Kompresja bezstratna

Sprawozdanie - Systemy Multimedialne

Hubert Rosiński

§ Implementacje kompresji RLE i ByteRun w Pythonie z użyciem biblioteki NumPy różnią się podejściem do zarządzania pamięcią, mimo że obie metody służą do redukcji rozmiaru danych przez eliminację redundancji. RLE sprowadza dane do jednowymiarowego wektora i zapisuje skompresowane dane w buforze, którego rozmiar jest dostosowywany do rzeczywistej długości danych. ByteRun również spłaszcza dane, ale różnicuje obsługę sekwencji powtarzających się i unikalnych wartości, co może zwiększać efektywność kompresji. Oba algorytmy przechowują informacje o wymiarach danych, umożliwiając ich pełną rekonstrukcję po dekompresji. Zarządzanie pamięcią w obu metodach polega na dynamicznym alokowaniu i dostosowywaniu buforów, co minimalizuje zapotrzebowanie na pamięć.

Teraz zaprezentujemy poprawność kodowania poszczególnych metod na kilku przykładach danych testowych, w ten sposób upewnimy się czy algorytmy kompresji i dekompresji działają poprawnie.

Najpierw przetestujemy funkcję z zaimplementowanym algorytmem RLE do kompresji i dekompresji danych, a to są dane testowe oraz funkcja to testowania.

```
t1 = np.array([5,1,5,1,5,5,1,1,5,5,1,1,5])
t2 = np.dstack([np.eye(7),np.eye(7),np.eye(7)])
t3 = np.eye(7)
tests = [t1, t2, t3]
def RLE_test(tests):
    for x, i in enumerate(tests):
        i = i.astype(int)
        print("Original_data: ", i)
        print(f"Original_size: {i.nbytes}")
        compress = test_RLE_encode(i)
        print(f"encode_test_{x+1}:",compress)
        print(f"Size_encode: {compress.nbytes}")
        decompress = test_RLE_decode(compress)
        print(f"decode_test_{x+1}:",decompress)
        print(f"Size_decode: {decompress.nbytes}")
RLE_test(tests)
```

W przypadku pierwszych danych testowych rozmiar wzrósł zamiast się zmniejszyć, ponieważ redundancja danych jest tu stosunkowo mała co prowadzi do tego że sam zapis ilości pojedynczych danych zwiększa rozmiar całej listy.

```
Original_data:  [5 1 5 1 5 5 1 1 5 5 1 1 5]
Original_size: 52
encode_test_1: [ 1 13  1  5  1  1  1  5  1  1  2  5  2  1  2  5  2  1  1  5]
Size_encode: 80
decode_test_1: [5 1 5 1 5 5 1 1 5 5 1 1 5]
Size_decode: 52
```

Podczas drugiego testu jest znacznie lepiej ponieważ oryginalne dane znacznie częściej się powtarzają w dużych ilościach (szczególnie 0) co prowadzi do dużego zmniejszenia ilości danych przy kodowaniu.

```
Original_data:  [[[1 1 1]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [1 1 1]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [1 1 1]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [0 0 0]
  [1 1 1]
  [0 0 0]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [1 1 1]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [1 1 1]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
```

```
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [1 1 1]]]
Original_size: 588
encode_test_2: [ 3  7  7  3  3  1 21  0  3  1 21  0  3  1 21  0  3  1 21  0  3  1 21  0
  3  1 21  0  3  1]
Size_encode: 120
```

```
decode_test_2: [[[1 1 1]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [1 1 1]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [1 1 1]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [0 0 0]
  [1 1 1]
  [0 0 0]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [1 1 1]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [1 1 1]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [0 0 0]
```

```
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [1 1 1]]]
Size_decode: 588
```

Ostatni przykład dotyczy tablicy np.eye(7), która po skompresowaniu trochę zmniejszyła swój rozmiar. Następnie dane zostały identycznie odtworzone za pomocą funkcji reshape() oraz danych z informacją o rozmiarze.

```
Original_data:  [[1 0 0 0 0 0 0]
 [0 1 0 0 0 0 0]
 [0 0 1 0 0 0 0]
 [0 0 0 1 0 0 0]
 [0 0 0 0 1 0 0]
 [0 0 0 0 0 1 0]
 [0 0 0 0 0 0 1]]
Original_size: 196
encode_test_3: [2 7 7 1 1 7 0 1 1 7 0 1 1 7 0 1 1 7 0 1 1 7 0 1 1 7 0 1 1]
Size_encode: 116
decode_test_3: [[1 0 0 0 0 0 0]
 [0 1 0 0 0 0 0]
 [0 0 1 0 0 0 0]
 [0 0 0 1 0 0 0]
 [0 0 0 0 1 0 0]
 [0 0 0 0 0 1 0]
 [0 0 0 0 0 0 1]]
Size_decode: 196
```

Jako drugi algorytm do kompresji i dekompresji danych użyjemy Byterun, który różni się działaniem od poprzedniego sposobu i jest trochę bardziej skomplikowany w implementacji, jednak jest wydajniejszy od standardowego RLE dla danych które mają zarówno długie powtórzenia jak i krótkie. Oto przykładowe działanie na tych samych danych co w przypadku algorytmu RLE:

```
ByteRun:
Original_data:  [5 1 5 1 5 5 1 1 5 5 1 1 5]
Original_size: 52
100%|                                                                                | 13/13 [00:00<?, ?it/s]
encode_test_1: [ 1 13  3  5  1  5  1 -1  5 -1  1 -1  5 -1  1  0  5]
Size_encode: 68
100%|                                                                                | 13/13 [00:00<?, ?it/s]
decode_test_1: [5 1 5 1 5 5 1 1 5 5 1 1 5]
Size_decode: 52
```

```
 [[0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [1 1 1]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [1 1 1]]]
Original_size: 588
100%|                                                                                | 147/147 [00:00<?, ?it/s]
encode_test_2: [  3   7   7   3  -2   1 -20   0  -2   1 -20   0  -2   1 -20   0  -2   1
 -20   0  -2   1 -20   0  -2   1 -20   0  -2   1]
Size_encode: 120
100%|                                                                                | 147/147 [00:00<?, ?it/s]
decode_test_2: [[[1 1 1]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [1 1 1]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [1 1 1]
  [0 0 0]
```

```
Original_data:  [[1 0 0 0 0 0 0]
 [0 1 0 0 0 0 0]
 [0 0 1 0 0 0 0]
 [0 0 0 1 0 0 0]
 [0 0 0 0 1 0 0]
 [0 0 0 0 0 1 0]
 [0 0 0 0 0 0 1]]
Original_size: 196
encode_test_3: [ 2  7  7  0  1 -6  0  0  1 -6  0  0  1 -6  0  0  1 -6  0  0  1 -6  0  0
  1 -6  0  0  1]
Size_encode: 116
100%|                                                                                | 49/49 [00:00<?, ?it/s]
decode_test_3: [[1 0 0 0 0 0 0]
 [0 1 0 0 0 0 0]
 [0 0 1 0 0 0 0]
 [0 0 0 1 0 0 0]
 [0 0 0 0 1 0 0]
 [0 0 0 0 0 1 0]
 [0 0 0 0 0 0 1]]
Size_decode: 196
```

Teraz przeprowadzimy testy na poszczególnych zdjęciach wybranych przeze mnie za pomocą każdej z metod. Udowodnimy poprawność (identyczność) danych po dekompresji z oryginalnymi danymi wejściowymi oraz policzymy skuteczność kompresji w dwóch różnych sposobach. To jest fragment kodu który odpowiada za wykonywane testy (Dopisałem w kodzie tytuły, aby było wiadomo które zdjęcie jest aktualnie testowane):

```
226  def CR(size_before, size_after):
227      return size_before / size_after
228
229  def PR(size_before, size_after):
230      return (size_after / size_before) * 100
231  def compression_test(img, encode_function, decode_function):
232      original_size = img.nbytes
233      encoded = encode_function(img)
234      decoded = decode_function(encoded)
235      encoded_size = encoded.nbytes
236      if np.array_equal(img, decoded):
237          print("Identyczne dane.")
238      else:
239          print("Nie identyczne dane.")
240      cr = CR(original_size, encoded_size)
241      pr = PR(original_size, encoded_size)
242      return original_size, encoded_size, cr, pr
243  def main():
244      images = [img_tech, img_wzor, img_color]
245      results = []
246      for img in images:
247          img = img.astype(int)
248          print(f"Rozmiar oryginalny: {img.nbytes} bajtów")
249          print("Test RLE:")
250          rle_results = compression_test(img, test_RLE_encode, test_RLE_decode)
251          results.append(('RLE', rle_results))
252          print(f"Rozmiar zakodowany: {rle_results[1]} bajtów, CR: {rle_results[2]}, PR: {rle_results[3]}%")
253          print("Test Byterun:")
254          byterun_results = compression_test(img, Byterun_encode, Byterun_decode)
255          results.append(('Byterun', byterun_results))
256          print(f"Rozmiar zakodowany: {byterun_results[1]} bajtów, CR: {byterun_results[2]}, PR: {byterun_results[3]}%")
257      return results
258
259  if __name__ == "__main__":
260      main_results = main()
```

Wyniki testów wyglądają następująco:

```
Obraz: Rysunek techniczny, Rozmiar oryginalny: 26099520 bajtów
Test RLE:
Identyczne dane.
Rozmiar zakodowany: 1484208 bajtów, CR: 17.584812910319847, PR: 5.686725273108471%
Test Byterun:
100%|                                                                                | 6524880/6524880 [00:08<00:00, 738423.09it/s]
100%|                                                                                | 6524880/6524880 [00:02<00:00, 2897064.09it/s]
Identyczne dane.
Rozmiar zakodowany: 1805512 bajtów, CR: 14.455467479584739, PR: 6.917797721950442%
Obraz: Wzor dokumentu, Rozmiar oryginalny: 6345216 bajtów
Test RLE:
Identyczne dane.
Rozmiar zakodowany: 73608 bajtów, CR: 86.2028040430388, PR: 1.160055071411280 5%
Test Byterun:
100%|                                                                                | 1586304/1586304 [00:01<00:00, 898548.32it/s]
100%|                                                                                | 1586304/1586304 [00:00<00:00, 6081333.67it/s]
Identyczne dane.
Rozmiar zakodowany: 159888 bajtów, CR: 39.68537976583608, PR: 2.5198196562575648%
Obraz: Kolorowe zdjęcie, Rozmiar oryginalny: 21964800 bajtów
Test RLE:
Identyczne dane.
Rozmiar zakodowany: 43145840 bajtów, CR: 0.5090826832899765, PR: 196.43174533799535%
Test Byterun:
100%|                                                                                | 5491200/5491200 [00:11<00:00, 498469.65it/s]
100%|                                                                                | 5491200/5491200 [00:03<00:00, 1707893.13it/s]
Identyczne dane.
Rozmiar zakodowany: 22434776 bajtów, CR: 0.9790514511934507, PR: 102.13967803030303%
```

Wnioski:

- Wszystkie obrazy zostały skutecznie skompresowane przez obie metody. W przypadku rysunku technicznego, wzoru dokumentu i kolorowego zdjęcia, kompresja Byterun okazała się być bardziej efektywna niż RLE, co widać po wyższym CR i niższym PR.
- Zapis „Identyczne dane.” potwierdza, że dane wejściowe i wyjściowe są identyczne po kompresji i dekompresji, co jest kluczowe dla każdego algorytmu kompresji danych.
- Efektywność kompresji i dekompresji znacznie zależy od metody jaką użyjemy w danych czasie oraz od danych które chcemy przetworzyć.