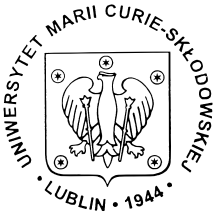


UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ
W LUBLINIE
WYDZIAŁ MATEMATYKI FIZYKI I INFORMATYKI

Michał Pańczyk

Wykorzystanie złożoności Kołmogorowa do wyznaczania związków kontekstowych poprzez analizę efektów pracy wyszukiwarek internetowych

Using Kolmogorov complexity to discover context associations
by analysis of web search engines' search effects

Praca magisterska wykonana
pod kierunkiem dra Jerzego Mycki

Lublin 2008

Składam serdeczne podziękowania
Panu dr. Jerzemu Mycce
za poświęcony czas, cenne wskazówki
i wszelką pomoc okazaną
podczas pisania niniejszej pracy.

Spis treści

Wprowadzenie

W niniejszej pracy zajmiemy się zagadnieniem bliskości kontekstowej między słowami z języka naturalnego. Idea przedstawiona przez nas została zaproponowana w [5]. Wykorzystuje ona teorię złożoności Kołmogorowa [6] oraz częstość występowania słów na stronach internetowych zindeksowanych przez popularne wyszukiwarki.

Otrzymaną miarę odległości kontekstowej zastosujemy w zaprojektowanym przez nas algorytmie, który mając podaną na wejściu listę bliskich sobie słów oraz jedno słowo spoza ich wspólnego kontekstu, wskazuje właśnie to niepasujące słowo. Program implementujący nasz algorytm napisaliśmy w języku Python.

Rozdział pierwszy przedstawia używaną notację oraz podstawy teorii złożoności Kołmogorowa. Prezentujemy w nim twierdzenia pozwalające na wyrobienie pewnej intuicji oraz mające wpływ na zastosowanie teorii, w szczególności prezentujemy własny dowód nieobliczalności C (tw. 1.2.11). Rozdział ten został opracowany na podstawie książki [6], a także [8] oraz pracy [7].

Rozdział drugi przedstawia zagadnienie bliskości informacyjnej i na jej podstawie wyprowadza odległość kontekstową głównie w oparciu o [1] oraz [5]. Omówiona została również przydatność różnych wyszukiwarek internetowych dla naszego zagadnienia.

Rozdział trzeci prezentuje zaproponowany przez nas algorytm odrzucania słowa niepasującego do kontekstu określonego przez listę słów. Przedstawiono implementację w języku Python, krótkie wprowadzenie do niego oraz przykłady działania programu.

Rozdział 1

Podstawy teorii złożoności Kołmogorowa

1.1. Notacja

W celu sprawnego posługiwania się terminami używanymi w dalszej części pracy, musimy zdefiniować kilka pojęć wstępnych.

1.1.1. Słowa i kody

Definicja 1.1.1. *Alfabet* jest niepustym, skończonym zbiorem niepodzielnych symboli.

Tak skonstruowana definicja gwarantuje nam, że słowa będą jednoznacznie rozkładalne na symbole. Przykładem wadliwie zbudowanego alfabetu jest $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{ab}\}$, gdyż słowo $\boldsymbol{ab}$ może być zbudowane z pojedynczego symbolu $\boldsymbol{ab}$ lub z dwóch symboli $\boldsymbol{a}$ i $\boldsymbol{b}$. Najczęściej będziemy oznaczać alfabet przez Σ.

Definicja 1.1.2. *Słowem (napisem)* x o długości n — co zapisujemy $l(x) = n$ — nazywamy funkcję $x : \mathbb{Z}_n^+ \longrightarrow \Sigma$, gdzie $\mathbb{Z}_n^+ = \{1, 2, \ldots, n\}$.

Zbiór wszystkich słów długości n oznaczamy przez Σ^n, przy czym Σ^0 zawiera jeden element, oznaczany przez ϵ, zwany słowem pustym.

Słowa zapisujemy wymieniając po kolei składowe symbole: $\Sigma^n \ni x = x(1)x(2)\ldots x(n)$ lub — stosując krótszy zapis — $x = x_1x_2\ldots x_n$, zatem $x(i) = x_i \in \Sigma$ oznacza i-ty symbol słowa x.

Definicja 1.1.3. *Domknięcie Kleene'ego* zbioru Σ definiujemy jako

$$\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n.$$

Mówimy, że Σ^* jest zbiorem wszystkich skończonych słów nad alfabetem Σ. Zauważmy, że do zbioru Σ^* należą słowa dowolnej, ale skończonej długości. Ponadto dla każdego słowa $x \in \Sigma^*$

$$\exists n \in \mathbb{N} : \ x \in \Sigma^n.$$

Przykład 1.1.4. Niech $\Sigma = \{\boldsymbol{a}, \boldsymbol{b}\}$, wtedy $\Sigma^* = \{\epsilon, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{aa}, \boldsymbol{ab}, \boldsymbol{ba}, \boldsymbol{bb}, \boldsymbol{aaa}, \boldsymbol{aab}, \ldots\}$.

Definicja 1.1.5. *Podsłowem* $x_{n..m}, 1 \leqslant n \leqslant m \leqslant l(x)$ słowa x nazywamy słowo długości $m - n + 1$, postaci $x_n x_{n+1} \ldots x_{m-1} x_m$.

Przykład 1.1.6. Niech $x = \boldsymbol{ab}$, wtedy $x_{1..1} = \boldsymbol{a}$, $x_{1..2} = \boldsymbol{ab}$, $x_{2..2} = \boldsymbol{b}$.

Definicja 1.1.7. *Konkatenacją* słów $x : \mathbb{Z}_n^+ \longrightarrow \Sigma$ i $y : \mathbb{Z}_m^+ \longrightarrow \Sigma$ jest słowo $xy : \mathbb{Z}_{m+n}^+ \longrightarrow \Sigma$, takie że $(xy)_{1..n} = x$ i $(xy)_{n+1..n+m} = y$.

Przykład 1.1.8. Niech $x = \boldsymbol{bba}$ i $y = \boldsymbol{ab}$, wtedy $xy = \boldsymbol{bbaab}$ i $yx = \boldsymbol{abbba}$.

Zauważmy, że konkatenacja jest operacją zamkniętą w zbiorze Σ^*. Ponadto jest łączna: $(xy)z = x(yz)$, a ϵ jest jej elementem neutralnym: $x\epsilon = \epsilon x = x$.

Definicja 1.1.9. *Słowem nieskończonym* x z alfabetu Σ będziemy nazywać dowolną funkcję $x : \mathbb{N}^+ \longrightarrow \Sigma$.

Zbiór wszystkich nieskończonych słów nad alfabetem Σ oznaczamy przez Σ^ω, a sumę zbiorów Σ^* i Σ^ω przez Σ^∞. Oczywiście długość słowa $x \in \Sigma^\omega$ jest nieskończona: $l(x) = \infty$.

W prosty sposób możemy rozszerzyć definicję podsłowa także na słowa nieskończone.

Definicja 1.1.10. *Podsłowem* $x_{n..m}$ $(1 \leqslant n < \infty;\ n \leqslant m \leqslant \infty)$ słowa nieskończonego $x \in \Sigma^\omega$ jest $x_n x_{n+1} \ldots x_{m-1} x_m \in \Sigma^*$ dla $m < \infty$ lub $x_n x_{n+1} \ldots \in \Sigma^\omega$ dla $m = \infty$.

Przykład 1.1.11. Niech $x = \boldsymbol{aabaaba}\ldots \in \Sigma^\omega$, wtedy $x_{4..\infty} = \boldsymbol{aabaaba}\ldots$, $x_{2..4} = \boldsymbol{aba} \in \Sigma^*$.

Podobnie rozszerzymy konkatenację — z zastrzeżeniem, że pierwsze słowo musi być skończone.

Definicja 1.1.12. *Konkatenacją* słów $x : \mathbb{Z}_n^+ \longrightarrow \Sigma$ i $y : \mathbb{Z}^+ \longrightarrow \Sigma$ jest słowo $xy : \mathbb{Z}^+ \longrightarrow \Sigma$, takie że $(xy)_{1..n} = x$ i $(xy)_{n+1..\infty} = y$.

Przykład 1.1.13. Niech $x = \boldsymbol{a} \in \{\boldsymbol{a}, \boldsymbol{b}\}^*$, $y = \boldsymbol{ababab}\ldots \in \{\boldsymbol{a}, \boldsymbol{b}\}^\omega$, wtedy $xy = \boldsymbol{aababa}\ldots \in \{\boldsymbol{a}, \boldsymbol{b}\}^\omega$.

W dalszej części pracy — jeśli nie określimy inaczej — będziemy zajmować się słowami nad alfabetem $B = \{\mathbf{0}, \mathbf{1}\}$ i nazywać je słowami binarnymi. Nie powoduje to zmniejszenia ogólności, ponieważ symbole z dowolnego alfabetu Σ można zakodować za pomocą słów binarnych o stałej długości równej $\lceil \log |\Sigma| \rceil$, gdzie log, jak wszędzie w dalszej części pracy, oznacza logarytm o podstawie 2.

Ponadto będziemy myśleć o liczbach naturalnych i elementach z B^* w sposób równoważny. W tym celu zdefiniujemy wzajemnie jednoznaczne odwzorowanie $n : B^* \longrightarrow \mathbb{N}$, takie że dla każdego $x \in B^*$

$$n(x) = \sum_{i=1}^{l(x)} (x_i + 1) 2^{l(x)-i}. \tag{1.1}$$

Poniższa tabela przedstawia przykładowe wartości funkcji n.

x	ϵ	**0**	**1**	**00**	**01**	**10**	**11**	**000**	**001**	**010**	**011**	**100**	**101**	...
$n(x)$	0	1	2	3	4	5	6	7	8	9	10	11	12	...

Tabela 1.1: Odwzorowanie $n : B^* \longrightarrow \mathbb{N}$.

Zwróćmy uwagę, że wiodące zera w słowach binarnych — w przeciwieństwie do klasycznego pozycyjnego zapisu binarnego liczb — mają znaczenie. Zatem $n : B^* \longrightarrow \mathbb{N}$ jest bijekcją, więc możemy zdefiniować funkcję odwrotną $n^{-1} : \mathbb{N} \longrightarrow B^*$. W ten sposób wprowadziliśmy izomorfizm pomiędzy B^* a $\mathbb{N}$, który pozwala nam pomijać zapis explicite funkcji n oraz n^{-1} — słowa binarne będą oznaczały (jeśli kontekst na to wskazuje) liczby naturalne, natomiast liczby naturalne będą zapisywane jako równoważne im słowa binarne.

Przykład 1.1.14. Niech $x = \mathbf{010}$, wtedy x reprezentuje liczbę 9. Natomiast $l(x) = 3 = \mathbf{00}$, ponadto $l(l(x)) = l(\mathbf{00}) = 2 = \mathbf{1}$.

W dalszej części zajmiemy się kodowaniem dwóch (lub iteracyjnie — wielu) słów w jedno słowo w taki sposób, by możliwe było jednoznaczne uzyskanie każdego słowa składowego z ich zakodowania. Nie możemy do tego użyć zwykłej

konkatenacji, ponieważ nie jest ona bijekcją. Na przykład słowo **001** mogło powstać zarówno przez złączenie **0** i **01** jak też **00** i **1**.

Definicja 1.1.15. Słowo y nazywamy *prefiksem właściwym* słowa x, jeśli istnieje $z \neq \epsilon$ takie, że $x = yz$.

Definicja 1.1.16. Zbiór $A \subset B^*$ nazywamy *bezprefiksowym*, jeśli dla każdej pary $x, y \in A, x \neq y$, ani x nie jest prefiksem właściwym y, ani y nie jest prefiksem właściwym x.

Wprowadzimy kodowanie bezprefiksowe, które będzie nam przydatne w dalszych częściach pracy. Niech a^n oznacza słowo zbudowane z n razy powtórzonego symbolu a, na przykład $\boldsymbol{a}^2 = \boldsymbol{aa}$, $\mathbf{1}^4 = \mathbf{1111}$. Oznaczmy przez $\overline{x}$ słowo postaci $\mathbf{1}^{l(x)}\mathbf{0}x$, którego długość wynosi $2l(x)+1$. Zauważmy, że zliczając liczbę wystąpień symbolu **1** na początku tak zakodowanego napisu, jesteśmy w stanie określić, w którym miejscu się on kończy. Dzięki temu konkatenacja $\overline{x}y$ jest jednoznacznie rozkładalna na słowa x oraz y.

Przykład 1.1.17. Niech $\overline{x}y = \mathbf{110011}$, wtedy $x = \mathbf{01}$ i $y = \mathbf{1}$.

Często będziemy oznaczać jednoznaczne kodowanie pary słów x i y w jedno jako $\langle x, y\rangle$. Kodowaniem takim może być wprowadzone przez nas powyżej $\langle x, y\rangle = \overline{x}y$. Niezależnie od wyboru kodowania, zawsze będziemy myśleć o parze $\langle x, y\rangle$ jako o takim zakodowaniu x i y, które gwarantuje nam możliwość jednoznacznego odkodowania.

Mając zdefiniowane zbiory bezprefiksowe, możemy podać twierdzenie związane z długością słów w takich zbiorach [6].

Twierdzenie 1.1.18. (nierówność Krafta). Niech l_1, l_2, ... będzie skończonym lub nieskończonym ciągiem liczb naturalnych. Istnieje zbiór bezprefiksowy o słowach binarnych długości l_1, l_2, ... wtedy i tylko wtedy, gdy

$$\sum_i 2^{-l_i} \leqslant 1. \tag{1.2}$$

Zauważmy, że nie każdy zbiór, spełniający tę nierówność, jest bezprefiksowy.

Przykład 1.1.19. Zbiór $\{\mathbf{0}, \mathbf{00}, \mathbf{11}\}$ spełnia nierówność Krafta, mimo że nie jest bezprefiksowy — **0** jest prefiksem **00**.

1.1.2. Funkcje rekurencyjne

Definiując złożoność Kołmogorowa, wielokrotnie będziemy odwoływać się do funkcji częściowo rekurencyjnych, zatem poniżej wprowadzimy ich definicję [7].

Definicja 1.1.20. *Funkcję zerowania* $Z : \mathbb{N} \longrightarrow \mathbb{N}$ definiujemy w ten sposób, że $Z(x) = 0$.

Definicja 1.1.21. *Funkcję następnika* $S : \mathbb{N} \longrightarrow \mathbb{N}$ definiujemy w ten sposób, że $S(x) = x + 1$.

Definicja 1.1.22. *Funkcję rzutowania (projekcji)* $I_n^i : \mathbb{N}^n \longrightarrow \mathbb{N}$ definiujemy w ten sposób, że $I_n^i(x_1, \ldots, x_n) = x_i$, $1 \leqslant i \leqslant n$.

Zauważmy, że funkcja $I = I_1^1$ jest funkcją tożsamościową.

Definicja 1.1.23. *Złożeniem funkcji* $h : \mathbb{N}^k \longrightarrow \mathbb{N}$ oraz $g_1, \ldots, g_k : \mathbb{N}^n \longrightarrow \mathbb{N}$ nazywamy funkcję $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ zdefiniowaną jako

$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n)).$$

Definicja 1.1.24. Mówimy, że funkcja $f : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$ jest zbudowana z funkcji $g : \mathbb{N}^n \longrightarrow \mathbb{N}$ oraz $h : \mathbb{N}^{n+2} \longrightarrow \mathbb{N}$ za pomocą *rekursji prostej*, jeśli jest zdefiniowana jako

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$$
$$f(x_1, \ldots, x_n, S(x_{n+1})) = h(f(x_1, \ldots, x_{n+1}), x_1, \ldots, x_{n+1}).$$

Definicja 1.1.25. *Funkcją pierwotnie rekurencyjną* nazywamy funkcję zerowania Z, następnika S, rzutowania I_n^i oraz każdą utworzoną za pomocą złożenia i rekursji prostej funkcji pierwotnie rekurencyjnych.

Przykład 1.1.26. Pokażemy, że funkcje poprzednika P oraz różnicy ograniczonej $\dot{-}$ są pierwotnie rekurencyjne.

$$\begin{cases} P(0) & = 0 \\ P(S(x)) & = I_2^2(0, x) \end{cases} \qquad \begin{cases} \dot{-}(x, 0) & = I_1^1(x) \\ \dot{-}(x, S(y)) & = P(I_3^1(\dot{-}(x, y), x, y)) \end{cases}$$

Klasę wszystkich funkcji pierwotnie rekurencyjnych oznaczamy przez $\mathcal{P}$. Funkcje z $\mathcal{P}$ są wszędzie zdefiniowane czyli całkowite na zbiorze $\mathbb{N}$. Klasa $\mathcal{P}$

zawiera przeliczalnie nieskończenie wiele funkcji, lecz mimo to nie zawiera wszystkich obliczalnych funkcji. W celu zdefiniowania zbioru wszystkich funkcji obliczalnych, wprowadzimy operację minimum.

Definicja 1.1.27. *Operację minimum (μ-rekursji)* $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ dla danej funkcji $h : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$ definiujemy jako

$$f(x_1, \ldots, x_n) = \begin{cases} y & \text{jeśli } h(x_1, \ldots, x_n, y) = 0 \\ & \text{oraz } \forall x < y : h(x_1, \ldots, x_n, x) \neq 0 \\ \text{niezdefiniowana} & \text{jeśli } \forall y \in \mathbb{N} : h(x_1, \ldots, x_n, y) \neq 0. \end{cases}$$

Operator minimum znajduje zatem pierwsze miejsce zerowe funkcji h względem jej ostatniego argumentu, przy ustalonych pozostałych. Jeśli takiego miejsca zerowego nie ma, wartość jest nieokreślona. Zapis $f(\bar{x}) = \mu_y[g(\bar{x}, y)]$ będzie oznaczać, że wartością $f(\bar{x})$ jest najmniejsze y spełniające warunek $g(\bar{x}, y)$.

Przykład 1.1.28. Przy użyciu operacji minimum zdefiniujemy funkcję pierwiastka całkowitoliczbowego $r(x) = \lceil \sqrt{x} \rceil$:

$$r(x) = \mu_y[(x \dot{-} y^2) = 0].$$

Definicja 1.1.29. *Funkcją częściowo rekurencyjną* nazywamy funkcję zerowania Z, następnika S, rzutowania I_n^i oraz każdą utworzoną za pomocą złożenia, rekursji prostej i operacji minimum funkcji częściowo rekurencyjnych.

Klasę wszystkich funkcji częściowo rekurencyjnych oznaczamy przez $\mathcal{PRF}$.

1.1.3. Maszyna Turinga

Zdefiniujemy teraz inny, równoważny funkcjom $\mathcal{PRF}$, model obliczalności [8].

Definicja 1.1.30. *Maszyną Turinga (MT)* nazywamy uporządkowaną czwórkę $M = (K, \Sigma, \delta, s)$, w której

- K jest skończonym zbiorem *stanów*, który zawiera — pośród innych — stany specjalne: „stop", „tak" oraz „nie",
- Σ jest skończonym zbiorem symboli (*alfabetem* maszyny M), który — oprócz zwykłych — zawiera dwa dodatkowe symbole: $\sqcup$ – *pusty* oraz $\triangleright$ – *końcowy*,

– δ jest *funkcją przejścia* odwzorowującą $K \times \Sigma$ w $K \times \Sigma \times \{\leftarrow, -, \rightarrow\}$,
– $s \in K$ jest *stanem początkowym*.

Maszyna Turinga działa na nieskończonej taśmie podzielonej na komórki, na których zapisywane są symbole z Σ. Wzdłuż taśmy porusza się głowica odczytująco-zapisująca. Na początku pracy maszyna znajduje się w stanie s, głowica wskazuje na pierwszy na taśmie symbol — zawsze $\triangleright$, za którym zapisane jest *słowo wejściowe* $x \in (\Sigma \setminus \{\sqcup, \triangleright\})^*$, a za nim nieskończony ciąg symboli $\sqcup$. Czas działania MT jest dyskretny, mierzony krokami. W każdym kroku maszyna zgodnie ze swoim stanem q i symbolem pod głowicą a dokonuje przejścia według funkcji δ w ten sposób, że jeśli $\delta(q, a) = (q', a', d)$, to ustawia stan q', zapisuje na taśmie symbol a' i ewentualnie przesuwa głowicę o jedną komórkę zgodnie z $d \in \{\leftarrow, -, \rightarrow\}$. Funkcja przejścia δ jest zdefiniowana dla wszystkich możliwych do osiągnięcia kombinacji stanów i symboli.

Maszyna kończy swoje działanie w momencie ustawienia swojego stanu na jeden ze zbioru {„stop", „tak", „nie"}. Gdy tym stanem będzie „stop", znaczy to, że maszyna dla danego słowa wejściowego x obliczyła słowo wyjściowe y, które znajduje się na taśmie bezpośrednio za symbolem $\triangleright$. Piszemy wtedy $M(x) = y$ i mówimy, że y jest wynikiem obliczeń maszyny M na słowie wejściowym x. Gdy stanem kończącym pracę będzie „tak" lub „nie", mówimy, że maszyna — odpowiednio — zaakceptowała lub nie zaakceptowała słowa wejściowego x.

Możliwa jest również sytuacja, gdy dla danego x maszyna nigdy się nie zatrzyma, wtedy piszemy $M(x) = \nearrow$.

Maszyny Turinga i funkcje częściowo rekurencyjne są równoważne w sensie obliczalności. To znaczy, że funkcja należy do $\mathcal{PRF}$ wtedy i tylko wtedy, gdy istnieje MT obliczająca tę funkcję. Ponadto hipoteza Churcha-Turinga głosi, że wszystko, co jest efektywnie obliczalne, można policzyć za pomocą maszyn Turinga lub funkcji częściowo rekurencyjnych, a także nie istnieje zagadnienie obliczane przez któryś z tych modeli, który nie byłby efektywnie obliczalny. Dlatego można mówić o funkcjach częściowo rekurencyjnych jako o *obliczalnych*.

Definicja 1.1.31. Niech będzie dana funkcja $f \in \mathcal{PRF}$. Mówimy, że maszyna M oblicza f, jeśli dla każdego słowa x zachodzi

$$M(x) = \begin{cases} f(x) & \text{jeśli } f(x) \text{ zdefiniowana} \\ \nearrow & \text{jeśli } f(x) \text{ niezdefiniowana.} \end{cases}$$

Zdefiniowana do tej pory MT ma przełożenie praktyczne na program komputerowy — wykonuje jedno, ustalone zadanie. Jednak komputery umożliwiają rozwiązywanie wielu różnych problemów algorytmicznych. Chcąc odzwierciedlić tę wszechstronność, zdefiniujemy uniwersalną maszynę Turinga.

Definicja 1.1.32. *Uniwersalną maszyną Turinga U* nazywamy MT, która symuluje działanie dowolnej MT.

Uniwersalna maszyna Turinga dostaje na swoim wejściu opis symulowanej maszyny M. Opis taki sprowadza się do wymienienia wszystkich reguł działania funkcji przejścia δ_M, tj. piątek postaci (q, a, q', a', d), oznaczających odpowiednio: stan i symbol zastany, nowy stan i symbol oraz kierunek przesunięcia głowicy, przy czym symbole taśmowe i stany symulowanej MT są opisywane za pomocą kolejnych liczb naturalnych (reprezentowanych na przykład binarnie). Opis jest tak skonstruowany, by U była w stanie określić jego koniec, za którym — na taśmie — znajduje się słowo wejściowe dla symulowanej maszyny. Piszemy $U(\langle T, x\rangle) = T(x) = y$, jeśli U symuluje działanie maszyny T na słowie wejściowym x i w obu przypadkach wynikiem jest słowo y.

Istnieje nieskończenie wiele uniwersalnych MT. Pisząc U, mamy na myśli jedną *ustaloną*, uniwersalną MT.

Jeśli ustawimy opisy (potraktowane literalnie jako słowa) wszystkich MT w porządku leksykograficznym, uzyskamy tzw. numerację Gödla maszyn Turinga. Otrzymamy w ten sposób ciąg $T_0, T_1, T_2, \ldots$, gdzie $T_0 = U$. Zapis $U(\langle n, x\rangle)$ będzie równoważny zapisowi $U(\langle T_n, x\rangle)$. Ponadto zgodnie z definicją 1.1.31, każdej maszynie T_i odpowiada funkcja częściowo rekurencyjna f_i.

1.2. Klasyczna złożoność Kołmogorowa

W 1948 roku Shannon stworzył podstawy probabilistycznej teorii informacji [11]. Swoją teorię nazwał *teorią komunikacji*, stąd też występuje w niej pojęcie źródła, które emituje komunikaty. Ilość informacji, jaką niesie ze sobą komunikat, była ściśle związana z charakterystyką źródła i nie była bezpośrednio uzależniona od struktury komunikatu. Toteż aparat matematyczny stworzony przez Shannona nie pozwalał na określenie informacji zawartej w pojedynczym obiekcie w oderwaniu od źródła — zbioru, do którego należał obiekt i rozkładu prawdopodobieństwa, z jakim emitowane są jego

elementy.

Między innymi dlatego w latach sześćdziesiątych XX wieku niezależnie od siebie Solomonoff, Kołmogorow i Chaitin stworzyli podstawy teorii, którą obecnie nazywamy złożonością algorytmiczną, czy też złożonością Kołmogorowa. Pozwala ona określić ilość informacji w pojedynczym obiekcie badając jego strukturę. W niniejszym oraz następnym podrozdziale opieramy się na pracy [6].

1.2.1. Definicja

Niech będzie dany zbiór obiektów S. Wprowadźmy przyporządkowanie $n : S \longrightarrow \mathbb{N}$ nadające każdemu obiektowi $x \in S$ unikalną liczbę naturalną n_x.

Ponadto niech będzie dana funkcja częściowa $f : \mathbb{N} \longrightarrow \mathbb{N}$. Jeśli nie napiszemy inaczej, p będzie oznaczało liczbę naturalną. Przez $l(p)$ oznaczymy długość zapisu binarnego liczby p, jak w przykładzie 1.1.14 (s. 3).

Definicja 1.2.1. *Złożonością* obiektu $x \in S$ względem funkcji f nazywamy

$$C_f(n_x) = \min\{l(p) : f(p) = n_x\}, \tag{1.3}$$

przy czym $C_f(n_x) = \infty$ jeśli nie istnieje takie p.

Intuicyjnie możemy myśleć, że p jest kodem programu dla komputera (interpretera) f, który po uruchomieniu — nie czekając na jakiekolwiek wejście — wypisze identyfikator n_x obiektu x. Interesuje nas długość najkrótszego takiego programu.

W dalszych rozważaniach będziemy utożsamiać obiekt x z jego identyfikatorem n_x. Dzięki temu możemy pisać $C_f(x)$ zamiast $C_f(n_x)$.

Definicja 1.2.2. Mówimy, że funkcja f *minoryzuje* funkcję g, jeśli dla każdego x zachodzi nierówność

$$C_f(x) \leqslant C_g(x) + c_{f,g},$$

gdzie $c_{f,g}$ jest stałą zależną jedynie od f i g.

Definicja 1.2.3. Dwie funkcje f i g są *równoważne*, jeśli f minoryzuje g oraz g minoryzuje f.

Na podstawie powyższej definicji możemy utworzyć klasy równoważności funkcji. Między tymi klasami w naturalny sposób pojawia się hierarchia

oparta na minoryzacji. Kołmogorow rozważał, czy w takim uporządkowaniu występuje element minimalny, tj. klasa funkcji, które minoryzują wszystkie inne (a z definicji 1.2.2 również same siebie).

Definicja 1.2.4. Niech $\mathcal{C}$ będzie podzbiorem funkcji częściowych nad zbiorem liczb naturalnych. Funkcja $f \in \mathcal{C}$ jest *addytywnie optymalna* dla $\mathcal{C}$, jeśli dla wszystkich $g \in \mathcal{C}$, f minoryzuje g.

Poniżej przekonamy się, że istnieją zbiory funkcji, które nie mają elementu optymalnego.

Przykład 1.2.5. Rozważmy zbiór *wszystkich* funkcji częściowych nad $\mathbb{N}$. Przypuśćmy, że zbiór ten zawiera funkcję f i jest ona addytywnie optymalna. Niech będzie dany nieskończony rosnący ciąg $p_1 < p_2 < \ldots$, taki że $f(p_i) = x_i$ dla $i \geqslant 1$ i jednocześnie p_i jest możliwie najmniejsze. Wybierzmy z p_1, p_2, $\ldots$ podciąg q_1, q_2, $\ldots$ spełniający warunek $\log q_i > 2 \log p_i$. Mając dane q_i zdefiniujmy funkcję g w ten sposób, że $g(p_i) = f(q_i)$ dla wszystkich $i \geqslant 1$.

Wtedy dla nieskończenie wielu x zachodzi nierówność $C_f(x) \geqslant 2C_g(x)$, co jest sprzeczne z założeniem, że f jest optymalna.

Powstanie teorii złożoności Kołmogorowa nie byłoby możliwe, gdyby nie istniał zbiór zawierający funkcję optymalną.

Twierdzenie 1.2.6. Zbiór funkcji częściowo rekurencyjnych $\mathcal{PRF}$ zawiera element addytywnie optymalny.

Dowód. Niech ϕ_0 będzie funkcją częściowo rekurencyjną obliczaną przez uniwersalną maszynę Turinga U, która przyjmuje na wejściu dane w postaci $\langle n, p\rangle = \overline{n}p$. Maszyna U może w jednoznaczny sposób odkodować z wejścia liczbę n oraz p. Ponadto z faktu, że funkcji częściowo rekurencyjnych jest *przeliczalnie* wiele, wynika, że możemy je ponumerować. Więc niech n określa maszynę T_n obliczającą funkcję ϕ_n i $T_0 = U$. Wtedy $\phi_0(\langle n, p\rangle) = \phi_n(p)$.

Funkcja ϕ_0 jest elementem optymalnym dla $\mathcal{PRF}$, ponieważ dla każdego n i x zachodzi

$$C_{\phi_0}(x) \leqslant C_{\phi_n}(x) + c_{\phi_n},$$

gdzie c_{ϕ_n} nie zależy od x i jest równe $2l(n) + 1$. □

Definicja 1.2.7. Niech x, y, p będą liczbami naturalnymi oraz ϕ funkcją częściowo rekurencyjną taką, że $\phi(\langle y, p\rangle) = x$. *Złożoność* C_ϕ liczby x

względem y definiujemy jako

$$C_\phi(x|y) = \min\{l(p) : \phi(\langle y, p\rangle) = x\}, \tag{1.4}$$

przy czym $C_\phi(x|y) = \infty$ jeśli nie istnieje takie p.

Podobnie jak w przypadku definicji 1.2.1, możemy myśleć o długości najkrótszego programu p, który działając na komputerze ϕ, wypisuje liczbę x. Jednak w tym przypadku program oczekuje na daną wejściową y, na podstawie której oblicza x.

Mając zdefiniowaną złożoność względną, możemy udowodnić jej niezmienniczość.

Twierdzenie 1.2.8. (o niezmienniczości). Istnieje addytywnie optymalna funkcja częściowo rekurencyjna dla $\mathcal{PRF}$, obliczająca x, mając dane y. Inaczej ujmując, dla każdego $\phi \in \mathcal{PRF}$ istnieje $\phi_0 \in \mathcal{PRF}$, takie że

$$C_{\phi_0}(x|y) \leqslant C_\phi(x|y) + c_\phi, \tag{1.5}$$

gdzie c_ϕ zależy jedynie od ϕ.

Dowód. Niech ϕ_0 będzie funkcją obliczaną przez uniwersalną maszynę Turinga U, taką że U, dostawszy na wejściu $\langle n, \langle y, p\rangle\rangle$, symuluje pracę maszyny T_n z wejściem $\langle y, p\rangle$. Zatem jeśli T_n oblicza $\phi_n \in \mathcal{PRF}$, mamy równość $\phi_0(\langle n, \langle y, p\rangle\rangle) = \phi_n(\langle y, p\rangle)$ i dla wszystkich n zachodzi

$$C_{\phi_0}(x|y) \leqslant C_{\phi_n}(x|y) + c_{\phi_n},$$

gdzie $c_{\phi_n} = 2l(n) + 1$. □

Powyższe twierdzenie ma bardzo duże znaczenie dla całej teorii złożoności Kołmogorowa. Zauważmy, że na podstawie niego nie możemy powiedzieć, że funkcja optymalna zawsze daje nam najkrótszy opis danej liczby. Jednak dla dowolnej funkcji $\phi \in \mathcal{PRF}$ i dowolnego x, C_{ϕ_0} zawsze będzie ograniczać C_ϕ od dołu z dokładnością do stałej zależnej tylko od ϕ.

Jeszcze mocniejszy warunek możemy nałożyć na dwie różne funkcje optymalne ψ, ψ'. Z definicji są one równoważne, więc istnieje $c_{\psi,\psi'}$, takie że

$$\forall x : |C_\psi(x) - C_{\psi'}(x)| \leqslant c_{\psi,\psi'}.$$

Definicja 1.2.9. Ustalmy addytywnie optymalną funkcję $\phi_0 \in \mathcal{PRF}$ zwaną dalej *funkcją odniesienia. Względną złożoność Kołmogorowa* C liczby x względem y definiujemy jako

$$C(x|y) = C_{\phi_0}(x|y). \tag{1.6}$$

Ustalonej funkcji odniesienia ϕ_0 odpowiada ustalona uniwersalna maszyna Turinga U, zwana maszyną odniesienia dla C.

Definicja 1.2.10. *Bezwzględną złożoność Kołmogorowa* liczby x definiujemy jako $C(x) = C(x|\epsilon)$.

1.2.2. Własności

Jedną z najważniejszych własności złożoności Kołmogorowa jest jej nieobliczalność.

Twierdzenie 1.2.11. (o nieobliczalności). Funkcja C nie należy do zbioru funkcji częściowo rekurencyjnych $\mathcal{PRF}$.

Dowód. Przeprowadzimy dowód nie wprost. Załóżmy zatem, że $C \in \mathcal{PRF}$. Zdefiniujmy funkcję $f(x) = \mu_y[x < C(y)]$. Weźmy dowolne x i niech $f(x) = y$. Z definicji f mamy $x < C(y)$, jednocześnie z twierdzenia 1.2.8 mamy $C(y) < C_f(y) + c$ i z definicji C: $C_f(y) = l(x)$. Wynika z tego nierówność

$$x < C(y) \leqslant C_f(y) + c = l(x) + c,$$

która dla odpowiednio dużych wartości x prowadzi do sprzeczności. □

Jest to ważne twierdzenie, ponieważ wyklucza możliwość *bezpośredniego* użycia C w zastosowaniach praktycznych. Jednak dobrą wiadomością jest to, że możemy nałożyć na funkcję złożoności pewne ograniczenia. Mówią o tym trzy następne twierdzenia.

Twierdzenie 1.2.12. Istnieje takie c, że dla wszystkich x spełniona jest nierówność

$$C(x) \leqslant l(x) + c.$$

Dowód. Weźmy funkcję tożsamościową I. Wtedy $C_I(x) = l(x)$ i na mocy twierdzenia 1.2.8 powyższa nierówność jest spełniona. □

Przekładając dowód na język praktyki programistycznej — jeśli nie znajdziemy krótszego sposobu wypisania słowa x, w ostateczności możemy napisać program, który będzie zawierał x w swoim kodzie. Wtedy wartości $l(x)$ odpowiada literalne wypisanie x w kodzie programu, a stałej c odpowiada długość reszty kodu programu.

Podobnie zachodzi twierdzenie dla złożoności względnej.

Twierdzenie 1.2.13. Istnieje takie c, że dla wszystkich x oraz y spełniona jest nierówność

$$C(x|y) \leqslant C(x) + c.$$

Dowód opiera się na spostrzeżeniu, że informację zawartą w y możemy w najgorszym przypadku zignorować i wytworzyć x bez udziału y, dzięki czemu mamy prawą stronę nierówności.

Następne twierdzenie pokaże nam, że poprzednie dwa dają nam dobre ograniczenia C.

Twierdzenie 1.2.14. (o niekompresowalności). Niech c będzie dodatnią liczbą naturalną. Dla każdego y, każdy zbiór A o mocy m zawiera co najmniej $m(1 - 2^{-c}) + 1$ elementów x, takich że $C(x|y) \geqslant \log m - c$.

Dowód. Moc zbioru $\{p : l(p) < \log m - c\}$, czyli programów o długości mniejszej niż $\log m - c$, jest

$$\sum_{i=0}^{\log m - c - 1} 2^i = 2^{\log m - c} - 1 = m2^{-c} - 1.$$

Zatem jest przynajmniej $m - m2^{-c} + 1$ elementów w A, które nie mają generującego ich programu o długości mniejszej niż $\log m - c$. □

Wnioskiem z tego twierdzenia jest fakt, że z powodu małej liczby krótkich programów, jest niewiele obiektów o małej złożoności.

Okazuje się, że w praktycznych zastosowaniach, najprostszym sposobem oszacowania złożoności danego napisu jest poddanie go kompresji przy użyciu ogólnie dostępnych algorytmów, takich jak GZIP, BZIP2 czy też PPM, a następnie sprawdzenie, jaką długość ma napis wynikowy. Algorytmy te wykorzystują regularności w napisie wejściowym. Oczywiście, jak wynika z tw. 1.2.11, nie istnieje program, który znajduje wszystkie możliwe regularności w napisie, stąd każdy istniejący algorytm kompresji jest górnym ograniczeniem złożoności Kołmogorowa.

Poniżej pokażemy, jak ma się złożoność dwóch obiektów liczona razem do złożoności pojedynczych obiektów.

Definicja 1.2.15. Złożoność pary obiektów x i y definiujemy jako $C(x, y) = C(\langle x, y\rangle)$.

Lemat 1.2.16. Dla dowolnych x i y zachodzi nierówność

$$C(x, y) \leqslant C(x) + C(y) + 2\log(\min(C(x), C(y))) + c.$$

Dowód. Niech $T(p) = x$, $T(q) = y$ oraz p i q są możliwie najmniejsze. Wtedy istnieje T', taka że $T'(\overline{l(p)}pq) = \langle x, y\rangle$, podobnie $T'(\overline{l(q)}qp) = \langle y, x\rangle$. Zatem wybierając krótszy zapis z dwóch ($\overline{l(q)}qp$ lub $\overline{l(p)}pq$), otrzymamy x oraz y wraz z możliwością ich oddzielenia, więc mamy spełnioną nierówność. □

W powyższej nierówności nie możemy pozbyć się składnika logarytmicznego, zatem klasyczna złożoność Kołmogorowa nie jest *subaddytywna*.

1.3. Prefiksowa złożoność Kołmogorowa

Rozwój klasycznej teorii złożoności Kołmogorowa przyniósł wiele ciekawych wniosków i zastosowań. Mimo to nie spełnia ona wszystkich potrzebnych nam warunków, abyśmy mogli na niej oprzeć dalszą teorię. Dlatego zmodyfikujemy ją w drobnym szczególe, co jednak będzie miało znaczące skutki.

1.3.1. Definicja

Definicja 1.3.1. Funkcję $\phi \in \mathcal{PRF}$ nazywamy *prefiksową funkcją częściowo rekurencyjną*, jeśli dla dowolnych p i q ($p \neq q$) należących do jej dziedziny, binarna reprezentacja p nie jest prefiksem binarnej reprezentacji q.

Zbiór wszystkich prefiksowych funkcji częściowo rekurencyjnych będziemy oznaczać przez $\mathcal{PPRF}$. Analogią programistyczną jest zbiór wszystkich języków programowania, których składnia nie dopuszcza tego, że poprawny program jest prefiksem innego.

Tak jak w definicji 1.1.31 stwierdziliśmy, że funkcje częściowo rekurencyjne są obliczane przez MT, tak prefiksowe funkcje częściowo rekurencyjne są obliczane przez prefiksowe MT. Analogicznie istnieje też uniwersalna prefiksowa maszyna Turinga. Dzięki temu możemy sformułować twierdzenie o niezmienniczości dla złożoności prefiksowej.

Twierdzenie 1.3.2. (o niezmienniczości K). Istnieje addytywnie optymalna prefiksowa funkcja częściowo rekurencyjna $\psi_0 \in \mathcal{PPRF}$, taka że dla dowolnej funkcji $\psi \in \mathcal{PPRF}$ istnieje stała c_ψ spełniająca nierówność

$$C_{\psi_0}(x|y) \leqslant C_\psi(x|y) + c_\psi \tag{1.7}$$

dla dowolnych x i y.

Twierdzenie to pozwala nam zdefiniować prefiksową złożoność Kołmogorowa.

Definicja 1.3.3. Ustalmy optymalną funkcję $\psi_0 \in \mathcal{PPRF}$. *Prefiksową złożonością Kołmogorowa* x względem y nazywamy $K(x|y) = C_{\psi_0}(x|y)$. *Bezwzględną prefiksową złożoność Kołmogorowa* definiujemy jako $K(x) = K(x|\epsilon)$.

1.3.2. Własności

Złożoność prefiksowa, podobnie jak klasyczna, jest nieobliczalna. Dowód jest analogiczny jak w tw. 1.2.11 (s. 12), z dodatkowym warunkiem: $f \in \mathcal{PPRF}$.

Poniżej przedstawimy, jak mają się do siebie wartości złożoności klasycznej i prefiksowej.

Lemat 1.3.4. Dla wszystkich x oraz y, z dokładnością do stałych składników, zachodzi nierówność

$$C(x|y) \leqslant K(x|y) \leqslant C(x|y) + 2\log C(x|y).$$

Dowód. Pierwsza część nierówności wynika z faktu, że zbiór $\mathcal{PPRF}$ jest podzbiorem $\mathcal{PRF}$.

Dowód drugiej części opiera się na idei wyrażenia złożoności prefiksowej za pomocą klasycznej. Załóżmy, że p jest minimalnym programem generującym x na podstawie y na maszynie odniesienia dla C, czyli $l(p) = C(x|y)$. Aby otrzymać złożoność prefiksową, musimy na podstawie p utworzyć element zbioru bezprefiksowego. Kodem takim jest $\overline{l(p)}p$, a jego długość wynosi $l(p) + 2\log l(p) + 1$, skąd wynika $K(x|y) \leqslant C(x|y) + 2\log C(x|y) + O(1)$. □

Dla K nie zachodzi twierdzenie 1.2.12. Jednakże możemy podać ograniczenie słabsze o składnik logarytmiczny.

Twierdzenie 1.3.5. Dla każdego x zachodzi nierówność

$$K(x) \leqslant l(x) + 2\log l(x) + c.$$

Dowód. Wystarczy zakodować x jako $\overline{l(x)}x$ i podać tak skonstruowany napis na wejściu prefiksowej MT obliczającej funkcję tożsamościową. □

Pokażemy również, że w przeciwieństwie do złożoności klasycznej, złożoność prefiksowa jest subaddytywna. Podobnie, jak w definicji 1.2.15, zdefiniujmy $K(x, y)$ jako $K(\langle x, y\rangle)$.

Lemat 1.3.6. Dla każdego x i y zachodzi nierówność

$$K(x, y) \leqslant K(x) + K(y) + c,$$

gdzie c jest stałą niezależną od x i y.

Dowód. Niech U będzie maszyną odniesienia dla K. Ponadto $U(p_x) = x$, $l(p_x) = K(x)$. Podobnie $U(p_y) = y$, $l(p_y) = K(y)$. Weźmy wtedy maszynę prefiksową T, która mając dane na taśmie $p_x p_y$, oblicza kolejno x i y, a następnie ich kodowanie w $\langle x, y\rangle$. Zatem $C_T(\langle x, y\rangle) = l(p_x) + l(p_y) = K(x) + K(y)$. Zastosowanie twierdzenia o niezmienniczości (1.3.2) kończy dowód. □

Otrzymanie takiego wyniku jest możliwe dlatego, że program p_x jest prefiksowy, czyli *samograniczający się*, i maszyna T potrafi określić koniec programu generującego x a tym samym rozgraniczyć p_x od p_y. Dzięki temu nie mamy składnika logarytmicznego po prawej stronie nierówności, jak miało to miejsce w przypadku złożoności klasycznej.

Z faktu, że $\{K(x) : x \in \mathbb{N}\}$ jest zbiorem długości elementów zbioru bezprefiksowego, wynika, że spełniona jest nierówność Krafta w postaci

$$\sum_x 2^{-K(x)} \leqslant 1. \tag{1.8}$$

Rozdział 2

Odległość informacyjna

W niniejszym rozdziale wykorzystamy wprowadzoną wcześniej teorię do wyjaśnienia pojęcia i wyprowadzenia wzorów na odległość informacyjną pomiędzy dwoma obiektami.

2.1. Intuicja zagadnienia

Złożoność Kołmogorowa pozwala nam określać ilość informacji w napisach lub też w liczbach. Słowa i liczby mogą reprezentować dowolną informację możliwą do zwerbalizowania. Wykorzystamy więc ideę długości minimalnych programów generujących napisy do określania *odległości informacyjnej* pomiędzy dwoma słowami. W ogólnym przypadku informacja zawarta w napisie będzie traktowana literalnie. Na przykład satelitarna fotografia cyfrowa terytorium Polski to informacja sama w sobie, w odróżnieniu od słowa *Polska*, za którym kryje się wiele skojarzeń i które występuje w różnych kontekstach w otoczeniu innych słów — mówimy, że ma znaczenie — *semantykę*.

Będziemy chcieli otrzymać sposób na określenie odległości znaczeniowej między dwoma słowami. Weźmy za przykład nazwy trzech krajów: *Austria*, *Australia* i *Nowa Zelandia*. Widzimy, że dwie pierwsze nazwy różnią się niewiele — wystarczy dodać do słowa *Austria* cząstkę *al* przed dwiema ostatnimi literami, by otrzymać *Australia*. Jednak gdy zwrócimy uwagę na znaczenie tych słów, intuicja podpowie nam, że więcej wspólnego mają ze sobą Australia i Nowa Zelandia.

Jak widać, nie chodzi nam o zwykłe badanie podobieństwa napisów, lecz o podobieństwo znaczeniowe. Na początku podamy sposób określania podobieństwa literalnego między słowami, a następnie, poprzez pewną

analogię, podobieństwa znaczeniowego. Niezbędne w tym celu będzie wykorzystanie bazy wiedzy, która odzwierciedla związki semantyczne między wyrazami.

2.2. Znormalizowana odległość informacyjna

Aby zdefiniować odległość informacyjną rozważmy następujące zagadnienie. Niech dane będą napisy x oraz y. Jaka jest długość najkrótszego programu (w ustalonym uniwersalnym języku programowania), który oblicza x mając dane y oraz oblicza y mając dane x? Stosując notację z pierwszego rozdziału, podamy definicję przy użyciu funkcji rekurencyjnych.

Definicja 2.2.1. *Odległością informacyjną* liczb x oraz y nazywamy

$$E(x, y) = \min\{l(p) : \psi_0(p, y) = x \wedge \psi_0(p, x) = y\},$$

gdzie ψ_0 jest uniwersalną prefiksową funkcją częściowo rekurencyjną.

Program p wykorzystuje wszelkie podobieństwa między słowami x i y do ich wzajemnego odtwarzania. W [1] pokazano, że z dokładnością do składnika logarytmicznego zachodzi równość

$$E(x, y) = K(x, y) - \min\{K(x), K(y)\}. \tag{2.1}$$

Biorąc przykładowe słowa z poprzedniego podrozdziału: *Austria* i *Australia*, widzimy, że odległość między nimi jest niewielka.

Rozważmy, jak zachowuje się odległość informacyjna w zależności od wielkości badanych słów. Weźmy dwa małe słowa, które są w dużej odległości od siebie — znacznie się różnią oraz dwa słowa, które dzieli taka sama odległość, jak wcześniejsze, ale są wielokrotnie większe. W tym przypadku, biorąc pod uwagę ich rozmiar, powiemy, że są one stosunkowo podobne do siebie. Widzimy zatem, że odległość informacyjna nie jest odpowiednia do wyrażania podobieństwa między słowami względem ich rozmiaru. Mówiąc o wielkości, mamy na myśli złożoność prefiksową a nie długość słów.

Rozwiązaniem jest normalizacja odległości informacyjnej.

Definicja 2.2.2. *Znormalizowaną odległością informacyjną* (ang. *normalized information distance*) nazywamy

$$\mathrm{NID}(x,y) = \frac{K(x,y) - \min\{K(x), K(y)\}}{\max\{K(x), K(y)\}}. \tag{2.2}$$

Ponieważ odległość NID definiujemy za pomocą złożoności Kołmogorowa, jest ona nieobliczalna. Jednak w rozdziale pierwszym wspomnieliśmy, że złożoność algorytmiczną możemy aproksymować używając powszechnych algorytmów kompresji bezstratnej. Metodę tę możemy zastosować również i tutaj. W ten sposób otrzymamy praktyczną metodę określania podobieństwa napisów.

Definicja 2.2.3. Oznaczmy przez $\mathrm{comp}(x)$ długość skompresowanego ustaloną metodą napisu x. *Znormalizowaną odległością kompresyjną* nazywamy

$$\mathrm{NCD}(x,y) = \frac{\mathrm{comp}(xy) - \min\{\mathrm{comp}(x), \mathrm{comp}(y)\}}{\max\{\mathrm{comp}(x), \mathrm{comp}(y)\}}. \tag{2.3}$$

Widać, że wzór ten powstał przez proste zastąpienie K przez comp w równaniu (2.2), przy czym wykorzystaliśmy fakt, że $K(\langle x,y\rangle) = K(xy)$ z dokładnością do składnika logarytmicznego. Metodę z powodzeniem zastosowano w [4] m.in. do grupowania sekwencji DNA, języków europejskich i muzyki ze względu na jej styl.

Przykład 2.2.4. Użyjemy narzędzia BZIP2 jako kompresora. Biorąc wcześniejsze przykłady, otrzymujemy

- $\mathrm{NCD}(\textit{Australia}, \textit{Nowa Zelandia}) = 0{,}2727$,
- $\mathrm{NCD}(\textit{Australia}, \textit{Austria}) = 0{,}0851$.

Podobnej metody użyto w [3] do wykrywania plagiatów w komputerowych programach zaliczeniowych pisanych przez studentów, przy czym zastosowano zmodyfikowany kompresor.

2.3. Znormalizowana odległość kontekstowa

2.3.1. Definicja

Teorie informacji Shannona oraz złożoności Kołmogorowa pozwalają określić ilość informacji w obiekcie. Opierają się na różnych podstawach,

lecz w pewnych obszarach dochodzą do jednakowych wyników, w innych dopełniając się wzajemnie. Odwołamy się teraz do probabilistycznej teorii informacji Shannona [11], aby z odmiennego, niż prezentowany dotychczas, punktu widzenia określić ilość informacji związanej z obiektem.

Definicja 2.3.1. Niech będzie dany zbiór $S = \{x_1, x_2, \ldots, x_n\}$ i miara prawdopodobieństwa P na zbiorze S taka, że $\sum_{i=1}^{n} P(x_i) = 1$. Mówimy, że wybór elementu x_i ze zbioru S niesie ze sobą $h(x_i) = -\log P(x_i)$ *bitów informacji*.

W powyższej definicji używamy logarytmu dwójkowego, stąd za jednostkę przyjmujemy bit (ang. *binary digit*).

Mamy zatem inną — różną od złożoności Kołmogorowa — definicję miary informacji. Zakładamy, że obie definicje dają ilościowo jednakowe wyniki. Możemy zatem powiązać rozkład prawdopodobieństwa występowania słów z ich złożonością algorytmiczną przez następujące twierdzenie [2].

Twierdzenie 2.3.2. Niech będą dane S i P jak w definicji 2.3.1. Dla każdego $x \in S$ zachodzi równość

$$K(x) = -\log P(x) + c, \tag{2.4}$$

gdzie c nie zależy od x.

Twierdzenie to pozwala nam zastąpić nieobliczalną złożoność Kołmogorowa prawdopodobieństwem występowania słowa z ustalonego zbioru. Znajduje to potwierdzenie choćby w algorytmach kompresji danych, na przykład w kodowaniu Huffmana: elementom występującym najczęściej przyporządkowuje się kod najkrótszy, a elementom występującym najrzadziej — najdłuższy.

Jeśli zastosujemy równość (2.4) do (2.2), będziemy mogli określać bliskość obiektów na podstawie częstości ich występowania. Naszym celem jest określanie bliskości znaczeniowej między słowami z języka naturalnego. Potrzebny nam więc będzie *korpus językowy* — reprezentatywny zbiór tekstów języka, na podstawie którego będziemy mogli określać częstości występowania słów. Powstało do tej pory wiele korpusów. Obecnie rozmiar żadnego z nich nie przekracza miliarda słów. Tłumaczyć to można dużym nakładem pracy, jaki trzeba ponieść tworząc tak duży zbiór, który jednocześnie powinien być reprezentatywny, czyli uwzględniać wszelkiego typu style wypowiedzi, grupy społeczne itp. Wspomnieć należy również, że język naturalny jest dynamiczny:

pewne słowa wychodzą z użytku, w ich miejsce pojawiają się nowe. Zdarza się również, że słowa nabierają nowego znaczenia i przez to pojawiają się w nowych kontekstach.

W [5] zastąpiono sztucznie tworzone korpusy zawartością sieci WWW. Jako narzędzia, określającego liczbę wystąpień słów, użyto wyszukiwarki internetowej Google. Jest to możliwe dzięki otrzymywaniu wraz z wynikami wyszukiwania liczby stron zawierających poszukiwaną frazę. Łatwą do dostrzeżenia zaletą tego rozwiązania jest liczba stron istniejących w Internecie. W momencie pisania tej pracy liczba stron w bazie danych wyszukiwarki np. Yahoo przekroczyła 50 miliardów. Zauważmy, że zdecydowana większość stron internetowych zawiera wiele słów, zatem ich liczbę w takim korpusie możemy liczyć w bilionach. Dlatego trudno przejść obojętnie obok zalet takiego zbioru tekstów. Następną korzyścią wynikającą z użycia wyszukiwarek jest aktualność. Zawartość baz danych wyszukiwarek uaktualniana jest automatycznie w miarę jak twórcy stron internetowych zaczynają posługiwać się słowami nowymi i ograniczają stosowanie słów mniej popularnych. Zauważmy również, że obecnie, aby publikować swoje teksty w sieci, wcale nie trzeba znać tak dużo szczegółów technicznych, jak to bywało jeszcze kilka lat temu. To również świadczy o zwiększeniu reprezentatywności (kosztem elitarności).

Oznaczmy przez Ω zbiór stron znajdujących się w bazie danych ustalonej wyszukiwarki, które mogą pojawić się w wynikach wyszukiwania, natomiast symbolem $\mathcal{S}$ — zbiór wszystkich słów występujących na stronach Ω. Niech $x \in \mathcal{S}$, wtedy przez $\boldsymbol{x} \subset \Omega$ będziemy oznaczać zbiór stron, które zawierają słowo x, a funkcja $f : \mathcal{S} \longrightarrow \mathbb{N}$ będzie określać liczbę znalezionych stron tak, że $f(x) = |\boldsymbol{x}|$. Ponadto oznaczmy przez $f(x, y)$ liczbę stron zawierających jednocześnie słowa x oraz y, czyli $f(x, y) = |\boldsymbol{x} \cap \boldsymbol{y}|$.

Zdefiniujmy miarę prawdopodobieństwa na zbiorze $\mathcal{S}$ określoną przez częstości występowania jego elementów. Oznaczmy ją przez P — niech $P(x)$ oznacza prawdopodobieństwo, że słowo x występuje na losowo wybranej stronie internetowej z Ω, a $P(x, y)$ prawdopodobieństwo, że oba słowa x i y występują na losowej stronie z Ω, czyli $P(x) = P(x, x)$. Warunkiem, jaki powinno spełniać P, jest

$$\sum_{x,y \in \mathcal{S}} P(x, y) = 1, \tag{2.5}$$

przy czym bierzemy również pod uwagę przypadki, gdy $x = y$. Jeśli przypomnimy sobie równość (2.4), zauważymy, że jest to również warunek brzegowy nierówności Krafta dla K ((1.8), s. 16). Prawdopodobieństwo $P(x, y)$

jest proporcjonalne do $f(x, y)$, możemy zatem zapisać

$$P(x, y) = \frac{f(x, y)}{N}. \tag{2.6}$$

Biorąc warunek (2.5), otrzymujemy

$$N = \sum_{x,y \in \mathcal{S}} |\boldsymbol{x} \cap \boldsymbol{y}|. \tag{2.7}$$

Dzięki zdefiniowaniu miary prawdopodobieństwa (2.6) i twierdzeniu 2.3.2 możemy wykorzystać znormalizowaną odległość informacyjną (2.2), by zdefiniować odległość kontekstową.

Definicja 2.3.3. Przyjmijmy oznaczenia f oraz N jak powyżej. *Znormalizowaną odległością internetową* (ang. *normalized web distance*) nazywamy

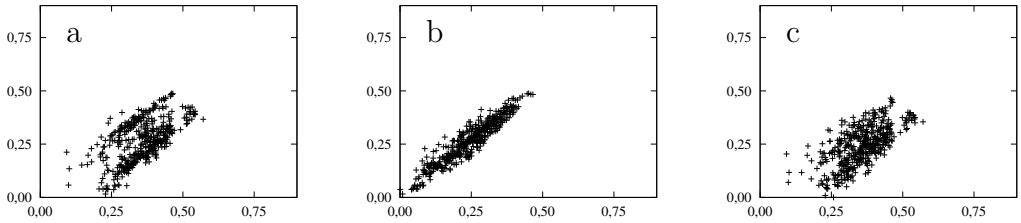
$$\mathrm{NWD}(x, y) = \frac{\log \max\{f(x), f(y)\} - \log f(x, y)}{\log N - \log \min\{f(x), f(y)\}}. \tag{2.8}$$

2.3.2. Własności

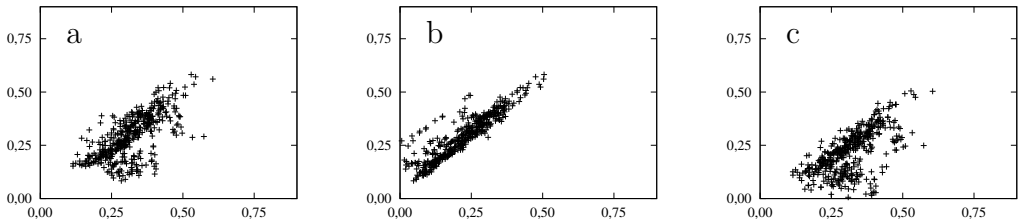
Funkcja NWD może przyjmować wartości z przedziału $[0, +\infty)$. Przypadek w którym $\mathrm{NWD}(x, y) = 0$ zajdzie, gdy $\boldsymbol{x} = \boldsymbol{y}$, czyli słowa x i y występują na stronach internetowych zawsze razem ze sobą. Natomiast $\mathrm{NWD}(x, y) = +\infty$, jeśli nie istnieje ani jedna strona w Ω, która zawiera jednocześnie x oraz y. W praktyce NWD rzadko przekracza liczbę 1, choć jest to zależne od przyjętej wartości N.

Ważną kwestią jest zgodność wyników otrzymywanych przy użyciu różnych wyszukiwarek. W naszych badaniach skupiliśmy się na wyszukiwarkach Yahoo, Google oraz Microsoft Live Search, ponieważ mają one obecnie najwięcej stron internetowych w swoich bazach. Przyjmiemy dla nich skrótowe oznaczenia odpowiednio Y, G, M, i będziemy stosować je w razie potrzeby jako indeksy dolne, np. f_M, N_Y, Ω_G. Jako dane testowe wzięliśmy 30 pseudolosowo wybranych słów z powieści *Lord Jim* Josepha Conrada — jedną próbkę z oryginału, drugą z polskiego tłumaczenia, przy czym dla obu wersji językowych wybory słów były niezależne od siebie. Zatem w przypadku obu języków otrzymaliśmy — biorąc każde z każdym — 435 par słów. Dla tych próbek sprawdziliśmy korelację odległości otrzymywanych za pomocą wszystkich trzech wyszukiwarek. Rysunek 2.1 przedstawia wykres zgodności między wynikami otrzymanymi dla próbki słów angielskich, a rysunek 2.2 —

polskich. Dla każdej wyszukiwarki przyjęliśmy $N = 2^{50}$.



Rysunek 2.1: Korelacja między wynikami NWD dla słów angielskich: a) G i Y, b) M i Y, c) G i M.



Rysunek 2.2: Korelacja między wynikami NWD dla słów polskich: a) G i Y, b) M i Y, c) G i M.

Największy współczynnik korelacji osiągnięto dla słów angielskich między wyszukiwarkami M i Y ($\rho_{M,Y} = 0{,}940$, rys. 2.1 b). Na wykresach widać również, że G słabo koreluje z pozostałymi dwiema wyszukiwarkami. Z drugiej strony okazuje się, że dla M istnieją słowa x, y takie, że $f(x,y) > \max\{f(x), f(y)\}$, co znaczyłoby, że $|\boldsymbol{x} \cap \boldsymbol{y}| > \max\{|\boldsymbol{x}|, |\boldsymbol{y}|\}$. Przykładem mogą być słowa *natychmiast* oraz *główna*, a liczby stron dla nich (w momencie pisania tej pracy) to: $f_M(natychmiast) = 102000$, $f_M(główna) = 299000$, $f_M(natychmiast, główna) = 516000$. Przez to otrzymywane odległości dla tych par słów są ujemne. Dlatego zdecydowaliśmy się w dalszych doświadczeniach skupić na wynikach uzyskiwanych przy użyciu wyszukiwarki Yahoo.

Rozważmy, jak się ma liczba N w stosunku do $|\Omega|$. Możemy przyjąć, że wszystkie strony internetowe z Ω zawierają przynajmniej jedno słowo z $\mathcal{S}$. Gdyby tak nie było, strona niezawierająca żadnego słowa nie mogłaby w żaden sposób pojawić się w wynikach wyszukiwania, co znaczyłoby, że nie należy do Ω. Zatem zliczając przypadki tylko gdy $x = y$ w (2.7), mamy $N \geqslant |\Omega|$. Równość zaszłaby jedynie, gdy wszystkie strony w Ω zawierałyby dokładnie jedno słowo. Zdecydowana większość stron internetowych zawiera wiele słów.

Przedstawimy teraz lemat, który pozwoli nam osiągnąć lepsze ograniczenie dolne liczby N.

Lemat 2.3.4. Dla funkcji $\nu : [0, a]^2 \times (a, +\infty) \longrightarrow [0, +\infty)$ danej wzorem

$$\nu(r, s, t) = \frac{r}{t - s}$$

zachodzi równość

$$\max_{r,s} \nu(r, s, t) = \frac{a}{t - a}.$$

Dowód. Wystarczy znaleźć maksimum dla licznika i minimum dla mianownika:

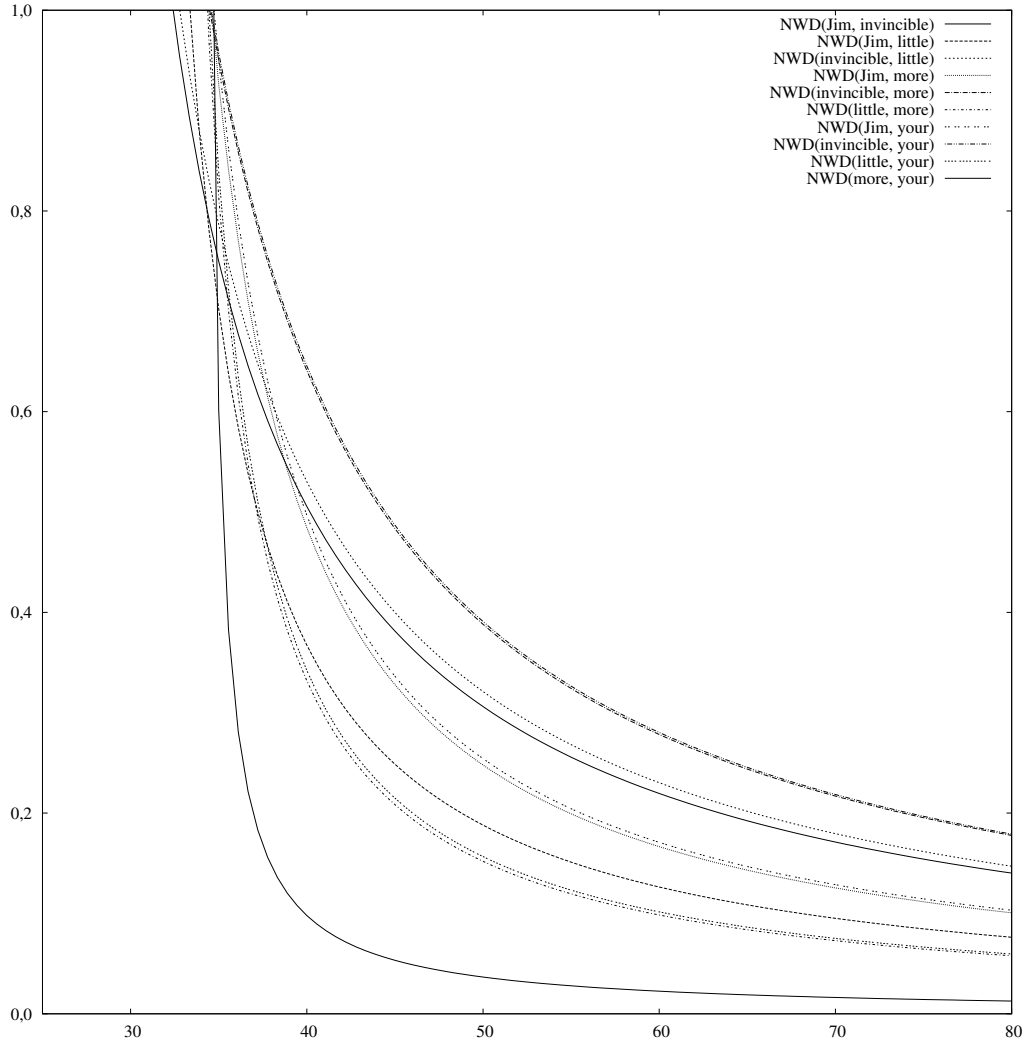
$$\max_{r,s} \frac{r}{t - s} = \frac{\max_{r,s} r}{\min_{r,s}(t - s)} = \frac{a}{t - \max_{r,s} s} = \frac{a}{t - a},$$

co kończy dowód. □

Jak już wspomnieliśmy, jeśli $f(x, y) = 0$, wtedy $\mathrm{NWD}(x, y) = +\infty$. Rozpatrzmy zatem przypadki, gdy $f(x, y) \geqslant 1$. Zauważmy, że $f(x)$ i $f(x, y)$ dla dowolnych słów $x, y \in \mathcal{S}$ jest ograniczone przez $|\Omega|$. Przyjmijmy oznaczenia: $a = \log |\Omega|$, $r = \log \max\{f(x), f(y)\} - \log f(x, y)$, $s = \log \min\{f(x), f(y)\}$, $t = \log N$. Możemy zatem zapisać $\mathrm{NWD}(x, y)$ jako $\nu(r, s, t)$. Aby dla wszystkich r i s zachodziło $\nu(r, s, t) \leqslant 1$, na mocy lematu 2.3.4 wystarczy przyjąć $t \geqslant 2a$. Dlatego będziemy przyjmować takie N, że $\log N \geqslant 2 \log |\Omega|$. Zagwarantuje nam to górne ograniczenie odległości przez 1, poza przypadkami gdy badane słowa nie występują ani na jednej wspólnej stronie internetowej.

Liczby stron zindeksowanych przez wyszukiwarki Google i Yahoo były upubliczniane do roku 2005. Obecnie, ze względu na wzmożoną konkurencję między nimi, liczby te nie są podawane do publicznej wiadomości. Będziemy więc szacować rozmiar bazy danych na podstawie liczby stron, które wyszukiwarka znajduje dla słów bardzo popularnych, takich jak przedimek *the* czy *a* w języku angielskim. Biorąc pod uwagę, że zdecydowana większość tekstów w internecie jest napisana po angielsku, przyjmiemy, że $\log N > 2 \log f(the)$. Wielkość $\log N$ w pewnych granicach nie ma dużego wpływu na wyniki wnioskowania opartego na odległości NWD, przy założeniu, że otrzymywane odległości mają charakter względny i nie zmieniamy N dla różnych par słów w obrębie tego samego doświadczenia. Będziemy wykorzystywać w naszych badaniach odległość kontekstową w ten sposób, że różnica czy też stosunek między odległościami dla danych dwóch par słów będzie ważny pod warunkiem, że wartość ta będzie duża. W razie gdy odległości w przybliżeniu będą sobie równe, będziemy uznawać je za jednakowe i wyniki polegające na ich porównywaniu uważać za mało pewne. Warto jeszcze wspomnieć, że z własności funkcji NWD (w uproszczeniu ν) wynika,

że jeśli dla różnych $N > |\Omega|^2$ uporządkowanie odległości NWD według relacji mniejszości zmienia się, to znaczy, że odległości te różnią się niewiele. Przykładem może być wykres odległości między pięcioma słowami wybranymi z wcześniejszej próbki słów angielskich w zależności od przyjętej wartości $\log N$ (rys. 2.3), na którym powyższe warunki są spełnione przy wartościach $\log N$ powyżej 40.



Rysunek 2.3: Wykres odległości NWD_Y w zależności od $\log N$ dla każdej pary między słowami *Jim*, *invincible*, *little*, *more*, *your*.

Przy okazji na rysunku tym możemy zaobserwować ciekawą właściwość. Załóżmy, że dla ustalonej pary słów liczba stron im odpowiadających przestanie rosnąć wraz z upływem czasu, lecz liczba stron w internecie nadal będzie się zwiększać, co wiąże się również ze wzrostem N. Wtedy odległość kontekstowa między słowami będzie maleć. Tłumaczyć to możemy

zwiększaniem się natężenia wspólnej cechy obu wyrazów — przeistaczanie się w archaizm, czyli według założeń zaprzestanie używania tych słów w internecie.

Pokażemy jeszcze, że odległość NWD nie jest metryką. Dla wszystkich słów x i y mamy $\mathrm{NWD}(x,x) = 0$ oraz $\mathrm{NWD}(x,y) \geqslant 0$ poza przypadkami anomalii jak we wspomnianym przypadku przy użyciu przeglądarki Microsoft Live Search. Nie zachodzi jednak konieczny dla metryki warunek, że $\mathrm{NWD}(x,y) > 0$ dla różnych x i y. Przekonamy się o tym, jeśli weźmiemy takie x i y, że $\boldsymbol{x} = \boldsymbol{y}$. Wtedy $f(x) = f(y) = f(x,y)$ i $\mathrm{NWD}(x,y) = 0$. Ponadto, jeśli weźmiemy $\boldsymbol{z} = \boldsymbol{x} \cup \boldsymbol{y}$, $\boldsymbol{x} \cap \boldsymbol{y} = \emptyset$, $\boldsymbol{x} = \boldsymbol{x} \cap \boldsymbol{z}$, $\boldsymbol{y} = \boldsymbol{y} \cap \boldsymbol{z}$ oraz $|\boldsymbol{x}| = |\boldsymbol{y}| = \sqrt{N}$, wtedy $f(x) = f(y) = f(x,z) = f(y,z) = \sqrt{N}$, $f(z) = 2\sqrt{N}$ i $f(x,y) = 0$, co daje w wyniku $\mathrm{NWD}(x,y) = \infty$, oraz $\mathrm{NWD}(x,z) = \mathrm{NWD}(z,y) = 2/\log N$. Więc nie mamy zachowanej również nierówności trójkąta $\mathrm{NWD}(x,y) \leqslant \mathrm{NWD}(x,z) + \mathrm{NWD}(z,y)$.

2.3.3. Przykłady

Poniżej zaprezentujemy na przykładach obliczone odległości kontekstowe. Jak już napisaliśmy, będziemy używać wyszukiwarki Yahoo, dla której przyjęliśmy $N = 2^{50}$.

Na początek weźmiemy kilka przykładów dla słów bliskich sobie kontekstowo.

$$\mathrm{NWD}(\textit{łyżka},\ \textit{widelec}) = 0{,}1118$$
$$\mathrm{NWD}(\textit{dętka},\ \textit{wentyl}) = 0{,}1675$$
$$\mathrm{NWD}(\textit{prostokąt},\ \textit{kwadrat}) = 0{,}1047$$
$$\mathrm{NWD}(\textit{żagiel},\ \textit{maszt}) = 0{,}1548$$

Porównajmy otrzymane wyniki z tymi dla par słów z różnych dziedzin.

$$\mathrm{NWD}(\textit{łyżka},\ \textit{wentyl}) = 0{,}3878$$
$$\mathrm{NWD}(\textit{dętka},\ \textit{kwadrat}) = 0{,}3318$$
$$\mathrm{NWD}(\textit{prostokąt},\ \textit{maszt}) = 0{,}3657$$
$$\mathrm{NWD}(\textit{żagiel},\ \textit{widelec}) = 0{,}3758$$

Bardzo bliskie odległości otrzymamy dla słów, które występują prawie nierozłącznie, często są to nazwy geograficzne.

$$\mathrm{NWD}(\textit{Skarżysko},\ \textit{Kamienna}) = 0{,}0403$$
$$\mathrm{NWD}(\textit{United},\ \textit{States}) = 0{,}0233$$
$$\mathrm{NWD}(\textit{Buenos},\ \textit{Aires}) = 0{,}0147$$
$$\mathrm{NWD}(\textit{zażółć},\ \textit{gęślą}) = 0{,}0075$$

Ostatnia para wyrazów wchodzi w skład bardzo popularnego w internecie zwrotu, który służy testowaniu poprawności kodowania liter z polskimi znakami diakrytycznymi.

Sprawdźmy jeszcze, jakie są odległości między nazwami państw wspomnianymi na początku tego rozdziału.

$$\mathrm{NWD}(\textit{Australia}, \textit{Austria}) = 0{,}1049$$
$$\mathrm{NWD}(\textit{Australia}, \textit{Nowa Zelandia}) = 0{,}3155$$

Nie jest to wynik, jaki chcieliśmy osiągnąć. Jest to spowodowane przewagą ilościową w internecie języka angielskiego, w którym występuje zarówno słowo *Austria*, jak i *Australia*, ale nie występuje słowo *Nowa Zelandia*. Z tego względu duża odległość dla drugiej pary wynika ze stosunkowo małej liczby stron, na których oba te słowa występują razem, w porównaniu do stron, na których występują one oddzielnie. Zobaczmy zatem, czy używając tylko języka angielskiego dla podanego przykładu, osiągniemy pożądany wynik.

$$\mathrm{NWD}(\textit{Australia}, \textit{Austria}) = 0{,}1049$$
$$\mathrm{NWD}(\textit{Australia}, \textit{New Zealand}) = 0{,}0807$$

Zgodnie z oczekiwaniem *Australia* jest bliższa kontekstowo *Nowej Zelandii*. Widzimy więc, że wyniki są zaburzone, gdy chociaż jedno badane słowo występuje w innym języku i błąd jest tym większy, im język ten jest bardziej popularny w internecie.

Rozdział 3

Program komputerowy

3.1. Wprowadzenie

Wykorzystamy odległość NWD przedstawioną w rozdziale drugim do badania bliskości kontekstowej w grupie wyrazów. Naszym celem będzie odpowiedź na pytanie: który wyraz z danej listy nie pasuje do reszty? Podobne zadanie często występuje w testach badających inteligencję. Wyniki, które otrzymujemy zaproponowaną metodą, nie dorównują z pewnością inteligencji ludzkiej. Nasz algorytm nie jest w stanie wychwycić tak drobnych niuansów, jak człowiek — na przykład zazwyczaj nie potrafi rozróżnić płci na podstawie imion, wskazuje jednak wśród listy słów to, które nie jest imieniem. Mimo tych niedoskonałości, rezultaty są dosyć interesujące.

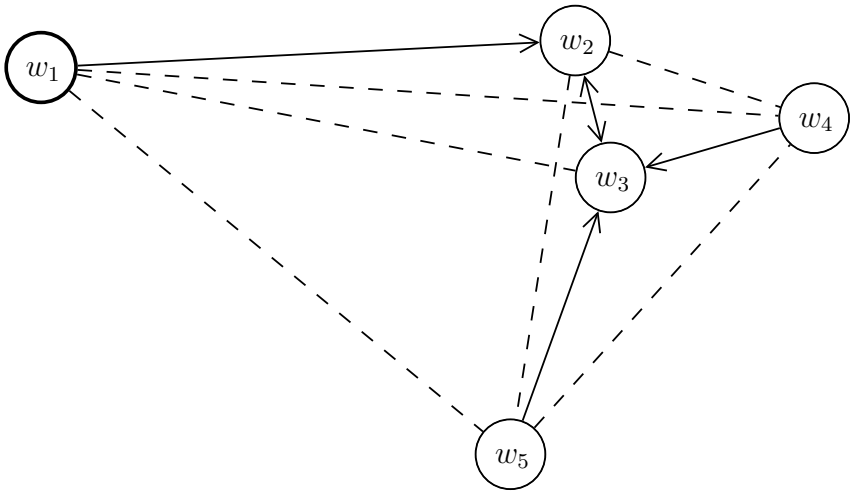
Jedną z najważniejszych zalet metody jest brak ograniczeń na dziedziny, do których należą badane słowa. Biorąc pod uwagę rozmiar korpusu językowego, który wykorzystujemy, widzimy, że zakres tematyczny tekstów, dla których możliwe jest zastosowanie naszego algorytmu, jest bardzo szeroki.

Zastosowaniem algorytmu może być na przykład wspomaganie automatycznej korekcji tekstów. Wyobraźmy sobie program komputerowy eliminujący błędy w tekstach wprowadzanych przez użytkownika za pomocą klawiatury. Będą to najczęściej proste zamiany i pominięcia liter oraz typowe błędy ortograficzne. Niech program taki dla każdego błędnie wpisanego wyrazu znajduje prawidłowy zamiennik. Możliwa jest sytuacja, że zostanie znaleziony więcej niż jeden wyraz, którym można zastąpić słowo wpisane błędnie. Wtedy nasz algorytm — na podstawie kontekstu, czyli grupy wyrazów otaczających poprawiany — umożliwiałby wybór najlepiej pasującego zamiennika.

3.2. Algorytm Find-Inadequate

Danymi wejściowymi jest lista słów. Algorytm konstruuje graf pełny *nieskierowany*, w którym wierzchołki są słowami z podanej listy, natomiast długościom krawędzi odpowiadają odległości NWD pomiędzy poszczególnymi słowami. Zadaniem jest wskazanie wierzchołka, który jest najbardziej oddalony od reszty wierzchołków. W tym celu dla każdego wierzchołka wyznaczamy najbliższego sąsiada. Słowo, które ma najbardziej oddalonego najbliższego sąsiada, jest poszukiwanym przez nas najmniej pasującym wierzchołkiem.

Spójrzmy na rysunek 3.1. Najbliżsi sąsiedzi poszczególnych wierzchołków są wskazywani przez skierowane krawędzie — przy czym nadanie kierunku krawędziom na rysunku ma na celu jedynie wskazanie najbliższego sąsiada. Na przykład wierzchołki w_2 i w_3 są sobie wzajemnie najbliższe. Poszukiwanym wierzchołkiem jest w_1, którego najbliższym sąsiadem jest w_2.

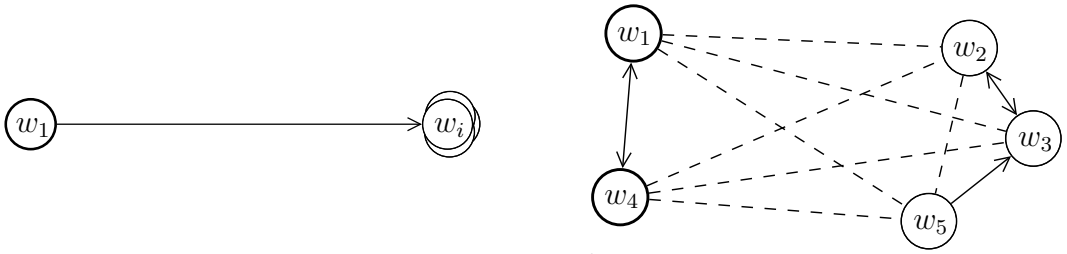


Rysunek 3.1: Przykładowy graf zbudowany w wyniku działania algorytmu.

Mamy zatem określony sposób znajdowania niepasującego słowa. Chcemy jeszcze określić pewność, z jaką dokonujemy wyboru. Na rysunku 3.2 (s. 30) przedstawiliśmy skrajne przypadki pewności, z jaką możemy dokonać wyboru.

Widzimy, że w grafie a) z całą pewnością najbardziej oddalonym wierzchołkiem jest w_1, ponieważ reszta wierzchołków (oznaczona wspólnie jako w_i) jest bardzo blisko siebie — przyjmijmy, że odległość między nimi wynosi 0. Zatem pewność, z jaką wybierzemy w_1 wynosi 1.

Natomiast z grafu b) równie dobrze może być wybrany wierzchołek w_1, jak i w_4. Ponieważ w specyfikacji programu zawarliśmy warunek, że wybrane ma być dokładnie jedno słowo, będzie to albo w_1, albo w_4, ale pewność tego wyboru będzie równa 0.



a b

Rysunek 3.2: Grafy ilustrujące skrajne przypadki pewności wyboru niepasującego słowa: a) $s = 1$, b) $s = 0$.

Powyższe rozważania pozwalają nam zaproponować sposób obliczenia pewności wyboru.

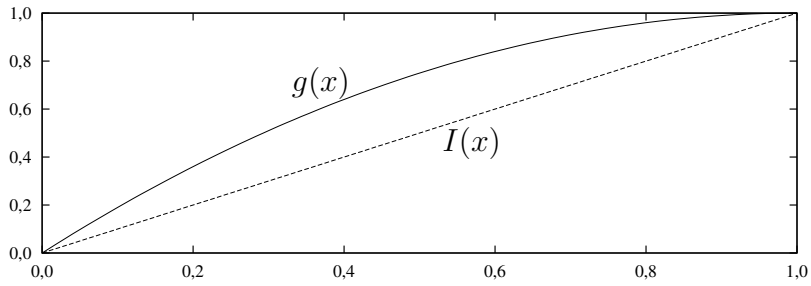
Definicja 3.2.1. Oznaczmy przez d_0 odległość między wybranym przez algorytm wierzchołkiem a jego najbliższym sąsiadem. Na rysunku 3.1 $d_0 = |\{w_1, w_2\}| = \mathrm{NWD}(w_1, w_2)$. Ponadto przez d_1 oznaczmy analogiczną odległość dla drugiego w kolejności wierzchołka (w porządku malejącym względem odległości od najbliższego sąsiada) — odpowiednio $d_1 = |\{w_3, w_5\}| = \mathrm{NWD}(w_3, w_5)$. Pewność wyboru wierzchołka definiujemy jako

$$s = g(1 - \frac{d_1}{d_0}), \tag{3.1}$$

gdzie g jest rosnącą funkcją ciągłą na przedziale $[0, 1]$, taką że $g(0) = 0$ oraz $g(1) = 1$.

Na rysunku 3.2 a) d_1 jest równe 0, zatem $s = 1$, natomiast na b) $d_1 = d_0$, więc $s = 0$.

Bardzo trudno jest otrzymać d_1 bliskie zera, zatem chcąc zrównoważyć ten fakt, obieramy za g funkcję wklęsłą. W naszym programie przyjęliśmy arbitralnie $g(x) = -x^2 + 2x$, co daje zadowalające wyniki.



Rysunek 3.3: Wykres funkcji $g(x) = -x^2 + 2x$ oraz $I(x) = x$.

Poniżej przedstawiamy algorytm zapisany w formie pseudokodu.

FIND-INADEQUATE(L)
1 $Q \leftarrow \emptyset$
2 **for** każde słowo $w \in L$
3 **do** $d \leftarrow \min\{\text{NWD}(w, x) : x \in L \setminus \{w\}\}$
4 APPEND-TO-LIST($Q, (d, w)$)
5 $d_0, w_0 \leftarrow$ EXTRACT-MAX(Q)
6 $d_1, w_1 \leftarrow$ EXTRACT-MAX(Q)
7 **return** $(w_0,\ g(1 - \frac{d_1}{d_0}))$

W linii 1 następuje utworzenie pustej listy, do której w liniach 2–4 dołączamy pary $(d,\ w)$, gdzie d to odległość do najbliższego sąsiada dla słowa w. Następnie wybieramy i usuwamy z listy Q parę, w której d jest największe, a otrzymane wartości przypisujemy do d_0 i w_0 (l. 5). Czynność powtarzamy i przypisujemy wynik odpowiednio do d_1 i w_1 (l. 6). Słowo w_0 jest poszukiwanym przez nas niepasującym do reszty wyrazem. Dodatkowo procedura wydaje pewność, z jaką otrzymaliśmy wynik, obliczoną zgodnie z (3.1).

3.3. Język programowania Python

Do zaimplementowania algorytmu użyliśmy języka Python w wersji 2.5. Jest to w pełni obiektowy język interpretowany. Mimo, że wszystko w nim jest obiektem, pozwala programiście pisać w stylu proceduralnym a nawet funkcyjnym. Jest bardzo łatwy w nauce i ma prostą, wręcz minimalistyczną składnię, przez to bywa mylony czasem z pseudokodem. Świadczy to również o jego dużej ekspresywności.

Ponadto zapewnia dużą przenośność programów — działa na różnorodnych platformach systemowych: od maszyn klasy *mainframe*, przez komputery osobiste, aż po urządzenia mobilne. Wyposażono go w bogatą bibliotekę standardową, co również zwiększa przenośność, a jednocześnie upraszcza wdrożenie gotowego produktu.

Dzięki automatycznemu zarządzaniu pamięcią (ang. *garbage collection*), wygodnym, abstrakcyjnym strukturom danych i dynamicznemu typowaniu stał się językiem wysokiego poziomu, ułatwiającym szybki rozwój pisanych w nim aplikacji.

Poniżej zaprezentujemy minimalny zestaw elementów języka, pomocny w zrozumieniu kodu przedstawionego przez nas programu komputerowego. Dobrym wprowadzeniem do Pythona jest dostępna nieodpłatnie w internecie książka [9]. W czasie kodowania przydatna jest również strona domowa języka [10], na której znajdziemy pomoc dotyczącą wielu aspektów Pythona, w szczególności funkcji i klas z biblioteki standardowej.

Pierwsze, co odróżnia Python od większości innych języków programowania, jest istotność tzw. białych znaków w kodzie, ponieważ bloki instrukcji zaznaczane są poprzez wcięcia w tekście programu. Popatrzmy na przykładową definicję klasy.

```
class Example:
  'Przykładowa klasa'
  def __init__(self, initValue):
    'Konstruktor'
    self.value = initValue
  def __call__(self, value):
    return self.value + value
```

Zauważmy, że definicje metod są przesunięte o stałą liczbę odstępów w prawo w stosunku do nagłówka klasy, podobnie jak ciała funkcji w stosunku do ich nagłówków. Zwykłe funkcje poza klasami definiujemy analogicznie, jak metody należące do klas — również poprzedzamy słowem kluczowym `def`. Napisy umieszczone tuż pod nagłówkiem dokumentują definiowany obiekt, tak że np. klasa `Example` jako obiekt posiada pole `__doc__` typu `str` o treści `'Przykładowa klasa'`. Metody o nazwach z podwójnymi znakami podkreślenia mają specjalne znaczenie. Na przykład `__init__` jest konstruktorem. Spis wszystkich metod specjalnych znajdziemy w [10]. Dzięki zdefiniowaniu metody `__call__`, obiekt zachowuje się jak funkcja.

```
>>> example = Example(10)
>>> print example(1)
11
```

Instrukcja `print` drukuje na standardowym wyjściu napisową reprezentację obiektu, niezależnie od jego klasy. Warto przy okazji wspomnieć, że Python jest językiem interaktywnym, czyli wszystko, co wpiszemy po uruchomieniu interpretera, jest natychmiast wykonywane. Dlatego przy uruchomieniu samego interpretera zostaje wyświetlany znak zachęty `>>>` — wszystko występujące po tym znaku do końca linii jest wpisywane przez nas. Natomiast w powyższym przykładzie `11` jest odpowiedzią programu.

Wiemy już, jak definiować klasy i funkcje, zobaczmy więc, jakie typy danych nam udostępniono. Do reprezentacji liczb całkowitych mamy typy `int` oraz `long`. Rozmiar liczb typu `long` ograniczony jest jedynie rozmiarem dostępnej pamięci operacyjnej i czasem procesora potrzebnym do operowania na tak dużych liczbach. Jeśli liczba nie mieści się w zakresie typu `int`, jest automatycznie konwertowana do typu `long`. Dla liczb zmiennoprzecinkowych mamy klasę `float`, która odpowiada typowi `double` znanemu z języka C.

Obsługę napisów zapewniają klasy `str` oraz `unicode`. Ważne jest, że obiekty obu tych klas są niemodyfikowalne — jeśli wykonujemy operację na napisie, w rzeczywistości tworzony jest nowy, a poprzedni jest usuwany z pamięci o ile nie wskazuje na niego żadna inna zmienna.

```
>>> format = u'%.2f'
>>> format % (22.0/7.0)
u'3.14'
```

W powyższym przykładzie definiujemy napis typu `unicode`, a następnie otrzymujemy napis zawierający sformatowaną liczbę. Aby dokonać konwersji np. pomiędzy klasą `int` a `str`, można też użyć nazwy klasy jak funkcji.

```
>>> n = 10
>>> int(str(n)) == n
True
```

Typy proste możemy łączyć w bardziej skomplikowane struktury m.in. za pomocą list, krotek (ang. *tuple*), słowników i zbiorów.

Listy służą do przechowywania obiektów dowolnych klas w ustalonej kolejności. Dostęp do ich elementów uzyskujemy za pomocą nawiasów `[]` — jest to charakterystyczne dla wszystkich sekwencji, którymi oprócz list są również napisy i krotki.

```
>>> lista = range(10)
>>> lista
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> lista[0]
0
>>> lista[-1]
9
>>> lista[3:5]
[3, 4]
>>> lista.reverse()
>>> lista
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Typem podobnym do list, lecz niemodyfikowalnym, są wspomniane krotki. Konstruujemy je przy pomocy nawiasów `()`, podając wewnątrz nich wszystkie elementy krotki.

```
1 >>> krotka = ('Lublin', 2008)
2 >>> print krotka[1]
3 2008
4 >>> miasto, rok = krotka
5 >>> print miasto
6 Lublin
```

Jak widać, krotki również mogą zawierać elementy różnych typów. Przypisanie widoczne w 4 linii jest możliwe dzięki automatycznemu rozwijaniu krotek.

Często używaną strukturą danych są słowniki, znane w innych językach jako mapy. Słownik służy do kojarzenia klucza z wartością. Kluczem może być każdy niemodyfikowalny obiekt, zatem nie mogą to być listy czy też omawiane niżej modyfikowalne zbiory (`set`).

```
>>> slownik = {'miasto':'Lublin', 'rok':2008}
>>> print slownik['rok']
2008
>>> slownik[5] = 120
>>> slownik
{'rok': 2008, 5: 120, 'miasto': 'Lublin'}
```

Ostatnimi omawianymi przez nas typami są `set` oraz `frozenset`. Służą one do przechowywania elementów dowolnego typu z pominięciem kolejności. Różnią się między sobą jedynie tym, że pierwszy jest modyfikowalny, a drugi nie, więc może być użyty jako klucz w słowniku.

```
>>> fset = frozenset([1,'miasto',0,3,2])
>>> fset
frozenset([0, 1, 2, 3, 'miasto'])
```

Warte wspomnienia są aspekty funkcyjne języka Python. Możemy używać funkcji anonimowych w miejscach, gdzie wymagane jest zastosowanie obiektu funkcji. Używamy w tym celu słowa kluczowego `lambda`. Często spotykanym zastosowaniem jest np. definiowanie w ten sposób klucza sortowania listy.

```
>>> lista = range(-5, 5, 3)
>>> lista
[-5, -2, 1, 4]
>>> lista.sort(key=lambda x: abs(x))
>>> lista
[1, -2, 4, -5]
```

W powyższym przykładzie posortowaliśmy listę biorąc jako klucz wartość bezwzględną jej elementów.

Ponadto przydatną konstrukcją są tzw. listy składane (ang. *list comprehension*).

```
>>> kwadratyNieparzystych = [ x*x for x in range(20) if x%2 == 1 ]
>>> kwadratyNieparzystych
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

Możemy pominąć wyrażenie warunkowe (`if ...`), jeśli tworzona lista ma zawierać tyle elementów, ile źródłowa.

3.4. Implementacja algorytmu

Omówimy teraz krótko kluczowe fragmenty kodu programu. Pominiemy np. kwestie buforowania na dysku lokalnym wyników z liczbą stron uzyskanych we wcześniejszych zapytaniach, czy też wczytywanie konfiguracji dotyczącej wyszukiwarki. Cały kod programu przedstawiamy w podrozdziale 3.6.

Nasz program, aby wyznaczyć odległość NWD, musi się połączyć za pomocą protokołu HTTP z daną wyszukiwarką i przetworzyć otrzymane dane tak, aby w rezultacie otrzymać liczbę stron z badanym słowem lub parą słów. Służy do tego celu klasa `WebChecker`.

Pierwszym krokiem jest uformowanie i wysłanie zapytania do wyszukiwarki oraz odbiór odpowiedzi. Zadanie to wykonuje metoda `__getHtml`.

```
1 def __getHtml(self, *words):
2   query = '+'.join([quote(('+"%s"' % word).encode('utf-8')) \
3       for word in words])
```

W powyższym fragmencie kodujemy słowa do postaci odpowiedniej dla protokołu HTTP (funkcja `quote`) i zapisujemy jako `+"słowo1" +"słowo2"`, jeśli `words` jest listą dwóch słów lub `+"słowo"` jeśli `words` zawiera tylko jedno słowo. Dodanie znaku `+` zapewnia nam wyszukiwanie stron zawierających wszystkie zadane słowa.

Następnie formujemy ostateczny adres URL, wysyłamy zapytanie i odbieramy dokument HTML.

```
4   url = self.url % query
5   request = urllib2.Request( url, \
6       headers={'User-Agent':'Normalized Web Distance Computer'} )
7   return urllib2.urlopen(request).read()
```

Zmienna `self.url` jest szablonem adresu, do którego wstawiamy wcześniej uzyskane zapytanie, na przykład dla wyszukiwarki Yahoo jest to `http://search.yahoo.com/search?p=%s&y=Search&ei=UTF-8`, gdzie za `%s` zostanie podstawiona treść zapytania.

Mając uzyskany dokument HTML, musimy wydobyć z niego liczbę stron.

```
1 def __parsed(self, html):
2   try:
3     foundStr = self.regex.search(html).groupdict()['number']
4     return ''.join([char for char in foundStr if char.isdigit()])
5   except:
6     return '1'
```

Przeszukujemy tekst strony internetowej uzyskanej z wyszukiwarki, aby znaleźć fragment pasujący do wyrażenia regularnego `self.regex` specyficznego dla danej wyszukiwarki (linia 3). Tak uzyskany fragment filtrujemy, zostawiając jedynie cyfry i składając je w liczbę (linia 4). W razie gdy nie uda się dopasować wyrażenia regularnego do tekstu, uznajemy, że liczba stron wynosi 1. Gdyby metoda zwracała jako liczbę stron 0, napotkalibyśmy kłopoty związane z reprezentacją nieskończoności po zlogarytmowaniu 0.

Obie zaprezentowane wcześniej metody stosujemy w metodzie `__call__`.

```
1 def __call__(self, *words):
2   "Zwraca liczbę stron zawierających słowo/słowa."
3   html = self.__getHtml(*words)
4   return int(self.__parsed(html))
```

Dzięki temu możemy używać obiektów klasy `WebChecker` jak funkcji.

```
>>> print checker('Australia')
2310000000L
```

W powyższym kodzie `checker` jest obiektem klasy `WebChecker`.

Mamy zatem możliwość uzyskiwania liczby stron zawierających badane słowa. Możemy więc przejść do budowania grafu przez nasz algorytm. Klasą za to odpowiedzialną jest `NwdComputer`. Pierwszym krokiem jest sprawdzenie odległości NWD dla każdej pary zadanych słów w metodzie `__checkAll`.

```
1 def __checkAll(self, words):
2   threads = [ Thread(target=self.__check, args=(word,) ) \
3       for word in words ]
```

```
4   for i in range(len(words)):
5     for j in range(i):
6       threads.append(Thread(target=self.__check, \
7           args=(words[i], words[j])))
8   for thread in threads:
9     thread.start()
10  for thread in threads:
11    thread.join()
```

Aby skrócić czas oczekiwania na wynik, wszystkie zapytania do wyszukiwarki wykonują się współbieżnie w oddzielnych wątkach programu. W liniach 2–3 tworzymy listę wątków, obsługujących zapytania dla pojedynczych słów, a w liniach 4–7 dodajemy do tej listy zapytania dla par słów. Następnie uruchamiamy wszystkie wątki (8–9) i czekamy na ich zakończenie (10–11).

Spójrzmy jeszcze na metodę `__check` i zwróćmy uwagę, że przechowuje ona wyniki swojego działania w polu `self.checked`, który jest słownikiem.

```
1  def __check(self, *words):
2    self.checked[frozenset(words)] = self.counter(*words)
```

Wywołanie `self.counter(*words)` zwraca liczbę stron uzyskaną z lokalnego bufora lub w razie konieczności pobraną z wyszukiwarki internetowej.

Teraz możemy już utworzyć graf.

```
1  def makeGraph(self, words):
2    '''Zwraca graf pełny między słowami z words;
3      długości krawędzi to odległości kontekstowe.'''
4    self.__checkAll(words)
5    result = {}
6    for i in range(len(words)):
7      for j in range(i):
8        key = frozenset((words[i],words[j]))
9        word1Count = self.checked[frozenset((words[i],))]
10       word2Count = self.checked[frozenset((words[j],))]
11       word12Count = self.checked[key]
12       result[key] = (log(max(word1Count, word2Count))-log(word12Count)) \
13           / (self.logN - log( min(word1Count, word2Count)))
14   return result
```

Wynikiem jest słownik `result` z kluczami typu `frozenset` zawierającymi pary słów, natomiast wartościami są odległości NWD między wyrazami.

Z grafu za pomocą funkcji `inadequate` uzyskujemy najmniej pasujące słowo wraz z pewnością jego wybrania.

```
1  def inadequate(graph):
2    '''Wybiera z grafu wierzchołek-słowo,
3    które jest najbardziej oddalone od reszty.
4    Zwraca słowo oraz pewność, z jaką zostało ono wybrane.'''
5    weights = {}
6    for edge, value in graph.items():
7      for vertex in edge:
8        if vertex in weights:
9          weights[vertex].append(value)
10       else:
11         weights[vertex] = [value]
12   minimalWeights = []
13   for vertex in weights.keys():
14     minimalWeights.append({'word':vertex, \
15       'value':reduce(min, weights[vertex])})
16   minimalWeights.sort(key=lambda x: x['value'], reverse=True)
17   chosen = minimalWeights[0]
18   second = minimalWeights[1]
19   sureness = scaleFunction(1.0 - second['value']/chosen['value'])
20   return chosen['word'], sureness
```

W liniach 5–11 tworzymy słownik przechowujący dla każdego słowa listy długości krawędzi z nim incydentnych. Następnie w 12–16 tworzymy posortowaną listę słowników zawierających słowo i długość najkrótszej krawędzi z nim incydentnej. Pozostałe linie odpowiadają za wybór niepasującego słowa i obliczenie pewności jego wyboru zgodnie z algorytmem Find-Inadequate (s. 31). Przy czym funkcję `scaleFunction` definiujemy w następujący sposób w kodzie programu.

```
scaleFunction = lambda x: -x*x + 2.0*x
```

Powyższy kod został umieszczony w dwóch modułach języka Python. Aby ułatwić użytkownikowi pracę, stworzyliśmy program ramowy, który daje możliwość sprawdzenia liczby stron zawierających słowo, odległości między dwoma słowami, jak też prezentuje działanie głównego algorytmu Find-Inadequate — wybór następuje automatycznie w zależności od liczby podanych argumentów programu. Kod tego programu również został zamieszczony w podrozdziale 3.6. Przykładowa sesja pracy z aplikacją wygląda następująco.

```
$ ./main.py
Proszę podać jako argument(y) programu:
```

```
1 słowo - aby poznać liczbę stron, które je zawierają,
2 słowa - aby poznać odległość kontekstową między nimi,
więcej słów - aby dowiedzieć się, które nie pasuje do reszty.
$ ./main.py Szwecja
Liczba stron ze słowem "Szwecja" wynosi 17900000.
$ ./main.py Szwecja Szczebrzeszyn
Odległość między słowem "Szwecja" a "Szczebrzeszyn" wynosi 0.3299.
$ ./main.py Szwecja Szczebrzeszyn Sztokholm
Z pewnością 85.86% niepasujące słowo to "Szczebrzeszyn".
```

3.5. Przykłady

Zaprezentujemy teraz działanie programu na kilku przykładach. Sprawdźmy, jak zadziała program dla wspomnianych w drugim rozdziale słów *Australia*, *Austria* oraz *New Zealand*.

```
$ ./main.py Australia Austria 'New Zealand'
Z pewnością 36.85% niepasujące słowo to "Austria".
```

Kolejny przykład ilustruje nam potencjalną możliwość zastosowania algorytmu w programie opisanym we wprowadzeniu do niniejszego rozdziału (s. 28).

```
$ ./main.py Volvo Renault Rower Rover Volkswagen
Z pewnością 98.81% niepasujące słowo to "Rower".
```

Widzimy, że jeśli użytkownik pomyliłby się we wprowadzaniu wyrazu *Rover* w kontekście marek samochodów, program z bardzo dużą pewnością byłby w stanie wykryć ten błąd. Podobna sytuacja ma miejsce w następnym przykładzie — tym razem przy pomyłce w czasie wpisywania słowa *kaczka*.

```
$ ./main.py krowa owca gęś taczka kaczka osioł
Z pewnością 60.48% niepasujące słowo to "taczka".
```

Pokażemy również rozróżnianie imion od innych wyrazów.

```
$ ./main.py Jan Romek Domek Adam Grzesiek
Z pewnością 21.85% niepasujące słowo to "Domek".
$ ./main.py Janek Kichał Michał Adam Grzesiek
Z pewnością 58.77% niepasujące słowo to "Kichał".
```

W ostatnim przykładzie program odróżnia *esperanto* od języków programowania.

```
$ ./main.py java perl python ruby cobol esperanto
Z pewnością 47.90% niepasujące słowo to "esperanto".
```

3.6. Kod programu

Zamieszczamy tutaj pełen kod programu. Szczegółowe informacje na temat funkcji i klas bibliotecznych użytych w aplikacji zawarte są w [10].

Moduł `se` zawarty w pliku `se.py` zawiera klasy związane z obsługą wyszukiwarek i buforowania wyników z nich uzyskanych.

Plik se.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# $Id: se.py 127 2008-06-15 15:10:41Z mikep $

import enc
import xml.etree.ElementTree as ElementTree
import re
import anydbm
import urllib2
from urllib import quote

def getEngineInfo(engineName, configFile='engines.xml'):
  "Pobiera informacje o przeglądarce z pliku konfiguracyjnego."
  xmlTree = ElementTree.fromstring(open(configFile).read())
  xmlEngineNode = [ node for node in xmlTree.getchildren() \
      if node.attrib['name'] == engineName ][0]
  url = xmlEngineNode.find('url').text
  regex = xmlEngineNode.find('regex').text
  logN = float(xmlEngineNode.find('logn').text)
  return url, regex, logN

class GetNumber:
  "Klasa zwracająca liczbę stron ze słowem/słowami."
  def __init__(self, engineName):
    self.cache = Cache(engineName)
    url, regex, self.logN = getEngineInfo(engineName)
    self.checker = WebChecker(url, regex)
  def __call__(self, *words):
    if words in self.cache:
```

```
30        return self.cache[words]
31      else:
32        number = self.checker(*words)
33        self.cache[words] = number
34        return number
35    def getLogN(self):
36      "Zwraca log N specyficzne dla wybranej przeglądarki."
37      return self.logN
38
39  class Cache:
40    "Klasa - Bufor dla liczby stron. Używa plikowej bazy np. BerkeleyDB."
41    def __init__(self, engineName):
42      self.db = anydbm.open('cache_%s.db' % engineName, 'c')
43    def __del__(self):
44      self.db.close()
45    def __makeKey(self, words):
46      return (':'.join(sorted(words))).encode('utf-8')
47    def __contains__(self, words):
48      key = self.__makeKey(words)
49      return key in self.db
50    def __getitem__(self, words):
51      key = self.__makeKey(words)
52      return int(self.db[key])
53    def __setitem__(self, words, number):
54      key = self.__makeKey(words)
55      self.db[key] = str(number)
56
57  class WebChecker:
58    "Klasa sprawdzająca liczbę stron w wyszukiwarce."
59    def __init__(self, url, regex):
60      self.url = url
61      self.regex = re.compile(regex)
62    def __call__(self, *words):
63      "Zwraca liczbę stron zawierających słowo/słowa."
64      html = self.__getHtml(*words)
65      return int(self.__parsed(html))
66    def __getHtml(self, *words):
67      query = '+'.join([quote(('+"%s"' % word).encode('utf-8')) \
68          for word in words])
69      url = self.url % query
70      request = urllib2.Request( url, \
71          headers={'User-Agent':'Normalized Web Distance Computer'} )
72      return urllib2.urlopen(request).read()
73    def __parsed(self, html):
74      try:
```

```
75      foundStr = self.regex.search(html).groupdict()['number']
76      return ''.join([char for char in foundStr if char.isdigit()])
77    except:
78      return '1'
```

Koniec pliku se.py

W pliku `nwd.py` zawarta jest klasa `NwdComputer` oraz funkcja `inadequate`, odpowiedzialne m.in. za utworzenie grafu i znalezienie niepasującego słowa.

Plik nwd.py

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3 # $Id: nwd.py 130 2008-06-17 20:46:52Z mikep $
4
5 from se import GetNumber
6 from threading import Thread
7 from math import log as logE
8
9 log = lambda x: logE(x, 2.0)
10
11 class NwdComputer:
12   "Klasa obliczająca odległości kontekstowe."
13   def __init__(self, engineName):
14     self.counter = GetNumber(engineName)
15     self.logN = self.counter.getLogN()
16     self.checked = {}
17   def __call__(self, word1, word2):
18     "Oblicza odległość między word1 a word2."
19     graph = self.makeGraph([word1, word2])
20     return graph.values()[0]
21   def __check(self, *words):
22     self.checked[frozenset(words)] = self.counter(*words)
23   def __checkAll(self, words):
24     threads = [ Thread(target=self.__check, args=(word,) ) \
25         for word in words ]
26     for i in range(len(words)):
27       for j in range(i):
28         threads.append(Thread(target=self.__check, \
29             args=(words[i], words[j])))
30     for thread in threads:
31       thread.start()
32     for thread in threads:
33       thread.join()
34   def makeGraph(self, words):
35     '''Zwraca graf pełny między słowami z words;
```

```
36      długości krawędzi to odległości kontekstowe.'''
37     self.__checkAll(words)
38     result = {}
39     for i in range(len(words)):
40       for j in range(i):
41         key = frozenset((words[i],words[j]))
42         word1Count = self.checked[frozenset((words[i],))]
43         word2Count = self.checked[frozenset((words[j],))]
44         word12Count = self.checked[key]
45         result[key] = (log(max(word1Count, word2Count))-log(word12Count)) \
46             / (self.logN - log( min(word1Count, word2Count)))
47     return result
48
49 scaleFunction = lambda x: -x*x + 2.0*x
50
51 def inadequate(graph):
52   '''Wybiera z grafu wierzchołek-słowo,
53   które jest najbardziej oddalone od reszty.
54   Zwraca słowo oraz pewność, z jaką zostało ono wybrane.'''
55   weights = {}
56   for edge, value in graph.items():
57     for vertex in edge:
58       if vertex in weights:
59         weights[vertex].append(value)
60       else:
61         weights[vertex] = [value]
62   minimalWeights = []
63   for vertex in weights.keys():
64     minimalWeights.append({'word':vertex, \
65       'value':reduce(min, weights[vertex])})
66   minimalWeights.sort(key=lambda x: x['value'], reverse=True)
67   chosen = minimalWeights[0]
68   second = minimalWeights[1]
69   sureness = scaleFunction(1.0 - second['value']/chosen['value'])
70   return chosen['word'], sureness
```

Koniec pliku nwd.py

Poniżej znajduje się tekst programu ramowego, o którym wspominaliśmy w podrozdziale 3.4.

Plik main.py

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3 # $Id: main.py 132 2008-06-18 14:19:38Z mikep $
4
```

```
5 import sys
6 import enc
7 import nwd
8
9 engine = nwd.NwdComputer('yahoo')
10
11 words = [unicode(arg) for arg in sys.argv[1:]]
12 wordCount = len(words)
13 if wordCount == 0:
14   print u'Proszę podać jako argument(y) programu:'
15   print u'1 słowo - aby poznać liczbę stron, które je zawierają,'
16   print u'2 słowa - aby poznać odległość kontekstową między nimi,'
17   print u'więcej słów - aby dowiedzieć się, które nie pasuje do reszty.'
18 elif wordCount == 1:
19   word = words[0]
20   count = engine.counter(word)
21   print u'Liczba stron ze słowem "%s" wynosi %d.' % (word, count)
22 elif wordCount == 2:
23   word1, word2 = words
24   distance = engine(word1, word2)
25   print u'Odległość między słowem "%s" a "%s" wynosi %.4f.' \
26       % (word1, word2, distance)
27 else:
28   inadequateWord, sureness = nwd.inadequate(engine.makeGraph(words))
29   print u'Z pewnością %.2f%% niepasujące słowo to "%s".' \
30       % (100.0 * sureness, inadequateWord)
```

Koniec pliku main.py

Moduł `enc.py` ma na celu uniezależnić działanie aplikacji od systemu operacyjnego a w szczególności kodowania znaków diakrytycznych w konsoli systemowej. Dzięki niemu program działa poprawnie (został przetestowany) bez żadnych modyfikacji w systemie GNU/Linux, FreeBSD oraz Microsoft Windows 2000 i Vista.

Plik enc.py

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3 # $Id: enc.py 128 2008-06-15 19:30:05Z mikep $
4
5 import sys, locale
6 reload(sys)
7 sys.setdefaultencoding(locale.getpreferredencoding())
```

Koniec pliku enc.py

Ostatni prezentowany plik zawiera konfigurację używanych przez nas wyszukiwarek internetowych. Ze względu na długość linii tekstu, niektóre z nich musimy złamać — oznaczamy to symbolem ◀.

Plik engines.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- $Id: engines.xml 127 2008-06-15 15:10:41Z mikep $ -->
<engines>
  <engine name="yahoo">
    <url>http://search.yahoo.com/search?p=%s&amp;y=Search&amp;ei=UTF-8</url>
    <regex>&lt;span id="infotext"&gt;&lt;p&gt;\d+ - \d+ of( about)? (?P&lt◀
;number&gt;[\d,]*) for &lt;strong&gt;</regex>
    <logn>50</logn>
  </engine>
  <engine name="google">
    <url>http://www.google.com/search?rls=en&amp;q=%s&amp;ie=utf-8&amp;oe=◀
utf-8</url>
    <regex>&lt;font size=-1&gt;Results &lt;b&gt;1&lt;/b&gt; - &lt;b&gt;\d+◀
&lt;/b&gt; of (about )?&lt;b&gt;(?P&lt;number&gt;[\d,]+)&lt;/b&gt; for &lt◀
;b&gt;</regex>
    <logn>50</logn>
  </engine>
  <engine name="msn">
    <url>http://search.live.com/results.aspx?q=%s</url>
    <regex>&lt;span class="sb_count" id="count"&gt;\d+.*?\d+ .*? (?P&lt;nu◀
mber&gt;[^ ]*?) .*?&lt;/span&gt;</regex>
    <logn>50</logn>
  </engine>
</engines>
```

Koniec pliku engines.xml

Spis rysunków

Bibliografia

[1] Charles Bennet, Péter Gács, Ming Li, Paul Vitányi, Wojciech Żurek, *Information distance*, IEEE transactions on information theory, vol. 44, no 4, July 1998

[2] Gregory Chaitin, *Exploring randomness*, Springer Verlag, London 2001

[3] Xin Chen, Brent Francia, Ming Li, Brian McKinnon, Amit Seker, *Shared information and program plagiarism detection*, IEEE transactions on information theory, vol. 50, no 7, July 2004, 1545–1551

[4] Rudi Cilibrasi, Paul Vitányi, *Clustering by compression*, IEEE transactions on information theory, vol. 51, no 3, April 2005, 1523–1545

[5] Rudi Cilibrasi, Paul Vitányi, *The Google similarity distance*, IEEE transactions on knowledge and data engineering, vol. 19, no 3, March 2007, 370–383

[6] Ming Li, Paul Vitányi, *An introduction to Kolmogorov complexity and its applications*, Springer Verlag, New York, 2nd Edition, 1997

[7] Jerzy Mycka, *Teoria funkcji rekurencyjnych*, Wrzesień 2000

[8] Christos Papadimitriou, *Złożoność obliczeniowa*, Wydawnictwa Naukowo-Techniczne, Warszawa 2002

[9] Mark Pilgrim, *Dive into Python*, APress, 2004

[10] Python Documentation, `http://www.python.org/doc/`

[11] Claude Shannon, *A mathematical theory of communication*, The Bell system technical journal, vol. 27, July, October 1948, 379–423, 623–656