Математичка гимназија

МАТУРСКИ РАД

- из предмета Рачунарство и информатика -

Компресија података

Ученик:
Игор Павловић IVд

Ментор:
Јелена Хаџи-Пурић

Београд, јун 2020.

Садржај

1

Увод

1.1 Дефиниција компресије

Компресијом података се назива њихово пресликавање из једног начина представљања (скупа симбола једног алфабета) у други, тако да се добије концизнији низ симбола. У теоријском рачунарству оваква пресликавања зовемо кодирање информација са циљем сажимања укупног броја битова.

Када говоримо о компресији говоримо о два алгоритма, алгоритму компресије и алгоритму реконструкције. Алгоритам компресије прихвата улаз X и на основу њега генерише његову репрезентацију X_c, која користи мање битова. Алгоритам реконструкције добија репрезентацију X_c и на основу ње формира реконструкцију Y.

На основу потреба корисника, алгоритме компресије можемо поделити у две класе: алгоритме без губитка информација, код којих је X идентично као Y, и алгоритме са губитком информација који углавном постижу знатно већу компресију, али дозвољавају да X буде различито од Y.

Један од примера компресије, који је настао још средином 19. века, је Морзеова азбука. Морзе је свако слово заменио сигналом који се састоји из кратких и дугих звучних или светлосних сигнала. Кратки сигнал се бележи као тачка а дуги као црта. Морзе је приметио да се нека слова, попут слова а и е, појављују чешће од осталих. Како би скратио просечну дужину поруке он је слова која се често појављују заменио краћим сигналима од оних које је доделио осталим словима.

1.2 Компресија без губитка

Компресија без губитка је врста компресије код које нема никакве разлике између података пре компресије и након декомпресије. Овај вид компресије се заснива на идентификацији и елиминацији редундансе и користи се у случајевима када не можемо толерисати разлике између оригиналне и реконстурисане информације, попут података у банкама. Најпознатији примери ове врсте компресије су PDF и PNG.

Најједноставнији представник ове класе алгоритама компресије је RLE алгоритам (енг. *Run-lenght encoding*). Овај алгоритам користи чињеницу да се у многим датотекама низови истих карактера појављују на узастопним позицијама.

Пример 1. Следећи низ карактера $AAAAXXXXXAA$ можемо записати као $4A5X2A$.

Карактеристике овог алгоритма су: једноставна имплементација, мала временска и просторна сложеност, једноставна валидација и ограничене могућности компресије. Уколико подаци које компресујемо садрже врло мало узастопних понављања истог карактера могуће је да компресована верзија садржи више битова од оригиналне.

Пример 2. Посматрајмо следећи низ бројева.

9	11	11	11	14	13	15	17	16	17	20	21

Да би запамтили овај низ потребно нам је по 5 битова за сваки број из низа. Уколико уведемо нови низ $a_n = n + 8$ можемо уместо почетног низа послати низ a и следеће бројеве {0,1,0,-1,1,-1,0,1,-1,-1,1,1}, то јест разлику ова два низа. На овај начин ће нам за сваки члан низа требати само 2 бита.

1.3 Компресија са губитком

Код компресије са губитком разлике између оригиналног и реконструисаног фајла су прихватљиве (имајући на уму несавршеност људских чула). У многим случајевима попут компресије аудио или видео сигнала ове разлике су прихватљиве или чак неприметне. Као замену за изгубљене информације добијамо већу ефикасност од компресије без губитка информација. Код овакве компресије неопходно је измерити разлику између оригиналног и реконструисаног фајла како бисмо одредили ефикасност алгоритма. Ова разлика се назива дисторзија.

1.4 Мерење ефикасности алгоритама

Ефикасност алгоритма компресије можемо мерити на више начина. Можемо мерити његову временску и меморијску комплексност, комплексност имплементације или сличност реконструкције са оригиналом и количину компресије.

Једноставан начин за мерење ефикасности је мерење односа броја битова пре и после компресије. Овај однос се зове однос компресије (енг. *compression ratio*). Још један добар начин за мерење ефикасности је просечан број битова по симболу.

Пример 3. Посматрајмо слику као квадратну матрицу са 256*256 пиксела која захтева 65,536 бајтова меморије. Након компресије ова слика заузима 16,384 бајта. Тада кажемо да је однос компресије 4:1. Овај однос можемо представити и као проценат смањења величине фајла. У овом случају је то 75%. Просечан број битова по пикселу у овом случају је 2.

2

Математичка основа

2.1 Увод у теорију информација

Амерички научник Клод Елвуд Шенон је увео величину за мерење информације и назвао је $self\text{-}information$, чиме је поставио темеље теорије информација. $Self\text{-}information$ можемо посматрати као количину информације коју носи чињеница да се неки догађај десио или као мерило иненађења неког догађаја. Ову величину можемо искористити за одређивање минималног могућег броја битова компресованог фајла.

Дефиниција 1. Претпоставимо да имамо неки догађај A који је резултат неког насумичног експеримента. Ако је $P(A)$ вероватноћа да се тај догађај догоди, онда се $self\text{-}information$ тог догађаја рачуна као негативни логаритам $P(A)$ са основом b.

$$i(A) = log_b(1/P(A)) = -log_b P(A)$$

Приметимо да основа логаритма није одређена. Уколико представљамо информацију у битовима треба узети логаритам са основом 2.

Ова дефиниција има смисла зато што информација догађаја расте када вероватноћа догађаја опада, то јест догађаји чија је вероватноћа велика носе малу информацију док догађаји са малом вероватноћом носе велику информацију.

Приметимо још да је информација два независна догађаја збир информација та два догађаја.

$$i(AB) = -log_b(P(AB))$$

Пошто су догађаји независни следи:

$$i(AB) = -log_b(P(A) \cdot P(B))$$
$$i(AB) = -log_b(P(A)) - log_b(P(B))$$
$$i(AB) = i(A) + i(B)$$

2.2 Ентропија

Посматрајмо скуп независних догађаја A_i неког експеримента S таквих да важи:

$$\cup A_i = S$$

Тада просечну информацију експеримента S рачунамо на следећи начин

$$H = \sum P(A_i) \cdot i(A_i) = -\sum P(A_i) \cdot log_b(P(A_i))$$

Ову величину називамо ентропија. Шенон је доказао да компресија без губитка информација не може имати мањи просечан број битова по симболу од ентропије изворног фајла.

Пример 4. Посматрајмо следећи низ бројева:

1 2 3 2 3 4 5 4 5 6 7 8 9 8 9 10

Уколико претпоставимо да је број појављивања неког броја сразмеран његовој вероватноћи добијамо следеће вероватноће:

$$P_1 = P_6 = P_7 = P_{10} = 1/16$$
$$P_2 = P_3 = P_4 = P_5 = P_8 = P_9 = 2/16$$

Ентропија овог низа је 3,25. Ако овај низ заменимо разликом суседних елемената добијамо следећи низ:

1 1 1 -1 1 1 1 -1 1 1 1 1 1 -1 1 1

Приметимо да је за запис оваквог низа довољан 1 бит по симболу, а његова ентропија је 0,7 што је знатно мање од почетне ентропије. Овде видимо да је применом неких особина низа могуће смањити његову ентропију.

2.3 Кодови који се могу јединствено декодирати

Из практичних разлога неопходно је да постоји једниствен начин да прочитамо компресовану поруку, то јест компресована порука не сме бити двосмислена. Након што конструишемо код за свако слово из алфабета треба извршити следећи тест за проверу да ли је увек могуће јединствено дешифровати компресовану поруку.

Дефиниција 2. Претпоставимо да имамо два бинарна кода a и b, нека је дужина првог кода k бита, а дужина другог кода n бита, и $k < n$. Ако је првих k бита кода b идентично коду a, тада кажемо да је код a префикс кода b. Последњих $n - k$ битова називамо преостали суфикс.

Конструишимо прво листу кодних речи свих симбола. Након тога упоредимо свака два кода из листе и проверимо да ли је један префикс другог. Уколико јесте проверимо да ли се преостали суфикс већ налази у листи, уколико не, додајемо га у листу. Понављамо овај процес све док не стигнемо до неког од два могућа случаја:

(1) Пронашли смо преостали суфикс који се поклапа са кодом за неко слово

(2) Нема више нових преосталих суфикса

Поруку је увек могуће јединствено прочитати у другом случају.

2.4 Префиксни кодови

Једна врста кода за који је увек могуће јединствено дешифровати поруку је префиксни код. Код оваквих кодова ниједна кодна реч није префикс неке друге кодне речи. Дакле, овакав код сигурно испуњава горе наведени услов. Најпознатији код из ове класе је Хафманов код.

Најједноставнији начин за проверу да ли је код префиксни је конструисати специјалну структуру података тзв. *trie* од кодних речи. Уколико свакој кодној речи одговара лист стабла тада знамо да ниједна кодна реч није префикс неке друге кодне речи.

У наставку ћемо показати да ако се ограничимо само на префиксне кодове нећемо изгубити на ефикасности компресије.

2.5 Крафт-Мек Миланова неједнакост

Теорема 1. Нека је C код са N кодних речи дужина $l_1, l_2, \ldots, l_N$. Ако је C могуће јединствено декодирати, онда важи следећа неједнакост.

$$K(C) = \sum_{i=1}^{N} 2^{-l_i} \leq 1$$

Ова неједнакост је позната као Крафт-Мек Миланова неједнакост.

Доказ. Да бисмо ово доказали степеноваћемо $K(C)$ на неки природан број n. Ако је $K(C)$ веће од један, онда ће $K(C)^n$ расти експоненцијално са n.

Нека је n произвољан природан број. Тада важи

$$[\sum_{i=1}^{N} 2^{-l_i}]^n = (\sum_{i_1=1}^{N} 2^{-l_{i_1}})(\sum_{i_2=1}^{N} 2^{-l_{i_2}}) \cdots (\sum_{i_n=1}^{N} 2^{-l_{i_n}})$$

$$= \sum_{i_1=1}^{N} \sum_{i_2=1}^{N} \cdots \sum_{i_n=1}^{N} 2^{-(l_{i_1}+l_{i_2}+\cdots+l_{i_n})}$$

Експонент $l_{i_1} + l_{i_2} + \cdots + l_{i_n}$ представља дужину неких n кодних речи из C, са могућношћу понављања истих кодних речи. Ако је $l = max(l_1, l_2, \ldots, l_N)$. Најмања вредност коју експонент може имати је n, а највећа nl. Дакле, посматрану суму можемо записати као:

$$K(C)^n = \sum_{k=n}^{nl} A_k \cdot 2^{-k}$$

где A_k представља број комбинација n кодних речи укупне дужине k, за $n \leq k \leq nl$. Овај број комбинација не може бити већи од броја различитих бинарних записа дужине k јер је увек могуће јединствено дешифровати код. Другим речима $A_k \leq 2^k$, што значи да

$$K(C)^n = \sum_{k=n}^{nl} A_k \cdot 2^{-k} \leq \sum_{k=n}^{nl} 2^k \cdot 2^{-k} = n \cdot l - n + 1$$

Ако је $K(C)$ веће од један онда лева страна ове неједнакости расте експоненцијално, док десна расте линеарно. Што значи да је могуће изабрати довољно велико n за које ова неједнакост не важи. Дакле $K(C)$ мора бити мање или једнако од један.

□

Сада ћемо доказати да сваки код који је могуће јединствено декодирати, то јест код који задовољава Крафт-Мек Миланову неједнакост, можемо заменити једнако ефикасним префиксним кодом.

Теорема 2. За дати низ дужина кодних речи $l_1, l_2, \ldots, l_N$ које задовољавају Крафт-Мек Миланову неједнакост, то јест $\sum_{i=1}^{N} 2^{-l_i} \leq 1$, увек можемо конструисати префиксни код са кодним речима дужине $l_1, l_2, \ldots, l_N$.

Доказ. Ово можемо доказати тако што ћемо конструисати префиксни код са задатим дужинама кодних речи. Без умањења општости, можемо претпоставити да

$$l_1 \leq l_2 \leq \cdots \leq l_n$$

Дефинишимо низ бројева $a_1, a_2, \ldots, a_N$ на следећи начин

$$a_1 = 0$$

$$a_j = \sum_{i=1}^{j-1} 2^{l_j - l_i}, \quad j \geq 2$$

Бинарни запис броја a_j захтева $\lceil log_2(a_j + 1) \rceil$ битова. Може се показати да је овај број битова увек мањи или једнак од l_j.

$$log_2(a_j + 1) = log_2(\sum_{i=1}^{j-1} 2^{l_j - l_i} + 1)$$

$$= log_2(2^{l_j} \cdot (\sum_{i=1}^{j-1} 2^{-l_i} + 2^{-l_j}))$$

$$= l_j + log_2(\sum_{i=1}^{j} 2^{-l_i})$$

Пошто је $\sum_{i=1}^{j} 2^{-l_i}$ сигурно мање или једнако од један, због услова теореме, одатле следи

$$log_2(\sum_{i=1}^{j} 2^{-l_i}) \leq log_2(1) = 0$$

Дакле, важи следеће

$$log_2(a_j + 1) = l_j + log_2(\sum_{i=1}^{j} 2^{-l_i}) \leq l_j$$

Кодне речи конструишемо тако што на бинарни запис броја a_j додамо $l_j - \lceil log_2(a_j + 1) \rceil$ нула на позицијама битова највеће тежине (тј. са његове леве стране), тако добијамо кодну реч c_j. Ове кодне речи читамо почев од најзначајнијег бита. Остаје још само да докажемо да је овако добијени код заиста префиксни.

Претпоставимо супротно. Ако постоје j и k, такви да је c_j префикс c_k. То значи да је l_j најзначајнијих битова a_k једнако бинарном запису a_j. То јест

$$a_j = \left\lfloor \frac{a_k}{2^{l_k - l_j}} \right\rfloor$$

$$\frac{a_k}{2^{l_k - l_j}} = \sum_{i=0}^{k-1} 2^{l_j - l_i} = a_j + \sum_{i=j}^{k-1} 2^{l_j - l_i} = a_j + 1 + \sum_{i=j+1}^{k-1} 2^{l_j - l_i} \geq a_j + 1$$

Одатле следи да је најмања вредност коју $\left\lfloor \frac{a_k}{2^{l_k - l_j}} \right\rfloor$ може имати $a_j + 1$ што је у контрадикцији са претпоставком да је $a_j = \left\lfloor \frac{a_k}{2^{l_k - l_j}} \right\rfloor$, што значи да c_j није префикс c_k.

□

3

Хафманов код

3.1 Опис алгоритма

Хафманов код је најефикаснији метод компресије у својој класи, а то је класа пресликавања карактера азбуке у битове за познату фреквенцију појаве сваког карактера. Овај код је оптималан бинарни префиксни код који је развио Дејвид Хафман и заснива се на две претпоставке:

(1) У оптималном коду симболи који се појављују чешће од других, то јест имају већу вероватноћу појављивања, ће имати краће кодне речи од оних симбола који се појављују ређе од њих

(2) У оптималном коду, два најређа симбола ће имати исту дужину кодне речи

Приметимо да уколико прва претпоставка није испуњена за нека два симбола можемо само заменити њихове кодне речи и добити оптималнији код. Дакле, у оптималном коду прва претпоставка мора бити задовољена.

Претпоставимо да постоји оптимални префиксни код у ком друга претпоставка не важи. Посматрајмо кодне речи два најређа симбола, нека су те кодне речи дужина l и k и важи $l < k$. На основу прве претпоставке знамо да су ове две кодне речи најдуже, а пошто је код префиксни ниједан симбол нема кодну реч која се поклапа са првих l битовова ове две кодне речи. Приметимо такође да се првих l битова ове две речи такође разликују, јер је код префикан. Дакле, отписивањем последњих $k - l$ битова дуже кодне речи добијамо валидан код који је оптималнији од почетног. Одавде закључујемо да у оптималном коду и друга претпоставка мора бити испуњена.

Срж овог алгоритма је конструкција префиксног стабла које задовољава ове две претпоставке. Такво стабло конструишемо од дна ка врху. Прво

се формира низ симбола који ће чинити листове стабла. Затим се сваком симболу додели тежина која описује његову вероватноћу појављивања.

Стабло се гради кроз следеће кораке:

1. Проналазе се два слободна чвора са најмањим тежинама
2. Креира се нови чвор спајањем та два чвора, тако да је нови чвор њихов заједнички родитељ. Тежина новог чвора је једнака збиру тежина ова два чвора.
3. Лева грана новог чвора добија вредност 0, а десна добија вредност 1.
4. Почетна два чвора се избацују из низа слободних чворова, а нови чвор се додаје у тај низ.
5. Ови кораци се понављају све док у низу слободних чворова не остане само један чвор. Тај чвор је корен Хафмановог стабла.

3.2 Оптималност Хафмановог кода

Индукцијом по броју симбола n можемо показати да је Хафманов код најефикаснији метод компресије у својој класи.

Доказ. База: За $n = 2$, Хафманов код је оптималан јер користи само 1 бит по симболу.

Корак: Претпоставимо да је Хафманов код оптималан за било који алфабет величине $n-1$(са било којим вероватноћама симбола).

Нека је L просечна дужина Хафмановог кода за алфабет S, са n симбола $s_1, \ldots, s_n$ и вероватноћама $p_1, \ldots, p_n$. Без умањења општости претпоставимо да важи $p_i \geq p_{n-1} \geq p_n$ за $i \in \{1, \ldots, n-2\}$.

Рекурзивном применом Хафмановог кода можемо конструисати код за алфабет S', са $n-1$ симбола $s_1, \ldots, s_{n-2}, s'$ и вероватноћама $p_1, \ldots, p_{n-2}, p_{n-1} + p_n$. По индуктивној хипотези такав код је оптималан. Нека је његова просечна дужина L'.

Претпоставимо да постоји неки префиксни код просечне дужине мање од L. Из Хафманових претпоставки закључујемо да се тај код може модификовати тако да симболи s_{n-1} и s_n буду суседи у стаблу, то јест да њихови кодови буду облика "$x0$" и "$x1$", и да при томе не повећамо просечну дужину кода. Нека је просечна дужина тог кода $\widehat{L}$.

Из овог модификованог кода можемо направити код за алфабет S'. Тако што симболу s' доделимо код x, а кодове осталих симбола оставимо исте. Нека је просечна дужина овог кода $\widehat{L}'$.

Просечне дужине ових кодова морају да задовољавају следећу једнакост.

$$L = L' + p_{n-1} + p_n$$
$$\widehat{L} = \widehat{L}' + p_{n-1} + p_n$$

По индуктивној хипотези добијамо $L' \leq \widehat{L}'$, одакле следи $L \leq \widehat{L}$, што је у контрадикцији са претпоставком да $\widehat{L} < L$.

Одавде закључујемо да је Хафманов код оптималан за сваки алфабет величине n.

□

3.3 Дужина Хафмановог кода

У овом делу ћемо показати да је просечна дужина кодне речи L у Хафмановом коду за неки извор S ограничена са $H(S)$ и $H(S)+1$. То јест

$$H(S) \leq L < H(S) + 1$$

Користећи Крафт-Мек Миланову неједнакост из другог поглавља доказаћемо следеће.

1. Просечна дужина L Хафмановог кода за извор S је већа или једнака од $H(S)$

2. Просечна дужина L Хафмановог кода за извор S је мања од $H(S)+1$

За извор S са алфабетом $A = \{a_1, a_2, \ldots, a_k\}$ и вероватноћама симбола $\{P(a_1), P(a_2), \ldots, P(a_k)\}$ просечна дужина се рачуна на следећи начин

$$L = \sum_{i=1}^{K} P(a_i) \cdot l_i$$

Одавде добијамо

$$H(S) - L = -\sum_{i=1}^{K} P(a_i) \cdot log_2(P(a_i)) - \sum_{i=1}^{K} P(a_i) \cdot l_i$$
$$= \sum_{i=1}^{K} P(a_i) \cdot (log_2[\frac{1}{P(a_i)}] + log_2(2^{-l_i}))$$
$$= \sum_{i=1}^{K} P(a_i) \cdot log_2[\frac{2^{-l_i}}{P(a_i)}] \leq log_2[\sum_{i=1}^{K} 2^{-l_i}]$$

Ова неједнакост је последица Јенсенове неједнакости, која тврди да ако је $f(x)$ конкавна функција онда важи $E[f(x)] \leq f(E[x])$, а log је конкавна функција. По договору, користимо $E[f(x)]$ да означимо очекивану вредност функције $f(x)$.

Одавде следи

$$H(S) - L \leq log_2[\sum_{i=1}^{K} 2^{-l_i}] \leq log_2(1) = 0$$

Горњу границу ћемо доказати тако што ћемо показати да постоји код чија је просечна дужина кодне речи $H(S)+1$, а пошто је Хафманов код оптималан за ову класу кодова онда је $L \leq H(S) + 1$.

Узмимо да је

$$l_i = \lceil log_2(\frac{1}{P(a_i)}) \rceil$$

Одавде следи

$$log_2(\frac{1}{P(a_i)}) \leq l_i < log_2(\frac{1}{P(a_i)}) + 1$$

$$2^{-l_i} \leq P(a_i)$$

$$\sum_{i=1}^{K} 2^{-l_i} \leq \sum_{i=1}^{K} P(a_i) = 1$$

Из Крафт-Мек Миланове неједнакости следи да постоји префиксни код са овим дужинама кодних речи.

Користећи неједнакост $l_i < log_2(\frac{1}{P(a_i)}) + 1$ добијамо

$$L = \sum_{i=1}^{K} P(a_i) \cdot l_i < \sum_{i=1}^{K} P(a_i) \cdot [log_2(\frac{1}{P(a_i)}) + 1]$$

$$L < H(S) + 1$$

Такође се може показати да је L ограничено одозго са $H(S) + p_{max} + 0.086$, где је p_{max} вероватноћа појављивања најчешћег карактера.

3.4 Имплементација алгоритма

```cpp
struct Cvor {
    Cvor(char c, int f):karakter(c),frekvencija(f){};
    char karakter;
    int frekvencija;
    Cvor *levo = nullptr;
    Cvor *desno = nullptr;
};

struct poredi {
    bool operator()(Cvor *c1, Cvor *c2)
    {
        return c1->frekvencija > c2->frekvencija;
    }
};

Cvor* huffman(vector<Cvor*> &karakteri)
{
    if(karakteri.size() == 0) return nullptr;
    priority_queue<Cvor*, vector<Cvor*>, poredi> hip;
    for(int i = 0; i < karakteri.size(); i++)
    {
        hip.push(karakteri[i]);
    }
    while(hip.size() > 1)
    {
        Cvor *levi = hip.top();
        hip.pop();
        Cvor *desni = hip.top();
        hip.pop();
        Cvor *novi = new Cvor('#', desni->frekvencija + levi->frekvencija);
        novi->levo = levi;
        novi->desno = desni;
        hip.push(novi);
    }
    Cvor *koren = hip.top();
    hip.pop();
    return koren;
}

void ispis(Cvor *koren, string niska)
{
    if(koren != nullptr)
    {
        if(koren->karakter == '#')
        {
            ispis(koren->levo, niska + "0");
            ispis(koren->desno, niska + "1");
        }
        else
        {
            cout << koren->karakter << ": " << niska << endl;
        }
    }
}
```

3.5 Адаптивни Хафманов код

Уколико порука коју шаљемо није унапред позната, већ је шаљемо карактер по карактер, потребно је после сваког новог симбола ажурирати Хафманово стабло. Наиван начин да ово урадимо је да након сваког симбола ажурирамо вероватноће и поново градимо Хафманово стабло. Овакав алгоритам има велику временску сложеност зато ћемо за ажурирање стабла користити FGK (скр. од Faller-Gallager-Knuth) алгоритам.

Дефиниција 3. Кажемо да бинарно стабло има близаначку особину (енг. *sibiling property*) ако и само ако

(1) Листови стабла имају ненегативне тежине и тежина сваког унутрашњег чвора је једнака збиру тежина његове деце.

(2) Чворови се могу нумерисати у неопадајућем редоследу по тежинама тако да чворови са бројем $2 \cdot j - 1$ и $2 \cdot j$ имају заједничког родитеља и њихов родитељ има већи број од њих.

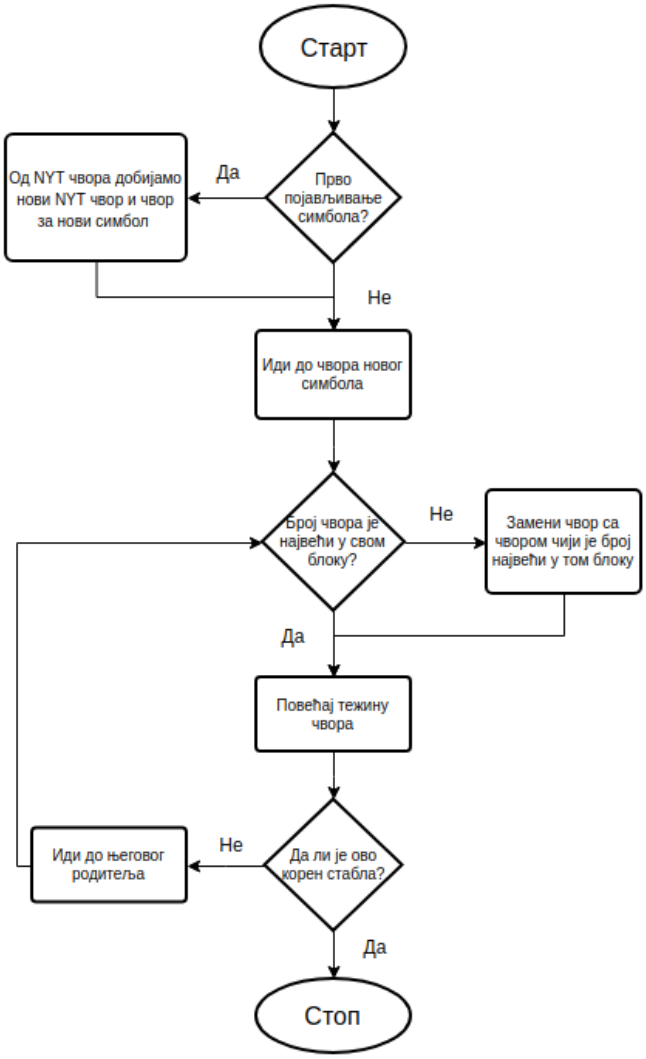
Дефиниција 4. Два чвора припадају истом блоку ако и само ако су њихове тежине једнаке

Може се доказати да је свако бинарно префиксно стабло са овом особином Хафманово стабло. Бројеви чворова одговарају редоследу спајања чворова у Хафмановом кодирању.

Листови адаптивног Хафмановог стабла ће представљати симболе, а њихове тежине ће бити број појављивања тог симбола. Један од листова ће бити NYT чвор (скр. од енг. Not Yet Transmitted) који је резервисан за симболе који се још увек нису појавили и његова тежина ће бити 0.

Нумерацију која задовољава близаначку особину постижемо тако што након сваког појављивања новог симбола од NYT чвора, који има најмањи број додељен досад x, формирамо нови NYT чвор и чвор за нови симбол са бројевима $x - 2$ и $x - 1$, тако да је заједнички родитељ ова два чвора стари NYT чвор.

Конструкцију стабла почињемо од једног NYT чвора, а симболе додајемо пратећи следеће кораке.

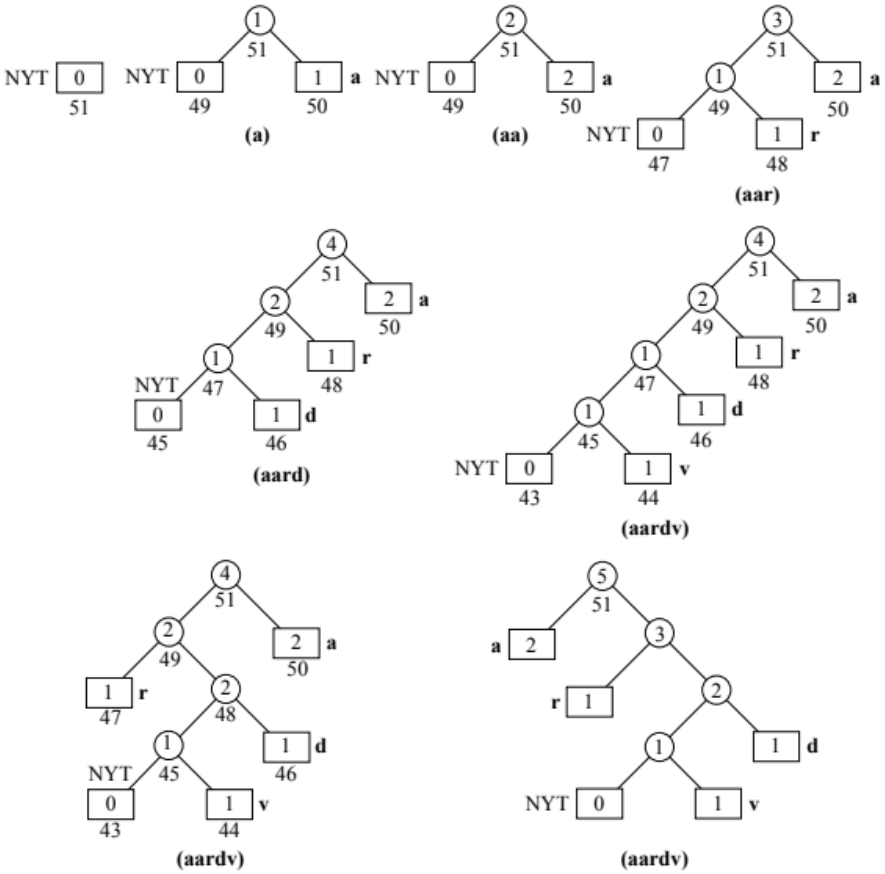


Замену треба извршити само ако други чвор није родитељ првог.

Да би пошиљалац и прималац поруке имали исто стабло неопходно је ову процедуру извршити тек након кодирања, то јест декодирања, симбола.

Приметимо да приликом замене чворова са бројевима x и y, за које важи $x < y$, сви потомци чвора y имају тежине мање од ова два чвора, осим када је y родитељ NYT чвора. Одавде следи да сви потомци чвора y имају број мањи од x, што знаци да заменом тих чворова нећемо нарушити близаначку особину. Увећавање највећег чвора у блоку такође неће нарушити ову особину што значи да ће стабло након ажурирања остати Хафманово стабо.

Пример 5. Посматрајмо следећу поруку [*aardv*] над Енглеским алфабетом са 26 слова. Укупан број чворова који можемо имати над овим алфабетом је $2 \cdot 26 - 1 = 51$, зато почетном чвору додељујемо вредност 51.



Приметимо да није неопходно извршити замену чворова све до појављивања слова v. У овом случају треба прво заменити чвор са бројем 47 са чвором 48, а затим треба заменити чворове 49 и 50.

Предност адаптивног кода је то што не захтева додатну меморију за складиштење стабла јер се оно може конструисати на основу послате поруке. Мана овог алгоритма је што један погрешно послати симбол може да уништи читаву поруку.

4

Аритметичко кодирање

4.1 Генерисање ознаке

Један начин кодирања је да сваком симболу доделимо неку ознаку из интервала $[0, 1)$. Процедура за генерисање ознаке ради рекурзивно смањивање интервала након сваког слова.

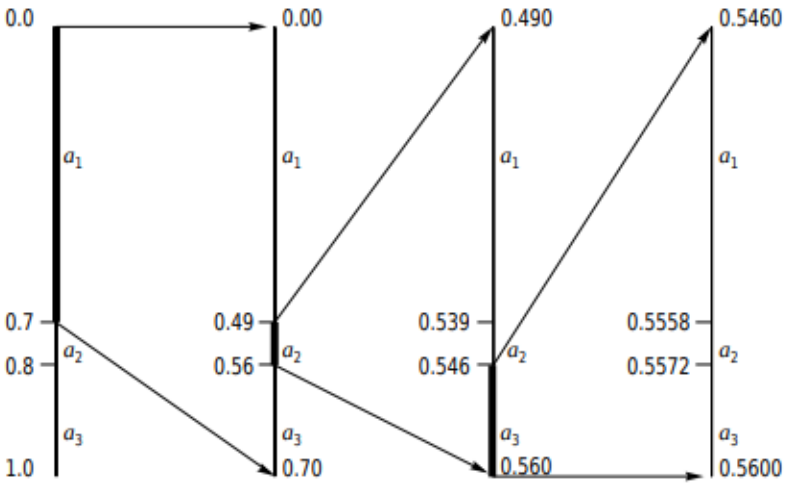
За алфабет $A = \{a_1, a_2, \ldots, a_m\}$, почињемо од интервала $[a, b)$ тако што га изделимо на интервале облика $[F_x(i-1), F_x(i))$, $i = 1, \ldots, m$.

$$F_x(0) = b$$

$$F_x(i) = b + \sum_{k=1}^{i} P(a_i) \cdot (b - a)$$

Интервал $[F_x(i-1), F_x(i))$ придружујемо i-том слову. Ако је прво слово које се појављује a_k онда се ознака речи налази у интервалу $[F_x(k-1), F_x(k))$. Овај процес понављамо рекурзивно почев од интервала $[F_x(k-1), F_x(k))$ све док не дођемо до последњег слова. Ради једноставности кодирање почињемо од интервала $[0, 1)$. Након што добијемо последњи интервал за ознаку речи бирамо произвољан број из тог интервала.

Пример 6. Посматрајмо следећи алфабет $A = \{a_1, a_2, a_3\}$ са вероватноћама $P(a_1) = 0.7$, $P(a_2) = 0.1$ и $P(a_3) = 0.2$. Тада добијамо $F_x(a_1) = 0.7$, $F_x(a_2) = 0.8$ и $F_x(a_3) = 1$. Кодирање речи $a_1a_2a_3$, користећи горе описани алгоритам, је представљено на следећој слици.



Приметимо да овакав начин кодирања различитим речима додељује дисјунктне интервале, што значи да две различите речи не могу имати исту ознаку. Чест избор ознаке је доња граница интервала или средина интевала.

Неопходно је ознаку представити у бинарном запису. Показаћемо да је за неку реч X са вероватноћом $P(X)$ довољан број битова за њену ознаку $l = \lceil log_2(\frac{1}{P(X)}) \rceil + 1$.

Нека је ознака речи T и нека она представља средину одговарајућег интервала. Нека је $\widehat{T}$ број добијен од првих l бинарних цифара броја T.

$$0 \leq T - \widehat{T} < \frac{1}{2^l} \leq \frac{P(X)}{2}$$

С обзиром на то да је дужина одговарајућег интервала $P(X)$, као и да је T средиште интервала закључујемо да се и $\widehat{T}$ налази у том интервалу.

Декодирање радимо тако што почетни интервал изделимо на одговарајуће интервале и проверимо ком слову одговара интервал у ком се налази ознака. Затим запишемо то слово и понављамо овај процес над новим интервалом све док не добијемо тражену реч.

4.2 Имплементација алгоритма

Ако x_n означава n-ти карактер у речи, кодирање речи $x_1x_2...x_m$ имплементирамо рекурзивно на следећи начин

$$l^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)}) \cdot F_x(x_n - 1)$$
$$u^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)}) \cdot F_x(x_n)$$

Након што изачунамо $l^{(m)}$ и $u^{(m)}$, за ознаку речи $x_1x_2...x_m$ узимамо првих $\lceil log_2(\frac{1}{P(X)})\rceil + 1$ битова броја $T = \dfrac{u^{(m)} - l^{(m)}}{2}$.

Пре него што почнемо са имплементацијом треба нагласити да је прецизност израчунавања ограничена. Уколико желимо да представимо све потребне интервале прецизност израчунавања мора расти након сваког корака. Ово можемо постићи помоћу следећег алгоритма.

Смањивањем интервала могу се десити три могућности:

1. Нови интервал се у потпуности налази у интервалу $[0, 0.5)$
2. Нови интервал се у потпуности налази у интервалу $[0.5, 1)$
3. Интервал се налази у обе половине почетног интервала истовремено

У неком од прва два случаја сигурно знамо први бит ознаке. Након тога можемо дуплирати величину неког од ова два интервала како бисмо повећали прецизност.

$$E_1 : [0, 0.5) \to [0, 1) \quad E_1(x) = 2 \cdot x$$
$$E_2 : [0.5, 1) \to [0, 1) \quad E_1(x) = 2 \cdot (x - 0.5)$$

Овај процес понављамо кад год је испуњена нека од прве две могућности.

4.3 Ефикасност алгоритма

Нека је

$$l(X) = \lceil log_2(\frac{1}{P(X)})\rceil + 1$$

Приметимо да је $l(X)$ број битова потребан за кодирање читаве речи. Дакле просечан број битова за реч дужине m се рачуна на следећи начин

$$l_{A_m} = \sum P(X) \cdot l(X)$$
$$= \sum P(X) \cdot (\lceil log_2(\frac{1}{P(X)})\rceil + 1) < \sum P(X) \cdot (log_2(\frac{1}{P(X)}) + 2)$$
$$= \sum P(X) \cdot log_2(\frac{1}{P(X)}) + 2 \cdot \sum P(X)$$
$$= H(X^{(m)}) + 2$$

Пошто је просечна дужина увек већа или једнака од ентропије онда је просечна дужина једног симбола огрничена са

$$\frac{H(X^{(m)})}{m} \leq l_A < \frac{H(X^{(m)})}{m} + \frac{2}{m}$$

4.4 Поређење са Хафмановим кодом

Уколико радимо са речима дужине m просечан број битова по симболу у Хафмановом коду се може огрничити са

$$\frac{H(X^{(m)})}{m} \leq l_A < \frac{H(X^{(m)})}{m} + \frac{1}{m}$$

На први поглед делује да је Хафманов код ефикаснији од аритметичког кодирања. Овде треба нагласити да је за Хафманов код неопходно запамтити одговарајуће стабло чија је величина сразмерна броју речи. С обзиром на то да је за алфабет са A карактера број речи дужине m једнак A^m, Хафманов код није примењив за велике вредности m. Код аритметичког кодирања немамо тај проблем, довољно је само да запишемо вероватноће за свако слово.

Такође треба нагласити да је имплементација адаптивног аритметичког кода знатно једноставнија и временски ефикаснија, јер је довољно ажурирати само вероватноће сваког слова.

5

Кодирање помоћу речника

5.1 Увод

У многим изворним фајловима се неке речи појављују веома често, а неке речи се не појављују уопште или се појављују врло ретко.

Један ефикасан начин кодирања оваквих фајлова је употреба листе, или речника, оних речи које се најчешће појављују. Уколико се реч налази у речнику за њен код узимамо адресу у речнику, у супротном користимо неки други мање ефикасан код. Да би овај метод био ефикасан неопходно је да речник буде знатно мањи од укупног броја могућих речи.

Пример 7. Посматрајмо фајл који садржи трословне Енглеске речи праћене интерпункцијским знаком (.,!?:;). За запис ових речи је потрено 32 различита карактера (26 слова и 6 знакова интерпункције). Уколико третирамо сваки карактер као једнако вероватан за запис ових речи ће бити потребно 5 бита по симболу, то јест потребно нам је 20 битова за сваку реч и њен интерпункцијски знак.

Уколико направимо речник са 256 најчешћих речи можемо сваку реч кодирати на следећи начин.

1. Ако се реч налази у речнику пошаљемо 0, а затим пошаљемо 8 битова који означавају адресу речи у речнику.

2. Ако се реч не налази у речнику пошаљемо 1, а затим пошаљемо 20 битова који означавају ту реч.

Ефикасност овог алгоритма зависи од процента речи које пронађемо у речнику. Нека је тај проценат p. Онда се просечан број битова по речи може изачунати овако

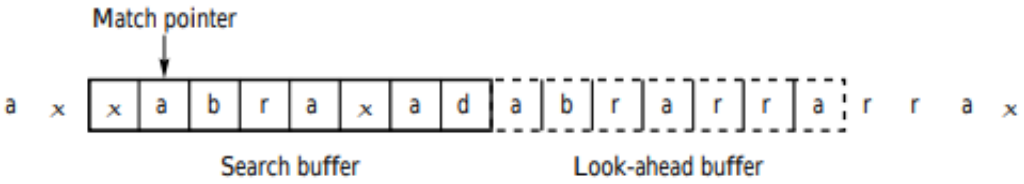
$$R = p \cdot 9 + (1 - p) \cdot 21$$

Да би наш алгоритам био примењив на овом примеру потребно је да R буде мање од 20. То се постиже када $p \geq 0.084$, што делује као мали проценат. Међутим треба нагласити да је у случају кад су све речи једнако вероватне проценат речи пронађених у речнику мањи од 0.00025.

Како би ова метода била што ефикаснија треба максимизовати p. Ово је лако учинити ако нам је улазни фајл унапред познат. Уколико то није случај можемо користити $LZ77$ или $LZ78$ алгоритам.

5.2 $LZ77$ алгоритам

У овом алгоритму уместо речника костимо део већ прочитаног текста. Кодирање користи клизећи прозор подељен на два дела, *search buffer* и *look-ahead buffer*.



Током кодирања померамо показивач у *search buffer*-у све док не наиђемо на почетак неког подниза који се поклапа са почетком *look-ahead buffer*-а. Удаљеност показивача од *look-ahead buffer*-а се назива *offset*. Након тога измеримо дужину поклапања подниза са почетком *look-ahead buffer*-а. Назовимо ову дужину l. Ово поклапање може изаћи ван *search buffer*-а уколико је оно довољно дугачко. Након тога шаљемо тројку $\langle o, l, c \rangle$ где o означава дужину *offset*-а, а c одговара коду првог следећег карактера после поклапања.

Разлог зашто шаљемо следећи карактер је да покријемо случај када нема поклапања, то јест када $l = 0$.

Ако је дужина *search buffer*-а S, а дужина целог прозора W и величина алфабета A. Тада је дужина поруке једнака

$$\lceil log_2(S) \rceil + \lceil log_2(W) \rceil + \lceil log_2(A) \rceil$$

Као што можемо видети $LZ77$ је једноставан адаптивни алгоритам који наизглед не прави никакве претпоставке о изворном фајлу. Овај алгоритам је најефикаснији на изворним фајловима код којих се исте секвенце карактера налазе довољно близу.

5.3 $LZ78$ алгоритам

Уколико се исте секвенце налазе на растојањима већим од величине прозора $LZ77$ може резултирати експанзијом изворног фајла уместо компресијом. Један овакав пример је периодичан низ са периодом већим од дужине прозора. Овај проблем можемо решити применом $LZ78$ алгоритма.

Овај алгоритам се заснива на динамичкој конструкцији речника. Неопходно је да прималац и пошиљалац поруке конструишу речник на исти начин. Поруке се кодирају као парови $\langle i, c\rangle$, где i означава индекс у речнику где се налази најдужа реч која се поклапа са наилазећом секвенцом, а c означава код наредног карактера. Након тога формирамо нову реч додавањем карактера c на крај i-те речи из речника и ту реч додајемо у речник.

Пример 8. Посматрајмо следећу секвенцу

AABABBBABAABABBBABBABB

Речник је празан на почетку, па ће прва порука бити $\langle 0, C(A)\rangle$. Следећи карактер је A, који се већ налази у речнику на позицији 1, па ће следећа порука бити $\langle 1, C(B)\rangle$. Након тога следе карактери A и B, пошто се секвенца AB већ налази у речнику на позицији 2, следећа порука ће бити $\langle 2, C(B)\rangle$. Ако наставимо даље, за кодирање читаве секвенце ће нам требати укупно 9 порука.

1	2	3	4	5	6	7	8	9
A	AB	ABB	B	ABA	ABAB	BB	ABBA	BB
0A	1B	2B	0B	2A	5B	4B	3A	7

Одговарајући речник за ову секвенцу ће изгледати овако

0	∅
1	A
2	AB
3	ABB
4	B
5	ABA
6	ABAB
7	BB
8	ABBA

Приметимо да величина речника није ограничена што у неким ситуацијама може представљати проблем. У том случају треба престати са додавањем нових речи када речник постане превише велики и наставити кодирање користећи статични речник.

Литература

[1] Khalid Sayood, *Introduction to Data Compression*, University of Nebraska, 2006

[2] Jeffrey Scott Vitter, *Design and Analysis of Dynamic Huffman Codes*, Brown University, 1987

[3] https://www.cs.toronto.edu/~radford/csc310.S02/week3b.pdf, 9.5.2020.

[4] http://poincare.matf.bg.ac.rs/~jelenagr/2d/huffman.cpp, 9.5.2020.

[5] http://math.mit.edu/~goemans/18310S15/lempel-ziv-notes.pdf, 9.5.2020.

[6] https://en.wikipedia.org/wiki/Data_compression, 9.5.2020.

[7] https://en.wikipedia.org/wiki/Jensen%27s_inequality, 9.5.2020.

[8] https://en.wikipedia.org/wiki/Entropy_(information_theory), 9.5.2020.