Huffmanovo kodiranje ternarnih izvora informacija
Teo Samaržija

Uvod

Digitalni izvori informacija nerijetko daju informacije u obliku koji ih čini znatno dužima nego što je potrebno, što otežava njihov prijenos ili spremanje, a ponekad i obradu. Huffmanov algoritam jedan je od najjednostavnijih algoritama sažimanja bez gubitaka, on preoblikuje digitalne informacije u oblik koji je kraći, a koji još uvijek sadrži istu informaciju, te se može preoblikovati u originalni oblik. Huffmanov algoritam prvi je opisao i na tadašnjim računalima isprogramirao David Albert Huffman 1952. godine.

Kao jedan brzi primjer kada to može biti korisno, recimo da na računalo želimo pohraniti veliku količinu teksta na nekom jeziku. Kada netko tipka tekst na tipkovnici, tipkovnica računalu općenito šalje 1 *byte* (simbol od osam znamenki, od kojih je svaka ili nula ili jedinica) za svaku tipku koju korisnik pritisne. No, to nije optimalan način zapisivanja teksta na računalu. Neka se slova pojavljuju znatno češće od drugih slova. Recimo, na engleskom jeziku, najčešće slovo jest *e*, jer se ono koristi i za glas *e* (koji je u engleskom jeziku čest) i kao tiho *e* na kraju riječi. Tim se tihim *e* na kraju riječi u staroengleskom označavao dugi naglasak na zadnjem slogu u riječi, pravopisno pravilo koje je preuzeto iz francuskog. Staroengleski dugi samoglasnici u današnjem su engleskom većinom prešli u diftonge (nizove od dva samoglasnika koje izvorni govornici percipiraju kao jednu cjelinu), recimo, *like* se na staroengleskom izgovaralo *lîk*, dok se danas izgovara *lȁik*, no ta se riječ nastavila pisati isto. Ustvari, slovo *e* se u tekstu na engleskom jeziku u prosjeku pojavljuje oko 170 puta češće nego slovo *z* (slovo *z* je u engleskom jeziku iznimno rijetko, jer se glas *z* u engleskom jeziku obično piše *s*)[1]. Slično vrijedi i za druge jezike, recimo, na hrvatskom jeziku najčešće slovo jest *a* (samoglasnici su, naravno, češći od suglasnika, inače jezik ne bi bio izgovorljiv), i ono se pojavljuje 50-ak puta češće nego slovo *đ* (jer glas koji ono predstavlja nastaje isključivo jotacijom, kad su se *d* i *j* našli jedan pored drugoga, što je rijetko)[2]. Pa onda bi, ako zapisujemo engleski tekst u memoriju računala, imalo smisla slovo *e* zapisivati kraćim nizom nula i jedinica nego slovo *z*. I na toj se ideji zasniva Huffmanov algoritam.

Prvi sličan algoritam sažimanja informacija bio je Shannon-Fanov algoritam, objavljen 1948. godine. Huffmanov algoritam je malo sporiji od njega (što za današnja računala nije važno, jer su ona beskrajno brža od računala koja su postojala 1952. godine), no on ima svojstvo da garantira optimalan rezultat pod pretpostavkom da se znakovi iz izvora informacija pojavljuju neovisno jedan o drugome[3].

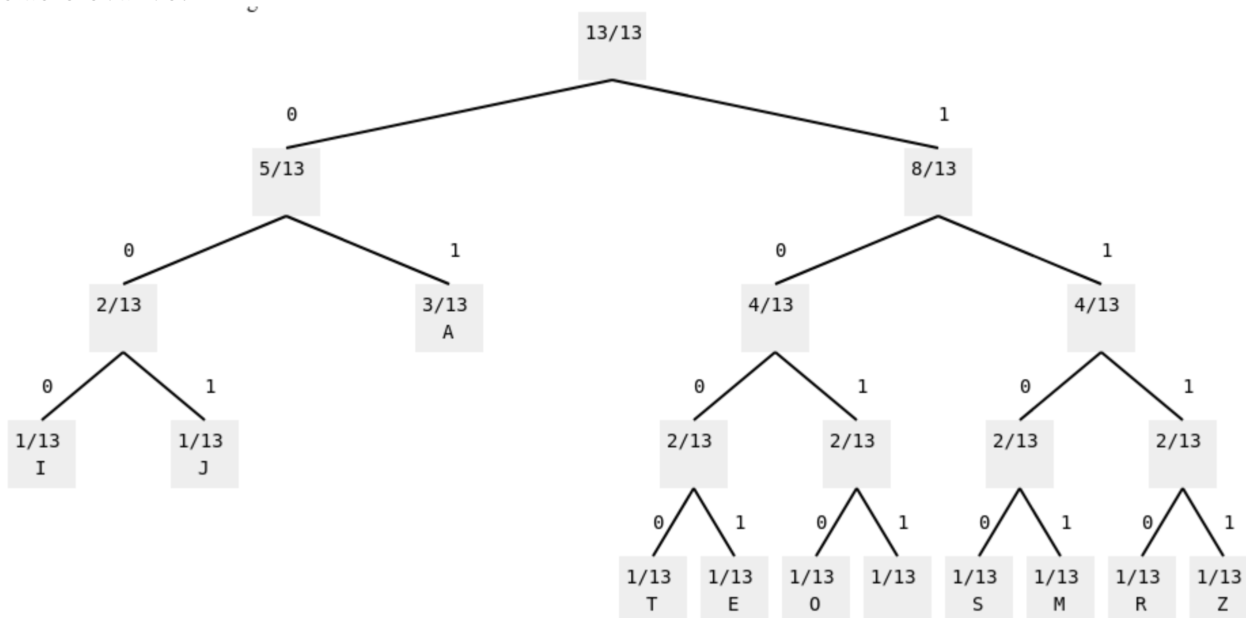
To svojstvo neovisnosti slova jedno o drugome, naravno, za ljudske jezike nije dobra aproksimacija. Najočitiji primjer, ako se na engleskom jeziku pojavi slovo *q*, možemo biti gotovo sigurni da je iduće slovo *u*, jer se slovo *q* u engleskom jeziku koristi gotovo isključivo u digrafu *qu* koji označava glas k^w. Slično u hrvatskom jeziku, nakon slova *p* obično slijedi ili samoglasnik ili bezvučni suglasnik kao što je *t* ili *s*, rijetko kada slijedi *b* ili *d*. I najčešće kombinacije od dva slova u hrvatskom jeziku su *je*, *na* i *ra*[4]. Bez obzira na to, Huffmanov algoritam može biti brz i učinkovit način da se smanji duljina zapisa teksta u memoriji računala.

Binarni Huffmanov algoritam

Huffmanov algoritam relativno se lagano isprogramira u skriptnim programskim jezicima, i u ovom ću seminaru napisati kako sam ga implementirao u programskom jeziku JavaScript. Računalo prvo treba proći kroz tekst te zabilježiti koji se simboli (slova, brojevi, interpunkcije...) pojavljuju u tekstu i koliko se često koji simbol pojavljuje. Te simbole, zajedno s podatkom koliko se često koji simbol pojavljuje, treba spremiti kao strukture, a memorijske adrese gdje se nalaze te strukture treba

1 Podatak preuzet s http://pi.math.cornell.edu/~mec/2003-2004/cryptography/subs/frequencies.html
2 Podatak preuzet s https://www.sttmedia.com/characterfrequency-croatian
3 Dokaz imate ovdje, to je članak gdje je Huffman prvi put opisao taj svoj algoritam i uvjerio druge informatičare da ga koriste: http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf
Iskreno, ne razumijem ga niti ja.
4 Podatak preuzet s https://www.sttmedia.com/syllablefrequency-croatian

spremiti u jedan niz. To je lagano isprogramirati u bilo kojem često korištenom programskom jeziku, uključujući i asemblerskom jeziku. Koraci koji slijede teži su za isprogramirati u niskim programskim jezicima. Sada trebamo pronaći dva simbola najmanje čestoće pojavljivanja i obrisati njihove memorijske adrese iz niza (ili, ako imamo memorije na bacanje, možemo jednostavno označiti u tim strukturama da ih više nećemo koristiti), a zatim u niz ubaciti memorijsku adresu novostvorene strukture u kojoj piše zbroj njihovih čestoća te gdje se one nalaze u memoriji. Drugim riječima, u niz trebamo ubaciti stablastu strukturu na čijem je vrhu podatak o zbroju čestoća (frekvencijā), a djeca su joj čvorovi koji sadrže podatke o pojedinim simbolima. I taj korak trebamo ponavljati sve dok se u nizu ne bude nalazila memorijska adresa samo jednog čvora, to je onda korijenski čvor u kojem piše frekvencija 100%. Zatim pokrenemo rekurziju iz tog korijenskog čvora koja se grana na lijevo i desno dijete svakog čvora koji ima djecu (dakle, sve čvorove u stablu osim onoga koji predstavljaju simbole), za svako desno dijete upisuje da je njegov Huffmanov kod jednak Huffmanovom kodu njegovog roditelja (Huffmanov kod korijenskog čvora je prazan niz znakova) s tim što mu se na kraju doda jedinica, a za svako lijevo dijete to isto osim što mu se na kraj doda nula. Nakon toga se svi simboli u ulaznom nizu znakova zamjenjuju odgovarajućim Huffmanovim kodom, te se tako dobiva Huffmanovo kodiranje tog niza znakova.
Recimo da je ulazni tekst `TEO SAMARZIJA`. On se sastoji od jedanaest simbola (i razmak je simbol), od kojih se svi pojavljuju samo jednom, osim simbola `A` koji se pojavljuje tri puta, dakle duljina mu je 13. Web-aplikacija[5] koju sam isprogramirao tvrdi da je odgovarajuće Huffmanovo stablo ovakvo:



Dakle, program je prvo odabrao znakove `I` i `J`, oba frekvencije 1/13, te ih spojio u jedan čvor frekvencije 2/13. Zatim je odabrao znakove `T` i `E`, te ih ponovno spojio u jedan čvor frekvencije 2/13. To je ponovio sa znakom `O` i razmakom, zatim sa znakovima `S` i `M`, te konačno sa znakovima `R` i `Z`. Tada više u nizu nije bilo neiskorištenih čvorova frekvencije 1/13, nego su najmanje česti čvorovi u nizu bili onaj 2/13 (spoj znakova `I` i `J`) i znak `A`, frekvencije 3/13, te njih spoji u jedan čvor frekvencije 2/13+3/13=5/13. U nizu je tada bilo preostalo još 4 čvora s frekvencijom 2/13, koji se spajaju prvo u dva čvora frekvencije 4/13, a zatim u jedan čvor frekvencije 8/13. Konačno, čvorovi frekvencije 8/13 i 5/13 spajaju se u korijenski čvor s frekvencijom 13/13 ili 100%. Zatim se pokreće rekurzija iz tog korijenskog čvora i svaki put kad ide lijevo, čvoru u kojeg je došla postavlja Huffmanov kod u Huffmanov kod čvora iz kojeg je došla plus nula, a kad se grana desno radi to isto, samo što dodaje jedinicu umjesto nule. Recimo, da dođe do slova *T*, ide desno u čvor 8/13 (Huffmanov kod je sada 1), zatim lijevo u čvor 4/13 (Huffmanov kod je sada 10), zatim lijevo

5 Dostupna je ovdje, može se pokrenuti u modernim internetskim preglednicima, a i u Internet Exploreru 11: https://flatassembler.github.io/huffman.html

u čvor 2/13 (Huffmanov kod je sada 100), te opet lijevo da dođe do `T` (Huffmanov kod je sada 1000). Tablica s Huffmanovim kodovima je, prema web-aplikaciji, ovakva:

symbol	frequency	Huffman code	equal-length code
Z	1/13	1111	0000
R	1/13	1110	0001
M	1/13	1101	0010
S	1/13	1100	0011
	1/13	1011	0100
O	1/13	1010	0101
E	1/13	1001	0110
T	1/13	1000	0111
A	3/13	01	1000
J	1/13	001	1001
I	1/13	000	1010

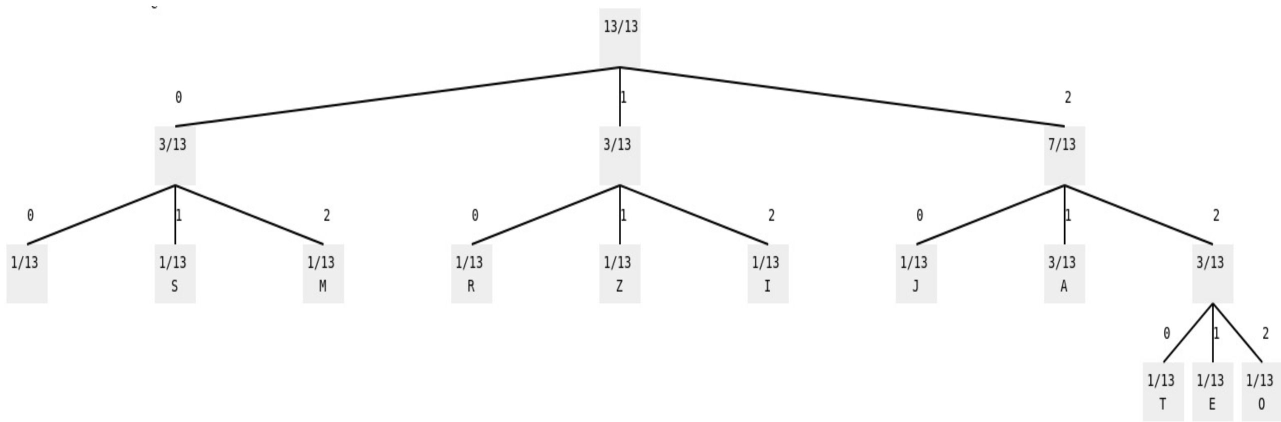
I Huffmanov kod stringa (niza znakova, *string* na engleskom, izvan informatike, znači *lanac*) je ovakav: `1000 1001 1010 1011 1100 01 1101 01 1110 1111 000 001 01` (računalu, naravno, ne trebaju razmaci ovdje). Da bismo ga dekodirali, treba nam tablica. Jedan od većih problema s algoritmima sažimanja jest upravo to, oni osim što sažimaju informaciju, dodaju *overhead* (*over* znači *preko*, a *head* znači *glava*), koji može biti veći od redukcije (smanjenja) veličine sažete informacije. Overhead je u ovom slučaju ova tablica. Prosječna dužina simbola u Huffmanovom kodu ovdje jest 3.38 bitova (jedan bit je najmanja moguća količina informacije, ona koja je potrebna da se predstavi razlika između nule i jedinice). Shannonova entropija jednog simbola u stringu `TEO SAMARZIJA` iznosi 3.33 bita. To je dobro, Huffmanov kod je ovdje veoma blizu teorijskom minimumu (digitalni signal[6] uvijek ima entropiju veću od teorijskog minimuma potrebnog da se prenese informacija). Za usporedbu, da nismo koristili nikakav algoritam sažimanja, jedan simbol bio bi velik 4 bita. Naravno, Huffmanov algoritam ne daje uvijek rezultate blizu Shannonovoj entropiji. Recimo, za string `ABBBBBBBBB` Shannonova entropija iznosi 0.47 bita, dok Huffmanov kod iznosi 1 bit. To je više od dvostruko više.

Primjena Huffmanova kodiranja na ternarni kod

U digitalnoj elektronici općenito postoje dva jasno definirana stanja, stanje visokog napona koje se označava jedinicom te stanje niskog napona koje se označava nulom, napon između tog dvoje signalizira ili da prelazimo iz jednog stanja u drugo ili da je došlo do pogreške. No, ponekad, pogotovo u starijoj digitalnoj elektronici, postoje tri jasno definirana stanja, koja se označavaju s 0, 1 i 2. Ternarna digitalna elektronika se često koristila 1960-ih i 1970-ih u Sovjetskom Savezu, jer je trošila manje struje (tada su se u digitalnoj elektronici često koristili bipolarni tranzistori, a za MOSFET tranzistore, kakvi se danas koriste, binarna logika je daleko efikasnija), a kompatibilnost s američkim računalima, iz političkih razloga, nije bila previše važna. Postavlja se pitanje je li bolje pretvoriti ternarni (s tri moguće vrijednosti) signal u binarni (s dvije moguće vrijednosti), pa zatim primijeniti Huffmanov algoritam, ili je bolje primijeniti Huffmanov algoritam izravno na ternarni signal. Huffmanov algoritam je relativno lagano modificirati da proizvodi ternarni kod: kada bismo tražili dva najmanja čvora u nizu i spajali ih, tražimo tri. Problem pri programiranju mogla bi stvarati činjenica da će se ponekad dogoditi da se u nizu nekad nalaze točno dva čvora, pa će pokušaj da spojimo tri najmanja čvora izazvati dereferenciranje nultog pokazivača. No, u modernim programskim jezicima to je trivijalno ispraviti. Prema web-aplikaciji, dijagram ternarnog Huffmanovog koda stringa `TEO SAMARZIJA` izgleda ovako:

---

6 Ovdje *signal* znači jednostavno *ono što nosi informaciju* (latinski *signum* znači *znak*), ne mora nužno biti elektromagnetsko zračenje.



Tablica izgleda ovako:

symbol	frequency	Huffman code	equal-length code
O	1/13	222	000
E	1/13	221	001
T	1/13	220	002
A	3/13	21	010
J	1/13	20	011
I	1/13	12	012
Z	1/13	11	020
R	1/13	10	021
M	1/13	02	022
S	1/13	01	100
	1/13	00	101

Shannonova entropija stringa je, naravno, još uvijek 3.33. Jedna ternarna znamenka nosi $\log_2(3)=1.58$ bita informacije. Budući da imamo 11 različitih simbola, treba nam $\text{ceil}(\log_3(11))=3$ ternarne znamenke za svaki simbol ako ne koristimo algoritam kompresije (*ceil* u informatici znači *najmanji cijeli broj veći od argumenta*, od engleskog *ceiling*, *strop*, od latinskog *celare, pokriti*), dakle ukupno 3*1.58=4.74 bita. Ako koristimo ternarni Huffmanov kod, dobivamo 3.54 bita po simbolu. Dakle, koliko god to bilo protiv intuicije, izgleda da je ternarni Huffmanov kod manje efikasan od binarnog. Je li to uvijek tako? Pa, najbolji slučaj za ternarni kod je kad je broj simbola potencija broja 3. Recimo, za string `AAAABBBC`, ako ne koristimo nikakav algoritam sažimanja, binarni simbol velik je 2 bita, a ternarni je velik 1.58 bitova. No, ako koristimo Huffmanov algoritam, ternarni kod je još uvijek velik 1.58 bita, a binarni kod je 1.5 bita. Za string `ABCDEFGHI`, simbol u binarnom kodu bez komprimiranja velik je 4 bita, a u ternarnom kodu je 4.75 bita. No, u binarnom Huffmanovom kodu, on je velik 3.22 bita, dok je u ternarnom Huffmanovom kodu velik 3.17 bita. Probajmo što će biti s nekim dugačkim stringom, kao što je, primjerice, `Ubi sunt qui ante nos in mundo fuere? Transeas ad inferos, transeas ad superos, si hos vis videre.`. Za njega je u ternarnom Huffmanovom kodu veličina simbola 4.02 bita, a u binarnom Huffmanovom kodu, velik je 3.99 bita. Dakle, koliko god se činilo zdravorazumski da će ternarni digitalni signal biti bliže analognom i time optimalniji, jednostavnim se eksperimentima ne može naći velika razlika, u stvari, čini se da je binarni većinom optimalniji.

Primjer implementacije Huffmanovog algoritma

Evo kako sam u JavaScriptu implementirao Huffmanov algoritam. To je relativno moderan JavaScript, koji funkcionira u Internet Exploreru 11, ali u Internet Exploreru 10 već ne (koristi

naredbe `const`, `let` i `for-in` petlju, što Internet Explorer 10 ne podržava). Različite vrste riječi u JavaScriptu različitom bojom obojao je VIM 7.4.

```
  1 "use strict"; //That means, approximately, "Interpret this
JavaScript according to the standards, and don't try to be
compatible with archaic JavaScript interpreters.".
  2 let letters,maxX,maxY,minX,maximumDepth,inputString,ternary;
  3 if (typeof Math.log2=="undefined") //Internet Explorer 11
  4     Math.log2=function(x){
  5         return Math.log(x)/Math.log(2);
  6     }
  7 function onButtonClick() {
  8     ternary=document.getElementById("ternary").checked;
  9     inputString=document.getElementById("input").value;
 10     if (inputString.length<2 || inputString.length<3 &&
ternary) {
 11         alert("Strings of length less than two can't be
Huffman encoded, and string of length less than three can't be
encoded ternary.");
 12         return;
 13     }
 14     console.log("Making a Huffman tree for the string
\""+inputString+"\".");
 15     letters=new Object();
 16     for (let i=0; i<inputString.length; i++) {
 17         if (letters[inputString[i]]==undefined) {
 18             letters[inputString[i]]={
 19                 frequency:0,
 20                 hasBeenUsed:false,
 21                 childrenNodes:[]
 22             };
 23         }
 24         letters[inputString[i]].frequency++;
 25     }
 26     let entropy=0,numberOfDistinctLetters=0;
 27     for (let i in letters) {
 28
letters[i].probability=letters[i].frequency/inputString.length;
 29         entropy-
=letters[i].probability*Math.log2(letters[i].probability);
 30         numberOfDistinctLetters++;
 31     }
 32     let bitsInEqualCode=Math.ceil(ternary?
Math.log(numberOfDistinctLetters)/
Math.log(3):Math.log2(numberOfDistinctLetters));
 33     if (numberOfDistinctLetters<2 || numberOfDistinctLetters<3
&& ternary) {
 34         alert("There need to be at least as many distinct
symbols in the string as the base is!");
 35         return;
 36     }
 37     let howManyUnused=numberOfDistinctLetters;
 38     let rootNode;
 39     do {
```

```
 40          let minimum1,minimum2,minimum3;
 41          for (let i in letters)
 42              if (letters[i].hasBeenUsed==false &&
(minimum1==undefined ||
letters[i].frequency<letters[minimum1].frequency))
 43                  minimum1=i;
 44          for (let i in letters)
 45              if (letters[i].hasBeenUsed==false && i!=minimum1
&& (minimum2==undefined ||
letters[i].frequency<letters[minimum2].frequency))
 46                  minimum2=i;
 47          if (ternary)
 48              for (let i in letters)
 49                  if (letters[i].hasBeenUsed==false && i!
=minimum1 && i!=minimum2 && (minimum3==undefined ||
letters[i].frequency<letters[minimum3].frequency))
 50                      minimum3=i;
 51          if (ternary && minimum3)
 52              console.log("Connecting \'"+minimum1+"\',
\'"+minimum2+"\' and \'"+minimum3+"\' into a single node.");
 53          else
 54              console.log("Connecting \'"+minimum1+"\' and
\'"+minimum2+"\' into a single node.");
 55          letters[minimum1].hasBeenUsed=true;
 56          letters[minimum2].hasBeenUsed=true;
 57          if (ternary && minimum3)
 58              letters[minimum3].hasBeenUsed=true;
 59          let nameofTheNewNode=(ternary && minimum3)?
(minimum1+minimum2+minimum3):(minimum1+minimum2);
 60          letters[nameofTheNewNode]=new Object();
 61          letters[nameofTheNewNode].childrenNodes=(ternary &&
minimum3)?[minimum1, minimum2, minimum3]:[minimum1, minimum2];
 62
letters[nameofTheNewNode].frequency=letters[minimum1].frequency+le
tters[minimum2].frequency+((ternary && minimum3)?
letters[minimum3].frequency:0);
 63          if
(letters[nameofTheNewNode].frequency==inputString.length)
 64              rootNode=nameofTheNewNode;
 65          letters[nameofTheNewNode].hasBeenUsed=false;
 66          howManyUnused=0;
 67          for (let i in letters)
 68              if (letters[i].hasBeenUsed==false)
 69                  howManyUnused++;
 70      }
 71      while (howManyUnused>1);
 72      let stackWithNodes=[rootNode];
 73      let stackWithCodes=[""];
 74      let stackWithDepths=[0];
 75      let averageSymbolLength=0;
 76      maximumDepth=0;
 77      let counter=0;
```

```
 78
document.getElementById("table").innerHTML="<tr><td>symbol</td><td
>frequency</td><td>Huffman code</td><td>equal-length
code</td></tr>";
 79     while (stackWithNodes.length>0) {
 80         let currentNode=stackWithNodes.pop();
 81         let currentCode=stackWithCodes.pop();
 82         let currentDepth=stackWithDepths.pop();
 83         maximumDepth=Math.max(maximumDepth,currentDepth);
 84         letters[currentNode].code=currentCode;
 85         if (letters[currentNode].childrenNodes.length==0) {
 86
averageSymbolLength+=letters[currentNode].probability*currentCode.
length*(ternary?Math.log2(3):1);
 87             let equalLengthCode=counter.toString(ternary?3:2);
 88             while (equalLengthCode.length<bitsInEqualCode)
 89                 equalLengthCode='0'+equalLengthCode;
 90
document.getElementById("table").innerHTML+="<tr><td>"+
 91
currentNode+"</td><td>"+
 92
letters[currentNode].frequency+"/"+inputString.length+
 93
"</td><td>"+currentCode+"</td><td>"+equalLengthCode+"</td></tr>";
 94             counter++;
 95             continue;
 96         }
 97
stackWithNodes.push(letters[currentNode].childrenNodes[0]);
 98
stackWithNodes.push(letters[currentNode].childrenNodes[1]);
 99         if (ternary &&
letters[currentNode].childrenNodes.length>2)
100
stackWithNodes.push(letters[currentNode].childrenNodes[2]);
101         stackWithCodes.push(currentCode+"0");
102         stackWithCodes.push(currentCode+"1");
103         if (ternary &&
letters[currentNode].childrenNodes.length>2)
104             stackWithCodes.push(currentCode+"2");
105         stackWithDepths.push(currentDepth+1);
106         stackWithDepths.push(currentDepth+1);
107         if (ternary &&
letters[currentNode].childrenNodes.length>2)
108             stackWithDepths.push(currentDepth+1);
109     }
110     console.log("The Huffman tree is constructed:");
111     console.log("node\tfreq\tcode\tleft\tright")
112     for (let i in letters)
113         console.log("'"+i+"'\
t"+letters[i].frequency+"/"+inputString.length+"\t"+
```

```
114                         letters[i].code+"\t"+
((letters[i].childrenNodes[0])?
("'"+letters[i].childrenNodes[0]+"'"):"null")+
115                         "\t"+(letters[i].childrenNodes[1]?
("'"+letters[i].childrenNodes[1]+"'"):"null"));
116      console.log("The Huffman encoding is:");
117      let output="";
118      for (let i=0; i<inputString.length; i++)
119          output+=letters[inputString[i]].code;
120      console.log(output);
121      console.log("The average length of a symbol in Huffman
code is: "+averageSymbolLength+" bits.");
122
document.getElementById("avgLength").innerHTML=averageSymbolLength
;
123      console.log("The average length of a symbol in the equal-
length code is: "+bitsInEqualCode+" bits.");
124
document.getElementById("bitsInEqualCode").innerHTML=(ternary?
Math.log2(3):1)*bitsInEqualCode;
125      console.log("The entropy of the input string is:
"+entropy+" bits.");
126      document.getElementById("entropy").innerHTML=entropy;
127      console.log("The efficiency of the Huffman code is: "+
(entropy/averageSymbolLength));
128      console.log("The efficiency of the equal-length code is:
"+(entropy/bitsInEqualCode));
129      document.getElementById("output").innerText=output;
130      let tree=document.getElementById("tree");
131      const svgNS=tree.namespaceURI;
132      while (document.getElementById("tree").childNodes.length)
//Clear the diagram ("innerHTML" won't work in Internet Explorer
11 because, to it, SVG is XML and not HTML).
133
document.getElementById("tree").removeChild(document.getElementByI
d("tree").firstChild);
134      maxX=maxY=minX=0;
135      draw(rootNode,0,0,(ternary?20:30)*Math.pow(ternary?
3:2,maximumDepth),0);
136      for (let i=0;
i<document.getElementById("tree").childNodes.length; i++) //In
case a node falls left of the diagram, move all nodes rightwards.
137      {
138          let
childNode=document.getElementById("tree").childNodes[i];
139          if (childNode.getAttribute("x"))
140              childNode.setAttribute("x",
childNode.getAttribute("x") * 1 - minX);
141          if (childNode.getAttribute("x1"))
142              childNode.setAttribute("x1",
childNode.getAttribute("x1") * 1 - minX);
143          if (childNode.getAttribute("x2"))
```

```
144             childNode.setAttribute("x2",
childNode.getAttribute("x2") * 1 - minX);
145     }
146     document.getElementById("tree").style.height = maxY + 100
+ "px";
147     document.getElementById("tree").style.width= maxX - minX +
100 + "px";
148     document.getElementById("diagramSpan").scrollLeft =
document.getElementById("node0").getAttribute("x") -
document.getElementById("diagramSpan").clientWidth / 2 + 75; //The
root of the tree will be in the center of the screen.
149 }
150 function draw(nodeName, x, y, space, id)
151 {
152     if (x > maxX)
153         maxX = x;
154     if (x < minX)
155         minX = x;
156     if (y > maxY)
157         maxY = y;
158     const svgNS =
document.getElementById("tree").namespaceURI;
159     let rectangle = document.createElementNS(svgNS, "rect");
160     rectangle.setAttribute("x", x);
161     rectangle.setAttribute("y", y);
162     rectangle.setAttribute("width", 50);
163     rectangle.setAttribute("height", 50);
164     rectangle.setAttribute("id", "node" + id);
165     rectangle.setAttribute("fill","#EEEEEE");
166     document.getElementById("tree").appendChild(rectangle);
167     let text = document.createElementNS(svgNS, "text");
168
text.appendChild(document.createTextNode(letters[nodeName].frequen
cy+"/"+inputString.length));
169     text.setAttribute("x", x+5);
170     text.setAttribute("y", y + 20);
171     text.style.fill = "black";
172     text.setAttribute("font-family", "monospace");
173     text.setAttribute("font-size", 14);
174     document.getElementById("tree").appendChild(text);
175     if (nodeName.length==1) {
176         let character = document.createElementNS(svgNS,
"text");
177
character.appendChild(document.createTextNode(nodeName));
178         character.setAttribute("x", x+20);
179         character.setAttribute("y", y + 40);
180         character.style.fill = "black";
181         character.setAttribute("font-family", "monospace");
182         character.setAttribute("font-size", 14);
183
document.getElementById("tree").appendChild(character);
184     }
```

```
185      for (let i = 0; i <
letters[nodeName].childrenNodes.length; i++) {
186          draw(letters[nodeName].childrenNodes[i], x + (i -
(ternary?3:1)/(ternary?3:2)) * space, y + 100, space / (ternary?
3:2), id + 1);
187          let line = document.createElementNS(svgNS, "line");
188          line.setAttribute("x1", x + 25);
189          line.setAttribute("y1", y + 50);
190          line.setAttribute("x2", x + (i -
(ternary?3:1)/(ternary?3:2)) * space + 25);
191          line.setAttribute("y2", y + 100);
192          line.setAttribute("stroke-width", 2);
193          line.setAttribute("stroke", "black");
194          document.getElementById("tree").appendChild(line);
195          let bit = document.createElementNS(svgNS,"text");
196          bit.appendChild(document.createTextNode(i));
197          bit.setAttribute("x", x + (i - (ternary?3:1)/(ternary?
3:2)) * space + 25);
198          bit.setAttribute("y", y + 80);
199          bit.style.fill = "black";
200          bit.setAttribute("font-family", "monospace");
201          bit.setAttribute("font-size", 14);
202          document.getElementById("tree").appendChild(bit);
203      }
204 }
```

Kod od 130. do 204. retka služi crtanju dijagrama. Veoma sličan kod koristi i compiler za moj programski jezik[7] za crtanje apstraktnog sintaksnog stabla. Tamo su komentari na hrvatskom jeziku.

Zaključak

Instinktivno je misliti da će korištenje ternarnog Huffmanovog kodiranja dati optimalniji kod od binarnog Huffmanovog kodiranja. Međutim, jednostavni eksperiment to ne potvrđuje, i zapravo ukazuje na suprotno.

7 Može se pokrenuti u modernom internetskom pregledniku, ovdje: https://flatassembler.github.io/compiler.html