Dynamiczne kodowanie słownikowe LZSS

Realizacja trzeciego etapu projektu

Domański Piotr, Pietrowcew Jakub, Skórka Kornel

2022-01-30

Implementacja algorytmu LZSS

Projekt został zaimplementowany w języku C++. Składa się z dwóch oddzielnych programów: kodera `coder_lzss` oraz dekodera `decode_lzss`. Oba programy posiadają jedynie interfejs konsolowy i są sterowane za pomocą parametrów ich wywołania.

Koder LZSS

Wywołanie

Argumentami wywołania kodera są:

- nazwa pliku wejściowego (kodowanego)
- nazwa pliku wyjściowego
- rozmiar słownika wyrażony w bajtach
- rozmiar bufora frazy wyrażony w bajtach

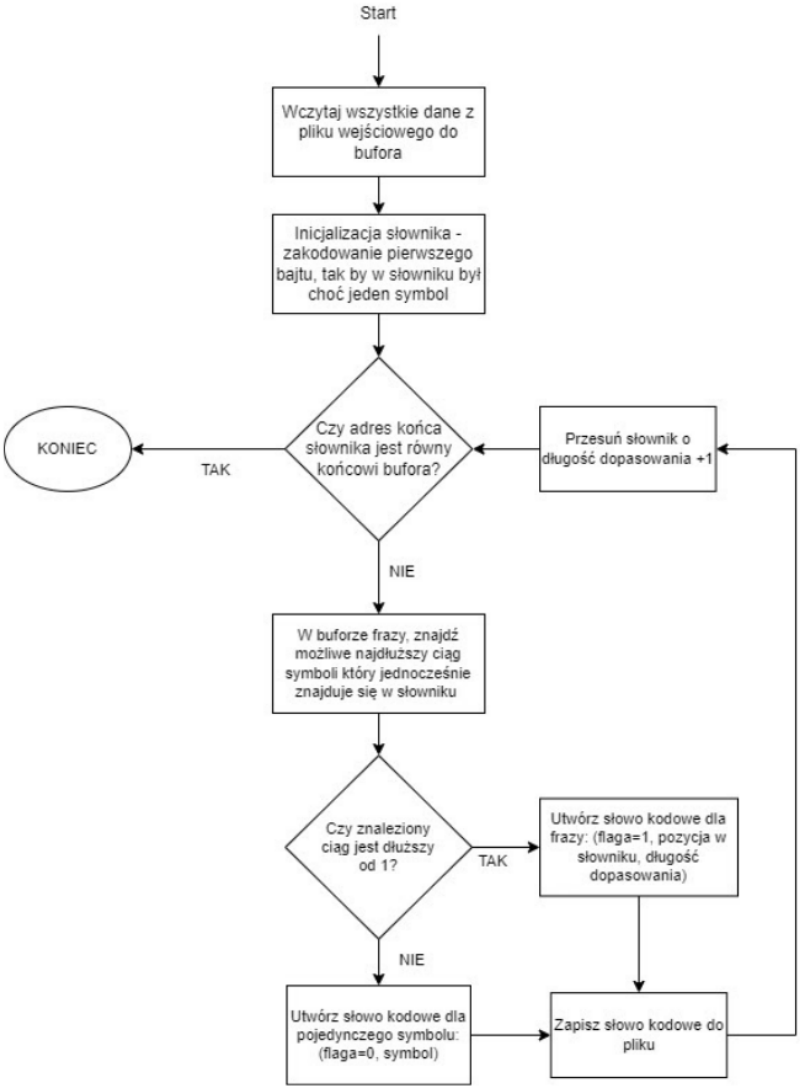
Rezultatem poprawnego wykonania programu są dwa pliki. Pierwszy plik o nazwie zgodnej z argumentem wywołania zwierające zakodowane dane. Drugi natomiast zawiera statystyki związane z procesem kodowania, a jego nazwa tworzona jest jako: nazwa_zakodowanego_stats.txt. Zawiera on poniższe dane:

- informację o wielkości słownika
- informację o wielkości bufora
- informację o czasie kodowania
- wielkość pliku przed kompresją
- wielkość pliku po kompresji
- stopień kompresji
- liczbę słów kodowych
- średnią długość bitową

Kluczowe szczegóły implementacji algorytmu kodera

Cała zawartość pliku, który ma zostać zakodowany jest na początku wczytywana do bufora “wejściowego” o rozmiarze równym wielkości pliku. Takie podejście jest wystarczające biorąc pod uwagę niewielkie rozmiary plików wejściowych oraz ilość pamięci dostępnej we współczesnych komputerach. Jednakże lepszym podejściem byłoby wczytywanie do bufora tylko części pliku wejściowego, tak aby przy rozmiarach plików wejściowych rzędu setek MB nie alokować, takich ilości pamięci.

Słownik jest implementowany jako wskaźnik, który informuje, gdzie znajduje się w danym momencie początek słownika w buforze “wejściowym”. Poza wskaźnikiem przechowujemy bieżącą długość słownika. Długość słownika jest nie większa niż maksymalna długość słownika podana jako parametr wywołania - na początku działania algorytmu, po inicjalizacji słownika jego długość wynosi jeden i rośnie do maksymalnej dopuszczalnej wartości wraz z wykonywaniem algorytmu.



Rysunek 1: Schemat działania

Słowa kodowe w postaci bitów są tworzone poprzez wykonywanie operacji bitowych (alternatywy i przesunięć) na zmiennej typu unsigned int o rozmiarze 32 bitów - odbywa się to w metodzie WriteBits klasy BitBuf. Po zapełnieniu 32 bitów zmiennej typu unsigned int jest ona umieszczana w buforze _puiBuf będącym polem prywatnym klasy BitBuf. Bufor ten jest cyklicznie zapisywany do pliku wyjściowego. Pewnym problem okazała się sytuacja, gdy na koniec kodowania 32 bitowa zmienna typu unsigned int była tylko częściowo wypełniona bitami związanymi ze słowem kodowym i to w taki sposób, iż nie były to pełne bajty (do pliku najmniej możemy zapisać 1 bajt) tylko np. 10 z 32 bitów. W takim przypadku bity pozostałe do pełnego bajtu ustawiane były na wartość jeden. Takie uzupełnienie jedynkami dla było jednocześnie sposobem na poinformowaniu dekodera o końcu sekwencji kodowej. Ponieważ dekoder napotykając uzupełniające jedynki będzie w pierwszej kolejności postępował tak jakby dekodował słowo kodowe dla frazy (odczyta bit flagi = 1), ale bitów będzie mniej niż dla typowego słowa kodowego frazy, bądź w skrajnym przypadku odczyta że dopasowanie o maksymalnej długości zaczyna się na końcu słownika, co jest naturalnie niemożliwe.

Dekoder LZSS

Wywołanie

Argumentami wywołania dekodera są:

- zakodowany plik
- wskazanie na docelowy plik ze zdekodowanymi danymi
- ewentualne wskazanie na plik ze statystykami

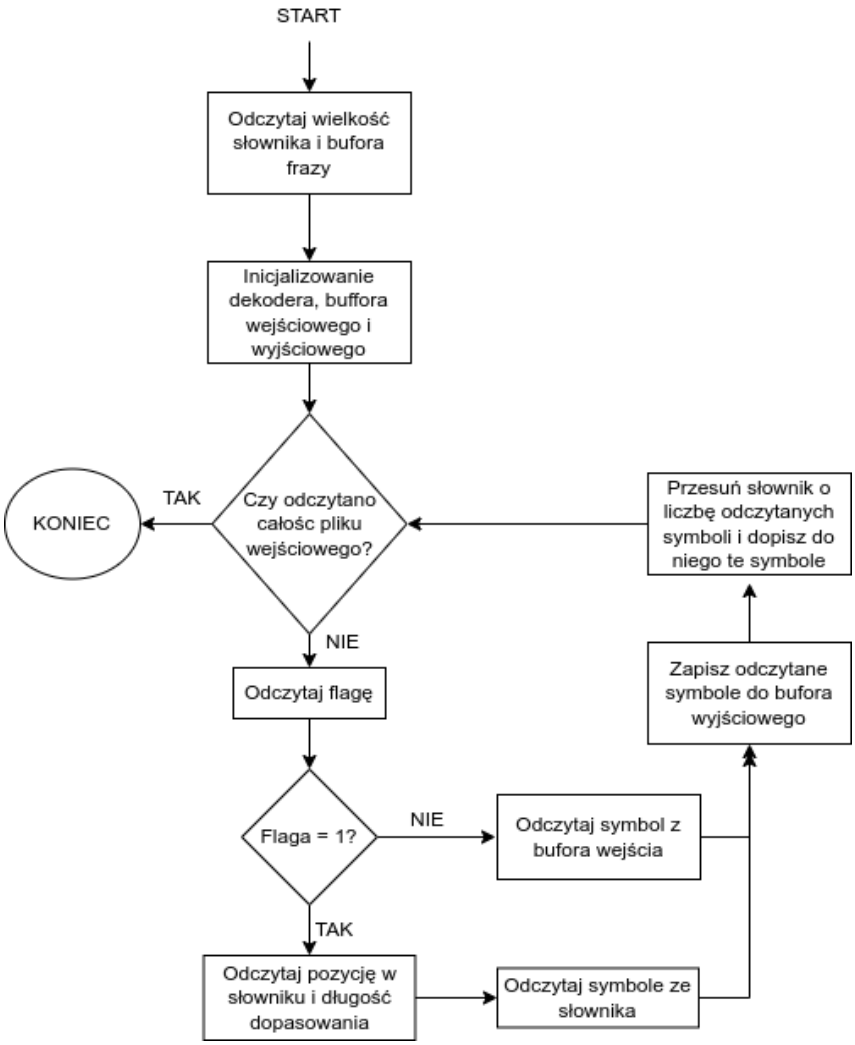
Rezultatem poprawnego wywołania dekodera jest odkodowanie danych do wskazanego pliku docelowego. Dodatkowo, jeżeli został podany plik ze statystykami zostanie do niego dopisania informacja o czasie dekodowania.

Kluczowe szczegóły implementacji algorytmu dekodera

W przeciwieństwie do kodera, dekoder nie wczytuje pliku wejściowego w całości do pamięci. Zamiast tego zaimplementowano specjalny buffer bitowy - BitBuf, który w pamięci przechowuje tylko aktualnie analizowane 4 bajty (32 bitów) pliku wejściowego. BitBuf umożliwia czytanie kolejnych bitów wejścia za pomocą operacji bitowych - alternatywy i przesunięć na zmiennej unsigned int (co wynika z rozmiaru analizowanych danych wejściowych, 4 bajty). Implementacja BitBuf zapewnia ciągłe wczytywanie kolejnych porcji danych wejściowych i dostarcza informacji o odczytaniu ostatniego słowa kodowego sygnalizującą ukończenie dekodowania danych.

Słownik został zaimplementowany jako tablica elementów typu char o określonej wartości i przechowuje jedynie aktualną zawartość słownika. Jest on aktualizowany i przesuwany w każdej iteracji algorytmu.

Dekoder trzyma otwarty strumień wyjścia do pliku wyjściowego i w każdej iteracji algorytmu zapisuje do niego dane odczytane z kolejnego słowa kodowego.



Rysunek 2: Schemat działania

Dane testowe

W pierwszej kolejności, w celu sprawdzenia poprawności działania przygotowanego algorytmu upewniono się, czy pliki przed kompresją są identyczne z plikami otrzymanymi po przejściu procesu kompresji i dekompresji. Oczywiście takie sprawdzenie jest możliwe tylko dla algorytmów kompresji bezstratnej, do której zalicza się kodowanie słownikowe LZSS.

Rozmiar plików testowych podlegających kodowaniu wynosił 262 182 bajty dla obrazów i 262 159 bajty dla rozkładów.

Testy dla: słownika = 4096 i bufora frazy = 32 bajty

Nazwa pliku	Czas kodowania [ms]	Czas dekodowania [ms]	Rozmiar po kodowaniu [bajty]	Wsp. kompresji	Liczba słów kodowych	Średnia długość bitowa
barbara.pgm	1614	1932	280 576	0.93	178 892	12.55
boat.pgm	1533	1693	276 096	0.95	156 878	14.08
chronometer.pgm	1200	1216	185 344	1.41	111 105	13.35
geometr_05.pgm	1289	547	103 552	2.53	46 405	17.85
geometr_099.pgm	2282	2551	294 656	0.89	233 412	10.10
geometr_09.pgm	1479	1411	250 752	1.05	119 755	16.75
laplace_10.pgm	1675	1742	285 824	0.92	153 096	14.94
laplace_20.pgm	2072	2232	293 376	0.89	191 261	12.27
laplace_30.pgm	2108	2430	294 400	0.89	215 992	10.90
lena.pgm	1705	1812	271 872	0.96	155 514	13.99
mandril.pgm	1790	2027	284 928	0.92	175 942	12.96
normal_10.pgm	1573	1584	282 112	0.93	138 604	16.28
normal_30.pgm	2063	2302	294 144	0.89	205 102	11.47
normal_50.pgm	2363	2681	294 656	0.89	234 429	10.06
peppers.pgm	980	1049	181 888	1.44	89 507	16.26
uniform.pgm	2520	2944	294 784	0.89	247 271	9.54

Testy dla: słownika = 2048 i bufora frazy = 32 bajty

Nazwa pliku	Czas kodowania [ms]	Czas dekodowania [ms]	Rozmiar po kodowaniu [bajty]	Wsp. kompresji	Liczba słów kodowych	Średnia długość bitowa
barbara.pgm	943	1087	278 272	0.94	197 098	11.29
boat.pgm	889	996	272 640	0.96	173 158	12.59
chronometer.pgm	648	695	187 776	1.39	119 927	12.52
geometr_05.pgm	733	301	106 496	2.46	50 716	16.79
geometr_099.pgm	1214	1385	292 736	0.89	245 988	9.52
geometr_09.pgm	823	733	252 032	1.04	131 888	15.28
laplace_10.pgm	958	995	278 656	0.94	169 910	13.12
laplace_20.pgm	1109	1234	287 872	0.91	212 567	10.83
laplace_30.pgm	1177	1394	291 072	0.90	233 684	9.964
lena.pgm	881	1004	269 568	0.97	175 611	12.28
mandril.pgm	967	1106	281 472	0.93	198 631	11.33
normal_10.pgm	863	890	274 816	0.95	153 420	14.33
normal_30.pgm	1147	1284	290 048	0.90	226 446	10.24
normal_50.pgm	1184	1365	292 864	0.89	246 781	9.49
peppers.pgm	536	572	186 496	1.40	100 234	14.88
uniform.pgm	1211	1471	293 888	0.89	254 422	9.24

Testy dla: słownika = 1024 i bufora frazy = 32 bajty

Nazwa pliku	Czas kodowania [ms]	Czas dekodowania [ms]	Rozmiar po kodowaniu [bajty]	Wsp. kompresji	Liczba słów kodowych	Średnia długość bitowa
barbara.pgm	483	578	278 272	0.94	214 740	10.37
boat.pgm	473	531	271 232	0.97	191 998	11.30
chronometer.pgm	351	366	191 488	1.37	129 351	11.84
geometr_05.pgm	366	158	110 080	2.38	55 990	15.73
geometr_099.pgm	585	697	292 608	0.90	253 471	9.24
geometr_09.pgm	443	405	251 264	1.04	146 449	13.73
laplace_10.pgm	613	541	274 560	0.95	190 566	11.53
laplace_20.pgm	593	770	286 720	0.91	230 924	9.93

Nazwa pliku	Czas kodowania [ms]	Czas dekodowania [ms]	Rozmiar po kodowaniu [bajty]	Wsp. kompresji	Liczba słów kodowych	Średnia długość bitowa
laplace_30.pgm	609	728	290 688	0.90	245 861	9.46
lena.pgm	496	545	269 952	0.97	197 288	10.95
mandril.pgm	539	620	281 088	0.93	219 278	10.26
normal_10.pgm	482	487	269 696	0.97	174 914	12.34
normal_30.pgm	778	948	289 664	0.91	241 842	9.58
normal_50.pgm	900	835	292 864	0.90	254 081	9.22
peppers.pgm	367	343	193 536	1.35	114 893	13.48
uniform.pgm	641	757	293 888	0.89	258 146	9.11

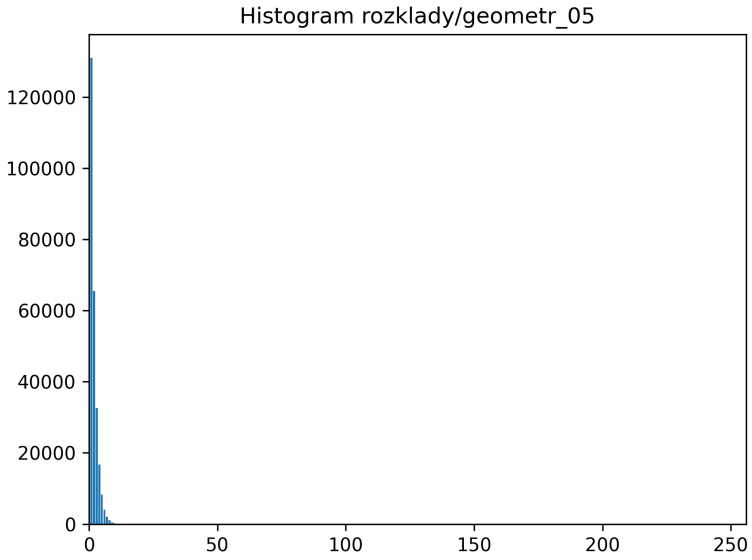
Testy dla: słownika = 2048 i bufora frazy = 64 bajty

Nazwa pliku	Czas kodowania [ms]	Czas dekodowania [ms]	Rozmiar po kodowaniu [bajty]	Wsp. kompresji	Liczba słów kodowych	Średnia długość bitowa
barbara.pgm	932	1076	285 440	0.92	197 098	11.59
boat.pgm	846	944	282 496	0.93	173 158	13.05
chronometer.pgm	638	658	193 792	1.35	119 624	12.96
geometr_05.pgm	702	283	112 768	2.32	50 716	17.79
geometr_099.pgm	1124	1305	294 784	0.89	245 988	9.59
geometr_09.pgm	789	724	264 960	0.99	131 888	16.07
laplace_10.pgm	874	942	289 664	0.91	169 910	13.64
laplace_20.pgm	1038	1144	293 888	0.89	212 567	11.06
laplace_30.pgm	1190	1270	294 528	0.89	233 684	10.08
lena.pgm	843	958	278 400	0.94	175 571	12.69
mandril.pgm	989	1076	288 768	0.91	198 631	11.63
normal_10.pgm	859	866	287 616	0.91	153 420	15.00
normal_30.pgm	1129	1275	294 528	0.89	226 446	10.41
normal_50.pgm	1168	1399	294 784	0.89	246 781	9.56
peppers.pgm	539	559	195 712	1.34	100 234	15.62
uniform.pgm	1213	1424	294 784	0.89	254 422	9.27

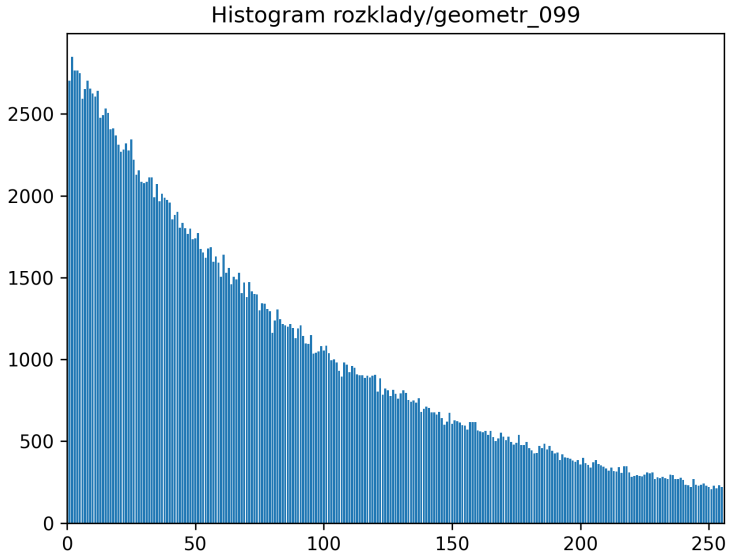
Histogram i entropia danych wejściowych

Program został przetestowany przy użyciu sztucznie wygenerowanych ciągów danych tekstowych oraz obrazów. Do testów zostały wykorzystane pliki o rozkładzie równomiernym, normalnym oraz Laplace. Poniżej znajdują się histogramy tych plików.

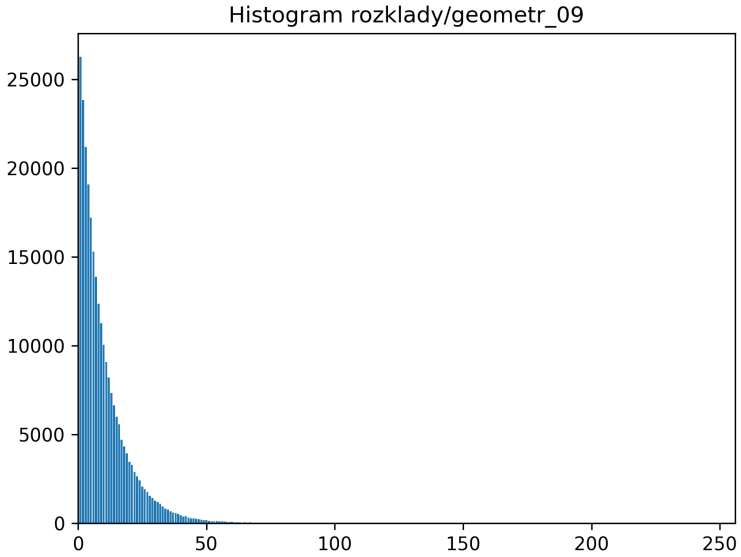
Entropia rozkładów



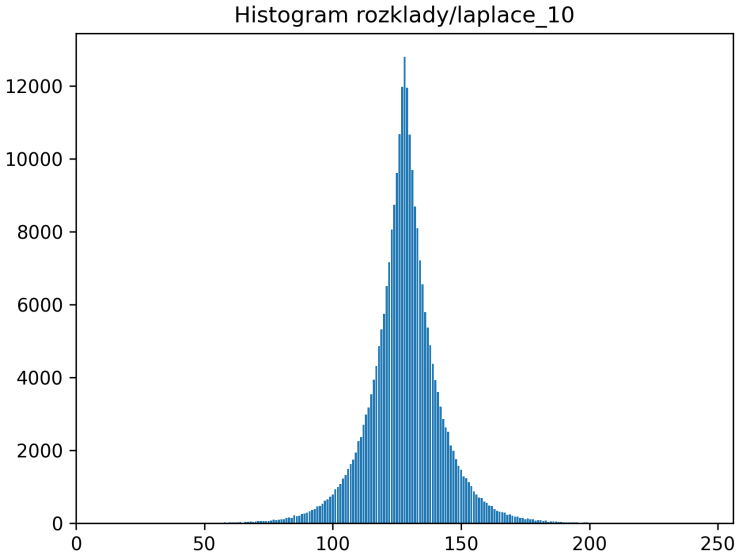
Rysunek 3: geometr_05.pgm - histogram.



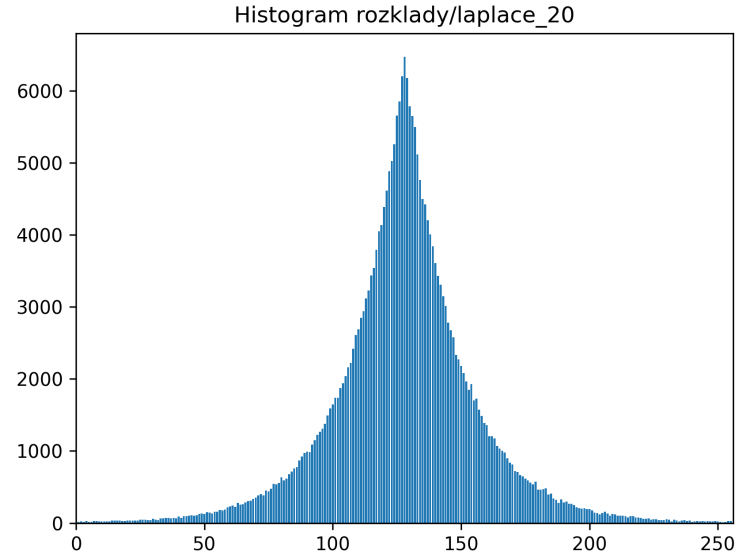
Rysunek 4: geometr_099.pgm - histogram



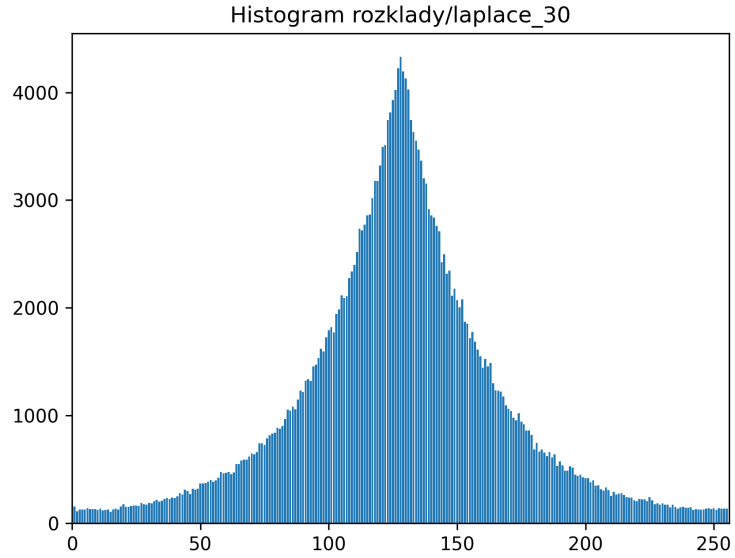
Rysunek 5: geometr_09.pgm - histogram



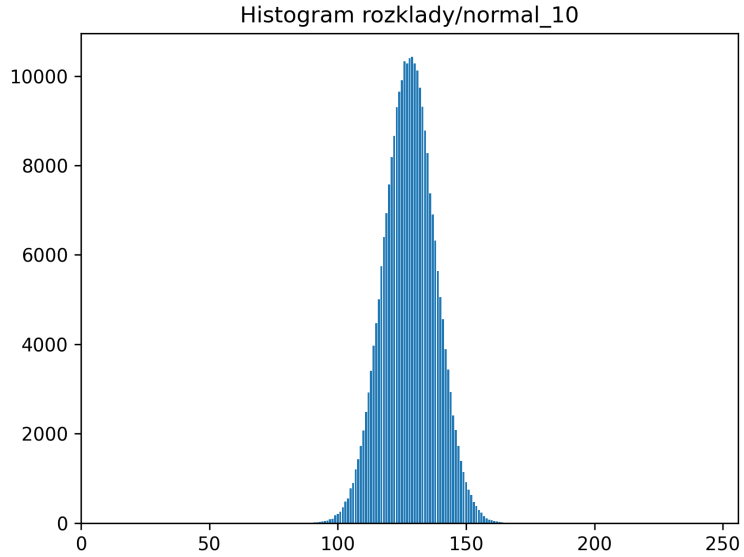
Rysunek 6: laplace_10.pgm - histogram



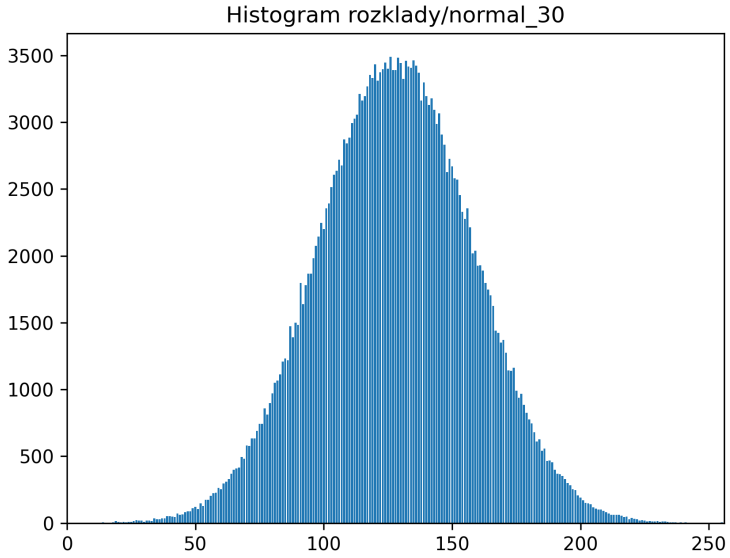
Rysunek 7: laplace_20.pgm - histogram



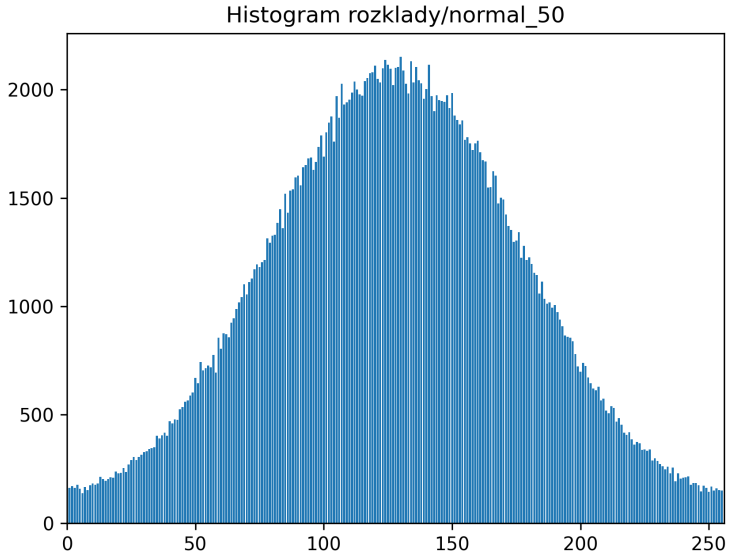
Rysunek 8: laplace_30.pgm - histogram



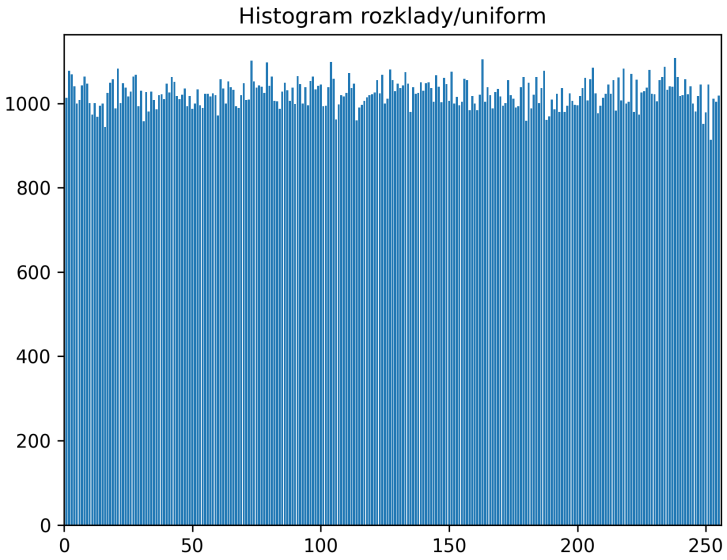
Rysunek 9: normal_10.pgm - histogram



Rysunek 10: normal_30.pgm - histogram

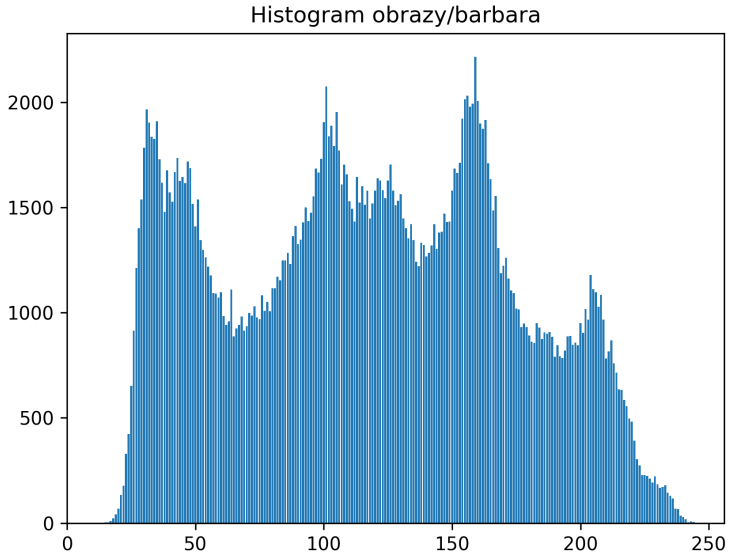


Rysunek 11: normal_50.pgm - histogram

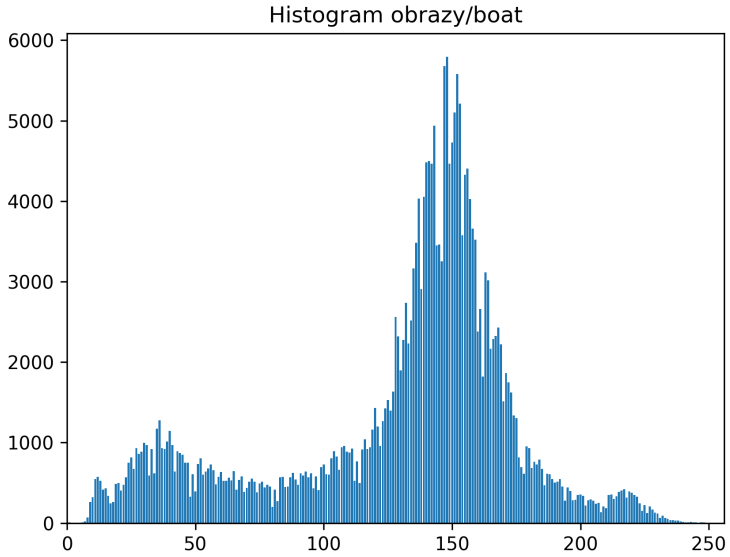


Rysunek 12: uniform.pgm - histogram

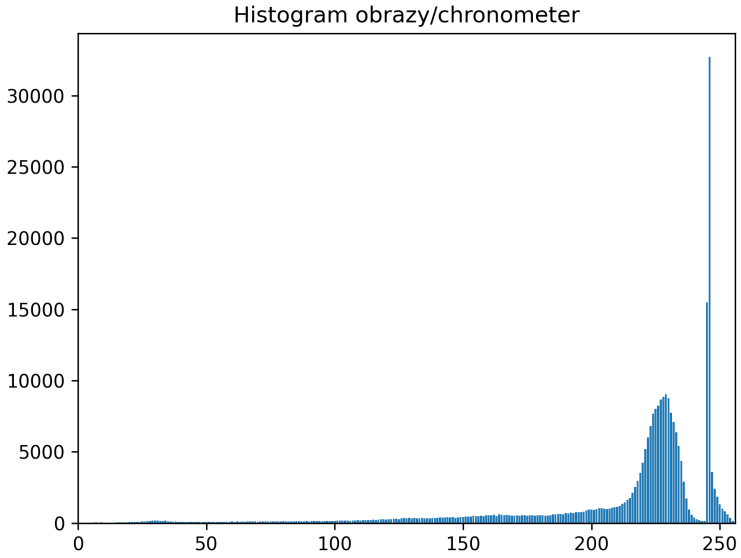
Entropia obrazów



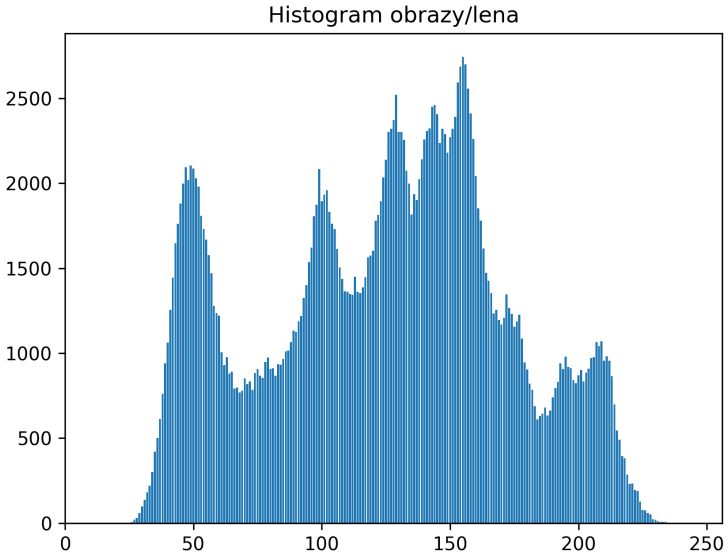
Rysunek 13: barbara.pgm - histogram



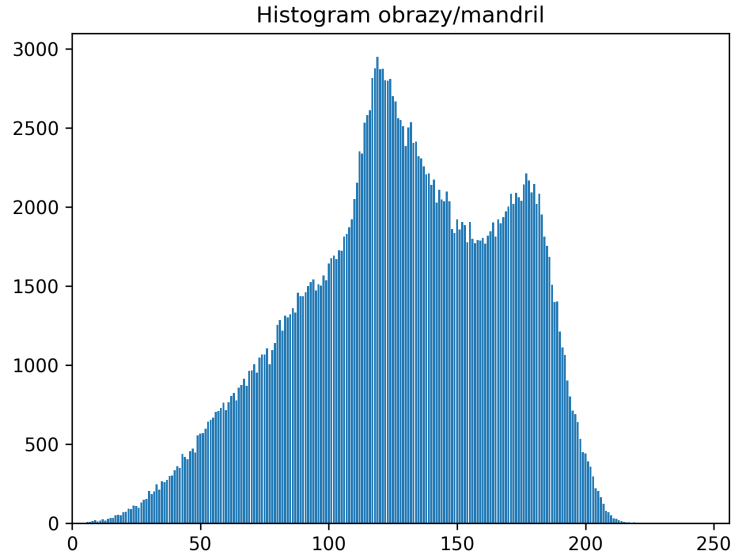
Rysunek 14: boat.pgm - histogram



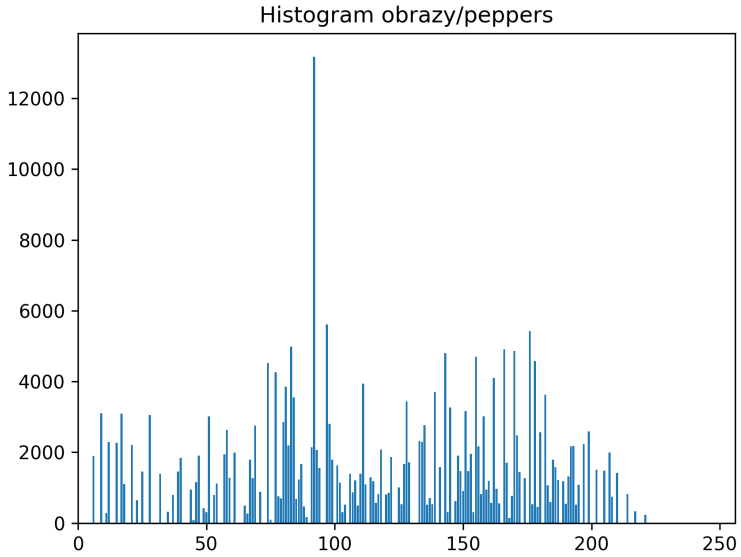
Rysunek 15: chronometer.pgm - histogram



Rysunek 16: lena.pgm - histogram



Rysunek 17: mandril.pgm - histogram



Rysunek 18: peppers.pgm - histogram

Entropia źródła blokowego rzędu 2 i 3

Tabela Entropii danych wejściowych

Plik	Entropia	Entropia Źródła blokowego 2 rzędu	Entropia Źródła blokowego 3 rzędu
barbara.pgm	7.632119011	8.830805669	11.44049797
boat.pgm	7.191370218	8.305931861	11.11675819
chronometer.pgm	6.113328353	6.649469668	8.413736565
lena.pgm	7.445061353	8.260271592	10.9359003
mandril.pgm	7.292548775	8.781243037	11.53755631
peppers.pgm	6.762425168	6.935477559	8.894681322
geometr_05.pgm	2.000995001	2.773701012	4.158251694
geometr_09.pgm	4.693658432	6.497106357	9.585845249
geometr_099.pgm	7.65440713	9.537849618	12.35606984
laplace_10.pgm	5.767144375	7.959112291	11.30569869
laplace_20.pgm	6.756458638	9.09743682	12.16752892
laplace_30.pgm	7.287118849	9.46102916	12.32849581
normal_10.pgm	5.368684963	7.432366604	10.8641293
normal_30.pgm	6.95431406	9.432597371	12.3270292
normal_50.pgm	7.649568445	9.667618415	12.39125859
uniform.pgm	7.999317791	9.672492641	12.39279973

Obserwacje

Wśród obrazów najmniejszą entropią cechowały się pliki chronometer i peppers, natomiast wśród danych losowych geometr_05 następnie geometr_09, normal_10 i laplace_10.

Największy wpływ na wyniki kompresji miał rodzaj danych testowych oraz ich entropia:

- Obrazy o najniższej wartości entropii pozwalały na uzyskanie mniejszego rozmiaru pliku (chronometer.pgm, peppers.pgm), a te o wyższej wartości nie doświadczały zmian w rozmiarze.
- W przypadku danych losowych sytuacja miała się podobnie – plik o najmniejszej entropii (geometr_05.pgm) dawał największy stopień kompresji – wynika to też z tego, że plik geometr_05.pgm cechuje się najmniejszą ilością różnych znaków.
- Widać tu więc potwierdzenie teorii stojącej za kodowanie LZSS – powtarzające się dane są tokenizowane (o ile token nie zajmuje więcej niż zastępowane przez niego dane) dzięki czemu widać zależność: im większa ilość powtarzających się danych w pliku, tym wyższy stopień kompresji.

Dla każdego pliku wejściowego uzyskana **średnia bitowa jest większa od entropii**, wynika to z bezstratności kompresji LZSS.

W przypadku przygotowanej implementacji czasy kodowania i dekodowania są dosyć zbliżone, choć według teorii dekoder powinien wykonywać się szybciej, gdy nie musi przeszukiwać słownika tak jak ma to miejsce w koderze, gdzie szukamy najdłuższej frazy. W naszym przypadku zbliżone czasy wynikają z faktu, że mierzony czas wykonania kodera nie uwzględnia czasu poświęconego na odczytanie całego pliku wejściowego. Natomiast dekoder na bieżąco odczytuje kolejne porcje danych z pliku wejściowego i po ich zdekodowaniu są one zapisywane do pliku wyjściowego, stąd też operacje te są uwzględnione w czasie wykonania. Należy zatem zakładać, iż operacje wejścia/ wyjścia zajmując najwięcej czasu podczas pracy dekodera. W przypadku dekodera można powiedzieć, że poświęcamy więcej czasu na czytanie i zapis, ale za to podczas wykonania zajmujemy mniej pamięci. Odwrotna sytuacja jest w koderze, gdzie wczytując cały plik do pamięci podręcznej zyskujemy na czasie wykonania poprzez eliminację częstych operacji odczytu z dysku.

Ocena efektywności algorytmu do kodowania obrazów naturalnych.

Jak wykazały testy algorytmu LZSS pomimo używania różnych rozmiarów słownika i bufora frazy, dla większości danych testowych, które stanowiły obrazy pliki wyjściowe były większe od plików wejściowych. Można zatem stwierdzić, iż kodowanie LZSS nie jest najlepszym rozwiązaniem do kodowania obrazów naturalnych. Prawdopodobnie algorytm LZSS lepiej sprawdza się przy danych tekstowych, w których to zwykle powtarzają się pewne sekwencje słów.