Raport z projektu LZ77 i LZSS

Łukasz Klasiński

21 czerwca 2020

Opis algorytmów.

Tematem projektu było zaimplementowanie dwóch algorytmów kompresji - LZ77 oraz LZSS. Oba są do siebie bardzo podobne, różnią się głównie sposobem reprezentacji danych wyjściowych. Polegają na zastępowaniu powtórzonych ciągów bajtów liczbami wskazującymi gdzie wcześniej wystąpił dany ciąg oraz jaką miał długość. Pod koniec wyjściowe krotki poddawane są kodowaniu Huffmana.

Implementacja

Projekt został zaimplementowany używając rust'a. Poza korzystaniem ze standardowych bibliotek takich jak np. hashmap, użyłem następujących gotowych bibliotek:

- bit-vec Pozwala zapisywać dane jako ciąg pojedyńczych bajtów - potrzebne do zakodowania danych w huffmanie
- bincode Biblioteka umożliwiająca przyjemną serializację danych - dzięki niej mogłem łatwo zapisywać oraz odczytywać odpowiednie informacje z pliku takie jak same dane do kompresji/dekompresji, rozmiar słownika użyty przy kodowaniu, oryginalna nazwa pliku itp.
- thiserror Biblioteka, która pozwala na proste tworzenie własnych typów błędów.
- getopts Pozwala na proste parsowanie inputu użytkownika w celu przekazywania parametrów do programu.
- huffman-compress Biblioteka wykonująca kodowanie Huffmana na podanych bajtach.
- fnv Biblioteka udostępniająca mapę hashującą szybszą od standardowej (dla krótkich ciągów).

Wkład własny

Implementację obu algorytmów napisałem od początku. Algorytmy mają modyfikowalne (podawane poprzez konsolę) wielości buforów wejścia oraz słownikowych. Bufory na dane zostały zaimplementowane jako vektory stałych rozmiarów, które oglądane są cyklicznie. W celu szybkiego przeszukiwania, czy istnieje match dla danych z bufora wejściowego, zostało zaimplementowane spamiętywanie miejsc występowania stringów w buforze słownikowych podczas wkładania do niego danych oraz usuwania ich kiedy dane są zastępowane nowymi. Dane te przechowywane są w hashmapie, więc podczas przeszukiwania można w czasie stałym sprawdzić, czy w słowniku kodowania istnieje dany znak, a następnie sprawdzić gdzie jest najdłuższe zmachowanie. Dodatkowo zachowanie to jest modyfikowalne przy pomocy parametru - im większa wartość, to coraz dłuższe indeksy wystąpień stringów przechowujemy w hashmapie. Np. przy wyrazie `test` oraz parametrze ustawionym na 2, w hashmapie będą indeksy wystąpień dla stringów t, e, s, t, te, es, st. Zwiększanie parametru zmniejsza czas przeszukiwania dopasowania w buforze.

Napotkane problemy

Poza typowymi problemami implementacyjnymi, takimi jak problemy z indeksami itp, to największym problemem okazało się znalezienie dobre zaimplementowanego kodowania Huffmana. W szczególności chodzi o Huffmana adaptywnego. Byłby on bardzo dobrym połączeniem do obu algorytmów, ponieważ tworzą one jakieś krotki, które mogłyby być od razu kodowane w takim Huffmanie. W standardowym algorytmie mamy tą niedogodność, że musimy czekać na zebranie wszystkich krotek

i dopiero wtedy możemy je zakodować (musimy znać prawdopodobieństwa symboli). Niestety algorytm ten wymaga także zapamiętywania owej tablicy prawdopodobieństw w celu zdekodowania pliku, więc potrzebna była dodatkowa serializacja tych i związana z tym pogorszona kompresja.

Ponadto w wersji LZ77 algorytmu, miałem problem z co zrobić z faktem, kiedy chcemy zakodować ostatni wyraz ciągu i nie mamy dodatkowego chara, który możnaby umieścić do krotki (podobnie, kiedy czytamy ostatni znak z buffora). Ostatecznie musiałem wprowadzić specjalny znak, aby móc cokolwiek wstawić do tej krotki i być w stanie odkodować tą informację dla decodera.

Kolejnym i chyba największym problemem było zapisywanie wyjściowych krotek jako bajty. Ostatecznie zabrakło mi czasu na jakieś bardziej sensowne rozwiązanie - zabrakło mi czasu. Ostatecznie krotki zapisywane są z użyciem bajta separującego (w pojedyńczej krotce lz77 są takie 3), przez co kompresja jest znacznie gorsza. Napisałem bardziej optymalne rozwiązanie, które z nich nie korzysta, ale powodowało ono, że dla niektórych przypadków teksty poprawnie się nie dekompresowały, dlatego musiałem to zamienić s powrotem.

Instrukcja obsługi

Program oferuje interfejs poprzez linie poleceń. Oto wyjście po uruchomieniu polecenia `--help`:

```
Usage: ./lzss FILE [options]

Options:
    -o, --out NAME      set output file name
    -d, --dict DICT_BUFF_SIZE
                        set dict buffer size in KB
    -i, --input INPUT_BUFF_SIZE
                        set input buffer sizein KB
    -7, --lz77          change compression algorithm to lz77
    -m, --match MAX_SIZE
                        set max match for search opt
    -u, --decompress    set programm to decompression mode
    -h, --help          print this help menu
```

Jak widać możemy ustawić kilka parametrów które determinują zachowanie. Domyślne wartości dla wielkości buffora kodowania, wejściowego oraz matchowania to kolejno 4KB, 1KB, 20. Poza tym domyślnie używana jest wersja algorytmu lzss, jako że w testach wypada lepiej.

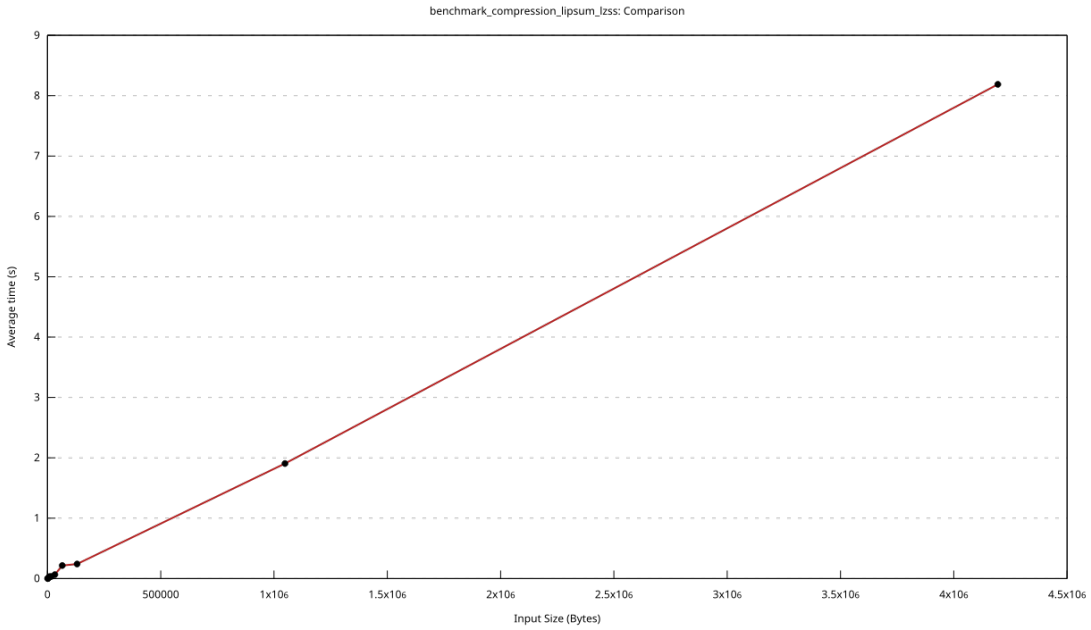
Kompilacja

Wymagane środowisko rust oraz Cargo. Wystarczy uruchomić cargo build –release. Pliki binarne będą wtedy w folderze target/release.

Przeprowadzone eksperymenty

Prędkość kompresji

Używając wbudowanego w Rusta systemu benchmarkowania programów, napisałem testy, weryfikujące prędkość kompresji w zależności od długości danych. Danymi w tym przypadku był ciąg losowego lorem ipsum zbudowanego za pomocą drzew Markova.



Na powyższym wykresie widać, że algorytm działa w czasie liniowym. Wyszło mi około 500KB/s, co nie jest tragicznym wynikiem biorąc pod uwagę sprzęt na którym były uruchamiane testy(i5 2.4GHz, niskonapięciowy). Poza tym w testowaniu wyszło, że samo zapisywanie plików zajmuje dużą część czasu - serializator bardzo długo zapisuje wszystkie dane na dysk, przez co działą to znacznie wolniej (około 2 razy).

Stopień kompresji

Jako że algorytmy z rodziny lz77 najlepiej kompresują teksty w języku naturalnym, to skupiłem się głównie na nich.

Biblia

Wielkość na wejściu - 4.2MB

buffor kodowania	lzss	lz77	stopień kompresji lzss/lz77
256	3.2MB	4.2MB	0.76/1.00
516	3.0MB	3.8MB	0.71/0.90
1024	2.9MB	3.5MB	0.69/0.83
2048	2.8MB	3.4MB	0.66/0.80
4096	2.7MB	3.1MB	0.64/0.73
8192	2.5MB	2.9MB	0.59/0.69

Lorem Ipsum

Wielkość na wejściu - 5MB

buffor kodowania	lzss	lz77	stopień kompresji lzss/lz77
256	3.9MB	5.4MB	0.78/1.08
516	3.8MB	4.9MB	0.76/0.98
1024	3.5MB	4.3MB	0.70/0.86
2048	3.2MB	3.9MB	0.64/0.78
4096	2.9MB	3.5MB	0.58/0.72
8192	2.6MB	3.2MB	0.52/0.64

Losowy ciąg

Wielkość na wejściu: 5MB

buffor kodowania	lzss	lz77	stopień kompresji lzss/lz77
256	5.0MB	8.4MB	1.00/1.65
516	5.0MB	8.5MB	1.00/1.70
1024	5.0MB	8.3MB	1.00/1.66
2048	5.0MB	8.3MB	1.00/1.56
4096	5.0MB	7.8MB	1.00/1.56
8192	5.0MB	7.2MB	1.00/1.44

Ciąg zer

Wielkość na wejściu - 4MB

buffor kodowania	lzss	lz77
256	14KB	16KB
516	16KB	16KB
1024	15KB	18KB
2048	15KB	18KB
4096	16KB	19KB
8192	16KB	20KB

Wnioski z testowania

Widać, że szczególnie kompresja za pomocą LZSS okazała się znacznie lepsze od LZ77. Myślę, że znaczącą rolę odegrał tutaj fakt, że w przypadku pojedyńczego znaku, zapisujemy znacząco mniej bitów - różnica w przypadku mojego algorytmu to około 4 bajty mniej, stąd taka różnica. Głównie dzięki temu, LZSS znacznie lepiej radzi sobie z losowymi ciągami. Widać również, że zwiększanie bufforu kodowania stopniowo polepsza kompresję. Jednocześniej jednak algorytm im większy buffor kodowania, tym działa wolniej ze względu na to, że ma więcej przeszukiwań do wykonania.