Compressió d'imatges de satèl·lit utilitzant asymmetric numeral systems

Oscar Sanchez Perez
Treball de Final de Grau
Universitat Autònoma de Barcelona
Bellaterra, Barcelona
oscar.sanchezpere@e-campus.uab.cat

***Abstract*—En aquest article es descriu un sistema de codificació d'imatges utilitzant Asymmetric Numeral System(ANS). Primer es farà una petita introducció on s'explicaran conceptes bàsics sobre els ANS i es definirà l'objectiu del projecte. A continuació s'explicarà com funciona la codificació i la descodificació amb el suport d'un petit exemple. Un cop explicats els mètodes principals s'introduirà el Streaming range ANS que serveix per poder codificar valors molt grans sense perdre informació. Tot seguit es mostrarà com es distribueixen les dades al fitxer codificat que es genera i s'explicarà una mica les característiques de les dades que s'han de guardar. També es parlarà sobre alguns dels problemes trobats a l'hora de programar rANS en Java. A continuació es mostraran les diferents experimentacions amb el suport d'alguns gràfics. Aquestes experimentacions s'han fet un cop el software ha estat acabat i mostren comparacions amb altres implementacions i codificadors per entropia. A la part final s'explicarà la metodologia utilitzada durant el desenvolupament, un petit apartat amb els agraïments i una conclusió final del projecte.**

***Index Terms*—rANS, ANS, codificador**

I. Introducció

Els *Asymmetric numeral systems* (ANS) són uns codificadors per entropia que van ser presentats per Jarek Duda a març de 2008 [2]. En l'actualitat, aquests codificadors s'utilitzen a Facebook, en el kernel de Linux, al compressor LZFSE d'Apple, a Dropbox i a altres aplicacions [1]. Les seves característiques principals són que no té pèrdues i que en comparació a mètodes utilitzats prèviament, els ANS podien arribar a ser fins a 30 vegades més ràpids. A més de les característiques mencionades anteriorment, els ANS es poden utilitzar per a la criptografia i la correcció d'errors [2] [3].

En aquest treball s'implementa el rANS [4], la seva característica principal és que els diferents valors que volem codificar tenen probabilitats diferents, és a dir, la quantitat de vegades que es repeteixen els valors depen de la imatge que estem codificant. L'objectiu del treball és programar un codificador rANS en Java per a imatges de satèl·lit. Aquest codificador s'integrarà dins d'un software ja desenvolupat on estan els tests preparats i es poden veure les característiques de les codificacions que es fan. Les proves que es faran són sobre dues imatges, la primera en blanc i negre, és a dir, que l'input serà una sola matriu. La segona imatge serà en color, concretament estarà formada per tres matrius però podria estar formada per més de tres.

El rANS és un codificador al qual li entren dos inputs i surt un sol output, els inputs són:

- El símbol a codificar
- El *state* que tenim en aquest moment

Cada cop que es codifica un valor, com a output, es genera un nou *state* que és un increment de l'anterior. A l'hora de la descodificació es necessita el *state* final i s'ha de fer l'operació inversa per poder recuperar els valors codificats. L'operació de descodificació rep com a input un *state* i retorna:

- Un símbol descodificat
- El nou *state* que necessitem per descodificar el següent símbol

Com a punt final cal remarcar que la descodificació es fa al contrari que la codificació, per tant, el primer valor que es descodifica és l'últim que s'ha codificat.

II. Codificació

En aquest apartat es parlarà sobre la codificació utilitzant rANS. També es mostrarà un petit exemple que s'utilitzarà durant tot el projecte.
Primer de tot cal introduir un petit exemple que s'utilitzarà durant els següents apartats i serà útil per poder donar suport visual a les explicacions que es donaran. La matriu de la Figura 1 és l'exemple que s'utilitzarà.

1	1	2	4
2	2	3	6
3	3	5	7
4	5	6	8

Fig. 1. Matriu d'exemple amb els valors assignats

Aquesta matriu s'ha utilitzat durant el projecte per poder fer un seguiment de les diferents fases de la codificació i

descodificació. Té 16 posicions i cada posició pot prendre valors dins del rang {0,1,2,3,4,5,6,7,8}. Tot i que aquí ens servirà per donar suport visual, aquesta matriu ha sigut molt útil per a poder desenvolupar rANS ja que les imatges ens generen matrius molt grans i és molt dificil trobar errors entre tantes dades.

Per a la codificació primer es necessita la matriu que es vol codificar, que s'anomena residuals (és la que ens dóna el predictor). Un cop es té la matriu hem de calcular dos arrays, el primer és el de freqüències i el segon és el de freqüències acumulades. Amb imatges reals, el primer array conté 256 posicions, que són els possibles valors que pot tenir cada un dels elements de la matriu residual. Aquest array s'anomena F i s'utilitza durant les funcions de codificació i descodificació. Si ens fixem en la matriu d'exemple, aquest array tindrà 9 posicions, de fet F queda com es mostra a la Figura 2.

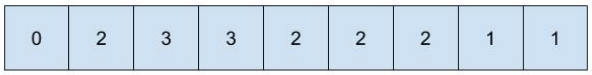

Fig. 2. Array F

Com es pot veure a la imatge, a la posició 0 de l'array hi ha un 0 perquè a la matriu no hi ha cap valor 0. En canvi, a la posició 1, hi ha un dos perquè a la matriu apareix el valor 1 dues vegades.
El segon array es calcula un cop tenim el primer perquè és el de freqüències acumulades. Aquest array s'anomena *Cumulative Distribution Function* (CDF) i el que va fent és acumular la freqüència de l'array F. S'utilitza durant la codificació i la descodificació, a més també és el que s'utilitza per obtenir el símbol descodificat. Seguint la matriu d'exemple i per il·lustrar millor el contingut d'aquest array, a la figura 3 es pot veure el seu contingut.

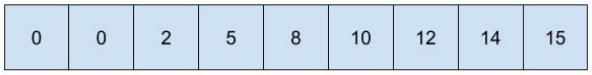

Fig. 3. Array CDF

Si ens fixem en el que fa exactament, per calcular la posició *i* de CDF, se suma el valor de la posició *i*-1 de l'array CDF amb el valor de la posició *i*-1 de l'array F excepte a la posició 0 on directament s'ha de posar un zero.

$$if(i > 0) : CDF[i] = CDF[i-1] + F[i-1]$$

$$else : CDF[i] = 0$$

Quan es tenen els dos arrays el que s'ha de fer és començar a codificar. Primer de tot s'ha de definir un *state* inicial, aquest no hauria de tenir gaire importància perquè la condició de parar el descodificador pot ser la de descodificar tants símbols com teníem al principi. Per a la codificació s'utilitza la següent formula:

$$\textit{nextState} = \frac{state}{sCount}M + CDF[s] + state \bmod sCount$$

on:
state = Estat que hem acumulat fins ara
s = Simbol que ens entra
sCount = F[s]
*M = Nombre d'iteracions (height*width)*

Com es pot veure a la funció, es guarda un nou *state* a la variable *nextState*. S'ha de cridar a aquesta funció per cada símbol que es vol codificar. Per tal que es pugui descodificar correctament s'han de passar les tres dades al descodificador. Les dues primeres dades són les arrays que s'han calculat al principi, són necessàries perquè el descodificador no té accés a la imatge original i és l'única forma d'obtenir-les. L'última dada que necessitarà el codificador és el *state* final. Aquestes tres dades són les que s'han de compartir entre el codificador i el descodificador, per tant són les que s'han d'emmagatzemar en el fitxer que serà la imatge comprimida.

III. Descodificació

Un cop s'han codificat les dades el que volem és recuperar-les completament. Per a recuperar la matriu s'utilitza una funció que fa els càlculs de forma inversa al codificador. L'*input* que rep aquesta funció és un *state* i retorna dos valors, el primer és un nou *state* i el segon és el símbol descodificat. Respecte al *state* el que passa en aquest cas és que cada cop és més petit en comptes d'anar incrementant com en la codificació. En la funció que s'utilitza a la descodificació es necessita tant l'array F com el CDF. Per tant, a la descodificació, el primer que s'ha de fer és llegir del fitxer els dos arrays i el *state* final. Un cop es tenen les tres dades es fan dos càlculs, un per a descodificar el símbol, i l'altre per actualitzar el *state*.
Per descodificar el símbol, el primer que s'ha de fer és el mòdul entre el *state* que tenim i el nombre total de símbols que hi ha. L'operació ens serveix per obtenir la variable slot, si es continua utilitzant l'exemple explicat al codificador, el que ens retorna és un valor que es troba entre 0 i 15. Un cop tenim aquest valor, que anomenarem *slot*, el que s'ha de fer és buscar dins de l'array CDF a quina posició es troba. La forma de trobar a quina posició correspon, donat que a CDF no estan tots els valors, és buscant el primer valor que és més gran que el *slot* i agafar la posició anterior. El símbol descodificat és l'índex de l'array corresponent a la posició trobada. A la figura 4 es pot veure un exemple on el valor de la variable slot és 9. Amb l'exemple de la matriu que hem fet fins ara, al calcular l'array CDF no hi ha cap posició que sigui 9, per tant, el que es fa és buscar el primer valor que sigui més gran que 9. $CDF[5] = 10$ i $CDF[4] = 8$, això vol dir que l'índex 5 és el primer el qual seu valor és més gran que 9. Un cop es té aquesta informació s'ha d'anar a la posició anterior que

equival a CDF[4]. Finalment, el símbol descodificat és aquest índex que hem trobat, per tant, el símbol en aquest cas és 4.

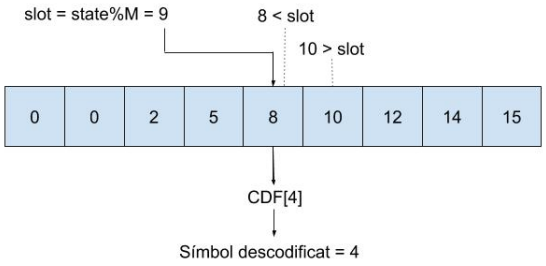

Fig. 4. Esquema que ens mostra com es descodifica el símbol tenint el *state*

Finalment s'ha d'actualitzar el *state*. Per fer això simplement hem d'utilitzar una funció com la de la codificació però que és una mica diferent. La funció és la següent:

$$nextState = \frac{state * sCount}{M} + slot - CDF[s]$$

on:
state = Estat que hem acumulat fins ara
s = Símbol que acabem de descodificar
sCount = F[s]
*M = Nombre d'iteracions (height*width)*
slot = state $\bmod M$

Com podem veure, abans d'actualitzar el *state* s'ha de descodificar el símbol per tal de poder fer les operacions correctament. Un cop es té el nou *state* podem continuar la descodificació fins a fer M iteracions, on M és el nombre de valors de la matriu, és a dir, 16. Finalment, respecte a la descodificació cal tornar a remarcar que els símbols es descodifiquen de forma inversa a com es codifiquen, per tant, l'últim valor que codifiquem és el primer que es descodifica. A l'hora de reconstruir la matriu hem de fer-ho de forma inversa a l'ordre de la codificació. A la figura 5 es veuen els primers vuit elements que es descodifiquen de la matriu d'exemple en l'ordre en el qual es fa.

Fig. 5. Esquema que ens mostra els símbols que es descodifiquen i amb quin ordre ho fan

IV. Streaming rANS

En aquest apartat s'introduirà el *Streaming rANS*, és una funcionalitat totalment necessària si tenim un gran nombre de valors a codificar.
Tot el que s'ha exposat anteriorment funciona correctament però té un problema, durant la codificació el que fem és incrementat un *state*, aquest valor pot arribar a la mida màxima del tipus de variable que s'està utilitzant. Si utilitzem el tipus de variable *int*, es pot arribar fins al *state*:

$$2^{32} - 1 = 4.294.967.295$$

Cal tenir en compte que al final, aquest nombre és només la meitat perquè l'altra part és per representar a nombres negatius. Independentment del màxim numero al que podem arribar, el problema que hi ha és que el *state* incrementa molt ràpidament i les matrius són molt grans. Necessitem un mètode per anar fent el *state* petit quan arribi a un llindar màxim, per solucionar això s'utilitza el que s'anomena *Streaming rANS*. El *Streaming rANS* consisteix a anar dividint entre dos o dit d'una altra forma, fer un shift binari cap a la dreta, el valor que volem fer més petit. Amb la divisió es va fent més petit el *state* però es necessita poder recuperar el *state* que es tenia abans de fer la divisió. Per poder recuperar el *state* el que es fa és guardar el bit que eliminem en fer el shift, és a dir, es guarda el bit menys significatiu.

Respecte a la part del codificador, si ens centrem en la programació, primer de tot s'ha de fer un *while* que miri que el *state* sigui més petit que el llindar màxim establert. En el nostre cas el llindar escollit ha sigut el producte d'un valor que es pot modificar anomenat highLevel multiplicat per F[s]:

$$llindar = highLevel * F[s]$$

De fet, és el mateix que s'ha utilitzat en el següent article [5]. Cal remarcar que és important posar el while perquè si dividim entre dos, no podem estar segurs que el *state* sigui més petit que el llindar, per tant, amb molta freqüència cal fer més d'una divisió quan volem fer més petit el *state*. Respecte a les operacions que es fan, primer hem d'obtenir el bit menys significatiu fent la següent operació:

$$value = state \bmod 2$$

Un cop tenim el valor del bit menys significatiu, l'afegim a un array anomenat *bitstream* i dividim entre dos el *state*. Per acabar, tornem a mirar si el *state* ja és més petit que el llindar i en cas positiu es continua amb la codificació utilitzant el nou *state*, en cas negatiu es continua dividint el *state*. A la figura 6 podem veure una representació gràfica del funcionament.

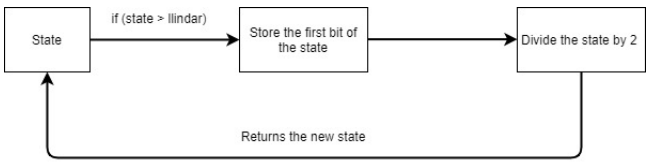

Fig. 6. Esquema on es veu el cicle que es fa quan el state és més gran que el llindar en la codificació

Un cop acabada tota la codificació el que tindrem és l'array *bitstream* de valors que poden ser '1' o '0' i un *state* final. Per a la part de la descodificació es necessiten tots els valors que s'han acumulat, per tant, a part de tot el que s'ha comentat

anteriorment, hem d'afegir l'array al fitxer comprimit (la forma com es guarda s'explicarà més endavant en aquest article).

Respecte a la descodificació, el primer que es necessita és el conjunt de bits que s'han acumulat. Un cop es tenen els bits s'ha de declarar un while semblant al del codificador, aquest ha de mirar que el *state* no sigui més petit que un llindar. El llindar en aquest cas és diferent que l'anterior, està format pel producte del mateix valor highLevel amb el nombre total d'elements a la matriu (també extret de l'article [5]):

$$llindar = highLevel * M$$

En el cas que el *state* sigui més gran que aquest llindar es podrà descodificar correctament, però en el cas contrari, s'ha de multiplicar per dos el *state* i sumar l'últim valor que trobem a l'array de bits. És molt important que se sumi l'últim valor de l'array de bits en comptes del primer perquè estem descodificant de forma inversa i per tant es necessita l'element del final, és a dir, l'últim element que va afegir el codificador. Finalment s'ha d'esborrar el bit que hem llegit de l'array i tornar a mirar si el *state* ja és més gran que el llindar o hem de seguir fent-lo gran. A la figura es pot veure la representació del seu funcionament.

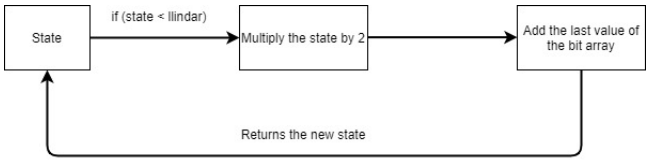

Fig. 7. Esquema on es veu el cicle que es fa quan el state és més petit que el llindar en la descodificació

Com a resum, el *Streaming rANS* és un mètode que ens permet poder codificar una gran quantitat d'elements i a més no ens afegeix cap error, per tant es continua fent la codificació sense cap error. Finalment cal remarcar que el valor que s'anomena highLevel es pot variar i el resultat de la codificació canvia, aquesta experimentació es farà més endavant en aquest article, en el punt dels resultats.

V. Format del fitxer codificat

En aquesta secció es parlarà sobre el format que té el fitxer que genera el codificador. És important que aquest format sigui estàndard perquè el descodificador pugui interpretar correctament les dades. S'indicaran les dades que es guarden amb les seves característiques i es donarà una mica de suport visual per veure com estan organitzades.
Un cop es codifica la imatge es genera un fitxer amb les dades necessàries per poder recuperar la imatge original. Les dades que s'han emmagatzemat al fitxer són:

- F
- CDF
- state
- size
- bitstream

Les dues primeres s'han comentat anteriorment en aquest article, són dos arrays del mateix tamany, cada un té tantes posicions com possibles valors pugui prendre cada element de la matriu que rebem com a input. Són arrays del tipus *int*. En el cas de la matriu que s'ha utilitzat com a exemple durant l'article, cada array té 9 posicions, per tant aquests dos arrays ocupen 9*32 = 288 bits. En el cas de la codificació real d'imatges, com cada element pot prendre 256 valors diferents, podem assegurar que aquests arrays tindran 256 posicions i per tant cada array ocuparà 256*32 = 8192 bits. Aquests arrays són totalment necessaris al descodificador però es podria estalviar enviar el CDF perquè el descodificador el podria calcular abans utilitzant l'array F. Tot i això, al principi es va decidir enviar els dos arrays i, per evitar errors, s'ha mantingut aquest disseny.
A continuació tenim el *state*. Aquest valor també s'ha comentat prèviament en aquest article, és un valor *int* i correspon al valor del *state* que tenim al final de fer tota la codificació. Tant en el cas de la matriu d'exemple com en el cas d'una imatge real ocuparà 32 bits.
Finalment tenim el *size* i el *bitstream*. Comencem parlant del segon perquè el *size* és un valor que necessitem per poder interpretar de forma correcta el *bitstream*. Els elements del *bitstream* es llegeixen de Byte en Byte, per tant no s'obté l'array de bits directament. El que s'ha de fer és llegir-los com a Byte, que és un conjunt de 8 bits, i anar-los interpretant per tal d'obtenir els bits i poder crear l'array que tenia el codificador al final. Aquest últim punt és bastant complex donat que si el Byte que llegim té un '1' a la posició més significativa s'interpreta com un nombre negatiu, per tant s'ha de fer la conversió a positiu i utilitzant el complement a 2 poder interpretar de forma correcta tots els bits. Aquesta última part alenteix bastant l'execució i segurament hi ha formes de llegir els bits més ràpidament. Per acabar tenim el valor *size*, com ja s'ha comentat, el *bitstream* es va llegint de Byte en Byte, per tant ens hem d'assegurar que el nombre de bits en el *bitstream* sigui múltiple de 8. El codificador és l'encarregat de fer tot això, quan escriu l'array de bits, escriu tants 0 com sigui necessaris al final per a assegurar que el nombre de bits escrits és múltiple de 8. El mateix codificador ha de registrar quants zeros ha escrit al final perquè el descodificador pugui ignorar-los a l'hora de la descodificació. Per tant, aquest valor *size* és un *int* que simplement ens indica quants zeros ha escrit el codificador, i per tant, pot contenir qualsevol valor entre 0 i 7.

El format final del fitxer codificat tindrà totes les dades anteriors, però es divideix en dos grans blocs, el primer s'anomena header i té una mida fixa. El header està format per tots els elements excepte el bitstream perquè és el conjunt de dades les quals la seva mida no varia. En el cas de la matriu d'exemple el header ocupa:

$$288 + 288 + 32 + 32 = 640Bits = 80Bytes$$

En el cas d'imatges reals la mida també és constant, la mida del seu header és:

$$8192 + 8192 + 32 + 32 = 16.448Bits = 2.056Bytes$$

Que la quantitat de Bytes sigui constant és molt important per poder interpretar les dades correctament i per tant, l'ordre en el qual es guarden les dades del *header* també és molt important. L'ordenació que es segueix és la que es mostra a la figura 8, on el primer element és l'array F, el segon és l'array CDF, el tercer és el *state* i finalment l'últim element és el *size*. L'ordre es podria canviar i no hi hauria cap problema, simplement s'hauria de canviar l'ordre en el qual el descodificador interpreta les dades.

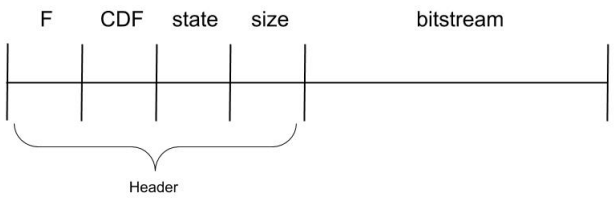

Fig. 8. Esquema que ens mostra es distribueix la informació en el fitxer, aquest esquema no manté proporcions exactes

El segon bloc, que està format només pel *bitstream*, sempre és variant perquè depèn de la matriu que arriba com a input, el que si podem assegurar és que la mida de Bytes serà un nombre enter, o millor dit, la quantitat de bits serà múltiple de 8. Aquest *bitstream* es podia posar al principi del fitxer i posar el header al final, no tindria cap variació, però es va plantejar el disseny des del principi amb la forma de la figura 8 i així és com ha quedat.

VI. RANS A JAVA

La finalitat d'aquesta secció és explicar dos dels problemes que s'han trobat durant la implementació del codificador en Java i que es considera que és important mencionar-los.
El primer problema ha sigut la forma en la qual es guarden les dades. En un principi, i per fer proves es guardava cada bit del *bitstream* com un Byte, per tant, per cada bit que s'havia d'escriure, s'afegien 7 bits de redundància. A la figura 9 es mostra l'exemple anterior.

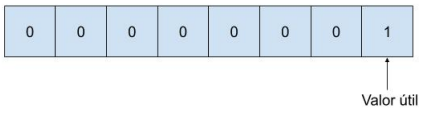

Fig. 9. Imatge on s'utilitza un Byte per representar un sol bit

Un cop tot funcionava es va fer un canvi. Com el tipus de dada més petit és 1 Byte es van fer conjunts de 7 bits en un Byte. Els conjunts eren de 7 per evitar omplir l'últim byte i que s'interpretessin com a negatius. A la figura 10 es mostra un exemple.

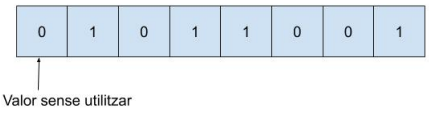

Fig. 10. Imatge on s'utilitza un Byte per representar un sol bit

Això afegia un bit de redundància per cada Byte que s'escrivia al fitxer. Finalment, i amb l'ajut d'una llibreria desenvolupada a la UAB que es diu GiciLibs, vaig aconseguir una funció que s'utilitza per escriure bit a bit dins d'un fitxer i per tant aprofitar totes les dades.

Per acabar parlaré sobre la codificació i descodificació de les matrius. Tal com està programat el software, es crida a la funció de codificar per cada banda que té una imatge. Per exemple les imatges en blanc i negre només tenen una banda, en canvi les que estan a color(RGB) en tenen tres. Les imatges poden tenir diverses bandes però per al correcte funcionament del rANS s'ha decidit que en les imatges que tenen més d'una banda, s'ha de crear una matriu més gran que encapsuli les diferents bandes com si fos una sola matriu. D'aquesta forma només cal cridar un sol cop al codificador i un sol cop al descodificador. Això no té cap influencia en imatges d'una sola banda però les que tenen més d'una incorpora matrius intermediaries que poden alentir l'execució.

VII. RESULTATS

En aquest apartat es mostraran els diferents resultats obtinguts i les diferents proves que s'han fet. Es mostraran dues característiques dels resultats obtinguts. La primera característica serà la mida del fitxer codificat i la segons serà el temps d'execució. Aquestes dades es compararan amb les del codificador aritmètic que havia implementat en el software on s'ha afegit el *rANS*. Es donaran quatre resultats de cada característica perquè hi ha dos codificadors i cada codificador s'ha provat amb dues imatges diferents. Les imatges que s'han utilitzat són diferents, la primera és en blanc i negre, per tant només té una matriu, la segona imatge és en color i aquesta esta formada per tres matrius. La imatge en blanc i negre ocupa originalment 5.242.881 Bytes i la imatge en color ocupa 15.728.640 Bytes. Per tal d'entendre correctament el temps d'execució cal aclarir que s'ha executat a una màquina virtual amb 4 processadors i 6349 MB de memòria. També es parlarà sobre els diferents resultats del codificador rANS utilitzant diferents llindars (els utilitzats al *streaming rANS*) per veure si aquests resultats tenen grans variacions o es mantenen parells. Finalment es mostrarà una comparativa entre el codificador rANS desenvolupat i el codificador rANS escrit en C [6].

Comencem amb la mida del fitxer. En el codificador rANS i amb la imatge en blanc i negre la mida del fitxer resultant és de 3.041.245 Bytes, això significa que la mida original s'ha reduït en 2.201.636 Bytes, i per tant, hi ha una

reducció final del 42% aproximadament. Per un altre costat tenim el codificador aritmètic que ja estava implementat, en aquest cas i amb la mateixa imatge el fitxer resultant té una mida de 3.080.895 Bytes, que vol dir que s'ha reduït en 2.161.986 Bytes la mida i per tant, el fitxer s'ha reduït en un 41,41%. En aquesta primera comparativa, tot i que no hi ha una gran diferencia, el codificador rANS és el que ha obtingut millors resultats, al gràfic de la figura 11 es poden veure els resultats més clarament.

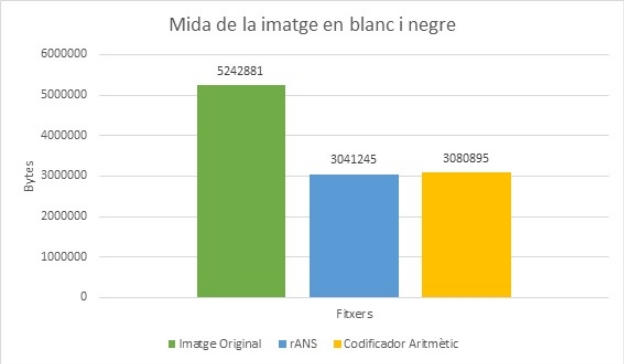

Fig. 11. Gràfic que mostra la mida dels fitxers en les proves a la imatge en blanc i negre

Ara comparem els dos codificadors amb la imatge en color que ocupa pràcticament el triple que l'anterior. Un cop codificada amb rANS, el fitxer resultant ha ocupat 12.514.727 Bytes, per tant el fitxer original s'ha reduït en 3.213.913 Bytes que significa una reducció del 20,43%. El codificador aritmètic per un altre costat, ens genera un fitxer de 12.578.793 Bytes, que són 3.149.847 Bytes menys que l'original i per tant el fitxer s'ha reduït un 20,03%. En el cas d'aquesta imatge, que és significativament més gran podem veure com la codificació ja no és tan eficient com el cas de la imatge en blanc i negre. A la figura 12 podem veure els resultats en forma de gràfic.

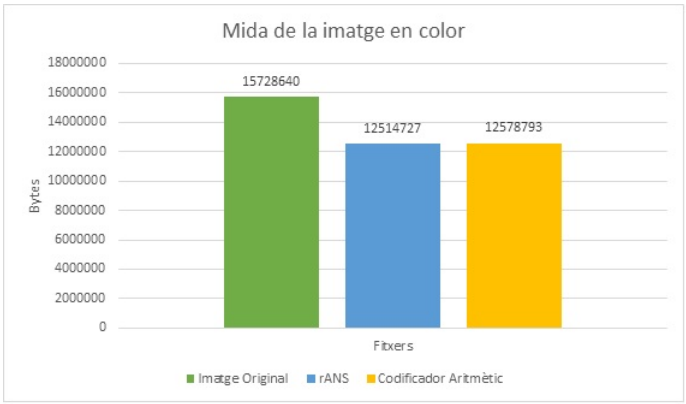

Fig. 12. Gràfic que mostra la mida dels fitxers en les proves a la imatge en blanc i negre

Respecte a l'efectivitat del codificador, rANS continua sent millor però en aquest cas la reducció dels dos codificadors és més semblant, per tant, podríem arribar a la conclusió que en imatges molt grans rANS podria no ser millor que el codificador aritmètic. Tot i que la conclusió anterior pot ser certa, fent una mica de recerca sobre imatges satèl·lit [7] podem trobar que les imatges no solen ocupar més de 3.000.000 Bytes, per tant, rANS ens aporta una millor compressió si ens referim a la mida del fitxer resultant.

A continuació es parlarà sobre el temps d'execució. Si s'executa el software sobre la imatge en blanc i negre codificant amb rANS triga 28,16 segons. En canvi, l'execució del codificador aritmètic sobre la mateixa imatge només triga 18,06 segons. Per tant, el que podem dir és que el codificador rANS és 1,56 vegades més lent que el codificador aritmètic. Les proves fetes a la imatge amb color han donat un temps d'execució amb la codificació rANS d'1 minut i 49,53 segons. Però amb la mateixa imatge el codificador aritmètic només ha trigat 53,82 segons. Aquests temps indiquen que el codificador rANS ha trigat 2,04 vegades més que el codificador aritmètic. La figura 13 ens mostra més clarament les dades anteriors.

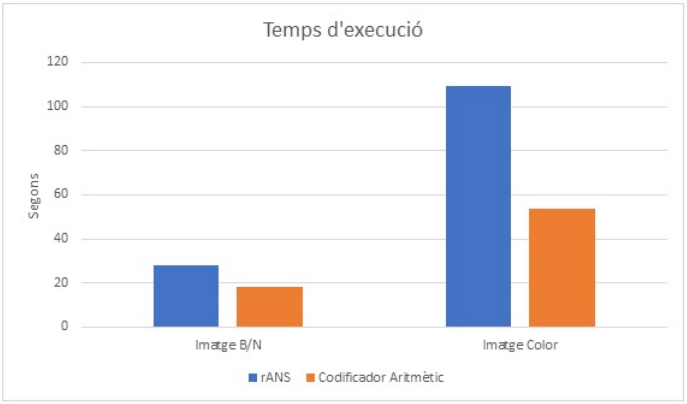

Fig. 13. Gràfic que mostra el temps que triga cada codificador en cada una de les imatges

Com a conclusió en les proves fetes a la part dels temps podem dir que el codificador rANS és molt més lent, de fet, com més gran sigui la imatge més es alenteix. Aquesta lentitud és deguda al fet que algunes parts de la implementació utilitzen matrius o arrays molt grans que s'utilitzen com intermediaries entre operacions. Un exemple de les arrays mencionades anteriorment és el array de bits, aquest array va acumulant els bits de la fase del Streaming per tal de poder recuperar el *state*, aquest array en comptes de generar-lo i després haver d'escriure els seus elements un a un, es podria anar escrivint directament durant l'execució de la codificació i això faria més ràpida l'execució.

A la figura 14 es pot veure una gràfica que ens indica a l'eix X la mida de la matriu. En el cas de la imatge en blanc i negre la matriu és de 5.242.880 elements i en el cas de la imatge en color és de 15.728.640 elements. A l'eix Y ens mostra el temps d'execució. Amb aquest gràfic, tot i que no hi ha moltes dades per poder dibuixar-lo correctament, podem veure que el codificador aritmètic incrementa el temps amb

un pendent petit però el rANS incrementa amb un pendent molt més gran, per tant, com més gran sigui la imatge, pitjor és el codificador rANS respecte a l'aritmètic.

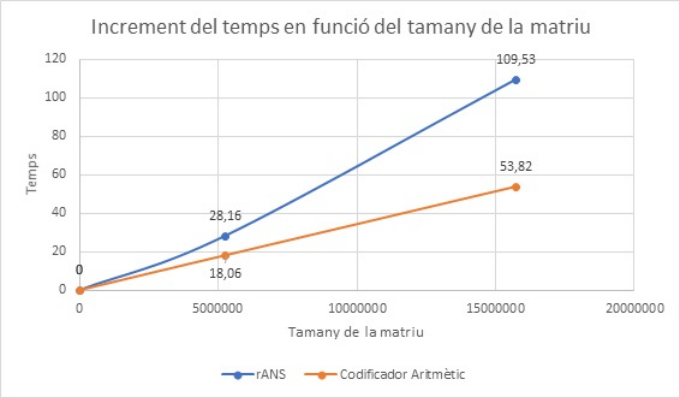

Fig. 14. Gràfic que mostra el temps en funció de la mida de la matriu que rep el codificador com a input

Tot i que els resultats en aquest apartat són força dolents, des del principi es va dir que el tems d'execució no era el punt principal del treball, a més a més, el temps tampoc és excessivament gran donat que com ja s'ha comentat prèviament, les imatges satèl·lit tendeixen a ocupar uns 3.000.000 Bytes [7], tot i que probablement hi ha de mides més grans.

Ara es mostraran les proves que s'han fet variant el valor highLevel, que és un dels dos valors que determina el llindar durant el *streaming rANS*. Abans de mostrar els resultats cal especificar que s'ha decidit que aquests valors siguin potències de dos on la potència de dos més baixa que s'ha escollit és 2^3. A més, és important destacar que totes les proves que s'han comentat abans d'això han sigut assignant 2 al cub a la variable highLevel.
La següent taula mostra els resultats obtinguts amb la imatge en blanc i negre:

highLevel	Mida del Fitxer	Temps d'execució
2^3	3041245 Bytes	28,17 segons
2^4	3038882 Bytes	33,30 segons
2^5	3037784 Bytes	30,74 segons
2^6	3037643 Bytes	38,83 segons
2^7	3037678 Bytes	30,24 segons

La seguent ens mostra el mateix que la primera pero sobre la imatge a color:

highLevel	Mida del FItxer	Temps d'execució
2^3	12578793 Bytes	1 minut i 49,85 segons
2^4	12508710 Bytes	2 minuts i 10,01 segons
2^5	12506955 Bytes	1 minut i 43,52 segons
2^6	12506782 Bytes	1 minut i 45,08 segons

Comencem parlant sobre la primera taula. A partir de 2^8 ja no funciona, això és degut al fet que el llindar també depèn d'altres valors i si escollim 2^8 pot haver-hi overflow i no es comprova correctament el llindar. Respecte als resultats el que sembla és que com més alt millor, això és degut al fet que com més gran sigui el llindar, menys vegades s'ha de dividir entre 2 el *state* i per tant tampoc s'han d'escriure tant uns i zeros al fitxer. Malgrat això es pot veure que amb el valor 2^7 sembla que augmenta el nombre de bits, això és probablement perquè canviar el valor highLevel fa que els *states* siguin diferents i pot passar que la combinació de *states* faci que s'hagin d'escriure uns pocs valors de més. A més sembla que amb valors més grans hi ha poca variació en la mida de l'output però sí que sembla recomanable agafar com a mínim el 2^5 perquè sí que hi ha una millora força bona. Si ens fixem en el temps, no sembla que hi hagi cap patró que ens indiqui com millorar-lo. Crec que el temps varia d'aquesta forma per culpa de l'array de bits. Això ja s'ha comentat en un punt anterior però en descodificar es fa de Byte en Byte i poden coincidir molts nombres negatius, això pot alentir l'execució.
En la segona taula podem veure que hi ha una fila menys, això és el mateix fenomen que s'ha explicat prèviament, amb el valor 2^7 hi ha overflow a l'hora de comprovar el llindar i per tant s'afegeixen errors. A més a més, com més gran és el highLevel, podem veure que també millora la mida del fitxer. Si mirem el temps passa el mateix que amb la primera taula, no segueix cap patró. Per la mida de les diferents proves, respecte a la mida el millor és 2^6 però també podem veure que on realment hi ha una bona diferència la trobem en el pas de 2^3 a 2^4.
Com a conclusió, la millor opció és seleccionar highLevel amb el valor 2^6 perquè ens aporta la millor compressió en els dos casos. Però, com es pot veure, utilitzar 2^4 o 2^5 són opcions pràcticament iguals i ens aporten més fiabilitat en cas que es volgués introduir un input més gran. Fins i tot, es podria fer que el highLevel fos variable i intentar agafar la potència de dos més gran que ens asseguri que no hi hagi overflow. El problema d'aquesta última opció és que s'hauria d'afegir un valor al header però possiblement seguiria sent útil. Per acabar, cal remarcar que tot i que hi ha petites variacions, no hi ha cap millora que pugui ser significativa, és a dir, amb els resultats que tenim, no hi ha cap solució que sigui molt bona, per tant, la meva opinió és que els valors més petits ens aporten més fiabilitat.

Per acabar aquesta secció es farà una petita comparativa entre el codificador rANS escrit en C i el que s'ha desenvolupat en Java. El codificador en C té dos possibles modes, el mode 'o0' i el mode 'o1'. Si parlem del temps d'execució, els dos modes en codificar i descodificar triguen aproximadament un segon. Cal tenir en compte que l'execució en Java no és només codificar i descodificar però el temps d'execució és molt pitjor el de Java. Per un altre costat, el codificador de Java obté millors resultats en la compressió. Si ens fixem en la imatge

en blanc i negre tenim els següents resultats:

highLevel	Mida del Fitxer
Java	3041245 Bytes
o0	4687158 Bytes
o1	3337338 Bytes

Com es pot veure, amb la imatge en blanc i negre, el codificador en Java obté millors resultats en comparació a qualsevol dels modes que té el de C. La següent taula mostra el mateix però respecte a la imatge en color.

highLevel	Mida del Fitxer
Java	12578793 Bytes
o0	14413843 Bytes
o1	1105776 Bytes

La taula de la imatge en color és una mica diferent, en aquest cas el mode 'o1' del codificador en C és millor que la del codificador desenvolupat en Java.
Com a conclusió, la versió desenvolupada en Java té problemes de velocitat i, a més a més, amb imatges grans hi ha marge de millora.

VIII. Metodologia

La metodologia que s'ha seguit ha sigut Kanban. S'ha utilitzat l'eina Trello i s'han afegit les tasques necessàries en les diferents etapes. Respecte a com s'ha distribuït el treball, ha sigut tot força seqüencial des del punt de vista que fins que no s'acabava una funcionalitat, no es podia passar a la següent perquè es depenia de l'anterior. El treball en general s'ha dividit en dues grans fases, la primera va ser la d'aprendre com funcionava el software, es va utilitzar diferents fonts d'informació i es va intentar passar un codificador ja implementat en C [6] a Java. Aquesta fase no va anar gaire bé perquè era bastant confús el funcionament del codificador en C i es va decidir crear una implementació des de zero utilitzant les fórmules mencionades al paper original de Jarek Duda [2] i amb tot el material de l'article escrit per Kedar Tatwawadi [5].

IX. Agraiments

M'agradaria agrair al meu tutor del treball de recerca Joan Bartrina l'ajut que m'ha donat. M'ha proporcionat documents per aprendre més sobre el funcionament de la compressió d'imatges i enllaços que m'han sigut molt útils per al desenvolupament del codificador rANS. A més, sempre que li he demanat tutories per resoldre dubtes ha estat disponible en un període de temps molt curt tot i l'estat excepcional en el qual s'ha desenvolupat aquest semestre el TFG.

X. Conclusión

En aquest article s'ha mostrat com funciona un codificador rANS utilitzant un exemple per intentar mostrar-ho millor. També s'ha explicat un mètode que s'utilitza per poder mantenir el *state* dins del llindar. Per un altre costat s'ha mostrat una possible forma de guardar les dades codificades dins d'un fitxer. A part s'han explicat alguns dels problemes trobats durant tot el desenvolupament amb Java relacionats amb rANS. Per acabar s'han fet una sèrie de proves per poder saber la qualitat del codificador.

References

[1] Andrew Polar, 'Asymmetric Numeral System'. [Online]. Available: http://www.ezcodesample.com/abs/abs_article.html
[2] Jarek Duda, 'Asymmetric numeral systems.'. [Online]. Available: https://arxiv.org/pdf/0902.0271.pdf
[3] Ahmad Mozaffar, 'Implement Symmetric And Asymmetric Cryptography Algorithms With C'. [Online]. Available: https://www.c-sharpcorner.com/article/implement-symmetric-and-asymmetric-cryptography-algorithms-with-c-sharp/
[4] Wikipedia, 'Asymmetric numeral systems'. [Online]. Available: https://en.wikipedia.org/wiki/Asymmetric_numeral_systems
[5] KEDAR TATWAWADI, 'What is Asymmetric Numeral Systems?'. [Online]. Available: https://kedartatwawadi.github.io/post–ANS/
[6] Jan Voges, 'rans'. [Online]. Available: https://github.com/voges/rans
[7] NASA, 'ASTER Most Popular Images'. [Online]. Available: https://asterweb.jpl.nasa.gov/toplist.asp