Algoritmul Huffman r-ar – documentație

Cerință: implementarea algoritmului huffman r-ar pentru codificarea și decodificarea unui text. Proiectul conține:

**Models/Node.py**

* Conține clasa *Node* ce reprezintă un nod al unui arbore;
* Membrii:
  + character - reprezintă un caracter din text
  + probability - reprezintă probabilitatea caracterului din text
  + children - reprezintă lista de copii de tip nod ai nodului curent
* Funcții:
  + **\_\_lt\_\_(self, other)**

Reprezintă un comparator care arată cum se fie comparate două obiecte de tip *Node*.

* + **add\_child(self,child)**

Adaugă un nod *child* la lista de copii.

* + **get\_probability(self)**

Getter pentru *probability*.

* + **set\_probability(self, probability)**

Setter pentru *probability*.

* + **get\_character(self)**

Getter pentru *character.*

* + **set\_character(self, character)**

Setter pentru *character.*

* + **get\_children(self)**

Getter pentru *children.*

* + **is\_leaf(self)**

Determină dacă nodul are copii sau nu.

**Models/PriorityQueue.py**

* Conține clasa *PriorityQueue* ce este o implementare pentru un heap min;
* Membrii:
  + values – reprezintă lista de valori stocate
* Funcții:
  + **insert(self, value)**

Adaugă valoarea *value* la lista de valori, după care se apelează funcția *sift\_up*(0).

* + **sift\_up(self, position)**

Funcția reface proprietatea de heap min începând cu nodul de pe poziția *position*. Cât timp poziția din mijlocul secvenței [0, *position*] este un nod cu valoarea mai mare decât nodul curent, cele două se interschimbă și după se repetă procedura pentru *position* egal cu mijloc. Cu alte cuvinte nodul de pe *position* urcă în heap până își găsește locul.

* + **extract\_min(self)**

Funcția scoate elementul minim din heap și îl returnează. Pentru a scoate minimul din heap, îl interschimb cu ultima valoare din *values* care este cea mai mare, după care elimin ultima valoare și refac proprietatea de heap apelând *sift*\_*down*(0).

* + **sift\_down(self, position)**

Funcția reface proprieteatea de heap min începând cu nodul de la poziția *position*. Dacă acesta are copii și unul din copii este mai mare, se interschimbă cei doi și se va apela *sift\_down* cu poziția nodului fiu. Cu alte cuvinte, nodul curent coboară în heap până își găsește locul.

* + **get\_min(self)**

Returnează valoarea minimă din heap, care este rădăcina.

* + **is\_empty(self)**

Determină dacă heap-ul are elemente sau nu.

* + **get\_size(self)**

Returnează numărul de elemente din heap.

* + **get\_values(self)**

Getter pentru *values*.

**Helper.py**

* Conține clasa *Helper* ce are funcții statice apelate în main sau în teste.
* Funcții:
  + **read\_file(filename)**

Funcția citește și returnează conținutul fișierului primit ca parametru.

* + **encoding(encoding\_dictionary, input)**

Funcția codează textul primit ca input folosind dictionarul de codare. Dicționarul are ca și chei caracterele textului, iar ca valori cuvintele cod aferente.

* + **decoding(encoding\_dictionary, input)**

Funcția decodează textul primit ca input folosind același dicționar de codare. Pentru a decoda, parcurg caracterele textului codat, și în momentul în care ceea ce am parcurs reprezintă un cuvânt cod, mă duc în dicționar și îl înlocuiesc cu caracterul pe care îl reprezintă.

**Huffman.py**

* Conține clasa *Huffman* ce are algoritmul Huffman n-ar
* Membrii:
  + nrSymbols – reprezintă lungimea codului, numărul de simboluri de codare folosite
* Funcții:
  + **get\_encoding\_dictionary(self, text)**

Funcția primește ca input un text și returnează dicționarul de codare folosit de funcțiile *encoding* și *decoding* de mai sus. Se calculează probabilitățile textului folosind funcția *process,* se construiește heap-ul corespunzător folosind funcția *buid\_heap,* se calculează s (nr simboluri din prima reducere) și r (nr simboluri din restul reducerilor, este egal cu *nrSymbols*) și se construiește arborele Huffman apelând funcția *build\_huffman\_tree*. Pentru a construi dicționarul de codare, se inițializează un dicționar gol, și se apelează funcția *build\_encoding\_dictionary* cu parametrii: arborele huffman, dicționarul inițializat anterior și un string gol în care se construiesc codurile.

* + **process(self, text)**

Funcția calculează probabilitățile de apariție ale caracterelor în textul dat și returnează un dicționar unde cheile sunt caracterele iar valorile sunt probabilitățile.

* + **build\_heap(self, dictionary)**

Funcția primește dicționarul cu probabilități și construiește un heap min, prin insertii succesive de noduri care reprezintă tuplele dicționarului. Returnează heap-ul construit.

* + **build\_huffman\_tree(heap, s, r)**

Funcția primește heap-ul construit anterior, *s* – ce reprezintă numărul de caractere din prima redurece și *r* – numărul de caractere din restul reducerilor. Pentru început funcția face prima reducere: se creează un nod nou care va avea ca și fii primele *s* noduri din heap, acestea sunt scoase din heap și probabilitățile lor sunt cumulate, iar nodului noi i se atribuie suma probabilităților și se inserează înapoi în heap. După aceea se fac reducerile de *r* caractere până nu mai sunt elemente în heap. Aceste reduceri se fac similar cu prima: se creează un nod nou ce are ca fii primele *r* noduri din heap, după care acestea sunt extrase și probabilitățile adunate, iar noul nod cu probabilitatea sumă se inserează înapoi în heap. În momentul în care nu mai sunt reduceri de făcut, în heap va fi un singur nod care reprezintă rădăcina arborelui Huffman. Funcția returnează acest arbore Huffman.

* + **build\_encoding\_dictionary(self, root\_node, coding\_dictionary, current\_code)**

Funcția primește ca și parametrii: *root\_node*, de la care începe construcția dicționarului de codare, *coding\_dictionary* unde se vor insera noi cuvinte cod, și *current\_code*, care reprezintă codul pentru nodul curent. Funcția este una recursivă iar condiția de oprire este dacă nodul curent este frunză; în acest caz în dicționar se inserează un nou tuplu de tip: (caracterul nodului, codul curent). Dacă nodul curent nu este frunză, atunci îi voi parcurge fii, iar pentru fiecare fiu voi apela funcția din nou cu parametrii: fiu, dicționar și cod actualizat, unde acest cod actualizat este codul curent la care face append cu un nou simbol, reprezentat de poziția fiecărui fiu în lista de fii ai nodului părinte.

**main.py**

* conține punctul de pornire al aplicației, funcția *\_\_main\_\_*
* funcții:
  + **run\_tests()**

Funcția execută testele din fișierul Tests.py.

* + **\_\_main\_\_()**

Aici se citește un text dintr-un fișier, se construiește dicționarul de codare apelând *get\_encoding\_dictionary* din clasa *Hufman* după care se codează și se decodează textul folosind tabela de codare construită.

**Tests.py**

* Fișierul conține clasa *Tests*, are mai multe teste ce verifică comportamentul obiectelor de tip *Node*, *PriorityQueue* și corectitudinea codării cu algoritmul Huffman.

**Output**

