Compresie de imagine folosind arbori quad liniari și comparație pentru diferite structuri de date

Ghiurțu Andrei și Olaru Karina-Elena
Facultatea de Matematică și Informatică, Universitatea Transilvania, Brașov

Rezumat

Pentru procesarea de imagine se adoptă, în general, algoritmi bazați pe arbori quad. Compresia de imagine este printre cele mai uzuale aplicații ale unui arbore quad. În prezenta lucrare se propubne un algoritm de compresia care are la bază o optimizare a unui arbore quad, cel liniar. În plus, pentru a compara performanța algoritmului de compresie, se folosesc diferite structuri de date și astfel se poate observa utilitatea uneia în detrimentul alteia. Testele au fost realizate pe matrice de pixeli pătratice, de dimensiuni 2^n.

Cuvinte cheie: arbori quad liniari, compresie de imagine, structuri de date

1 Introducere

Un arbore quad este o structură de date arborescentă în care fiecare nod poate avea 0 sau 4 copii, fiecare conținând informațiile unui cadran al părintelui. Într-un arbore quad liniar se rețin doar frunzele și doar cu ajutorul lor se efectuează restul operațiilor. Acest lucru este posibil, deoarece fiecare nod are un cod de n cifre din care poate fi determinată poziția în arbore (spațiu) și nivelul de divizare. Compresia de imagine este o problemă veche, întrucât, dacă pe vremuri stocarea de informație și fișiere avea un raport preț-capacitate foarte mare, în prezent există multe metode ce se folosesc de observațiile și metodele predecesorilor, pentru a se obține imagini clare, dar de o dimensiune cât mai mică. Pentru această lucrare s-au folosit două dintre cele mai uzuale structuri de date, lista dublu înlănțuită și vectorul din STL.

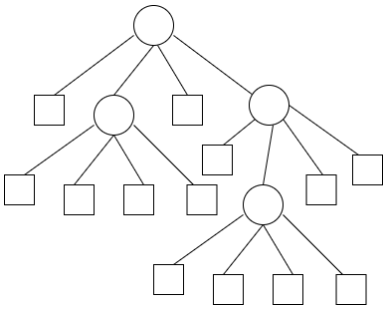


Fig.1 *Arbore quad*

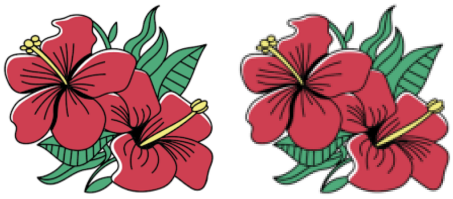
2 Compresia de imagine

Compresia de imagine este o formă de reducere a dimensiunii unei imagini, fără a pierde detalii semnificative din aceasta, chiar dacă se pierde din calitate și, implicit, unele detalii, mai mici. Acest fapt este necesar în ziua de azi pentru a nu încetini încărcarea paginilor web, pentru a facilita transmiterea fișierelor de tip imagini și chiar în algoritmi mai complecși de machine learning. Există mai multe tipuri de compresie de imagine, cea mai naturală fiind reducerea numărului de biți necesar salvării unui pixel, tehnică regăsită sub numele de **uniform quantization**. [Mut] Acest lucru duce la reducerea paletei de culori, scăzând calitatea imaginii. Metoda duce la crearea de artefacte și, implicit, la o imagine rezultată mult mai puțin estetică, dar de o dimensiune considerabil mai mică.

Alte variante mai sofisticate folosesc diferite observații și metode matematice complexe pentru a determina dacă valoarea unui pixel trebuie modificată sau nu, acestea rezultând imagini de dimensiuni mai mici, dar cât mai aproape de original. [Win14]

Metadatele sunt câmpuri de tip cheie-valoare ce descriu un fișier. Imaginile au astfel de metadate pentru a reține informații despre valorile pixelilor și aria pe

care aceștia trebuie afișați. Cu cât este mai scăzut numărul de pixeli diferiți și sunt mai puține intrări ce descriu zona de afișare, cu atât dimensiunea imaginii va scădea. Pe baza acestei observații, a fost creat algoritmul ce urmează a fi descris.



Necompresat - Compresat

3 Arbori quad liniari

Arborii quad liniari folosesc un cod format din n cifre, n fiind rezultatul logaritmului în baza 2 din lungimea unei laturi a imaginii. Matricea de pixeli corespunzătoare imaginii trebuie să fie pătratică, iar dimensiunea unei laturi trebuie să fie putere a lui 2. Având aceste detalii în vedere, considerăm codurile drept chei în găsirea poziției nodului, atât în raport cu părintele său (indicele copilului), cât și raportat la întreaga matrice (al câtelea copil este pe fiecare nivel de divizare). Numerotarea se face în felul următor: după o divizare, copilul din nord-vest primește indicele 0, cel din nord-est 1, copilul din sud-vest primește indicele 2, iar cel din sud-est va fi numerotat cu 3. Astfel, a *i*-a cifră determină poziția nodului pe nivelul i.

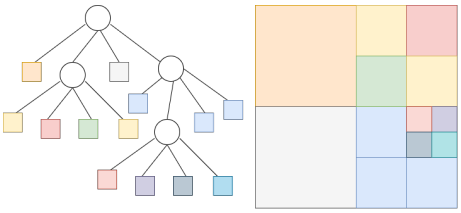


Fig.2 *Arborele quad asociat divizării*

Diferite situații pot necesita diferite moduri de îmbinare a copiilor, și implicit căutarea acestora în lista de frunze. În situația prezentată, algoritmul de îmbinare a constat în alăturarea tuturor copiilor de nivel *i* ai aceluiași nod într-un singur nod frunză de nivel *i-1*, ce pornește de la coordonata *(x, y)*, are lățimea de două ori mai mare decât un copil, iar câmpul culoare este determinat de media culorilor celor 4 noduri îmbinate.

4 Structuri de date

Structurile de date reprezintă diferite metode de stocare a datelor în memoria internă. Există o multitudine de structuri de date, fiecare cu avantajele și dezavantajele ei. Câteva dintre acestea, foarte utilizate, sunt structurile de date liniare, de tipul vectorului și listelor simplu și dublu înlănțuite și arborescente, dintre care merită menționate *map*-ul și *set*-ul. Pentru algoritmul propus s-au efectuat comparații bazate pe combinații de liste dublu înlănțuite și vectori.

Lista dublu înlănțuită stochează elementele astfel încât fecare își cunoaște doar predecesorul și succesorul. În memorie, elementele sunt salvate în locații aleatorii, făcând astfel parcurgerea unei liste mai înceată decât a unei structuri de date contiguă. Totuși această structură de date are un avantaj major în cadrul creării unui arbore quad liniar, înrtrucât acesta necesită inserări și stergeri frecvente, operații care se execută în cadrul unei liste în timp constant, adică complexitate *O(1)*.

Fig.3 *Reprezentarea unei liste dublu înlănțuite*

Vectorul, spre deosebire de listă, stochează informația într-o zonă contiguă de memorie, ceea ce are ca urmare parcurgerea mai rapidă și accesarea elementelor cu ajutorul indecșilor. Cu toate acestea, inserarea și ștergerea în interiorul vectorului implică mutarea pozițiilor elementelor de la poziția în/din care se inserează/șterge până la final, operație care se efectuează în timp liniar (complexitate *O(n)*).

Fig.4 *Reprezentarea unui vector*

Alte structuri de date care au fost luate în considerare pentru acest algoritm au fost coada cu două capete (*deque*), *map*-ul sau *set*-ul. *Deque*-ul preia elemente ale listei și ale vectorului, precum memoria parțial contiguă, accesul prin indecși, iar ștergerea și inserarea la începutul structurii se efectuează în timp constant, dar în mijloc se comportă ca un vector. Cu siguranță această structură de date nu ar fi îmbunătățit algoritmul și nu ar fi fost relevant în testare. *Map*-ul/*set*-ul ar putea fi folosite în cazul căutării vecinilor pe baza codului, unde găsirea și inserarea elementelor s-ar putea face în timp constant, cât timp factorul de încărcare ne-ar asigura evitarea coliziunilor. Totuși acest lucru nu se pretează pe algoritmul utilizat, deci nu a fost luat în considerare pentru a fi testat.

5 Prezentarea algoritmului

Pentru încărcarea și salvarea imaginilor se folosește librăria **OpenCV**.

5.1 Divizarea

5.1.1 Prima variantă

După ce imaginea a fost încărcată în totalitate, este salvată sub forma unei matrice de pixeli RGB (salvați drept *unsigned char*, deci fiecare pixel ocupă 24 de biți), iar aceasta este împărțită recursiv în patru părți, lucru ce asigură că nodurile sunt salvate în ordine într-o listă de pointeri către nodurile create. Fiecare celulă este divizată până când se ajunge la un nod ce conține o singură culoare. Acest lucru s-ar putea realiza și până când se ajunge la culori aproximativ egale, astfel imaginea încărcându-se mai rapid. În acest caz, imaginea compresată are o dimensiune mai mică, dar pierde mai multe detalii.

Fig.5 *Vectorul de noduri după divizare*

```
1  void Image::construct()
2  {
3    float size = log2(pixelMatrix.
       size());
4    if (pixelMatrix.size() < 8 ||
       size != (int)size ||
       pixelMatrix.size() !=
       pixelMatrix[0].size())
5    {
6      return;
7    }
8    codeSize = (int)size;
9    std::vector<int> firstCode(
       codeSize, 0);
10   leafNodes.emplace_back(new Node
       (0, Node::Info(Point2D(0, 0),
        pixelMatrix.size()),
       firstCode));
11   divide(leafNodes.front());
12   std::cout << "Image has been
       loaded\n";
13 }
14
15 void Image::divide(Node*& parent)
16 {
17   if ((parent->info.edge == 1) ||
       (shallDivide(parent) ==
       false))
18   {
19     parent->color = pixelMatrix[
       parent->info.upperLeftCorner.
       y][parent->info.
       upperLeftCorner.x];
20     if (maxLevel < parent->level)
21     {
22       maxLevel = parent->level;
23     }
24     return;
25   }
26
27   //Crearea unui vector din copii
       ce urmeaza a fi introdusi (
       si divizati daca este cazul)
       in lista frunzelor
28   std::vector<std::vector<int>>*
       codes = parent->split();
29   std::vector<Node*> children({
       constructChildNode(parent, (*
       codes)[0]),
       constructChildNode(parent, (*
       codes)[1], parent->info.edge
       / 2), constructChildNode(
       parent, (*codes)[2], 0,
       parent->info.edge / 2),
       constructChildNode(parent, (*
       codes)[3], parent->info.edge
       / 2, parent->info.edge / 2)
       });
30
31   leafNodes.remove(parent);
32   delete codes;
33
34   for (auto& child : children)
35   {
36     leafNodes.emplace_back(child)
       ;
37     divide(child);
38   }
39 }
```

Algoritm 1: Divizarea imaginii

5.1.2 A doua variantă

Această variantă se folosește de proprietatea imaginilor de a avea latura de dimensiune putere a lui 2. Astfel, matricea de pixeli se parcurge direct, în complexitate pătratică ($O(n^2)$). Algoritmul are la bază 3 operații fundamentale:

- construirea codului - această operație constă în interclasarea coordonatelor, scrise în baza 2, și gruparea a câte două cifre pentru a se ajunge la codul în baza 4 corespunzător locației;
- inserarea unui nou nod frunză - constă în parcurgerea frunzelor curente pentru a determina poziția de spargere a nodurilor, lucru ce se întâmplă repetitiv până se ajunge la codul căutat;
- divizarea efectivă - algoritmul se folosește de o structură adițională pentru a reține nodurile active, cu proprietatea că pixelul de pe coloana j are

corespondent activ un nod de pe poziția j/2. Un nod se consideră că nu mai este activ din momentul ajungerii în colțul din dreapta jos, calculat folosind nivelul pe care se află nodul activ și poziția pixelului curent în matrice. Un nod ce nu se află în lista de noduri active și are culoare diferită față de corespondentul său este inserat drept un nou nod activ (deci, un cel mai apropiat corespondent pentru nodurile din raza sa). [Cri]

5.2 Primul algoritm de compresie

Acesta a pus baza algoritmului final. Acesta era mai mult un algoritm de *downsizing*, procentual, dar care realiza și o compresie, în timpul rulării. Modificarea lungimii laturilor nu este un rezultat dorit în urma unei compresii de imagine, ceea ce a dus la modificarea parțială a algoritmului de îmbinare a nodurilor existente în listă și create în urma divizării.

```
1  LQuadTree::Node* LQuadTree::
       createLowerLeafNode(Node* it,
        PERCENTAGE percentage, int
       percPow) {
2     std::vector<int> newCode(
        codeSize - percPow);
3     for (int i = 0; i < newCode.
        size(); ++i) {
4       newCode[i] = it->code[i];
5     }
6     Node* node = new Node(
7       it->level,
8       Node::Info(
9         Point2D(it->info.
        upperLeftCorner.x /
        percentage, it->info.
        upperLeftCorner.y /
        percentage),
10          it->info.edge / percentage
11        ),
12        newCode
13    );
14    node->color = it->color;
15    return node;
16 }
```

Algoritm 2: Îmbinarea nodurilor cu redimensionare

Metoda *createLowerLeafNode* creează un nou nod, utilizând nodul îmbinat în metoda *mergeNodes* descrisă mai jos, dar modifică și dimensiunea acestuia, ceea ce nu se potrivește cu ideea de compresie.

5.3 Al doilea algoritm de compresie

Modificările algoritmului, explicate mai sus, au dus la forma finală a programului de compresie. Acesta constă în eliminarea unui număr de niveluri, specificat de utilizator. Lista de frunze este parcursă secvențial, iar într-un vector se salvează copii ale nodurilor ce au un nivel îndeajuns de mic sau un nod rezultat din îmbinarea tuturor nodurilor obținute în urma divizării părintelui (care s-ar afla pe nivelul maxim de adâncime acceptat). Pentru a evita folosirea unui algoritm recursiv este utilizat un vector de mărime n, folosit cu scopul de a evidenția numărul de ordine al fratelui pe un anumit nivel.

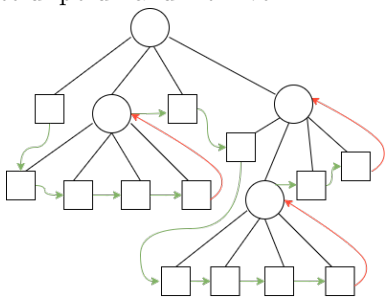


Fig.6 *Parcurgerea secvențială și îmbinarea*

```
1  LQuadTree::Node* LQuadTree::
       mergeNodes(const std::vector<
       Node*>& nodes) {
2     Node* newNode = new Node(nodes[
       VIER - 1]->level - 1, Node::
       Info(nodes[VIER - 1]->info.
       upperLeftCorner, nodes[VIER -
        1]->info.edge * 2), nodes[
       VIER - 1]->code);
3     newNode->color = findAverage(
       nodes);
4     return newNode;
5  }
```

Algoritm 3: Îmbinarea celulelor

Metoda *mergeNodes* se foloseste de ultimul element din vector, acesta fiind de fapt cel din nord-vest.

```
1  std::vector<LQuadTree::Node*>*
       Image::
       createLeavesCompressedImage(
       int noLevelsToDelete) {
2     auto nodes = new std::vector<
       Node*>();
3     std::vector<int>
       currentIndexInLevel(codeSize
       + 1);
4     std::vector<Node*> neighbours(
       VIER);
5
```

```
6    for (const auto& it : this->
       leafNodes) {
7      nodes->emplace_back(new Node
       (*it));
8      if (it->level <= maxLevel -
       noLevelsToDelete) {
9        continue;
10     }
11
12     currentIndexInLevel[it->level
       ]++;
13     for (int level = it->level;
       currentIndexInLevel[level] ==
        VIER && level > maxLevel -
       noLevelsToDelete; --level) {
14       currentIndexInLevel[level]
       = 0;
15       if (level - 1 > 0) {
16         currentIndexInLevel[level
        - 1]++;
17       }
18
19       for (int i = 0; i < VIER; i
       ++) {
20         neighbours[i] = nodes->
       back();
21         nodes->pop_back();
22       }
23       Node* merged = mergeNodes(
       neighbours);
24       for (int i = 0; i < VIER; i
       ++) {
25         delete neighbours[i];
26       }
27       nodes->emplace_back(merged)
       ;
28     }
29   }
30   return nodes;
31 }
```

Algoritm 4: Creare listă noduri pentru imaginea compresată

```
1 std::pair<std::vector<std::vector
    <Pixel>*>*, std::vector<
    LQuadTree::Node*>*> Image::
    compressMatrix(int
    noLevelsToDelete)
2 {
3   auto compressedImage = new std
    ::vector<std::vector<Pixel
    >*>(pixelMatrix.size());
4   for (int i = 0; i <
    compressedImage->size(); ++i)
5     (*compressedImage)[i] = new
    std::vector<Pixel>(
    pixelMatrix.size());
6   std::vector<Node*>* nodes =
    createLeavesCompressedImage(
    noLevelsToDelete);
7
8     //Creare matrice interand
    prin lista de noduri ->
    compressedImage
9
10  return { compressedImage, nodes
    };
11 }
```

Algoritm 5: Compresie

5.4 Salvarea imaginilor

Pentru a testa toate nivelurile de compresie s-a ales introducerea unei structuri repetitive care compresează, pe rând, imaginea originală, reducând 0,1,...,m niveluri, unde m este adâncimea maximă de divizare a imaginii încărcate. Salvarea se face folosind funcția **imwrite**, disponibilă în OpenCV, chiar dacă acest lucru constă în conversia matricei din matrice de pixeli în matricea **Mat**, componentă a librăriei menționate.

```
1 Image* com=image.compress(<
    noLevelsToDelete>);
2 std::string path=std::string(<
    path>.<extenstion>);
3 com->save(path);
```

Algoritm 6: Salvarea imaginilor

5.5 Programul prin perspectiva memoriei

Lucrul cu memoria în C++ se realizează manual, după cum dorește programatorul, astfel punându-se la dispoziție două zone de memorie diferite ca funcționalitate. Pe de o parte se regăsește memoria aflată pe stivă, mai rapidă decât cea aflata pe heap, dar totodată mult mai limitată, și care poate duce la copii inutile ale datelor folosite în exteriorul metodei în care au fost create, în funcție de compilator. Astfel, s-a ales utilizarea memoriei alocate dinamic, pe heap, care deși are un acces mai încet asuprea datelor, ajută în stocarea structurilor de mari dimensiuni, precum o matrice de pixeli. Totodată, în acest mod se asigură evitarea creării unor copii inutile în momentul trecerii de la o metodă la alta, și permițând eliberarea memoriei în momentul în care este necesar.

6 Comparare performanțe

6.1 Pentru primul algoritm de divizare

Comparația următoare este făcută pe aceeași imagine de 512x512 pixeli.

Combinația de structuri de date	Divizare(s)	Îmbinare(s)
Listă-Vector	60.9	0.273
Listă-Listă	63.9	0.332
Vector-Listă	237	0.345
Vector-Vector	242	0.281

Tabelul de mai sus este o privire de ansamblu asupra diferențelor de eficiență dintre structurile de date menționate, liste și vectori, în cadrul divizării și îmbinării. Deși se pot observa diferențe de la listă la listă sau de la vector la vector, atât în divizare, cât și în îmbinare, acestea sunt nesemnificative raportat la durata totală de divizare și putând fi puse pe seama utilizării resurselor de către sistemul de operare. Se poate observa diferența majoră între utilizarea unei liste în detrimentul unui vector (pentru divizare și memorarea frunzelor). Acest lucru se datorează ștergerilor frecvente. O ștergere într-un vector se efectuează în timp liniar (*O(n)*), în timp ce pentru o listă, aceeași operație se efectuează în timp constant (*O(1)*). Pentru îmbinare, totuși, se observă că vectorul este o structură mai eficientă, întrucât îmbinarea implică inserări și ștergeri repetate, de la final, memoria contiguă reprezentând un avantaj în acest caz.

Următoarele comparații s-au realizat pe aceeași imagine, de dimensiuni diferite.

Operație	256x256	512x512	1024x1024
Divizare(s)	2.14	60.9	634
Îmbinare(s)			
Nivel 0	0.104	0.375	1.60
Nivel 1	0.081	0.297	1.21
Nivel 2	0.075	0.273	1.10
Nivel 3	0.073	0.263	1.08
Nivel 4	0.072	0.269	1.05
Nivel 5	0.070	0.259	1.05
Nivel 6	0.070	0.257	1.07
Nivel 7	0.072	0.257	1.05
Nivel 8	0.072	0.260	1.06
Nivel 9	-	0.257	1.04
Nivel 10	-	-	1.05

Tabelul se aplică pentru Listă-Vector. Se poate observa, comparând toate tabelele, că cea mai optimă combinație este cea în care se folosește o listă pentru reținerea nodurilor frunză și un vector pentru îmbinarea acestora. Diferențele, în ceea ce privește gradul de compresie, sunt semnificativ mai mari la primele 3 nivele, după accea timpul fiind aproape similar. Comparând datele din tabel cu testele făcute pentru alte imagini, este evident faptul că timpul de compresie începe să fie aproximativ egal din momentul în care se pierd detalii mari și, implicit, imaginea devine vizibil neclară. Pe măsură ce crește dimensiunea imaginii se observă că algoritmul de divizare este ineficient, ceea ce duce la o creștere semnificativă a timpului de creare a listei de frunze. Totuși, utilizarea unei liste păstrează algoritmul la o durată decentă de încărcare a imaginii, dar de la o dimensiune mai mare de 1024x1024 durata de așteptare ar crește semnificativ. Această combinație de structuri de date este, însă, cea mai eficientă, observându-se din celelalte tabele faptul că durata de așteptare este foarte mare, chiar de la dimensiuni reduse ale imaginii.

Operație	256x256	512x512
Divizare(s)	2.16	63.9
Îmbinare(s)		
Nivel 0	0.129	0.498
Nivel 1	0.103	0.405
Nivel 2	0.094	0.332
Nivel 3	0.093	0.413
Nivel 4	0.091	0.334
Nivel 5	0.090	0.354
Nivel 6	0.091	0.347
Nivel 7	0.092	0.375
Nivel 8	0.091	0.330
Nivel 9	-	0.337

Tabelul se aplică pentru Listă-Listă. Pentru divizare rezultatele sunt aproape similare, marja de eroare fiind nesemnificativă. Cu toate acestea, metoda prezentată se lovește de lipsa memoriei contigue și a optimizărilor prezente într-un vector (de exemplu, pentru alocări și dealocări). Se ajunge la o creștere a timpului de compresie cu 15-20 %.

Operație	256x256	512x512
Divizare(s)	17.2	237
Îmbinare(s)		
Nivel 0	0.115	0.469
Nivel 1	0.097	0.349
Nivel 2	0.092	0.345
Nivel 3	0.089	0.335
Nivel 4	0.086	0.430
Nivel 5	0.088	0.332
Nivel 6	0.088	0.328
Nivel 7	0.087	0.338
Nivel 8	0.087	0.315
Nivel 9	-	0.320

Tabelul se aplică pentru Vector-Listă. Folosirea unui vector pentru divizare duce la multe parcurgeri pentru găsirea elementului căutat, pentru ștergere, cât și parcurgerea necesară pentru efectuarea ștergerii efective. Spre exemplu, pentru un vector cu 90 000 elemente, se parcurg toate pentru ștergerea unui singur element, iar acest lucru se întâmplă de mai multe ori în program. Se poate ajunge

chiar la ordinul milioanelor de elemente. Vectorul este astfel o alegere ineficientă pentru păstrarea frunzelor, fapt ce reiese chiar din testele realizate, unde creșterea este chiar și de 8 ori. Dezavantajele folosirii unui vector pentru divizare se alătură dezavantajele folosirii unei liste pentru îmbinare și rezultă cea mai ineficientă combinație de structuri de date pentru problema expusă.

Operație	256x256	512x512
Divizare(s)	18	242
Îmbinare(s)		
Nivel 0	0.102	0.364
Nivel 1	0.083	0.319
Nivel 2	0.076	0.281
Nivel 3	0.075	0.313
Nivel 4	0.075	0.337
Nivel 5	0.073	0.287
Nivel 6	0.075	0.310
Nivel 7	0.074	0.268
Nivel 8	0.073	0.259
Nivel 9	-	0.270

Tabelul se aplică pentru Vector-Vector. Timpul de compresare al unei imagini ar putea deveni mai mic în ambele situații în care se folosește vectorul drept structură de memorare a nodurilor compresate, dacă realocările vectorului s-ar face mai rar (creșterea/scăderea capacității vectorului să se facă cu un număr multiplu de 4, întrucât în timpul îmbinării frunzelor se șterg câte 4 elemente de la final). Totuși, această metodă devine inutilizabilă din cauza folosirii vectorului pentru divizare, deoarece timpul de așteptare pentru încărcarea imaginii este mult prea ridicat.

6.2 Pentru al doilea algoritm de divizare

Pentru al doilea algoritm s-a observat o creștere semnificativă în durata de așteptare a încărcării imaginii. Acest lucru a îngreunat testarea pe imagini cu dimensiuni mai mari decât 256x256:

Posibile îmbunătățiri ce pot fi aduse algoritmului constau în folosirea unei căutări binare în cadrul funcției de inserare sau folosirea unei tabele de dispersie (*unordered_map*), astfel încât căutarea și ștergerea să se facă în timp aproximativ constant. Această ultimă variantă implică și modificarea funcției de compresie.

7 Concluzii

- Compresia cu ajutorul unui arbore quad este eficientă, mai ales pentru o reducere a unui număr relativ mic de niveluri, pentru a păstra claritatea imaginii, dar suficient de mare astfel încât dimensiunea fișierului să se înjumătățească.
- Pe testele de dimensiuni reduse folosirea unei structuri de date în detrimentul celeilalte pare redundantă, însă pe măsură ce cantitatea de dare crește, se observă diferențe semnificative între diferitele structuri de date folosite, în cadrul acelorași teste.
- Folosirea unui algoritm recursiv pentru divizare simplifică scrierea codului, ajungând astfel la rezultate într-un timp relativ redus, iar timpii de așteptare pentru încărcarea imaginilor sunt suportabili.
- Reducerea numărului de culori diferite duce la un factor de compresie ridicat.

Bibliografie

[Win14] Stefany Franco; Dr. Tanvir Prince; Ildefonso Salva; Charlie Windolf. "Mathematics Behind Image Compression". **in***Journal of Student Research*: 3.1 (2014), **pages** 46–62.

[Cri] Plajer Ioana Cristina. *Linear Quadtrees - suport curs*.

[Mut] Muthukrishnan. *Reduce the number of colors of an image using Uniform Quantization*. URL: https://muthu.co/reduce-the-number-of-colors-of-an-image-using-uniform-quantization/. (accesat: 19.06.2022).