| Telekomunikacja - laboratorium | | | | | | Studia stacjonarne - inżynierskie | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Nazwa zadania** | | **Implementacja algorytmu kodowania Huffmana** | | | | | | |
| **Dzień** | **Poniedziałek** | | | **Godzina** | **14:00 – 15:30** | | **Rok akademicki** | **2020/2021** |
| **Imię i Nazwisko** | | | **Adam Kapuściński 229907** | | | | | |
| Imię i Nazwisko | | | **Damian Szczeciński 230016** | | | | | |
| Imię i Nazwisko | | |  | | | | | |
| **Opis programu, rozwiązania problemu.** | | | | | | | | |
| Celem zadania było opracowanie programu zdolnego do skompresowania, wysłania i rozpakowania pliku przez gniazdo sieciowe. Wymagany algorytm kompresji to algorytm kodowania Huffmana. Algorytm działa w następujący sposób:   * obliczamy ilość wystąpień danego słowa (jednego lub więcej bajtów) w pliku * tworzymy drzewo kodowe w taki sposób, że liście tego drzewa zawierają słowa, a wszystkie węzły zawierają wagi. Węzły łączymy w taki sposób, że mniejszy jest po lewej, a większy po prawej, natomiast ich rodzicem jest węzeł o wadze sumy wagi dzieci. Każde przejście po drzewie (od korzenia) w lewo to bitowe „0”, a przejście w prawo to bitowe „1”. W ten sposób optymalizujemy ilość zużytych bitów na dane słowo (im słowo występuje częściej tym ma krótszą definicję) * W naszym programie, dla wygody przepisujemy drzewo na słownik, gdzie kluczem słownika jest słowo, a wartością pod kluczem jest definicja   W przypadku komunikacji sieciowej przyjęliśmy następującą konwencję:   * Serwer wysyła informację o rozmiarze drzewa kodowego * Serwer wysyła drzewo kodowe * Serwer kompresuje i wysyła plik   Jako, że klient na początku otrzymuje rozmiar drzewa, to jest w stanie odróżnić gdzie powinien zakończyć odbieranie drzewa, a zacząć odbieranie pliku. Wielkość paczek w jakich transmitowane są te dane jest konfigurowalna w naszym programie, ale domyślnie używamy pakietów o rozmiarze 512B.  Nasz program charakteryzuje się tym, że procesy odczytu pliku z dysku, kompresji i wysyłania są ze sobą ściśle powiązane, dzięki czemu nie zużywamy dużo zasobów, jednak wadą tego rozwiązania jest niezwykle wolna transmisja po sieci. | | | | | | | | |
| **Najważniejsze elementy kodu programu z opisem.** | | | | | | | | |
| Funkcje odpowiedzialne za odpowiednio: obliczenie wystąpień słów w pliku, utworzenie drzewa, utworzenie słownika na podstawie drzewa:   1. def \_\_analyze(self): 2. freq = {} 3. # zaanalizuj plik bajt po bajcie 4. while True: 5. byte = self.FILE.read(WORD\_SIZE) 6. # jeżeli koniec pliku to zakończ 7. if not byte: break 8. # dopisz / zapisz ilość wystąpień danego byte'u 9. if not byte in freq: freq[byte] = 1 10. else: freq[byte] += 1 11. # posortuj po ilości wystąpień 12. freq = dict(sorted(freq.items(), key=lambda item: item[1])) 13. # wróć na początek pliku 14. self.FILE.seek(0) 15. # przepisz słownik frekwencji na listę node'ów 16. tmp = [] 17. i = 0 18. for key in freq: 19. tmp.append(Node(key, freq[key])) 20. tmp[i].leaf = True 21. i += 1 22. self.tree = tmp 24. def \_\_createTree(self): 25. if type(self.tree) != list: return 26. while len(self.tree) > 1: 27. # posortuj listę po freq 28. self.tree = sorted(self.tree, key=lambda x : x.freq) 30. # weź 2 node'y o najmniejszym freq 31. left = self.tree[0] 32. right = self.tree[1] 33. # utwórz nowy node o wartości freq sumy lewego i prawego 34. newNode = Node(b'', left.freq + right.freq, left, right) 36. # usuń node'y wykorzystane do zbudowania pod drzewa 37. self.tree.remove(left) 38. self.tree.remove(right) 39. # dodaj poddrzewo do listy 40. self.tree.append(newNode) 41. # dla wygody przepisz obiekt 42. self.tree = self.tree[0] 44. def \_\_rewriteAsDict(self, node=None, code=''): 45. # jeżeli pierwsze odwołanie, to zacznij od roota 46. if node == None: node = self.tree 48. # jak idziesz w lewo, to dopisz '0' 49. if node.left: 50. self.\_\_rewriteAsDict(node.left, code + "0") 52. # jak idziesz w prawo, to dopisz '1' 53. if node.right: 54. self.\_\_rewriteAsDict(node.right, code + "1") 56. # jak to liść, to zapisz kod w słowniku 57. if node.isLeaf(): 58. self.dic[node.char] = code   odczyt i kompresja pliku:   1. def readNext(self): 2. to\_send = BitArray() 3. # okreslenie ile razy ma się wykonać pętla (ilość bitów) 4. bits\_to\_read = PACKET\_SIZE 5. while len(to\_send.bin) != bits\_to\_read: 6. # jeżeli wartość jest pusta to 7. if len(self.buffer) == 0: 8. # przypisujemy pierwszy bajt z pliku 9. byte = self.FILE.read(WORD\_SIZE) 10. # jeżeli nie ma to kończymy 11. if not byte: 12. self.EOF = True 13. break 14. # przypisujemy wartości definicję 15. self.buffer = self.dic[byte] 16. # dodajemy bit do naszej tablicy 17. to\_send.bin += self.buffer[0] 18. # ucinamy dodany bit 19. self.buffer = self.buffer[1:] 20. return to\_send.tobytes()   funkcja dekompresująca i zapisująca dane do pliku   1. def write(self, data): 2. # data (bytes) -> BitArray 3. data = BitArray(data) 4. result = b'' 5. # przejdź po każdym bicie 6. for bit in data.bin: 7. # jeżeli napotkany node jest liściem 8. if self.buffer.isLeaf(): 9. # dodaj jego 'char' do wyniku 10. result += self.buffer.char 11. # wróć do korzenia 12. self.buffer = self.root 13. # jeżeli bit to '1', przejdź w prawo 14. if bit == '1': self.buffer = self.buffer.right 15. # jeżeli bit to '0', przejdź w lewo 16. if bit == '0': self.buffer = self.buffer.left 17. # zapisz bity do pliku 18. self.FILE.write(result) 19. # jeżeli nie dokończono jeszcze odczytu jakiegoś bajtu 20. # to jest to zapisane w bufforze   Klient naszego programu (strona odbierająca):   1. print('Nawiązywanie połączenia.') 2. with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s: 3. s.connect((HOST, PORT)) 4. print('Utworzono połączenie z {}:{}'.format(HOST,PORT)) 5. tree\_size = int.from\_bytes(s.recv(PACKET\_SIZE), "big") # odbierz rozmiar drzewa 6. # odbierz drzewo 7. data = b'' 8. while len(data) != tree\_size: 9. data += s.recv(PACKET\_SIZE) 10. tree = pickle.loads(data) 11. # utwórz obiekt Writera 12. writer = HuffmanWriter(filename, tree) 14. # odbierz dane 15. i = 1 16. while data != b'': 17. print("Odbieram pakiet nr: {}".format(i), end="\r") 18. i += 1 19. data = s.recv(PACKET\_SIZE) 20. TRANSMITTED\_DATA\_SIZE += len(data) 21. writer.write(data) 22. print('') 23. # zamknij plik 24. writer.close()   Serwer naszego programu (strona nadająca):   1. with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s: 2. s.bind((HOST, PORT)) 3. s.listen() 4. print("Serwer uruchomiono na porcie: {}".format(PORT)) 5. conn, addr = s.accept() 6. with conn: 7. print('Nawiązano połączenie z: {}'.format(addr)) 8. conn.sendall(len(tree\_str).to\_bytes(8, byteorder='big')) # wyślij rozmiar drzewa 9. # wyślij drzewo 10. while len(tree\_str) > 0: 11. data = tree\_str[0:int(PACKET\_SIZE/8)] 12. conn.sendall(data) 13. tree\_str = tree\_str[int(PACKET\_SIZE/8):] 14. # wyślij plik 15. i = 1 16. while not reader.isEOF(): 17. print("Wysyłam pakiet nr: {}".format(i), end="\r") 18. i+=1 19. data = reader.readNext() 20. TRANSMITTED\_DATA\_SIZE += len(data) 21. conn.sendall(data) 22. print('') 23. # zamknij plik 24. reader.close() | | | | | | | | |
| **Podsumowanie wnioski.** | | | | | | | | |
| Cel zadania możemy uznać za wykonany. Program jest w stanie bez problemu nadać i odebrać dowolny plik przez gniazdo sieciowe, gdzie serwer wcześniej kompresuje dane, a klient je dekompresuje.  Nasza metoda odczytywania i kompresji pliku na bieżąco jest bardzo dobra w przypadku transmisji dużych plików które dobrze kompresują się tym algorytmem, czyli np. duże pliki wykonywalne albo bitmapy, jednak dodaje niepotrzebne opóźnienie między wysyłaniem następnych pakietów przy plikach, które z założenia są dobrze skompresowane (mp4, jpg, zip). Można by to poprawić oddelegowując odczyt, kompresję i transmisję na oddzielne wątki które miałyby własne bufory, jednak wymagałoby to przepisania aplikacji z myślą o takiej architekturze. | | | | | | | | |