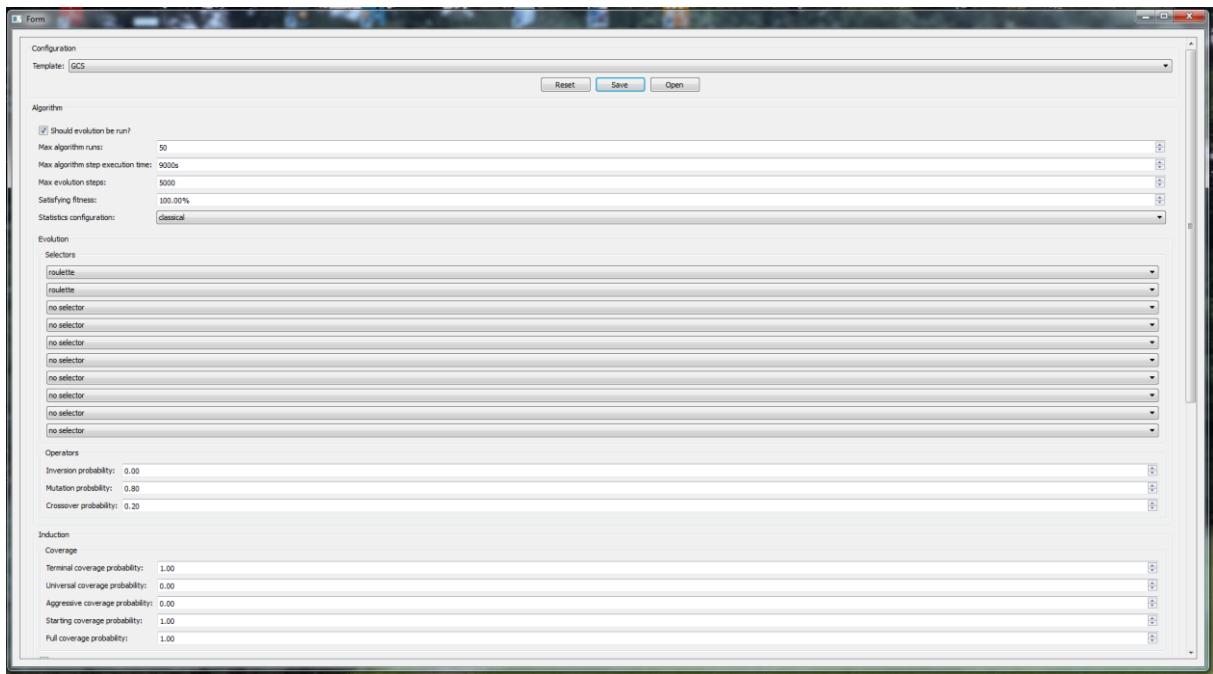
Elitism – na tym etapie możliwe jest podjęcie decyzji na temat elityzmu stosowanego w aplikacji. Możemy go włączyć lub wyłączyć (pole Is elitism used?), można tu też ustalić rozmiar elity.

Max non terminal rules population size: Maksymalny, nieprzekraczalny rozmiar nieterminalnej części populacji. Wartość ta nie może być w algorytmie nigdy przekroczena i jeżeli by to miało nastąpić, to taki precedens ma priorytet nad ustawieniami ścisiku i nawet jeżeli jest on domyślnie wyłączony, zostanie zastosowany.

Starting population size – rozmiar populacji reguł przed uruchomieniem algorytmu. Na początku działania algorytmu populacja będzie się składać z reguł dostarczonych w pliku populacji początkowej, powiększonej ewentualnie o losowe reguły w ilości brakującej do tej wartości.

Max non terminal symbols – liczba symboli nieterminalnych, która może wystąpić w algorytmie. Jest to wartość stosunkowo ważna – od rozmiaru tego parametru zależy w końcu przedział symboli, na którym dokonujemy wszelkich operacji losowych. Im mniejszy, tym częściej przy konieczności losowania symbolu będziemy trafiać na te posiadające już jakieś reguły. Z kolei większe wartości pozwolą nam na generowanie bardziej złożonych gramatyk. Ponieważ symbole specjalne (tj. symbol startowy oraz uniwersalny) posiadają specjalne oznaczenia, zatem parametr ten nie posiada górnego ograniczenia.

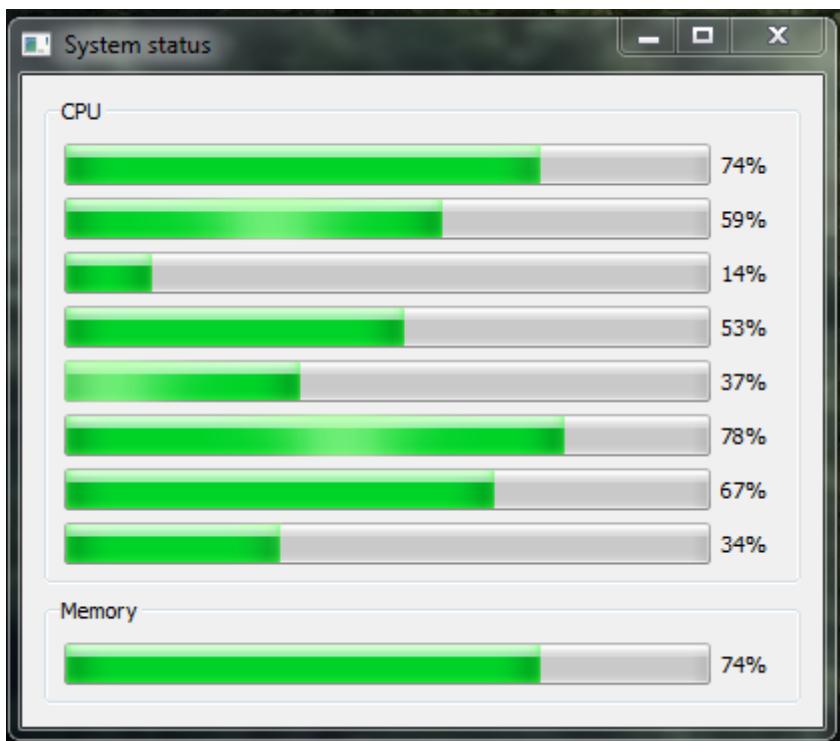
Pliki konfiguracyjne posiadają rozszerzenie *.parconf.



Rysunek 34: Edycja konfiguracji

System status

Okno to (Rysunek 35) pełni funkcję czysto diagnostyczną, jego zadaniem jest pokazywanie obecnego obciążenia wszystkich rdzeni procesora oraz zużycie pamięci. Informacja ta jest przydatna, gdyż każde uruchomienie cyklu uczącego jest niezależne od pozostałych i możliwe jest puszczenie ich w niezależnych procesach w celu maksymalnego wykorzystania maszyny, na której algorytm został uruchomiony. Dzięki takiemu podgląowi jesteśmy w stanie ocenić, czy algorytm w optymalny sposób wykorzystuje zapewnione mu zasoby. Uruchomienie niektórych zbiorów danych może spowodować wielogodzinne stuoprocentowe obciążenie procesora, z czym należy się liczyć podczas uruchamiania algorytmu. Jeżeli nasz sprzęt nie jest przygotowany na takie obciążenia (np. nie dysponuje wystarczająco sprawnym układem chłodzącym) uruchamianie algorytmu nie jest wskazane, jako że bez przedsięwzięcia dodatkowych środków ostrożności może ono doprowadzić do wyłączenia komputera, a nawet do trwałego uszkodzenia sprzętu. Biblioteka zawiera jednowątkową wersję algorytmu, aczkolwiek do interfejsu graficznego jest obecnie podpięty jedynie wariant wielowątkowy.



Rysunek 35: Okno System Status

Edit population

Z pomocą przedstawionego poniżej okna możliwe jest zarządzanie populacjami (Rysunek 36). Można przy jego pomocy otworzyć istniejący plik populacji *.pop (uzyskany na przykład w wyniku działania algorytmu), czy też utworzyć taką populację od podstaw (na przykład w celu wykorzystania jej w algorytmie jako populacji początkowej). Należy tutaj pamiętać o tym, że wewnątrz aplikacji celem przyspieszenia obliczeń wszystkie symbole gramatyki posiadają wewnętrzne identyfikatory będące liczbami całkowitymi i na potrzeby komunikacji z użytkownikiem są tłumaczone przez warstwę pośrednią (jest to jedna z kluczowych funkcjonalności modułu sgcs.datalayer). W celu prawidłowej obsługi reguł przez poniższe narzędzie niezbędne jest zachowanie kontekstu reguł – umożliwia to plik zbioru uczącego. Jeżeli w jakikolwiek sposób zmodyfikujemy nasz zbiór uczący, dodając, usuwając, modyfikując lub zmieniając kolejność zdań, obiekty populacji mogą przestać być prawidłowo interpretowane. Dlatego też, jeżeli planujemy dokonać jakichkolwiek zmian w zestawie zdań uczących, to po ich dokonaniu należy ponownie wygenerować powiązane pliki *.pop, lub też przynajmniej upewnić się, że ich zawartość po wczytaniu jest zgodna z naszymi oczekiwaniemi.

Ustalenie kontekstu

W celu ustalenia kontekstu populacji należy nacisnąć przycisk *Load* i wybrać plik, co powinno poskutkować odblokowaniem przycisków *Save* i *Open* przy właściwie sformatowanym pliku danych wejściowych, lub komunikat o błędzie w przypadku wystąpienia błędu w trakcie parsowania. Bez ustalenia kontekstu populacji nie będzie możliwe jej wczytanie lub zapisanie, gdyż pliki populacji wykorzystują wewnętrzną reprezentację reguł i podczas odczytu oraz zapisu musi nastąpić tłumaczenie danych na przyjazne człowiekowi i odwrotnie.

Modyfikacje populacji

Populację można zmodyfikować przy pomocy przycisków *Remove*, *Add new terminal rule* oraz *Add new non-terminal rule*. W przypadku reguł nieterminalnych w każdym okienku należy podać *identyfikator symbolu*. Dla reguł terminalnych po lewej stronie produkcji podajemy *identyfikator symbolu*, po prawej zaś *symbol terminalny*. *Identyfikator symbolu* rzadzi się następującymi prawami:

```

<identyfikator_symbolu> := <identyfikator_symbolu_nieterminalnego> | <identyfikator_symbolu_specjalnego>

<identyfikator_symbolu_specjalnego> := '<$>' | '<U>'

<identyfikator_symbolu_nieterminalnego> := <ciag_wielkich_liter>

<ciag_wielkich_liter> := <wielka_litera> | <wielka_litera> <ciag_wielkich_liter>

<wielka_litera> := 'A'...'Z'

<symbol_terminalny> := dowolny ciąg znaków niebędących znakami białymi
  
```

Przykłady identyfikatorów symboli nieterminalnych:

A, H, AABC

Należy zauważyć, iż w populacji może istnieć więcej symboli, niż jest znaków alfabetu. W pierwszej kolejności algorytm wykorzystuje do reprezentacji pojedynczą wielką literę alfabetu (*A, B* i tak dalej), w dalszej kolejności następuje dołożenie kolejnej litery, lub zwiększenie litery na kolejnej pozycji (tj. mamy *A, B, ..., Y, Z, AA, AB, ..., AY, AZ, BA, BB, ...*). Ludzka reprezentacja zezwala na dowolne symbole z powyższego zbioru, aczkolwiek należy pamiętać, że algorytm wykorzystuje podczas swojego działania jedynie k pierwszych symboli (gdzie k to liczba symboli terminalnych zdefiniowana w danych konfiguracyjnych **.parconf*). Przy pomocy *<S>* oznaczamy startowy symbol gramatyki, przy pomocy *<U>* symbol uniwersalny.

Zapisywanie/otwieranie populacji

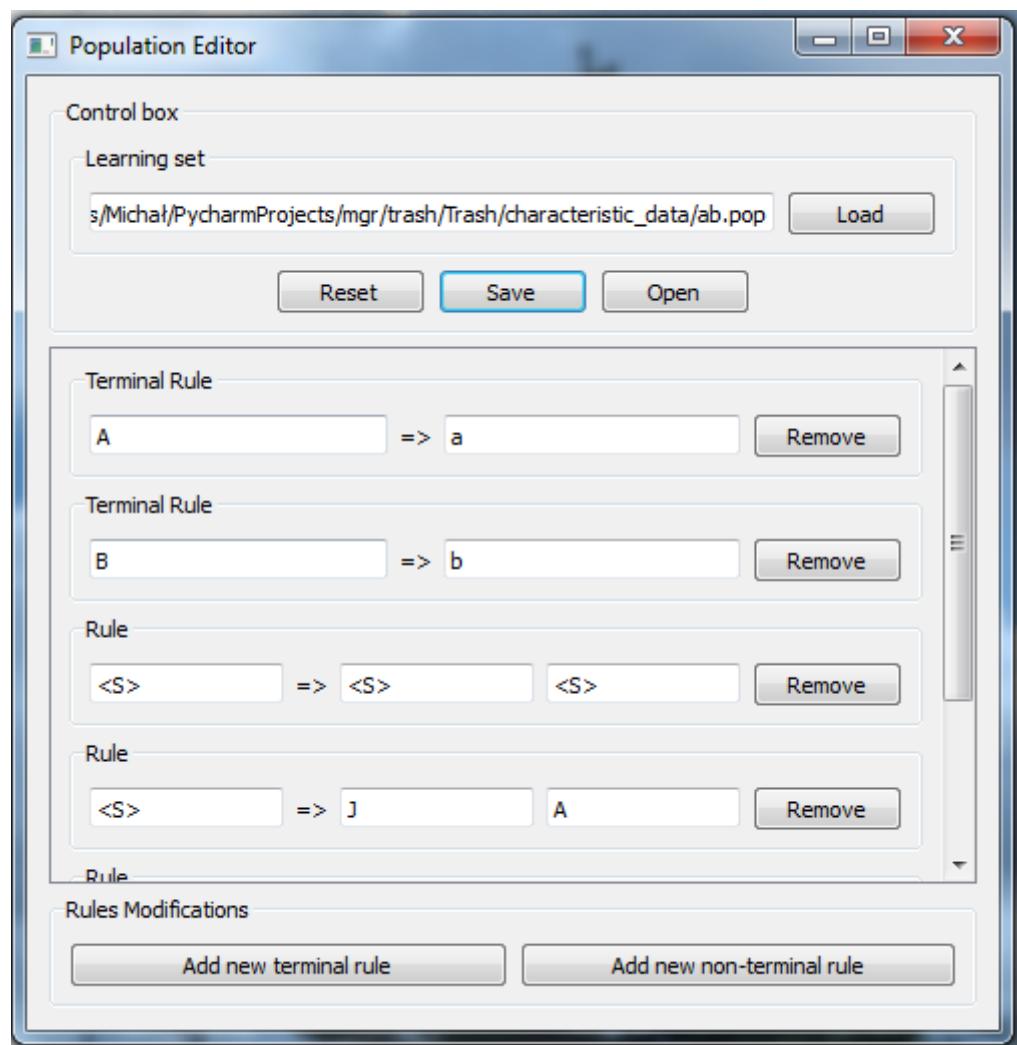
Przy pomocy przycisków *Save / Open* jest możliwe zapisanie lub wczytanie istniejącej populacji.

Ponowne generowanie plików populacji

Jeżeli dokonujemy zmian w zbiorze uczącym, możemy wygenerować populację ponownie, podejmując następujące kroki:

1. Wczytujemy stary zbiór uczący.
2. Otwieramy powiązany z nim plik populacji.
3. Wczytujemy nowy zbiór uczący.
4. Zapisujemy otwartą populację.

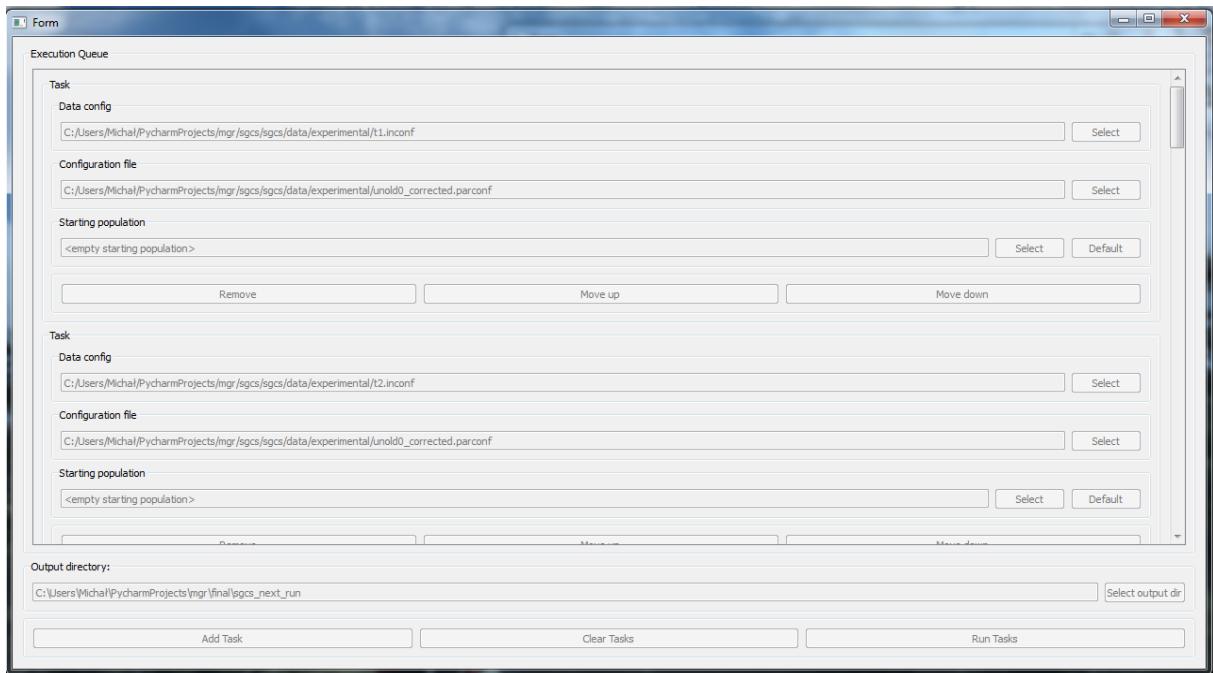
Tym sposobem jest możliwe dokonanie konwersji reguł populacji pomiędzy różnymi zbiorami uczącymi.



Rysunek 36: Edytor populacji

Scheduler

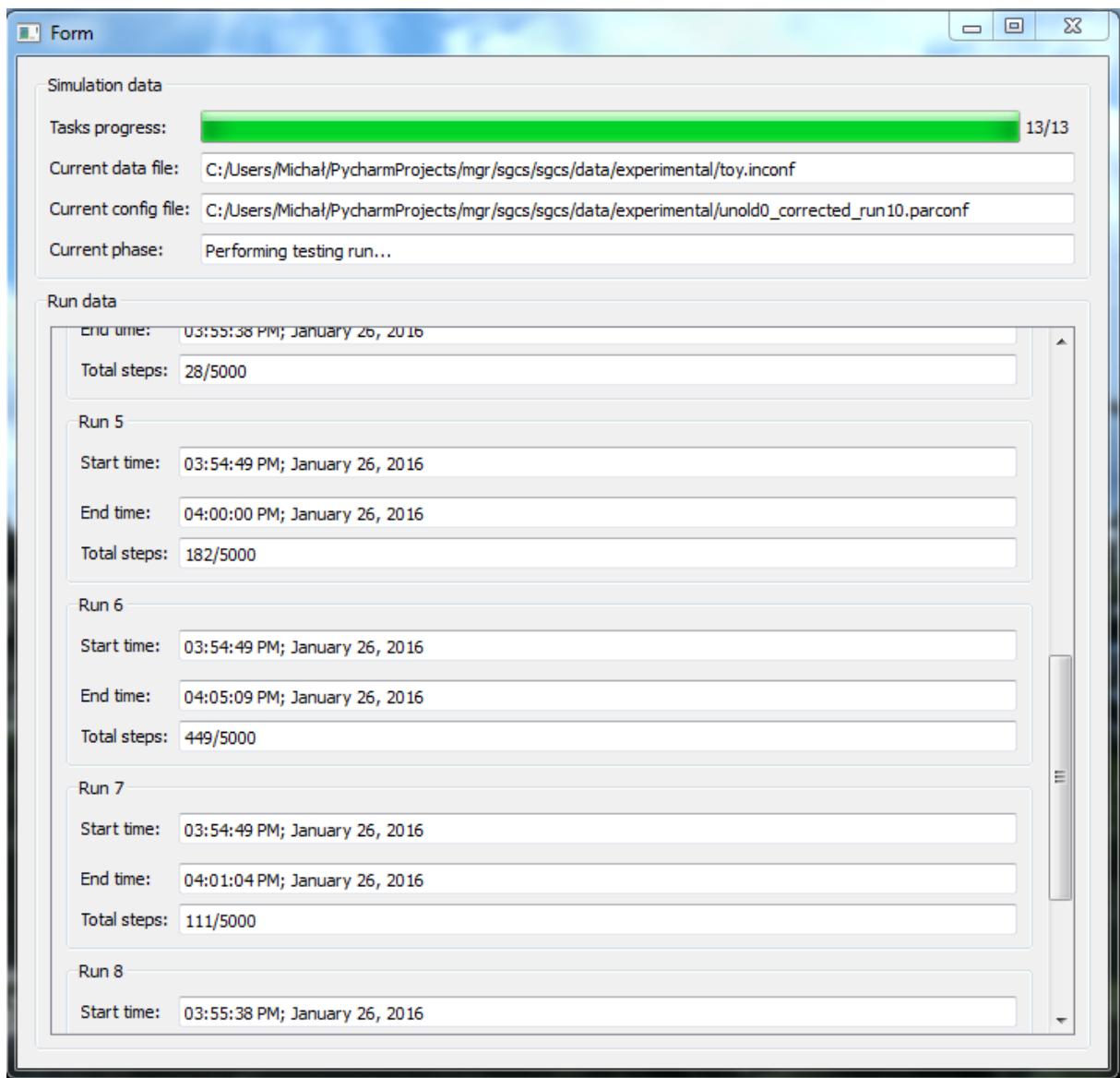
W menu tym (Rysunek 37) można przygotować zestaw badań do przeprowadzenia. Naciskając przycisk *Add Task*, dodajemy kolejne zadanie obliczeniowe. Przez zadanie rozumiemy pełne odpalenie algorytmu (cykl uczący i testowy) dla zadanego pliku danych wejściowych (skonstruowanego wcześniej przy pomocy *Prepare input data*), konfiguracji (przygotowanego przy pomocy *Prepare configuration*) i startowej populacji. Jeżeli w trakcie działania programu trzeba będzie coś zmienić, możliwe jest przerwanie wykonania, usunięcie obliczonych, poprawienie pożądanych rzeczy i wreszcie ponowne wystartowanie procesu obliczeń. Ścieżkę do folderu z danymi wynikowymi ustalamy poprzez naciśnięcie przycisku *Select output dir. Folder* ten zostanie w całości wyczyszczony na wczesnym etapie działania algorytmu, więc należy wybrać go z rozwagą w celu uniknięcia przypadkowego skasowania ważnych danych! Można również zmieniać kolejność wykonania zadań przy pomocy przycisków *Move up*, *Move down*.



Rysunek 37: Menu Scheduler

Runner

Zadaniem tego okna (Rysunek 38) jest poinformowanie użytkownika o obecnym stanie obliczeń algorytmu. W *Simulation data* można znaleźć szczegółowe informacje na temat obecnie realizowanego zadania, takie jak nr obecnie wykonywanego zadania, liczba wszystkich zadań, ścieżkę do pliku z danymi wejściowymi, konfigurację oraz obecnie wykonywany etap prac (patrz omówienie modułu sgcs.gui.runner). W obszarze Run data znajduje się osobna rubryka dla każdego cyklu uczącego, z datą początkową, końcową oraz ilością wykonanych kroków ewolucyjnych. Kiedy wszystkie zadania zostaną wykonane, na ekranie pojawi się informujący nas o tym odpowiedni komunikat. Wszystkie artefakty generowane przez algorytm są zbierane na końcu każdego zadania, a następnie zapisywane do sprecyzowanego w menu *Schedulera* katalogu wyjściowego. Każde zadanie posiada osobny katalog numerowany w kolejności zgodnej z kolejnością zadań w *Schedulerze*.



Rysunek 38: Okno przebiegu algorytmu

Wykorzystanie console_fetcher.py

Drugi skrypt uruchamiający aplikację został napisany podczas uruchamiania testów w chmurze. Jego powstanie było związane z jednej strony z trudnościami w postawieniu pełnego środowiska graficznego w chmurze oraz z drugiej strony w celu stworzenia prostego konsolowego interfejsu, który można by wykorzystywać w skryptach konsolowych. Jego składnia jest następująca:

```
console_fetcher [-p] -o output_dir [-s] [input_file config_file [starting_population]]
```

Parametr *-p* oznacza, że nie algorytm nie uruchomi się, a jedynie wyście na standardowe wyjście informację o przygotowanej konfiguracji (dzięki czemu możliwe jest wcześniejsze upewnienie się, że skrypt wywołano z właściwymi parametrami). Parametr *-o* jest obowiązkowy – po nim podajemy ścieżkę do folderu, w którym będą zapisywane artefakty. Następnie należy podać dowolną liczbę naprzemiennie występujących plików z danymi wejściowymi i plików konfiguracji. Jeżeli chcielibyśmy dodatkowo podać konfigurację początkową, należy wówczas podać również flagę *-s*, która sprawi, że algorytm nie będzie konstruował zadań na podstawie dwójki, lecz trójki parametrów. Te trzy parametry to dane wejściowe, konfiguracja i startowa populacja. Jako że całość może wydawać się dosyć skomplikowana, poniżej zostało przedstawione kilka scenariuszy uruchomienia skryptu.

```
console-fetcher -o ~/data t1.inconf sgcs.parconf t2.inconf gcs.parconf
```

Powyższe polecenie uruchomi algorytm o konfiguracji sgcs.parconf (czyli najprawdopodobniej sgcs) dla zbioru tomita 1, a następnie gcs dla tomity 2. Artefakty z działania obu algorytmów zostaną zapisane w folderze ~/data.

```
console-fetcher -p -o ~/data t1.inconf sgcs.parconf t2.inconf gcs.parconf
```

Flaga -p spowoduje, że program wyświetli tylko informację o tym co by zrobił w przypadku uruchomienia tej samej komendy bez flagi -p.

```
console-fetcher -s -o ~/data t1.inconf sgcs.parconf t1.pop t2.inconf gcs.parconf t2.pop
```

Wymienione wcześniej algorytmy rozpoczęły działanie odpowiednio z populacją początkową t1.pop oraz t2.pop.

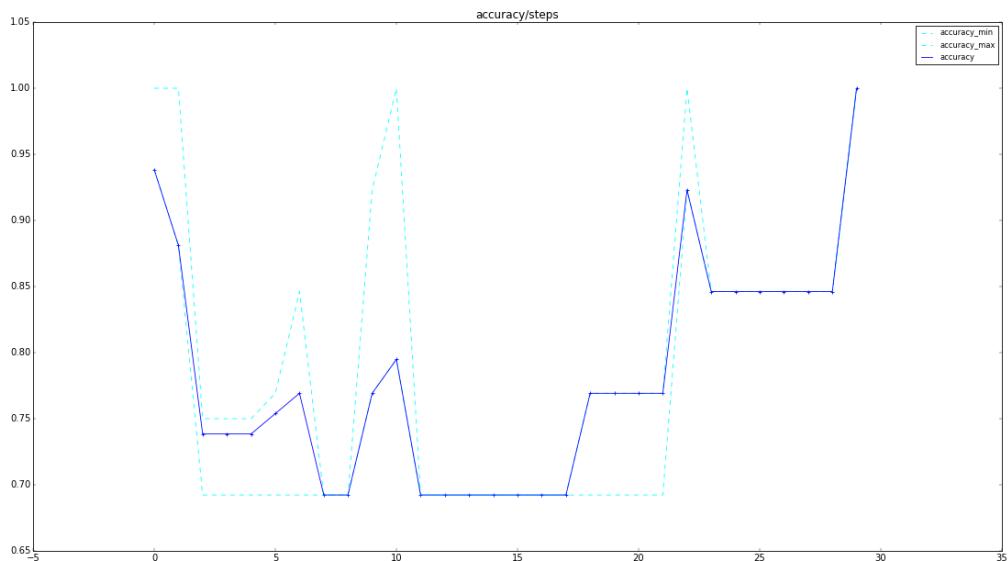
```
console-fetcher -s -o ~/data t1.inconf sgcs.parconf t1.pop t2.inconf gcs.parconf
```

Skrypt otrzymał złe argumenty (pomimo flagi -s drugie z planowanych uruchomień nie posiada jawnie podanego pliku populacji. Zostanie wyświetlone ostrzeżenie, po czym skrypt zakończy działanie.

Artefakty

Wynikiem działania algorytmu jest wygenerowanie katalogu dla każdego zdefiniowanego zadania. Poniżej została omówiona typowa zawartość takiego katalogu:

pliki *.png (*accuracy.png*, *fallout.png*, *falsediscovery.png*, *falseomission.png*, *missrate.png*, *negpredictive.png*, *precision.png*, *sensitivity.png*, *specificity.png*, *fitness.png*) – są to diagramy średnich dla wszystkich uruchomień cyklu uczącego miar macierzy pomyłek (ang. *confusion matrix*) od kroku ewolucyjnego (Rysunek 39). Można na nich znaleźć również wartość minimalną oraz maksymalną (przerywana linia).



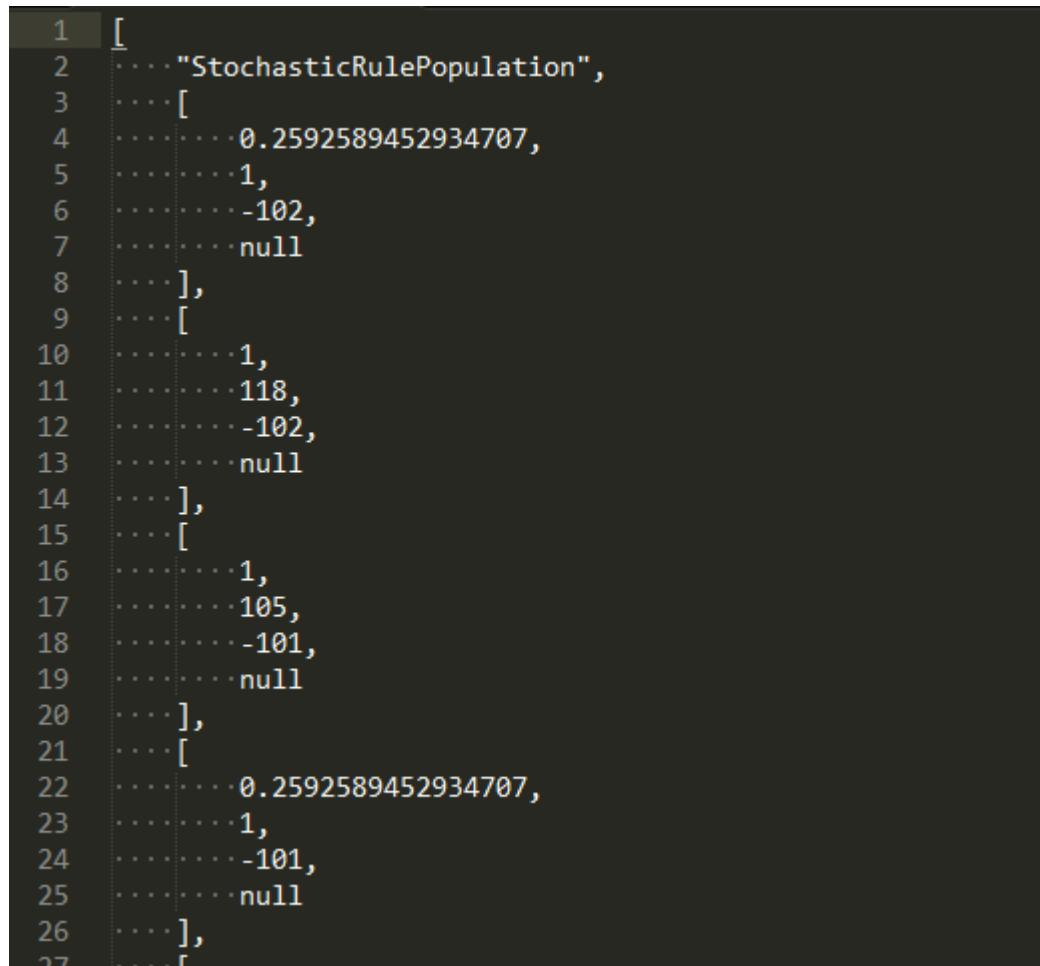
Rysunek 39: Przykładowy diagram generowany przez system

final_population.pop – W tych plikach znajduje się mało czytelny dla ludzkiego oka zapis stanu populacji wykorzystanej w cyklu testowym (Rysunek 40). Pliki są zapisywane w formacie json. Stosuje się w nich następującą reprezentację wewnętrzną:

- Całość jest listą;
- Pierwszy element listy to typ populacji reguł (stochastyczna lub nie);
- Pozostałe elementy to wszystkie reguły populacji o nieokreślonej kolejności, na każdą z nich składają się:
 - Nieznormalizowane prawdopodobieństwo (tylko w przypadku populacji stochastycznych);
 - Rodzic, lewe i prawe dziecko – rodzic zawsze jest symbolem nieterminalnym, lewe dziecko może być symbolem terminalnym lub też nie, prawe dziecko jest symbolem nieterminalnym lub wartością null.

Symbol x zostanie zapisany w postaci pojedynczej liczby całkowitej y wg następujących zasad:

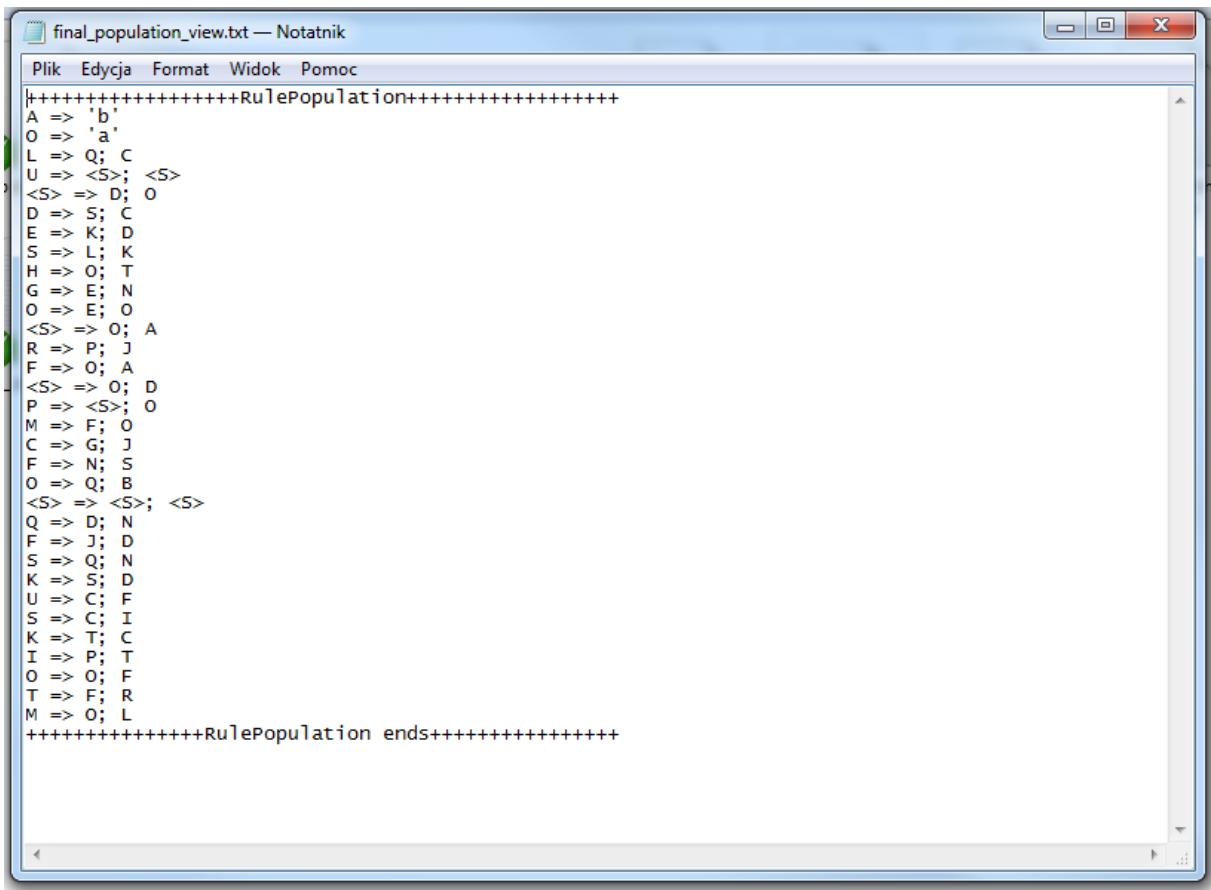
- $y = -101$ – occurrence(x) dla symboli terminalnych, gdzie occurrence(x) to wartość równa numerowi pierwszego wystąpienia symbolu terminalnego w zbiorze uczącym (np. jeżeli pierwsze zdanie w zbiorze uczącym to „a b a a c”, wówczas symbole a, b, c, otrzymają identyfikatory -101, -102, -103);
- $-100 \leq y \leq 100$, jeżeli x to symbol specjalny. W obecnej implementacji biblioteki istnieją tylko dwa symbole specjalne – startowy ($y=1$) oraz uniwersalny ($y=2$), reszta może zostać rozbudowana w przyszłości wraz z algorytmem;
- $y = 101 + \text{ord}(x)$ jeżeli x to symbol nieterminalny. Przez $\text{ord}(x)$ rozumiemy takie odwzorowanie identyfikatora symbolu x na zbiór liczb całkowitych, że A=1, B=2, itd.;



```
1 [ [ "StochasticRulePopulation", [ [ 0.2592589452934707, 1, -102, null ], [ 1, 118, -102, null ], [ 1, 105, -101, null ], [ 0.2592589452934707, 1, -101, null ] ] ] ]
```

Rysunek 40: Przykładowy plik wynikowej populacji

final_population_view.txt – Czytelny zapis populacji wykorzystanej w cyklu testowym (Rysunek 41). Dzięki temu plikowi jest możliwy szybki podgląd populacji wynikowej bez konieczności uruchamiania edytora populacji.



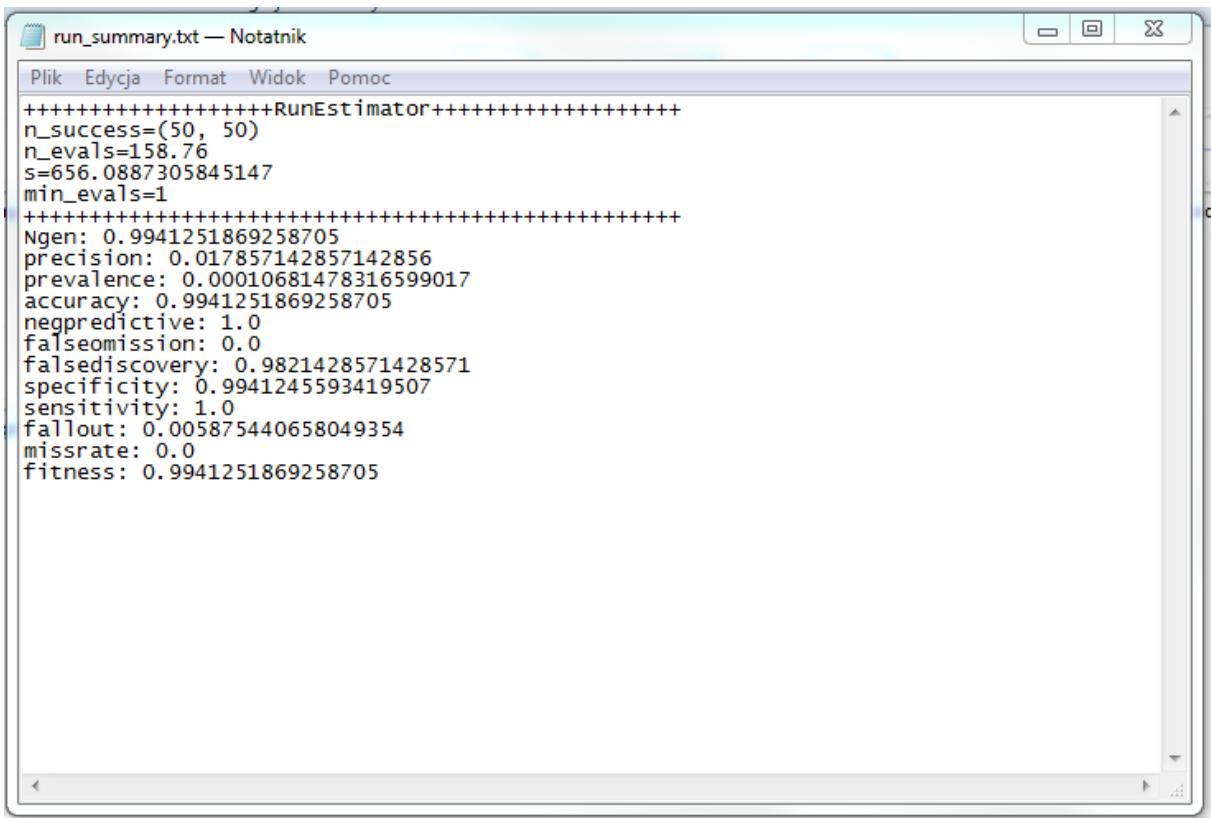
The screenshot shows a Windows Notepad window with the title bar 'final_population_view.txt — Notatnik'. The menu bar includes 'Plik', 'Edycja', 'Format', 'Widok', and 'Pomoc'. The main content area contains a text-based grammar rule definition:

```
|||||RulePopulation|||||
A => 'b'
O => 'a'
L => Q; C
U => <S>; <S>
<S> => D; O
D => S; C
E => K; D
S => L; K
H => O; T
G => E; N
O => E; O
<S> => O; A
R => P; J
F => O; A
<S> => O; D
P => <S>; O
M => F; O
C => G; J
F => N; S
O => Q; B
<S> => <S>; <S>
Q => D; N
F => J; D
S => Q; N
K => S; D
U => C; F
S => C; I
K => T; C
I => P; T
O => O; F
T => F; R
M => O; L
|||||RulePopulation ends|||||
```

Rysunek 41: Przykładowy plik podglądu końcowej populacji

`grammar_estimator.grest` – plik `.json` będący zrzutem obiektu oceny jakości procesu uczenia. Dane te to nic innego jak tekstowa reprezentacja diagramów macierzy omylek.

`run_summary.txt` – podsumowanie przebiegu algorytmu (Rysunek 42). Zawiera liczbę uruchomień uczących zakończonych sukcesem (`n_success`), średnią liczbę kroków ewolucyjnych potrzebnych do znalezienia rozwiązania (`n_evals`), odchylenie standardowe tejże (s) oraz minimum (`min_evals`). Oprócz tego znajduje się tutaj macierz omylek dla etapu testowego.



```
++++++RunEstimator+++++
n_success=(50, 50)
n_evals=158.76
s=656.0887305845147
min_evals=1
+++++
Ngen: 0.9941251869258705
precision: 0.017857142857142856
prevalence: 0.00010681478316599017
accuracy: 0.9941251869258705
negpredictive: 1.0
falseomission: 0.0
falsediscovery: 0.9821428571428571
specificity: 0.9941245593419507
sensitivity: 1.0
fallout: 0.005875440658049354
missrate: 0.0
fitness: 0.9941251869258705
```

Rysunek 42: Przykładowy plik podsumowania przebiegu algorytmu

*.inconf – plik danych wejściowych (Rysunek 43). Zawiera pełną ścieżkę do zbioru testowego oraz uczącego zapisane w formacie json.

```
1 {"testing": "C:/Users/Micha\u0142/PycharmProjects/mgr/sgcs/sgcs/data/example·gramatics/t1·opt·15·max", "learning": "C:/Users/Micha\u0142/PycharmProjects/mgr/sgcs/sgcs/data/example·gramatics/tomita·1.txt"}
```

Rysunek 43: Przykładowy plik danych wejściowych

*.parconf – plik konfiguracji (Rysunek 44). Zawiera wartości wszystkich parametrów algorytmu zapisane w formacie json.

```
1  {
2      "_node_id": "AlgorithmConfiguration",
3      "algorithm_variant": "sGCS",
4      "evolution": {
5          "_node_id": "EvolutionConfiguration",
6          "custom_rule_adding_hint": null,
7          "operators": {
8              "_node_id": "EvolutionOperatorsConfiguration",
9              "crossover": {
10                  "_node_id": "EvolutionOperatorConfiguration",
11                  "chance": 0.2
12              },
13              "inversion": {
14                  "_node_id": "EvolutionOperatorConfiguration",
15                  "chance": 0.0
16              },
17              "mutation": {
18                  "_node_id": "EvolutionOperatorConfiguration",
19                  "chance": 0.8
20              }
21          },
22          "selectors": [
23              {
24                  "_node_id": "EvolutionRouletteSelectorConfiguration",
25                  "type": 2
26              },
27              {
28                  "_node_id": "EvolutionRouletteSelectorConfiguration",
29                  "type": 2
30              }
31          ]
32      }
33  }
```

Rysunek 44: Przykładowy plik konfiguracji

Biblioteka

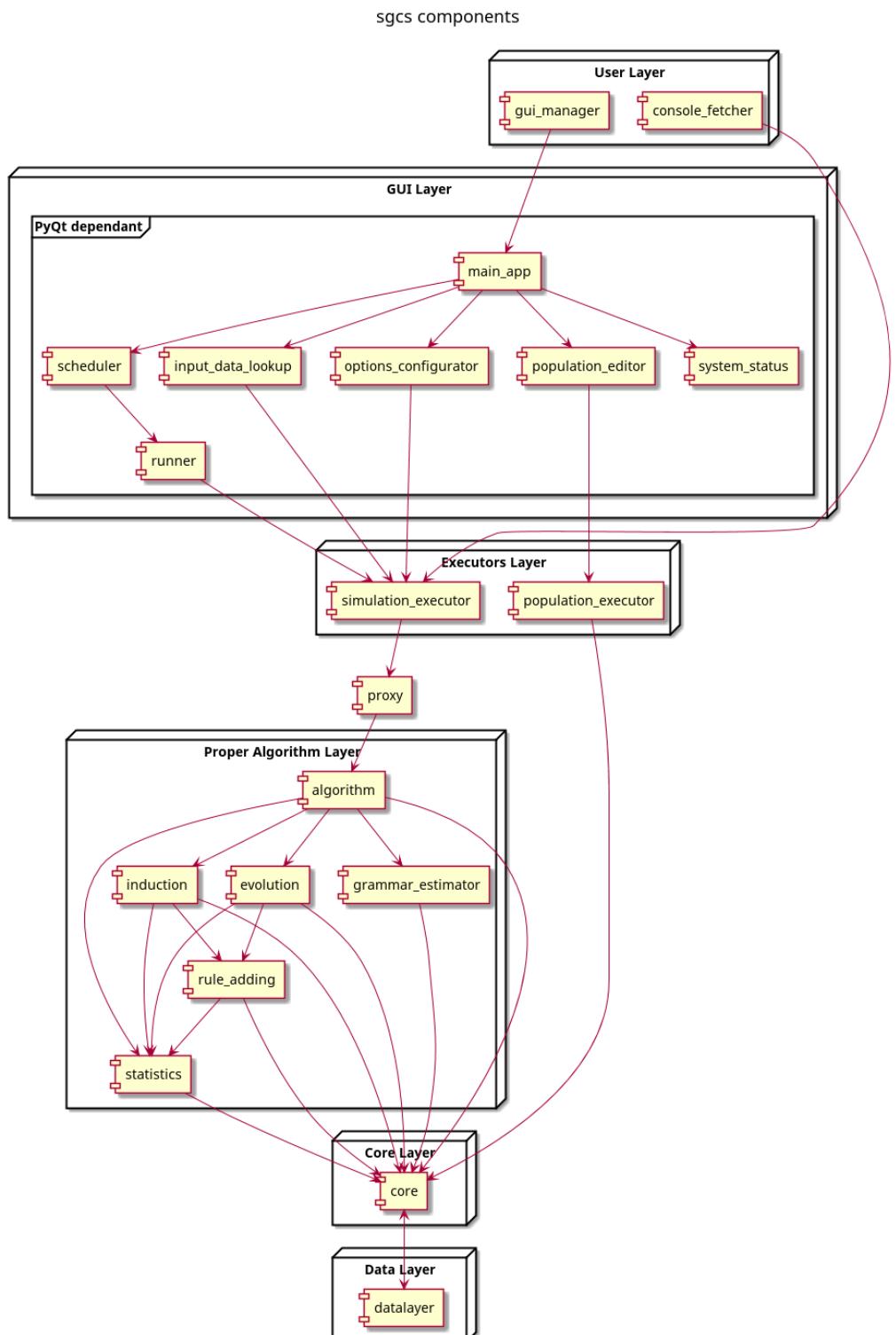
Na najwyższym poziomie projektu znajduje się plik „requirements.txt”. Zawiera on listę wszystkich bibliotek niezbędnych do dziania biblioteki, z wyjątkiem PyQt4, na którym oparto gui i który należy zainstalować osobno, w celu korzystania z elementów interfejsu graficznego.

Instalacji zależności wymienionych w pliku requirements.txt można dokonać następującym poleceniem:

```
sudo pip install -r requirements.txt
```

Dodatkowo zależnie od systemu jedna z wymienionych w tym pliku bibliotek (matplotlib) może wymagać dodatkowych bibliotek, które to zależności należy rozwiązać zgodnie z zaleceniami twórców biblioteki (na przykład na ubuntu polecenie sudo apt – get install python3 – matplotlib rozwiązuje wszystkie problemy z zależnościami).

Warstwowa budowa biblioteki



Rysunek 45: Diagram modułowy biblioteki sgcs

W całej budowie biblioteki (Rysunek 45) można wyróżnić sześć warstw logicznych:

- Warstwa Użytkownika – pierwsza warstwa, na jaką natrafia użytkownik podczas uruchamiania aplikacji. Jest to skrypt inicjujący GUI lub aplikację konsolową;
- Warstwa GUI – warstwa z którą użytkownik aplikacji również ma bezpośrednią styczność (przynajmniej w przypadku aplikacji z interfejsem graficznym, w przypadku aplikacji konsolowej interfejsem użytkownika są parametry wywołania skryptu);

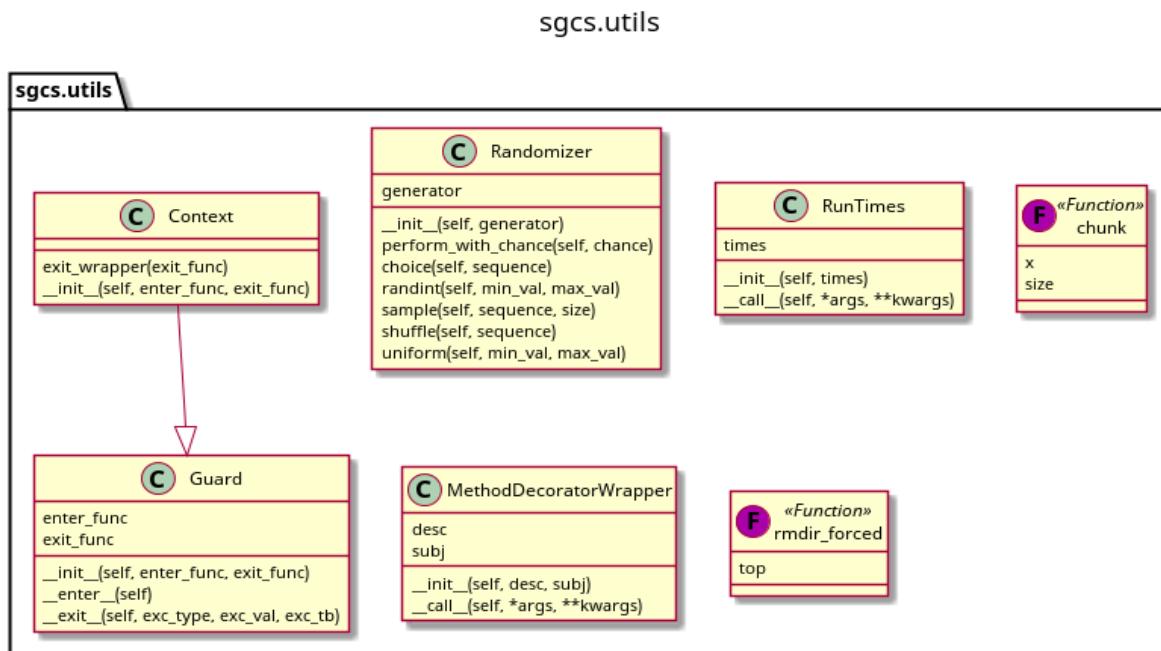
- Warstwa Wykonawców – jest to warstwa realizująca sporą część logiki biznesowej, czyli zbieranie artefaktów, edycję danych wejściowych algorytmu itp.;
- Warstwa Właściwego Algorytmu – warstwa zawierająca „czysty” algorytm GCS/sGCS/neg-sGCS, nieskażona logiką wyższych, ani niższych warstw;
- Warstwa Rdzenia – warstwa odpowiadająca za przechowywanie w pamięci najbardziej podstawowych elementów środowiska algorytmu, ściśle powiązana z warstwą niższą;
- Warstwa Danych – warstwa odpowiadająca za serializację oraz deserializację danych Warstwy Rdzenia.

Dodatkowo mamy jeszcze moduł sgcs.gui.proxy rozszerzający możliwości warstwy algorytmu w celu łatwiejszego wykorzystania go przez GUI. Moduł sgcs.core bywa też używany w niektórych klasach GUI, co trochę przelamuje zaprezentowany diagram zależności, ale chroni przed wprowadzaniem niepotrzebnych warstw abstrakcji. Oprócz tego w bibliotece występują moduły pomocnicze wykorzystywane praktycznie w każdym innym module – z tego powodu nie zostały naniesione na diagram.

Moduły pomocnicze

Biblioteka posiada zbiór kilku klas oraz funkcji wykorzystywanych praktycznie w całej aplikacji. Zostały one zebrane w jeden wygodny w wykorzystaniu moduł – sgcs.utils. Praktycznie każdy moduł wykorzystuje sporą część zawartych w nim funkcjonalności.

sgcs.utils (Rysunek 46)



Rysunek 46: Moduł sgcs.utils

Randomizer

Klasa zapewniająca warstwę abstrakcji pomiędzy całą biblioteką a standardową klasą bibliotecną random.Random. Dzięki temu jest możliwe wykorzystanie innej klasy zapewniającej dostęp do lepszej maszyny losowej (na przykład klasą wykorzystującą specyficzną architekturę maszyny, na której uruchamiamy naszą aplikację) albo łatwe uruchomienie całej aplikacji z dokładnie tym samym seedem, co może być niezwykle przydatne w przypadku rozwijania oraz debugowania algorytmu w znacznym stopniu opartego w końcu na losowości.

generator – pole przechowujący prawdziwy obiekt generatora (np. instancję klasy random.Random)

init_(self, generator) – konstruktor pobierający instancję obiektu prawdziwego generatora, dla którego klasa ma być fasadą.

perform_with_chance(self, chance) – metoda zwraca True, jeżeli liczba wylosowana z przedziału $[0,1] \leqslant \text{chance}$, False w pp.

choice(self, sequence) – metoda zwracająca losowy obiekt z sekwencji sequence (wszystkie mają równą szansę na bycie wylosowanym).

randint(self, min_val, max_val) – zwraca losową liczbę całkowitą z przedziału [min_val, max_val].

sample(self, sequence, size) – zwraca size obiektów z sekwencji sequence bez powtórzeń.

shuffle(self, sentence) – zwraca kopię sekwencji sequence, nadając jej losowy porządek.

uniform(self, min_val, max_val) – zwraca losową rzeczywistą liczbę zmiennoprzecinkową z przedziału [min_val, max_val].

RunTimes

Klasa udostępniająca cukier syntaktyczny na wywołania typu:

```
run_one_more_time = RunTimes(1)

while remainder > self.ALPHABET_SIZE or run_one_more_time():
```

chunk(x, size)

Funkcja dzieląca x na listy długości size, a następnie zwracająca taką listę list. Jeżeli x % size != 0 wówczas ostatni chunk zostanie dopełniony elementami z początku x.

Guard

Klasa pomocnicza umożliwiająca łatwe tworzenie zasobów wykorzystywanych w popularnej w języku Python bloku kontekstu with/as.

enter_func – delegat wywołyany przy wejściu w blok with/as.

exit_func – delegat wywoływany przy opuszczeniu bloku with/as.

Context

Klasa dziedzicząca po klasie Guard, zapewniającą nieco cukru syntaktycznego na funkcję wyjścia (o ile nie ma potrzeby korzystania z zaawansowanych właściwości bloku with/as), i potrzebny jest prosty strażnik, który po prostu wywoła funkcję na początku i końcu bloku, wówczas powinno korzystać się z tej klasy zamiast klasy Guard.

MethodDecoratorWrapper

Wrapper przydatny przy tworzeniu dekoratorów metod potrzebujących dostępu do instancji obiektu, którego metodę dekorują.

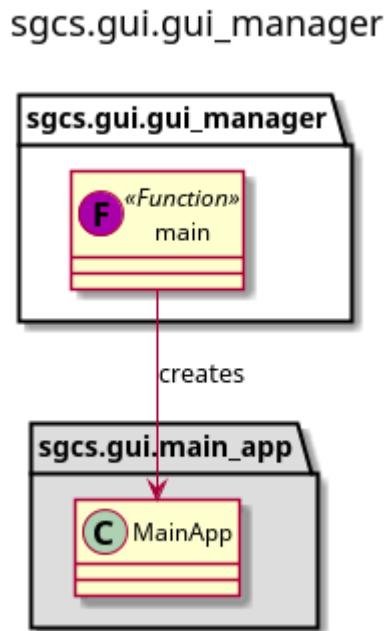
rmdir_forced(top)

Funkcja usuwająca rekurencyjnie cały folder wraz z jego zawartością.

GUI

Ponad całą biblioteką znajduje się dwa proste interfejsy umożliwiające użytkownikowi przetestowanie możliwości biblioteki zaraz po dostarczeniu i wykorzystanie w pełni jej możliwości. Jest to wspominany już w tej pracy konsolowy skrypt console_fetcher.py oraz utworzone w PyQt GUI uruchamiane przy pomocy skryptu gui_manager.py. Poniżej znajdują się diagramy zależności obu skryptów.

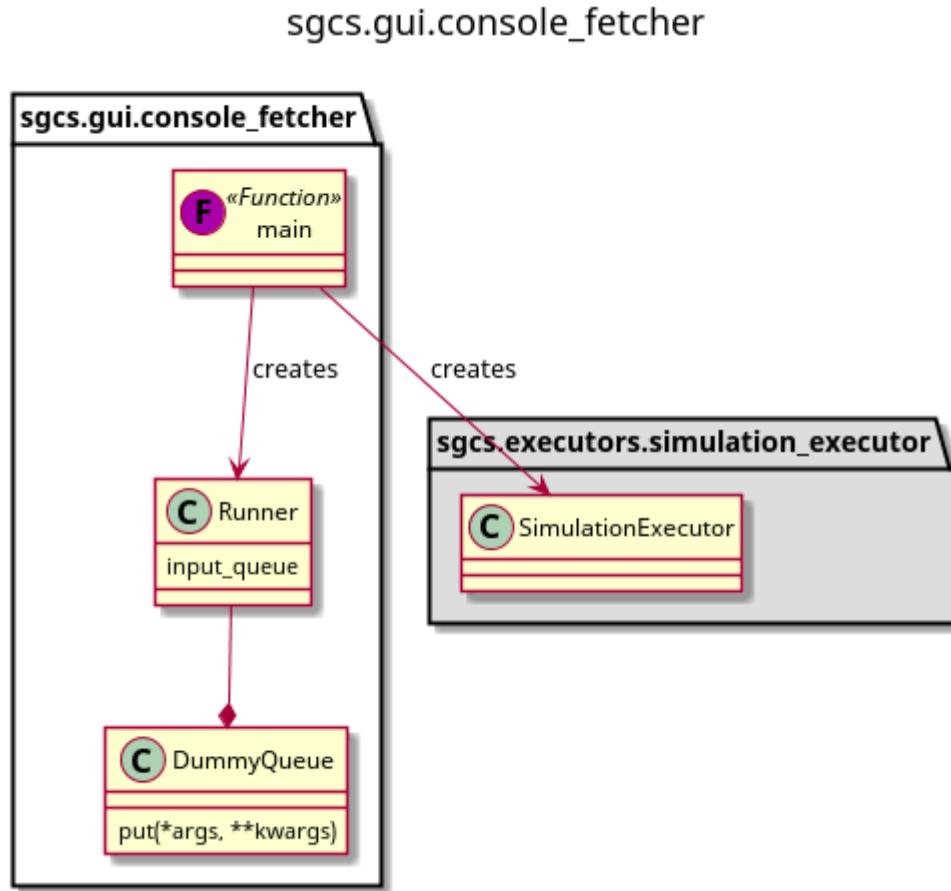
`sgcs.gui.gui_manager` (Rysunek 47)



Rysunek 47: Moduł `sgcs.gui.gui_manager`

Skrypt `gui_manager.py` powoduje inicjalizację środowiska PyQt, a następnie uruchomienie widgetu `MainApp`, odpowiadającego za obsługę opisanego wcześniej menu głównego aplikacji. Następuje oddelegowanie kontroli do `MainApp`, powraca ona do `gui_manager.py` dopiero podczas czyszczenia środowiska PyQt podczas finalizowania działania aplikacji.

sgcs.gui.console_fetcher.py (Rysunek 48)



Rysunek 48: Moduł `sgcs.gui.console_fetcher`

W przypadku skryptu `console_fetcher.py` postawiono na minimum funkcjonalności, jego uruchomienie powoduje natychmiastowe uruchomienie algorytmu. Z tego też powodu jest potrzeba istnienia uproszczonych klas `Runner` oraz `DummyQueue`.

Runner

Klasa ta służy do przechowywania oraz ustalenia kolejki zadań, które mają być wykonane przez naszą bibliotekę. W przypadku `console_fetchera` nie mamy do czynienia z interaktywną konfiguracją listy zadań (która to jest generowana automatycznie na podstawie podanej listy argumentów wywołania skryptu), toteż klasa ta jest wydmyśką i odpowiedzialność na wygenerowaniu listy zadań spoczywa na funkcji `main`. Oprócz tego zawiera ona pole `input_queue`, również wydmuszki, która w przypadku interaktywnego `gui` jest wykorzystywana przez podprocesy algorytmu do zwracania szczegółowych danych diagnostycznych, niewykorzystywanych w przypadku uruchomienia aplikacji w trybie chmury.

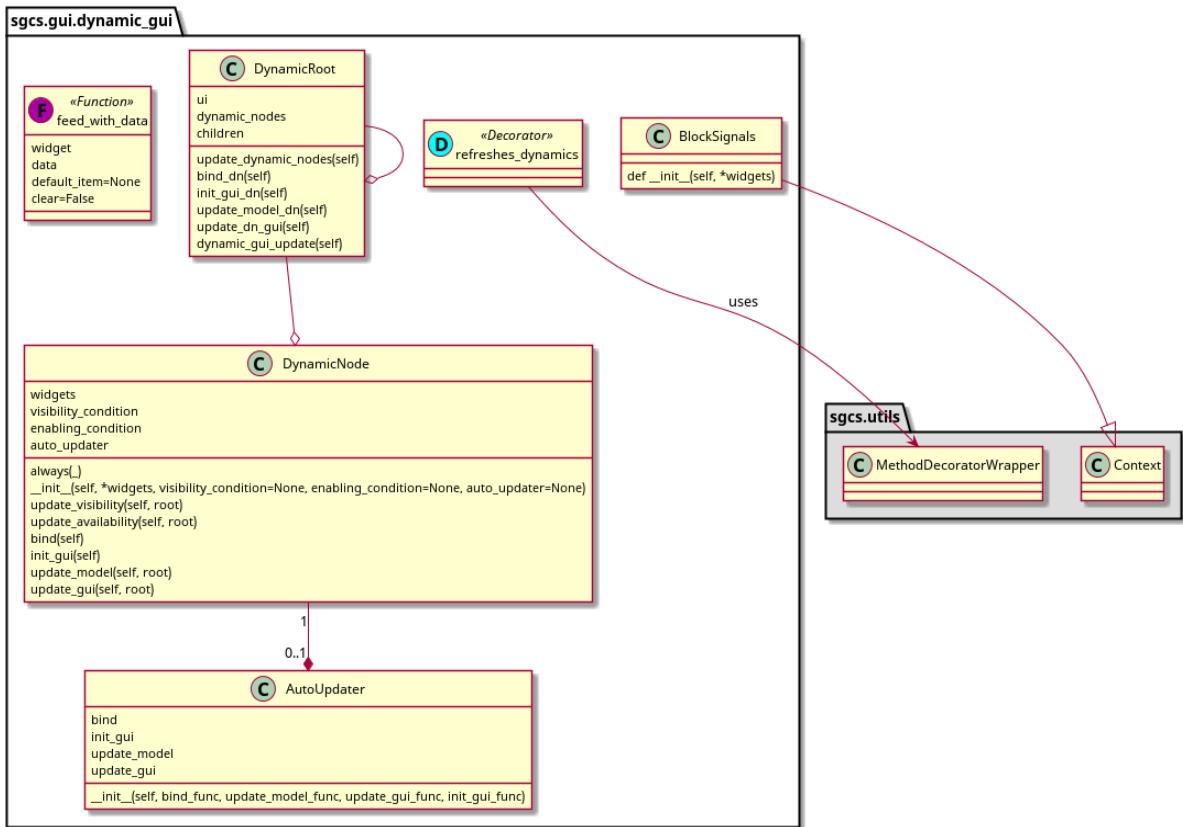
DummyQueue

Klasa ta jest wydmyśką klasy `multiprocessing.Queue`, umożliwiającej asynchroniczną wymianę informacji pomiędzy procesami aplikacji. W tej aplikacji można rozróżnić dwa systemy przesyłania danych z procesów – jako wartość zwracana poleceniem `imap` oraz właśnie przy pomocy tej kolejki. Ponieważ jednak przy pomocy kolejki wysyłamy jedynie informacje diagnostyczne, które mają większą rację bytu przy zastosowaniu `gui` (nieobecnego w przypadku uruchomienia aplikacji w trybie chmury), więc `DummyQueue` implementuje wyłącznie pustą metodę `put` (wszystkie informacje diagnostyczne są ignorowane).

sgcs.gui.dynamic_gui (Rysunek 49)

Moduł `sgcs.gui.dynamic_gui` pełni w zasadzie tą samą rolę dla wszystkich powiązanych z `gui` modułów biblioteki co `utils` dla wszystkich modułów biblioteki – jego zadaniem jest zgromadzenie całego wspólnego kodu dla wszystkich modułów `GUI`. Większość klas modułu skupia się według idei dynamicznego `gui` – umożliwiają one tworzenie prostego wiązania danych, co umożliwia programowanie z wykorzystaniem wzorca projektowego `Model-View-ViewModel`, domyślnie niewspieranego przez bibliotekę `PyQt` (istnieje wsparcie dla tegoż wzorca w przypadku kilku bardziej skomplikowanych widgetów, jednakże oddzielenie logiki od implementacji zgodnie z zaleciami MVVM nie jest możliwe w przypadku prostszych widgetów bez zastosowania dodatkowego kodu).

sgcs.gui.dynamic_gui



Rysunek 49: Moduł `sgcs.gui.dynamic_gui`

`feed_with_data(widget, data, default_item=None, clear=False)`

Funkcja ta spowoduje wypełnienie widgetu dysponującego metodami `clear(self)`, `addItems(self, items)` oraz `setCurrentIndex(self, index)` danymi. Można wykorzystać tą metodę na przykład w celu utworzenia comboboxa na podstawie listy napisów. Pole `default_item` umożliwia sprecyzowanie domyślnie zaznaczonej opcji. Podanie opcjonalnego argumentu `clear` z wartością `False` spowoduje wyczyszczenie dotychczasowej zawartości widgetu. Funkcja ta zakłada przechowywanie przez widget danych w porządku i może nie działać prawidłowo, jeżeli widget nie zapewnia kolejności pozycji.

`refreshes_dynamics`

Dekorator, którym można udekorować metodę klasy dziedziczącej po klasie `DynamicRoot`. Funkcja w ten sposób udekorowana będzie zawsze pod koniec swojego działania wołać metodę `DynamicRoot.update_dynamic_nodes(self)` w celu odświeżenia dynamicznych elementów gui.

`BlockSignals`

Obiekt kontekstu (można go wykorzystać w bloku `with/as`), który powoduje zablokowanie wysyłania sygnałów PyQt przez podane widgety (przydatne przy modyfikowaniu wartości, które mogłyby zostać potraktowane jako akcja użytkownika w przypadku gdy nie jest to pożądane zachowanie). `BlockSignals` jest bezpieczne ze względu na wielokrotne wywołanie (tj. jeżeli wewnętrz bloku A `BlockSignals` znajdzie się blok `BlockSignals` B, oba blokujące ten sam widget x, wówczas x zostanie prawidłowo odblokowany pod koniec najbardziej zewnętrznego bloku A, a nie po opuszczeniu bloku B).

`init_(self, *widgets)` – Konstruktor, wszystkie widgets zostaną zablokowane.

`DynamicRoot`

Klasa realizująca bindowanie widoków View do obiektów ViewModel.

`ui` – pole przechowujące obiekt View.

`dynamic_nodes` – pole przechowujące listę tzw. dynamicznych węzłów, umożliwiających sprecyzowanie logiki interfejsu bez konieczności modyfikowania jego samego.

children – jest możliwe podczepianie jednego obiektu typu DynamicRoot do drugiego jako dziecko w celu utworzenia struktury drzewiastej. Dzięki temu żądając odświeżenia gui lub modelu korzenia takiej struktury w efekcie nastąpi odświeżenie jej w całości. Ten mechanizm działa tylko w przypadku struktur drzewiastych i nie jest zabezpieczony przed wystąpieniem cyklu!

update_dynamic_nodes(self) – powoduje odświeżenie widoczności oraz dostępności gui na podstawie wszystkich dynamicznych węzłów.

bind_dn(self) – powoduje podpięcie logiki wystawionej przez metody bind(self) wszystkich dynamicznych węzłów.

init_gui_dn(self) – powoduje inicjację GUI poprzez wywołanie wszystkich metod init_gui(self) dynamicznych węzłów.

update_model_dn(self) – powoduje odświeżenie modelu na podstawie obecnego stanu gui (czyli dokładnie wywołanie metod update_model() wszystkich dynamicznych węzłów).

update_dn_gui(self) – powoduje odświeżenie gui na podstawie obecnego stanu modelu, blokując przy tym wszystkie sygnały PyQt na czas procesu.

dynamic_gui_update(self) – powoduje odświeżenie widoczności, dostępności oraz stanu GUI na podstawie obecnego modelu.

DynamicNode

Klasa umożliwiająca zdefiniowanie części lub całości logiki obsługi widgetu. Może ona przejąć kontrolę przede wszystkim nad widocznością widgetu, jego dostępnością oraz powiązać go z konkretnym viewmodelem. Jeden obiekt DynamicNode reprezentuje jakiś fragment logiki, w efekcie są one powiązane z widgetami relacją wiele do wielu – jeden dynamiczny węzeł może opisywać zachowanie wielu widgetów, ale równocześnie kilka innych węzłów może skupiać się na pozostałych aspektach jednego z tych widgetów itd.

widgets – pole precyzujące widgety, których widoczność i dostępność będzie kontrolowana przez ten obiekt.

visibility_condition – delegat określający warunki widoczności widgetów.

enabling_condition – delegat określający warunki dostępności widgetów.

auto_updater – instancja obiektu AutoUpdater, zajmującego się powiązaniem modelu z widokiem.

always_() - pomocniczy delegat zwracający zawsze True.

__init__(self, *widgets, visibility_condition=None, enabling_condition=None, auto_updater=None) – konstruktor. Jeżeli nie poda się któregoś z parametrów opcjonalnych, wówczas ten aspekt zostanie zignorowany przez ten konkretny węzeł (na przykład niesprecyzowanie delegata visibility_condition sprawi, że węzeł ten nie będzie zarządzać widocznością widgetów).

AutoUpdater

Klasa grupująca funkcje bind, init_gui, update_model oraz update_gui w jednym miejscu.

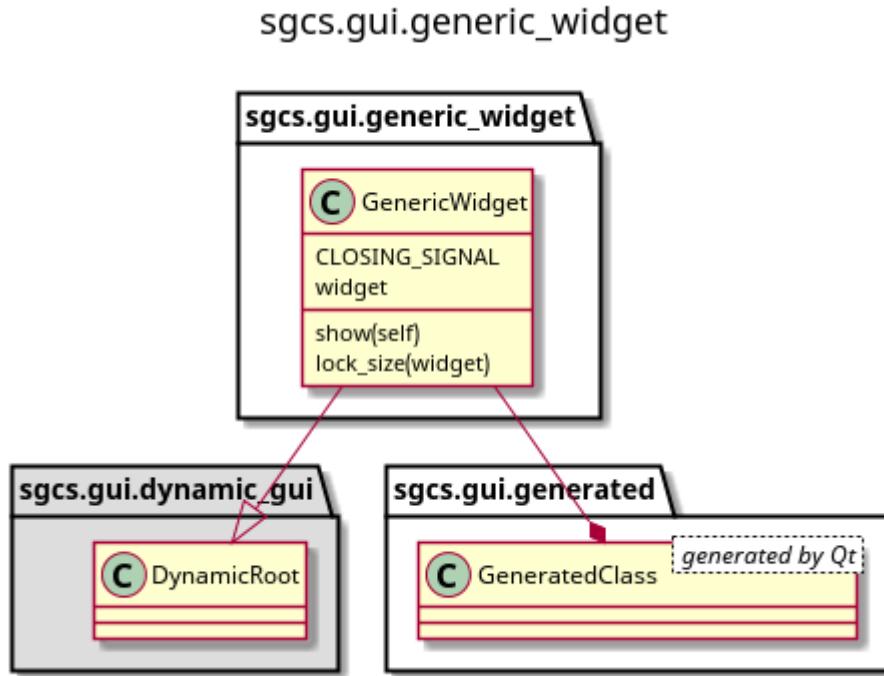
bind – delegat zawierający logikę, która powinna zostać wykonana na późnym etapie tworzenia gui (np. rejestracja na sygnały).

init_gui – delegat zawierający logikę inicjującą GUI przy tworzeniu interfejsu po raz pierwszy.

update_model – delegat odświeżający model na podstawie obecnego stanu GUI.

update_gui – delegat odświeżający GUI na podstawie obecnego stanu modelu.

`sgcs.gui.generic_widget` (Rysunek 50)



Rysunek 50: Moduł `sgcs.gui.generic_widget`

Klasa `GenericWidget` stanowi rozszerzenie klasy `DynamicRoot` o kilka przydatnych mechanizmów. Po pierwsze stanowi most pomost pomiędzy klasami wygenerowanymi przy pomocy QtDesignera a zdefiniowanymi przez użytkownika widokami. Po drugie nadpisuje obsługę zamknięcia okna w taki sposób, żeby został wysłany sygnał informujący o tym wydarzeniu (brakujące zachowanie w obecnej wersji PyQt). Po trzecie umożliwia łatwą blokadę rozmiarów okna (kolejna czynność, której nie da się wykonać w PyQt za pomocą jednej metody).

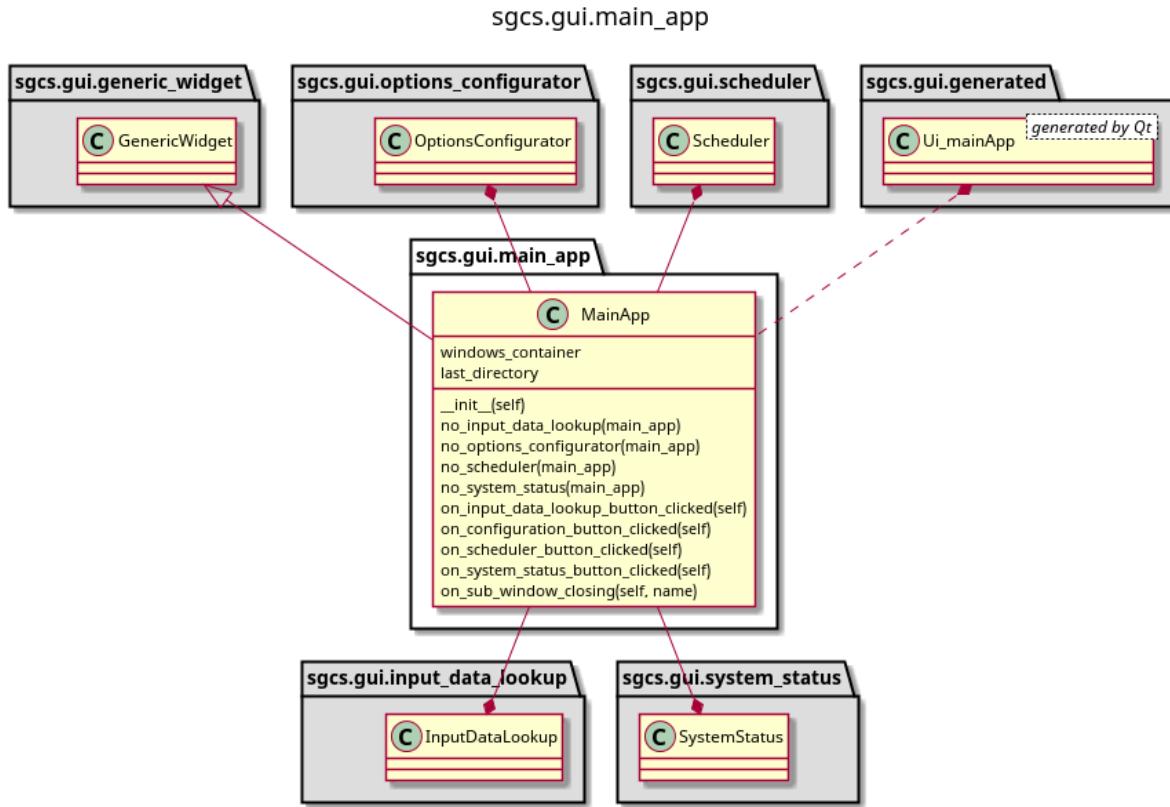
CLOSING SIGNAL – identyfikator sygnału wysyłanego podczas zamykania okna. Jedyne argument sygnału to nazwa klasy widgeta, który jest właśnie zamykany.

widget – obiekt `QtGui.QWidget`.

show(self) – metoda umożliwiająca pokazanie widgetu

lock_size(widget) – statyczna metoda umożliwiająca zablokowanie możliwości zmiany rozmiarów okna.

`sgcs_main_app` (Rysunek 51)



Rysunek 51: Moduł `sgcs.gui.main_app`

Klasa odpowiadająca za wyświetlenie i obsługę menu głównego. Wszystkie otwierane przez nią okna są singletonami – w danej chwili może być otwarte wyłącznie jedno okno danego typu, nie możemy na przykład uruchomić dwóch okien statusu systemu równocześnie. Jest to związane z wykorzystaniem specyficznych zasobów systemowych przez większość okien, co sprawia, że utworzenie ich kilku instancji mogłoby prowadzić do dziwnych problemów (i tylko w kilku przypadkach miałoby jakiś sens).

windows_container – fabryka okien zajmująca się kontrolą ich czasu życia

last_directory – zmienna przechowująca ostatni otwarty w jakiekolwiek części aplikacji folder. Ułatwia to powrót do ostatnio używanych danych (nie musimy za każdym razem szukać tego samego folderu).

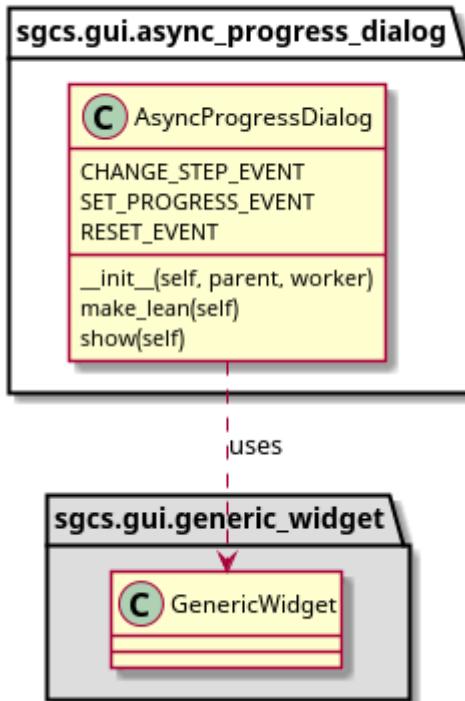
no_input_data_lookup/no_options_configurator/no_scheduler/no_system_status – funkcje sprawdzające czy dane okno zostało już stworzone.

on_input_data_lookup_button_clicked/on_configuration_button_clicked/on_scheduler_button_clicked/on_system_status_button_clicked – funkcje obsługujące otwieranie okien poszczególnych aplikacji.

on_sub_window_closing – funkcja obsługująca proces zamknięcia okna aplikacji, nasłuchiwanie na `GenericWidget.CLOSING_SIGNAL`.

`sgcs.gui.async_progress_dialog` (Rysunek 52)

`sgcs.gui.async_progress_dialog`



Rysunek 52: Moduł `sgcs.gui.async_progress_dialog`

Klasa umożliwiająca wyświetlenie maksymalnie odchudzonego dialogu z komunikatem oraz paskiem postępu. Po wyświetleniu go można sterować jego zachowaniem przy pomocy odpowiednich sygnałów.

CHANGE_STEP_EVENT – identyfikator sygnału wysyłanego w celu zmiany tekstu obecnego komunikatu.

SET_PROGRESS_EVENT – identyfikator sygnału wysyłanego w celu zmiany wartości obecnego postępu.

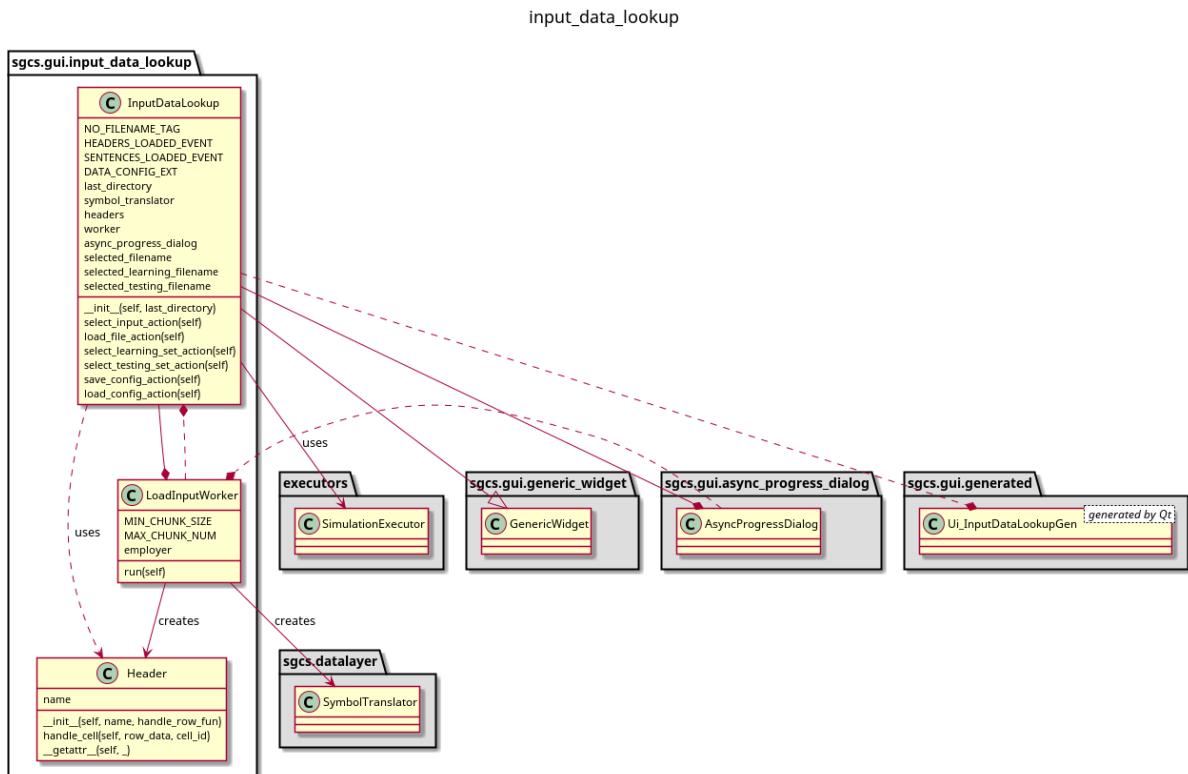
RESET_EVENT – identyfikator sygnału powodującego schowanie/ponowne wyświetlenie okna dialogowego.

__init__(self, parent, worker) – jako parent należy podać okno – rodzica komunikatu. Worker jest obiektem, który będzie emitować sygnały (na przykład osobny wątek).

make_lean(self) - „odchudza” komunikat.

show(self) – powoduje wyświetlenie komunikatu.

sgcs.gui.input_data_lookup (Rysunek 53)



Rysunek 53: Moduł `sgcs.gui.inout_data_lookup`

Moduł ten zawiera klasy niezbędne do obsługi podglądu plików z danymi wejściowymi do gramatyk oraz tworzenie plików konfiguracyjnych danych wejściowych *inconf.

`InputDataLookup`

Główna klasa odpowiedzialna za obsługę menu „Prepare input data”.

NO_FILENAME_TAG – napis widoczny do czasu wyboru pliku.

HEADERS_LOADED_EVENT – identyfikator sygnału wysyłanego, gdy już znamy wszystkie nagłówki kolumn tabeli.

SENTENCES_LOADED_EVENT – identyfikator sygnału wysyłanego, gdy tabela została załadowana i jest już gotowa do użytku.

DATA_CONFIG_EXT – stała przechowująca rozszerzenie plików konfiguracyjnych danych wejściowych („.inconf”).

last_directory – ostatni otwarty folder.

symbol_translator – obiekt wykorzystywany do wczytania danych z dysku (obiekt typu `SymbolTranslator`).

headers – nagłówki tabeli.

worker – uchwyt wątku, któremu zleca się wczytanie pliku.

async_progress_dialog – uchwyt okna z postępem wczytywania wyświetlanego w trakcie ładowania danych wejściowych.

selected_filename – wybrany plik danych wejściowych.

selected_learning_filename – ścieżka do wybranego zestawu uczącego.

selected_testing_filename – ścieżka do wybranego zestawu testowego.

*_action(self) – funkcje obsługujące naciśnięcie odpowiedniego przycisku.

`LoadInputWorker`

Wątek zajmujący się wczytaniem danych do tabeli. Wbrew pozorom nie jest to zadanie aż tak trywialne, jak mogłyby się wydawać – mimo wykonywania najbardziej czasochłonnej operacji, czyli ładowania danych z dysku do pamięci w osobnym wątku to w przypadku dużych

danych wejściowych samo dodawanie danych do widgetu tabeli może spowodować zawieszenie się głównego okna. Dlatego też worker ten dokonuje podziału danych wejściowych na chunki odpowiedniego rozmiaru, po czym wczytuje je jeden po drugim. Chunk ma rozmiar co najmniej równy MIN_CHUNK_SIZE i równocześnie taki, żeby liczba chunków nie przekroczyła MAX_CHUNK_NUM. Worker na czas swojej pracy powoduje również wyświetlenie komunikatu ze statusem prac.

MIN_CHUNK_SIZE – minimalny rozmiar chunka.

MAX_CHUNK_NUM – maksymalna liczba chunków, jakie mogą powstać.

employer – uchwyt zleceniodawcy zadania, czyli InputDataLookup.

run(self) – główna metoda workera, w celu wykonania asynchronicznego nie należy wołać jej bezpośrednio, lecz zamiast tego QtCore.Qthread.start.

Header

Klasa zajmująca się obsługą prawidłowego wyświetlania zawartości danej kolumny tabeli.

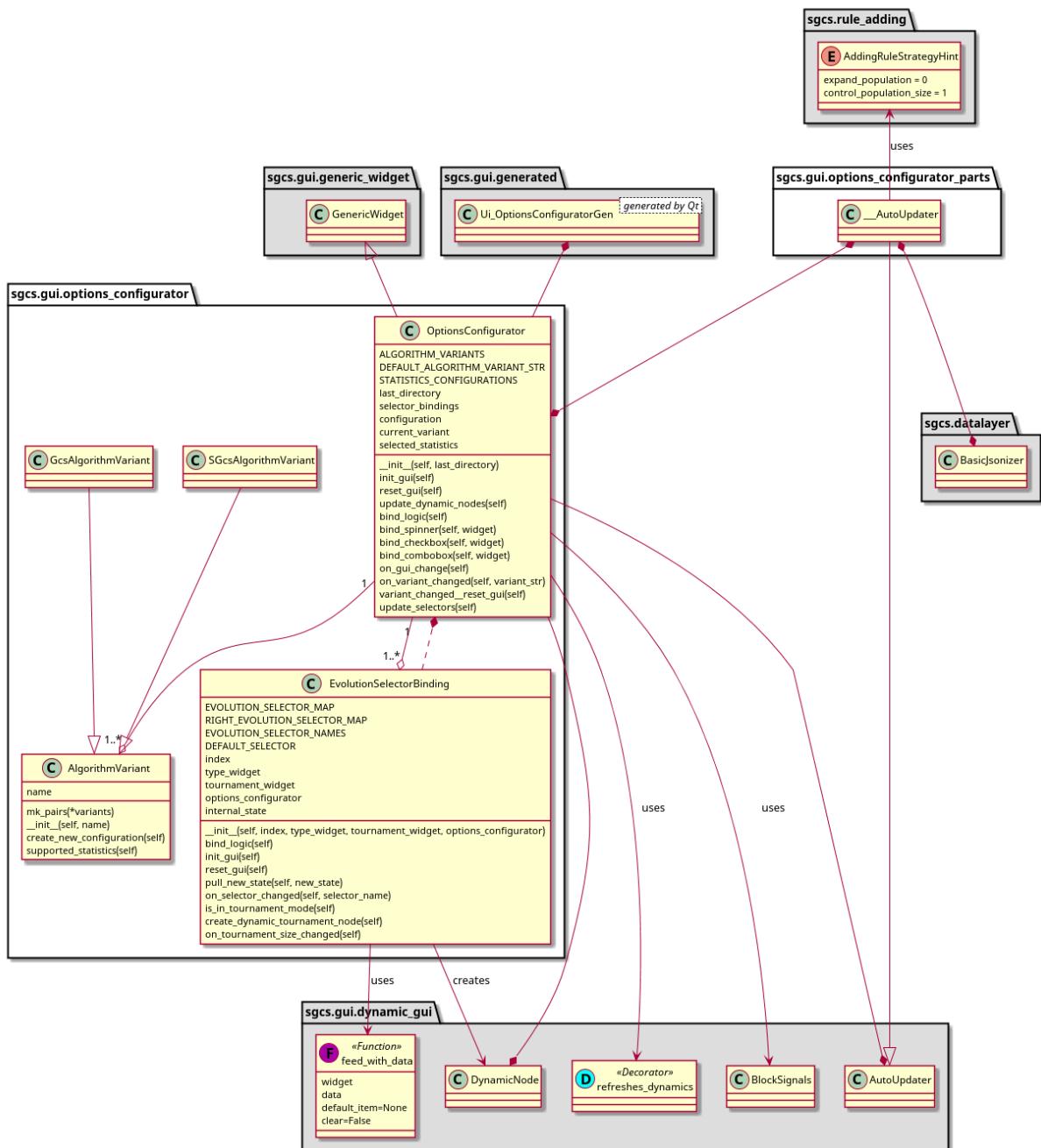
name – nagłówek danej kolumny.

__init__(self, name, handle_row_fun) - konstruktor, należy przekazać mu nagłówek kolumny name oraz delegata zdolnego do odpowiedniego wyświetlania zawartości danej kolumny.

handle_cell(self, row_data, cell_id) – funkcja zajmująca się prawidłowym wypełnieniem danej komórki tabeli.

sgcs.options_configurator i sgcs.options_configurator_parts (Rysunek 54)

sgcs.gui.options_configurator



Widok „Prepare configuration” jest najbardziej skomplikowaną częścią GUI, co widać, chociażby po jego diagramie klas. Diagram ten byłby

Rysunek 54: Moduły `sgcs.options_configurator` i `sgcs.options_configurator_parts`

jeszcze bardziej rozbudowany, gdyby nie zwijać wszystkich składowych obiektów `AutoUpdater` z przestrzeni nazw `sgcs.gui.options_configurator_parts` w jedną klasę `__AutoUpdater`. Jest to spowodowane niezwykłym rozbudowaniem tego menu (ma za zadanie obsłużyć dobrą około 30 parametrów algorytmu), dynamiczną budową (nie wszystkie pola są zawsze dostępne, istnieją zależności pomiędzy niektórymi parametrami) oraz dosyć wcześnie napisaniem go, kiedy jeszcze struktura GUI nie była aż tak uporządkowana.

AlgorithmVariant

Abstrakcyjna klasa stosowana do rozróżnienia pomiędzy dostępnymi algorytmami na poziomie GUI. Obecnie posiada dwie konkretne implementacje: `GcsAlgorithmVariant` oraz `SgcsAlgorithmVariant`.

name – wyświetlna nazwa algorytmu

mk_pairs(*variants) – statyczna metoda tworząca mapę `variant.name: variant`.

create_new_configuration – abstrakcyjna metoda, której wywołanie powinno spowodować utworzenie domyślnego obiektu konfiguracji dla danego wariantu algorytmu.

supported_statistics – lista nazw funkcji fitness, które są wspierane przez dany algorytm. Od doboru funkcji fitness (wykorzystującej lub nie zdania negatywne) zależy również czy proces uczenia będzie przebiegał również na zdaniach negatywnych.

AutUpdater

Grupa klas dziedziczących po klasie AutoUpdater umiejscowionych w przestrzeni nazw sgcs.gui.options_configurator_parts, jak sama nazwa przestrzeni wskazuje, są klasami przechowującymi fragmenty logiki menu „Prerare Configuration”. W przestrzeni tej możemy odnaleźć następujące klasy:

- ClassicalStatisticsAutoUpdater – klasa obsługująca część menu odpowiedzialną za klasyczną funkcję płodności;
- CrowdingAutoUpdater – klasa obsługująca część menu odpowiedzialną za funkcjonowanie ścisku;
- ElitismAutoUpdater – klasa obsługująca część menu odpowiedzialną za funkcjonowanie elityzmu;
- EvolutionAutoUpdater – klasa obsługująca część menu odpowiedzialną za parametry operatorów ewolucyjnych;
- InductionAutoUpdater – klasa obsługująca część menu odpowiedzialną za parametry operatorów pokrycia oraz korekcję gramatyki;
- MainBoxAutoUpdater – klasa obsługująca część menu odpowiedzialną za wczytywanie i zapisywanie konfiguracji oraz wybór wariantu algorytmu;
- RootAutoUpdater – klasa obsługująca część menu odpowiedzialną za ogólne parametry algorytmu (liczba uruchomień cyklu uczącego, satysfakcjonujący poziom przystosowania, rodzaj funkcji przystosowania);
- RulesAutoUpdater – klasa obsługująca część menu odpowiedzialną za ogólny stan populacji reguł (maksymalny rozmiar populacji, liczba symboli nieterminalnych itd.).

Każda z tych klas pilnuje, aby parametry, za które jest odpowiedzialna, były zgodne z tym, co jest widoczne w danym momencie w menu. Dbają również o widoczność i dostępność części menu (na przykład chowając część dotyczącą parametryzacji klasycznej funkcji fitness, jeżeli zdecydujemy się na inną niż klasyczna funkcję fitness).

EvolutionSelectorBinding

Jest to klasa zapewniająca obsługę części menu związanego z doborem selektorów algorytmu genetycznego. Jest to jeden z protoplastów architektury DynamicRoot, który nie został ze względu na swoją skomplikowaną logikę oraz brak czasu w pełni przetłumaczony na nowy mechanizm. Zadaniem tej klasy jest obsługa wyboru selektorów algorytmu genetycznego oraz udostępnienie pól dodatkowych parametrów, jeżeli dany selektor takie parametry posiada.

EVOLUTION_SELECTOR_MAP – mapa odwzorowująca nazwę na węzeł konfiguracyjny selektora.

RIGHT_EVOLUTION_SELECTOR_MAP – mapa odwzorowująca węzeł konfiguracyjny węzeł selektora na nazwę.

EVOLUTION_SELECTOR_NAMES – lista kluczy EVOLUTION_SELECTOR_MAP.

DEFAULT_SELECTOR – nazwa selektora domyślnego.

index – nr selektora, za który odpowiada instancja.

type_widget – uchwyt do pola combobox odpowiedzialnego za wybór typu selekcji.

tournament_widget – uchwyt do pola spinner odpowiedzialnego za określenie rozmiaru turnieju.

options_configurator – uchwyt do głównej klasy okna konfiguracji.

internal_state – przechowywana węzeł konfiguracyjny selektora.

bind_logic(self) – metoda odpowiedzialna za rejestrację funkcji obsługujących wydarzenia oraz późną inicjalizację niektórych wartości pól.

init_gui(self) – metoda odpowiedzialna za inicjalizację menu (dostępne pola oraz zakres wartości).

reset_gui(self) – funkcja odświeżania GUI.

pull_new_state(self, new_state) – metoda obsługująca zmianę selektora z równoczesnym odświeżeniem menu (tj. zmianę stanu selektora na przykład poprzez wczytanie istniejącej konfiguracji).

on_selector_change(self) – metoda obsługująca zmianę selektora przez bezpośrednią interakcję użytkownika z kontrolką (nie poprzez wczytanie istniejącej konfiguracji).

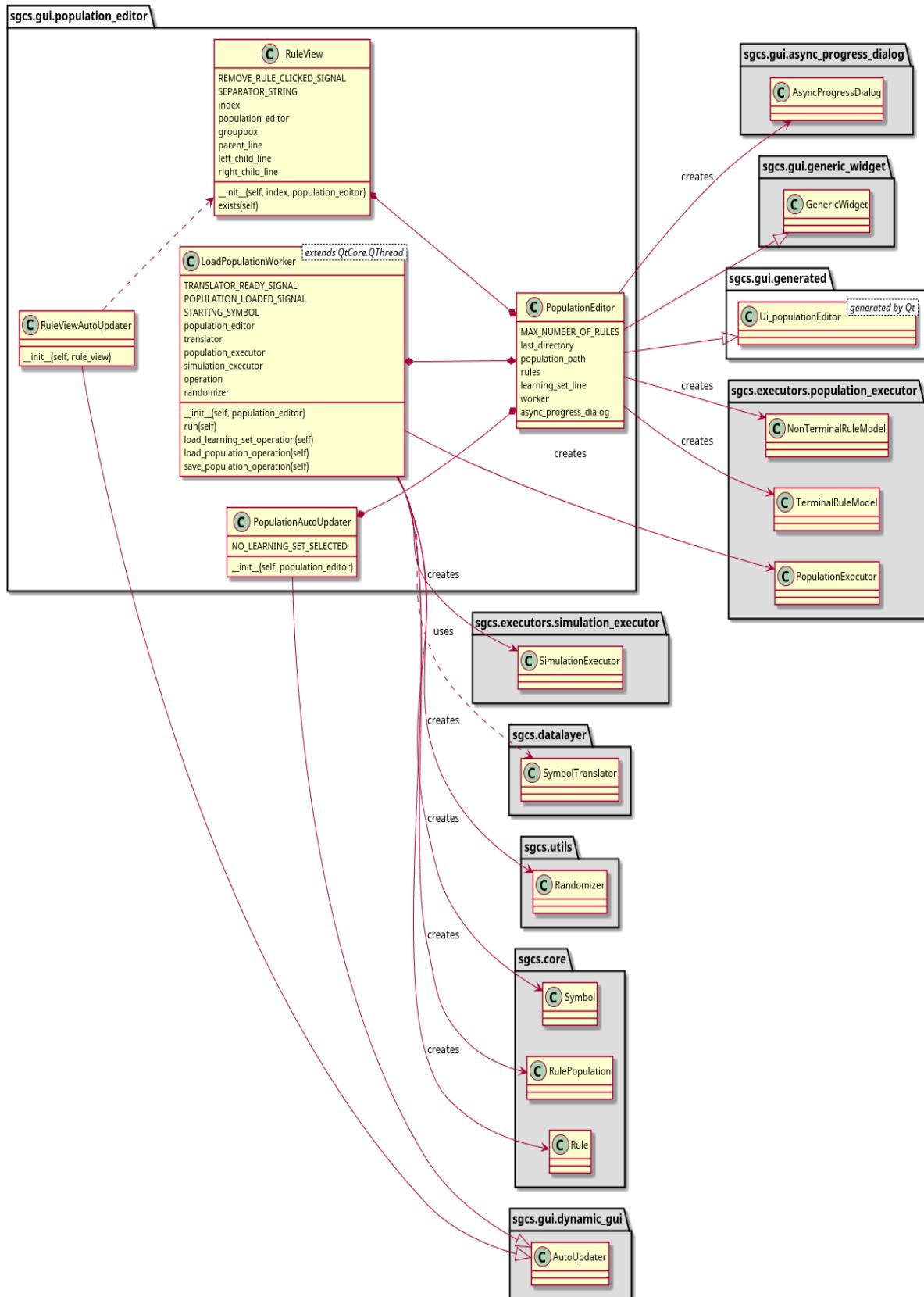
is_in_tournament_mode(self) – metoda zwracająca True, jeżeli obecną selekcją jest selekcja turniejowa, False w przeciwnym przypadku.

create_dynamic_tournament_node(self) – metoda tworząca dynamiczny węzeł odpowiedzialny za widoczność kontrolki rozmiaru turnieju.

on_tournament_size_changed(self) – metoda obsługująca zmianę rozmiaru turnieju.

sgcs.gui.population_editor (Rysunek 55)

sgcs.gui.population_editor



Rysunek 55: Moduł `sgcs.gui.population_editor`

Okno „Edit population” jest tworzone i sterowane przy pomocy klas z modułu `sgcs.gui.population_editor`.

RuleViewAutoUpdater

Klasa ta zapewnia synchronizację danych oraz GUI pojedynczej reguły.

PopulationAutoUpdater

Klasa ta zapewnia synchronizację ścieżki do zbioru uczącego z odpowiednim widgetem.

NO_LEARNING_SET_SELECTED – napis wykorzystywany w kontrolce w przypadku braku ścieżki.

RuleView

Klasa ta zarządza widokiem oraz edycją reguły. Informuje również klasę RuleEditor o potrzebie usunięcia reguły.

REMOVE_RULE_CLICKED_SIGNAL – identyfikator sygnału usunięcia reguły.

SEPARATOR_STRING – napis rozdzielający lewą oraz prawą stronę produkcji.

index – identyfikator reguły.

population_editor – uchwyt obiektu PopulationEditor.

groupbox – groupbox, w którym klasa umieszcza wszystkie swoje elementy GUI (musi być widoczna na zewnątrz w celu podłączenia do GUI PopulationEditor).

parent_line, left_child_line, right_child_line – uchwyty linii tekstu, które składają się na regułę.

exists(self) – Zwraca True, jeżeli istnieje reguła odpowiadająca temu obiekty (tj. w tablicy reguł na pozycji index coś jest).

LoadPopulationWorker

Zadaniem tej klasy jest odciążenie głównego wątku z kilku czasochłonnych procesów. Każdemu uruchomieniu takiego procesu będzie towarzyszyć wyświetlenie odpowiedniego okna, apelującego do użytkownika o cierpliwość. Obecnie Worker może w danej chwili realizować jedno z trzech zadań:

- Wczytywanie zbioru uczącego;
- Wczytywanie pliku populacji;
- Zapisywanie populacji do pliku.

TRANSLATOR_READY_SIGNAL – identyfikator sygnału wysyłanego zaraz po wczytaniu pliku ze zbiorzem uczącym.

POPULATION_LOADED_SIGNAL – identyfikator sygnału wysyłanego po wczytaniu pliku populacji.

STARTING_SYMBOL – stała, symbol startowy gramatyki.

population_editor – uchwyt do instancji PopulationEditor.

translator – zmienna przechowująca obiekt SymbolTranslator. Bez tego obiektu nie jest możliwe tłumaczenie z wewnętrznej reprezentacji symboli na przyjazną człowiekowi oraz odwrotnie. Zmienna zostaje zainicjowana po wczytaniu zbioru uczącego.

population_executor – zmienna przechowującainstancję obiektu PopulationExecutor, zajmującą się logiką biznesową obsługi populacji.

simulation_executor – zmienna przechowującainstancję obiektu SimulationExecutor, zajmującą się logiką biznesową uruchamiania algorytmu oraz zbierania artefaktów.

operation – obecnie realizowane przez workera zadanie.

randomizer – instancja randomizera niezbędna dla prawidłowego utworzenia niektórych części systemu (na przykład populacji reguł).

run(self) – główna metoda wątku. W celu asynchronicznego wywołania należy wykorzystać metodę QtCore.QThread.start(). Przed wywołaniem którejkolwiek z tych dwóch metod należy ustawić odpowiednią wartość pola operation (na przykład load_learning_set_operation w celu załadowania zbioru uczącego).

load_learning_set_operation(self) – operacja wczytująca zbiór uczący. Wczytanie zbioru uczącego powoduje zainicjowanie zmiennej translator odpowiednim instancją klasy SymbolTranslator, co umożliwia (i odblokowuje na poziomie GUI) wczytywanie oraz zapisywanie populacji. Należy pamiętać, że GUI chroni jedynie przed wczytaniem lub zapisaniem populacji w przypadku braku istnienia instancji SymbolTranslator. Nie chroni jednak przed wczytywaniem lub zapisywaniem populacji z innym zestawem uczącym niż ten, przy pomocy którego wygenerowano

populację reguł. Nie może przed tym chronić, gdyż jest to często zachowanie pożądane (tworzymy zestaw reguł wspólnych dla kilku gramatyk, po czym zapisujemy je dla każdej z nich). Brak ostrożności może jednak spowodować powstanie źle wygenerowanych zbiorów.

load_population_operation – operacja umożliwiająca wczytanie istniejącego pliku populacji. Przed wczytaniem należy upewnić się, że wybrano pożądany zbiór uczący.

save_population_operation – operacja umożliwiająca zapisanie obecnej populacji do pliku. Przed zapisem należy upewnić się, że wybrano pożądany zbiór uczący.

PopulationEditor

Klasa ta niejako łączy wszystkie powyższe, zapewniając komunikację pomiędzy nimi, dodatkowo zajmuje się aktualizacją modelu populacji (dodawanie i usuwanie reguł w odpowiedzi na wysypane przez siebie oraz workera sygnały).

MAX_NUMBER_OF_RULES – stała definiująca jak wiele reguł może znaleźć się w populacji.

last_directory – pole przechowujące ścieżkę ostatnio otwieranego przez nas folderu, dzięki czemu szukanie plików rozpoczynamy od miejsca, gdzie je ostatnio skończyliśmy.

population_path – ścieżka do pliku ze zbiorem uczącym.

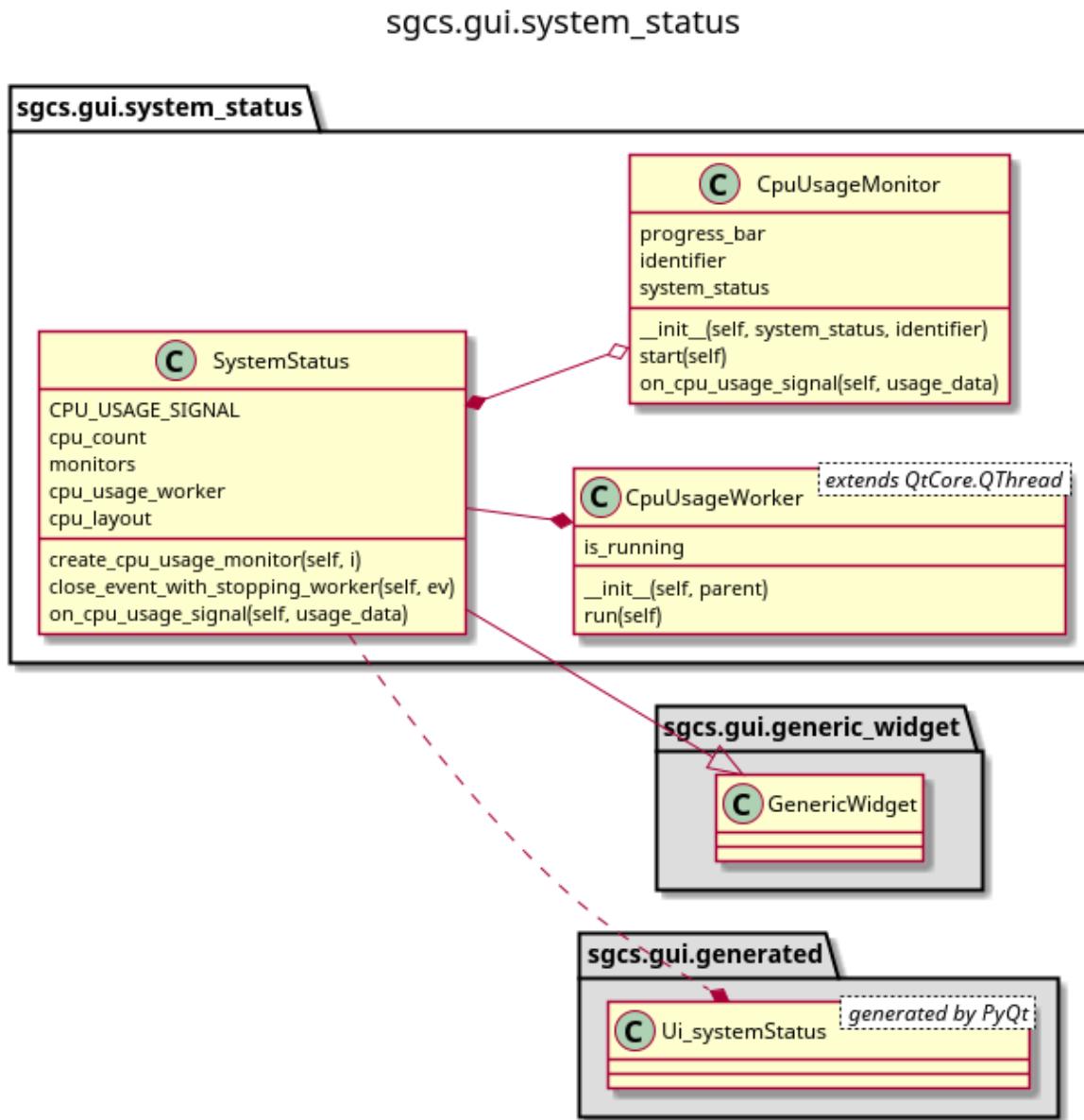
rules – lista obiektów RuleModel, będących reprezentacją reguł na potrzeby tego okna.

learning_set_line – linia odpowiadająca za wyświetlanie ścieżki do zbioru uczącego.

worker – instancja klasy LoadPopulationWorker.

async_progress_dialog – instancja widgetu asynchronicznego okna postępu (komunikat wyświetlany w trakcie trwania długich operacji).

sgcs.gui.system_status (Rysunek 56)



Rysunek 56: Moduł `sgcs.gui.system_status`

Zadaniem tego modułu jest monitorowanie stanu systemu (dokładnie pamięci oraz procesora) i informowanie o jego użyciu przez użytkownika. Nie jest to moduł bezpośrednio związany z samym algorytmem sGCS, jednakże jest niezwykle przydatny. Dzięki niemu można w łatwy sposób śledzić wykorzystanie wszystkich rdzeni oraz ilość zarezerwowanej pamięci. Domyślnie algorytm stara się optymalnie wykorzystywać przydzielone mu zasoby, toteż takie narzędzie jest przydatne w ustalaniu czy system rzeczywiście optymalnie wykorzystuje przydzielone mu zasoby i czy nie da się przyspieszyć całego procesu poprzez modernizację maszyny uruchamiającej aplikację.

CpuUsageWorker

Klasa ta w regularnych odstępach (1s) odpytuje system przy pomocy biblioteki psutil o jego obciążenie procesorów i ilość wolnej pamięci. Następnie wysyła tę informację przy pomocy sygnału `SystemStatus.CPU_USAGE_SIGNAL`.

is_running – zmienna boolowska umożliwiają prawidłowe zakończenie działania wątku (po zmianie jej wartości na False nie dalej jak sekundę później nastąpi zakończenie działania wątku).

__init__(self, parent) – konstruktor. W tym przypadku pole parent to widget, do którego `CpuUsageWorker` ma wysyłać sygnały informujące o użyciu procesora (domyślnie widget powiązany z instancją `SystemStatus`).

run(self) – główna funkcja wątku. Zamiast niej powinno wywoływać się metodę `QtCore.QThread.start` w celu asynchronicznego wykonania.

CpuUsageMonitor

Klasa zajmująca się nasłuchiwaniem na sygnały wykorzystania procesora. Jej zadaniem jest obsługa tylko jednego rdzenia.

progress_bar – widget pasek postępu powiązany z obserwowanym rdzeniem procesora. Przyjmuje wartości od 0% do 100% zużycia.

identifier – Liczba naturalna jednoznacznie identyfikująca obserwowany rdzeń.

start(self) – Wywołanie tej metody powoduje rejestrację klasy na sygnały użycia procesora.

on_cpu_usage_signal(self, usage_data) – funkcja zajmująca się obsługą przybyłego sygnału zużycia procesora (aktualizuje progress_bar).

SystemStatus

Klasa tworząca i koordynującą działanie pojedynczej instancji CpuUsageWorker oraz co najmniej jednej instancji CpuUsageMonitor. Oprócz tego zajmuje się obserwowaniem i informowaniem o zużyciu pamięci.

CPU USAGE SIGNAL – identyfikator sygnału zużycia procesora.

cpu_counts – zmienna przechowująca całkowitą liczbę rdzeni w systemie.

monitors – instancje klasy CpuUsageMonitor (tyle ile wynosi cpu_counts).

cpu_usage_worker – instancja obiektu CpuUsageWorker.

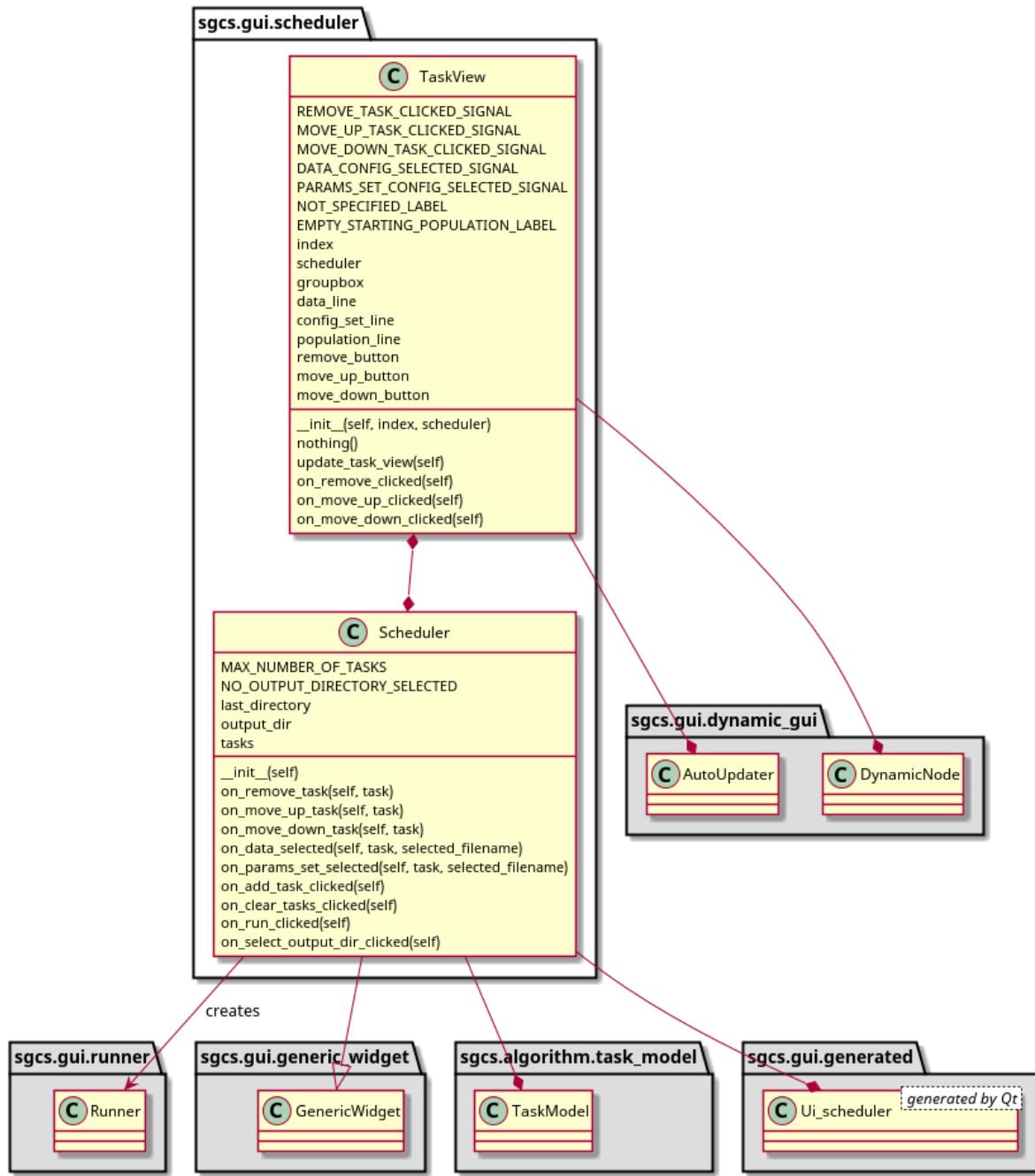
cpu_layout – część interfejsu, do którego dodają się paski postępu monitorów.

create_cpu_usage_monitor(self, i) – metoda nadzorująca utworzenie oraz rejestrację pojedynczego monitora. Parametr i to numer rdzenia, którego obserwacją ma zająć się monitor.

close_event_with_stopping_worker(self, ev) – metoda upewniająca się, że w przypadku zamknięcia okna CpuUsageWorker również prawidłowo zakończy działanie (wywoywane automatycznie tuż przed oryginalną obsługą wydarzenia zamknięcia).

on_cpu_usage_signal(self, usage_data) – metoda rejestrowana na sygnały zużycia procesora. Wyciąga z nich informację o zużyciu pamięci i dokonuje odpowiedniej aktualizacji GUI.

sgcs.gui.scheduler (Rysunek 57)



Rysunek 57: Moduł `sgcs.gui.scheduler`

Moduł ten odpowiada za obsługę menu „Scheduler”. Znajduje się tutaj zatem logika umożliwiająca przygotowanie zestawu zadań (tj. odpalenie konkretnego wariantu algorytmu z konkretnymi parametrami dla konkretnych danych wejściowych), zakolejkowanie ich i ustalenie ścieżki folderu na artefakty z przebiegów algorytmu. Odpowiedzialność za uruchomienie i monitoring przebiegu spoczywa na module `sgcs.gui.runner`, do którego ten moduł wysyła jedynie listę zaplanowanych zadań oraz ścieżkę folderu z artefaktami. Po uruchomieniu algorytmu modyfikacja tych danych nie będzie już możliwa.

TaskView

ViewModel obiektu zadania. Wyświetla jego dane (ścieżkę konfiguracji danych wejściowych *inconf, konfiguracji parametrów *.parconf oraz populacji początkowej *.pop) i zajmuje się ich obsługą.

REMOVE_TASK_CLICKED – identyfikator sygnału informujący, że powiązany z ViewModelem model zadania powinien zostać usunięty.

MOVE_UP_TASK_CLICKED_SIGNAL – identyfikator sygnału informujący, że powiązany z ViewModelem model zadania powinien być przesunięty w górę kolejki.

MOVE_DOWN_TASK_CLICKED_SIGNAL - identyfikator sygnału informujący, że powiązany z ViewModelem model zadania powinien być przesunięty w dół kolejki.

DATA_CONFIG_SELECTED_SIGNAL - identyfikator sygnału informujący o podaniu nowej ścieżki do danych wejściowych *.inconf.

PARAMS_SET_CONFIG_SELECTED_SIGNAL - identyfikator sygnału informujący o podaniu nowej ścieżki do parametrów algorytmu *.parconf.

NOT_SPECIFIED_LABEL – napis wyświetlany w przypadku niesprecyzowanych danych wejściowych lub parametrów algorytmu.

EMPTY_STARTING_POPULATION_LABEL – napis wyświetlany, gdy nie podano ścieżki do populacji początkowej (przyjmowana jest wówczas domyślnie populacja pusta, rozszerzona ewentualnie o losowe reguły początkowe, oczywiście o ile tak stwierdzą parametry algorytmu).

index – indeks powiązanego modelu,

scheduler – uchwyt do obiektu klasy Scheduler.

groupbox – GUI generowane przez.viewmodel, wystawione na potrzeby dołączenia do GUI Schedulera.

data_line – element GUI wyświetlający obecną ścieżkę do pliku *.inconf.

config_set_line – element GUI wyświetlający obecną ścieżkę do pliku *.parconf.

population_line – element GUI wyświetlający obecną ścieżkę do pliku *.pop.

remove_button – przycisk umożliwiający usunięcie powiązanego modelu.

move_up_button – przycisk umożliwiający przesunięcie zadania w górę kolejki.

move_down_button – przycisk umożliwiający przesunięcie zadania w dół kolejki.

nothing() - pusty delegat, przydatny w miejscach gdzie prawdziwy delegat nie jest potrzebny.

update_task_view(self) – metoda zajmująca się aktualizacją GUI na podstawie modelu.

on_remove_clicked(self) – metoda obsługująca naciśnięcie przycisku usunięcia zadania.

on_move_up_clicked(self) – metoda obsługująca naciśnięcie przycisku przesunięcia zadania w górę kolejki.

on_move_down_clicked(self) – metoda obsługująca naciśnięcie przycisku przesunięcia zadania w dół kolejki.

Scheduler

Klasa zarządzająca modelem kolejki zadań. Nasłuchiwa na sygnały wysyłane przez obiekty TaskView i dokonuje odpowiednich modyfikacji kolejki.

MAX_NUMBER_OF_TASKS – maksymalna liczba zadań w kolejce. GUI jest dynamicznie generowane i jest w stanie obsłużyć maksymalnie taką liczbę zaplanowanych zadań.

NO_OUTPUT_DIRECTORY_SELECTED – napis wyświetlany zamiast ścieżki folderu na artefakty, jeżeli ten nie został jeszcze sprecyzowany.

last_directory – zmienna pamiętająca ostatnio otwarty folder.

output_dir – zmienna pamiętająca ścieżkę folderu na artefakty.

tasks – model kolejki zadań.

on_remove_task(self, task) – metoda zarejestrowana na sygnał REMOVE_TASK_CLICKED_SIGNAL, usuwająca zadanie task z kolejki.

on_move_up_task(self, task) – metoda zarejestrowana na sygnał MOVE_UP_TASK_CLICKED_SIGNAL, przenosząca zadanie task w górę kolejki.

on_move_down_task(self, task) – metoda zarejestrowana na sygnał MOVE_DOWN_TASK_CLICKED_SIGNAL, przenosząca zadanie task w dół kolejki.

on_data_selected(self, task, selected_filename) – metoda zarejestrowana na sygnał DATA_CONFIG_SELECTED_SIGNAL, aktualizująca ścieżkę *.inconf danego zadania task.

on_params_set_selected(self, task, selected_filename) – metoda zarejestrowana na sygnał PARAMS_SET_CONFIG_SELECTED_SIGNAL, aktualizująca ścieżkę *.parconf danego zadania task.

on_add_task_clicked(self) – metoda zarejestrowana na naciśnięcie przycisku dodania nowego zadania. Powoduje ona dodanie nowego pustego (tj. z niesprecyzowaną ścieżką konfiguracji wejściowej i parametrów oraz pustą populacją) na końcu kolejki.

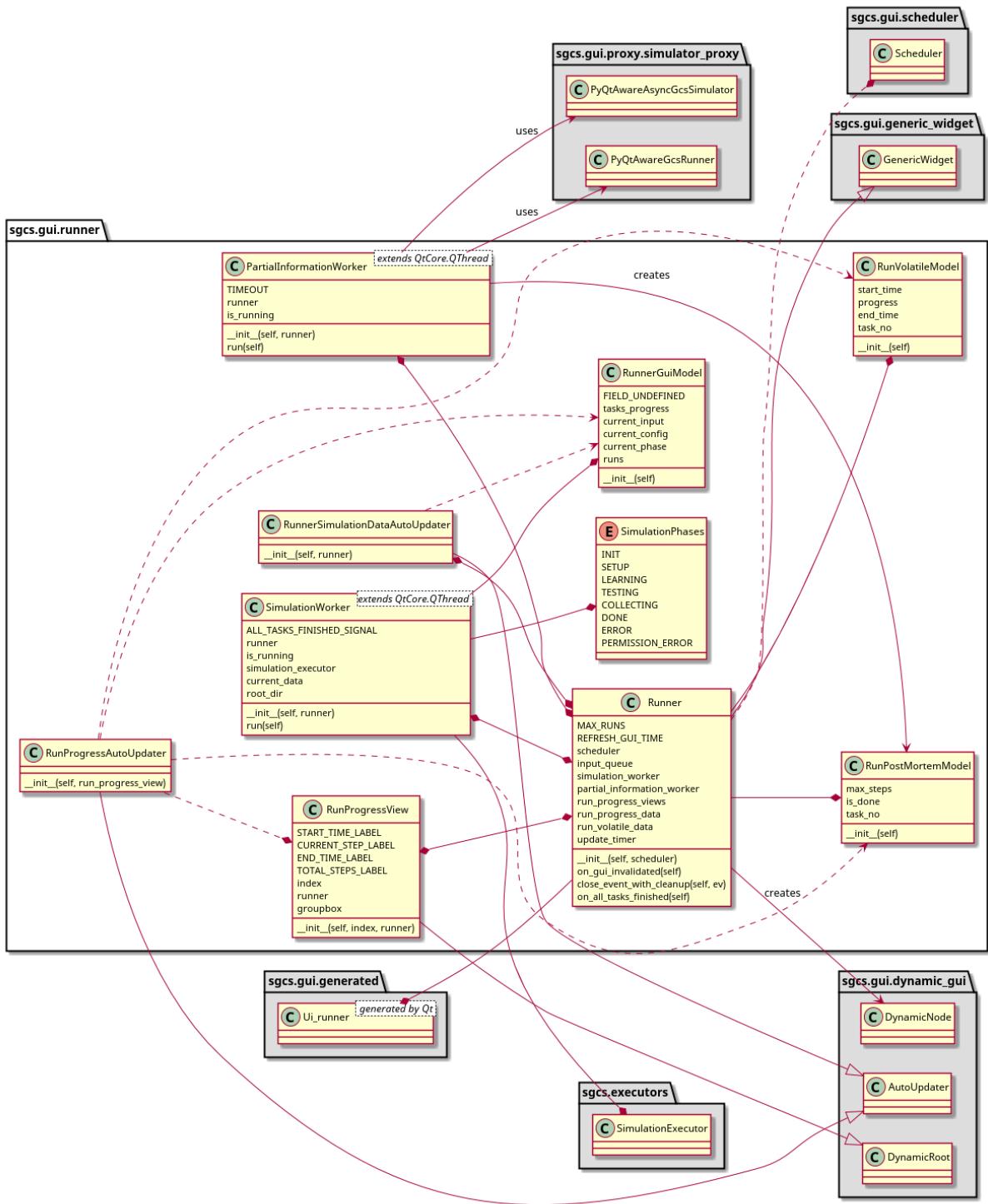
on_clear_tasks_clicked(self) – metoda zarejestrowana na naciśnięcie przycisku wyczyszczenia kolejki zadań. Usuwa wszystkie zakolejkowane zadania.

on_run_clicked(self) – przekierowuje obecną kolejkę zadań i sterowanie do modułu sgcs.gui.runner. Oczywiście pod warunkiem poprawności danych (ścieżka artefaktów musi być podana, każde zadanie musi posiadać sprecyzowane ścieżki do pliku *.inconf i *.parconf, plik *.pop jest opcjonalny).

on_select_output_dir_clicked(self) – medosa obsługująca wybór ścieżki folderu na artefakty.

sgcs.gui.runner (Rysunek 58)

sgcs.gui.runner



Rysunek 58: Moduł `sgcs.gui.runner`

Moduł ten odpowiada za uruchomienie algorytmu i zapewnienie możliwości obserwacji jego działania przy pomocy GUI.

RunPostMortemModel

Model przechowujący niezmienne lub rzadko zmieniające się (na przykład pod koniec przebiegu uczącego). Nie zawiera danych przesyłanych przy pomocy kolejki `Runner.input_queue`. W kodzie modułu ma czasami miejsce zamienne stosowanie terminologii „task” w odniesieniu do zadania (czyli całego uruchomienia algorytmu dla danej populacji startowej, parametrów i danych wejściowych) oraz w odniesieniu do przebiegu uczącego (pojedyncze uruchomienie etapu uczącego dla algorytmu). Należy więc zachować szczególną ostrożność w analizowaniu kodu modułu pod tym względem.

max_steps – maksymalna liczba kroków ewolucyjnych, jaką może podjąć algorytm.

is_done – czy dany przebieg uczący został zakończony.

task_no – numer przebiegu uczącego.

RunVolatileModel

Model przeznaczony do przechowywania danych ulotnych. Są to dane ulegające dosyć częstym zmianom, przesyłane z algorytmu przy pomocy kolejki Runner.input_queue. Mogą czasami nie być aktualne z obecnym stanem algorytmu, w szczególności jeżeli jest ich generowanych w danym momencie dużo, aczkolwiek są to nadal dane diagnostyczne i nie muszą być w 100% aktualne, tak długo jak pozostają w miarę aktualne.

start_time – data rozpoczęcia przebiegu uczącego.

progress – numer obecnego kroku ewolucyjnego.

end_time – data zakończenia przebiegu ewolucyjnego.

task_no – numer przebiegu uczącego.

RunProgressAutoUpdater

Klasa odpowiedzialna za aktualizowanie obecnego stanu przebiegu uczącego.

RunProgressView

Klasa odpowiedzialna za prezentację obecnego stanu przebiegu uczącego. Zajmuje się początkowym ustawniem stylu całego bloku oraz zapewnieniem widoczności odpowiednich jego elementów, podczas gdy samo zadanie odświeżania wartości zostało oddelegowane do instancji RunProgressAutoUpdater. Wyświetla GUI tylko dla przebiegów rzeczywiście istniejących (to jest RunProgressView o warotści pola index = 60 nic nie wyświetli jeżeli jest tylko 50 zaplanowanych przebiegów cyklu uczącego).

START_TIME_LABEL – stała, część dynamicznego GUI.

CURRENT_STEP_LABEL – stała, część dynamicznego GUI.

END_TIME_LABEL – stała, część dynamicznego GUI.

TOTAL_STEPS_LABEL – stała, część dynamicznego GUI.

index – numer przebiegu uczącego.

runner – uchwyt instancji klasy Runner.

groupbox – GUI generowane przez tą klasę, wystawione na potrzeby podpięcia do reszty GUI klasy Runner.

SimulationPhases

Klasa przechowująca fazy, które mają miejsce podczas działania algorytmu, wraz z ich opisami. W trakcie działania algorytmu wyróżniamy następujące fazy:

- INIT – faza początkowa, podczas której ma miejsce głównie przygotowanie interfejsu.
- SETUP – faza podczas której jest brane kolejne zadanie z kolejki, ma miejsce update GUI i przygotowanie środowiska do uruchomienia algorytmu.
- LEARNING – faza podczas której są wykonywane asynchronicznie przebiegi uczące w liczbie sprecyzowanej w odpowiednim pliku konfiguracyjnym *.parconf.
- TESTING – faza testowa następująca po fazie uczącej, podczas której ma miejsce przebieg testowy algorytmu.
- COLLECTING – faza zbierania artefaktów. Po zakończeniu działania algorytmu wszystkie dane zgromadzone z jego przebiegu, wynikowa populacja, itd. są zapisywane w wyznaczonym wcześniej folderze. Po tym kroku następuje przejście do kroku ENVIRONMENT jeżeli kolejka zadań jest niepusta, przejście do kroku DONE w przeciwnym przypadku.
- DONE – przejście do tej fazy następuje po opróżnieniu kolejki zadań. Można bezpiecznie zamknąć okno Runner.
- ERROR – pojawia się w momencie, gdy w trakcie działania wystąpił krytyczny błąd i nie jest możliwe prawidłowe kontynuowanie działania algorytmu.

- **PERMISSION_ERROR** – błąd pojawiający się, gdy podczas działania algorytm nie ma prawa dostępu do jakiegoś miejsca. Najczęściej wiąże się to z brakiem praw do zapisu w folderze z artefaktami, gdyż na przykład jakiś plik jest otwarty przez zewnętrzną aplikację. Algorytm będzie ponawiać próbę dostępu co sekundę, pozostając w tym stanie do czasu rozwiązania przez użytkownika problemu z dostępem do zasobu, po czym wznowi działanie od miejsca, gdzie je ostatnio skończył.

RunnerSimulationDataAutoUpdater

Klasa odpowiedzialna za aktualizację danych dotyczących całej obsługi kolejki zadań. Jej zadaniem jest wyświetlanie postępu w realizacji zadań z kolejki oraz ścieżki do obecnie wykorzystywanych plików wejściowych, co pozwala łatwo zidentyfikować wykonywane właśnie zadanie.

RunnerGuiModel

Model reprezentujący obecny stan kolejki zadań.

FIELD_UNDEFINED – napis wyświetlany w przypadku, gdy dana wartość jest niezdefiniowana (na przykład czas startu przebiegu uczącego, który jeszcze się nie zaczął).

tasks_progress – numer obecnie wykonywanego zadania (lub FIELD_UNDEFINED jeżeli wykonywanie zadań jeszcze się nie rozpoczęło).

current_input – ścieżka do pliku *.inconfig obecnie wykonywanego zadania (lub FIELD_UNDEFINED jeżeli wykonywanie zadań jeszcze się nie rozpoczęło).

current_config – ścieżka do pliku *.parconfig obecnie wykonywanego zadania (lub FIELD_UNDEFINED jeżeli wykonywanie zadań jeszcze się nie rozpoczęło).

current_phase – faza SimulationPhase, w której obecnie Runner się znajduje.

runs – kopia kolejki zadań wykonana na podstawie tej w klasie Scheduler.

SimulationWorker

Klasa odpowiedzialna za obsługę kolejki zadań i uruchamianie algorytmu dla kolejnych zadań.

ALL_TASKS_FINISHED_SIGNAL – identyfikator sygnału informującego, że wszystkie zadania z kolejki zostały wykonane.

runner – uchwyt do instancji Runner.

is_running – zmienna boolowska wykorzystywana do delikatnego wyłączenia workera. Niestety zatrzymanie SimulationWorkera nie jest łatwym zadaniem z całymi pulami procesów zarządzanych przez niego i może nie nastąpić natychmiast, toteż po zatrzymaniu algorytmu w połowie wykonania zaleca się ponowne uruchomienie aplikacji w celu uzyskania pewności, że wątek ten został ostatecznie zniszczony. Jest to o tyle ważne, że właśnie ten wątek jest odpowiedzialny za uruchomienie całego algorytmu, nieprawidłowo zaś zatrzymany algorytm może pozostawić osierocone procesy nadal zajmujące sporą część zasobów komputera. Ponowne uruchomienie aplikacji po nagłym przerwaniu algorytmu daje 100% pewności, że wszystkie zasoby zostały zwolnione.

simulation_executor – instancja obiektu SimulationExecutor zajmującego się logiką biznesową przebiegu symulacji oraz zbieraniem artefaktów.

current_data – instancja RunnerGuiModel.

root_dir – ścieżka do folderu na artefakty.

run(self) – główna metoda wątku. Zdecydowanie nie zaleca się jej ręcznego wywoływanego, jako że jest to metoda zajmująca niezwykle dużo czasu (wykonuje po kolei wszystkie zadania). Zamiast tego dobrze jest wystartować ją w osobnym wątku poprzez wywołanie metody QtCore.QThread.start(self).

PartialInformationWorker

Klasa zajmująca się odbieraniem wiadomości diagnostycznych przesyłanych z wnętrza algorytmu przy pomocy Runner.input_queue. Są to czasy rozpoczęcia, zakończenia przebiegu uczącego oraz obecny krok ewolucyjny. **TIMEOUT** – maksymalny czas (w sekundach) oczekiwania na wiadomość

w Runner.input_queue. W przypadku przekroczenia tego czasu worker sprawdzi swój warunek zakończenia pracy i jeżeli będzie miał ją kontynuować, to przejdzie do dalszego czekania na wiadomość. Przerwy w oczekiwaniu na wiadomość są konieczne w celu umożliwienia workerowi bezpieczne zakończenie pracy w przypadku problemu z działaniem algorytmu. Wybrano 10 sekund w celu uniknięcia obciążania procesora częstym łapaniem wyjątków i sprawdzaniem warunków działania. Oznacza to też, że w przypadku przedwczesnego zakończenia

działania algorytmu wątek zakończy swoje działanie najpóźniej w 10 sekund (czas obsługi pojedynczej wiadomości jest w tym przypadku pomijalnie mały).

runner – uchwyt instancji klasy Runner.

is_running – zmienna boolowska, której przestawienie na False spowoduje zakończenie działania wątku w co najwyżej 10 sekund (patrz opis pola TIMEOUT powyżej).

run(self) – główna metoda wątku. W celu asynchronicznego wywołania należy stosować QtCore.QThread.start(self).

Runner

Klasa odpowiedzialna za obsługę GUI podczas działania algorytmu. W zasadzie oddelegowuje ona całą logikę do swoich klas składowych, sama przechowując kilka współdzielonych przez nie obiektów Zapewnia też prawidłowe zwalnianie zasobów w przypadku przedwczesnego zakończenia działania algorytmu, wyświetlenie odpowiedniego powiadomienia po planowanym zakończeniu działania algorytmu oraz okresowe odświeżanie GUI.

MAX_RUNS – maksymalna liczba przebiegów procesu uczenia. Jeżeli algorytm uruchomi większą liczbę cykli uczących, wówczas nie będą one prawidłowo wyświetlane. Wartość tą można zwiększyć, aczkolwiek PyQt zawsze zarezerwuje zasoby niezbędne do stworzenia maksymalnego GUI, co może spowodować wolniejsze działanie programu.

REFRESH_GUI_TIME – wartość tego pola definiuje co ile milisekund następuje odświeżenie GUI.

scheduler – uchwyt schedulera, w którym zaprojektowano to uruchomienie.

input_queue – kolejka wiadomości typu multiprocessing.Queue. Wykorzystywana do przesyłania wiadomości diagnostycznych z wnętrza działającego algorytmu.

simulation_worker – instancja klasy SimulationWorker.

partial_information_worker – instancja klasy PartialInformationWorker.

run_progress_views – lista (o długości MAX_RUNS) obiektów typu RunProgressView.

run_progress_data – lista obiektów RunVolatileModel.

update_timer – instancja obiektu QtCore.QTimer, wywołujący odświeżenie GUI co REFRESH_GUI_TIME milisekund.

on_gui_invalidated(self) – metoda wymuszająca odświeżenie całego GUI.

close_event_with_cleanup(self, ev) – rozszerzenie standardowej obsługi zamknięcia okna, wymuszające zatrzymanie wszystkich podległych wątków w pierwszej kolejności.

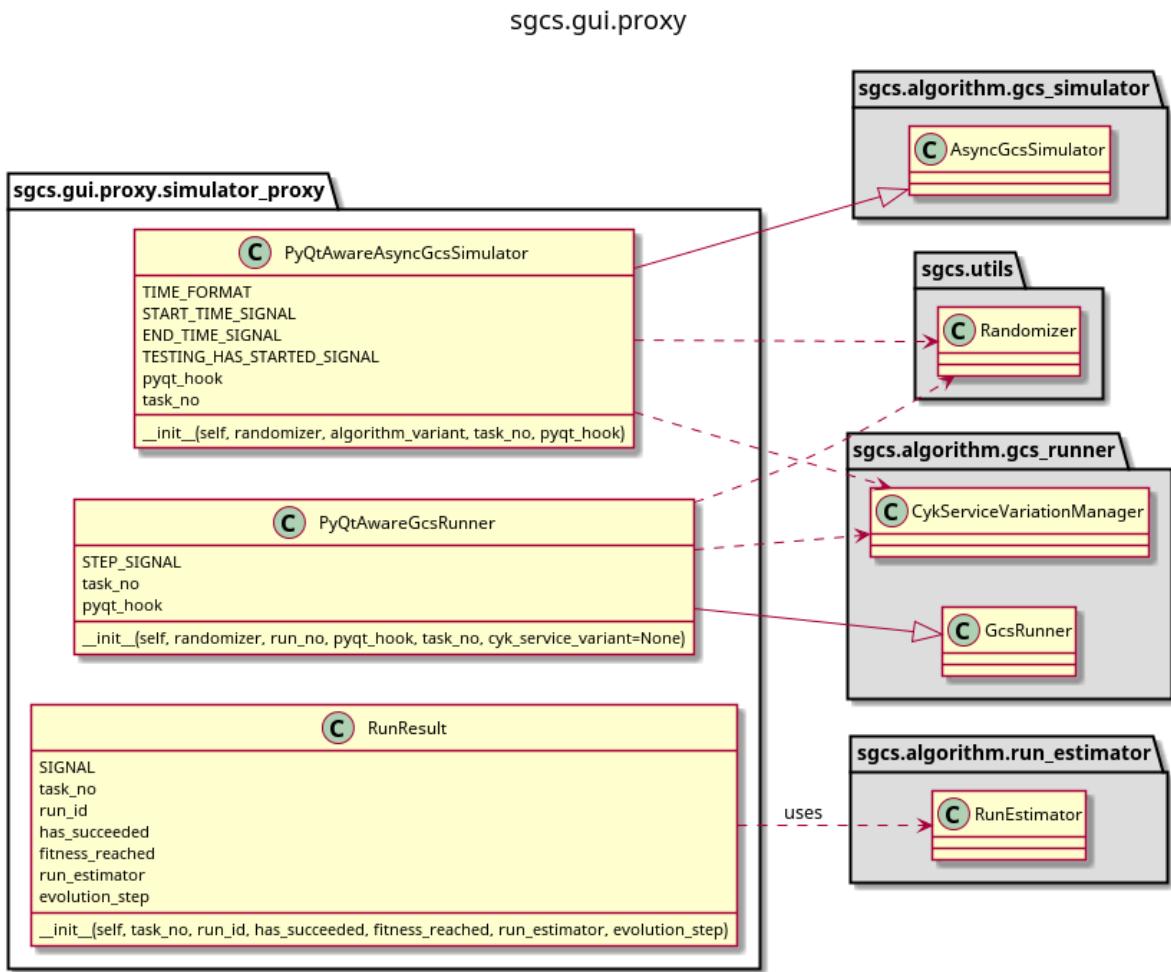
on_all_tasks_finished(self) – metoda wywoływana w chwili planowanego ukończenia wszystkich zadań – powoduje wyświetlenie stosownego komunikatu.

sgcs.gui.proxy (Rysunek 59)

W bibliotece podjęto starania mające na celu uniezależnienie biblioteki od wykorzystywanego GUI. Niestety GUI potrzebuje różnego rodzaju informacji na różnych etapach działania algorytmu i pełne rozdzielenie ich od siebie okazało się nietykalnym zadaniem. Ostatecznie zdecydowano się na następujące rozwiązanie – cały algorytm został napisany tak, że będzie on praktycznie nieświadomy istnienia GUI, natomiast moduł sgcs.gui.proxy stanowić będzie rozszerzenie jego najwyższego poziomu abstrakcji w celu udostępnienia niezbędnych GUI danych. Umożliwia to uruchamianie algorytmu w całkowitej separacji od GUI.

Zatem moduł ten zawiera prosty protokół asynchronicznej komunikacji pomiędzy algorytmem a GUI oraz klasy świadome tego nowego mechanizmu.

Wbrew mylnemu nazewnictwu stosowanemu wewnątrz modułu nie zawiera on żadnych zależności względem biblioteki PyQt. Stanowi rozszerzenie interfejsu algorytmu, które może być wykorzystane przez dowolne GUI, nawet przez aplikację konsolową.



Rysunek 59: Moduł `sgcs.gui.proxy`

RunResult

Klasa odpowiedzialna za wysyłanie pełnego zestawu danych zakończonego przebiegu uczącego algorytmu.

SIGNAL – identyfikator sygnału wysyłanego po zakończeniu przebiegu uczącego algorytmu.

task_no – numer aktualnie wykonywanego zadania.

run_id – numer przebiegu uczącego algorytmu.

has_succeeded – zmienna boolowska informująca czy przebieg zakończył się sukcesem.

fitness_reached – osiągnięty poziom fitness f z przedziału [0;1]

run_estimator – obiekt `run_estimator` zawierający statystyki danego przebiegu.

evolution_step – liczba wykonanych kroków ewolucyjnych.

PyQtAwareGcsRunner

Klasa rozszerzająca klasę `GcsRunner` o wysyłanie sygnałów kroku ewolucyjnego po każdym zakończonym kroku ewolucyjnym.

STEP_SIGNAL – identyfikator sygnału zakończonego kroku ewolucyjnego.

task_no – numer wykonywanego zadania.

pyqt_hook – uchwyt do kolejki `Runner.input_queue` z pomocą której następuje wysyłanie wiadomości.

PyQtAwareAsyncGcsSimulator

Klasa rozszerzająca klasę `AsyncGcsSimulator` o wysyłanie sygnałów rozpoczęcia, zakończenia oraz przejścia do cyklu testowego algorytmu.

TIME_FORMAT – stała, format zapisu wysyłanego czasu.

START_TIME_SIGNAL – identyfikator sygnału czasu rozpoczęcia działania danego przebiegu algorytmu (zarówno uczącego, jak i testowego).

END_TIME_SIGNAL – identyfikator sygnału czasu zakończenia działania danego przebiegu algorytmu (zarówno uczącego, jak i testowego).

TESTING_HAS_STARTED – identyfikator sygnału rozpoczęcia cyklu testowego algorytmu.

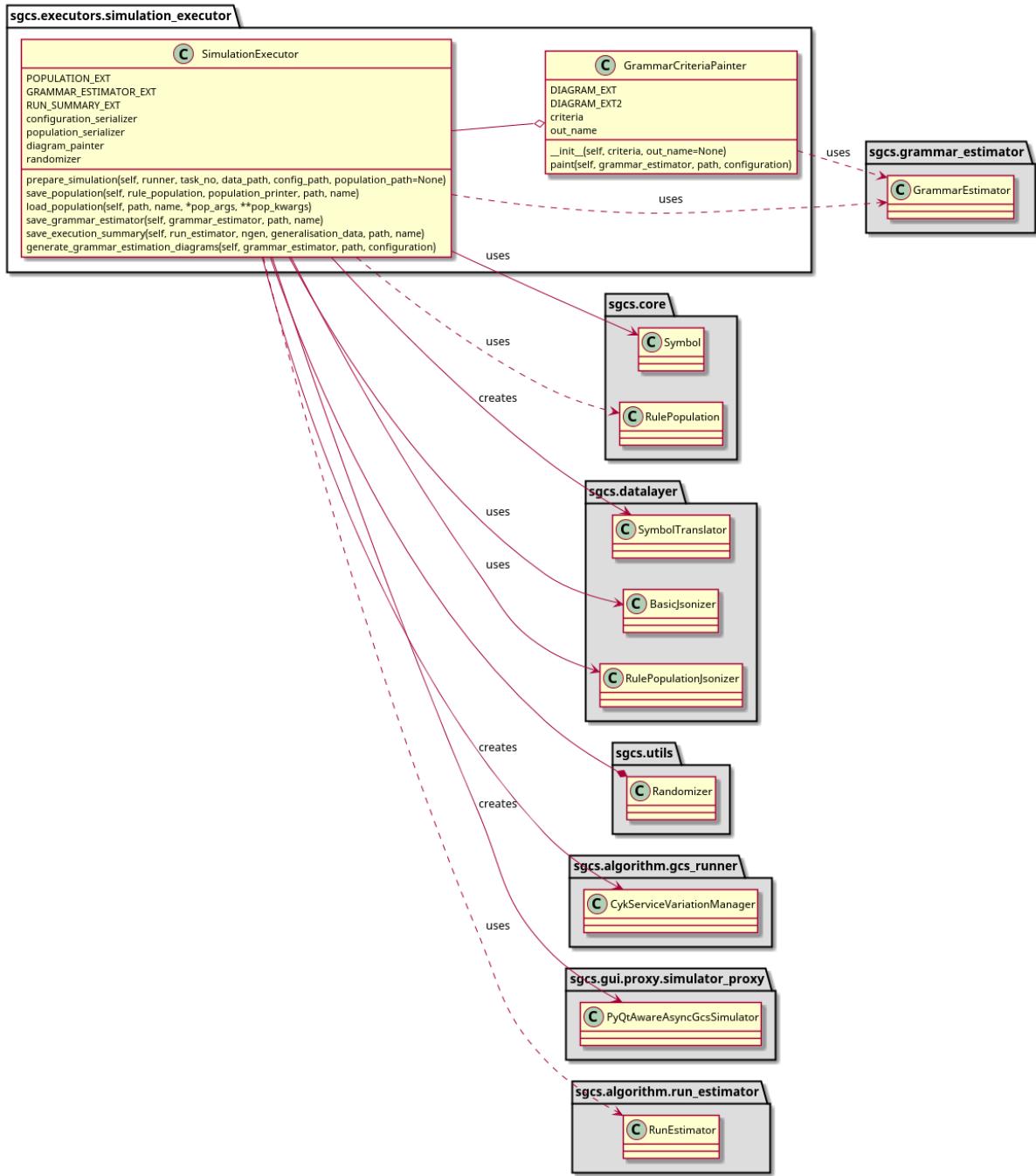
task_no – numer wykonywanego zadania.

pyqt_hook – uchwyt do kolejki Runner.input_queue z pomocą której następuje wysyłanie wiadomości.

Warstwa wykonawców

Cała logika biznesowa biblioteki znajduje się w przestrzeni nazw sgcs.executors. Warstwa ta posiada stosunkowo niewiele funkcjonalności – główne zadania egzekutorów to tłumaczenie danych z reprezentacji wewnętrznej na łatwą do odczytu dla człowieka, generowanie diagramów oraz zbieranie artefaktów z działania algorytmu.

`sgcs.executors.simulation_executor` (Rysunek 60)



Rysunek 60: Moduł `sgcs.executors.simulation_executor`

Moduł ten skupia się na obsłudze logiki występującej podczas działania algorytmu. Zajmuje się prawidłowym uruchomieniem algorytmu oraz zapisywaniem artefaktów, w tym rysowaniem diagramów.

SimulationExecutor

Klasa ta łączy większość logiki systemu, zapewniając możliwość załadowania niezbędnych danych z pomocą modułu `sgcs.datalayer`, zainicjować klasy z modułu `sgcs.algorithm` odpowiednimi danymi, a po wykonaniu wygenerować artefakty i zapisać je ponownie z pomocą modułu `sgcs.datalayer` do folderu artefaktów.

POPULATION_EXT – stała, rozszerzenie plików populacji wykorzystywane w całym systemie („pop”).

GRAMMAR_ESTIMATOR_EXT – stała, rozszerzenie plików oceny gramatyki wykorzystywane w całym systemie („grest”).

RUN_SUMMARY_EXT – stała, rozszerzenie plików podsumowania symulacji wykorzystywane w całym systemie („txt”).

configuration_serializer – instancja BasicJsonizer, która umożliwia tłumaczenie węzłów konfiguracji na format *.json i odwrotnie.

population_serializer – instancja RulePopulationJsonizer, która umożliwia tłumaczenie obiektów typu RulePopulation na format *.json i odwrotnie.

diagram_painter – lista instancji klasy GrammarCriteriaPainter dla wszystkich wspieranych rodzajów diagramów.

randomizer – instancja obiektu Randomizer.

prepare_simulation(self, runner, task_no, data_path, config_path, population_path=None) – metoda konstruująca z podanych danych delegata (który otrzymawszy obiekt konfiguracji uruchomi cały algorytm, a po jego zakończeniu zwróci krotkę zawierającą wyniki wykonania algorytmu), obiekt konfiguracji oraz delegata umożliwiającego wypisanie w czytelny dla człowieka sposób reguł populacji.

save_population(self, rule_population, population_printer, path, name) – metoda umożliwiająca zapisanie populacji reguł rule_population w folderze path w dwóch plikach – szczegółowego i ciężkiego w odczycie przez człowieka pliku name.pop oraz łatwego w odczycie pliku name_view.txt.

load_population(self, path, name, *pop_args, **pop_kwargs) – metoda umożliwiająca odczytanie populacji z pliku path/name.pop. Ponieważ populacja reguł wymaga dodatkowych parametrów przy tworzeniu, parametry te muszą być również dostarczone (pop_args, pop_kwargs).

save_grammar_estimator(self, grammar_estimator, path, name) – metoda powoduje zapisanie obiektu typu GrammarEstimator w pliku path/name.grest

save_execution_summary(self, run_estimator, ngen, generalization_data, path, name) – metoda generuje plik podsumowania przebiegu algorytmu, a następnie zapisuje go w path/name.txt

generate_grammar_estimation_diagrams(self, grammar_estimator, path, configuration) – metoda powoduje wygenerowanie wszystkich wspieranych (lista diagram_painter) rodzajów diagramów i zapisanie ich w folderze path.

GrammarCriteriaPainter

Klasa zdolna do wygenerowania pojedynczego diagramu. W obecnej postaci klasa brakuje trochę do bycia skończonej – wiele cech diagramów, które mogłyby być parametryzowalne, nie są, oprócz tego brak wsparcia dla generowania diagramów w postaci grafiki wektorowej, co byłoby bardzo wygodne w szczególności właśnie w przypadku diagramów. Większość logiki umożliwiającej to jest już zaimplementowana i niewiele potrzeba, aby rozszerzyć tę klasę o nowe możliwości, tymczasem spełnia ona wszystkie wymagania niezbędne do rysowania wymaganych przez aplikację diagramów.

DIAGRAM_EXT – domyślne rozszerzenie grafiki rastrowej, do którego są zapisywane diagramy.

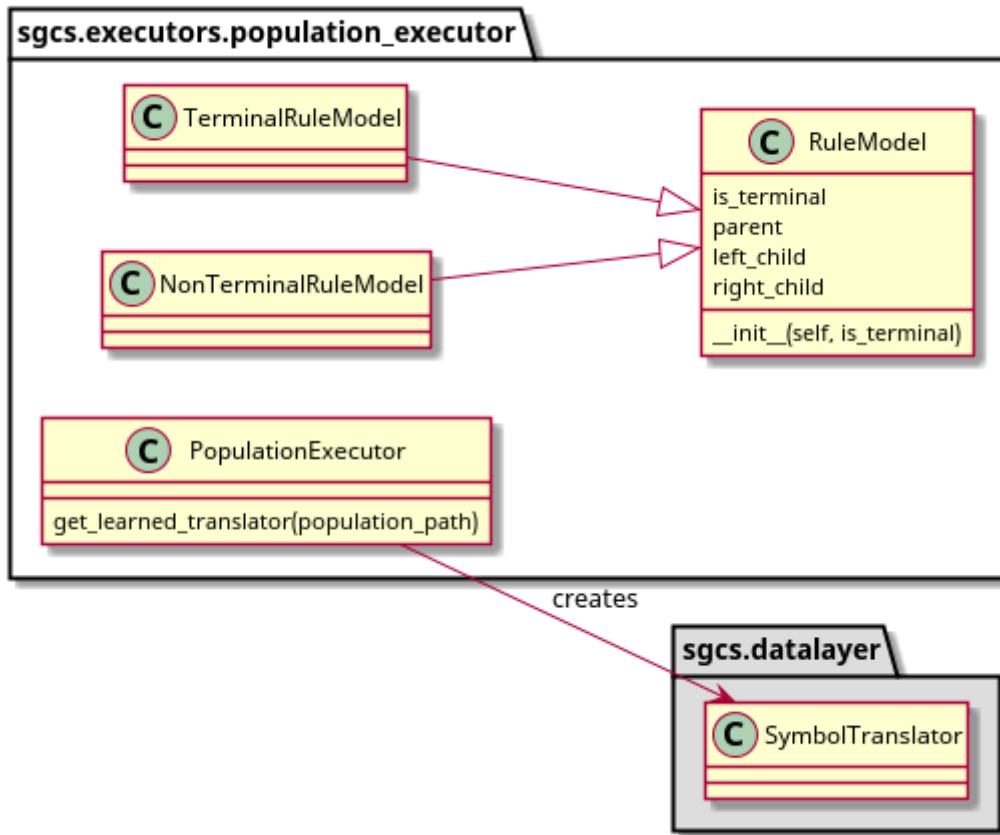
DIAGRAM_EXT2 – domyślne rozszerzenie grafiki wektorowej, do której mogłyby być zapisywane diagramy.

criteria – nazwa i równocześnie identyfikator kryterium, które ma być przedstawione na diagramie.

out_name – opcjonalna nazwa, która ma być stosowana na diagramach zamiast criteria (alternatywna reprezentacja).

paint(self, grammar_estimator, path, configuration) – metoda rysująca, a następnie zapisująca diagram do plików (na razie tylko postać rastrowa, wektorowa może zostać dodana w przyszłości).

`sgcs.executors.population_executor` (Rysunek 61)



Rysunek 61: Moduł `sgcs.executors.population_executor`

Moduł skupia się na zapewnieniu obsługi populacji poza działającym algorytmem. Jest to stosunkowo prosty moduł, większość funkcjonalności jest realizowana przez warstwy niższe (sgcs.core oraz sgcs.datalayer).

RuleModel

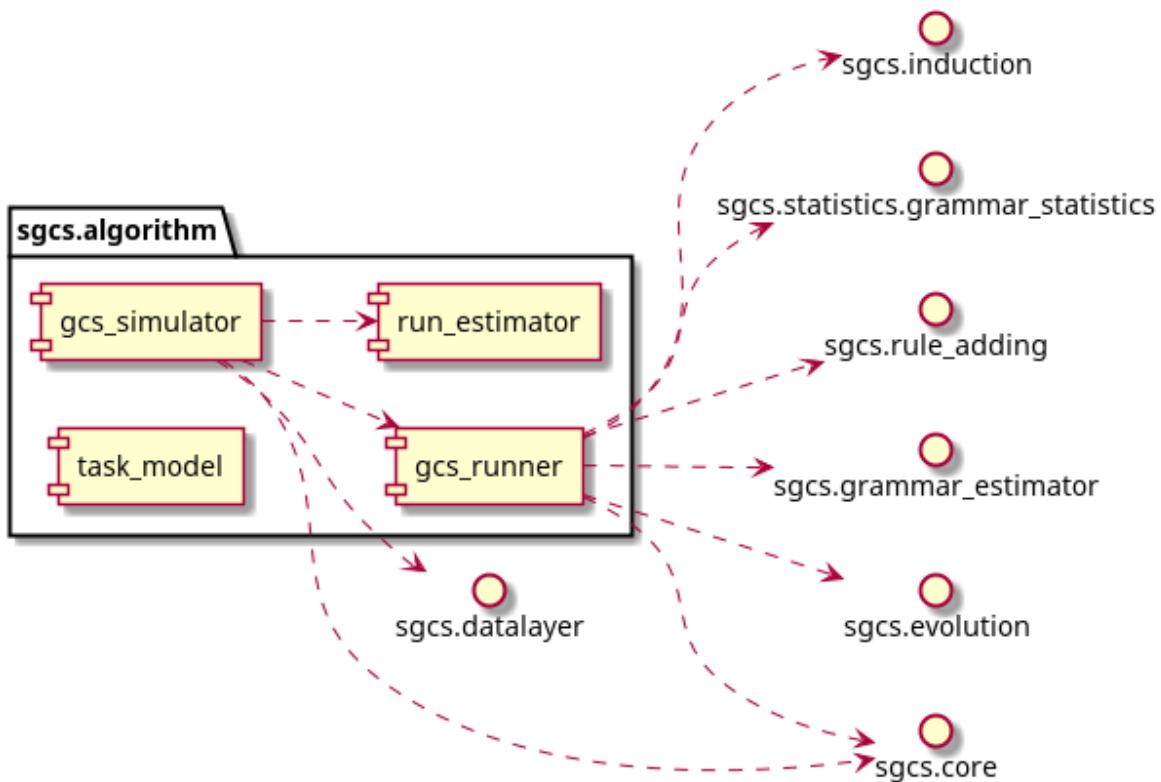
Prosta klasa do przechowywania reguł w czytelnej dla człowieka formie. Posiada dwie klasy dziedziczące po niej (`TerminalRuleModel` oraz `NonTerminalRuleModel`).

PopulationExecutor

Klasa umożliwiająca dostęp do obiektów populacji poza działającym algorytmem.

`get_learned_translator(population_path)` – po podaniu ścieżki do zbioru uczącego `population_path` wczytuje go w całości, w efekcie zwracając obiekt `SymbolTranslator`. Obiekt ten jest zdolny do tłumaczenia symboli w dwie strony – z reprezentacji wewnętrznej do przyjaznej człowiekowi i odwrotnie, jest zatem niezbędny do interaktywnej obsługi populacji poza działającym algorytmem.

Właściwy algorytm



Rysunek 62: Zależności modułu `sgcs.algorithm`

Właściwy algorytm zawiera całą logikę działania algorytmu klasyfikacji gramatycznej, całkowicie odseparowaną od interfejsu graficznego oraz od niskopoziomowych funkcjonalności jak na przykład obsługa systemu plików. Warstwa ta zapewnia spójny interfejs do uruchamiania algorytmu w dowolnym wariantie – domyślnie wspiera GCS, sGCS oraz neg-sGCS wraz z licznymi parametrami i mechanizmami usprawniającymi ich działanie. Powyżej przedstawiono zależności najwyższego modułu algorytmu (Rysunek 62).

`sgcs.algorithm.task_model`

Moduł ten zawiera jedynie prostą klasę – `TaskModel`. Jego pozostałe funkcjonalności zostały przeniesione do innych modułów i powinien zostać usunięty po przeniesieniu klasy `TaskModel`.

`TaskModel`

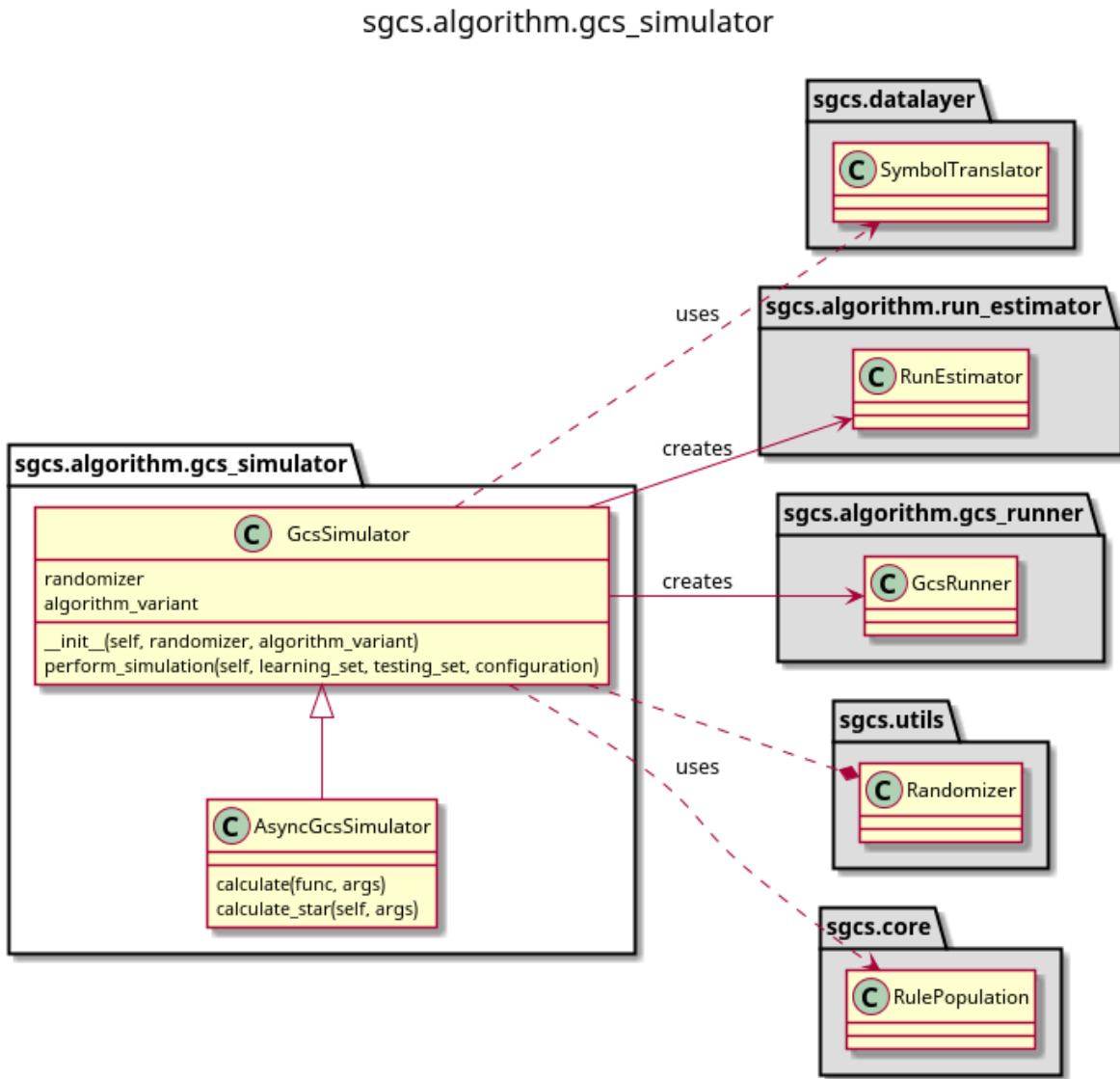
Klasa mająca za zadanie reprezentować pojedyncze zadanie w kolejce zaplanowanych uruchomień algorytmów. Zawiera 3 pola przechowujące ścieżki do związków z zadaniem zasobów.

`data_configuration` – ścieżka do pliku `*.inconf` dla zadania.

`params_configuration` – ścieżka do pliku `*.parconf` dla zadania.

`population_configuration` – ścieżka do pliku `*.pop` dla zadania.

`sgcs.algorithm.gcs_simulator` (Rysunek 63)



Rysunek 63: Moduł `sgcs.algorithm.gcs_simulator`

Moduł ten zawiera klasy `GcsSimulator` oraz `AsyncGcsSimulator`. Są to klasy najwyższego poziomu wykonujące w swoim ciele cały algorytm klasyfikacji gramatyki.

GcsSimulator

Wbrew nazwie klasa ta zajmuje się obsługą dowolnego algorytmu klasyfikacji gramatycznej. Obiektem decydującym o różnym zachowaniu jest `algorithm_variant`, aczkolwiek na tym poziomie jest on tylko przekazywany w dół struktury do klasy `GcsRunner`. `GcsSimulator` uruchamia `configuration.max_algorithm_runs` przebieg uczący, po czym przechodzi do pojedynczego przebiegu testowego. Całość jest wykonana sekwencyjnie.

randomizer – uchwyt obiektu `Randomizer`.

algorithm_variant – obiekt typu `CykServiceVariationManager`.

`perform_simulation(self, learning_set, testing_set, configuration)` – funkcja pobiera `SymbolTranslator` wygenerowany na podstawie zbioru uczącego, `SymbolTranslator` wygenerowany na podstawie zbioru testowego oraz obiekt konfiguracji. Dokonuje przebiegów uczących, a następnie testowych, po czym zwraca krótką zawierającą dane zebrane podczas procesu.

AsyncGcsSimulator

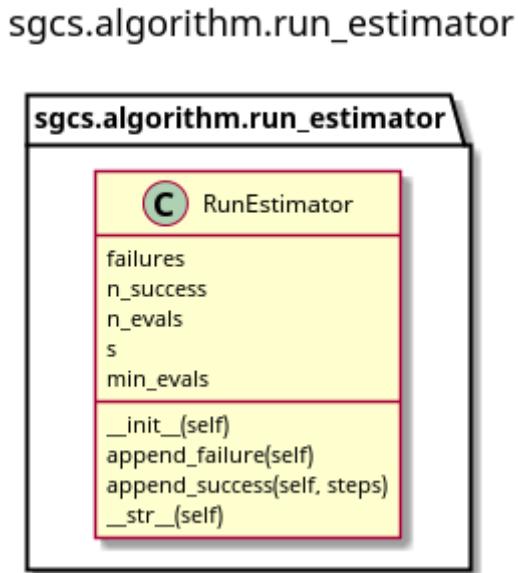
Klasa ta w przeciwieństwie do swojej nadklasy uruchamia cykle uczące równolegle zamiast sekwencyjnie. Na samym początku zostaje utworzona pula procesów o takiej liczności jak liczbę rdzeni logicznych dostępnych na danym komputerze (czyli uwzględnia liczbę rdzeni fizycznych oraz technologie typu Hyper-Threading). Następnie wszystkie przebiegi cyklu uczącego są wykonywane asynchronicznie przez tą

pulę procesów. Główny proces zajmuje się jedynie zbieraniem wyników oraz ewentualnym łączeniem niektórych danych (np. przeliczanie średniej wartości funkcji fitness w danym kroku ewolucyjnym dla wszystkich przebiegów uczących algorytmu). Dzięki temu wykorzystanie asynchronicznego zamiast sekwencyjnego symulatora pozostaje transparentne i daje zauważalne przyspieszenie w przypadku parsowania gramatyk wymagających dużej ilości kroków ewolucyjnych do wyuczenia się gramatyki.

calculate(func, args) – funkcja pomocnicza wywołująca pierwszy parametr z listą parametrów przekazaną w postaci drugiego argumentu.

calculate_star – funkcja przygotowana do wykorzystania z multiprocessing imap, w wygodny sposób przekazuje parametry do funkcji, zapewnia inną wartość seeda dla każdego z procesów oraz rozszerza zwracaną wartość funkcji o dodatkowe informacje wymuszone przez asynchroniczność wykonania obliczeń.

sgcs.algorithm.run_estimator (Rysunek 64)



Rysunek 64: Moduł sgcs.algorithm.run_estimator

Klasa ta odpowiada za gromadzenie statystyk wykonania algorytmu. Po każdym ukończeniu przebiegu uczącego jest wołana metoda append_failure lub append_success. W efekcie po zakończeniu uczenia instancja tej klasy jest w stanie podać wartości n_success, n_evals, s oraz min_evals umożliwiające ocenę przebiegu procesu uczenia.

failures – liczbę przebiegów uczących zakończonych porażką.

n_success – parametr nSuccess (liczba przebiegów uczących zakończonych znalezieniem satysfakcjonującej gramatyki).

n_evals – parametr nEvals (średnia liczba kroków potrzebnych do znalezienia satysfakcjonującej gramatyki).

s – parametr s (odchylenie standardowe wartości nEvals).

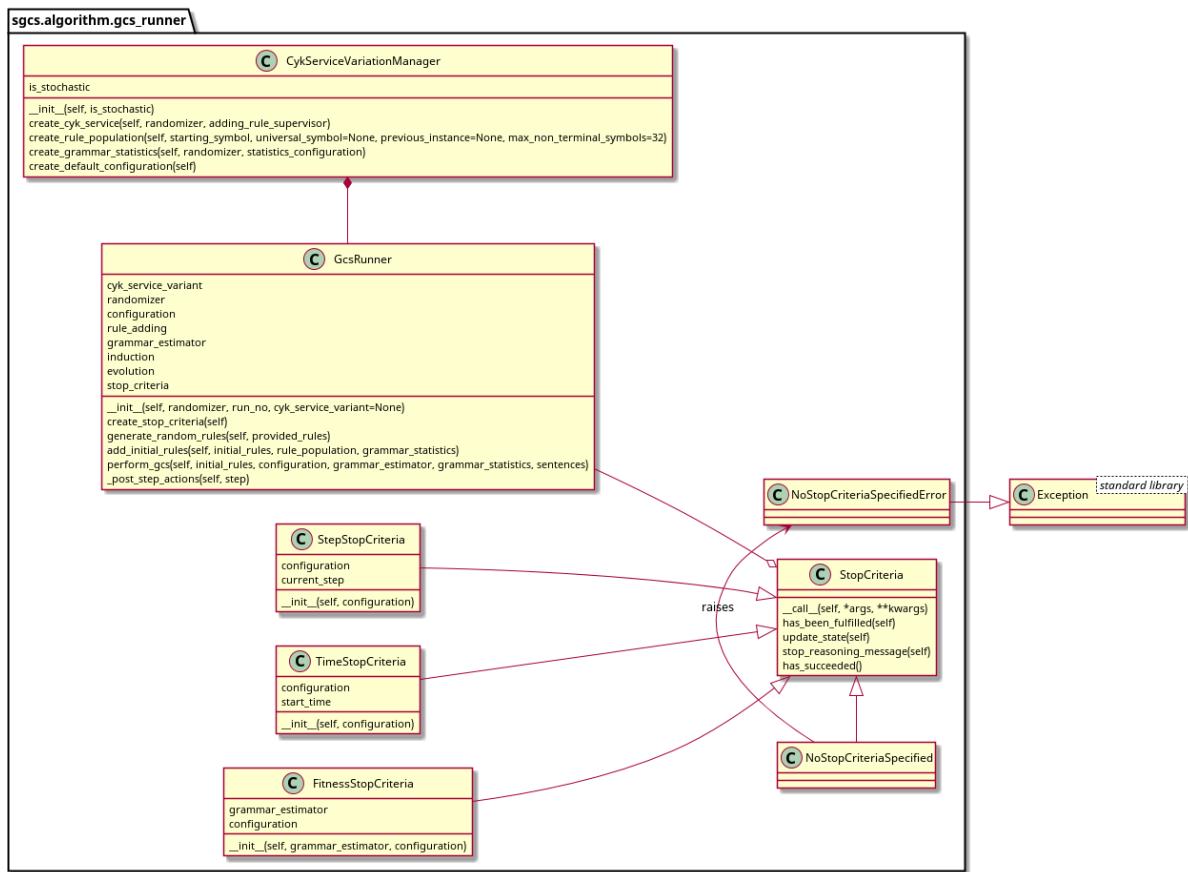
min_evals – parametr minEvals (minimalna liczba kroków potrzebnych do znalezienia satysfakcjonującej gramatyki).

append_failure(self) – zapisuje przebieg jako zakończony porażką.

append_success(self, steps) – zapisuje przebieg jako zakończony sukcesem (wraz z ilością kroków).

sgcs.algorithm.gcs_runner (Rysunek 65)

sgcs.algorithm.gcs_runner (hidden outer dependencies)



Rysunek 65: Moduł `sgcs.algorithm.gcs_runner`

Moduł `sgcs.algorithm.gcs_runner` gromadzi całą logikę pojedynczego przebiegu algorytmu w jedną metodę. Poszczególne klasy tego modułu posiadają wiele zależności od innych modułów i przedstawienie ich w czytelny sposób na diagramie jest praktycznie niemożliwe, dlatego też zależności międzymodułowe nie zostały przedstawione. Jest on zależny od wielu klas należących do następujących modułów:

- `sgcs.utils`;
- `sgcs.rule_adding`;
- `sgcs.core`;
- `sgcs.statistics,grammar_statistics`;
- `sgcs.grammar_estimator`;
- `sgcs.evolution`;
- `sgcs.induction`;

CykServiceVariationManager

Klasa ta zarządza różnicami w zachowaniu pomiędzy algorytmami stochastycznymi (sGCS, neg-sGCS) a klasycznymi (GCS). Zamysłem jest tutaj wystawienie wspólnego API dla obu grup algorytmów, ukrywając w ciałach metod wszelkie różnice pomiędzy nimi. Dzięki temu klient tej klasy (`GesRunner`) może pozostać w całkowitej nieświadomości czy wykorzystywany przez niego algorytm jest stochastyczny, czy też nie.

`is_stochastic` – przechowuje informację czy obecnie wykorzystywany algorytm jest stochastyczny, czy też nie.

`create_cyk_service(self, randomizer, adding_rule_supervisor)` – metoda tworzy instancję typu `CykService` w przypadku algorytmu klasycznego lub `StochasticCykService` w przypadku algorytmu stochastycznego.

create_rule_population(self, starting_symbol, universal_symbol=None, previous_instance=None, max_non_terminal_symbols=32) – metoda tworząca instancję RulePopulation lub StochasticRulePopulation.

create_rule_statistics(self, randomizer, statistics_configuration) – metoda tworząca odpowiednią instancję liczącą statystyki gramatyki (klasyczną lub pasieki). Należy zwrócić uwagę, że decyzja jest w tym przypadku zależna od tego czy dana konfiguracja akceptuje uczenie na zdaniach negatywnych (Pasieka nie, klasyczna tak).

create_default_configuration(self) – metoda tworząca domyślny obiekt konfiguracji dla danego algorytmu.

StopCriteria

Abstrakcyjna klasa realizująca pojedynczy warunek stopu dla głównej pętli algorytmu.

Przykład użycia:

```
criterias = [StopCriteria1(), StopCriteria2()]
```

```
while not any(cr() for cr in criterias):
```

```
    ...
```

```
message = „Success!” if any(cr.has_succeeded() for cr in criterias) else „Failure!”
```

```
print(message)
```

has_been_fullfilled(self) – metoda sprawdzająca spełnienie warunku.

update_state(self) – metoda aktualizująca wewnętrzny stan instancji kryterium. Kiedy wywołujemy metodę __call__ kryterium, wówczas najpierw zostanie wywołana metoda update_state, a następnie dopiero zwrócony rezultat has_been_fullfilled.

stop_reasoning_message(self) – metoda wypisująca przyczynę zatrzymania (przydatne przy kilku warunkach stopu).

has_succeeded() – metoda informująca czy nastąpienie tego warunku stopu powoduje sukces, czy też nie. Dzięki temu możemy mieszać różne kryteria razem i na podstawie ich stanu po opuszczeniu pętli ustalić, czy nastąpił sukces, czy też porażka. Domyślnie zwraca wartość False.

NoStopCriteriaSpecified

Klasa dziedzicząca po StopCriteria. Przy próbie wywołania metody has_been_fullfilled podnosi wyjątek NoStopCriteriaSpecifiedError. Jest to przydatne gdy dysponuje się klasą nieposiadającą domyślnych kryteriów stopu i chce się mieć pewność, że użytkownik zapewni własne.

FitnessStopCriteria

Warunek stopu spełniony, kiedy przejście gramatyki zakończy się uzyskaniem fitness na poziomie configuration.satisfying_fitness. Jego metoda has_succeeded() zwraca wartość True.

grammar_estimator – uchwyt do instancji GrammarEstimator, która pozwala sprawdzić obecną wartość fitness gramatyki.

configuration – uchwyt do węzła konfiguracyjnego pozwalający odczytać wartość configuration.satisfying_fitness.

StepStopCriteria

Warunek stopu spełniony, kiedy current_step przekroczy configuration.max_algorithm_steps. current_step, jest preinkrementowany przy każdym wywołaniu metody update_state (czyli również przy __call__).

configuration – uchwyt do węzła konfiguracyjnego pozwalający odczytać wartość configuration.max_algorithm_steps.

current_step – obecny krok (aczyna liczenie od 0).

TimeStopCriteria

Warunek stopu spełniony, kiedy obecny czas jest większy od start_time + configuration.max_execution_time.

configuration – uchwyt do węzła konfiguracyjnego pozwalający na odczytanie wartości configuration.max_execution_time

start_time – czas w chwili utworzenia tego obiektu.

GesRunner

Klasa ta realizuje główną pętlę algorytmu w sposób niezależny od tego, czy jest stochastyczny i czy wykorzystuje zdania negatywne w procesie uczenia. Wykorzystuje wszystkie warunki stopu (FitnessStopCriteria, TimeStopCriteria oraz StepStopCriteria).

cyk_service_variant – klasa zajmująca się obsługą różnic pomiędzy różnymi wariantami algorytmu.

randomizer – uchwyt instancji Randomizer.

configuration – uchwyt węzła konfiguracyjnego.

rule_adding – instancja klasy RuleAddingSupervisor.

grammar_estimator – obiekt klasy GrammarEstimator.

induction – instancja klasy CykService.

evolution – instancja klasy EvolutionService.

stop_criteria – lista obiektów StopCriteria wykorzystywanych jako warunek stopu głównej pętli algorytmu.

create_stop_criteria(self) – Funkcja inicjująca listę stop_criteria nowymi instancjami klas FitnessStopCriteria, StepStopCriteria oraz TimeStopCriteria.

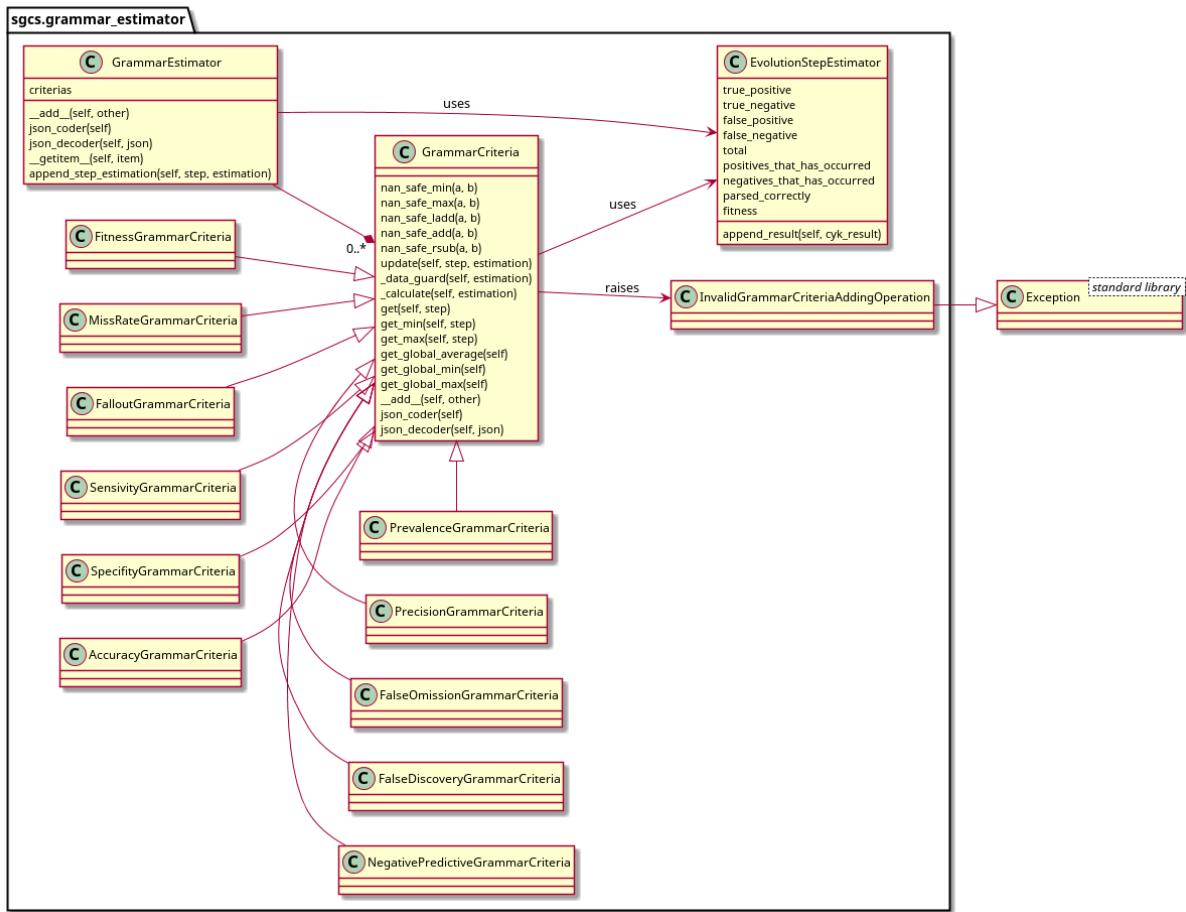
generate_random_rules(self, provided_rules) – metoda zwiększająca zbiór reguł początkowych o configuration.rule.random_starting_population_size – len(provided_rules).

add_initial_rules(self, initial_rules, rule_population, grammar_statistics) – metoda dodająca listę reguł startowych initial_rules do populacji rule_population z uwzględnieniem aktualizacji grammar_statistics.

perform_gcs(self, initial_rules, configuration, grammar_estimator, grammar_statistics, sentences) – główna metoda algorytmu. Inicjuje zestaw reguł początkowych i wykonuje kolejne kroki ewolucyjne aż do momentu nastąpienia któregoś z warunków stopu.

post_step_actions(self, step) – metoda zawierająca zestaw działań wykonywanych po zakończeniu pojedynczego kroku ewolucyjnego. Istniejąca implementacja jest pusta, daje to jednak możliwości rozwoju klasom dziedziczącym (jak na przykład AsyncGcsRunner).

sgcs.grammar_estimator (Rysunek 66)



Rysunek 66: Moduł `sgcs.grammar_estimator`

Moduł ten wykorzystuje się do gromadzenia informacji o kondycji algorytmu. Na podstawie liczby należących sparsowanych, nienależących sparsowanych, nienależących niesparsowanych oraz należących niesparsowanych zdań są obliczane wartości macierzy omylek dla każdego kroku ewolucyjnego (oraz samego testu generalizacji).

EvolutionStepEstimator

Klasa przechowująca liczbę zdań pozytywnych sparsowanych (True Positive), liczbę zdań negatywnych niesparsowanych (True Negative), liczbę zdań pozytywnych niesparsowanych (False Negative) oraz liczbę zdań negatywnych sparsowanych (False Positive). Zawiera jeszcze kilka pomocniczych miar i sama zajmuje się kontrolą tych wszystkich wartości – wystarczy wywołać jej metodę `append_result` wraz z obiektem CykResult zwróconym przez instancję CykService.

true_positive – pole służące do odczytu ilości sparsowanych zdań prawdziwych.

true_negative – pole służące do odczytu ilości niesparsowanych zdań negatywnych.

false_positive – pole służące do odczytu ilości sparsowanych zdań negatywnych.

false_negative – pole służące do odczytu ilości niesparsowanych zdań pozytywnych.

total – liczbę wszystkich sparsowanych i niesparsowanych zdań

positives_that_has_occurred – zmienna opisująca liczbę pozytywnych zdań, które wystąpiły.

negatives_that_has_occurred – zmienna opisująca liczbę negatywnych zdań, które wystąpiły.

parsed_correctly – liczbę prawidłowo sparsowanych zdań.

fitness – osiągnięty w kroku poziom fitness.

GrammarCriteria

Klasa odpowiedzialna za gromadzenie analiz wielu kroków ewolucyjnych algorytmów. Przyjmuje kolejne obiekty EvolutionStepEstimator ze wszystkich uruchomień i na ich podstawie generuje daną statystykę na przestrzeni wielu kroków, w każdym z nich obliczając ją na podstawie wszystkich dostępnych przebiegów algorytmu. Wiele innych klas dziedziczy w tym module po tej klasie, nie ma jednak sensu poświęcać im osobnego miejsca na omówienie, jako że implementują one po prostu metody `_calculate` oraz `_data_guard` zgodnie ze wzorem na statystykę, którą reprezentują (co jest zawarte w ich nazwie).

`nan_safe_min(a, b)` – funkcja min odporna na wartość 'nan' (zwraca drugą z wartości w przypadku gdy pierwsza nie jest liczbą).

`nan_safe_max(a, b)` – funkcja max odporna na wartość 'nan' (zwraca drugą z wartości w przypadku gdy pierwsza nie jest liczbą).

`nan_safe_ladd(a, b)` – operacja dodawania lewostronne odporna na wartość 'nan' (zwraca b w przypadku gdy a nie jest liczbą).

`nan_safe_add(a, b)` – operacja dodawania odporna na wartość 'nan' (zwraca drugą z wartości w przypadku gdy pierwsza z nich nie jest liczbą).

`nan_safe_rsub(a, b)` – prawostronne odporna na wartość 'nan' operacja odejmowania (zwraca a w przypadku gdy b nie jest liczbą).

`update(self, step_estimation)` – aktualizuje przebieg statystyki na podstawie kolejnego obiektu `step_estimation`.

`_data_guard(self, estimation)` – abstrakcyjna metoda, której konkretyzacje powinny zwracać True, jeżeli dla tego `estimation` dana statystyka istnieje (na przykład statystyka `Precision` nie jest zdefiniowana jeżeli żadne zdanie nie zostało sparsowane). Zazwyczaj wystarczy, aby funkcja ta pilnowała czy mianownik wzoru danej statystyki jest niezerowy.

`_calculate(self, estimation)` – abstrakcyjna metoda, której konkretyzacje zajmują się liczeniem danej statystyki (dla poprawnych danych, funkcja ta wywołuje się tylko wtedy, kiedy metoda `_data_guard` zwróciła True).

`get(self, step)` – pobranie średniej wartości statystyki w danym kroku.

`get_min(self, step)` – pobranie wartości minimalnej statystyki w danym kroku.

`get_max(self, step)` – pobranie wartości maksymalnej statystyki w danym kroku.

`get_global_average(self)` – pobranie globalnej średniej statystyki na podstawie wszystkich kroków.

`get_global_min(self)` – pobranie globalnej wartości minimalnej statystyki na podstawie wszystkich kroków.

`get_global_max(self)` – pobranie globalnej wartości maksymalnej statystyki na podstawie wszystkich kroków.

`_add_(self, other)` – umożliwia dodawanie do siebie obiektów `GrammarCriteria`. W wyniku dodania dwóch obiektów tworzony jest trzeci, który posiada połączoną wiedzę dwóch pozostałych na temat gramatyki. Funkcja przydatna w przypadku asynchronicznego wykonania przebiegów uczących, kiedy to trzeba zgromadzić wiedzę zdobytą przez osobne procesy.

`json_coder(self)` – metoda umożliwiająca serializację do napisu w formacie json.

`json_decoder(self, json)` – metoda umożliwiająca wczytanie wewnętrznego stanu obiektu z obiektu json. Metoda nie sprawdza poprawności obiektu json.

GrammarEstimator

Klasa gromadząca wszystkie statystyki gramatyki. Przechowuje je w postaci mapy nazwa – obiekt `GrammarCriteria` w polu `criterias`. Udostępnia dostęp do tych statystyk przez nazwę, interfejs do dodawania kolejnych obiektów `EvolutionStepEstimation` oraz operację dodawania obiektów `GrammarEstimator`.

`criterias` – pole mapujące nazwę statystyki do obiektu `GrammarCriteria`. Domyślnie mapa ta jest zainicjowana statystykami z macierzy pomyłek, można ją dowolnie zmodyfikować.

`_add_(self, other)` – funkcja umożliwiająca dodawanie obiektów `GrammarEstimator`, szczególnie przydatna przy asynchronicznym wykonywaniu przebiegów uczących.

`json_coder(self)` – metoda umożliwiająca serializację do napisu w formacie json.

`json_decoder(self, json)` - metoda umożliwiająca wczytanie wewnętrznego stanu obiektu z obiektu json. Metoda nie sprawdza poprawności obiektu json.

`get_item_(self, item)` – metoda wbudowana wprowadzająca poniższy cukier syntaktyczny:

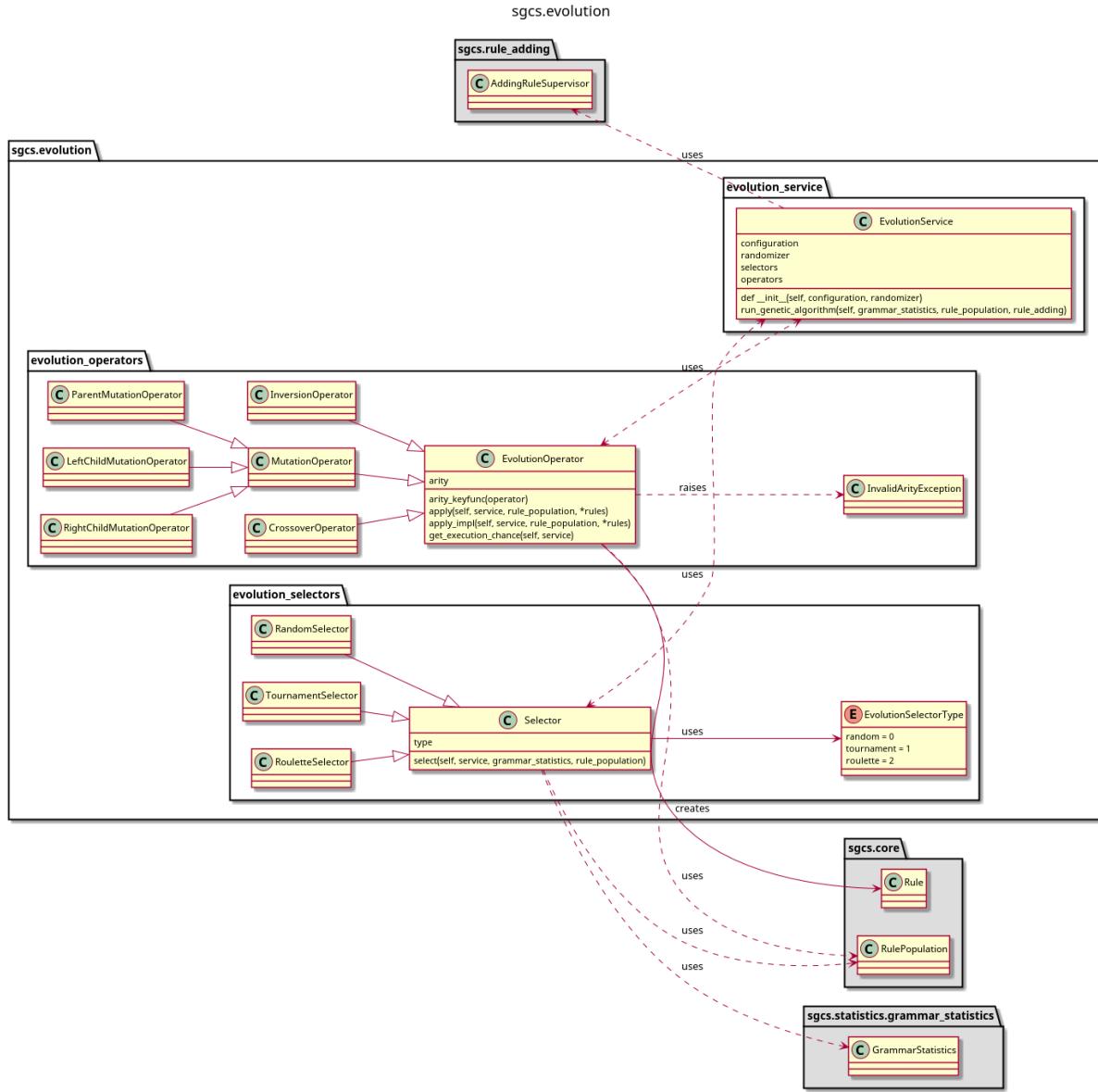
```
fitness_criteria = grammar_criteria['fitness']
```

Który tak naprawdę oznacza:

```
fitness_criteria = grammar_criterias.criterias['fitness']
```

append_step_estimation(self, step_estimation) – metoda dodająca obiekt EvolutionStepEstimator step_estimation do wszystkich statystyk w criterias (poprzez zwołanie na nich metody update).

sgcs.evolution (Rysunek 67)



Rysunek 67: Moduł `sgcs.evolution`

Moduł ewolucji zawiera całą logikę algorytmu genetycznego wykorzystywanego w algorytmie. Algorytm genetyczny można było w łatwy sposób podmienić na jakiś inny algorytm ewolucyjny, obecnie jednak moduł posiada jedynie implementację algorytmu genetycznego. Moduł ten ma na celu wygenerować nowe reguły na podstawie istniejącej populacji i jej stanu (przechowywanego w `grammar_statistics`), a następnie przekazać wygenerowane reguły do modułu `sgcs.rule_adding`.

EvolutionSelector

Klasa abstrakcyjna reprezentująca pojedynczy selektor.

type(self) – metoda zwracająca typ selektora (niezbędne do zgrania z węzłami konfiguracyjnymi algorytmu).

select(self, service, grammar_statistics, rule_population) – abstrakcyjna metoda dokonująca selekcji jednej reguły z `rule_population`, przy pomocy wiedzy otrzymanej z `grammar_statistics` oraz `service` (`EvolutionService`).

RandomSelector

Klasa dziedzicząca po EvolutionSelector, realizująca logikę selekcji losowej.

TournamentSelector

Klasa dziedzicząca po EvolutionSelector, realizująca logikę selekcji turniejowej.

RouletteSelector

Klasa dziedzicząca po EvolutionSelector, realizująca logikę selekcji ruletkowej.

EvolutionOperator

Abstrakcyjna klasa realizująca logikę operatora ewolucyjnego. Zdecydowano się tutaj na wysoce abstrakcyjną definicję operatora jako funkcję przyjmującą n reguł na wejściu i zwracającą k nowych reguł na wyjściu, co umożliwia dodawanie praktycznie dowolnych operatorów genetycznych oraz czyni model algorytmu genetycznego wysoce jednolitym. Szczegóły na temat wykorzystania operatorów o nietypowej arity i k-krotności wartości zwracanej zostaną podane przy okazji omówienia klasy EvolutionService.

arity – arność, czyli liczba argumentów przyjmowanych na wejściu przez algorytm.

arity_keyfunc – metoda zwracająca wartość pola arity, wygodna w użyciu w funkcjach sortujących.

apply(self, service, rule_population, *rules) – metoda wywołująca operator genetyczny na podanych regułach. Jeżeli len(rules) != self.arity wówczas zostaje zgłoszony wyjątek InvalidArityException informujący o nieprawidłowej arności wywołania operatora genetycznego (operator musi być zawsze wywoływany dokładnie z tyloma argumentami, ile wynosi jego arność). Następnie metoda sprawdza, czy operator powinien zostać zaaplikowany (każdy operator genetyczny jest aplikowany z pewnym prawdopodobieństwem). Jeżeli tak zwraca wynik działania metody apply_impl, w przeciwnym przypadku zwraca niezmodyfikowane reguły.

apply_impl(self, service, rule_population, *rules) – abstrakcyjna metoda skupiająca się na wykonaniu samej operacji genetycznej.

get_execution_chance(self, service) – metoda zwracająca wartość prawdopodobieństwa zaaplikowania danego operatora (zazwyczaj wartość ta jest odczytywana z odpowiedniego pola service.configuration.operators).

CrossoverOperator

Binarny operator genetyczny krzyżownia.

InversionOperator

Unarny operator genetyczny inwersji.

MutationOperator

Abstrakcyjny unarny operator genetyczny mutacji. Jest ukonkretniony przez trzy klasy, które zajmują się osobno mutacją poszczególnych wartości reguły (tj. każdy z nich traktujemy jako niezależny operator, wykonując niezależne losowanie czy dana mutacja ma być zaaplikowana itd.).

ParentMutationOperator

Unarny operator genetyczny mutacji ojca reguły.

LeftChildMutationOperator

Unarny operator genetyczny mutacji lewego dziecka reguły.

RightChildMutationOperator

Unarny operator genetyczny mutacji prawego dziecka reguły.

EvolutionService

Klasa odpowiedzialna za obsługę całego algorytmu genetycznego. Działanie algorytmu genetycznego odbywa się w następujących krokach

1. Wybierz x reguł z powtórzeniami z populacji, gdzie x to liczba podanych w konfiguracji selektorów przez użytkownika (mogą być to selektory różnego rodzaju) i zapisz je jako selected_rules.
2. Wybierz elitę populacji elite (jeżeli elityzm jest włączony).
3. Zgrupuj operatory względem arności.
4. Dla każdej dostępnej arności n:
 1. Dla każdego dostępnego operatora op o arności n i krotności wartości zwracanej k:

- Podziel selected_rules na krotki długości n (powtarzając początek selected_rules w ostatniej krotce jeżeli $\text{len}(\text{selected_rules}) \% n \neq 0$).

1. Zaaplikuj op do wszystkich krotek, listy k wyników zsumuj i zapisz jako nowe selected_rules.

- Dodaj selected_rules z elitymem do populacji przy pomocy instancji klasy AddingRuleSupervisor z modułu sgcs.rule_adding.

Grupowanie operatorów jest stabilne względem listy operators (tak więc kolejność dwóch operatorów o tej samej arności na liście nie ulegnie zmianie).

configuration - uchwyt do węzła konfiguracyjnego.

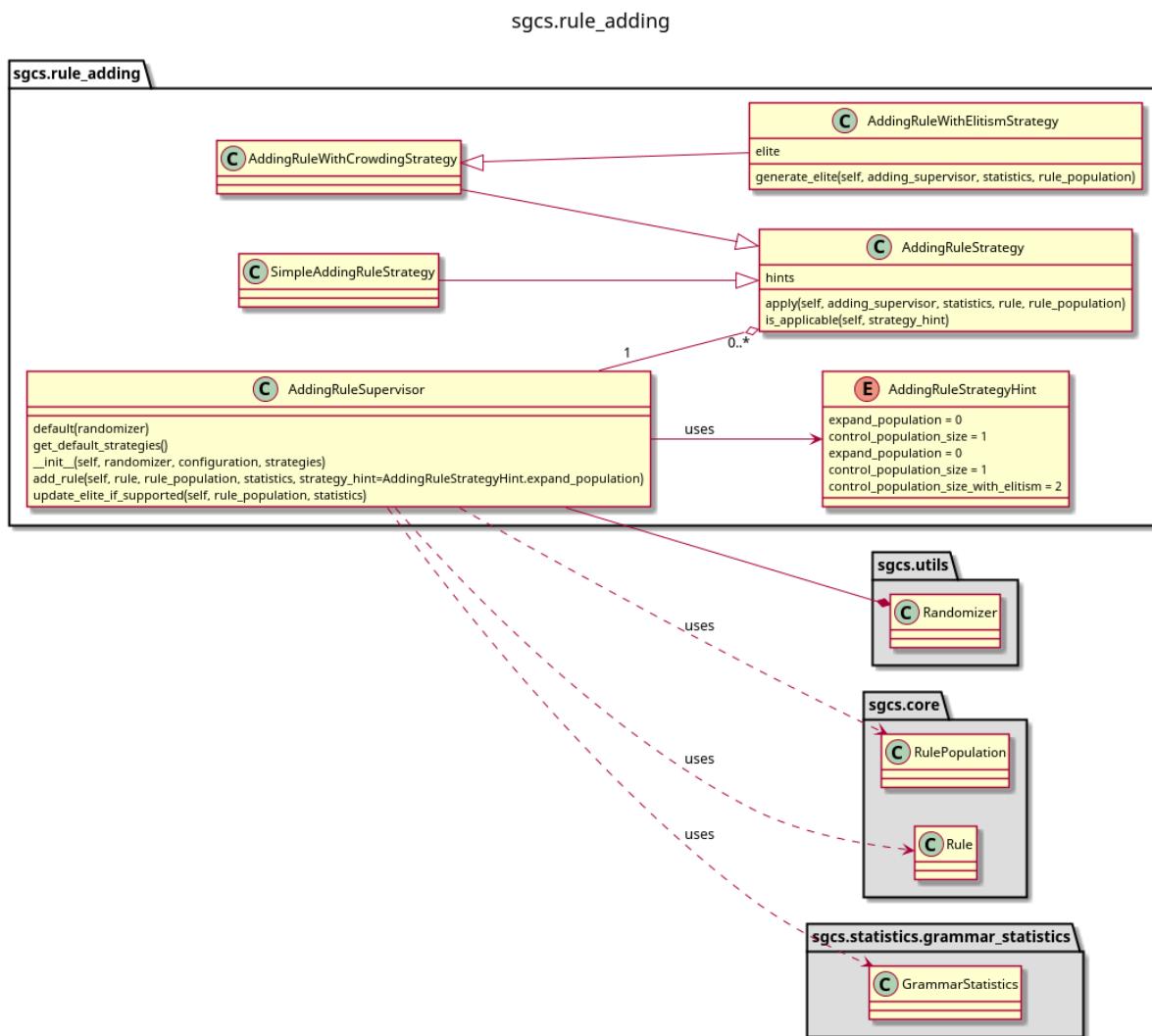
randomizer – uchwyt do instancji klasy Randomizer.

selectors – lista wspieranych selektorów (o rzeczywiście wykorzystanych decyduje konfiguracja).

operators – lista operatorów genetycznych.

run_genetic_algorithm(self, grammar_statistics, rule_population, rule_adding, configuration) – Metoda uruchamiająca algorytm genetyczny.

sgcs.rule_adding (Rysunek 68)



Rysunek 68: Moduł sgcs.rule_adding

Zadaniem tego modułu jest nadzorowanie dodawania nowych reguł do populacji. Praktycznie każde dodawanie nowej reguły odbywa się za pośrednictwem tego modułu.

AddingRuleSupervisor

Klasa odpowiedzialna za zarządzanie dodawaniem reguł.

default(randomizer) – funkcja tworzącainstancję o domyślnej konfiguracji.

get_default_strategies() - funkcja zwracająca listę zawierającą domyślny zestaw strategii.

add_rule(self, rule, rule_population, statistics, strategy_hint = AddingRuleStrategyHint.expand_population) – metoda powodująca dodanie reguły do populacji z uwzględnieniem strategy_hint. Należy tutaj zwrócić uwagę na słowo „hint” - AddingRuleSupervisor może nadpisać to zachowanie w jednym z przypadków:

- Jeżeli strategy_hint == AddingRuleStrategyHint.expand_population, a osiągniemy już maksymalny rozmiar populacji, wówczas będzie miało dodanie reguły z AddingRuleStrategyHint.control_population_size;
- Jeżeli strategy_hint wymuszałby dodanie reguły do pustej populacji bez dokonywania zmiany jej rozmiaru, wówczas supervisor doda populację, zwiększając tym samym jej rozmiar (do 1).

Pomijając te wyjątkowe sytuacje AddingRuleSupervisor znajdzie obiekt odpowiedniej strategii i z jej pomocą doda regułę do populacji.

update_elite_if_supported(self, rule_population, statistics) – funkcja aktualizująca stan elity przechowywanej przez funkcję. Metoda ta musi być wywoływana osobno, gdyż w przypadku dodawania kilku nowych reguł elita powinna być sprecyzowana przed dodaniem jakiekolwiek z nich, nie powinna być generowana przy każdym dodaniu reguły.

AddingRuleStrategyHint

Klasa ta pełni funkcję wyliczenia z następującymi możliwymi wartościami:

- expand_population;
- control_population_size;
- control_population_size_with_elitism.

AddingRuleStrategy

Abstrakcyjna klasa definiująca podstawową logikę funkcjonowania strategii dodawania reguł.

hints – lista AddingRuleStrategyHint, w przypadku których strategia jest aplikowalna.

apply(self, adding_supervisor, statistics, rule, rule_population) – metoda abstrakcyjna powodująca zaaplikowanie danej strategii.

is_applicable(self, strategy_hint) – metoda zwracająca True, jeżeli daną strategię da się zastosować z danym strategy_hint.

SimpleAddingRuleStrategy

Najprostsza strategia dodawania reguł – poprzez rozszerzenie zbioru. Reguła zostaje dodana, obiekt statistics poinformowany o zmianach.

AddingRuleWithCrowdingStrategy

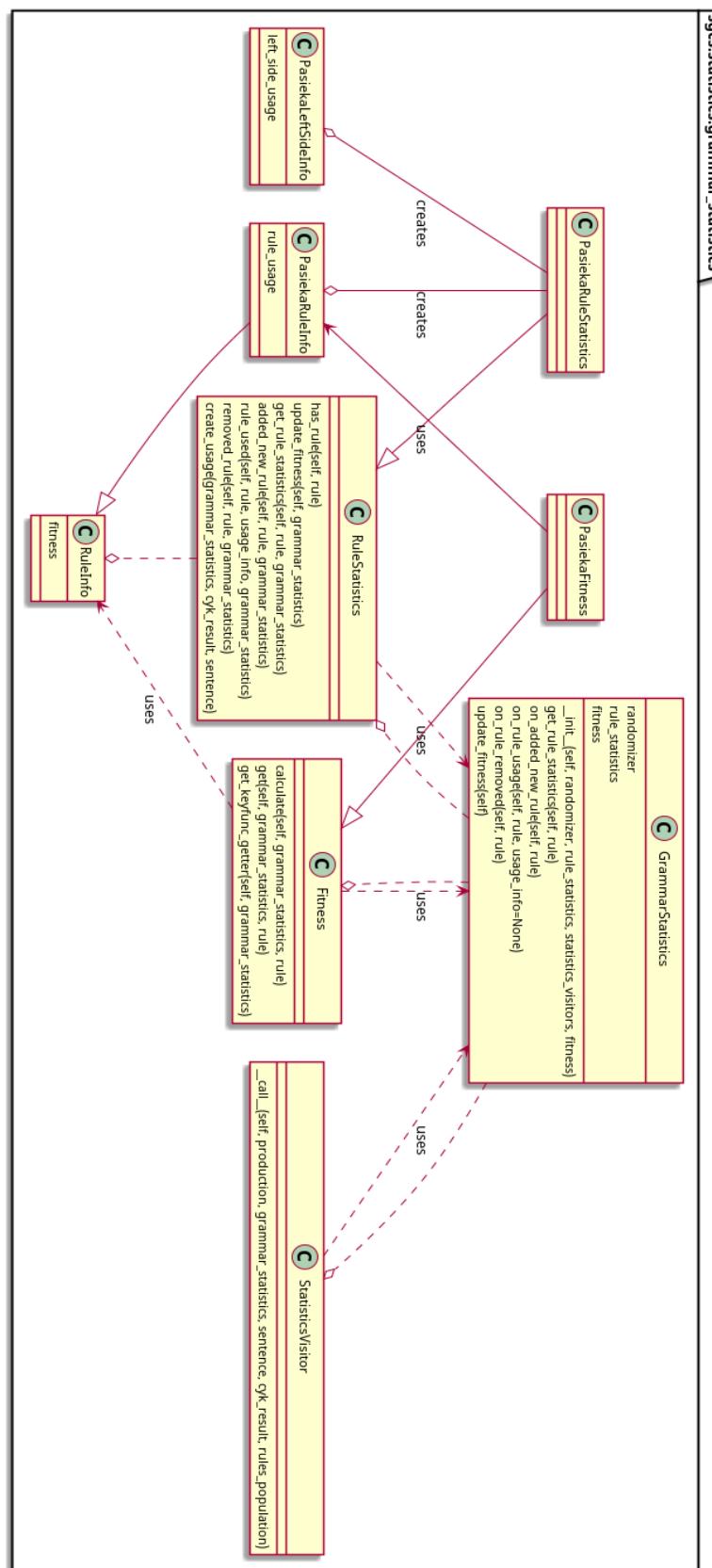
Strategia wykorzystująca mechanizm ścisłu do dodawania reguł.

AddingRuleWithElitismStrategy

Strategia wykorzystująca mechanizm ścisłu oraz elityzm do dodawania reguł.

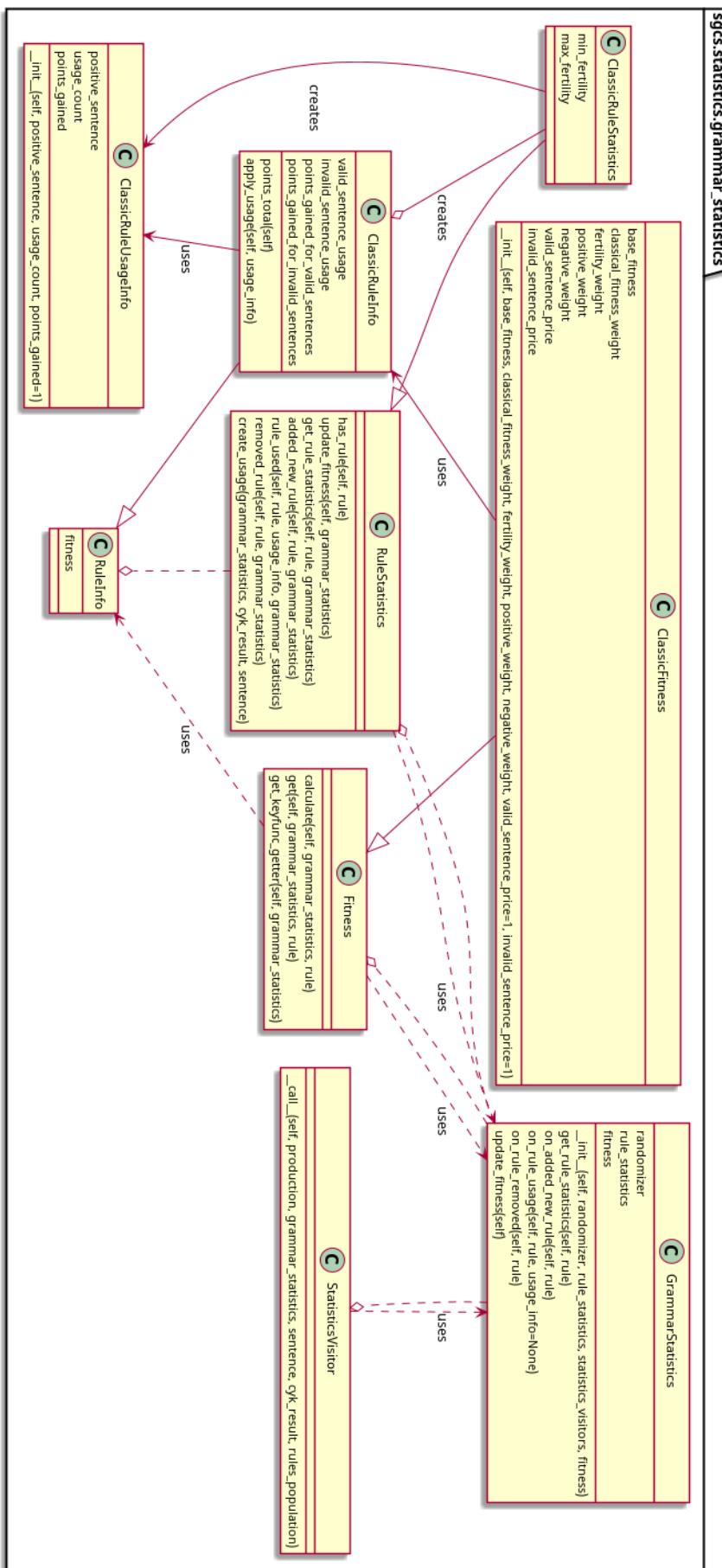
generate_elite(self, adding_supervisor, statistics, rule_population) – funkcja generująca elitę. Nie trzeba jej ręcznie wywoływać – zostanie automatycznie wywołana przez metodę AddingRuleSupervisor.update_elite_if_supported.

sgcs.statistics.grammar_statistics (Rysunek 69 oraz Rysunek 70)



Rysunek 69: Moduł sgcs.statistics.grammar_statistics (tylko klasy przystosowania Pasieki)

sgcs.statistics.grammar_statistics (hidden dependencies, no Pasieka)



Rysunek 70: Moduł `sgcs.statistics.grammar_statistics` (tylko klasy przystosowania klasycznego)

Powyższe dwa diagramy dopiero po połączeniu dają całkowity obraz modułu sgcs.statistics.grammar_statistics. Niestety nawet bez dołączania zewnętrznych zależności modułu jest to obraz wysoce nieczytelny. Dlatego też zdecydowano się na przedstawienie modułu w postaci dwóch diagramów – na jednym ukryto klasy związane z funkcją przystosowania Pasieka, na drugim ukryto klasy związane z klasyczną funkcją przystosowania. Obie grupy są od siebie wzajemnie niezależne, natomiast część wspólna obu diagramów została zdublowana.

Moduł ten zajmuje się obliczaniem wartości funkcji przystosowania.

GrammarStatistics

Główna klasa zajmująca się obsługą parametru fitness. Większość jej funkcji oddelegowuje swoją logikę do obiektu rule_statistics, co czyni ją praktycznie fasadą tejże klasy.

randomizer – uchwyt obiektu typu Randomizer.

rule_statistics – instancja obiektu dziedziczącego po abstrakcyjnym RuleStatistics. Klasa ta zajmuje się obsługą większości logiki GrammarStatistics, pozostawiając mu jedynie wspólną dla wszystkich funkcji fitness logikę.

fitness – instancja obiektu dziedziczącego po klasie Fitness.

statistics_visitors – lista instancji obiektu StatisticsVisitor. Jest on na tyle generyczny, że bez względu na liczbę i rodzaj obsługiwanych funkcji fitness nie ma potrzeby umieszczania na niej więcej niż jednego obiektu tego typu – inne obiekty mogłyby być przydatne przy zastosowaniu naprawdę niestandardowych funkcji przystosowania (patrz opis klasy StatisticsVisitor).

default(randomizer, configuration) – funkcja tworzącainstancję GrammarStatistics przygotowaną pod względem liczenia funkcji klasycznej.

sgcs_variant(randomizer, configuration) – funkcja tworzącainstancję GrammarStatistics przygotowaną pod względem liczenia funkcji Pasieka.

get_rule_statistics(self, rule) – metoda zwracająca statystyki reguły rule (postać zależna od implementacji statystyk).

on_added_new_rule(self, rule) – metoda obsługująca utworzenie nowej reguły.

on_rule_usage(self, rule_usage, usage_info=None) – metoda wywoływana, gdy reguła zostanie użyta. Można dołączyć opcjonalny parametr UsageInfo (umożliwiający zbieranie dodatkowych informacji przez bardziej skomplikowane funkcje fitness jak na przykład funkcja klasyczna).

on_rule_removed(self, rule) – metoda obsługująca usunięcie reguły.

update_fitness(self) – przeliczenie funkcji fitness na podstawie obecnie zebranych statystyk.

RuleStatistics

Klasa abstrakcyjna, której konkretyzacje realizują większość logiki GrammarStatistics. Posiada ona dwie dodatkowe metody:

has_rule(self, rule) – zwraca True, jeżeli reguła rule już istnieje.

create_usage(grammar_statistics, cyk_result, sentence) – tworzy obiekt UsageInfo, jeżeli jest on wspierany przez funkcję fitness. Obiekty te nie mają określonego z góry interfejsu, jako że są to obiekty służące właściwie do przekazywania danych typowych wyłącznie dla poszczególnych funkcji fitness i to one są ich producentem i konsumentem. Jeżeli nie ma potrzeby wykorzystania takiego obiektu, zwrócona zostanie wartość None.

RuleInfo

Klasa przechowująca informacje niezbędne do wyliczenia wartości fitness (oraz obecną wartość tegoż parametru).

StatisticsVisitor

Obiekt ten powstał w celu bycia wykorzystanym przez algorytm traceback. W chwili kiedy zostanie wywołany jak funkcja, w odpowiedni sposób zapisze informację o wykorzystaniu reguły. Ukreście logiki liczenia funkcji fitness czyni go wysoce uniwersalnym, dzięki czemu wydaje się, że raczej nie będzie konieczności dodawania kolejnych wizytatorów (przy założeniu, że funkcje przystosowania nie zaczynają wymagać wysoce specyficznych danych).

Fitness

Abstrakcyjna klasa reprezentująca sam wzór na funkcję przystosowania.

calculate(self, grammar_statistics, rule) – abstrakcyjna metoda licząca wartość funkcji przystosowania.

`get(self, grammar_statistics, rule)` – metoda zwracająca wartość funkcji przystosowania. Jeżeli wartość ta nie została jeszcze nigdy policzona, wówczas wywołanie tej metody spowoduje te obliczenia.

`get_keyfunc_getter(self, grammar_statistics)` – metoda zwracająca delegata funkcji get, który jest wygodny w zastosowaniu w wyrażeniach funkcyjnych (jak na przykład polecenie map()).

Classic__

Funkcja klasyczna wykorzystuje szereg różnorodnych klas w celu wyliczenia wartości fitness. Są to:

- ClassicRuleStatistics;
- ClassicRuleInfo;
- ClassicRuleUsageInfo;
- ClassicFitness.

Klasy te nie wymagają większego komentarza – są ukonkretionymi klasami omówionych klas abstrakcyjnych, mających za zadanie obsłużyć liczenie klasycznej funkcji fitness. Warto tutaj jedynie zauważyc fakt istnienia klasy ClassicRuleUsageInfo – w klasie tej są zbierane informacje jak punktów zdobytych przez regułę, czy zdanie było pozytywne oraz pozostałe informacje wymagane przez klasyczną funkcję przystosowania.

Pasieka__

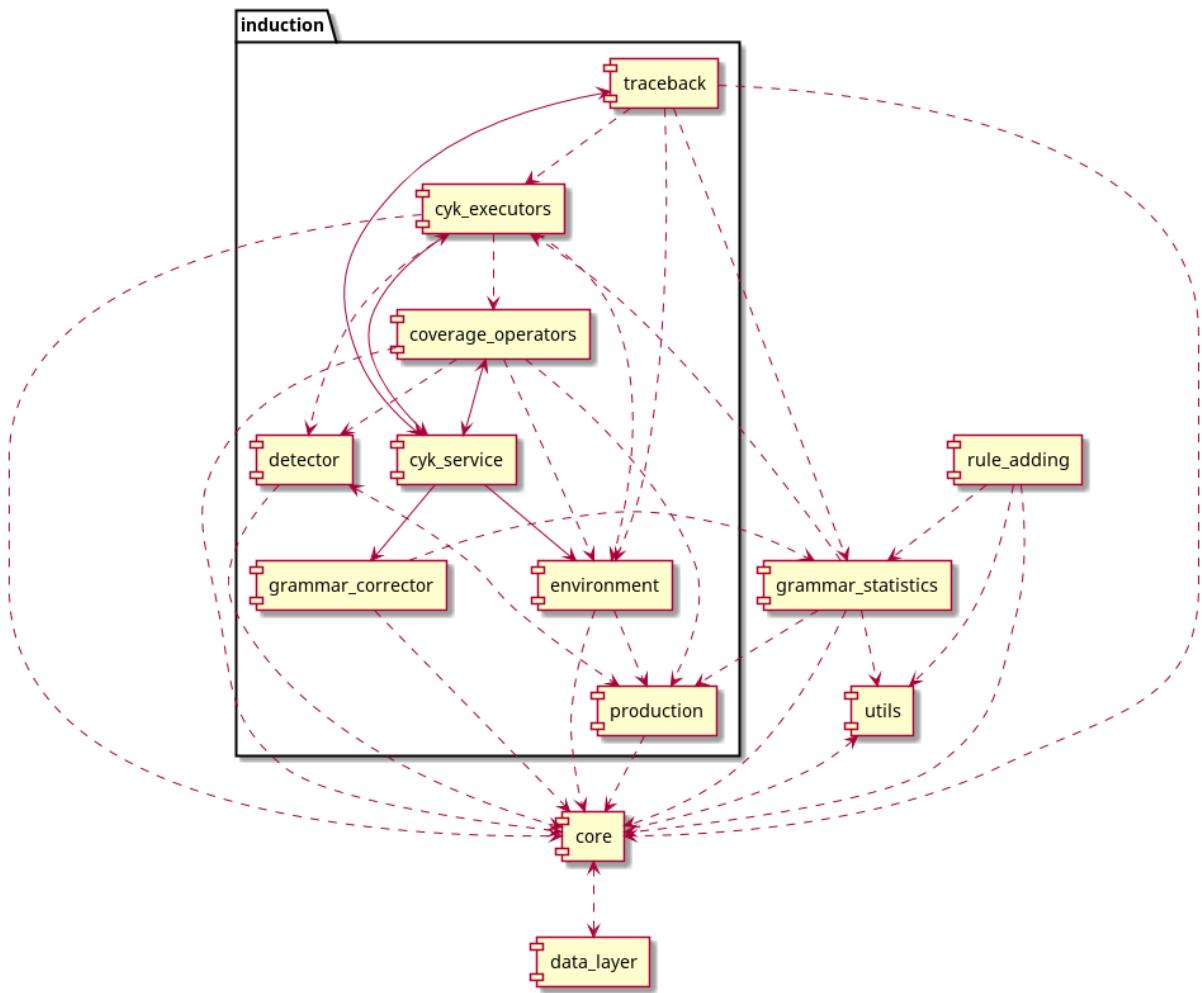
Funkcja Pasieka również posiada swój zestaw klas niezbędnych do wyliczenia wartości przystosowania. Są to:

- PasiekaRuleStatistics;
- PasiekaFitness;
- PasiekaRuleInfo;
- PasiekaLeftSideInfo.

PsiekaLeftSideInfo gromadzi informacje na temat wykorzystania reguł o Symbolu lewej strony produkcji, trzymanie tej informacji jest niezbędne i nie ma sensu jej duplikować w każdej regule posiadającej ten symbol w lewej części produkcji. Pokazuje to równocześnie uniwersalność modułu, który może ukryć w ten sposób pod omówionymi wcześniej klasami spore fragmenty nietrywialnej mechaniki, nie wymagając przy tym samym modyfikacji interfejsu.

Właściwy algorytm – część indukcyjna

sgcs.induction dependency diagram



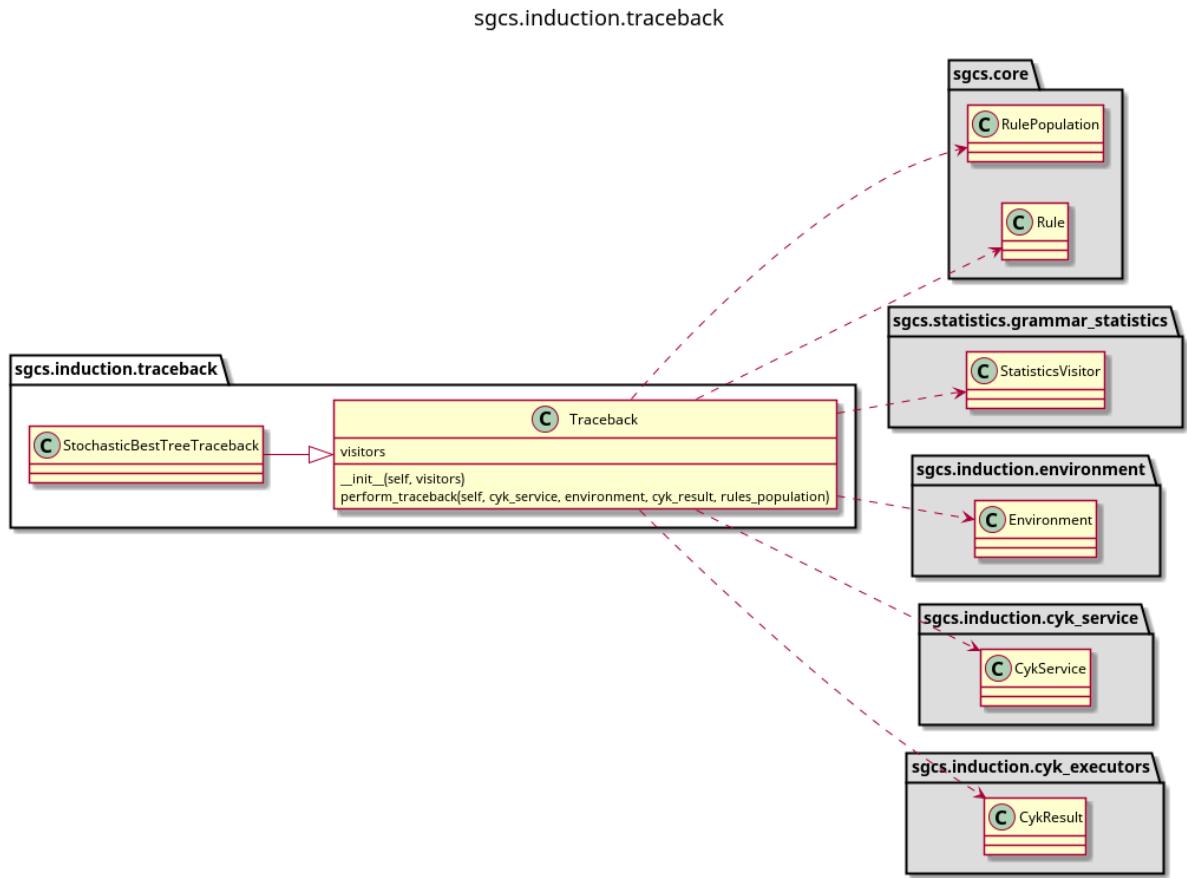
Rysunek 71: Diagram zależności modułów przestrzeni sgcs.induction

Powyżej (Rysunek 71) znajduje się diagram zależności modułów indukcyjnej części algorytmu genetycznego. Jest to oczywiście najważniejsza i najbardziej złożona część algorytmu. Wszystkie moduły związane z indukcją gramatyki umieszczone w przestrzeni nazw sgcs.induction.

sgcs.induction.cyk_service

Moduł ten zawiera dwie klasy – CykService oraz dziedziczącą po niej StochasticCykService. Klasy te gromadzą obiekty z całego modułu indukcji i składają je w jedną, dostępną dla wszystkich składowych strukturę, w zależności od wykorzystanej klasy struktura ta jest nastawiona na indukcję gramatyk stochastycznych lub klasycznych. Znajduje się tu też główna pętla indukcji gramatyki.

`sgcs.induction.traceback` (Rysunek 72)



Rysunek 72: Moduł `sgcs.induction.traceback`

Moduł ten zawiera klasy implementujące różne warianty algorytmu traceback. Wykorzystują one wzorzec wizytatora – traceback zapewnia jedynie spójny interfejs do poruszania się po drzewie rozbiór zdania, zaś sam wizytator decyduje w jaki sposób wykorzysta zebrane podczas wędrówki dane. Pozwala to na separację systemu nagradzania odwiedzonych węzłów od samej metody poruszania się po drzewie rozbioru.

Traceback

Standardowa implementacja algorytmu traceback wykorzystywana przez algorytm GCS. Algorytm ten gorliwie przechodzi po wszystkich możliwych drzewach rozkładu zdania.

visitors – lista wizytatorów. Wizytatorzy muszą mieć następującą sygnaturę:

1. Aktualnie odwiedzana produkcja.
2. Obiekt GrammarStatistics.
3. Obiekt Sentence (reprezentacja całego zdania).
4. Obiekt CykResult.
5. Obiekt RulePopulation.

Kolejność tych parametrów ma znaczenie.

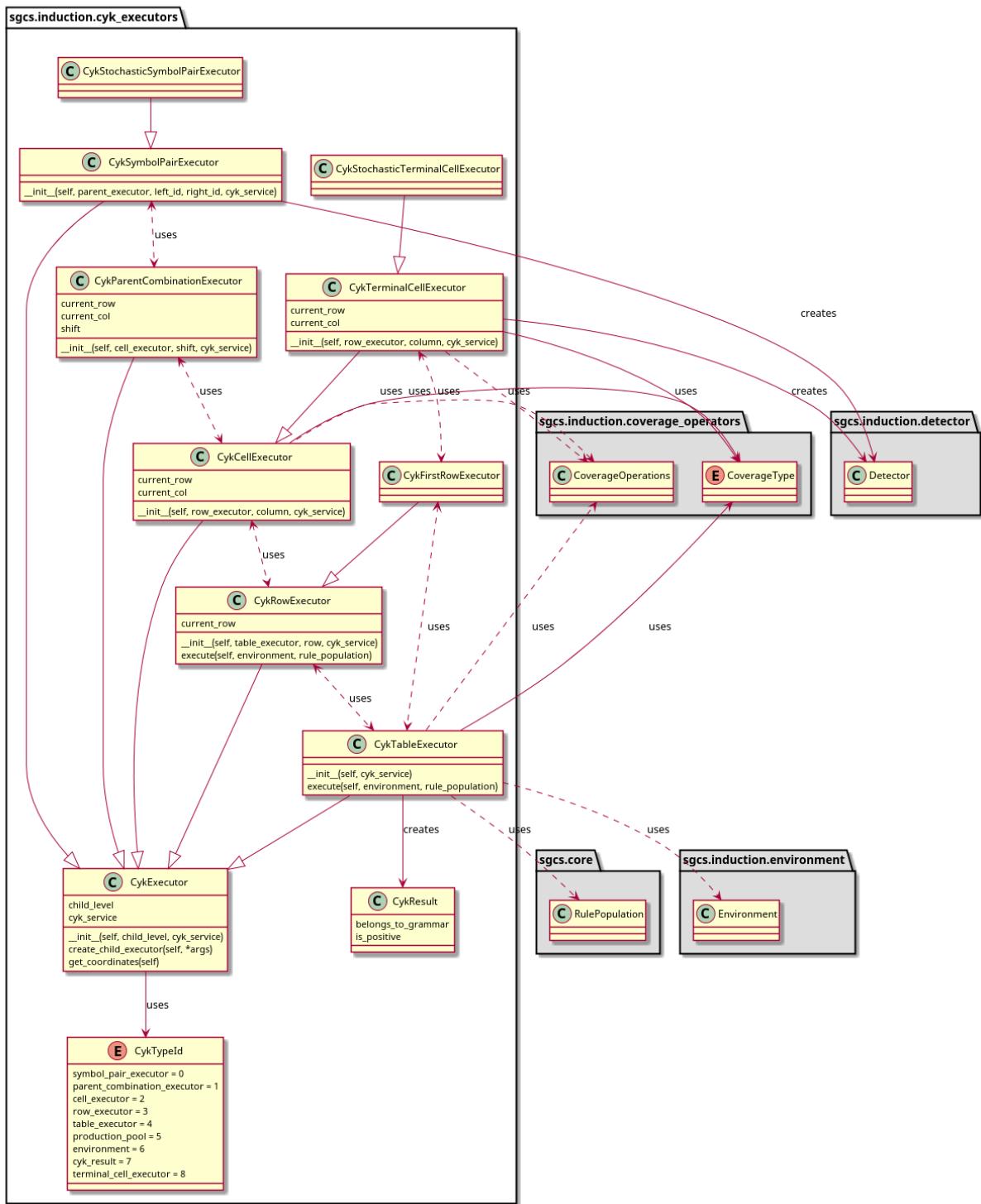
`perform_traceback(self, cyk_service, environment, cyk_result, rules_population)` – metoda powodująca wywołanie algorytmu traceback. Obiekt environment stanowi reprezentację tablicy CYK, `cyk_result` zawiera między innymi werdykt przydzielony mu przez algorytm oraz informację o tym, czy zdanie w rzeczywistości należy do gramatyki, czy też nie.

StochasticBestTreeTraceback

Implementacja algorytmu traceback wykorzystywana przez algorytmy stochastyczne. Algorytm ten przechodzi jedynie po najbardziej prawdopodobnym drzewie rozkładu zdania. Dziediczy po klasie Traceback i nie rozszerza istniejącego interfejsu.

`sgcs.induction.cyk_executors` (Rysunek 73)

`sgcs.induction.cyk_executors`



Rysunek 73: Moduł `sgcs.induction.cyk_executors`

Moduł `sgcs.induction.cyk_executors` jest rozbity na czynniki pierwsze pętlą algorytmu CYK. Kolejni wykonawcy skupiają się na coraz głębszych elementach algorytmu CYK, wszyscy są tworzeni przez odpowiednio przygotowaną wcześniej fabrykę. Takie rozdrobnienie tego kluczowego miejsca algorytmu umożliwia łatwe modyfikowanie funkcjonowania procesu indukcji.

CykTypeId

Wyliczenie przedstawiające identyfikatory typów klas, z jakimi mamy do czynienia w module. Jest ono wykorzystywane przez fabrykę obiektów, w efekcie czego żadna z klas powiązanych z poniższymi wartościami nie jest tworzona w inny sposób niż przy pomocy fabryki. Domyślnie moduł ten promuje następujący podział klas (który może zostać rozszerzony, jeżeli zmienimy zachowanie fabryki):

1. `table_executor` – klasa wykorzystywana na najwyższym poziomie, odpowiada za wypełnianie całej tablicy CYK.

2. row_executor – klasa odpowiadająca za uzupełnienie całego wiersza tablicy CYK.
3. cell_executor – klasa odpowiedzialna za uzupełnienie pojedynczej komórki tabeli CYK.
4. parent_combination_executor – klasa odpowiedzialna za wpisanie do komórki tabeli CYK reguł powstały z reguł z pojedynczej kombinacji komórek – rodziców.
5. symbol_pair_executor – najniższy poziom uzupełniania tabeli CYK, generująca reguły powstałe z pojedynczej pary symboli.
6. environment – klasa zarządzająca zawartością tabeli CYK.
7. production_pool – klasa zarządzająca zawartością pojedynczej komórki tabeli CYK.
8. cyk_result – klasa niosąca informacje o rezultacie parsowania (oraz o prawdziwej przynależności zdania).

Klasy oznaczone identyfikatorami typów 1-5 są klasami wykonawców (uporządkowanych malejąco względem ich odpowiedzialności). Klasa bezpośrednio kontrolowana przez wykonawcę nadziedzkiego będzie od tej pory nazywana „podwykonawcą”.

CykResult

Prosta klasa przechowująca wynik parsowania zdania przy pomocy algorytmu CYK oraz prawdziwą przynależność zdania.

belongs_to_grammar – zmienna boolowska ustalająca czy zdanie rzeczywiście należy do gramatyki.

is_positive – zmienna boolowska stwierdzająca czy CYK zdołał sparsować zdanie.

CykExecutor

Abstrakcyjna klasa dostarczająca trochę podstawowej logiki klasom wykonawców. Zawiera uchwyt do instancji CykService, jest w stanie stworzyć podwykonawcę oraz podać swoje koordynaty w tabeli CYK.

child_level – identyfikator podwykonawcy.

cyk_service – uchwyt instancji CykService.

create_child_executor(self, *args) – metoda zwracająca obiekt podwykonawcy. Musimy wiedzieć dokładnie jakie parametry są wymagane przez konstruktor.

get_coordinates(self) – metoda zwracająca koordynaty wykonawcy w tabeli CYK.

CykTableExecutor

Klasa najwyższego poziomu algorytmu CYK. Jest w stanie ustalić, czy zdane zostało sparsowane przez tabelę CYK, może uruchomić operatory pokrycia zarejestrowane na CoverageType.no_starting_symbol oraz zlecić podwykonawcom wypełnianie wierszy tabeli CYK. Jej podwykonawcy to CykFirstRowExecutor (dla pierwszego wiersza) oraz CykRowExecutor (dla kolejnych).

init (self, cyk_service) – konstruktor.

execute(self, environment, rule_population) – metoda wywołująca algorytm CYK.

CykRowExecutor

Podwykonawca klasy CykTableExecutor. Przeprowadza algorytm CYK na poziomie pojedynczego wiersza, przeznaczony do wszystkich wierszy powyżej pierwszego). Jego podwykonawcą jest CykCellExecutor.

current_row – zwraca numer obecnego wiersza.

init (self, table_executor, row, cyk_service) – konstruktor. Przekazywanie siebie podwykonawcy często ma miejsce w klasach wykonawców.

CykFirstRowExecutor

Klasa uzupełniająca pierwszy wiersz tabeli CYK. Dziedziczy po CykRowExecutor. Jej podwykonawcą jest CykTerminalStateExecutor lub CykStochasticTerminalCellExecutor
(w przypadku algorytmu stochastycznego).

CykCellExecutor

Klasa uzupełniająca pojedynczą komórkę (powyżej pierwszego wiersza). Może wywołać operatory pokrycia zarejestrowane na CoverageType.no_effector_type oraz zlecić podwykonawcom wpisywanie przodukcji powstałych z konkretnych par komórek – rodziców. Jej podwykonawcą jest CykParentCombinationExecutor.

current_col – zwraca numer obecnej kolumny.

CykCellTerminalExecutor

Klasa uzupełniająca pojedynczą komórkę pierwszego wiersza tabeli CYK. W przypadku nieznalezienia żadnych produkcji może wywołać operatory pokrycia zarejestrowane na typ pokrycia CoverageType.unknown_terminal_symbol. Nie posiada podwykonawcy.

CykStochasticTerminalCellExecutor

Klasa dziedzicząca po CykCellTerminalExecutor, uzupełniająca dodatkowo prawdopodobieństwa produkcji.

CykParentCombinationExecutor

Klasa tworząca wszystkie produkcje powstałe z jednej pary komórek – rodziców. Jej podwykonawcą jest CykSymbolPairExecutor lub CykStochasticSymbolPairExecutor w przypadku gramatyk stochastycznych.

shift – koordynat przesunięcia w tabeli CYK – określa jednoznacznie koordynaty komórek rodziców.

CykSymbolPairExecutor

Klasa tworząca produkcje powstałe z pojedynczej kombinacji symboli. Nie ma podwykonawcy.

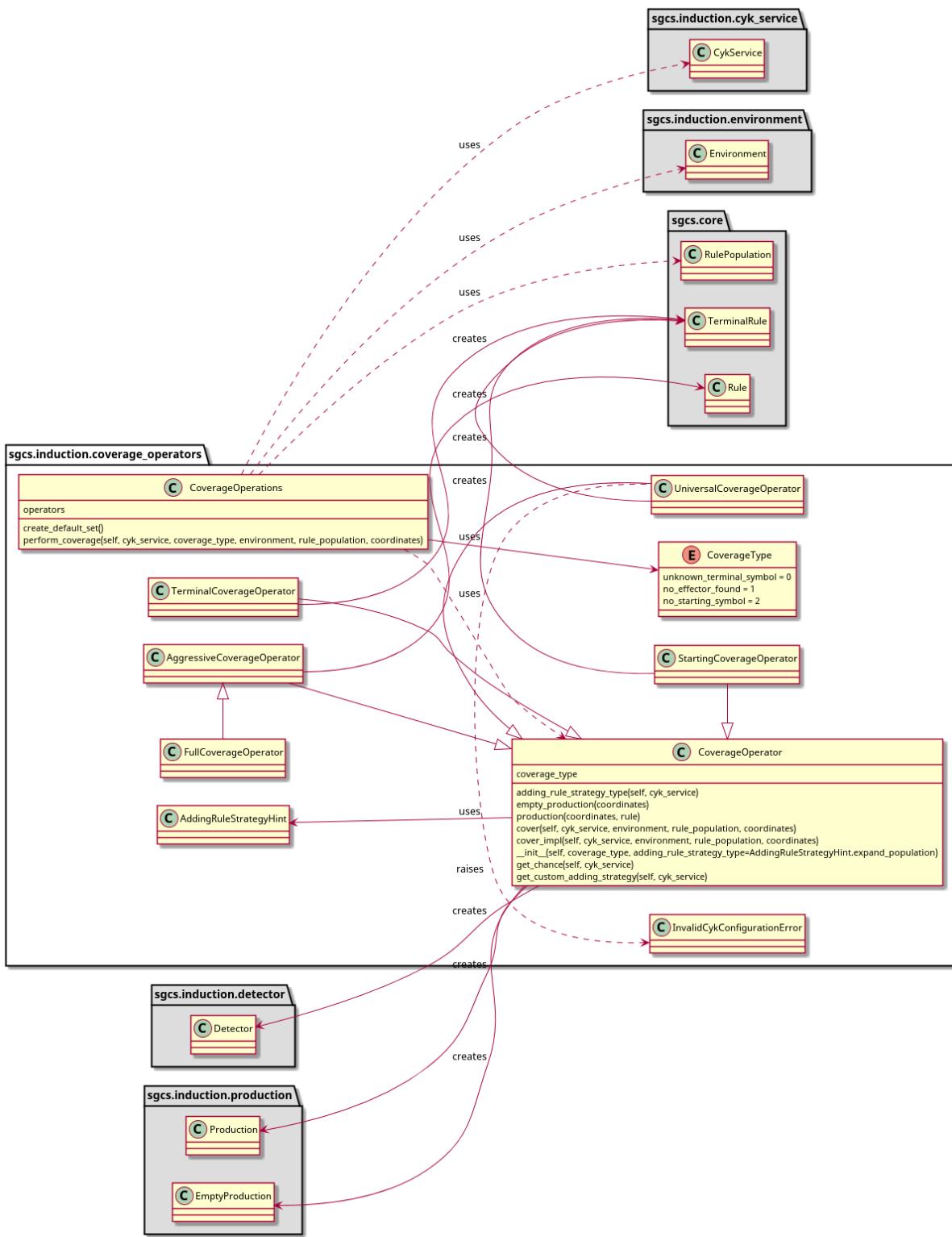
left_id – koordynat lewego symbolu – jednoznacznie określa symbol z lewej komórki odpowiedzialny za tworzone produkcje.

right_id – koordynat prawego symbolu – jednoznacznie określa symbol z prawej komórki odpowiedzialny za tworzone produkcje.

CykStochasticSymbolPairExecutor

Klasa rozszerzająca CykSymbolPairExecutor o aktualizację prawdopodobieństw produkcji.

sgcs.induction.coverage_operators (Rysunek 74)



Rysunek 74: Moduł sgcs.induction.coverage_operators

Moduł sgcs.induction.coverage_operators dostarcza implementację standardowych operatorów pokrycia. Zostały one oddzielone od modułu sgcs.induction.cyk_executors przy pomocy identyfikatorów typu pokrycia CoverageType. Kiedy w głównej pętli algorytmu następuje potrzeba uruchomienia operatorów pokrycia, wówczas jest wywoływana metoda CoverageOperations.perform_coverage z odpowiednim identyfikatorem typu pokrycia, na który zareagują wyłącznie operatory pokrycia zarejestrowane na ten typ. Na przykład przy natrafieniu na nieznany symbol nieterminalny jest wykorzystywany identyfikator CoverageType.unknown_terminal_symbol, na który reagują odpowiednio operatory pokrycia terminalnego oraz uniwersalnego. Umożliwia to łatwe rozszerzanie algorytmu o nowe operatory pokrycia bez potrzeby wprowadzania modyfikacji do głównej pętli algorytmu, zmiany dotyczące operatorów pokrycia dokonuje się wyłącznie w module

operatorów pokrycia. Po stworzeniu nowych reguł są one dodawane do populacji reguł z pomocą klasy AddingRuleSupervisor. Operatory pokrycia posiadają domyślne strategie dodawania reguł (na przykład operator pokrycia agresywnego zawsze dodaje reguły ze ściskiem), aczkolwiek można nadpisać to zachowanie przy pomocy odpowiedniego ustawienia węzła konfiguracyjnego tego operatora.

W celu ujednolicenia modelu wszystkie operatory posługują się prawdopodobieństwem – oznacza to, że jest możliwe uruchomienie na przykład operatora pokrycia uniwersalnego z pewnym prawdopodobieństwem (zamiast dyskretnie zawsze lub nigdy).

CoverageType

Typ wyliczeniowy przechowujący dostępne rodzaje operatorów pokrycia – są to obecnie unknown_terminal_symbol, no_effector_found oraz no_starting_symbol. CoverageType najłatwiej zrozumieć jako powód, dla którego może być konieczne uruchomienie operatora pokrycia – uruchamiamy je gdy natrafimy na nieznany symbol gramatyki, nie udało się wpisać nic do jednej z komórek tabeli CYK lub komórka nie zawiera symbolu startowego gramatyki.

CoverageOperations

Klasa agregująca operatory pokrycia. Jest zdolna do wygenerowania ich podstawowego zestawu oraz uruchomienia operatorów pokrycia danego typu, kiedy zaistnieje taka potrzeba.

operators – lista operatorów pokrycia

create_default_set() - funkcja zwracająca domyślną listę (czyli zawierającą wszystkie dostarczone wraz z tym modułem) operatorów pokrycia.

perform_coverage(self, cyk_service, coverage_type, environment, rule_population, coordinates) – funkcja ta znajduje wszystkie operatory zarejestrowane na coverage_type i uruchamia dany operator (który ostatecznie zostanie uruchomiony lub też nie, zależnie od prawdopodobieństwa oraz innych dodatkowych warunków wymaganych przez operator). Jeżeli udało się wygenerować nową produkcję, to jest ona dodawana do tabeli CYK, zaś nowa reguła z nią związana dodana z pomocą AddingRuleSupervisor do populacji reguł. Metoda próbuje zachłannie wszystkich operatorów pokrycia.

CoverageOperator

Klasa abstrakcyjna dostarczająca części wspólnej logiki operatorom pokrycia.

coverage_type – powiązany z operatorem identyfikator typu pokrycia.

init (self, coverage_type, adding_rule_strategy_type = AddingRuleStrategyHint.expand_population) – konstruktor, pobierający typ pokrycia operatora oraz domyślną strategię dodawania nowych reguł.

adding_rule_strategy_type(self, cyk_service) – zwraca domyślną dla operatora strategię dodawania nowych reguł, chyba że została ona nadpisana w konfiguracji własną strategią.

empty_production(coordinates) – Funkcja pomocnicza zwracająca obiekt pustej produkcji dla koordynatów coordinates.

production(coordinates, rule) – Funkcja pomocnicza zwracająca obiekt produkcji powiązany z regułą rule i koordynatami coordinates.

cover(self, cyk_service, environment, rule_population, coordinates) – metoda z prawdopodobieństwem wywołania operatora pokrycia zwraca rezultat metody cover_impl (której zachowanie jest zależne od implementacji konkretnego operatora pokrycia, ale spodziewa się od niej zwrotienia nowej produkcji, lub też nie).

cover_impl(self, cyk_service, environment, rule_population, coordinates) – abstrakcyjna metoda, która jeżeli warunki uruchomienia operatora są spełnione (na przykład operator pokrycia startowego próbuje pokryć zdanie należące do gramatyki) wykona odpowiednią operację pokrycia.

get_chance(self, cyk_service) – abstrakcyjna metoda zwracająca prawdopodobieństwo uruchomienia operatora pokrycia (zazwyczaj jest to wartość zapisana w którymś z węzłów konfiguracji).

get_custom_adding_strategy(self, cyk_service) – abstrakcyjna metoda zwracająca podaną własną strategię dodawania reguł (zazwyczaj jest to wartość zapisana w którymś z węzłów konfiguracji).

TerminalCoverageOperator

Klasa reprezentująca operator pokrycia terminalnego. Typ pokrycia tego operatora to CoverageType.unknown_terminal_symbol. Indywidualna strategia dodawania reguł jest zlokalizowana w cyk_service.configuration.coverage.operators.terminal.adding_hint, zaś prawdopodobieństwo w cyk_service.configuration.coverage.operators.terminal.chance. Domyślna strategia dodawania nowych reguł to AddingRuleStrategyHint.expand_population.

UniversalCoverageOperator

Klasa reprezentująca operator pokrycia uniwersalnego. Typ pokrycia tego operatora to CoverageType.unknown_terminal_symbol. Indywidualna strategia dodawania reguł jest zlokalizowana w cyk_service.configuration.coverage.operators.universal.adding_hint, zaś prawdopodobieństwo w cyk_service.configuration.coverage.operators.universal.chance. Domyślna strategia dodawania nowych reguł to AddingRuleStrategyHint.expand_population. Korzystanie z tego operatora wymaga sprecyzowania symbolu uniwersalnego w populacji reguł – jeżeli symbol ten nie jest zdefiniowany, przy próbie użycia operatora zostanie zgłoszony wyjątek InvalidCykConfigurationError (gdzię oznacza to sprzeczność w samej konfiguracji).

StartingCoverageOperator

Klasa reprezentująca operator pokrycia startowego. Typ pokrycia tego operatora to CoverageType.no_starting_symbol. Indywidualna strategia dodawania reguł jest zlokalizowana w cyk_service.configuration.coverage.operators.starting.adding_hint, zaś prawdopodobieństwo w cyk_service.configuration.coverage.operators.starting.chance. Domyślna strategia dodawania nowych reguł to AddingRuleStrategyHint.control_population_size. Operator ten posiada dodatkowy warunek uruchomienia – można go wykorzystać wyłącznie w zdaniu pozytywnym.

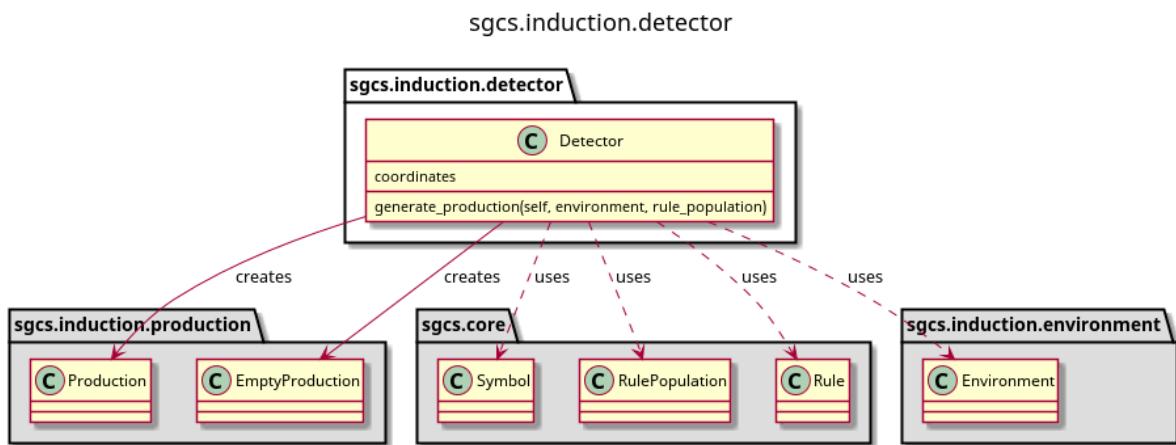
AggressiveCoverageOperator

Klasa reprezentująca operator pokrycia agresywnego. Typ pokrycia tego operatora to CoverageType.no_effector_found. Indywidualna strategia dodawania reguł jest zlokalizowana w cyk_service.configuration.coverage.operators.aggressive.adding_hint, zaś prawdopodobieństwo w cyk_service.configuration.coverage.operators.aggressive.chance. Domyślna strategia dodawania nowych reguł to AddingRuleStrategyHint.control_population_size.

FullCoverageOperator

Klasa reprezentująca operator pokrycia pełnego. Dziedziczy po AggressiveCoverageOperator. Typ pokrycia tego operatora to CoverageType.no_starting_symbol. Indywidualna strategia dodawania reguł jest zlokalizowana w cyk_service.configuration.coverage.operators.full.adding_hint, zaś prawdopodobieństwo w cyk_service.configuration.coverage.operators.full.chance. Domyślna strategia dodawania nowych reguł to AddingRuleStrategyHint.control_population_size.

sgcs.induction.detector (Rysunek 75)



Rysunek 75: Moduł sgcs.induction.detector

Moduł zawierający klasę Detector.

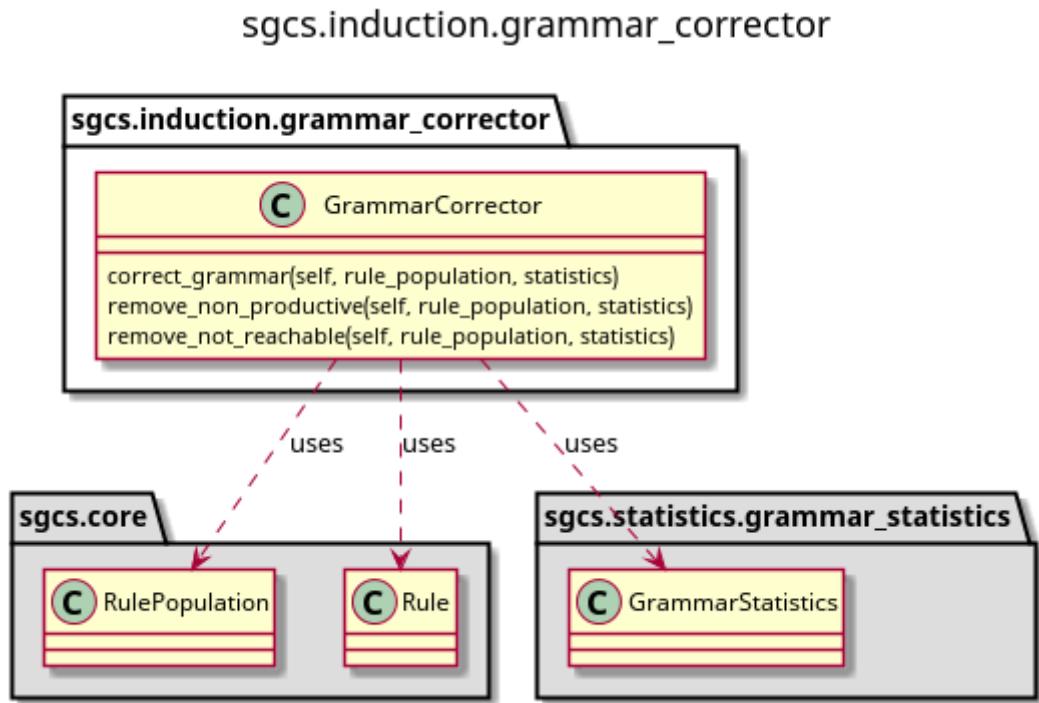
Detector

Wygodny wrapper na koordynaty produkcji. Zawiera również logikę generowania produkcji na podstawie symboli jednoznacznie wskazywanych przez krotkę coordinates w tabeli CYK.

coordinates – jednoznaczne określenie położenia symboli wskazywanych przez detektor. Dysponując tablicą CYK można powiedzieć, których dwóch symboli dotyczy.

generate_productions(self, environment, rule_population) – metoda znajdująca wszystkie możliwe produkcje dla danego detektora.

sgcs.induction.grammar_corrector (Rysunek 76)



Rysunek 76: Moduł sgcs.induction.grammar_corrector

Moduł zawierający klasy odpowiedzialne za korekcję gramatyki. Obecnie wspiera usuwanie reguł nieproduktywnych oraz nieosiągalnych; usuwanie reguł redundantnych nie jest potrzebne, gdyż budowa populacji reguł chroni przed wystąpieniem powtarzających się reguł.

GrammarCorrector

Klasa odpowiedzialna za korekcję gramatyki.

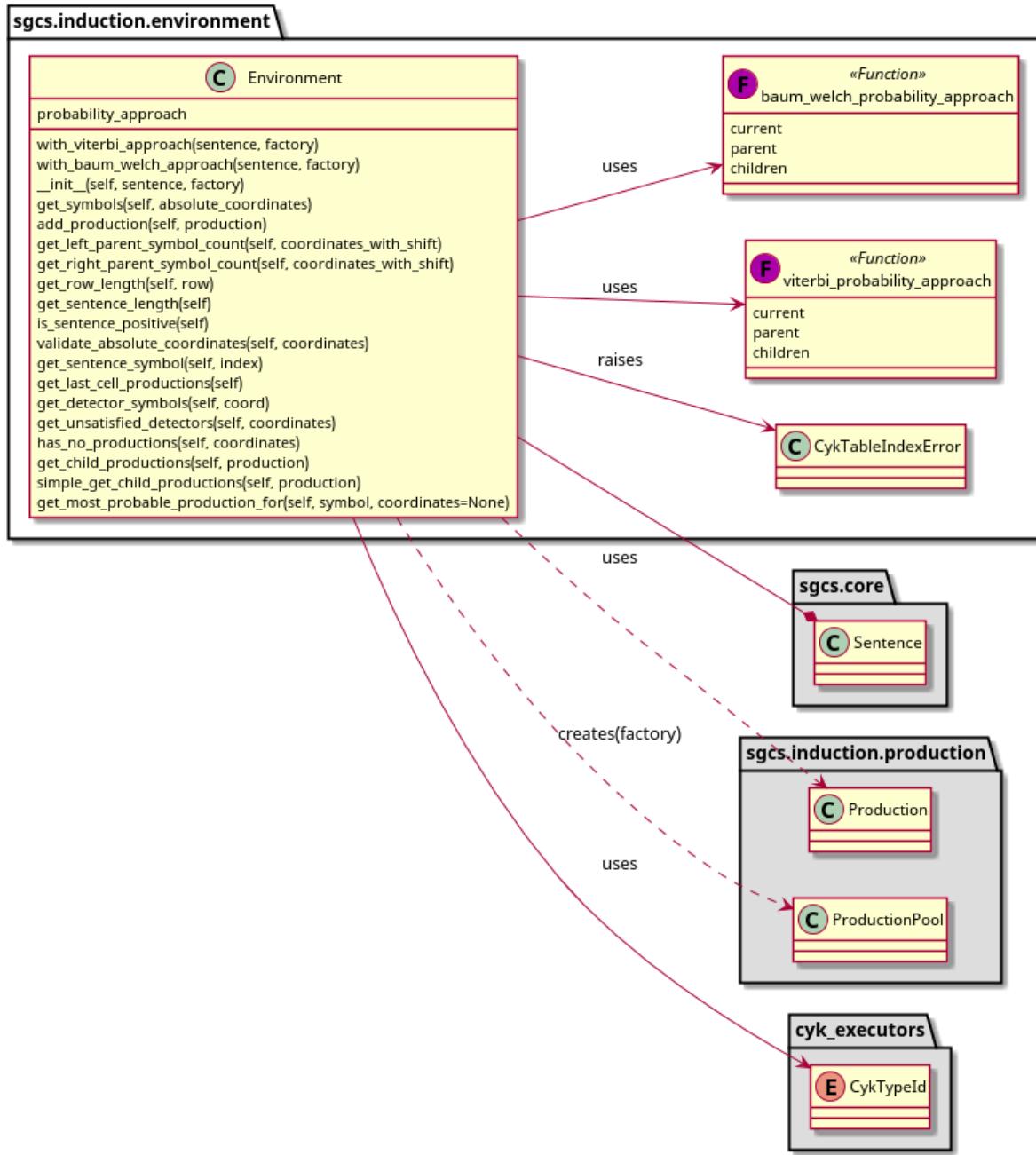
correct_grammar(self, rule_population, statistics) – metoda wywołująca remove_non_productive, a następnie remove_not_reachable.

remove_non_productive(self, rule_population, statistics) – metoda usuwająca reguły, z których nie da się wyprowadzić reguł terminalnych.

remove_not_reachable(self, rule_population, statistics) – metoda usuwająca reguły, których nie da się wyprowadzić z symbolu startowego gramatyki.

sgcs.induction.environment (Rysunek 77)

sgcs.induction.environment



Rysunek 77: Moduł `sgcs.induction.environment`

Moduł ten zawiera klasy ściśle powiązane z obsługą tabeli CYK.

Environment

Klasa reprezentująca tabelę CYK, zdolna do obsługi zarówno danych stochastycznych, jak i klasycznych.

`probability_approach` – delegat sposobu liczenia prawdopodobieństw kolejnych komórek. Dostępni obecnie delegaci to `viterbi_probablity_approach` (liczenie prawdopodobieństw metodą Viterbiego) oraz `baum_welch_probability_approach` (liczenie prawdopodobieństw metodą Bauma-Welcha). Jeżeli klasa nie jest wykorzystywana przez algorytm stochastyczny, wówczas pole `to` przybiera wartość `None`.

`with_viterbi_approach(sentence, factory)` – funkcja tworząca instancję `Environment` z podejściem Viteriego.

`with_baum_welch_approach(sentence, factory)` – funkcja tworząca instancję `Environment` z podejściem Bauma-Welcha.

init (self, sentence, factory) – konstruktor pobierający zdanie, na podstawie którego zostanie wygenerowana tabela CYK oraz fabrykę factory, z której będzie możliwe uzyskanie instancji obiektów ProductionPool (reprezentujących pojedyncze komórki tabeli).

get_symbols(self, absolute_coordinates) – metoda zwracająca wszystkie efektorystyczne symboli w komórce położonej na pozycji absolute_coordinates.

add_production(self, production) – metoda dodająca nową produkcję (wraz z ewentualnym przeliczeniem prawdopodobieństw).

get_left_parent_symbol_count(self, coordinates_with_shift) – metoda zwracająca liczbę efektorów w komórce lewego rodzica.

get_right_parent_symbol_count(self, coordinates_with_shift) – metoda zwracająca liczbę efektorów w komórce prawego rodzica.

get_row_length(self, row) – metoda zwracająca rzeczywistą długość wiersza tabeli CYK (tabela CYK jest tabelą trójkątną).

get_sentence_length(self) – metoda zwracająca długość zdania parsowanego w tabeli.

is_sentence_positive(self) – metoda zwracająca True, jeżeli zdanie należy do gramatyki.

get_sentence_symbol(self, i) – metoda zwracająca i-ty Symbol zdania.

get_last_cell_productions(self) – metoda zwracająca produkcje ostatniej komórki tabeli CYK.

get_detector_symbols(self, coord) – metoda zwracająca symbole rodziców w postaci krotki kolekcji.

get_unsatisfied_detectors(self, coordinates) – metoda zwracająca detektory, dla których nie dopasowano żadnych reguł (przydatne przy generowaniu nowych).

has_no_productions(self, coordinates) – metoda zwraca True, jeżeli w danej komórce tabeli CYK nie zapisano żadnych produkcji.

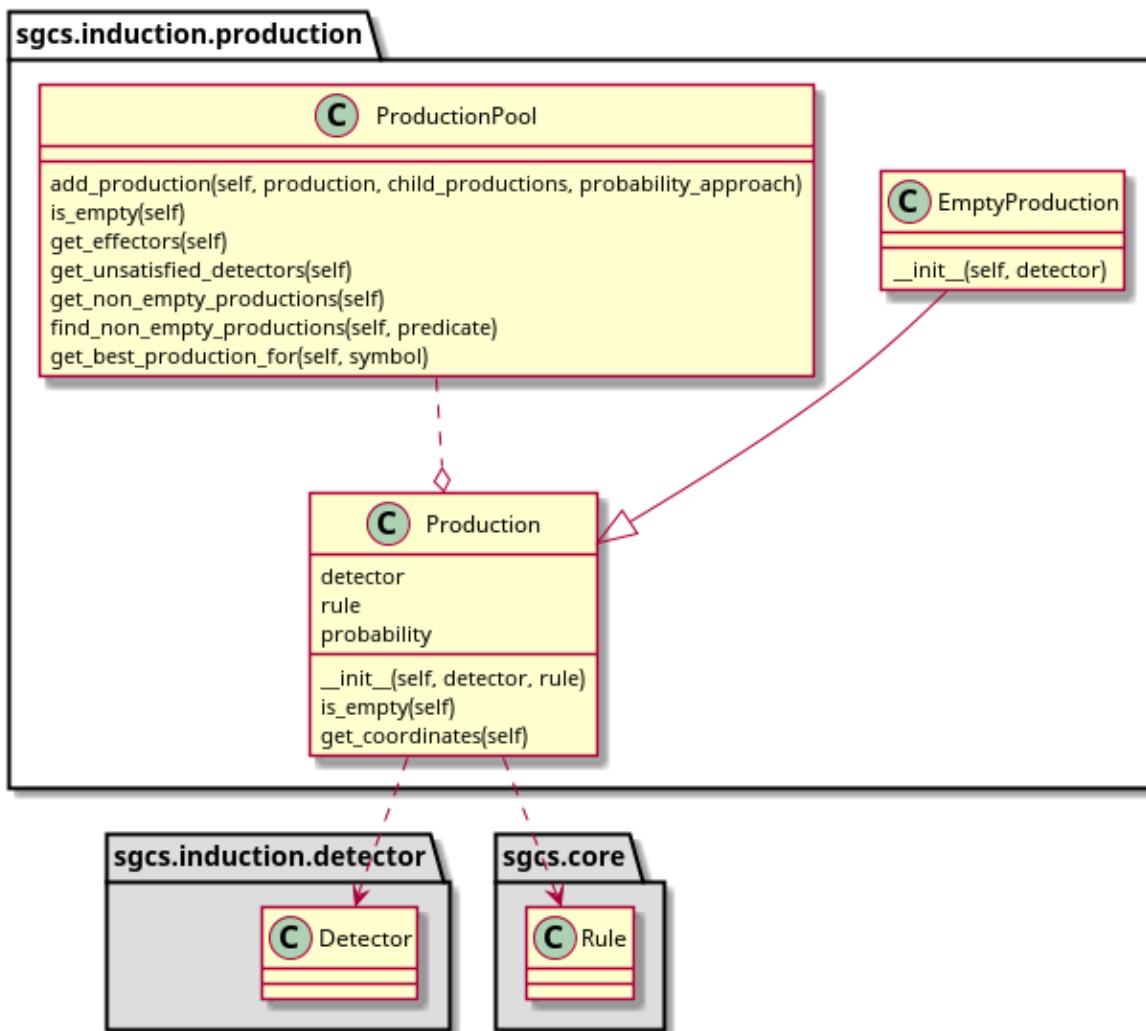
simple_get_child_productions(self, production) – metoda zwracająca produkcje produkowane przez produkcję production (wykorzystywane między innymi przy przemierzaniu drzewa algorymem Traceback). Jeżeli mamy do czynienia z algorymem stochastycznym, prawdopodobieństwa są również zwracane.

get_most_probable_production_for(self, symbol, coordinates=None) – Zwraca z komórki CYK o koordynatach coordinates produkcję o rodzicu symbol taką, że jej prawdopodobieństwo jest największe. Jeżeli pominie się pole coordinates, wówczas metoda przyjmie koordynaty ostatniej komórki tabeli CYK.

CykTableIndexError

Wyjątek zgłoszany w przypadku wykroczenia indeksem poza granice tabeli CYK.

`sgcs.induction.production` (Rysunek 78)



Rysunek 78: Moduł `sgcs.induction.production`

Moduł służący do zarządzania produkcjami w systemie. Przez produkcję należy tutaj rozumieć obiekt wiążący detektor z regułą populacji.

Production

Produkcja jest obiektem relacji Detector oraz Rule. Oznacza to, że daje on wiedzę, skąd dana reguła została uzyskana oraz w jakim miejscu tabeli CYK się znajduje. Są to dane niezmiernie przydatne, ułatwiają między innymi wykorzystanie algorytmu traceback, który musi tylko poruszać się po zapamiętanej wcześniej sieci współrzędnych.

detector – pole przechowujące obiekt klasy Detector.

rule – pole przechowujące obiekt klasy Rule.

probability – obiekt przechowujący wartość prawdopodobieństwa danej produkcji (jest to prawdopodobieństwo całego drzewa generowanego przez daną produkcję, nie zaś prawdopodobieństwo pojedynczej produkcji). To pole jest opcjonalne – jeżeli klasa jest wykorzystywana poza algorytmem stochastycznym,ówczas jego wartość jest ignorowana.

is_empty(self) – metoda zwracająca True, jeżeli produkcja jest „pusta”. Pusta, czyli nie generuje żadnej reguły (rule = None).

get_coordinates(self) – zwraca współrzędne przechowywane przez detector.

EmptyProduction

Produkcja nieposiadającą reguły. Klasa ta w zasadzie nie rozszerza klasy Production, a stanowi jedynie ekspresywny alias na `Production(detector, None)`.

ProductionPool

Pula produkcji, wykorzystywana między innymi do reprezentacji zawartości pojedynczej komórki w tabeli CYK.

add_production(self, production, child_productions, probability_approach) – metoda dodająca produkcję do puli. Wykonywane są przy okazji różne czynności pomocnicze jak zapisywanie efektorów, przeliczanie prawdopodobieństw itp.

is_empty(self) – zwraca True, jeżeli pula nie zawiera niepustych produkcji, False w przeciwnym przypadku.

get_effectors(self) – zwraca efektry zarejestrowane w puli.

get_unsatisfied_detectors(self) – zwraca detektory pustych populacji (przydatne przy generowaniu nowych reguł).

get_non_empty_productions(self) – zwraca wszystkie niepuste produkcje.

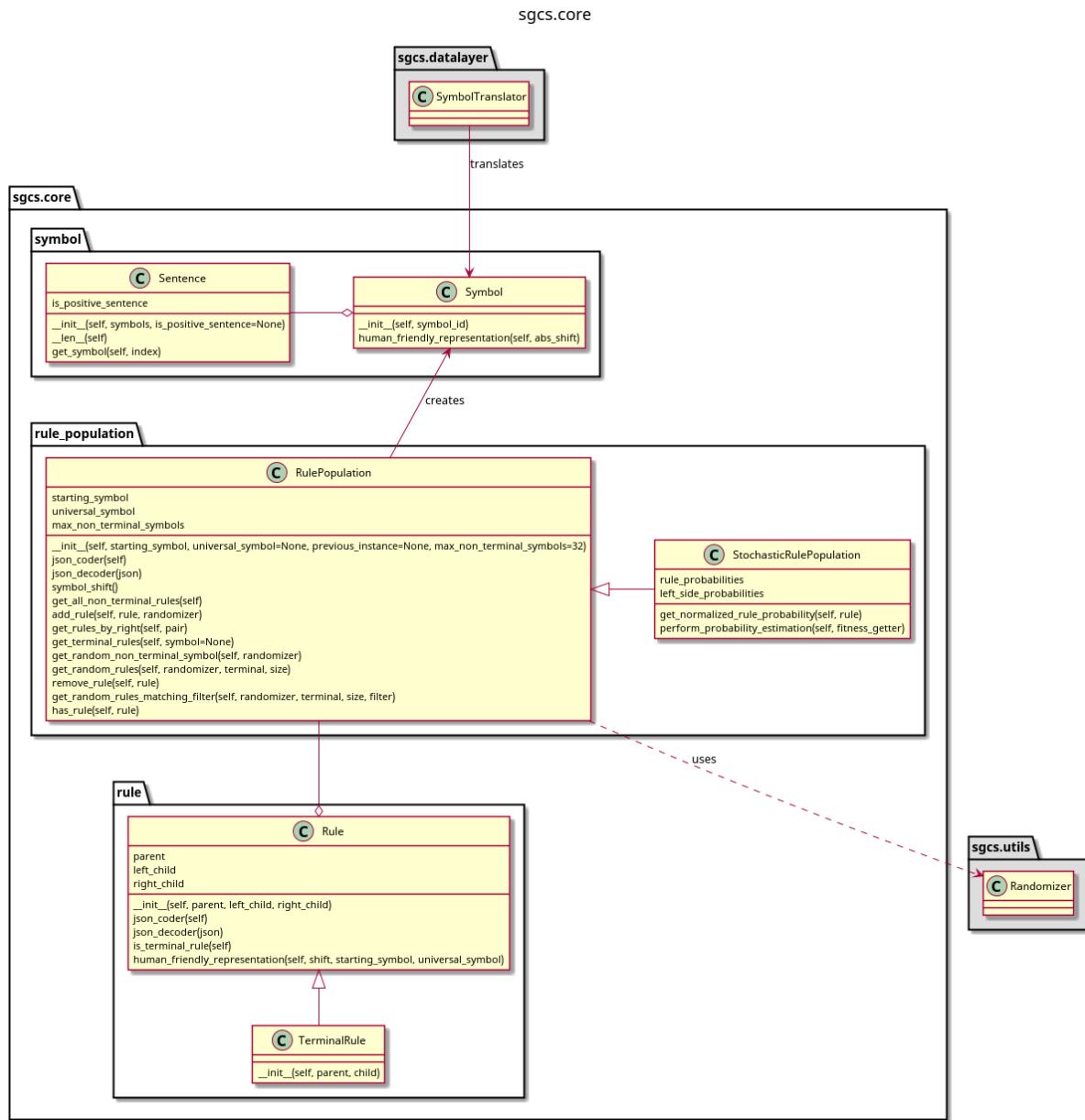
find_non_empty_productions(self, predicate) – zwraca wszystkie niepuste produkcje spełniające predykat predicate.

get_best_production_for(self, symbol) – zwraca najbardziej prawdopodobną produkcję dla efektora symbol.

Warstwa rdzenia

Warstwa ta jest sercem algorytmu. Klasy zawarte w jej jedynym module sgcs.core są wykorzystywane w wielu miejscach aplikacji, zazwyczaj reprezentują najbardziej podstawowe pojęcia związane z algorytmem wnioskowania gramatycznego. Są to klasy reprezentujące podstawowe dane pamięci algorytmu.

sgcs.core (Rysunek 79)



Rysunek 79: Moduł `sgcs.core`

RulePopulation

Klasa odpowiedzialna za zarządzanie populacją reguł.

starting_symbol – reprezentacja symbolu startowego.

universal_symbol – reprezentacja symbolu uniwersalnego.

max_non_terminal_symbols – stała, określa maksymalną liczbę symboli nieterminalnych.

json_coder(self) – metoda dokonująca tłumaczenia populacji reguł na format `.json`.

json_decoder(self, json) – metoda wczytująca stan wewnętrzny obiektu z obiektu w formacie `.json`.

symbol_shift – wartość bezwzględna, od której zaczyna się numerowanie indeksów (każdy index taki, że $|index| < symbol_shift$ jest traktowany jako index zarezerwowany na symbol specjalny).

get_all_non_terminal_rules(self) – metoda zwracająca wszystkie reguły nieterminalne.

add_rule(self, rule, randomizer) – dodaje nową regułę do populacji, dokonując losowania prawdopodobieństwa, jeżeli jest to wymagane.

get_rules_by_right(self, pair) – zwraca wszystkie reguły, których prawa strona produkcji = pair.

get_terminal_rules(self, symbol=None) – zwraca wszystkie terminale reguły lub terminalne reguły produkujące symbol (jeżeli podany).

get_random_non_terminal_symbol(self, randomizer) – tworzy losowy symbol.

get_random_rules(self, randomizer, terminal, size) – zwraca maksymalnie size losowych reguł.

remove_rule(self, rule) – usuwa wybraną regułę z populacji.

get_random_rules_matching_filter(self, randomizer, terminal, size, filter) – metoda zwracająca maksymalnie size losowych reguł, które spełniają warunek filter.

has_rule(self, rule) – metoda zwraca True, jeżeli reguła rule znajduje się w populacji.

StochasticRulePopulation

Klasa rozszerzająca klasę RulePopulation o obsługę reguł stochastycznych.

rule_probabilities – mapa reguła – prawdopodobieństwo.

left_side_probabilities – mapa ojciec reguły – prawdopodobieństwo symbolu.

get_normalized_rule_probability(self, rule) – metoda zwracająca znormalizowaną wartość prawdopodobieństwa reguły.

perform_probability_estimation(self, fitness_getter) – metoda powodująca wyliczenie nowych prawdopodobieństw dla reguł.

Rule

Klasa stanowiąca reprezentację reguły.

parent – rodzic Rule.

left_child – lewe dziecko Rule.

right_child – prawe dziecko Rule.

STARTING_SYMBOL_REPR – stała, reprezentacja symbolu startowego.

UNIVERSAL_SYMBOL_REPR – stała, reprezentacja symbolu uniwersalnego.

json_coder(self) – metoda dokonująca tłumaczenia reguły na format json.

json_decoder(self, json) – metoda wczytująca stan wewnętrzny obiektu z obiektu w formacie json.

is_terminal_rule(self) – zwraca True, jeżeli reguła jest terminalna.

human_friendly_representation(self, shift, starting_symbol, universal_symbol) – funkcja tłumacząca obiekt Rule na obiekt łatwy w odczytaniu przez człowieka.

TerminalRule

Cukier syntaktyczny na wyrażenie Rule(A, B, None).

Sentence

Obiekt stanowiący reprezentację zdania.

is_positive_sentence – czy zdanie jest tak naprawdę zdaniem pozytywnym.

init_(self, symbols, is_positive_sentence=None) – konstruktor. Pobiera zestaw symboli, z jakich zostanie skonstruowane oraz informację czy podane zdanie w rzeczywistości należy do gramatyki, czy też nie.

len_(self) – zwraca długość zdania.

get_symbol(self, index) – Zwraca symbol na pozycji index.

Symbol

Klasa stanowiąca reprezentację pojedynczego symbolu gramatyki. Jeżeli ma się do czynienia z instancją Symbol reprezentującą symbol terminalny, to klasa ta nie posiada informacji wystarczającej do odtworzenia skrywanej wartości – potrzeba do tego dodatkowej wiedzy w postaci obiektu SymbolTranslator z modułu sgcs.datalayer.

human_friendly_representation(self, abs_shift) – Wyświetla ludzką reprezentację symbolu (jak zostało to szczegółowo omówione przy oknie „Population Editor”). abs_shift jest wartością przesunięcia symboli gramatyki, czyli powinna być tu użyta wartość RulePopulation.rule_shift.

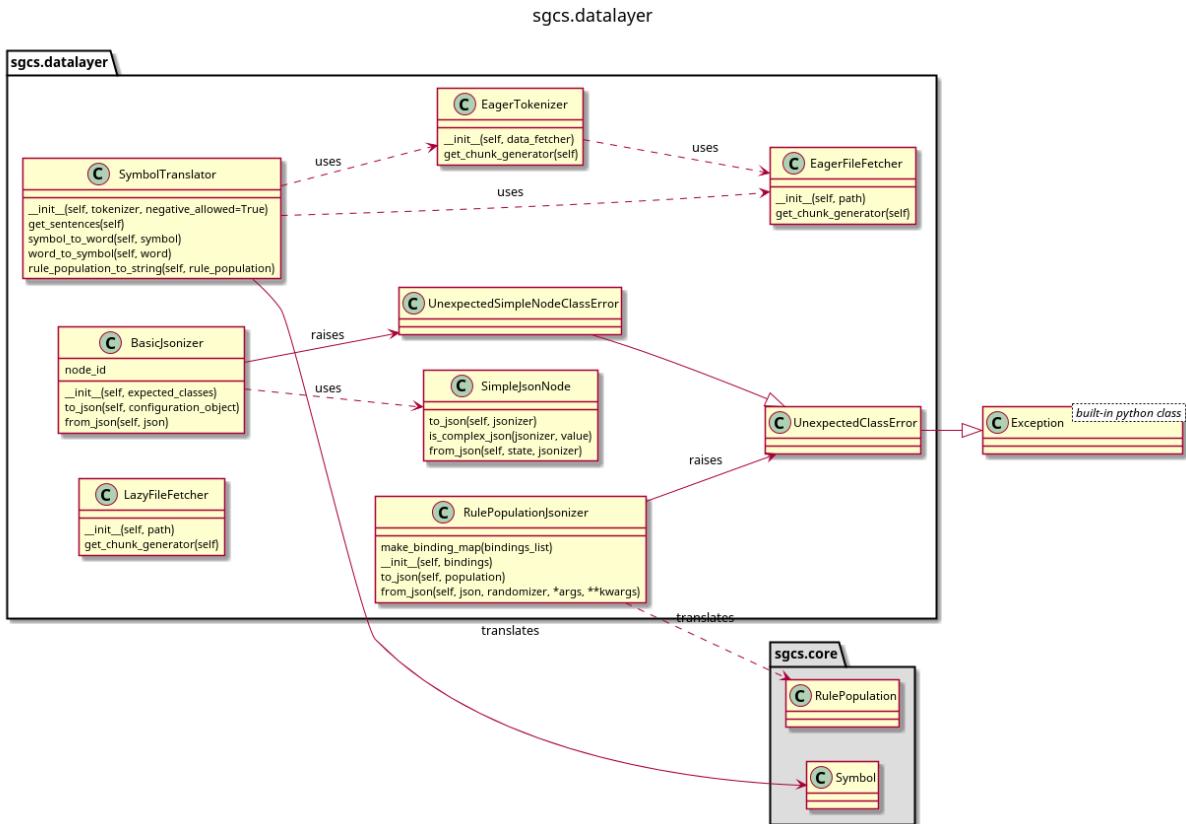
Metody tej powinniśmy używać tylko wyłącznie w przypadku gdy dany Symbol jest symbolem nieterminalnym – nie należy stosować tej metody w przypadku symboli terminalnych oraz specjalnych. Najlepiej zostawić wówczas proces tłumaczenia samej klasie SymbolTranslator, która potrafi prawidłowo wykorzystywać tę metodę.

from_human_friendly_representation(human_repr, abs_shift) – Metoda umożliwiająca odtworzenie reprezentacji wewnętrznej symbolu nieterminalnego z ludzkiej reprezentacji. Powstała ona z myślą o byciu wywoływaną przez klasę SymbolTranslator i wszelkie jej wywołania w innym kontekście powinny być wykonywane ze szczególną ostrożnością.

Warstwa danych

Warstwa ta skupia się na prawidłowej serializacji i deserializacji danych zawartych w module sgcs.core. Zawiera tylko jeden moduł – sgcs.datalayer.

sgcs.datalayer (Rysunek 80)



Rysunek 80: Moduł sgcs.datalayer

Moduł odpowiedzialny za serializację, deserializację oraz zapisywanie i wczytywanie danych z lokalnego systemu plików.

SymbolTranslator

Klasa zapewniająca dwustronne tłumaczenie symboli z wewnętrznej na ludzką reprezentację i odwrotnie. Jest to klasa o stosunkowo wysokim poziomie abstrakcji, nie wymagając od użytkownika wiedzy na temat sposobu przechowywania danych.

create(path) – funkcja tworząca domyślnie skonfigurowaną instancję klasy SymbolTranslator z gorliwym ładowaniem pliku path w chwili pierwszej potrzeby tłumaczenia.

__init__(self, tokenizer, negative_allowed=True) – konstruktor pobierający instancję tokenizera (obiekt zwracający dane wejściowe już w poddanej tokenizacji formie) oraz informację czy zdania negatywne również powinny być parsowane (w przypadku negative=False zdania negatywne zwracane przez tokenizera będą ignorowane).

get_sentences(self) – zwraca generator zwracający kolejne zdania gramatyki. Zdania są pobierane leniwie z tokenizera (od klas położonych niżej zależy czy są wczytywane gorliwie, czy też leniwie).

symbol_to_word(self, symbol) – klasa tłumacząca symbol. Generuje reprezentację dla symboli nieterminalnych (z wykorzystaniem metod klasy sgcs.core.Symbol), zwraca reprezentację symboli specjalnych oraz odczytuje na podstawie zdobytej podczas parsowania wiedzy symbole terminalne. Aby metoda ta prawidłowo zadziałała, symbol terminalny musiał wystąpić już w jakimś zdaniu, aczkolwiek ponieważ klasa ta jest jedynym źródłem zdań, a zatem również symboli terminalnych, to każdy symbol musiał już wystąpić w jakimś zdaniu podczas działania algorytmu. Niekoniecznie jest to prawdą w przypadku edycji populacji poza działającym algorytmem i to jest powodem, dla którego należy przy jej dokonywaniu zachować szczególną ostrożność.

word_to_symbol(self, word) – metoda tłumacząca słowo terminalne na instancję klasy Symbol. Dotyczą się jej te same uwagi i obostrzenia co metody symbol_to_word.

rule_population_to_string(self, rule_population) – funkcja zwracająca napis stanowiący wygodną w czytaniu reprezentację populacji reguł. Klasa ta wykorzystuje metodę symbol_to_word, co może być potencjalnym źródłem problemów napotykanych przy wykorzystaniu tej metody.

EagerTokenizer

Klasa, której zadaniem jest podział linii zwróconych przez obiekt data_fetcher na łatwe w parsowaniu tokeny. Nazwa może być myląca, EagerTokenizer dokonuje leniwej tokenizacji.

init_(self, data_fetcher) – konstruktor przyjmujący obiekt data_fetcher odpowiedzialny za dostarczanie zdań w postaci linii tekstu.

get_chunk_generator(self) – metoda tworząca generator zwracający kolejne linie w postaci listy tokenów

EagerFileFetcher

Klasa odpowiedzialna za gorliwe wczytywanie pliku z danymi wejściowymi. Zamiast tej klasy można zdecydować się na LazyFileFetcher (wczytujący plik leniwie).

init_(self, path) – konstruktor zapamiętujący ścieżkę path z danymi wejściowymi.

get_chunk_generator(self) – zwraca generator zwracający kolejne zdania (w sposób leniwy lub chowając pod sobą listę).

BasicJsonizer

Klasa tłumacząca obiekty dziedziczące po klasie SimpleJsonNode na format json i odwrotnie. Tłumaczona klasa oraz jej składowe musi być typu podstawowego albo równocześnie dziedziczyć po klasie SimpleJsonNode i być na liście expected_classes. Klasę tę wykorzystuje się głównie do tłumaczenia węzłów konfiguracji.

node_id – stała będąca kluczem, pod którym zapisuje się w json nazwy klas, dzięki czemu jest możliwe ich późniejsze odtworzenie.

init_(self, expected_classes) – konstruktor pobierający listę obiektów klas, które mają być obsługiwane przezinstancję (wszystkie powinny dziedziczyć po klasie SimpleJsonNode).

to_json(self, obj) – metoda tłumacząca obj na napis w formacie json.

from_json(self, json) – metoda tłumacząca napis w formacie json na obiekt.

RulePopulationJsonizer

Klasa odpowiedzialna za serializację obiektów typu RulePopulation.

make_binding_map(binding_list) – funkcja przyjmująca listę klas, z których następnie tworzy mapę nazwa_klasy – klasa.

init_(self, bindings) – konstruktor przyjmujący mapę nazwa-klasa obsługiwanych klas (na przykład RulePopulation oraz klasa dziedziczące).

to_json(self, population) – klasa zwracająca napis w formacie json będący tłumaczeniem obiektu population.

from_json(self, json, randomizer, *args, **kwargs) – metoda zwracająca obiekt obiekt populacji tworzony na podstawie napisu w formacie json. Ponieważ populacje potrzebują zazwyczaj dodatkowych parametrów, są one również przekazywane.

Parametry algorytmu

Algorytm przetestowano dla trzech wariantów algorytmu klasyfikacji gramatycznej – GCS, sGCS oraz neg-sGCS. Poniżej przedstawiono omówienie parametrów, z jakimi uruchamiano algorytm.

f_{GA} – czy należy uruchomić algorytm genetyczny (tak/nie)? Podano: tak.

f_{GA}^s – lista typów stosowanej selekcji (gdzie typy to losowa, ruletka, turniej). Podano: {ruletka, ruletka}

f_{kor} – zmienna zezwalająca na uruchomienie korekcji gramatyki (tak, nie). Podano: nie.

p_{cs} – prawdopodobieństwo zastosowania operatora pokrycia startowego ([0,1]). Podano: 1.

p_{cp} – prawdopodobieństwo zastosowania operatora pokrycia pełnego ([0,1]). Podano: 1.

p_{cp} – prawdopodobieństwo zastosowania operatora pokrycia uniwersalnego ([0,1]). Podano: 0.

n_{max} – maksymalna liczba kroków ewolucyjnych ([1, ...]). Podano: 5 000.

n_{run} – liczba iteracji ([1, ...]). Podano: 50.

n_p – rozmiar populacji ([[1, ...]]). Podano: 40.

n_{start} – liczba startowych produkcji nieterminalnych ([0, ...]). Podano: 30.

n_N – liczba symboli nieterminalnych ([1, ...]). Podano: 19.

p_k – prawdopodobieństwo krzyżowania dla algorytmu genetycznego ([0,1]). Podano: 0.2.

p_m – prawdopodobieństwo mutacji dla algorytmu genetycznego ([0,1]). Podano: 0.8.

p_i – prawdopodobieństwo inwersji ([0,1]). Podano 0.

p_a – prawdopodobieństwo zastosowania operatora pokrycia agresywnego ([0,1]). Podano: 0.

cf – współczynnik ścisłu ([0, ..]). Podano: 18.

cs – podpopulacja ścisła ([1, ...]). Podano: 3.

ba – kwota bazowa. Podano 0.5.

raf – współczynnik zmniejszania kwoty bazowej. Podano: 0.5.

n_{elit} – rozmiar elity. ([0, ...]). Podano: 0.

w_p – waga rozbioru zdania poprawnego. Podano: 1.

w_n – waga rozbioru zdania niepoprawnego. Podano: 2.

w_c – waga funkcji klasycznej przystosowania. Podano: 1.

w_c – waga funkcji plodności. Podano: 0.

f_0 – miara użyteczności klasyfikatora niebiorącego udziału w parsowaniu. Podano: 0.5.

f_{R+} – zmienna informująca czy powinniśmy uczyć się też na podstawie zdań negatywnych. Ustawiona na false w przypadku sGCS, true w pozostałych algorytmach.

Algorytm sGCS oraz neg-sGCS korzystają dodatkowo z następujących parametrów:

p_{type} – rodzaj algorytmu stosowanego do obliczania prawdopodobieństwa derywacji (Viterbi lub Baum-Welch). Podano: Viterbi.

Wyniki badań

Algorytm był uruchamiany na laptopie wyposażonym w 8 rdzeni logicznych oraz na zestawie serwerów postawionych w chmurze Amazon Web Services, o łącznej mocy obliczeniowej 100 rdzeni logicznych. Ze względu na wysoką przenosalność biblioteki uruchomienie aplikacji w chmurze okazało się być stosunkowo prostym zadaniem, po uprzednim zrezygnowaniu z zapewnienia funkcjonującego gui PyQt, którego instalacja i konfiguracja okazały się niełatwym zadaniem na serwerach firmy Amazon. Algorytmy poddano trzem rodzajom testów. Po pierwsze powtórzono badania GCS oraz sGCS, przedstawione w pracy Kępy [8], co umożliwiło pokazanie poprawności implementacji i dało bazę do której można było porównać wyniki kolejnych badań. Następnie przetestowano działanie algorytmu neg-sGCS. Wreszcie przetestowano wpływ reguł charakterystycznych na proces uczenia algorytmu. Dodatkowo rozpoczęto badania wpływu powtarzających się zdań w zbiorze uczącym na proces uczenia, aczkolwiek zabrakło czasu na przeprowadzenie sensownej ilości eksperymentów, toteż badania te nie zostaną szerzej omówione.

Standardowa losowa populacja

Na początku przeprowadzono badania z zastosowaniem populacji złożonej wyłącznie z losowych reguł. Nie określono z góry żadnego zestawu reguł charakterystycznych i nie dodano ich do programu. Poniżej przedstawiono wyniki tych badań – GCS, sGCS oraz neg-sGCS.

GCS

Algorytm GCS uruchomiono ze standardowymi parametrami, wymienionymi w poprzednim rozdziale. Stosowano klasyczną funkcję przystosowania.

Tomita 1

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	137,98	100,00%	100,00%	100,00%
2	7,60	100,00%	100,00%	100,00%
3	1,64	100,00%	100,00%	100,00%
4	1,64	100,00%	100,00%	100,00%
5	1,52	100,00%	100,00%	100,00%
Average:	30,08	100,00%	100,00%	100,00%

Rysunek 81: Tabela skuteczności GCS dla populacji standardowej – Tomita 1

Tomita 2

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	16,08	100,00%	100,00%	100,00%
2	98,30	100,00%	100,00%	100,00%
3	38,86	100,00%	100,00%	100,00%
4	5,96	100,00%	100,00%	100,00%
5	36,23	100,00%	100,00%	100,00%
Average:	39,09	100,00%	100,00%	100,00%

Rysunek 82: Tabela skuteczności GCS dla populacji standardowej – Tomita 2

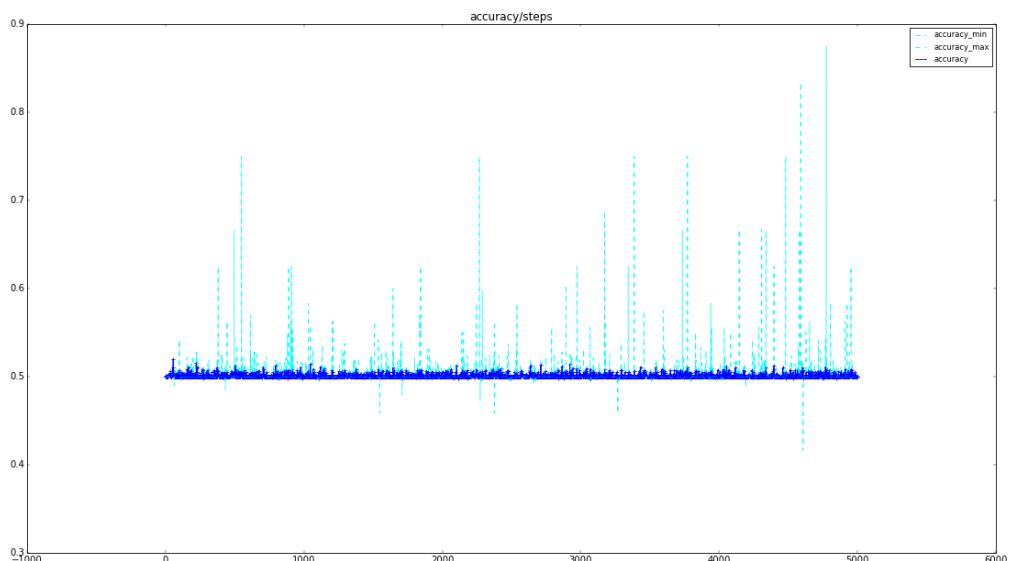
W przypadku gramatyk Tomita 1 (Rysunek 81) oraz Tomita 2 (Rysunek 82) nie ma wielkiego zaskoczenia. Każdy wyuczony zestaw klasyfikatorów uzyskał 100% przystosowania w teście generalizacji.

Tomita 3

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	937,43	63,32%	93,91%	89,50%
2	nan	100,00%	0,00%	14,42%
3	1637,43	83,28%	77,62%	78,44%
4	2271,70	100,00%	100,00%	100,00%
5	3353,38	62,80%	93,20%	88,82%
Average:	2049,99	81,88%	72,95%	74,24%

Rysunek 83: Tabela skuteczności GCS dla populacji standardowej – Tomita 3

Gramatyka Tomita 3 (Rysunek 83) ze względu na posiadanie błędnych przykładów w zbiorze uczącym [1] okazała się trudną gramatyką do wyuczenia. W przypadku drugiego uruchomienia żaden z 50 cykli uczących nie zdął znaleźć rozwiązania. Jak widać na rysunku (Rysunek 84) algorytm w drugim przebiegu przez większość czasu osiągał przystosowanie rzędu 50%, czasami tylko osiągając krótkotrwałą poprawę. W pozostałych przypadkach otrzymano przystosowanie ok. 74%, co daje trochę słabszy wynik niż w przypadku dotychczasowych prac [2] [8].



Rysunek 84: Accuracy dla kolejnych kroków przebiegu algorytmu (GCS, Tomita 3)

Tomita 4

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1347,46	87,58%	100,00%	95,59%
2	2086,47	5,34%	83,84%	56,00%
3	1830,18	99,99%	98,07%	98,75%
4	2240,00	91,26%	67,54%	75,95%
5	1847,62	0,12%	99,99%	64,56%
Average:	1870,35	56,86%	89,89%	78,17%

Rysunek 85: Tabela skuteczności GCS dla populacji standardowej – Tomita 4

Tomita 5

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	710,63	90,26%	53,04%	59,24%
2	958,00	66,71%	73,34%	72,24%
3	941,26	90,18%	63,09%	67,61%
4	970,24	81,79%	69,96%	71,93%
5	824,94	72,04%	86,28%	83,91%
Average:	881,01	80,20%	69,14%	70,99%

Rysunek 86: Tabela skuteczności GCS dla populacji standardowej – Tomita 5

Tomita 6

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	2132,31	100,00%	100,00%	100,00%
2	1960,82	86,25%	32,69%	50,54%
3	1741,39	97,11%	1,71%	33,51%
4	1924,84	42,82%	90,19%	74,40%
5	1861,03	100,00%	100,00%	100,00%
Average:	1924,08	85,24%	64,92%	71,69%

Rysunek 87: Tabela skuteczności GCS dla populacji standardowej – Tomita 6

Tomita 7

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1848,32	60,91%	80,10%	79,36%
2	1792,28	55,63%	84,97%	83,84%
3	2124,00	46,52%	97,89%	95,92%
4	2109,53	78,81%	100,00%	99,19%
5	1506,62	48,79%	93,75%	92,03%
Average:	1876,15	58,13%	91,34%	90,07%

Rysunek 88: Tabela skuteczności GCS dla populacji standardowej – Tomita 7

Pomiary dla Tomita 4-7 (Rysunek 85 do Rysunek 88) również wydają się być nieznacznie niższe niż te osiągnięte w publikacji Unolda [1]. Nie odbiegają one jednak zbytnio od wyników przedstawionych w pracy Kępy [8]. Może to być spowodowane potencjalnym błędem implementacji lub rozbieżnością któregoś z parametru. Ponieważ jednak wyniki pozostają z grubsza takie same (jedyna większa różnica to trochę gorsze rezultaty Tomita 4, które uzyskało przystosowanie rzędu 100% w pracy Kępy [8]), a sam algorytm w znacznej mierze opiera się na losowości i może generować różne wyniki, więc zdecydowano się na kontynuowanie badań i porównanie wyników pozostałych badań z otrzymanymi powyżej.

ab

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1118,50	100,00%	100,00%	100,00%
2	1776,17	100,00%	100,00%	100,00%
3	2034,00	100,00%	100,00%	100,00%
4	295,67	100,00%	100,00%	100,00%
5	676,33	100,00%	100,00%	100,00%
Average:	1180,13	100,00%	100,00%	100,00%

Rysunek 89: Tabela skuteczności GCS dla populacji standardowej – ab

anbn

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	798,80	100,00%	100,00%	100,00%
2	1181,26	100,00%	100,00%	100,00%
3	1036,11	100,00%	99,98%	99,98%
4	832,00	100,00%	99,97%	99,97%
5	1108,95	100,00%	100,00%	100,00%
Average:	991,42	100,00%	99,99%	99,99%

Rysunek 90: Tabela skuteczności GCS dla populacji standardowej – anbn

bra1

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	267,86	0,00%	99,04%	98,09%
2	479,07	0,00%	99,04%	98,09%
3	813,74	0,00%	99,04%	98,09%
4	813,74	0,00%	99,04%	98,09%
5	276,65	0,00%	99,04%	98,09%
Average:	530,21	0,00%	99,04%	98,09%

Rysunek 91: Tabela skuteczności GCS dla populacji standardowej – bra1

bra3

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	213,88	0,68%	99,62%	65,99%
2	495,58	0,53%	99,55%	65,90%
3	539,28	12,32%	98,97%	69,52%
4	248,03	0,60%	99,64%	65,96%
5	252,22	0,84%	98,88%	65,56%
Average:	349,80	2,99%	99,33%	66,59%

Rysunek 92: Tabela skuteczności GCS dla populacji standardowej – bra3

pal2

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	348,67	100,00%	92,46%	92,49%
2	2180,25	100,00%	100,00%	100,00%
3	3506,80	100,00%	100,00%	100,00%
4	nan	33,86%	96,18%	95,94%
5	550,00	100,00%	98,88%	98,88%
Average:	1646,43	86,77%	97,50%	97,46%

Rysunek 93: Tabela skuteczności GCS dla populacji standardowej – pal2

toy

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	845,33	0,00%	99,59%	99,43%
2	1500,00	0,08%	99,76%	99,76%
3	2087,83	0,00%	99,83%	99,67%
4	1138,33	0,00%	99,77%	99,61%
5	594,40	0,08%	99,76%	99,76%
Average:	1233,18	0,03%	99,74%	99,65%

Rysunek 94: Tabela skuteczności GCS dla populacji standardowej – toy

W przypadku gramatyk bezkontekstowych (Rysunek 89 do Rysunek 94) również uzyskaliśmy wysokie wyniki. ab, anbn, pal2 osiągnęły przystosowanie bliskie 100%. Wysoki wynik cechuje również gramatyki toy i bra1, aczkolwiek widać, że w przypadku żadnej z tych gramatyk algorytm nie poradził sobie dobrze ze zdaniami pozytywnymi. O ile nie dziwi to zbytnio w przypadku bra1, to jest trochę niepokojąca w przypadku bra3. Poniżej (Rysunek 95) znajduje się przykładowy zestaw reguł opisujący gramatykę bra3 (za Unold [1]) oraz przykładowy zestaw wygenerowany przez jeden z przebiegów (Rysunek 96):

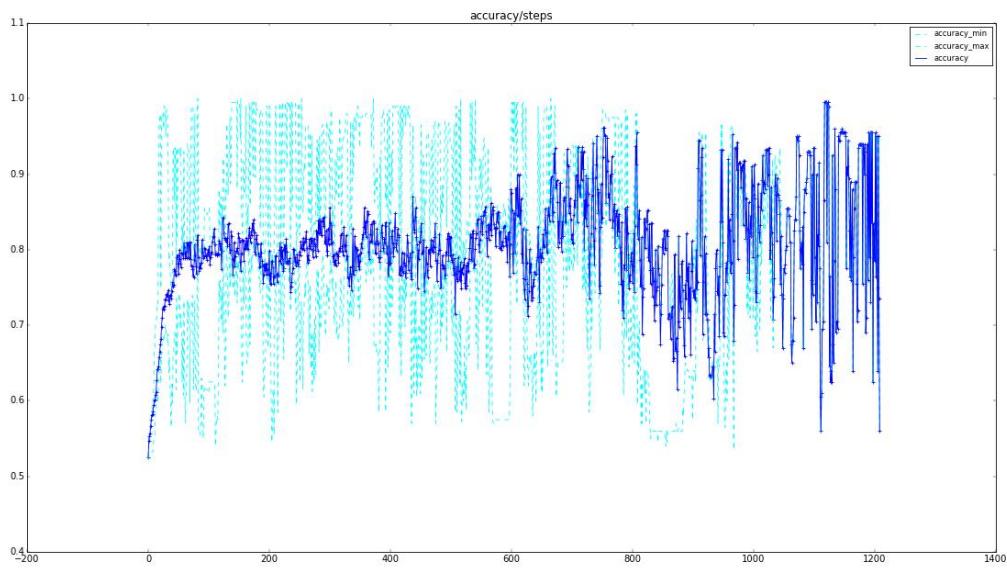
Rysunek 97 przedstawia przystosowanie dla bra3 w kolejnych krokach algorytmu, Rysunek 98 pokazuje miejsce w którym tkwi problem uczenia się gramatyki bra3 – algorytm osiąga już na samym początku bardzo wysokie wyuczenie zdań negatywnych, ale nie klasyfikuje prawidłowo wszystkich zdań pozytywnych. Z czasem radzi sobie z coraz większą ilością zdań pozytywnych, kosztem niestety rozpoznawania zdań negatywnych. To oraz uzyskany specyficzny zestaw reguł pokazują, że zazwyczaj zostaje wybrana populacja reguł wysoce wyspecjalizowanych, co prowadzi do uzyskania niskiego wyniku testu generalizacji dla zdań pozytywnych.

$\langle S \rangle \rightarrow A D$	$A \rightarrow a$
$\langle S \rangle \rightarrow B E$	$B \rightarrow b$
$\langle S \rangle \rightarrow C F$	$C \rightarrow c$
$\langle S \rangle \rightarrow \langle S \rangle \langle S \rangle$	$D \rightarrow d$
$\langle S \rangle \rightarrow A \langle S \rangle$	$E \rightarrow e$
$C \rightarrow C \langle S \rangle$	$F \rightarrow f$
$B \rightarrow B \langle S \rangle$	

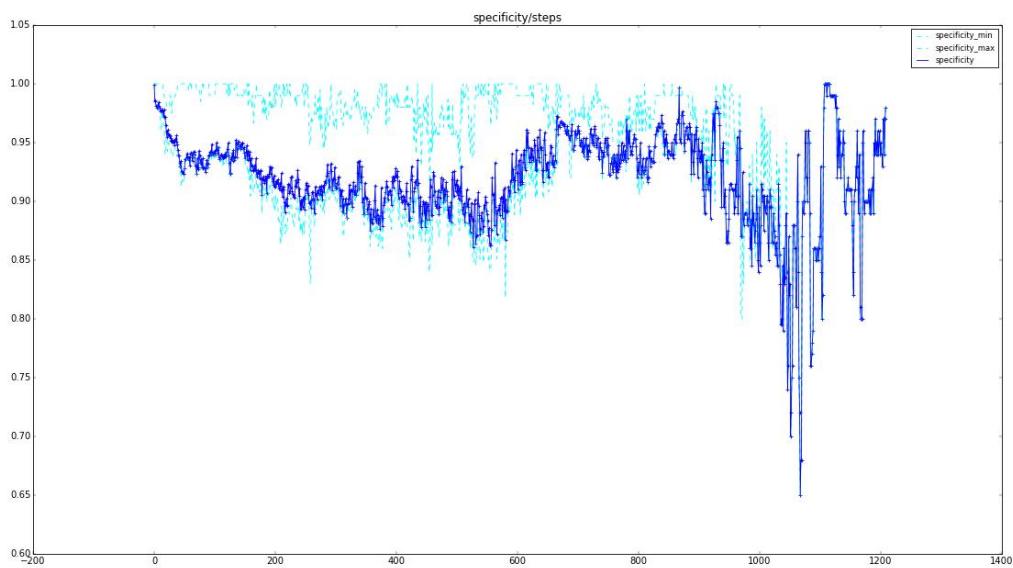
$O \Rightarrow 'b'$	$C \Rightarrow 'e'$
$R \Rightarrow 'c'$	$N \Rightarrow 'a'$
$B \Rightarrow 'f'$	$R \Rightarrow 'd'$
$T \Rightarrow C; S$	$M \Rightarrow N; \langle S \rangle$
$\langle S \rangle \Rightarrow R; B$	$\langle S \rangle \Rightarrow R; M$
$\langle S \rangle \Rightarrow \langle S \rangle; C$	$\langle S \rangle \Rightarrow M; R$
$\langle S \rangle \Rightarrow O; C$	$\langle S \rangle \Rightarrow N; R$
$\langle S \rangle \Rightarrow \langle S \rangle; \langle S \rangle$	$\langle S \rangle \Rightarrow O; \langle S \rangle$
$D \Rightarrow R; \langle S \rangle$	$B \Rightarrow C; D$
$\langle S \rangle \Rightarrow D; B$	$\langle S \rangle \Rightarrow N; D$
$E \Rightarrow D; F$	$I \Rightarrow \langle S \rangle; E$
$G \Rightarrow \langle S \rangle; E$	$C \Rightarrow I; O$
$B \Rightarrow O; I$	$M \Rightarrow C; I$
$Q \Rightarrow O; B$	$G \Rightarrow O; E$
$\langle S \rangle \Rightarrow F; M$	$\langle S \rangle \Rightarrow D; \langle S \rangle$
$U \Rightarrow M; T$	$U \Rightarrow Q; U$

Rysunek 95: "Książkowy" zestaw reguł

Rysunek 96: Wygenerowany zestaw reguł



Rysunek 97: Accuracy bra3



Rysunek 98: Specificity bra3

sGCS

Algorytm sGCS uruchomiono ze standardowym zestawem parametrów, czyli w szczególności z zastosowaniem algorytmu Viterbiego do obliczania prawdopodobieństwa drzew derywacji. Zastosowano funkcję fitness przedstawioną w pracy Pasieki [2], czyli parametr przystosowania był oparty jedynie na ilości zastosowań w parsowaniu zdań (pozytywnych, jako że oczywiście tylko takie wzięły udział w procesie uczenia).

Tomita 1

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1,00	100,00%	100,00%	100,00%
2	1,00	100,00%	100,00%	100,00%
3	1,00	100,00%	100,00%	100,00%
4	1,00	100,00%	100,00%	100,00%
5	1,00	100,00%	100,00%	100,00%
Average:	1,00	100,00%	100,00%	100,00%

Rysunek 99: Tabela skuteczności sGCS dla populacji standardowej – tomita 1

Tomita 2

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	51,98	100,00%	93,96%	93,96%
2	45,32	100,00%	99,78%	99,78%
3	84,38	100,00%	98,30%	98,30%
4	71,24	100,00%	99,99%	99,99%
5	89,80	100,00%	99,78%	99,78%
Average:	68,54	100,00%	98,36%	98,36%

Rysunek 100: Tabela skuteczności sGCS dla populacji standardowej – tomita 2

Tomita 3

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1,00	100,00%	0,00%	14,42%
2	1,00	100,00%	0,00%	14,42%
3	1,00	100,00%	0,00%	14,42%
4	1,00	100,00%	0,00%	14,42%
5	1,00	100,00%	0,00%	14,42%
Average:	1,00	100,00%	0,00%	14,42%

Rysunek 101: Tabela skuteczności sGCS dla populacji standardowej – tomita 3

Tomita 4

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	2,18	100,00%	0,00%	35,47%
2	2,48	100,00%	0,00%	35,47%
3	2,02	100,00%	0,00%	35,47%
4	1,60	100,00%	0,00%	35,47%
5	1,60	100,00%	0,00%	35,47%
Average:	1,98	100,00%	0,00%	35,47%

Rysunek 102: Tabela skuteczności sGCS dla populacji standardowej – tomita 4

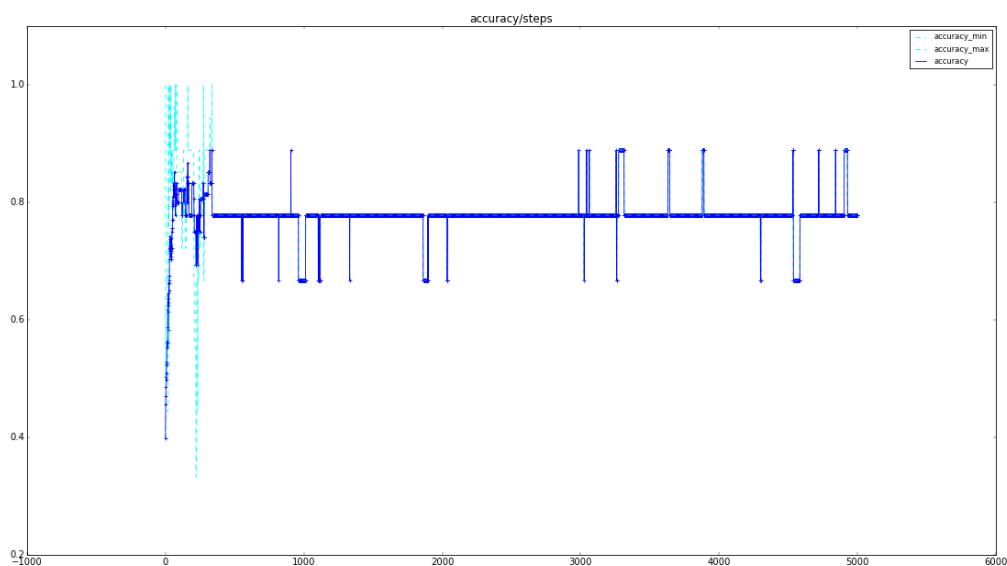
W przypadku gramatyk Tomita 1-4 (Rysunek 99 do Rysunek 102) nie ma żadnych zaskoczeń. Wyniki są praktycznie identyczne z pracą Kępy [8].

Tomita 5

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	267,96	100,00%	80,00%	83,33%
2	75,86	100,00%	80,00%	83,33%
3	99,10	100,00%	80,00%	83,33%
4	51,43	100,00%	80,00%	83,33%
5	85,30	100,00%	80,00%	83,33%
Average:	115,93	100,00%	80,00%	83,33%

Rysunek 103: Tabela skuteczności sGCS dla populacji standardowej – tomita 5

Gramatyka Tomita 5 uzyskała interesujący rezultat, generując za każdym razem praktycznie identyczną gramatykę o przystosowaniu 83,33%. W pracy Kępy [8] widać o wiele większy rozrzut tych wartości, osiągając one również średnio gorsze rezultaty (NGen na poziomie 67,13%, NegGen oscyluje pomiędzy 3,36% a 88,71%, PosGen również nigdy nie osiąga 100%). Jak widać to na diagramie (Rysunek 104), w zestawieniu z plikiem run_summary.txt, w prawie każdym przypadku bardzo szybko znajduje rozwiązanie, tylko raz nie kończy poniżej 500 kroków i zaczyna błądzić w celu znalezienia rozwiązania



Rysunek 104: Diagram Accuracy dla Tomita 5

Tomita 6

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	143,74	100,00%	0,00%	33,34%
2	179,30	100,00%	0,00%	33,34%
3	195,49	100,00%	0,00%	33,34%
4	229,34	100,00%	0,00%	33,34%
5	314,53	100,00%	0,00%	33,34%
Average:	212,48	100,00%	0,00%	33,34%

Rysunek 105: Tabela skuteczności sGCS dla populacji standardowej – tomita 6

Tomita 6 (Rysunek 105) osiąga wyniki zbieżne z pracą Kępy [8].

Tomita 7

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1,50	100,00%	0,00%	3,84%
2	1,38	100,00%	0,00%	3,84%
3	1,46	100,00%	0,00%	3,84%
4	1,36	100,00%	0,00%	3,84%
5	1,44	100,00%	0,00%	3,84%
Average:	1,43	100,00%	0,00%	3,84%

Rysunek 106: Tabela skuteczności sGCS dla populacji standardowej – tomita 7

Da się zauważyć również odstępstwo od pracy Kępy [8] w przypadku gramatyki Tomita 7 (Rysunek 106). Jest ono o tyle dziwne, że oba algorytmy uzyskawszy dokładnie takie same wyniki PosGen oraz NegGen powinny mieć również identyczne wyniki NGen. Tymczasem przy każdym przebiegu zaprezentowanym w pracy Kępy znajdujemy wynik 0,02%. Sprawdzono dokładnie zbiór uczący i wynik ów nie jest możliwy do uzyskania – zbiór testowy Tomita 7 to 65 535 zdań, z czego 2515 zdań pozytywnych. Uzyskanie wyuczenia zdań pozytywnych na poziomie 100% w teście generalizacji oznacza więc skuteczność na poziomie 3,84%, a nie 0,02% jak przedstawiono w pracy magisterskiej Kępy. Zakładając zatem, że w pracy tej popełniono błąd lub wykorzystywano zbiór Tomita 7 o innym zestawie zdań testowych, to prawdopodobnie uzyskano identyczny wynik (również 100% skuteczności w rozpoznaniu zdań pozytywnych i 0% w rozpoznaniu zdań negatywnych).

ab

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	65,70	100,00%	0,00%	7,18%
2	33,00	100,00%	71,83%	73,85%
3	17,67	100,00%	72,03%	74,04%
4	17,90	100,00%	0,03%	7,21%
5	36,60	100,00%	0,05%	7,23%
Average:	34,17	100,00%	28,79%	33,90%

Rysunek 107: Tabela skuteczności sGCS dla populacji standardowej – ab

W przypadku ab (Rysunek 107) uzyskano wyniki o wiele lepsze niż w pracy Kępy [8] (średnio 8,87%), aczkolwiek gorsze niż w pracy Pasieki [2] (92%). Widzimy jednak po pojedynczych uruchomieniach, że algorytm zachowuje się w sposób daleki od stabilnego w przypadku indukcji tej gramatyki (bardzo udane przebiegi 2 i 3, fatalny rezultat w 1). Podobnie zachowuje się on w pracy Kępy [8].

anbn

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	22,92	100,00%	75,01%	75,01%
2	60,40	100,00%	75,01%	75,01%
3	27,76	100,00%	99,71%	99,71%
4	28,78	100,00%	87,67%	97,67%
5	28,52	100,00%	75,01%	75,01%
Average:	33,68	100,00%	82,48%	84,48%

Rysunek 108: Tabela skuteczności sGCS dla populacji standardowej – anbn

Dobre rezultaty otrzymujemy również w przypadku anbn (Rysunek 108). Jak widać zawsze mamy 100% skuteczności w rozpoznaniu zdań pozytywnych, co do negatywnych to oscyluje ona w granicach 75-100%. Jest to zatem wynik zbliżny z pracą Kępy [8] (około 99%) oraz Pasieki [2] (95%).

bra1

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	14,84	100,00%	100,00%	100,00%
2	14,21	100,00%	50,48%	50,96%

3	31,65	100,00%	67,31%	67,62%
4	33,40	100,00%	50,48%	50,96%
5	17,98	100,00%	50,48%	50,96%
Average:	22,42	100,00%	63,75%	64,10%

Rysunek 109: Tabela skuteczności sGCS dla populacji standardowej – bra1

Duża rozbieżność skuteczności rozpoznania zdań negatywnych bra1 (Rysunek 109), zaobserwowana w pracy Kępy [8], wystąpiła również przy okazji tych badań. Uzyskano wyniki podobne do przedstawionych przez poprzednie prace.

bra3

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	71,54	0,00%	99,66%	65,79%
2	25,85	7,15%	96,00%	65,82%
3	152,72	9,15%	90,88%	63,10%
4	137,26	5,77%	85,52%	58,41%
5	78,39	0,49%	96,47%	63,84%
Average:	93,15	4,51%	93,71%	63,39%

Rysunek 110: Tabela skuteczności sGCS dla populacji standardowej - bra3

W przypadku bra3 (Rysunek 110) da się zauważyc interesujący rezultat – algorytm uzyskuje jedynie kilka procent skuteczności w rozpoznawaniu zdań pozytywnych, a w zamian za to bardzo dużą skuteczność w rozpoznawaniu zdań negatywnych. Najwyraźniej miało tu miejsce zjawisko przeuczenia, co potwierdzają uzyskane reguły (Rysunek 111):

pal2

R => 'd'	(0.45)
R => 'a'	(0.46)
U => 'f'	(0.82)
B => 'e'	(0.5)
M => 'b'	(1.0)
J => 'c'	(1.0)
B => R; R (0.5)	
<S> => M; B	(0.23)
<S> => J; <S>	(0.01)
<S> => <S>; B	(0.03)
<S> => <S>; U	(0.23)
<S> => M; <S>	(0.03)
<S> => <S>; <S>(0.23)	
<S> => J; U	(0.24)
U => R; U (0.18)	
C => M; B	(1)
R => R; <S>	(0.09)

Rysunek 111: Reguły wygenerowane dla bra3

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	40,20	100,00%	66,93%	67,06%
2	15,40	100,00%	52,02%	52,20%
3	11,20	100,00%	50,20%	50,39%
4	151,10	100,00%	66,93%	67,06%
5	15,11	100,00%	50,20%	50,39%
Average:	46,60	100,00%	57,26%	57,42%

Rysunek 112: Tabela skuteczności sGCS dla populacji standardowej – pal2

toy

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	449,60	26,67%	93,58%	93,48%
2	227,10	100,00%	0,09%	0,25%
3	286,44	0,00%	97,19%	97,04%
4	269,90	0,95%	84,64%	84,50%
5	198,90	0,00%	99,87%	99,71%
Average:	286,39	25,52%	75,07%	75,00%

Rysunek 113: Tabela skuteczności sGCS dla populacji standardowej – toy

W przypadku gramatyk pal2 (Rysunek 112) oraz toy (Rysunek 113) uzyskano podobne rezultaty jak w dotychczasowych pracach. Statystykę toy zanosi tutaj wyjątkowo niefortunny przebieg nr 2 (gdzie jak widać uzyskano jedynie 0,09% skuteczności w rozpoznaniu zdań negatywnych). Również wartości PosGen wskazują na przeuczenie się algorytmu.

Przedstawione do tej pory wyniki odbiegają nieznacznie od osiągniętych przez pozostałe prace. Różnice te pozostają jednak w akceptowalnej granicy, toteż będzie możliwe przetestowanie nowego wariantu algorytmu oraz wpływu reguł charakterystycznych na proces indukcji.

neg-sGCS

Jest to nowy mechanizm sGCS zaproponowany w tej pracy. Opiera się on na algorytmie sGCS, któremu umożliwiono uczenie się również na przykładach negatywnych, wykorzystując część zagadnień z GCS jako składowe algorytmu będącego również w stanie wykorzystać przykłady negatywne, jak chociażby klasyczną funkcję przystosowania. Właśnie ją zastosowano do określania przystosowania poszczególnych reguł.

Tomita 1

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1,06	100,00%	100,00%	100,00%
2	1,06	100,00%	100,00%	100,00%
3	1,02	100,00%	100,00%	100,00%
4	1,08	100,00%	50,01%	50,02%
5	1,08	100,00%	50,01%	50,02%
Average:	1,06	100,00%	80,00%	80,01%

*Rysunek 114: Tabela skuteczności neg-sGCS dla populacji standardowej – tomita 1***Tomita 2**

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	108,45	100,00%	100,00%	100,00%
2	143,50	100,00%	100,00%	100,00%
3	124,70	100,00%	100,00%	100,00%
4	95,33	100,00%	100,00%	100,00%
5	95,33	100,00%	100,00%	100,00%
Average:	113,46	100,00%	100,00%	100,00%

Rysunek 115: Tabela skuteczności neg-sGCS dla populacji standardowej – tomita 2

Nowa wariant nie wpłynął na proces uczenia gramatyki Tomita 2 (Rysunek 114). Jak w przypadku GCS i sGCS uzyskano 100% przystosowania.

Tomita 3

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	nan	100,00%	0,00%	14,42%
2	3998,67	100,00%	82,95%	85,41%
3	665,50	82,07%	45,63%	50,88%
4	3210,00	100,00%	99,51%	99,58%
5	3210,00	100,00%	99,51%	99,58%
Average:	2771,04	96,41%	65,52%	69,97%

Rysunek 116: Tabela skuteczności neg-sGCS dla populacji standardowej – tomita 3

Podczas indukcji gramatyki Tomita 3 (Rysunek 116) udało się uzyskać całkiem wysokie rezultaty. Przebiegi 2, 4, 5 uzyskały rezultaty bliskie 100% przystosowania. Są one niestety zanione przez wyniki przebiegów 3 i 1. Jest to spowodowane niepowodzeniem w pełnym nauczeniu się na zbiorze uczącym w którymkolwiek z 50 uruchomień uczących przebiegu pierwszego oraz wygenerowaniem bardzo wyspecjalizowanego zestawu reguł w przebiegu 3. Nadal jest to wynik znacznie lepszy niż przy zastosowaniu zwykłego sGCS.

Tomita 4

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	2430,33	85,86%	100,00%	81,48%
2	1826,10	0,58%	100,00%	70,37%
3	2189,60	86,47%	90,12%	97,02%
4	2625,80	91,26%	100,00%	72,04%
5	2625,80	91,26%	100,00%	66,87%
Average:	2339,53	71,09%	98,02%	77,56%

Rysunek 117: Tabela skuteczności neg-sGCS dla populacji standardowej – tomita 4

W przypadku Tomita 4 (Rysunek 117) osiągnął on rezultaty zbliżone do tych osiąganych przez wariant GCS. Po wynikach widać, że algorytm preferował prawidłowe rozpoznanie zdań negatywnych, w przypadku prawie każdego przebiegu uzyskano 100% przystosowania do zdań negatywnych przy równoczesnym zachowaniu wysokiej skuteczności identyfikowania zdań pozytywnych.

Tomita 5

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	650,07	45,75%	88,63%	81,48%
2	816,70	78,58%	68,73%	70,37%
3	886,63	82,15%	100,00%	97,02%
4	880,07	93,66%	67,72%	72,04%
5	1199,29	100,00%	60,25%	66,87%
Average:	886,55	80,03%	77,07%	77,56%

Rysunek 118: Tabela skuteczności neg-sGCS dla populacji standardowej – tomita 5

W przypadku Tomita 5 (Rysunek 118) rezultaty wszystkich trzech algorytmów są zbliżone do siebie, neg-sGCS plasuje się pośrodku ze swoją skutecznością.

Tomita 6

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1355,76	100,00%	100,00%	100,00%
2	1302,38	41,19%	95,03%	77,09%
3	1822,06	66,83%	100,00%	88,94%
4	2121,09	96,04%	100,00%	98,68%
5	1971,26	18,66%	91,12%	66,97%
Average:	1714,51	64,54%	97,23%	86,34%

Rysunek 119: Tabela skuteczności neg-sGCS dla populacji standardowej – tomita 6

Dla Tomita 6 (Rysunek 119) algorytm neg-sGCS uzyskał najlepszy rezultat ze wszystkich trzech gramatyk, pozostawiając wariant sGCS daleko w tyle. Nie odbiega on znacząco w wynikach od GCS (widzimy nawet podobny rozrzuć w przypadku obu algorytmów, oba uzyskują 100% przystosowania w pierwszym przebiegu, obo posiadają również kilka gorszych rezultatów, aczkolwiek GCS częściej uzyskuje dobre wyniki dla zdań pozytywnych, zaś neg-sGCS dla zdań negatywnych).

Tomita 7

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	2833,33	1,75%	100,00%	96,23%
2	2353,50	48,79%	93,75%	92,03%
3	2480,67	48,79%	93,76%	92,03%
4	2235,25	72,09%	81,17%	80,83%
5	2410,50	46,56%	95,23%	93,36%
Average:	2462,65	43,60%	92,78%	90,90%

Rysunek 120: Tabela skuteczności neg-sGCS dla populacji standardowej – tomita 7

W indukcji Tomita 7 (Rysunek 120) neg-sGCS zdecydowanie króluje nad wariantem sGCS, uzyskując około 90% skuteczności w miejsce 4% sGCS. Wyniki są porównywalne z rezultatami osiągniętymi przez GCS.

ab

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1849,17	100,00%	100,00%	100,00%
2	1641,86	100,00%	100,00%	100,00%
3	1534,17	100,00%	100,00%	100,00%
4	234,00	100,00%	100,00%	100,00%
5	903,33	100,00%	91,58%	92,19%
Average:	1232,51	100,00%	98,32%	98,44%

Rysunek 121: Tabela skuteczności neg-sGCS dla populacji standardowej – ab

Dla ab (Rysunek 121) neg-sGCS generuje wyniki niemal tak dobre jak GCS. Zostały one zaburzone jedynie przez przebieg 5, którego skuteczność (92,19%) również trudno nazwać daleką od satysfakcyjającej.

anbn

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1445,39	100,00%	100,00%	100,00%
2	1204,33	100,00%	100,00%	100,00%
3	1251,89	100,00%	99,82%	99,82%
4	1119,24	100,00%	99,84%	99,84%
5	1370,39	100,00%	100,00%	100,00%
Average:	1278,25	100,00%	99,93%	99,93%

Rysunek 122: Tabela skuteczności neg-sGCS dla populacji standardowej – anbn

Dla anbn (Rysunek 122) wszystkie algorytmy osiągają bardzo dobry poziom generalizacji bliski 100%.

bra1

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	422,80	0,00%	99,04%	98,09%
2	488,24	0,00%	99,04%	98,09%
3	484,35	0,00%	96,12%	95,21%
4	199,93	0,00%	99,04%	98,09%
5	261,57	0,00%	99,04%	98,09%
Average:	371,38	0,00%	98,46%	97,51%

Rysunek 123: Tabela skuteczności neg-sGCS dla populacji standardowej – bra1

bra1 (Rysunek 123) stoi na znacznie wyższym poziomie niż miało to miejsce w przypadku sGCS w poprzednich pracach [2] [8]. Mimo to podobnie jak przy zastosowaniu GCS mamy tutaj do czynienia z praktycznie zerowym rozpoznaniem zdań pozytywnym, co oznacza przeuczenie algorytmu na etapie uczenia. Poniżej (Rysunek 124) przedstawiono jeden z zestawów reguł uzyskiwany w wyniku działania algorytmu.

bra3

I => 'a'	(0.5)
Q => 'b'	(1)
<S> => <S>; <S>	(0.5)
<S> => I; Q	(0.5)
H => I; Q	(1)
I => I; H	(0.5)

Rysunek 124: Reguły uzyskane przez bra1

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	585,13	22,70%	98,85%	72,97%
2	570,40	3,61%	99,06%	66,62%
3	501,77	2,02%	98,96%	66,01%
4	219,90	4,91%	98,52%	66,70%
5	238,50	0,62%	99,83%	66,11%
Average:	423,14	6,77%	99,04%	67,68%

Rysunek 125: Tabela skuteczności neg-sGCS dla populacji standardowej – bra3

Dla neg-sGCS z bra3 (Rysunek 125) uzyskano wynik zbieżny z wynikami pozostałych algorytmów. Podobnie jak jego poprzednicy neg-sGCS nie jest w stanie w prawidłowy sposób wyuczyć się gramatyki, tworząc wysoce wyspecjalizowane reguły, co kończy się niską skutecznością w rozpoznawaniu zdań pozytywnych. Jest to jednak dosyć powszechna przypadłość, dotycząca również implementacji sGCS przedstawionej w pracy Pasieki [2].

pal2

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	2869,17	100,00%	100,00%	100,00%
2	1653,67	100,00%	97,76%	97,77%
3	1767,00	100,00%	98,52%	98,53%
4	1158,00	100,00%	100,00%	100,00%
5	1552,00	100,00%	100,00%	100,00%
Average:	1799,97	100,00%	99,26%	99,26%

Rysunek 126: Tabela skuteczności neg-sGCS dla populacji standardowej – pal2

pal2 to gramatyka, w której neg-sGCS odniósł zdecydowany sukces (Rysunek 126). Trzy z pięciu przebiegów kończą się odnalezieniem gramatyki w 100% poprawnej, pozostałe dwa przebiegi generują zestawy reguł zapewniające skuteczność rzędu 98%. Są to wyniki lepsze od obu poprzednich algorytmów.

toy

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1417,75	0,00%	99,09%	98,93%
2	1433,57	0,00%	99,87%	99,71%
3	1810,80	0,00%	99,96%	99,80%
4	551,00	0,00%	99,34%	99,18%
5	865,40	0,00%	99,81%	99,65%
Average:	1215,70	0,00%	99,61%	99,45%

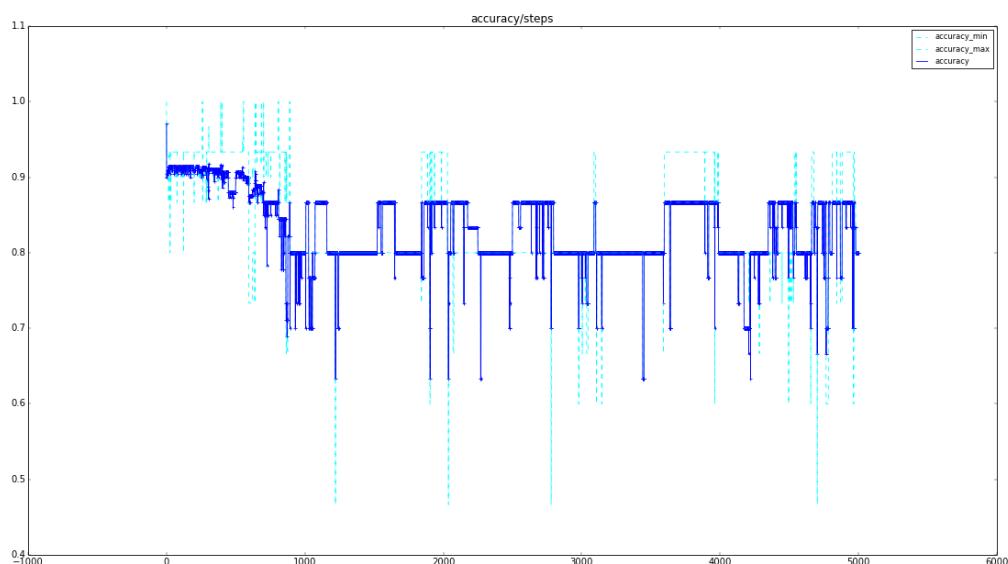
Rysunek 127: Tabela skuteczności neg-sGCS dla populacji standardowej – toy

Podobnie jak w przypadku poprzednich algorytmów neg-sGCS ma problem z poprawnym rozpoznaniem zdań pozytywnych (Rysunek 127). Pod pozornie wysokim wynikiem prawie 100% kryje się wysoce wyspecjalizowany zestaw klasyfikatorów, odpowiadający „Nie”

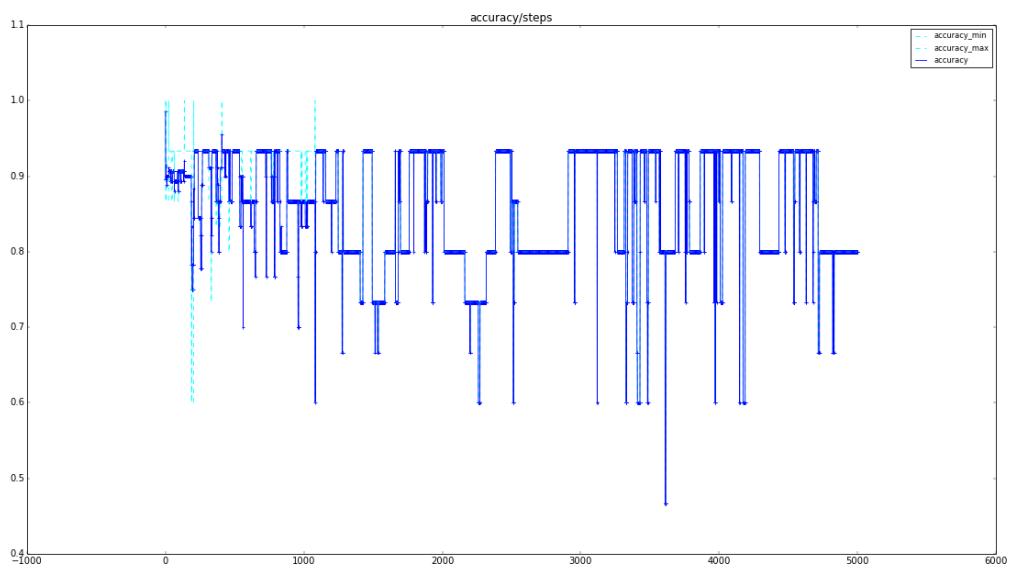
w większości przypadków i wysoki rezultat jest uzależniony od budowy zbioru testowego (którego prewalencja to jedyne 0,16%). Mimo to są to najlepsze wyniki spośród wszystkich algorytmów.

Nowy algorytm stanowi ciekawe rozszerzenie algorytmu sGCS. Mimo, że pozorne zdania pozytywne powinny wystarczać w procesie uczenia to widać, że użycie również zdań negatywnych w znacznym stopniu poprawia wynik. Wyniki generalizacji neg-sGCS plasują się między wynikami sGCS a GCS w przypadku prostszych gramatyk, w przypadku tych bardziej skomplikowanych zdała się czasami (w przypadku Tomita 6, Tomita 7, bra3 i pal2), że osiągnięty wynik jest lepszy niż poprzednich gramatyk. Niestety wykorzystanie zdań negatywnych znacznie wydłuża proces uczenia, sprawiając, że zazwyczaj zajmuje on co najmniej tyle czasu, ile w przypadku GCS. Nie da się jednak zaprzeczyć, że jeżeli interesuje nas wygenerowanie gramatyki stochastycznej to okazuje się on o wiele skuteczniejszym algorytmem niż sGCS.

Jak widać to po poniższych przebiegach wykresu Accuracy (Rysunek 128) od kroku ewolucyjnego (przykład przedstawia przebieg dla algorytmu Tomita 2, aczkolwiek ten wzorzec występuje również w przypadku innych gramatyk) algorytm neg-sGCS ma większą niż GCS (Rysunek 129) tendencję do „błędzenia” jeżeli nie zdoła znaleźć rozwiązania w pewnej ilości kroków. Należy w szczególności skupić się na części diagramu dotyczącej funkcjonowania obu algorytmów poniżej 1000 kroków – reszta przedstawia pojedyncze uruchomienia i jedyną interesującą fakt jaki można z niej wywnioskować, to że w przypadku neg-sGCS większa liczba uruchomień zdążyła „zabłądzić” w okolicy wyższych kroków. Tymczasem na przedziale [0;1000] widzimy, że neg-sGCS bardziej dąży średnią wartością do niższych wartości przystosowania niż jego poprzednik. Ta cecha algorytmu powoduje właśnie częstsze niepowodzenia w znalezieniu w stu procentach przystosowanego rozwiązania, z drugiej strony może być pożądana w przypadku gramatyk o większej złożoności, gdzie takie szersze poszukiwanie rozwiązania jest pożądane.



Rysunek 128: Accuracy neg-sGCS dla Tomita 2



Rysunek 129: Accuracy GCS dla Tomita 2

Reguły charakterystyczne

Tomita 1:	$\langle S \rangle \rightarrow \langle S \rangle \langle S \rangle$
Tomita 2:	$A \rightarrow a$ $B \rightarrow b$ $\langle S \rangle \rightarrow A B$
Tomita 3:	$C \rightarrow A A$ $\langle S \rangle \rightarrow C \langle S \rangle$ $A \rightarrow a$ $B \rightarrow b$ $\langle S \rangle \rightarrow B \langle S \rangle$
Tomita 4:	$\langle S \rangle \rightarrow A \langle S \rangle$ $A \rightarrow a$ $B \rightarrow b$ $\langle S \rangle \rightarrow H \langle S \rangle$ $H \rightarrow \langle S \rangle A$ $\langle S \rangle \rightarrow B B$
Tomita 5:	$\langle S \rangle \rightarrow A A$ $\langle S \rangle \rightarrow B B$ $A \rightarrow a$ $B \rightarrow b$
Tomita 6:	$\langle S \rangle \rightarrow \langle S \rangle \langle S \rangle$ $\langle S \rangle \rightarrow Q F$ $\langle S \rangle \rightarrow F Q$ $Q \rightarrow b$ $F \rightarrow a$ $F \rightarrow Q Q$
Tomita 7:	$B \rightarrow B B$ $B \rightarrow b$ $A \rightarrow AA$ $A \rightarrow a$
ab:	$\langle S \rangle \rightarrow \langle S \rangle \langle S \rangle$ $\langle S \rangle \rightarrow J A$ $\langle S \rangle \rightarrow A J$ $\langle S \rangle \rightarrow \langle S \rangle B$
anbn:	$\langle S \rangle \rightarrow A B$ $A \rightarrow a$ $B \rightarrow b$
bra1:	$\langle S \rangle \rightarrow \langle S \rangle \langle S \rangle$ $A \rightarrow A \langle S \rangle$ $A \rightarrow a$
bra3:	$\langle S \rangle \rightarrow A \langle S \rangle$ $\langle S \rangle \rightarrow \langle S \rangle \langle S \rangle$ $A \rightarrow a$
pal2:	$\langle S \rangle \rightarrow O A$ $O \rightarrow A \langle S \rangle$ $A \rightarrow a$
toy:	$\langle S \rangle \rightarrow A \langle S \rangle$ $\langle S \rangle \rightarrow P \langle S \rangle$ $P \rightarrow B C$ $A \rightarrow a$ $B \rightarrow b$ $C \rightarrow c$

Rysunek 130: Zastosowane podczas badań reguły charakterystyczne

Powyżej (Rysunek 130) przedstawiono zestaw reguł z jakimi uruchamiano wszystkie algorytmy w testach reguł charakterystycznych. Przez reguły charakterystyczne będziemy rozumieć reguły przedstawiające istotne elementy gramatyki, jak na przykład rekurencyjną rozwijalność symboli gramatyki Tomita 7 (reguły postaci $A \rightarrow A A$), reguły o wysokiej płodności, czy reguły których wygenerowanie

z jakiegoś powodu nie jest dla algorytmu proste i posiłkuje się on innym zestawem reguł, który jest mniej generyczny i powoduje zjawisko przeuczenia się. Pozostałe brakujące reguły początkowe zostały wylosowane przed każdym uruchomieniem uczącym. Podczas wyboru zdecydowano się na kładzenie nacisku na rekurencyjne reguły z symbolem startowym – jako trudniejsze do wygenerowania przez algorytm, a także reguły opisujące interesujące nas fragmenty gramatyki. Dla żadnej gramatyki nie zdefiniowano więcej niż połowy potrzebnych reguł, ograniczając się do minimum zdolnego do szybkiego zdobycia niezerowej wartości fitness. Reguły nie były w żaden sposób zabezpieczone przed usunięciem, więc zabezpieczenie populacji przed ich usunięciem poprzez upewnienie się, że już od samego początku będą posiadać niezerową wartość fitness wymusiło zazwyczaj zdefiniowanie całej grupy powiązanych reguł.

Ograniczenia czasowe nie pozwoliły na ukończenie piątej serii eksperymentów, dlatego każdej gramatyce towarzyszą tu tylko 4 pomyary.

GCS

Tomita 1

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1,00	100,00%	100,00%	100,00%
2	1,00	100,00%	100,00%	100,00%
3	1,00	100,00%	100,00%	100,00%
4	1,00	100,00%	100,00%	100,00%
Average:	1,00	100,00%	100,00%	100,00%

Rysunek 131: Tabela skuteczności GCS z regułami charakterystycznymi – tomita 1

Tomita 2

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	450,60	100,00%	100,00%	100,00%
2	492,72	100,00%	100,00%	100,00%
3	572,02	100,00%	100,00%	100,00%
4	521,56	100,00%	100,00%	100,00%
Average:	509,23	100,00%	100,00%	100,00%

Rysunek 132: Tabela skuteczności GCS z regułami charakterystycznymi – tomita 2

Gramatyki Tomita 1-2 (Rysunek 131, Rysunek 132) są bardzo prostymi gramatykami, obecnie osiągającymi już 100% przystosowania, więc dodanie reguł nie wpłynęło na ich proces uczenia (poza znacznym wydłużeniem uczenia Tomita 2 spowodowanym zmniejszeniem przestrzeni akceptowalnych rozwiązań).

Tomita 3

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1818,75	76,10%	79,58%	79,07%
2	1395,14	100,00%	100,00%	100,00%
3	1933,11	100,00%	100,00%	100,00%
4	1641,25	100,00%	100,00%	100,00%
Average:	1697,06	94,03%	94,90%	94,77%

Rysunek 133: Tabela skuteczności GCS z regułami charakterystycznymi – tomita 3

Względem poprzednich przeprowadzonych testów widać znaczną poprawę Tomita 3 (Rysunek 133). Dodanie części reguł odpowiedzialnych za generowanie parzystej liczby a po lewej stronie spowodowało również zwiększenie ilości uruchomień cykli uczących zakończonych sukcesem.

Tomita 4

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	588,87	91,26%	100,00%	96,90%
2	813,31	91,26%	100,00%	96,90%
3	588,87	91,26%	100,00%	96,90%
4	1178,58	91,26%	100,00%	96,90%
Average:	792,41	91,26%	100,00%	96,90%

Rysunek 134: Tabela skuteczności GCS z regułami charakterystycznymi – tomita 4

W przypadku Tomity 4 (Rysunek 134) dodano reguły wykorzystujące w znacznym stopniu symbol startowy gramatyki. Spowodowało to skrócenie czasu poszukiwania rozwiązania, ustabilizowanie skuteczności wyników na wysokim poziomie (które miały do tej pory dosyć spory rozrzut 50-100%).

Tomita 5

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1893,63	100,00%	100,00%	100,00%
2	1429,97	100,00%	100,00%	100,00%
3	1372,72	51,45%	62,84%	60,94%
4	1745,68	85,85%	65,22%	68,66%
Average:	1610,50	84,33%	82,02%	82,40%

Rysunek 135: Tabela skuteczności GCS z regułami charakterystycznymi – tomita 5

Dodanie reguł generujących ciąg a lub b o długości 2 spowodowały wygenerowanie stu procentowo przystosowanej gramatyki w dwóch z czterech przypadków (Rysunek 135). Zmniejszyło jednak również przestrzeń rozwiązań, co w efekcie doprowadziło do wydłużenia procesu uczenia.

Tomita 6

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	579,74	100,00%	100,00%	100,00%
2	588,15	66,83%	63,74%	64,77%
3	955,44	100,00%	100,00%	100,00%
4	992,82	66,83%	100,00%	88,94%
Average:	779,04	83,42%	90,94%	88,43%

Rysunek 136: Tabela skuteczności GCS z regułami charakterystycznymi – tomita 6

W przypadku Tomity 6 (Rysunek 136) zdecydowano się na dodanie reguły o wysokiej płodności. Poskutkowało to niewielkim wzrostem skuteczności oraz skróceniem czasu uczenia o połowę.

Tomita 7

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	532,03	100,00%	100,00%	100,00%
2	632,07	100,00%	100,00%	100,00%
3	392,13	100,00%	100,00%	100,00%
4	607,62	100,00%	100,00%	100,00%
Average:	540,96	100,00%	100,00%	100,00%

Rysunek 137: Tabela skuteczności GCS z regułami charakterystycznymi – tomita 7

Tomitę 7 również uruchomiono z regułami o wysokiej płodności (Rysunek 137). Dodanie ich skróciło trzykrotnie czas uczenia i spowodowało generowanie populacji klasyfikatorów zapewniających 100% przystosowania.

ab

Run	nEval	PosGen [%]	NegGen [%]	NGen [%]
1	1635,50	100,00%	100,00%	100,00%
2	2266,80	100,00%	100,00%	100,00%
3	1637,44	100,00%	100,00%	100,00%
4	2114,80	100,00%	100,00%	100,00%
Average:	1913,64	100,00%	100,00%	100,00%

Rysunek 138: Tabela skuteczności GCS z regułami charakterystycznymi – ab

Dodanie reguł o wysokiej płodności spowodowało podwojenie czasu uczenia algorytmu dla ab, równocześnie zapewniając wygenerowanie gramatyki o 100% przystosowania (Rysunek 138).

anbn

Run	nEval	PosGen [%]	NegGen [%]	NGen [%]
1	1660,92	100,00%	100,00%	100,00%
2	1607,36	100,00%	99,84%	99,84%
3	1610,96	100,00%	100,00%	100,00%
4	1513,92	100,00%	100,00%	100,00%
Average:	1598,29	100,00%	99,96%	99,96%

Rysunek 139: Tabela skuteczności GCS z regułami charakterystycznymi – anbn

Ze względu na prostą budowę zdecydowano się na dodanie jedynie trzech reguł charakterystycznych do anbn Rysunek 139. Wybór padł na reguły umożliwiające wyprowadzenie najprostszych zdań. Nie wpłynęło to w żadnym stopniu na proces uczenia, poza niewielkim wydłużeniem potrzebnego czasu ze względu na okrojenie przestrzeni akceptowalnych rozwiązań.

bra1

Run	nEval	PosGen [%]	NegGen [%]	NGen [%]
1	2,42	0,00%	99,04%	98,09%
2	2,42	0,00%	99,04%	98,09%
3	2,00	0,00%	99,04%	98,09%
4	2,00	0,00%	99,04%	98,09%
Average:	2,21	0,00%	99,04%	98,09%

*Rysunek 140: Tabela skuteczności GCS z regułami charakterystycznymi – bra1***bra3**

Run	nEval	PosGen [%]	NegGen [%]	NGen [%]
1	363,33	1,02%	99,66%	66,13%
2	337,06	2,12%	99,00%	66,07%
3	249,55	1,98%	99,23%	66,18%
4	238,98	0,73%	99,42%	65,87%
Average:	297,23	1,46%	99,33%	66,06%

Rysunek 141: Tabela skuteczności GCS z regułami charakterystycznymi – bra3

Dodanie reguł wysoce płodnych reguł generujących ciągi liter ‘a’ nie wpłynęły na uczenie bra1 (Rysunek 140) oraz bra3 (Rysunek 141).

pal2

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	3848,00	100,00%	99,70%	99,70%
2	1622,00	100,00%	98,94%	98,94%
3	nan	33,86%	96,18%	95,94%
4	1258,00	100,00%	100,00%	100,00%
Average:	2242,67	83,47%	98,71%	98,65%

Rysunek 142: Tabela skuteczności GCS z regułami charakterystycznymi – pal2

Dodanie reguł generujących ciągi a w sposób „palindromiczny” (tj. dodając ‘a’ po lewej stronie dodajemy je też po stronie prawej) spowodowało niewielki wzrost skuteczności algorytmu dla pal2 (Rysunek 142).

toy

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	2258,33	0,00%	99,89%	99,73%
2	1301,00	0,00%	99,91%	99,75%
3	1948,00	0,00%	99,98%	99,82%
4	2564,50	0,00%	99,90%	99,74%
Average:	2017,96	0,00%	99,92%	99,76%

Rysunek 143: Tabela skuteczności GCS z regułami charakterystycznymi – toy

W przypadku toy wykorzystano wysoko umieszczone w drzewach derywacji produkcje, których bezpośredni sąsiedzi produkują symbole terminalne. Spowodowało to nieznaczną poprawę, nie rozwiązuje problemu rozpoznawania zdań pozytywnych (Rysunek 143).

sGCS**Tomita 1**

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1,00	100,00%	100,00%	100,00%
2	1,00	100,00%	100,00%	100,00%
3	1,00	100,00%	100,00%	100,00%
4	1,00	100,00%	100,00%	100,00%
Average:	1,00	100,00%	100,00%	100,00%

Rysunek 144: Tabela skuteczności sGCS z regułami charakterystycznymi – tomita 1

Tomita 2

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	2003,89	100,00%	100,00%	100,00%
2	1943,88	85,71%	99,95%	99,95%
3	1370,39	100,00%	100,00%	100,00%
4	1940,65	100,00%	100,00%	100,00%
Average:	1814,70	96,43%	99,99%	99,99%

Rysunek 145: Tabela skuteczności sGCS z regułami charakterystycznymi – tomita 2

Tomita 3

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1,00	100,00%	0,00%	14,42%
2	1,00	100,00%	0,00%	14,42%
3	1,00	100,00%	0,00%	14,42%
4	1,00	100,00%	0,00%	14,42%
Average:	1,00	100,00%	0,00%	14,42%

Rysunek 146: Tabela skuteczności sGCS z regulami charakterystycznymi – tomita 3

Tomita 4

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1,00	100,00%	0,00%	35,47%
2	1,00	100,00%	0,00%	35,47%
3	1,00	100,00%	0,00%	35,47%
4	1,00	100,00%	0,00%	35,47%
Average:	1,00	100,00%	0,00%	35,47%

Rysunek 147: Tabela skuteczności sGCS z regulami charakterystycznymi – tomita 4

Tomita 5

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	62,16	100,00%	80,00%	83,33%
2	117,16	100,00%	80,00%	83,33%
3	75,70	99,99%	0,02%	16,68%
4	68,56	100,00%	60,00%	66,67%
Average:	80,90	100,00%	55,01%	62,50%

Rysunek 148: Tabela skuteczności sGCS z regulami charakterystycznymi – tomita 5

Tomita 6

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	67,72	100,00%	0,01%	33,34%
2	58,60	100,00%	0,01%	33,34%
3	58,72	100,00%	100,00%	100,00%
4	64,68	100,00%	0,01%	33,34%
Average:	62,43	100,00%	25,01%	50,01%

Rysunek 149: Tabela skuteczności sGCS z regulami charakterystycznymi – tomita 6

Tomita 7

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1,00	100,00%	0,00%	3,84%
2	1,00	100,00%	0,00%	3,84%
3	1,00	100,00%	0,00%	3,84%
4	1,00	100,00%	0,00%	3,84%
Average:	1,00	100,00%	0,00%	3,84%

Rysunek 150: Tabela skuteczności sGCS z regulami charakterystycznymi – tomita 7

Dodanie reguł charakterystycznych do populacji reguł stochastycznych nie wpłynęło w żadnym stopniu na proces generowania gramatyk Tomita L1-L7 (od Rysunek 144 do Rysunek 150). Reguły te były z reguły dosyć szybko usuwane.

ab

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	17,20	100,00%	2,69%	9,67%
2	14,20	100,00%	100,00%	100,00%
3	15,90	100,00%	100,00%	100,00%
4	16,80	100,00%	100,00%	100,00%
Average:	16,03	100,00%	75,67%	77,42%

Rysunek 151: Tabela skuteczności sGCS z regulami charakterystycznymi – ab

W przypadku gramatyki ab dodanie reguł charakterystycznych spowodowało znaczną poprawę rozpoznania zdań negatywnych (Rysunek 151).

anbn

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	30,90	100,00%	99,99%	99,99%
2	31,30	100,00%	75,01%	75,01%
3	42,80	100,00%	98,79%	98,79%
4	31,46	100,00%	99,71%	99,71%
Average:	34,12	100,00%	93,38%	93,38%

Rysunek 152: Tabela skuteczności sGCS z regulami charakterystycznymi – anbn

bra1

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	20,84	100,00%	84,14%	84,29%
2	23,50	100,00%	96,75%	96,78%
3	20,00	100,00%	75,72%	75,95%
4	18,80	100,00%	100,00%	100,00%
Average:	20,79	100,00%	89,15%	89,26%

Rysunek 153: Tabela skuteczności sGCS z regulami charakterystycznymi – bra1

bra3

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	80,38	0,00%	99,37%	65,59%
2	53,08	2,97%	69,95%	47,18%
3	112,64	0,00%	99,76%	65,85%
4	36,88	9,15%	72,24%	53,34%
Average:	70,75	3,03%	85,33%	57,99%

Rysunek 154: Tabela skuteczności sGCS z regulami charakterystycznymi – bra3

W przypadku gramatyk anbn (Rysunek 152), bra1 (Rysunek 153) oraz bra3 (Rysunek 154) nie nastąpiła zauważalna poprawa. bra1 zdaje się mieć trochę lepsze średnie wyniki, aczkolwiek nie odbiegają one zbytnio od tych osiągniętych przed dodanie reguł.

pal2

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	14,70	100,00%	93,18%	93,21%
2	50,20	100,00%	0,23%	0,61%
3	26,60	100,00%	2,41%	2,78%
4	52,60	100,00%	52,02%	52,20%
Average:	36,03	100,00%	36,96%	37,20%

Rysunek 155: Tabela skuteczności sGCS z regulami charakterystycznymi – pal2

Dosyć interesujące zachowanie, a mianowicie pogorszenie skuteczności algorytmu można zaobserwować w przypadku pal2 (Rysunek 155). Okazało się, że algorytm nadal dosyć duże prawdopodobieństwa regułom charakterystycznym, i ostatecznie generował gramatyki z przeważającą ilością symboli ‘a’, będące zawsze nadzbiorem populacji palindromów nad alfabetem $\{a, b\}$, co z kolei spowodowało niskie rezultaty NegGen, a co za tym idzie niski wynik NGen (ze względu na niską prewalencję 0,39% zbioru testowego). Poniżej (Rysunek 156) przedstawiono gramatykę wygenerowaną przez najgorsze pod względem rozpoznawania zdanie negatywnych, a co za tym idzie rozpoznawaniem wszystkich zdań, drugie uruchomienie:

A => 'a'	(1)
C => 'b'	(1)
<S> => A; A	(0.17)
<S> => C; C	(0.23)
O => A; <S>	(0.62)
<S> => O; O	(0.24)
<S> => <S>; C	(0.15)
O => C; <S>	(0.38)
<S> => O; A	(0.14)
<S> => <S>; <S>	(0.05)
<S> => C; <S>	(0.02)

Rysunek 156: Reguły wygenerowane dla pal2

toy

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	51,67	53,33%	98,14%	98,07%
2	nan	0,00%	99,93%	99,77%
3	104,67	59,05%	94,11%	94,06%
4	56,00	0,00%	99,60%	99,44%
Average:	70,78	28,10%	97,95%	97,84%

Rysunek 157: Tabela skuteczności sGCS z regułami charakterystycznymi – toy

Dodanie reguł zapewniających krótką ścieżkę z symbolu startowego do terminalnych skróciło czas niezbędny do znalezienia rozwiązania oraz poprawiło nieco skuteczność algorytmu dla zdań pozytywnych (Rysunek 157).

Reguły charakterystyczne w niewielkim stopniu wpłynęły na proces uczenia sGCS. Było to głównie spowodowane dosyć szybkim wyrzucaniem takich reguł. Czasami odnosiły one efekt odwrotny od zamierzonego (na przykład w przypadku pal2 algorytm zbyt na nich polegał i nie dysponując zdaniami negatywnymi, które mogłyby wyprowadzić z błędu zazwyczaj wygenerował nadzbiór szukanej gramatyki). Na działanie tego algorytmu mogłyby mieć wpływ zablokowanie możliwości usuwania reguł charakterystycznych z populacji reguł. Niestety zabrakło czasu na przeprowadzenie badań pod tym kątem.

neg-sGCS

Tomita 1

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1,00	100,00%	100,00%	100,00%
2	1,00	100,00%	100,00%	100,00%
3	1,00	100,00%	100,00%	100,00%
4	1,00	100,00%	100,00%	100,00%
Average:	1,00	100,00%	100,00%	100,00%

Rysunek 158: Tabela skuteczności neg- sGCS z regułami charakterystycznymi – tomita 1

Tomita 2

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	869,74	100,00%	100,00%	100,00%
2	709,78	100,00%	99,98%	99,98%
3	722,32	100,00%	100,00%	100,00%
4	670,54	100,00%	99,62%	99,62%
Average:	743,10	100,00%	99,90%	99,90%

Rysunek 159: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – tomita 2

Tomita 3

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	nan	100,00%	0,00%	14,42%
2	1,00	100,00%	100,00%	100,00%
3	6,00	100,00%	100,00%	100,00%
4	1,00	100,00%	100,00%	100,00%
Average:	2,67	100,00%	75,00%	78,61%

Rysunek 160: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – tomita 3

Tomita 4

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	384,88	91,26%	100,00%	96,90%
2	520,91	91,26%	100,00%	96,90%
3	303,20	91,26%	100,00%	96,90%
4	480,50	91,26%	100,00%	96,90%
Average:	422,37	91,26%	100,00%	96,90%

Rysunek 161: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – tomita 4

Jeżeli było to możliwe to dodanie reguł charakterystycznych dla doświadczeń z gramatykami Tomita 1-4 (Rysunek 158 do Rysunek 161) spowodowało poprawę algorytmu neg-sGCS, dzięki czemu prawie wszędzie udało mu się uzyskać 100% przystosowania. Jedyny słaby rezultat wystąpił przy pierwszym przebiegu dla gramatyki Tomita 3, aczkolwiek jest to spowodowane niezdolnością algorytmu do wyuczenia się gramatyki (nie nastąpiło to w żadnym z 50 cykli uczących).

Tomita 5

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1285,27	66,53%	100,00%	94,42%
2	1726,89	77,30%	68,83%	70,25%
3	1827,58	68,90%	70,87%	70,55%
4	1515,50	40,20%	99,05%	89,24%
Average:	1588,81	63,23%	84,69%	81,12%

Rysunek 162: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – tomita 5

Tomita 6

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	614,77	66,83%	100,00%	88,94%
2	473,82	66,83%	100,00%	88,94%
3	916,06	100,00%	100,00%	100,00%
4	1056,03	66,83%	100,00%	88,94%
Average:	765,17	75,12%	100,00%	91,71%

Rysunek 163: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – tomita 6

Jak widać Tomita 5 i 6 nie osiągnęły praktycznie żadnej poprawy.

Tomita 7

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	3,75	100,00%	100,00%	100,00%
2	1049,29	100,00%	100,00%	100,00%
3	920,09	100,00%	100,00%	100,00%
4	211,09	100,00%	100,00%	100,00%
Average:	546,06	100,00%	100,00%	100,00%

Rysunek 164: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – tomita 7

Podobnie jak w przypadku GCS zastosowanie reguł charakterystycznych powoduje zmniejszenie wartości nEvals oraz generowanie zestawów reguł zapewniających 100% przystosowania w teście generalizacji (Rysunek 162 do Rysunek 164).

ab

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	810,57	100,00%	100,00%	100,00%
2	1741,78	100,00%	100,00%	100,00%
3	578,43	100,00%	100,00%	100,00%
4	89,50	100,00%	100,00%	100,00%
Average:	805,07	100,00%	100,00%	100,00%

Rysunek 165: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – ab

anbn

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1536,55	100,00%	100,00%	100,00%
2	1934,55	100,00%	99,49%	99,49%
3	1730,27	100,00%	100,00%	100,00%
4	1580,14	100,00%	100,00%	100,00%
Average:	1695,38	100,00%	99,87%	99,87%

Rysunek 166: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – anbn

bra1

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	2,00	0,00%	99,04%	98,09%
2	2,00	0,00%	99,04%	98,09%
3	2,00	0,00%	99,04%	98,09%
4	2,00	0,00%	99,04%	98,09%
Average:	2,00	0,00%	99,04%	98,09%

Rysunek 167: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – bra1

bra3

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	281,85	0,67%	99,57%	65,96%
2	202,52	0,69%	99,35%	65,81%
3	267,44	0,61%	99,60%	65,96%
4	224,59	3,28%	99,54%	66,82%
Average:	244,10	1,31%	99,52%	66,14%

Rysunek 168: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – bra3

pal2

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	301,67	100,00%	100,00%	100,00%
2	1134,33	100,00%	99,78%	99,78%
3	462,00	100,00%	100,00%	100,00%
4	223,20	100,00%	100,00%	100,00%
Average:	530,30	100,00%	99,95%	99,95%

Rysunek 169: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – pal2

Gramatyki ab (Rysunek 165), anbn (Rysunek 166), bra1 (Rysunek 167), bra3 (Rysunek 168) oraz pal2 (Rysunek 169) pozostają na względnie tym samym poziomie skuteczności. Zmniejszeniu za to z pewnością ulega czas niezbędny do wyuczenia tych gramatyk, w szczególności w przypadku bra1 (spadek z nEvals=371 do 2). Powraca też znany niestety z symulacji GCS problem wyuczenia zdań pozytywnych gramatyk nawiasowych (bra1, bra3).

toy

Run	nEvals	PosGen [%]	NegGen [%]	NGen [%]
1	1932,75	0,00%	99,88%	99,72%
2	1047,00	0,00%	99,94%	99,78%
3	1327,00	0,00%	99,94%	99,78%
4	1347,00	0,00%	99,59%	99,44%
Average:	1413,44	0,00%	99,84%	99,68%

Rysunek 170: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – toy

W przypadku toy (Rysunek 170) algorytm osiągnął najlepsze dotychczas przystosowanie NGen. Niestety, dzieje się to kosztem przystosowania PosGen, które tak jak w przypadku sGCS spadło do 0%.

Jak widać połączenie dodawania reguł charakterystycznych do początkowej populacji oraz stosowanie algorytmu neg-sGCS stanowi dobre połączenie i daje bardzo dobre rezultaty.

Podsumowanie

Algorytmy generujące klasyfikatory gramatyczne są interesujące i ułatwiają zrozumienie skomplikowanych zależności, jakimi są powiązane występujące w świecie rzeczywistym zjawiska. Kolejne prace rozwijają dalsze mechanizmy oraz badają wpływ poszczególnych elementów na funkcjonowanie całego mechanizmu. Praca ta pokazała potencjał stosowania zdań negatywnych w procesie uczenia oraz zaproponowała sposób obsługi takich zdań przy pomocy prostej implementacji neg-sGCS. Jest to najprawdopodobniej kierunek, w którym powinien pójść dalszy rozwój algorytmu. Niestety neg-sGCS nie radzi sobie idealnie z każdą gramatyką i zdarzają mu się zachowania odwrotne do tych spotykanych w sGCS – a konkretnie czasami generuje on gramatyki dobrze rozpoznające zdania negatywne, mające problemy z rozpoznawaniem zdań pozytywnych. Kluczowy może tu być odpowiedni dobór parametrów – być może należałoby zmniejszyć wagę rozbioru negatywnego zdania w przypadku neg-sGCS.

Inną możliwością jest równoczesne symulowanie kilku populacji i próba wymiany informacji pomiędzy nimi przy pomocy genetycznego algorytmu wyspowego [9] [10]. Można by wówczas uruchomić obok siebie kilka algorytmów, na przykład sGCS i neg-sGCS i pozwalać im na wymianę informacji. Podejście to jednak wymagałoby ustalenia sposobu wymiany reguł, które będąc regułami stworzonymi w całkowicie innej populacji inną ścieżką ewolucji mogłyby się okazać nieprzenaszalne pomiędzy populacjami. Można by tu zastanowić się nad zastosowaniem podejścia Pittsburg lub hybrydowego [2] w algorytmie genetycznym lub jego części migracyjnej w celu przesyłania całych zbiorów reguł zamiast pojedynczych osobników.

Równoczesne generowanie kilku populacji może również sprawiać poważny problem czasowy, jako że na wygenerowanie i przetestowanie pojedynczej gramatyki trzeba było czekać czasami nawet kilkanaście godzin. Taka ewolucja równoległa daje jednak pole opisu do dalszego zrównoleglenia algorytmu, w wyniku czego poprzez dysponowanie odpowiednią architekturą obliczeniową (na przykład własną chmurą) można zredukować wpływ dodawania kolejnych populacji.

Interesujące rezultaty dało również wykorzystanie reguł charakterystycznych w procesie uczenia. Zastosowanie ich w przypadku GCS oraz neg-sGCS zazwyczaj poskutkowało poprawą działania tych algorytmów. Wykorzystanie ich w przypadku sGCS nie dało dużego zysku, co było czasami spowodowane przez przedwczesne usunięcie tych reguł w procesie uczenia, a czasami uniwersalność i płodność tych reguł zwiększyła szansę sGCS na stworzenie nadgramatyki zamiast poszukiwanej (jak miało to miejsce na przykład w przypadku badań nad gramatyką pal2). Tak czy inaczej generalnie dodawanie reguł charakterystycznych do populacji początkowej znacznie poprawia rezultaty działania algorytmów i jest to praktyka, którą warto stosować. Interesującym zagadnieniem mogłoby być zautomatyzowanie wyszukiwania reguł charakterystycznych, dzięki czemu taki zestaw reguł mógłby być w jakiś sposób wstępnie określany na podstawie cech gramatyki odnalezionych przez inny algorytm uruchamiany przed systemem sGCS.

Literatura

- [1] O. Unold, Ewolucyjne wnioskowanie gramatyczne, Wrocław: Oficyna wydawnicza Politechniki Wrocławskiej, 2006.
- [2] G. Pasieka, Praca magisterska. Nowe mechanizmy adaptacji w modelu sGCS, Wrocław, 2014.
- [3] L. Pacholski i W. Charatonik, Logika dla informatyków. Materiały do zajęć, ósme poprawione red., Wrocław: Uniwersytet Wrocławski, 2010.
- [4] M. Skrzypczyk, Praca magisterska. LibGCS - biblioteka dla systemu xGCS, Wrocław, 2011.
- [5] P. Skórzewski, Praca magisterska. Gramatyki i automaty probabistyczne, Poznań, 2010.
- [6] M. E. Błędowski, Projekt inżynierski: Implementacja uczącego się systemu klasyfikującego GCS, Wrocław, 2012.
- [7] J. Arabas, Wykłady z algorytmów ewolucyjnych, Warszawa: WNT, 2009.
- [8] M. Kępa, Praca magisterska: Stochastyczny model GCS, Wrocław, 2008.
- [9] A. Reza and Z. Koorush, "A multilevel evolutionary algorithm for optimizing numerical functions," *International Journal of Industrial Engineering Computations*, vol. 2, 2011.
- [10] Y. Tsuruokazy and J. Tsujii, Iterative CKY parsing for Probabilistic Context-Free Grammars, First International Joint Conference, Hainan Island, China: Revised Selected Papers, 2004.

Spis ilustracji

Rysunek 1: Algorytm CYK.....	13
Rysunek 2: Przykładowa gramatyka	13
Rysunek 3: Przykładowa tabela CYK	14
Rysunek 4: Przykładowa tabela CYK z powtarzającymi się produkcjami	15
Rysunek 5: Algorytm GCS	16
Rysunek 6: Procedura IndukujGramatykę.....	16
Rysunek 7: Procedura parsowania zdania	17
Rysunek 8: Algorytm ścisiku	18
Rysunek 9: Operator pokrycia terminalnego	18
Rysunek 10: Operator pokrycia uniwersalnego.....	19
Rysunek 11: Operator pokrycia startowego	19
Rysunek 12: Operator pokrycia agresywnego	19
Rysunek 13: Operator pokrycia pełnego	19
Rysunek 14: Algorytm genetyczny	20
Rysunek 15: Operator inwersji.....	20
Rysunek 16: Operator mutacji	21
Rysunek 17: Operator krzyżowania	21
Rysunek 18: Algorytm korekcji gramatyki	21
Rysunek 19: Algorytm usuwania produkcji nieproduktywnych.....	22
Rysunek 20: Algorytm usuwania produkcji nieosiągalnych.....	22
Rysunek 21: Przykładowa gramatyka stochastyczna	23
Rysunek 22: Przykładowa tabela CYK+	23
Rysunek 23: Prawdopodobieństwo przy podejściu Viterbi	23
Rysunek 24: Prawdopodobieństwo przy podejściu Baum-Welch	24
Rysunek 25: Przechowywanie śladu drzewa derywacji	24
Rysunek 26: Algorytm Traceback.....	24
Rysunek 27: Wzór na normalizację prawdopodobieństwa	25
Rysunek 28: Przystosowanie w sGCS.....	25
Rysunek 29: Estymacja prawdopodobieństw w sGCS	25
Rysunek 30: Diagram przypadków użycia aplikacji	27
Rysunek 31: Aplikacja w trakcie użytkowania	27
Rysunek 32: Okno menu głównego	28
Rysunek 33: Okno przygotowania danych wejściowych	29
Rysunek 34: Edycja konfiguracji	31
Rysunek 35: Okno System Status	32
Rysunek 36: Edytor populacji	33
Rysunek 37: Menu Scheduler	34
Rysunek 38: Okno przebiegu algorytmu	35
Rysunek 39: Przykładowy diagram generowany przez system	36
Rysunek 40: Przykładowy plik wynikowej populacji	37
Rysunek 41: Przykładowy plik podglądu końcowej populacji	38
Rysunek 42: Przykładowy plik podsumowania przebiegu algorytmu	39
Rysunek 43: Przykładowy plik danych wejściowych.....	39
Rysunek 44: Przykładowy plik konfiguracji	40
Rysunek 45: Diagram modułowy biblioteki sgcs	41
Rysunek 46: Moduł sgcs.utils	42
Rysunek 47: Moduł sgcs.gui.gui_manager	44
Rysunek 48: Moduł sgcs.gui.console_fetcher	45
Rysunek 49: Moduł sgcs.gui.dynamic_gui	46
Rysunek 50: Moduł sgcs.gui.generic_widget	48
Rysunek 51: Moduł sgcs.gui.main_app	49
Rysunek 52: Moduł sgcs.gui.async_progress_dialog	50
Rysunek 53: Moduł sgcs.gui.inout_data_lookup	51
Rysunek 54: Moduły sgcs.options_configurator i sgcs.options_configurator_parts	53
Rysunek 55: Moduł sgcs.gui.population_editor	56
Rysunek 56: Moduł sgcs.gui.system_status	59
Rysunek 57: Moduł sgcs.gui.scheduler	61
Rysunek 58: Moduł sgcs.gui.runner	64
Rysunek 59: Moduł sgcs.gui.proxy	68
Rysunek 60: Moduł sgcs.executors.simulation_executor	70
Rysunek 61: Moduł sgcs.executors.population_executor	72
Rysunek 62: Zależności modułu sgcs.algorithm	73
Rysunek 63: Moduł sgcs.algorithm.gcs_simulator	74
Rysunek 64: Moduł sgcs.algorithm.run_estimator	75
Rysunek 65: Moduł sgcs.algorithm.gcs_runner	76
Rysunek 66: Moduł sgcs.grammar_estimator	79
Rysunek 67: Moduł sgcs.evolution	81
Rysunek 68: Moduł sgcs.rule_adding	83
Rysunek 69: Moduł sgcs.statistics.grammar_statistics (tylko klasy przystosowania Pasieki)	85
Rysunek 70: Moduł sgcs.statistics.grammar_statistics (tylko klasy przystosowania klasycznego)	86
Rysunek 71: Diagram zależności modułów przestrzeni sgcs.induction	89
Rysunek 72: Moduł sgcs.induction.traceback	90
Rysunek 73: Moduł sgcs.induction.cyk_executors	91
Rysunek 74: Moduł sgcs.induction.coverage_operators	94
Rysunek 75: Moduł sgcs.induction.detector	96

Rysunek 76: Moduł sgcs.induction.grammar_corrector.....	97
Rysunek 77: Moduł sgcs.induction.environment	98
Rysunek 78: Moduł sgcs.induction.production	100
Rysunek 79: Moduł sgcs.core	102
Rysunek 80: Moduł sgcs.datalayer.....	104
Rysunek 81: Tabela skuteczności GCS dla populacji standardowej – Tomita 1	107
Rysunek 82: Tabela skuteczności GCS dla populacji standardowej – Tomita 2	107
Rysunek 83: Tabela skuteczności GCS dla populacji standardowej – Tomita 3	108
Rysunek 84: Accuracy dla kolejnych kroków przebiegu algorytmu (GCS, Tomita 3).....	108
Rysunek 85: Tabela skuteczności GCS dla populacji standardowej – Tomita 4	108
Rysunek 86: Tabela skuteczności GCS dla populacji standardowej – Tomita 5	109
Rysunek 87: Tabela skuteczności GCS dla populacji standardowej – Tomita 6	109
Rysunek 88: Tabela skuteczności GCS dla populacji standardowej – Tomita 7	109
Rysunek 89: Tabela skuteczności GCS dla populacji standardowej – ab	109
Rysunek 90: Tabela skuteczności GCS dla populacji standardowej – anbn	110
Rysunek 91: Tabela skuteczności GCS dla populacji standardowej – bra1	110
Rysunek 92: Tabela skuteczności GCS dla populacji standardowej – bra3.....	110
Rysunek 93: Tabela skuteczności GCS dla populacji standardowej – pal2.....	110
Rysunek 94: Tabela skuteczności GCS dla populacji standardowej – toy.....	111
Rysunek 95: "Książkowy" zestaw reguł.....	111
Rysunek 96: Wygenerowany zestaw reguł.....	111
Rysunek 97: Accuracy bra3	112
Rysunek 98: Specificity bra3	112
Rysunek 99:Tabela skuteczności sGCS dla populacji standardowej – tomita 1	113
Rysunek 100: Tabela skuteczności sGCS dla populacji standardowej – tomita 2	113
Rysunek 101: Tabela skuteczności sGCS dla populacji standardowej – tomita 3	113
Rysunek 102: Tabela skuteczności sGCS dla populacji standardowej – tomita 4	113
Rysunek 103: Tabela skuteczności sGCS dla populacji standardowej – tomita 5	114
Rysunek 104: Diagram Accuracy dla Tomita 5	114
Rysunek 105: Tabela skuteczności sGCS dla populacji standardowej – tomita 6	114
Rysunek 106: Tabela skuteczności sGCS dla populacji standardowej – tomita 7	115
Rysunek 107:Tabela skuteczności sGCS dla populacji standardowej – ab	115
Rysunek 108: Tabela skuteczności sGCS dla populacji standardowej – anbn.....	115
Rysunek 109: Tabela skuteczności sGCS dla populacji standardowej – bra1	116
Rysunek 110: Tabela skuteczności sGCS dla populacji standardowej - bra3	116
Rysunek 111: Reguły wygenerowane dla bra3	116
Rysunek 112: Tabela skuteczności sGCS dla populacji standardowej – pal2	116
Rysunek 113: Tabela skuteczności sGCS dla populacji standardowej – toy	117
Rysunek 114: Tabela skuteczności neg-sGCS dla populacji standardowej – tomita 1	117
Rysunek 115: Tabela skuteczności neg-sGCS dla populacji standardowej – tomita 2	117
Rysunek 116: Tabela skuteczności neg-sGCS dla populacji standardowej – tomita 3	118
Rysunek 117: Tabela skuteczności neg-sGCS dla populacji standardowej – tomita 4	118
Rysunek 118: Tabela skuteczności neg-sGCS dla populacji standardowej – tomita 5	118
Rysunek 119: Tabela skuteczności neg-sGCS dla populacji standardowej – tomita 6	118
Rysunek 120: Tabela skuteczności neg-sGCS dla populacji standardowej – tomita 7	119
Rysunek 121: Tabela skuteczności neg-sGCS dla populacji standardowej – ab.....	119
Rysunek 122: Tabela skuteczności neg-sGCS dla populacji standardowej – anbn.....	119
Rysunek 123: Tabela skuteczności neg-sGCS dla populacji standardowej – bra1	119
Rysunek 124: Reguły uzyskane przez bra1	120
Rysunek 125: Tabela skuteczności neg-sGCS dla populacji standardowej – bra3	120
Rysunek 126: Tabela skuteczności neg-sGCS dla populacji standardowej – pal2.....	120
Rysunek 127: Tabela skuteczności neg-sGCS dla populacji standardowej – toy	120
Rysunek 128: Accuracy neg-sGCS dla Tomita 2	121
Rysunek 129: Accuracy sGCS dla Tomita 2	122
Rysunek 130: Zastosowane podczas badań reguły charakterystyczne	123
Rysunek 131: Tabela skuteczności GCS z regułami charakterystycznymi – tomita 1	124
Rysunek 132: Tabela skuteczności GCS z regułami charakterystycznymi – tomita 2	124
Rysunek 133: Tabela skuteczności GCS z regułami charakterystycznymi – tomita 3	124
Rysunek 134: Tabela skuteczności GCS z regułami charakterystycznymi – tomita 4	125
Rysunek 135: Tabela skuteczności GCS z regułami charakterystycznymi – tomita 5	125
Rysunek 136: Tabela skuteczności GCS z regułami charakterystycznymi – tomita 6	125
Rysunek 137: Tabela skuteczności GCS z regułami charakterystycznymi – tomita 7	125
Rysunek 138: Tabela skuteczności GCS z regułami charakterystycznymi – ab.....	126
Rysunek 139: Tabela skuteczności GCS z regułami charakterystycznymi – anbn	126
Rysunek 140: Tabela skuteczności GCS z regułami charakterystycznymi – bra1	126
Rysunek 141: Tabela skuteczności GCS z regułami charakterystycznymi – bra3	126
Rysunek 142: Tabela skuteczności GCS z regułami charakterystycznymi – pal2.....	127
Rysunek 143: Tabela skuteczności GCS z regułami charakterystycznymi – toy.....	127
Rysunek 144: Tabela skuteczności sGCS z regulami charakterystycznymi – tomita 1	127
Rysunek 145: Tabela skuteczności sGCS z regulami charakterystycznymi – tomita 2	127
Rysunek 146: Tabela skuteczności sGCS z regulami charakterystycznymi – tomita 3	128
Rysunek 147: Tabela skuteczności sGCS z regulami charakterystycznymi – tomita 4	128
Rysunek 148: Tabela skuteczności sGCS z regulami charakterystycznymi – tomita 5	128
Rysunek 149: Tabela skuteczności sGCS z regulami charakterystycznymi – tomita 6	128
Rysunek 150: Tabela skuteczności sGCS z regulami charakterystycznymi – tomita 7	128
Rysunek 151: Tabela skuteczności sGCS z regulami charakterystycznymi – ab	129
Rysunek 152: Tabela skuteczności sGCS z regulami charakterystycznymi – anbn	129
Rysunek 153: Tabela skuteczności sGCS z regulami charakterystycznymi – bra1	129

Rysunek 154: Tabela skuteczności sGCS z regulami charakterystycznymi – bra3	129
Rysunek 155: Tabela skuteczności sGCS z regulami charakterystycznymi – pal2	129
Rysunek 156: Reguły wygenerowane dla pal2.....	130
Rysunek 157: Tabela skuteczności sGCS z regulami charakterystycznymi – toy	130
Rysunek 158: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – tomita 1	130
Rysunek 159: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – tomita 2	131
Rysunek 160: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – tomita 3	131
Rysunek 161: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – tomita 4	131
Rysunek 162: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – tomita 5	131
Rysunek 163: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – tomita 6	131
Rysunek 164: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – tomita 7	132
Rysunek 165: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – ab.....	132
Rysunek 166: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – anbn.....	132
Rysunek 167: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – bra1	132
Rysunek 168: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – bra3	132
Rysunek 169: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – pal2	133
Rysunek 170: Tabela skuteczności neg- sGCS z regulami charakterystycznymi – toy	133

Spis definicji

Alfabet	7
Bezpośrednia wyprowadzalność	9
Częściowy porządek	7
Długość słowa.....	7
Dobry porządek	7
Drzewo wyprowadzenia/derywacji/rozkładu	10
Gramatyka bezkontekstowa	9
Gramatyka formalna	9
Gramatyka GCS.....	10
Gramatyka kombinatoryczna	9
Gramatyka kontekstowa.....	9
Gramatyka lewostronne liniowa	9
Gramatyka prawostronnie liniowa	9
Gramatyka regularna.....	9
Gramatyka sGCS	11
Język generowany przez gramatykę.....	9
Język nad dowolnym alfabetem	8
Język, język nad alfabetem	7
klasyfikator GCS	10
klasyfikator sGCS	10
Kres górnny	7
Ograniczenie górne	7
Osiągalność.....	9
Pelna reprezentacja gramatyki	12
Porządek liniowy	7
Porządek regularny	7
Porządek zupełny	7
Postać Chomsky'ego, PNC, CNF	9
Poztywna reprezentacja gramatyki.....	12
Produktywność	9
Przechodniość	6
Przystosowanie	12
Relacja	6
Równoważność CNF	9
Równoważność gramatyk	9
Słaba antysymetryczność	6
Słowo puste.....	7
Słowo, słowo nad alfabetem	7
Stochastyczna gramatyka bezkontekstowa.....	10
Symbol.....	7
Symetryczność	6
Wyprowadzalność	9
Zbiór częściowo uporządkowany	7
Zbiór skierowany	7
Zdanie	12
Znormalizowane prawdopodobieństwa	11
Zwrotność	6

Załączniki

Wraz z niniejszą pracą dostarczono płytę DVD z następującą zawartością:

- DOC – katalog zawierający elektroniczną wersję tej pracy magisterskiej;
- BADANIA – folder zawierający arkusz kalkulacyjny w formacie xlsx, zawierający kompletny zestaw danych zebranych podczas pomiarów oraz inne artefakty wygenerowane przez algorytm;
- PROJEKT – katalog zawierający kod aplikacji. Ponieważ Python jest językiem interpretowanym możliwe jest natychmiastowe uruchomienie aplikacji (po uprzednim zainstalowaniu wymaganych bibliotek);
- ZBIORY – w tym folderze znajdują się wszystkie zbiory uczące oraz testowe, które wykorzystywano podczas przeprowadzania badań.