Kompresja słów z użyciem dynamicznego słownika i algorytmu optymalnego parsowania

Michał Stobierski

Czerwiec 2020

1 Wprowadzenie

Istnieje wiele technik i algorytmów służących do kompresji tekstu. Jednymi z najpopularniejszych, a zarazem prostymi przykładami są te autorstwa Lempela i Ziva: LZ78 oraz jego późniejsze rozszerzenie LZW. Oba oparte są na konstruowaniu w trakcie czytania słowa tzw. słownika fraz (dalej oznaczanego przez D) wraz z odpowiadającymi im kodami (w angielskiej literaturze znane szerzej jako *phrases* i *codes*), wykorzystywanego przy parsowaniu powtarzających się fragmentów.

Najczęściej spotykana (oryginalna) implementacja powyższych algorytmów polega w skrócie na utrzymywaniu i wydłużaniu obecnie przetwarzanego fragmentu tekstu, dopóki znajduje się on w słowniku. W momencie, gdy fragment wydłużony o następny przeczytany znak przestaje należeć do słownika, na wyjście wypisywany jest kod fragmentu bez tego znaku i dodawana jest nowa fraza. Szczegółowy opis algorytmów można znaleźć bez problemu w sieci. Różnią się one głównie zasadą dodawania fraz i ich kodów do słownika.

Istotne spostrzeżenie dotyczące tych metod jest takie, że w gruncie rzeczy są to algorytmy zawierające podejście zachłanne. W danym momencie kandydatem na kod do wypisania jest ten odpowiadający najdłuższej frazie ze słownika zgodnej z aktualnie parsowanym fragmentem. Takie rozwiązanie w praktyce okazało się mieć zaskakująco dobre rezultaty i znalazło zastosowanie m.in. w UNIX-owych programach jak *gzip* czy formacie kompresji obrazów $.gif$.

Okazuje się jednak, że potrafi być ono dalekie od optymalnego z punktu widzenia liczby wypisanych kodów składających się na skompresowane wyjście (co, przy założeniu, że **kody są jednakowej długości**, bezpośrednio przekłada się na współczynnik kompresji). Dokładnie 20 lat po opracowaniu LZ78, Matias i Şahinalp [MS98] wskazali optymalną metodę wykorzystania słownika, udowodnili jej poprawność i zaproponowali efektywną implementację. Metoda ta jest uniwersalna nie tylko dla algorytmów LZ, ale dla całej klasy kompresorów, których słowniki spełniają pewne konkretne własności.

W poniższym tekście przedstawiamy ideę wzorcowego rozwiązania, szkic dowodu poprawności oraz omówienie efektywnej jego implementacji.

2 Opis kompresji

Wspomnianą, kluczową własnością algorytmów kompresji, dla której uzyskamy nasz pożądany, optymalny wynik jest tzw. *prefiksowość* słownika (ang. *prefix property*):

Definicja 1 (Prefiksowość)**.** *Algorytm kompresji A spełnia kryterium prefiksowości, gdy w każdym momencie wykonania algorytmu spełniony jest warunek: jeżeli słowo x należy do słownika D, to również wszystkie prefiksy x znajdują się w D.*

Okazuje się, że własność tę spełnia wiele algorytmów opartych o dynamiczną konstrukcję słownika, w tym wspomniane LZ78 i LZW (dowód wynika z konstrukcji). Bez straty ogólności skupimy się na tym ostatnim.

2.1 Kompresja

Zasadniczo podstawowym krokiem algorytmu kompresji tekstu jest przeczytanie i analiza pojedynczego znaku. To na jej podstawie aktualizowany jest wewnętrzny stan (w naszym przypadku stan słownika) i zawartość wyjścia. Przyjrzyjmy się zatem temu, co w danej iteracji może się wydarzyć. Możemy wyszczególnić dwie potencjalne akcje (mogące oczywiście wystąpić jednocześnie):

- do słownika zostanie dodana nowa fraza
- na wyjście zostanie wypisany kod pewnej frazy (stanowiącej prefiks jeszcze nieskompresowanego fragmentu tekstu)

Wygodnie będzie myśleć o nich jako o dwóch (częściowo!) niezależnych elementach.
I tak, niech P_d oznacza tzw. *parser słownika*, a P_o *parser wyjścia*. Intuicyjnie, k-ty krok kompresji będzie miał następującą postać: (1) Wczytaj t_k (2) Przekaż t_k do P_d, który kontroluje stan słownika D. (3) Przekaż t_k do P_o, który na podstawie aktualnego stanu D i t_k może wypisać na wyjście $code(x)$, dla pewnej frazy x z D.

Szczegółowy opis modelu znajduje się w pracy [MS98]. Dzięki jego wprowadzeniu wyraźnie nakreślona zostaje granica między tym, co jest ustaloną cechą danego algorytmu, a tym, co chcemy poprawić. Działanie P_d jest cechą danego kompresora (inne dla LZ78, inne dla LZW etc.), P_o natomiast (najczęściej realizowane jako wspomniany już algorytm zachłanny) możemy modyfikować. Co więcej, jak zauważają autorzy, model ten jest na tyle elastyczny, że nie tylko obejmuje szeroką gamę algorytmów kompresji, ale pozwala również na swobodne "łączenie" P_d i P_o w pary i upraszcza implementację kompresora w wersji *online*.

2.2 Dekompresja

W niniejszym tekście nie będziemy poświęcać wiele miejsca dekompresji, gdyż z algorytmicznego punktu widzenia jest ona taka podobna dla wszystkich algorytmów w klasie "prefiksowych". To, co tu być może warto podkreślić to fakt, że dekompresja odbywa się jedynie z pomocą P_d (zdefiniowanego identycznie jak w kompresorze), nie ma zatem znaczenia w jaki sposób parsowaliśmy wyjście kompresora (wybór P_o).

3 Algorytm i dowód poprawności

W ramach wprowadzenia do zasadniczej części przekonajmy się, że rzeczywiście P_o realizowany w postaci algorytmu zachłannego może dać wyniki dalekie od oczekiwanych:

Twierdzenie 1 (Nieoptymalność zachłannego parsera [MS99])**.** *Zachłanny parser wyjścia w każdej iteracji algorytmu wybiera najdalej sięgający prefiks nieskompresowanej części tekstu. Istnieją słowa, które używając słownika D z alg. LZW można zakodować przy użyciu $O(l)$ fraz, a dla których zachłanny P_o używa $O(l^{3/2})$ fraz.*

Szkic dowodu. Niech k będzie l. pierwszą. Zdefiniujmy $\Sigma = \{0, 1, ..., k + \sqrt{k}\}$, $R = 1, ..., k$, $R_i = R^i$, $S = 1, 2, 1, 2, 3, ..., 1, 2, ..., k$. Za słowo do skompresowania weźmy

$$T = 0, S, k + 1, 0, R_1, 1, k + 2, 0, R_2, 1, ..., k + \sqrt{k}, 0, R_{\sqrt{k-1}}, 1$$

Pozostaje przeanalizować, co zrobi zachłanny parser, a co można uzyskać w sprytniejszy sposób (np. podział na $O(k)$ fraz). □

Teraz pora na najważniejszą część niniejszego rozważania. Okazuje się, że **jeżeli zamiast wybierania prefiksu, który w danej iteracji pokryje najwięcej tekstu wybierzemy taki, który spowoduje największy postęp w następnej iteracji, to uzyskamy algorytm optymalny (oznaczmy go FP)**. Stwierdzenie to formalizuje poniższe twierdzenie:

Twierdzenie 2 (Optymalność FP [MS99])**.** *Dla jakiegokolwiek P_d które tworzy słownik D mający własność prefiksowości przez cały czas trwania algorytmu, FP generuje skompresowany ciąg o najmniejszej liczbie użytych kodów (fraz), bez względu na dobór tekstu T.*

Dowód. Niech P_d będzie takie jak w wypowiedzi. Niech C i C' to będą kompresory używające kolejno $P_o = FP$ i jakiegokolwiek inny sposób parsowania wyjścia. Chcemy pokazać, że

$$\#output(C) \leq \#output(C')$$

Niech i-ta fraza wypisana przez C kończy się na pozycji $E(i)$. Niech najdalej sięgająca fraza w D będąca prefiksem $T[E(i) + 1 : n]$ kończy się na pozycji $L(i)$. Analogicznie definiujemy $E'(i)$ oraz $L'(i)$ dla C'. Jeżeli pokażemy, że $\forall_i\ L(i) \geq L'(i)$ to otrzymujemy tezę. Dowód przebiega indukcyjnie:
Baza: jest trywialnie spełniona, bo C i C' startują z tej samej pozycji.
Krok indukcyjny: zauważmy, że $t_0 = T[E'(i) + 1 : L'(i)]$ jest frazą w D (z definicji L') i zachodzi:

$$E'(i) \overset{\text{(z def. L')}}{\leq} L'(i-1) \overset{\text{(z zał. ind.)}}{\leq} L(i-1)$$

Przypadek 1 $(E'(i) > E(i-1))$**.** *Wtedy, z prefiksowości D i nierówności powyżej wynika, że $t_1 = T[E(i-1) + 1 : E'(i)]$ również należy do D. Zgodnie z zasadą działania P_d słowo $t_1 t_0$ musiało być rozpatrzone przy wyborze $E(i)$. Zatem $L(i) \geq L'(i)$.*

Przypadek 2 $(E'(i) \leq E(i-1))$**.** *tr.*

□

Widzimy więc, że również zachłanna, z pozoru naiwna modyfikacja pierwotnego podejścia skutkuje algorytmem dającym teoretyczne gwarancje (można by myśleć, że patrzenie o 2 i więcej kroków w przód polepszy wynik, tak się jednak nie dzieje)!
W literaturze FP można spotkać również pod nazwą *FlexibleParsing* lub *semi-greedy parsing*.

4 Implementacja

Skoro mamy już pomysł na efektywne parsowanie wyjścia, pozostaje jedynie zastanowić się nad implementacją, która sprawi, że znajdzie on zastosowanie w praktyce.
Nieoczywiste jest, jak możemy szybko wyznaczyć prefiks, którego wybór poskutkuje największym progresem przy następnym wypisywaniu wyjścia. Wzorcowy pomysł realizacji $P_o = FP$ polega na parsowaniu kolejnych t_k dopóki $T[E(i-1)+1 : k]$ należy do D. Dojdziemy w ten sposób do $T[a : b] = T[E(i-1)+1 : L(i-1)]$. W tym momencie wiemy już, że $E(i)$ leży między a i b. Chcemy więc wybrać takie $E(i)$, że $L(i)$ będzie możliwie największe. Prosta obserwacja: kandydatami na takie $E(i)$ są tylko te pozycje, dla których $T[E(i)+1 : b]$ należy do D.
Próbujmy zatem znaleźć kolejnych kandydatów. "Ucinajmy" po literze od przodu $T[a : b]$ (uzyskując $T[a+1 : b]$ etc.) dopóki nie trafimy na takie a', że $T[a' : b]$ jest w D. Wtedy przechodzimy do drugiej fazy, która polega na wydłużaniu $T[a' : b]$ (poprzez parsowanie kolejnych t_k) ponownie dopóki fraza należy do D. Uzyskane tak podsłowo $T[a' : b']$ ponownie "ucinamy" od przodu i powtarzamy cały proces. Robimy tak dopóki nie wyjdziemy z a' poza $T[a : b]$, czyli dopóki $a' < b$. Nietrudno dowieść, że ostatnie napotkane, prawidłowe a' jest poszukiwaną wartością $E(i)+1$.

No dobrze, ale jak szybko sprawdzać, czy $T[a+x : b]$ jest w D oraz rozszerzać frazy? Drzewa sufiksowe mogą być niezłym pomysłem, użyjemy jednak pary drzew *Trie*: drzewo T będzie utrzymywać po prostu D, a T^r wszystkie słowa z D, ale odwrócone. Ponadto, każda fraza drzewa D będzie posiadać link do odpowiadającej frazy odwrotnej w T^r (i w drugą stronę). W ten sposób, operacje *insert* i *search* odbywają się jak w normalnym *Trie* (uwaga: *search* pamięta swój ostatni stan, aby po dodaniu jednej litery nie iść znowu od korzenia po całym istniejącym prefiksie), natomiast dodatkowa operacja *contract* bierze węzeł aktualnej frazy, "skacze" po linku do T^r, wychodzi w górę o jeden węzeł i "wraca" po linku do drzewa T. W ten sposób, w czasie $O(1)$ "ucina" pierwszy znak frazy.

Teraz zauważamy, że *contract* wykona się maksymalnie raz na każdy znak tekstu, tak samo *search*, suma operacji *insert* zaś sumarycznie nie przekracza $|T|$, zatem sumarycznie nasz algorytm działa w amortyzowanym czasie $O(|T|)$ - optymalnie!

Jeżeli istotna jest również używana pamięć, to gdy zamiast zwykłych *Trie* użyjemy *skompresowanych Trie* i odpowiednio zadbamy o utrzymywanie etykiet krawędzi w T^r to złożoność pamięciowa poprawi się do optymalnego $O(|D|)$.

Oto pełna analiza techniki *optimal parsing* dla kompresorów z dynamicznym słownikiem. Dla przypomnienia, powyższe rozumowanie jest dla algorytmu LZW, w szczególności indeksy dla pozostałych algorytmów mogą się różnić, własności jednak pozostają te same.

Bibliografia

[MS98] Yossi Matias and Suleyman Cenk Sahinalp. *On the Optimality of Parsing in Dynamic Dictionary Based Data Compression*. 1998.

[MS99] Yossi Matias and Süleyman Cenk Sahinalp. "On the Optimality of Parsing in Dynamic Dictionary Based Data Compression". In: *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, 17-19 January 1999, Baltimore, Maryland, USA*. Ed. by Robert Endre Tarjan and Tandy J. Warnow. ACM/SIAM, 1999, pp. 943–944. URL: `http://dl.acm.org/citation.cfm?id=314500.314935`.