Raport: Kodowanie gramatykowe Sequitur

Marcin Witkowski

20 czerwca 2020

Wstęp

Tematem projektu była implementacja algorytmu kodowanie gramatykowego *Sequitur*, działającego w czasie liniowym. Algorytm *Sequitur* jest kodowaniem gramatykowym, tj. dla konkretnych danych wejściowych wyznacza *gramatykę bezkontekstową*, która jednoznacznie definiuje dane wejściowe. Dodatkowo muszą być zachowane dwa niezmienniki: unikalność diagramów oraz użyteczność produkcji. Tak stworzoną gramatykę następnie kompresujemy używając jakiegoś kodowania entropijnego jak np. kodowanie arytmetyczne bądź Huffmana. W mojej implementacji użyłem tego drugiego.

Instrukcja obsługi

> Wersją kompilatora, z której korzystałem podczas pisania projektu była wersja nightly z dnia `2020-01-06`. Jeżeli z jakiegoś powodu najnowsza wersja nie działa, należy zainstalować wersję kompilatora w tej właśnie wersji przy pomocy polecenia `$ cargo install nightly-2020-01-06`.

Do skompilowania programu potrzebny jest kompilator języka Rust. Ten najłatwiej pozyskać używając narzędzia **rustup**, a następnie zainstalować środowisko w wersji nightly przy pomocy polecenia `$ rustup install nightly`. Następnie aby skompilować projekt, wystarczy wydać polecenie `$ cargo +nightly build --release`, plik wykonywalny pojawi się pod nazwą `sequitur` w katalogu `target/release/`.

Wykonanie polecenia `$ sequitur` bez żadnych argumentów wyświetli krótki opis jak korzystać z aplikacji. Aby skompresować plik `foo` pod nazwą `bar` musimy wykonać polecenie `$ sequitur foo bar`. Możemy także wydać polecenie `$ sequitur foo`, wtedy plik zostanie skompresowany pod nazwą `foo.seq`. Aby natomiast zdekompresować plik o nazwie `bar` do pliku `foo` musimy wywołać polecenie `$ sequitur -d bar foo`. Ponownie, jeżeli pominiemy nazwę pliku wyjściowego i wykonamy `$ sequitur -d bar`, plik zostanie zdekompresowany do pliku o nazwie `bar.raw`.

Aby uruchomić testy jednostkowe, należy wykonać komendę `$ cargo +nightly test --release`. Wszystkie testy znajdują się w pliku `src/sequitur/tests.rs`. Aby uruchomić benchmarki, należy wykonać polecenie `$ cargo +nightly bench`. Kod benchmarków można podejrzeć w katalogu `benches`. Wyniki benchmarków pojawia się natomiast w katalogu `target/criterion`.

Wykorzystane biblioteki

> Wszystkie biblioteki można znaleźć w pliku `Cargo.toml`, a ich źródła na stronie `https://cargo.io/`

Po pierwsze, aby nie mieć problemów ze wskaźnikami (o czym będzie w kolejnym rozdziale), skorzystałem z bibliotek `generational-arena` oraz `slot-map`. Do łatwego przechowywania tablic bitowych po kompresji Huffmana, skorzystałem z biblioteki `bitvec`. Do samej kompresji Huffmana skorzystałem natomiast z biblioteki `huffman-compress`. Jako że potrzebowałem jakieś metody serializacji danych po ich zakodowaniu, skorzystałem z biblioteki do serializacji binarnej `bincode`. Ostatecznie, jako że język Rust jest silnie typowanym językiem bez mechanizmu wyjątków, skorzystałem z bibliotek `thiserror` oraz `anyhow` aby w komfortowy sposób móc zwracać błędy gdy *coś pójdzie nie tak*.

Z bibliotek, które nie są ściśle powiązane z implementacją algorytmu Sequitur, użyłem dodatkowo biblioteki `lipsum` pozwalającej na wygenerowanie *Lorem ipsum* dowolnej długości przy pomocy łańcuchów Markova. Poza tym, użyłem także "narzędzia" `criterion`, pozwalającego na wykonanie w łatwy sposób testów wydajnościowych.

Napotkane problemy

Pierwszy z napotkanych przez mnie problemów wynikał z faktu, że w języku `Rust` dosyć trudno jest napisać struktury oparte o pointerach. Wynika to bezpośrednio z faktu, iż kompilator nie pozwala na posiadanie *mutowalnej* i *niemutowalnej* referencji do tego samego obiektu w tym samym czasie. Z tego względu skorzystałem z bibliotek, które implementują odpowiednio *arena allocator* i *slab allocator*. Dzięki temu, nie musiałem już operować na pointerach, a na indeksach obiektów, które należą do jakiejś areny. Takie podejście jest wystarczająco szybkie do zaimplementowania listy dwukierunkowej, jako że algorytm Sequitur nie potrzebuje nigdy łączyć dwóch list dwukierunkowych.

Kolejnym problemem było kodowanie `NMW`. W opisie z wykładu przy drugim pojawieniu się *nieterminala* kodowanie powinno wyemitować krotkę $\langle \text{offset}, \text{length} \rangle$. To rozwiązanie nie chciało mi jednak działać, więc ostatecznie rozwiązałem je przez kodowanie krotki $\langle \text{prod\_index}, \text{length} \rangle$. Poza tym drobnym niuansem, reszta kodowania wygląda tak samo.

Innym problemem była ostateczna kompresja entropijna gramatyki przy pomocy kodowania Huffmana. Okazuje się, że trudno znaleźć jakąś porządną implementacje kodowania Huffmana, a w szczególności w jego adaptatywnej wersji. Ostatecznie, znalazłem jakąś bibliotekę dla standardowego Huffmana, którą to jednak i tak musiałem przerobić, ponieważ nie oferowała ona żadnej metody serializacji drzewa. Jako że postanowiłem kompresować tylko i wyłącznie ciągi bajtów, a zatem tylko wartości z przedziału od 0 do 255, najlepszą metodą serializacji okazało się po prostu sprowadzenie drzewa do postaci kanonicznej, a następnie zakodowanie drzewa jako ciąg 256 wartości definiujących długości kolejnych słów kodowych.

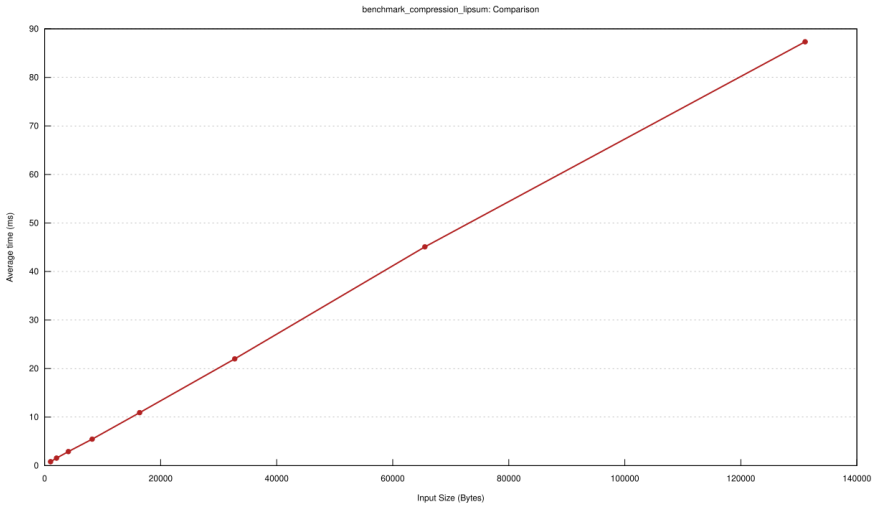
Inne problemy raczej wynikały już ściśle z drobnych potknięć w kodzie takich jak pomijanie zapisywania niektórych digramów czy też nieusuwanie starych produkcji.

Testy

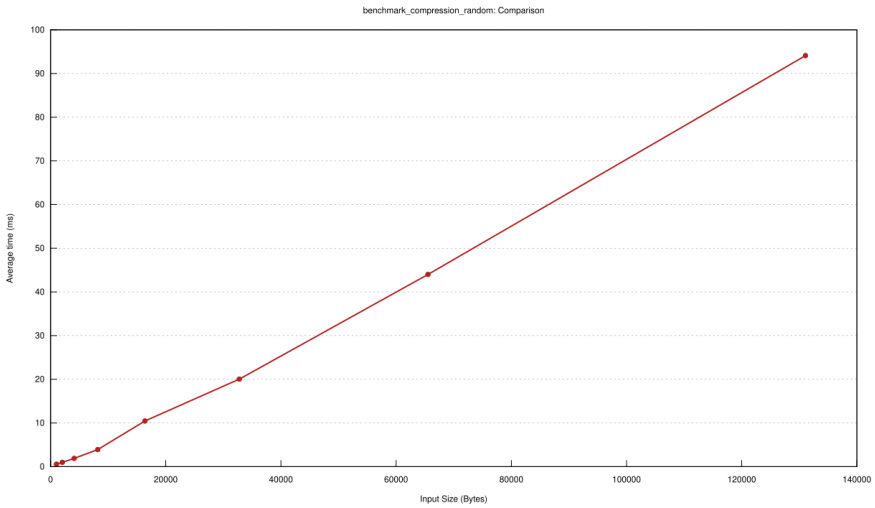
To przeprowadzenia testów wydajnościowych skorzystałem z biblioteki `criterion` dla języka Rust. Testy jakości kompresji zostały natomiast przeprowadzone ręcznie, przy użyciu danych znalezionych w internecie bądź też wygenerowanych. Lista źródeł znajduje się w pliku `SOURCES` w katalogu `test_data`.

Testy wydajnościowe

Testy wykonywałem na trzech rodzajach danych: losowych bajtach, losowych sekwencjach DNA oraz na losowo generowanych ciągach *Lorem ipsum*. Wielkości które testowałem były natomiast rzędu kolejno 1KB, 2KB, 4KB, 8KB, 16KB, 32KB, 64KB oraz 128KB.

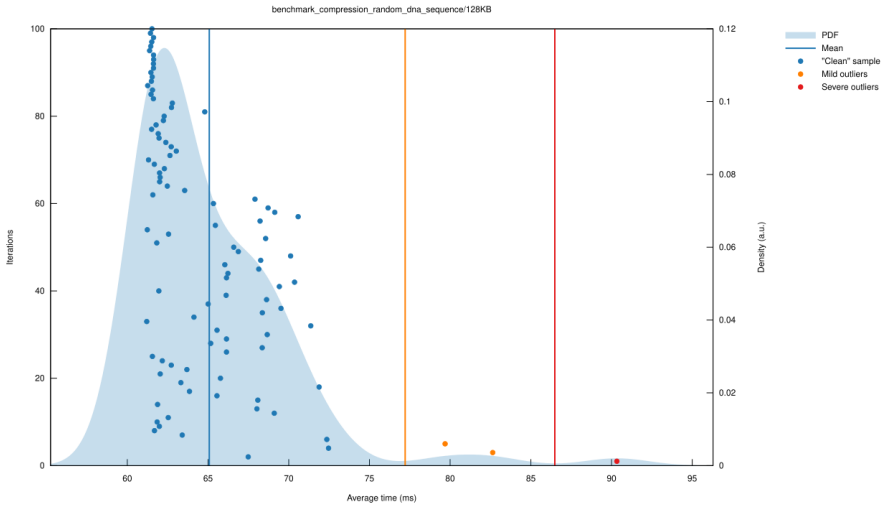


Rysunek 1: Wykres czasu od wielkości danych dla *Lorem ipsum*



Rysunek 2: Wykres czasu od wielkości danych dla danych losowych

Jak widać z powyższych wykresów, algorytm kompresji działa z grubsza w czasie liniowym od wielkości danych. Jeżeli chodzi natomiast o surową szybkość, popatrzmy na wykres średniego czasu kompresji dla losowych ciągów DNA o wielkości 128KB.



Rysunek 3: Średni czas działania algorytmu dla sekwencji DNA

Jak można zauważyć, średni czas kompresji danych tej wielkości to około 65ms. Daje to nam średnią prędkość kompresji $\approx$ 2MB/s. Więcej wykresów można zobaczyć w katalogu `charts`.

Testy jakości kompresji

Nazwa pliku	Rozmiar przed kompresją	Rozmiar po kompresji	Współczynnik kompresji $(1 - \frac{\text{po}}{\text{przed}})$
dna_sequences	49642KB	22401KB	55%
lorem_ipsum	6315KB	2404KB	61%
bible	4347KB	2191KB	50%
orders.xml	5252KB	787KB	85%
mario.mid	2745B	1376B	50%
zeros	1MB	368B	99.97%
mod.rs	20KB	11KB	48%
random	1MB	2MB	-100%

Moim głównym celem było przetestowanie algorytmu Sequitur na danych tekstowych, w szczególności takich, które posiadają jakąś wewnętrzną strukturę. Jak widać, pliki XML bardzo dobrze się kompresują – otrzymaliśmy plik o 85% mniejszy niż plik wejściowy. Dobrze się kompresuje także tekst naturalny jak i sekwencje DNA. Ponadto, algorytm skompresował swoje własne źródło

z sensownym współczynnikiem 48% tak jak było to oczekiwane. Dla danych losowych algorytm w mojej implementacji działa natomiast tragicznie, generując plik wyjściowy o dwukrotnym rozmiarze pliku wyjściowego. Prawdopodobnie główną część tej pamięci "zjada" serializator tokenów i wymieniając go, można by to minimalnie poprawić, aczkolwiek wielkość pliku wyjściowego cały czas byłaby pewnie większa od wejściowego, jako że są to trudne dane dla *każdego* algorytmu kompresji. Miłym zaskoczeniem jest fakt, że algorytm Sequitur całkiem sensownie kompresuje niektóre pliki MIDI: tu w przypadku tematu muzycznego Mario, udało się zmniejszyć rozmiar pliku o połowę.

Wnioski

Kompresowanie danych algorytmem Sequitur okazuje się *bardzo słabym* pomysłem. Dobrym pomysłem jest natomiast próba kompresowania plików XML. Jeżeli mamy dużo powtarzających się danych, jak na przykład listę wykonanych transakcji, możemy osiągnąć naprawdę fajne stopnie kompresji. Kompresowanie ciągów sekwencji DNA ma także sporo sensu – w tym konkretnym przypadku ograniczamy się tak naprawdę tylko do czterech wartości reprezentujących cztery nukleotydy. Z tego powodu w danym ciągu może powstać sporo powtórzeń, które dalej zostaną zastąpione przez stworzone gramatyki. Tekst naturalny także jest dobrą rzeczą do kompresji – po zmniejszeniu otrzymamy pliki średnio 50% mniejsze.