**ЮЗУ “Неофит Рилски”**

**Природо-математически факултет**

**Компютърни системи и технологии**

**Курсова работа**

Тема: Компресиране на данни

**УВОД**

През последните години беше постигнат наистина забележителен напредък в областта на технологиите за запис и съхранение на данни. Само допреди десетина години човек можеше да побере на 2-3 дискети ( по 180-360 КВ) целия необходим му за работа софтуер, а капацитетът на тогавашните твърди дискове- 10-20 МВ, се считаше за астрономически. По- късно нещата се промениха, капацитетът на запаметяващите устройства започна да нараства, а заедно с това увеличиха и изискванията на използвания софтуер.

Успоредно с усьвьршенстването на хардуерните технологии започна развитие и на съответни софтуерни решения, позволяващи по-ефективно използуване на наличните ресурси на системата. Появиха се компресиращи програми (COMPRESS, PKZIP, ARJ, RAR, WinZip и др.), които позволяваха данните да се преобразуват по подходящ начин така, че да се намали необходимото пространство. Положени са усилия за развитие на компресиране на графика, звук и видео.С настъпването на Интернет се появява нужда от кодиране и предаване данни по мрежа. Кодирането е обширна област но настоящата работа е посветена на алгоритмите за компресиране. За компресиращ алгоритъм ще съществуват входни данни (съобщения), чиято дължина при компресиране ще се запазва или дори ще нараства.

**Обща класификация**

Методите за компресиране на данни могат да се класифицират по различни признаци.

В зависимост от резултата който се получава при декомпресирането методите биват:

-със загуба на данни;

-без загуба на данни.

В зависимост от типа на компресиране:

**Кодиране на последователности**

Това е прост метод, при който последователности от повтарящи се букви се кодират чрез двойка, съставена от буквата и броя на последователните й появи.Методът може да бъде много ефективен при черно-бели графични изображения,факсове и др.

**Статически (вероятности) методи**

Основават се на статистически наблюдения за вероятността на поява на буквите (по-рядко групи от букви) във входното съобщение.Идеята е че ако са известни честотите на срещане за всяка от буквите, може да се кодират с някакъв неравномерен код съпоставяйки на най-често срещаните букви по-кратки битови последователности.

**Речникови методи**

Речниковите методи извършват кодиране на цели думи, изречения и произволни последователности.За целта се съставя таблица (речник) на съответствията. Всяка дума се кодира с индекса си в речника, а в случай, че липсва там се предава без изменение.Речниковото кодиране е много ефективно при кодиране на текстови файлове, като в някои случаи размерът на входното съобщение може да се намали няколко пъти.

**Вълнови методи**

Използват се при компресиране на звук, графични и видеоизображения.Входната последователност се описва с помощта на набор от вълни (функции) въз основа на някакви нейни характеристики.Постига се много висока степен на компресия, за сметка на невъзможност за 100% точно декомпресиране.Въпреки това при декомпресиране се полуава резултат, достатъчно близък до оригинала, което за такива файлове е допустимо.

**Фрактални методи**

Тук целта е намиране на формула, описваща данните.Фракталните функции позволяват представяне на сложни данни чрез прости функции, при което се постига огромна степен на компресия.

**Адаптивно компресиране**

Адаптивното компресиране позволява самонастройване на кодиращия алгоритъм въз основа на локалните характеристики на кодираната последователност така, че максимално да се възползва от тях.Повечето от широкоразпространените методи за компресиране имат адаптивен вариант.

**1.Кодиране на последователности.**

**1.1.Премахване на нулите**

Премахване на нулите ( null suppression) e една от най-ранните техники за компресиране. Днес тя има ограничено приложение,главно при комуникационния протокол IBM 3780 и някои текстови редактори.

Основната идея на метода е дългите последователности от нули да се заменят с кодираща двойка, сьстояща се от специален символ, указващ наличието на компресия,последван (или предшестван) от брояч, указващ броя последователни нули. Процесьт на декодиране е тривиален и се свежда до просто заместване на кодиращата двойка със съответния брой нули. Най- често за брояча се отделя един байт. По този начин могат да се кодират последователности с дължини 0,1,.....255. Ясно е че 0 е безсмислена като стойност на брояча. Ето защо може да се измести с 1 наляво интервала от стойности, получавайки се: 1,2,.....256.Например:

12 17 86 93 0 0 1 2 0 0 0 0 0 19 20 0 8 3 12 0 0 0 6

След кодиране се получава:

12 17 86 93 **0 1** 1 2 **0 4** 19 20 **0 0** 8 3 12 **0 2** 6

Понякога броят на последователните нули може значително да надвишава 256. Едно възможно решение в такъв случай е да се използват повече от една кодиращи двойки за една и сьща последователност. Така например последователност от 1000 нули се кодира по следния начин:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **първа двойка** | | **втора двойка** | | **трета двойка** | | **четвърта двойка** | |
| Специален символ | 255 | Специален символ | 255 | Специален символ | 255 | Специален символ | 231 |

*Таблица1.Кодиране на последователност от 1000 нули*

В случай на по-дълги последователности от нули може да се окаже по-ефективно за брояча да се отдели повече от един байт.Това води до премахване на ненужното повторение на специалния символ в подобни случаи, но увеличава пространството, заемано от брояча.Ето защо брояча трябва да се избира много внимателно.

В случай, че специалния символ и броячът заемат по един байт, се оказва неефективно кодирането на единичните нули.Действително, в този случай това води не до намаляване, а до увеличавне размера на кода.Кодирането на последователности от две нули също не дава нищо, тъй като дължината се запазва.Кодирането следва да се прилага само към последователности от поне два (три) символа.С увеличаването дължината на брояча нараства и общата дължина на кодиращата двойка,а оттам и на последователностите, които няма да се кодират.

Основен проблем при премахването на нулите е изборът на специален символ.Обикновено това е символ, за който може да се гарантира със сигурност, че не се съдържа във входното съобщение. В случай, че текущата кодова таблица не съдържа такъв символ, тя следва да се разшири по подходящ начин.Едно възможно решение е кодиращата двойка да се загради с двойка допълнителни редки символи SI (shift in) и SO(shift out).

|  |  |  |  |
| --- | --- | --- | --- |
| **SI** | специален символ | брояч | **SO** |

*Таблица 2.Заграждане на кодиращата двойка с друга двойка редки символи*

Така ще могат да се кодират ефективно последователности от поне 4 (5) нули.

Ползата от прилагането на описания метод силно зависи от съдържанието на входното съобщение.То се компресира добре, само ако съдържа достатъчно дълги последователности от нули.

**1.2.Компресиране с битови карти**

В случай, че някой от символите от входното съобщение се среща доста по-често от останалите (например интервал, нула и др.) е удобно използването на битови карти.Битовата карта представлявапоследователност от 0 и 1, всеки бит на която показва присъствието (стойност 1) или отсъствието (стойност 0) на въпросния символ. Кодирането с битови карти може да се разглежда като обобщение на премахването на нулите.Тук от значение е само вероятността на срещане на символа, без значение дали срещанията са в дълги последователности или не. С помощта на битова карта може да се реализира премахване на нулите.Например:

0 13 0 0 89 0 37 0

Вижда се, че 0 е доминиращ символ и се среща с вероятност 5/8.Същевременно, компресирането чрез премахване на нулите не носи нищо, тъй като най-дългата последователност от нули е с дължина 2.Използването на битова маска води до значително съкращаване дължината на съобщението.В горното съобщение ненулеви елементи има на позиции 2,5 и 7.Така за съответната битова маска се получава 01001010.За да се получи кодираното съобщение, след битовата карта трябва да се запишат ненулевите символи по реда на появата им,т.е.

74 13 89 37

Постига се компресия от 50% намалявайки броя на необходимите байтове от 8 на 4.В случай, че нулата се среща пo-рядко степента на компресия зависи от честотата на срещане на нулата в последователност от 8 символа. Забелязва се, че при по-малко от 12,5% нули дължината на кодираното съобщение нараства, вместо да намалее.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Брой нули** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **%нули** | 0 | 12,5 | 25 | 37,5 | 50 | 62,5 | 75 | 87,5 | 100 |
| **Дължина на кода в байтове** | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| **Степен на компресия** | 0,888 | 1,000 | 1,143 | 1,334 | 1,600 | 2,000 | 2,667 | 4,000 | 8,000 |

*Таблица 3. Дължина на кода и степен на компресия при различен брой нули*

Основен недостатък на битовите карти е, че са приложими само върху последователности с фиксирана дължина в битове: символи,байтове, думи и др.Директно следствие от тован е, че дължината на кодираното съобщение е обратнопропорциална на вероятността за срещане на нула.В случай, че има няколко често срещащи се символа с близки честоти на срещане, битовата карта позволява съкращаване на кода, пропорционално само на едната честота и по никакъв начин не отчита високата вероятност за срещане на другия символ.Решение на проблема дава кодирането на последователности.

Декопдирането става по следния начин: прочита се битовата карта, след което се прочитат толкова символа, колкото е броят на единиците в нея. Между символите евентуално се вмъкват нули, съобразно данните от битовата карта. След това се прочита нова битова карта и т.н. Процесът продължава до цялостното декодиране на съобщението.

**1.3.Полубайтово пакетиране**

Полубайтовото пакетиране се основава на структурата на кода на някои от символите в кодовата таблица. При някои кодови таблици част от кода на цял набор от важни символи се повтаря. Например в EBCDIC старшите четири бита в представянето на цифрите 0,1.....9 са единици:

|  |  |  |  |
| --- | --- | --- | --- |
| **цифра** | **двоичен запис** | **16-ичен код EBCDIC** | **Двоичен кодEBCDIC** |
| 0 | 0000 0000 | F0 | 1111 0000 |
| 1 | 0000 0001 | F1 | 1111 0001 |
| 2 | 0000 0010 | F2 | 1111 0010 |
| 3 | 0000 0011 | F3 | 1111 0011 |
| 4 | 0000 0100 | F4 | 1111 0100 |
| 5 | 0000 0101 | F5 | 1111 0101 |
| 6 | 0000 0110 | F6 | 1111 0110 |
| 7 | 0000 0111 | F7 | 1111 0111 |
| 8 | 0000 1000 | F8 | 1111 1000 |
| 9 | 0000 1001 | F9 | 1111 1001 |

*Таблица.4 Представяне на десетичните цифри в EBCDIC*

Eдна възможност за компресиране в този случай дава методът на полубайтовото пакетиране. Идеята е в един байт да се записват две цифри. За целта едната цифра се записва в старшия, а другата – в младшия полубайт. Декодирането става чрез отделяне на двата полубайта и прибавяне пред всеки от тях на липсващата четворка битове: в случая 1111. Така например, числото 38 ще се кодира като 00111000.

**1.4.Съвместно използване на битови карти и полубайтово пакетиране**

Някои от по-старите приложения работят със 7-битов ASCH код, при който най-старшият бит не се взема предвид, а служи само за проверка по четност при предаване на данни. Така кодовете 00100100 и 10100100 представляват един и същи символ, а именно “$”. В такъв случай може да се опрости кодиращата група, премахвайки специалния символ. Броячът ще има най-старши бит 1, а останалите символи-0.

Тази прилика навежда на мисълта за създаване на хибриден вариант, съчетаващ предимствата на двата метода. Това се постига лесно при използването на още един бит. Ако стойността му е 1, то има кодиране с битова карта, в противен случай е налице полубайтово пакетиране. В първия случай следващите 6 бита следва да се интерпретират като 6-битова карта, а във втория – като 6-битов брояч, приемащ 64 различни стийности.

**1.5.Двуатомно кодиране**

Двуатомното кодиране е обобщение на полубайтовото пакетиране. Ако се предположи, че разполагаме с достатъчно допълнителни символи, несрещащи се във входната последователност може да се постигне до 50% степен на компресия, съпоставяйки на най-често срещаните двойки последователни символи единствен съответен символ. Ключов момент е изборът на подходящите двойки, които могат да бъдат включени в кодирането. Най-добра ефективност се постига при кодирането на най-често срещаните двойки. Това обаче предполага наличието на някаква предварителна статистика относно честотата на срещане на всяка от тях. В случай, че липсва нужната статистика, може да се определят двойките с едно преминаване през входното съобщение. Това обаче е възможно само за крайни съобщения, води до забавяне на алгоритъма и изисква предаване на речника, но от друга страна по-добре моделира съобщението. Затова е желателно използването на предварително фиксирана таблица. Това следва да се извърши зиключително внимателно и изисква статистика относно типа и съдържанието на предаваното съобщение.

Джуел предлага следната таблица в случай на текстов файл на английски език:

|  |  |  |
| --- | --- | --- |
| **двойка** | **Общ брой решения** | **Среден брой срещания на хиляда символа** |
| E\_ | 328 | 26,89 |
| \_T | 292 | 23,94 |
| TH | 249 | 20,41 |
| \_A | 244 | 20,00 |
| S\_ | 217 | 17,79 |
| RE | 200 | 16,40 |
| IN | 197 | 16,15 |
| HE | 183 | 15,00 |
| ER | 171 | 14,02 |
| \_I | 156 | 12,79 |
| \_O | 153 | 12,54 |
| N\_ | 152 | 12,46 |
| ES | 148 | 12,13 |
| \_B | 141 | 11,56 |
| ON | 140 | 11,48 |
| T\_ | 137 | 11,23 |
| TI | 137 | 11,23 |
| AN | 133 | 10,90 |
| D\_ | 133 | 10,90 |
| AT | 119 | 9,76 |
| TE | 114 | 9,35 |
| \_C | 113 | 9,26 |
| \_S | 113 | 9,26 |
| OR | 112 | 9,18 |
| R\_ | 109 | 8,94 |

*Таблица.5.Най-чести двойки последователни букви в английски текст*

**1.6 Замяна на шаблони**

Замяната на шаблони може да се разглежда като обобщение на двуатомното кодиране. Идеята е да се кодират не двойки символи, а произволни думи. Така ограничението от 50% за степента на компресия отпада автоматично. Методът демонстрира скромна ефективност при обикновен текстов файл, но дава много добри резултати при кодиране на компютърни програми. Подобно на двуатомното кодиране, и тук се изгражда таблица на съответствията. Нами­рането на идеалната таблица по принцип е сложен проблем, но в някои конкретни случаи кан­дидатите за заместване са очевидни. В български език евентуални кандидати за кодиране са думи като:“не**”,** “на”, "от", "за", "който" и др: В английски някои кандидати са: "the", “for”,”of” и др. Тези думи най-често носят силно ограничен собствен смисъл, поради което се включват в списъка на стоп-думите*,* т.е. на пропусканите при индексиране и търсене от търсещи­те машини в Интернет.

Най-висока степен на компресия се получава при кодиране на компютърни програ­ми. написани на език за програмиране (от високо ниво). Възможни кандидати в този случай са предимно запазените думи на езика, както и някои от често срещаните имена на променливи. Таблица 6 показва някои възможни какдидати за някои езици за програмиране от високо ниво.

|  |  |
| --- | --- |
| Език за програмиране | Възможни кандидати за заместване |
| Бейсик | for,goto,gosub,if,input,let,next,print,rem,then |
| Си | do,else,for,if,int,void,while |
| Фортран | do,format,read,write |
| Паскал | if,else,for,function,procedure,repeat,then,until,whrite |

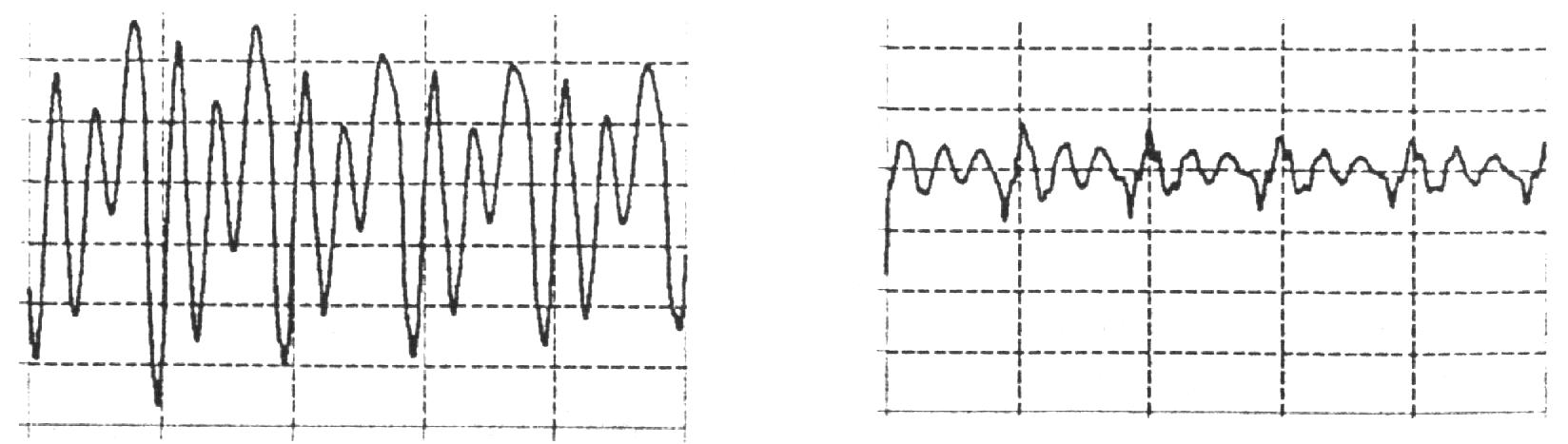
*Таблица .6.Кандидати за заместване при някои езици за програмиране*

# **1.7.Относително кодиране**

Понякога входното съобщение не съдържа дълги последователности от повтарящи се символи, а множество от близки едно до друго числа. В този случай бихме могли да постигнем добри резултати, използвайки друг подход. Нека е дадено входно съобщение, представляващо последователност от години:

1991 1992 1990 1992 1995 1994 1991.

Ясно е, че за кодирането на всяка година са достатъчни 11 бита (211 = 2048). Бихме могли обаче значително да съкратим дължината на съобщението, запазвайки само първата година и представяйки останалите чрез относителното им отместване спрямо нея. Така получаваме:

1991 1 - 2 2 3 - 1 0 - 3

|  |  |
| --- | --- |
| *Фигура 1.а.  Сигналът преди кодиране.* | *Фигура 1.б.  Сигналът след кодиране* |

Относителното кодиране е особено ефективно при архивиране на последователните версии на даден текст. За целта оригиналният текст се запазва изцяло върху диска (лентата), а за всяка следваща редакция се пазят само разликите спрямо предходната версия. Възможни са редица вариации на изложения метод. Всички разлики могат да бъдат зададени не спрямо предходната редакция, а спрямо първоначалната версия на текста. Така се премахва необходимостта от изчитане на всички предходни версии при декодирането. На практика се получава проста транзакция на текста. Когато броят на версиите започне да нараства, е разумно да се помисли за кодиране, при което на всеки 30 копия например, се сменя началният файл, използван при транзакцията, т.е. въвежда се ново начало на "координатната система".

Много добри резултати се постигат при комбиниране на няколко метода на компресиране. Добър пример в това отношение са факс-машините.

Операцията ⊕ удовлетворява следните равенства:

1 ⊕ 0 = 0 ⊕ 1 = 1

0 ⊕ 0 = 1 ⊕ 1 = 0

Целта на тази стъпка е да се получат повече достатъчно дълги еднородни последователности от нули и от единици. Ясно е, че тя запазва дължината на съобщението, но сама по себе си е безсмислена. Така обаче се подготвя по-добра входна последователност за втората стъпка, при която обикновено се прилага разгледаното по-горе премахване на нулите. В случай на предаване на текст този метод работи особено добре, тъй като след първата стъпка ще се получи достатъчно дълги еднородни последователности от "0" и "1". Най-добър резултат ще се получи при бели (черни) страници.

# **1.8.Математическо очакване**

Математическото очакване, известно още като средна стойност, е основно понятие в теорията на вероятностите. Математическото очакване на дискретната случайна величина Х, приемаща стойности х1, х2, … хn с вероятности Р (х1), Р(х2), … Р (хn) се пресмята по формулата:

.

Статическото определение за вероятност на случайната величина Х се свързва с извършване на някои опити и наблюдения на Х и се дефинира като отношението на броя на благоприятните изходи към общия брой опити. Да разгледаме случайна величина, която приема стойности 0 или 1 в зависимост от това дали се пада лице или гръб при хвърляне на монета. Да предположим че сме подхвърлили монетата във въздуха 10 000 пъти, при което 5 037 пъти се е паднал герб. В такъв случай вероятността тази случайна величина да приеме стойност 1, т.е. да се падне герб, е   
5 037/10 000 = 0,5037.

В процеса на преминаване през символите от входното съобщение се натрупват статистически данни, позволяващи да пресметне математическото очакване на следващата буква, като на изхода се подават само "грешката", т.е. само разликата между средната стойност и кода на символа. Вместо да се натрупва статистика за символите на входното съобщение, както и за честотата (или вероятността) на срещане на всеки от тях се използва проста рекурентна връзка между старата стойност на   
ЕХстар = (х1, х2, … хк) и новата ЕХнов= (х1, х2, … хк, хк-1), т.е. след прибавяне на новия символ. В сила е връзката

.

Същата формула се използва и при декодирането.

Освен еднопасовата обработка изложеният метод има същественото предимство, че отчита локалните особености на входното съобщение. Така например, ако даден сегмент на текста съдържа символа х с вероятност 70%, докато вероятността на срещане на х в целия текст е едва 3%, то описаният метод на кодиране с линейно предсказване ще успее да отчете тази особеност и да коригира по подходящ начин математическото очакване за този сегмент. Описаният метод може да се разглежда като статистически и адаптивен едновременно.

Алгоритъмът води до намаляване на амплитудата на изменение на кодираните стойности, като ги приближава до нулата от двете й страни. Това означава по-малки по абсолютна стойност числа (и по-малко като разнообразие от различни стойности), а оттук - и по-малко необходими двоични битове за запазването им: старшите битове просто могат да се отхвърлят. От друга страна, кодираното съобщение често съдържа дълги последователности от съседни (или почти съседни) еднакви числа. Което е добър вход за други алгоритми като кодирането на последователности или с битови карти.

# **1.9.Кодиране на последователности**

Кодирането на последователности е естествено обобщение на премахването на нулите. Класическото премахване на нулите допуска евентуалното кодиране на последователните повторения на произволен, но единствен и предварително фиксиран символ, обикновено най-често срещаният. За съжаление, най-често срещаните символи могат да бъдат повече от един.

Да предположим, че съобщението съдържа само букви. Тогава може значително да намали дължината на горния низ, заменяйки всяка последователност от еднакви символи с един-единствен негов екземпляр, предшестван от число, указващо броя на срещанията му. Изложеният метод може да доведе до значително съкращаване дължината на съобщението, особено при по-дълги последователности от еднакви символи.

# **2.Статистически методи**

# **2.1.Алгоритъм на Шенън-Фано**

Втората световна война дава силен тласък на развитието на кодирането във всяко едно от трите основни направления: шумозащитно кодиране, криптиране и компресиране. Правят се редица изследвания и теоретични разработки, като постепенно започва да си проправя път идеята, че за ефективното кодиране на едно съобщение е достатъчно да се знае вероятността на срещане на всеки от символите в него. По това време се достига и до идеята за използването на двоичната бройна система като основа на кодиращия алгоритъм. Последното представлява наистина сериозен пробив, особено предвид на факта, че по това време все още няма компютри. Първият универсален ефективен алгоритъм за кодиране е разработен от Клод Шенън, Bell Labs и М.Фано, М/Т, като двамата го предлагат почти едновременно и независимо един от друг. Основната идея на алгоритъма е по зададени вероятности на срещане на всеки от символите във входното съобщение да се съпостави двоичен код.

Алгоритъмът притежава следните характерни свойства:

1. Дължината на кодовете е променлива.
2. Буквите с по-голяма вероятност се кодират с по-малко битове от тези с по-малка вероятност.
3. Съобщението се декодира еднозначно.

Код, удовлетворяващ 3-тото изискване се нарича разделим. Очевидно, всички кодове с фиксирана дължина са разделими. Декодирането в този случай се извършва чрез разбиване на кодираното съобщение на подпоследователности (поддуми) с дължина, равна на дължината на кода, след което се извършва просто заместване на всяка поддума с първообраза й съгласно таблицата на кодиране. В случай, че някоя от поддумите няма първообраз, декодирането е невъзможно, а в останалите случаи винаги се гарантира еднозначност. Пример за равномерен код е ASCll.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| буква | честота | буква | честота | буква | честота | буква | честота | буква | честота | буква | честота |
| А | 12,68% | Е | 9,64% | К | 3,43% | П | 2,50% | Ф | 0,78% | Щ | 0,60% |
| Б | 1,08% | Ж | 0,76% | Л | 3,04% | Р | 4,67% | Х | 0,35% | Ъ | 1,56% |
| В | 4,77% | З | 2,21% | М | 3,30% | С | 4,41% | Ц | -,87% | Ь | 0,02% |
| Г | 1,01% | И | 8,54% | Н | 8,17% | Т | 7,80% | Ч | 1,62% | Ю | 0,10% |
| Д | 3,33% | Й | 0,48% | О | 9,05% | У | 1,44% | Ш | 0,18% | Я | 1,61% |

*Таблица 7.   
Средна вероятност за срещане на буквите от българската азбука.*

Равномерните кодове дават ограничена възможност за намаляне дължината на входното съобщението. Действително, ако кодира текст, съдържащ само главните латински букви и интервала (общо 27 на брой символа). Може да се използва 5-битов равномерен код, вместо стандартния 8-битов ASCll съкращавайки по този начин дължината на съобщението с 37,5%. В 5 бита могат да се запишат 25 = 32 различни стойности, а трябва да се кодират една 27 букви.

Използването на равномерни кодове по никакъв начин не отчита честотите на срещане на символите във входното съобщение. Интуитивно обаче е ясно, че ако се кодират с по-малко битове символите с по-голяма вероятност на срещане могат да се получат значително по-добри резултати.

Въпреки посоченото предимство, неравномерните кодове имат редица недостатъци. На първо място следва да се посочи изключителната им чувствителност към грешни битове. Докато при равномерните кодове сгрешаването на един бит ще води до повреждане само на един символ. При неравномерните това може да доведе до невъзможност за декодиране на цялото съобщение до края. Освен това не винаги може да се гарантира еднозначно декодиране.

Съществуват различни класове разделими кодове, като най-често използвани са така наречените префиксни кодове. Един код е префиксен, ако кодът на никой символ не е префикс (начало) на кода на никой друг символ. Всеки префиксен код е разделим. Обратното обаче не е вярно.

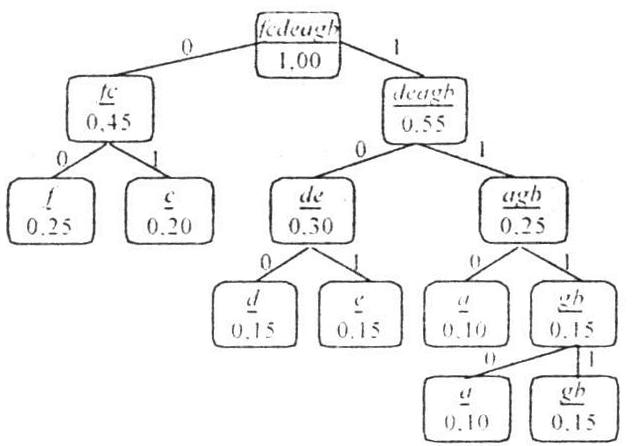
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **символ** | **вероятност** | **код** | | | |
|  |  | *стъпка 1* | *стъпка 2* | *стъпка 3* | *стъпка 4* |
| *f* | 0,25 | 1 | 1 |  |  |
| *c* | 0,20 | 1 | 0 |  |  |
| *d* | 0,15 | 0 | 1 | 1 |  |
| *e* | 0,15 | 0 | 1 | 0 |  |
| *a* | 0,10 | 0 | 0 | 1 |  |
| *g* | 0,10 | 0 | 0 | 0 | 1 |
| *b* | 0,05 | 0 | 0 | 0 | 0 |

Таблица 8. Алгоритъм на Шенън-Фано за източника: (*а*:0,1), (*b*:0,05), (*c*:0,20), (*d*:0,15), (*e*:0,15), (*f*:0,25) и (*g*:0,10).

# **2.2.Алгоритъм на Шенън-Фано**

1. Сортираме символите по вероятност на срещане в намаляващ ред.
2. Разделяме множеството от символите на две подмножества с (почти) равни вероятности.
3. На едното подмножество съпоставяме 0, а на другото - 1 като поредна буква.
4. Ако някое от подмножествата съдържа повече от един елемент, прилагаме за него същия процес, започвайки от стъпка 2.

Пример. Нека е дадено някакво множество от символи заедно с вероятностите на поява от тях: (*а*:0,1), (*b*:0,05), (*c*:0,20), (*d*:0,15), (*e*:0,15), (*f*:0,25) и (*g*:0,10). Следвайки горния алгоритъм се извършват съответно сортиране: *f*, *c*, *e*, *a*, *g*, *b*. След това последователно разделят на групи.

Получава се кода: (*а*:001), (*b*:0000), (*c*:10), (*d*:011), (*e*:010), (*f*:11) и (*g*:0001). Описаният процес на разделяне всъщност строи двоично дърво, по върховете на което се разположени множествата: в корена е входното множество, съдържащо всички букви от азбуката (с положителна честота на срещане), а в листата са всичките му едноелементни подмножества, т.е. отделните букви. Разделянето на дадено множество на две подмножества се интерпретира като създаване на два наследника на съответния връх. Това дърво се нарича кодиращо.

Фиг.2. Кодиращо дърво на Шенън-Фано за източника:  
(а:0,1), (b:0,05), (c:0,20), (d:0,15), (e:0,15), (f:0,25) и (g:0,10).

Започваме от корена на дървото: Ако текущият бит е 0, тръгваме по левия наследник, ако е 1 - по десния. При достигане до листо записваме на изхода кода на съдържащия се там символ, след което продължаваме декодирането, започвайки отново от корена.

Алгоритъмът на Шенън-Фано не винаги строи оптимални кодове. Основният проблем е в начина на разделяне. В случай на почти равно разделяне е възможно да възникнат две възможности, което да доведе до две различни дървета с различни тегла.

# **2.3.Алгоритъм на Хъфман**

Хъфман успява да избегне недостатъците на алгоритъма на Шенън-Фано, като строи дървото от листата към корена.

# Алгоритъм на Хъфман

1. Образуваме от всеки символ тривиално дърво в корена (единствения връх), на което записваме вероятността на срещане на съответния символ.
2. Намираме двата върха с най-малки вероятности и ги обединяваме в ново дърво с корен, съдържащ сумата от вероятностите им.
3. Ако има поне две дървета, преход към 2.

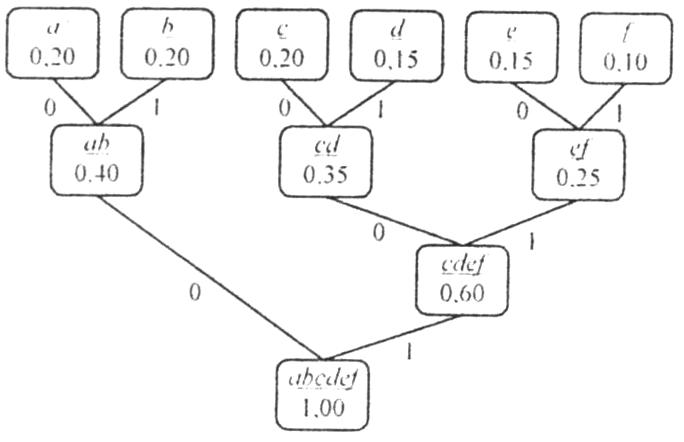
Алгоритъмът на Хъфман строи дърво със същите свойства като изгражданото от алгоритъма на Шенън-Фано. Търсеният код, както и декодирането на съобщението се извършва абсолютно аналогично. Тук обаче символите с по-голяма честота се включват по-късно в дървото. Така те се оказват по-близо до корена, т.е. съпоставя им се по-кратък код.

Най-важното свойство на дървото на Хъфман е, че винаги гарантира построяването на оптимален побуквен код, за разлика от алгоритъма на Шенън-Фано, който както вече видяхме понякога греши. Ако на някоя стъпка има повече от една възможности, се избира коя да е от тях. Всички дървета на Хъфман, построени по един и същи честоти, имат еднаква дължина на претегления външен път.

Пример:

Пресмятаме честотите на срещане на отделните символи и построяване дървото на Хъфман.

Дървото от фигура 3. е начертано обърнато по отношение на дървото на Шенън-Фано. Това е направено нарочно с цел да се подчертае, че се изгражда от листата към корена, за разлика от алгоритъма на Шенън-Фано.



Фиг.3. Кодиращо дърво на Хъфман за източника:  
(а:0,20), (b:0,20), (c:0,20), (d:0,15), (e:0,15), (f:0,10).

Горното дърво ни дава следните кодове:

*а* = 00, *b =* 01, *c =* 100, *d* = 101, *e =* 110 и *f =* 111. Кодираното съобщение ще изглежда така (оставили сме разстояния с цел по-лесна читаемост):

00 111 01 00 01 100 101 110 111 00 100 01 00 01 100 101 110 100 101 110

За съхраняването на входното съобщение в стандартен 8-битов АSll формат ще са необходими 8,20 = 160 бита. От друга страна, той като имаме само 6 различни символа, можем да използваме 3-битов равномерен код, при което ще са ни необходими 3,20 = 60 бита. Кодирането по Хъфман обаче изисква едва 52 бита, т.е. средно 52/20 = 2,6 бита за символ.

Зависи от броя *n* на различните символи, участващи във входното съобщение, но не и от съдържанието му. Тривиалният подход при решаването дава сложност - (*n*2). Действително, на всяка стъпка се обединяват точно две дървета, т.е. броят на стъпките е пропорционален на *n.* Ако на всяка стъпка за избора на двете дървета с минимални честоти ни е необходимо време, пропорционално на *n*, то получаваме посочената по-горе сложност. Броят на стъпките не би могъл да бъде намален.

Съставяне: ще съберем нужните ни статически данни за честотата. Ще работим с честота, вместо вероятност на срещане на всеки от символите от входното съобщение MSG. Така няма да се налага да работим с реални числа. След това за всеки символ се ненулева честота на срещане ще изградим съответно тривиално дърво, съставящо се от единствен възел - символа, който представя (виж по-долу функцията *init Model ( )*). Получаваме *Хъфманова гора*. На всяка стъпка ще избираем двете дървета с най-ниски тегла по начина, описан по-горе. Дърветата ще изграждаме като динамични структури от данни и за всяко от тях ще пазим указател в специален масив от дървета *forestц* [ ]. Освен въпросния указател масивът ще съдържа и данни за теглата на всяко от дърветата: сумата от честотите на срещане на символите от листата на дървото. На не листата няма да съпоставяме символи, а само честоти на срещане. Най-общо алгоритъмът на Хъфман изглежда така.

След построяването на кода ще можем да изведем дървото на Хъфман (функция *print Tree ( )*), както и кодовете на всички символи с ненулева честота на срещане (функция *write Codes ( )*) на екрана. Следва пълна реализация (функцията *prin Mins ( )* намира двете най-редки дървета).

*Резултат от изпълнението на програмата:*

Дърво на Хъфман за afbabcdefacbabcbcdecde:

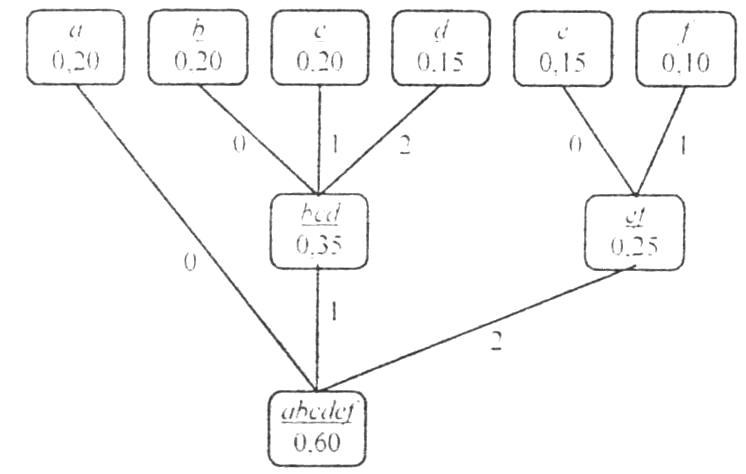
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | 4 | b |  |
|  |  | 8 |  |  |  |
|  |  |  | 4 | c |  |
|  | 20 |  |  |  |  |
|  |  |  |  | 2 | f |
|  |  |  | 5 |  |  |
|  |  |  |  | 3 | d |
|  |  | 12 |  |  |  |
|  |  |  |  | 3 | e |
|  |  |  | 7 |  |  |
|  |  |  |  | 4 | a |
| b | = | 00 |  |  |  |
| c | = | 01 |  |  |  |
| f | = | 100 |  |  |  |
| d | = | 101 |  |  |  |
| e | = | 110 |  |  |  |
| a | = | 111 |  |  |  |

# **2.3.Обобщен алгоритъм на Хъфман**

Идеята на разгледания по-горе класически двоичен Хъфманов код може да се обобщи по естествен начин за построяване на троичен, четвъртичен и изобщо *n*-ичен Хъфманов код. Да разгледаме задачата за намиране на оптимален троичен код, т.е. съставен не от две, а от три различни цифри (например 0, 1 и 2), по зададена вероятност за срещане на всеки от символите от входното съобщение. За целта се изгражда *тернарно* дърво, т.е. всеки негов връх има до 3 наследника включително. Подобно на двоичното дърво, където имахме ляв и десен наследник, и тук за наследниците има фиксирана наредба. Ще ги наричаме: ляв, среден и десен. Как се изгражда дървото? На всяка стъпка се избират трите букви с най-малка вероятност на срещане и се обединяват в нова обобщена буква с вероятност, равна на сумата от вероятностите им. Процесът продължава до обединяване на всички букви в обща супербуква с вероятност 1.

Тъй като на всяка стъпка се извършва обединяване на три (обобщени) букви в една обобщена супербуква, то броят на буквите намалява с 2. В случай на четен брой букви, на последната стъпка ще останат само две букви. Това е доста неприятно, тъй като ни лишава от фундаментален клон на дървото, а оттук евентуално и от някои ценни къси кодове. Очевидно в общия случай такова дърво няма да бъде оптимално. Не е трудно да се забележи, че колкото по-далеч от корена е разположено двойното разклонение, толкова повече къси кодове ще имаме на разположение. Остава да съобразим, че алгоритъмът на Хъфман строи дървото от листата към корена. Тогава за получаване на оптимален код на първата стъпка следва да се изберат двете най-малко вероятни букви, при четно *n*, и трите най-малко вероятни букви, при нечетно *n*. На всяка следваща стъпка се избират трите най-малко вероятни букви.

Кодът се получава: Проследява се пътят от корена до съответното листо. Всеки път, когато преминаваме към ляв наследник пишем 0, към среден - 1, а към десен - 2. Ще илюстрираме метода върху същия пример, върху който илюстрирахме класическото двоично кодиране по Хъфман, виж фигура 4.



Фиг.4. Тернарно кодиращо дърво на Хъфман за източника:  
(а:0,20), (b:0,20), (c:0,20), (d:0,15), (e:0,15), (f:0,10).

Кодът е разделим (истинските символи са в листата) и има същите свойства като двоичния Хъфманов код. Да пресметнем средния брой троични битове за кодиране на някоя буква (цената на така конструирания троичен код):

L = 0,20.1 + 0,20.2 + 0,20.2 + 0,15.2 + 0,15.2 + 0,10.2 = 1,8

Оказва се, че всяка буква се кодира средно с 1,8 троични бита. Сравнено с 2,6 при двоичния Хъфманов код. Ясно е, че с увеличаване броя на цифрите този показател ще се подобрява. Следва да обърнем внимание обаче, че тези битове са троични!

Как изглеждат нещата при *k*-ичен код? Оказва се, че алгоритъмът на Хъфман строи оптимални кодове и в този случай. На всяка стъпка се избират *k*-те най-малко вероятни букви и се обединяват в обща супербуква. Първата стъпка евентуално прави изключение по същите съображения, както при 3-знаковия случай. Тук се избират *s*-те най-малко вероятни букви, като *s* е това число измежду 2, 3, … *k*, за което *k*-1 дели *n-s*. Кодът, съпоставен на всяка буква, се получава при проследяване на пътя от корена и съпоставяне на съответна цифра, съгласно някаква предварително фиксирана наредба на наследниците. Например, на първия се съпоставя 0, на втория - 1, на третия -2, … на *k*-тия - (*k*-1).

# **2.4.Код с разделители**

Разглежданите по-горе алгоритми на Шенън-Фано притежават безспорни предимства: свойството префиксност, позволяващо еднозначност на декодирането, оптималност и почти оптималност съответно. Същевременно обаче, те имат и някои недостатъци, като най-неприятният от тях е надеждността. Действително, повреждането на единствен бит може да доведе до невъзможност за декодирането на цялото съобщение, поради неравномерността на кода. Ето защо при предаването на такова съобщение се налага предвиждането на строг механизъм за контрол и евентуална корекция на грешките.

Равномерните кодове не са особено ефективни. Нека разгледаме като пример входно съобщение със следните символи и честоти: (*a*:0,4), (*b*:0,2), (*c*:0,2), (*d*:0,15) и (*e*:0,05). При използване на равномерен код всяка буква ще се записва с 3 бита. Същевременно, използвайки кодиране по Хъфман, получаваме кода: (*а* = 11), (*b* = 10), (*c* = 01), (*d* = 001), (*e* = 000). Нека пресметнем цената му:

L = 2.0,41 + 2.0,2 + 2.0,2 + 3.0,15 + 3.0,05 = 2,2

Не бихме ли могли да получим код, съчетаващ висока надеждност и относително добра ефективност, заемащ междинно положение между равномерните кодове и кодирането по Хъфман? Пример за такъв код е така нареченият код с разделители (англ. *сomma code*). При него кодът на всеки символ завършва с разделител, указващ края му. Това позволява точно определяне на началото на следващия символ. Как се конструира код с разделители? Идеята е проста: Сортираме буквите по вероятност на срещане. На първия символ съпоставяме код 1, на втория - 01, на третия - 001, на четвъртия - 0001 и т.н. Очевидно така конструираният код е префиксен, позволява бързо и еднозначно декодиране и в общия случай е по-ефективен от равномерните кодове. Повреждането на единствен бит води до невъзможност за декодиране на най-много две букви.

За горния пример получаваме кода : *а* = 1, *b* = 01, *c* = 001, *d* = 0001, *e* = 00001. Да пресметнем съответната му цена:

L = 1.0,4 + 2.0,2 + 3.0,2 + 4.0,15 + 5.0,05 = 2,25

Получихме сравнително надежден код с ефективност, близка до тази на кода на Хъфман.

# **2.5.Аритметично кодиране**

Кодирането по Хъфман на теория ни дава оптималния побуквен код, основан на честотата на срещане на всеки от символите във входното съобщение. На теория то е оптимално. На практика, ефективността му е доста скромна и при текстови файлове най-често варира между 20% и 25%. Същевременно, практически всички комерсиални програми успяват да се справят по-добре от стандартното кодиране по Хъфман, като в повечето случаи успяват да го бият в пъти. Като че ли теорията лъже… По-горе посочихме една от основните причини за скромната ефективност на кодирането по Хъфман: неподходящ избор на азбука. Оказва се, че има и друг проблем. Нека предположим, че искаме да кодираме факс-изображение. Както по-горе вече споменахме факс-изображенията са черно-бели, т.е. двуцветни. Да предположим, че изображението е 90% бяло. Очевидно кодирането по Хъфман ще съпостави на единия цвят код 0, а на другия - код 1. Получаваме равномерен код, т.е. нулева компресия. На практика, дължината на оптималния код за белия цвят трябва да бъде 0,15 бита, т.е. около шест пъти по-малко. За съжаление, кодирането по Хъфман съпоставя на всяка буква целочислен брой битове, при което не се отчита достатъчно прецизно вероятността за поява на всяка от тях. Как да се справим с проблема? Едно възможно решение може да се търси по посока на смяна на източника. За целта следва да разглеждаме като букви от входната азбука не отделните точки от изображението, а групи от няколко последователни точки. При групи от 8 точки получаваме азбука с 265 символа, позволяваща ни да се възползваме по-добре от високата честота на срещане на белия цвят.

Въпреки това резултатът не винаги ще бъде задоволителен, тъй като избирането на подходящата входна азбука в общия случай е неразрешим проблем. Но дори и да бъде решен за конкретната задача, проблемът остава: Хъфмановото кодиране съпоставя на всяка буква целочислен брой битове, поради което не дава потенциално оптималния резултат (макар да дава оптимален целочислен код). За съжаление, за да се възползваме по-добре от точните вероятности за срещане на всяка от буквите, е необходимо да можем да им съпоставяме нецелочислен брой битове. Как може да стане това?

Един възможен отговор на въпроса дава аритметичното кодиране. Подобно на Хъфмановото кодиране, то също се основава на статистически наблюдения за честотата на срещане на всеки от символите във входното съобщение, но може да постигне значително под-добри резултати, тъй като изобщо се отказва от побуквеното кодиране. Вместо това на цялото входно съобщение се съпоставя едно-единствено дробно число между 0 и 1 с необходимата точност. Как става това? Да илюстрираме метода върху някакво конкретно входно съобщение, например: "АРИТМЕТИКА".

Първото нещо, което трябва да направим, е да построим статистика на вероятността за поява на всяка буква от входното съобщение. След това, въз основа на тези данни, на всяка буква съпоставяме някакъв подинтервал от вероятностната крива, която по дефиниция е ограничена в интервала [0:1]. Подинтервалите трябва да удовлетворят следните условия:

1. затворени са отляво и са отворени отдясно, т.е. имат вида [t:r];
2. не се пресичат;
3. обединението им дава интервала [0:1];
4. дължините им се отнасят така, както се отнасят вероятностите за поява на съответните букви.

|  |  |  |  |
| --- | --- | --- | --- |
| ***Буква*** | ***Вероятност на срещите*** | ***Долна граница*** | ***Горна граница*** |
| *А* | 20% | 0,00 | 0,20 |
| *Е* | 10% | 0,20 | 0,30 |
| *И* | 20% | 0,30 | 0,50 |
| *К* | 10% | 0,50 | 0,60 |
| *М* | 10% | 0,60 | 0,70 |
| *Р* | 10% | 0,70 | 0,80 |
| *Т* | 20% | 0,80 | 1,00 |

Таблица .9.а. Начални дробни интервали за буквите при кодиране на "АРИТМЕТИКА".

Таблица 10.4.5а. показва една от възможните таблици с начални интервали за буквите. Ясно е, че тя не е единствена, тъй като бихме могли да разместим буквите. На практика, съществуват 7 = 5040 различни възможни таблици. Няма значение коя от тях ще бъде избрана, стига при декодирането да се използва същата таблици. По-горе сме избрали тази, при която буквите са подредени по азбучен ред.

|  |  |  |
| --- | --- | --- |
| ***Символ*** | ***Долна граница*** | ***Горна граница*** |
| *няма* | 0,000000000 | 1,000000000 |
| *А* | 0,000000000 | 0,200000000 |
| *Р* | 0,140000000 | 0,160000000 |
| *И* | 0,146000000 | 0,150000000 |
| *Т* | 0,149200000 | 0,150000000 |
| *М* | 0,149680000 | 0,149760000 |
| *Е* | 0,149696000 | 0,149704000 |
| *Т* | 0,149702400 | 0,149704000 |
| *И* | 0,149702880 | 0,149703200 |
| *К* | 0,149703040 | 0,149703072 |
| *А* | 0,149703040 | 0,199703046 |

*Таблица9.б. Кодиране на "АРИТМЕТИКА": дробни интервали за всяка буква.*

Как става кодирането? Започваме с интервала [0,00; 1,00], като постепенно го стесняваме. Първата буква от входното съобщение принадлежи на интервала [0,00; 0,20]. Тогава търсеното число ще лежи в този интервал. Разглеждаме втората буква *Р*. Тя принадлежи на интервала [0,70; 0,80]. Тогава нашето дробно число ще лежи между 70% и 80% от вече определения интервал [0,00; 0,20]. Така получаваме интервала [0,14; 0,16]. Следва буквата *И*, принадлежаща на интервала [0,30; 0,50]. Отрязваме интервала [0,14; 0,16] отляво на 30%, а отдясно - на 50%, стеснявайки го до [0,146; 0,150]. След това постъпва буквата *Т*, на която е съпоставен интервалът [0,80; 1,00]. Извършваме съответно ново стесняване на интервала и получаваме [0,1492; 0,1500]. Таблица 10.4.5б. описва процеса по-ясно.

Процесът продължава до изчерпване на буквите от входното съобщение, при което достигаме до интервала: [0,149703040; 0,149703046]. Той определя еднозначно нашето съобщение. Оказва се, че не е необходимо да пазим и двете граници на интервала, а ни е напълно достатъчно да запазим например долната. На практика, всяко число от интервала [0,149703040; 0,149703046] би могло да се счита за код на нашето съобщение. Формално процесът на кодиране би могъл да се опише с помощта на следния програмен код:

*l* = 0.0; *r* = 1.0;

*while* (*get Symbol* *(ch)*) {

*range* = *r* - *l*;

*l* = *l* + *range* \* *low­*\_*range* *(ch)*;

*r* = *l* + *range* \* *high­*\_*range* *(ch)*;

}

*output (l)*.

А как става декодирането? Нещата протичат аналогично на кодирането. В процеса на кодиране започвахме с интервала, определен от първата буква на входното съобщение, като после непрекъснато го стеснявахме. Така всеки следващ интервал се включваше в предходния. Това означава, че полученият код принадлежи на интервала на първата буква от съобщението. Тогава бихме могли да я изведем на изхода, след което да премахнем ефекта от появата й. Ще получим ново число между 0 и 1, представляващо аритметичния код на останалата част от съобщението. Сега отново можем еднозначно да определим първата буква от този код (т.е. вторият от нашето съобщение), да го изведем на екрана и да премахнем ефекта от него. Получаваме кода на съобщението, без първите му два символа и т.н. Възниква проблемът: Кога да спрем? Оказва се, че това е съществен проблем, тъй като кодът по никакъв начин не ни показва каква е дължината на съобщението. Съществуват два основни метода за справяне с проблема: въвеждане на стоп-символ, указващ края на съобщението, или запазване на дължината на входното съобщение отделно от получения код. В приложената по-долу реализация е избран вторият подход. Таблица 10.4.5в. проследява процеса на декодиране на съобщението.

|  |  |  |  |
| --- | --- | --- | --- |
| ***Аритметичен код*** | ***Буква*** | ***Долна граница*** | ***Горна граница*** |
| 0,149703040 | *А* | 0,00 | 0,20 |
| 0,748515200 | *Р* | 0,70 | 0,80 |
| 0,485152000 | *И* | 0,30 | 0,50 |
| 0,925760000 | *Т* | 0,80 | 1,00 |
| 0,628800000 | *М* | 0,60 | 0,70 |
| 0,288000000 | *Е* | 0,20 | 0,30 |
| 0,880000000 | *Т* | 0,80 | 1,00 |
| 0,400000000 | *И* | 0,30 | 0,50 |
| 0,500000000 | *К* | 0,50 | 0,60 |
| 0,000000001 | *А* | 0,00 | 0,20 |

*Таблица 9.в. Декодиране на "АРИТМЕТИКА" по дробния код.*

Формално процесът на декодиране би могъл да се опише с помощта на следния код:

*for (i = 0; i < message Len; i ++)* {

*symbol = get Symbol (msg);*

*write Out (symbol);*

*range* = *high­*\_*range* *(symbol) - low­*\_*range* *(symbol)*;

*msg =*  *low­*\_*range* *(symbol)*;

msg / = range.

Очевидно, аритметичното кодиране ще се справи по-добре от Хъфмановото в случай на двуцветно изображение с 90% вероятност за поява на бял пиксел. Въпреки това от изложеното по-горе изобщо не става ясно защо и дали то е по-добро в общия случай. Да разгледаме съобщението "БААААААААА" и съответната му таблица 9.г.

|  |  |  |
| --- | --- | --- |
| ***Символ*** | ***Долна граница*** | ***Горна граница*** |
| *няма* | 0,0 | 1,0 |
| *А* | 0,0 | 0,9 |
| *Б* | 0,9 | 1,0 |

*Таблица 9.г. Дробни интервали на буквите за съобщението "БААААААААА".*

Да проследим процеса на кодиране:

|  |  |  |
| --- | --- | --- |
| ***Б*** | 0,900000000 | 1,000000000 |
| ***А*** | 0,900000000 | 0,990000000 |
| ***А*** | 0,900000000 | 0,981000000 |
| ***А*** | 0,900000000 | 0,972900000 |
| ***А*** | 0,900000000 | 0,965610000 |
| ***А*** | 0,900000000 | 0,959049000 |
| ***А*** | 0,900000000 | 0,953144100 |
| ***А*** | 0,900000000 | 0,947829690 |
| ***А*** | 0,900000000 | 0,943046721 |
| ***А*** | 0,900000000 | 0,938742049 |

При нарастване вероятността за срещане на някой от символите степента на компресия ще нараства значително. При кодиране на последователност от 100 000 нули с предварително зададена вероятност за срещане на нула 16 383/16 383 и вероятност за срещане на символ за край на файла 1/16 383, е получава трибайтов код! Същевременно, кодирането по Хъфман би изисквало най-малко 12 501 байта. Разбира се, подобни примери са в значителна степен "нагласени" и рядко се случват на практика. Въпреки това те ни подсказват, че в общия случай следва да се очаква, че аритметичното кодиране ще се дължи по-добре от Хъфмановото. Доказано е, че аритметичното кодиране е оптимално при предложения модел. По-късно ще видим, че променяйки модела, бихме могли да постигнем още по-висока степен на компресия.

Примерна реализация на аритметичното кодиране:

*# include <stdio.h>*

*# include <string.h>*

*# define SHOW\_MORE*

*# define MESSAGE "АРИТМЕТИКА"*

*# define MAX 256*

*struct {*

*double low, high;*

*} sym (MAX);*

*unsigned freq (MAX);*

*void get Statistics (char mesg)* ***/*** Намира броя срещания на всеки символ \*/

*{ unsigned i*;

*for (i = 0; i < MAX; i ++)*

*fred (i) = 0;*

*while (`/0` ! = \* mеsg)*

*fred [ (unsigned char) \* mesg ++] ++;*

*void build Model (char \* mesg) / \** Построява модела \*/

*{ unsigned i, cnt, n;*

*for (n = strlen (mesq), cnt = i = 0; i < MAX; i ++) {*

*sym (i).low = (double) cnt /n;*

*cnt + = fred (i);*

*sym (i).high = (double) cnt /n;*

*void print Model (void)*

*{ unsigned i;*

*printf ("/n ГРАНИЦА");*

*printf ("/n СИМВОЛ ДОЛНА ГОРНА");*

*for (i = 0; i < MAX; i ++)*

*if (freq [i])*

*printf ("/n % 4c %1. 4f % 4f", i, sym (i).low, sym (i).high'*

*}*

*double arithmetic Encode (char mesg)* ***/****\** Извършва аритметично кодиране \*/

{ *double range, low, high;*

*low = 0.0, high = 1.0;*

*while (`/0` ! = \* mesg) {*

*range = high - low;*

*high = low + range \* sym [(unsigned char) \* mesg].high;*

*low = low + range \* sym [(unsigned char) \* mesg].low;*

*# ifdef SHOW MORE*

*printf ("/n% c %1.9f % 9f", \* mesq, low, high);*

*# endif*

*mesg ++;*

*}*

*return low;*

*}*

*char get Symbol (double enc Msg)*

*{ unsigned i;*

*for (i = 0; i < MAX; i ++)*

*if (sim [i].low < = enc Msg && sim [i].high > enc Msg)*

*break;*

*return (char) i.*

Предложената програма представлява само демонстрация на аритметичното кодиране и за използването й за практически цели трябва да се модифицира по подходящ начин.

Процесът на декодиране също изисква някои модификации. Така например, границите следва да се изменят по същия начин, както при кодирането, с аналогични текстове за недостатъчна точност и др. Входното съобщение трябва да се чете на части. Най-добре е за целта да се използва подходящо буфериране. При определянето колко цифри от входното съобщение да се разглеждат едновременно трябва да се спазва условието: разгледани като едно число, те да попадат в интервала между долната и горната граница. Освен това не бива да се забравя, че дясната част на интервала е скъсена с 1.

**3.Адаптивно компресиране**

Основен проблем при статичните методи с фиксирана вероятност е разминаването между очакваната и действителната вероятност на поява на отделните букви. Така се оказва, че при кодиране на често срещани букви се използват по-дълги кодове, а при по-рядко срещани - по-къси. Подобна ситуация възниква дори при отчитане на вероятностите за поява на всяка буква в конкретното съобщение.

За постигане на по-добра статически побуквена компресия е необходимо използваният код да се изменя в процеса на кодиране, отчитайки действителната честота на срещане на буквите. Едно възможно решение е адаптивното компресиране. При него се фиксира някакъв начален код за всяка буква, като в процеса на компресиране кодовете за отделните букви могат да се разменят така, че да се отчитат локалните честоти на срещане. Наборът от кодиращи последователности не се променя, а само се пермутират елементите на таблица на съответствието между буква и побуквен код. Най-често началната таблица се инициализира с код на Хъфман, построен въз основа на честотата на срещане на буквите от входната последователност.

При адаптивното компресиране същи се използва подобна таблица, разширена с още една колона, съдържаща честотата на срещане на отделните букви. В процеса на кодиране таблицата се поддържа сортирана в намаляващ ред по тази колона. При всяко търсене на буква в таблицата се извършва актуализиране на честотата й на срещане. Обосновката на това е, че търсенето е възникнало, защото буквата се е срещала още веднъж в текста. Следва последователно сравняване на тази честота с честотата на предходния елемент дотогава, докато буквата не заеме позицията, съответстваща на новата й честота. В случай на равенство на честотите, буквата се поставя възможно най-нагоре в таблицата. В процеса на размяна се разменят само буквите заедно с честотите, като кодовете остават по местата си. Така по-често срещаните букви ще имат по-къс код. Ще илюстрираме метода върху съобщението.

На всяка стъпка таблицата се изменя заедно с изменението на честотата на срещане на отделните букви. Същевременно, за да бъде успешно декодирането, във всеки един момент трябва да бъде известна текущата таблица на съответствията. Очевидно запазването на всички таблици е невъзможно, тъй като това би довело до значително нарастване на дължината на кода. Оказва се, че не е и необходимо. Действително, ако е известна началната таблица, можем да получаваме всички останали по начина, по който ги получавахме и при кодирането: поддържа се допълнителната колона с честотите на срещане, която на всяка стъпка се актуализира, като това води до евентуално разместване на някои от редовете и т.н.

**3.1.Адаптивно компресиране по Хъфман**

Изграждането на Хъфманово кодиране вдействително изисква предварително изграждане на таблица на честотите (вероятностите) на срещане на всяка буква от буквите от входното съобщение. Съществува подход, при който съобщението се разбива на отделни части, за всяка от които се прилага различен код на Хъфман. Подобен подход извежда на преден план два основни проблема:

1)Как да се определят броят и границите на отделните части?

2)Как да се предпазим от прекомерно нарастване на кода в резултат на предаването на повече кодиращи таблици?

Ето един специален адаптивен метод, който се справя успешно и с двата проблема. Идеята е кодирането да започва с някакъв начален код, който впоследствие да се променя динамично заедно с изменението на вероятностите за срещане на буквите от входното съобщение. Идеята на адаптивното компресиране е универсална и лесно приложима към практически всеки статистически алгоритъм на компресиране. Една адаптивна компресираща схема изглежда най-общо така:

initModel ();

while (!eof(input)){

sym=getSymbol(input);

encSym=encode(sym);

writeOut(output,encSym);

updateModel(sym);

}

И съответно декомпресиращата схема:

initModel ();

while(!eof(input)) {

sym=getSymbol(input);

decSym=decode(sym);

writeOut(output,decSym);

updateModel(sym);

}

Синхронизацията между кодирането и декодирането се осъществява посредством използване на общи функции за изграждане и поддържане на модела: initModel () и updateModel(). Какво точно се крие зад тях зависи от конкретния алгоритъм.

**3.2.Модели на Марков**

Разгледаните до момента модели на побуквено кодиране не използват контекста, в който се среща буквата във входното съобщение. Оказва се обаче, че това може да бъде от значение и да подобри нивото на компресия.

Всички разгледани до момента статистически алгоритми работеха с таблица на честотите на срещане на отделните букви (най-често символи), без да се интересуват от контекста. В случай, че на 256 различни символа, беше необходима единствена таблица на честотите, съдържаща по един брояч за всеки символ. По същество това представлява модел от нулев ред. В случай че решим да отчитаме предходния символ, получаваме модел на Марков от първи ред. За реализирането му, в случай на 256 различни символа, са необходими 256 различни таблици, всяка от които съдържа 256 различни елемента. Т.е. имаме нужда от 65536 единици (напр. байтове или думи) памет. i-тата таблица съдържа условните вероятности за срещане на всеки от символите, при условие, че непосредствено предхождащият го символ е бил с АSCH код i. Преминаването към модел на Марков от по-висок ред е свързано с редица проблеми. Така например, ако решим на преминем към модел от ред 2, т.е. да отчитаме каква е била предходната двойка символи, ще са необходими 16 777 216 единици оперативна памет или 65 536 таблици с по 256 реда. Вижда се, че когато редът на модела расте линейно, необходимата памет расте експоненциално. Това силно ограничава възможностите за прилагане на модели от по-висок ред. Още по-лошо: по-голямата част от клетките в таблицата ще бъдат празни, тъй като тази последователност от букви никога не се е срещала.

Използването на модели от по-висок ред, например трети, е свързано с поддържането на 16 777 216 различни честотни таблици, всяка с по 256 елемента. На пръв поглед реализирането на подобен метод може да изглежда нереалистично: заделянето на толкова памет е чисто прахосничество. Действително далеч не всички възможни последователности от четворки последователни символи ще се срещнат изобщо в текста, т.е. огромна част от елементите на таблиците и дори цели таблици ще бъдат празни. Така например, при компресиране на текст на български език е излишно да предвиждаме място за последователности от символи от вида “ънбх”, “ййьз” или “ъъбб”, просто защото такива едва ли ще се срещнат някога.

Макар това да не се вижда на пръв поглед, при адаптивно компресиране с използване на модели от трети ред, например, се налага поддържане на съответните таблици за всички модели от по-нисък ред: нулев, първи и втори. Това е така, тъй като не винаги можем да кодираме новопостъпилия символ в текущия контекст. Компресирането започва с празна таблица, като всеки символ се добавя в нея, едва когато възникне нужда.Когато дадена четворка символи постъпи за пръв път, се извършва установяване на 1 в съответният брояч в таблицата. На изхода се подава специален escape код, след което се прави опит символът да бъде закодиран в таблица, съответстваща на контекст от ред, по-нисък с 1. Ако кодирането отново се окаже невъзможно, се генерира нов escape и се прави опит за кодиране в модел от първи ред. Ако кодирането отново се окаже невъзможно дори в нулев ред, то се преминава към специален контекст от (-1)-ви ред, чиито броячи първоначално са били инициализирани с единици, т.е. там със сигурност ще може да се кодира символа. Актуализирането се извършва само в тези таблици, които са участвали в кодирането на текущия символ, т.е. чиито броячи са били променяни.

Обикновено честотата на escape се приема за 1 и не се променя в процаса на кодиране. Това означава, че на escape се съпоставя най-дългият достъпен код. Подобен подход не е много ефективен, тъй като символът escape се среща доста често, особено при модели от по-висок ред. Понякога на escape се съпоставя честотата на срещане, равна на броя на символите от текущата таблица без значение на честотата им. Това работи добре, тъй като с увеличаване на броя на попълнените символи в таблицата вероятността за срещане на escape намалява. В случай на пълно запълване вероятността става 0. От друга страна е добре да се отчита и вероятността на “познаване” на символ от таблицата, т.е. с каква вероятност постъпващите символи се откриват в таблицата. С нарастването на тази вероятност за срещане на escape намалява. Тимоти Стенли предлага формула за пресмятане честотата на escape , като въвежда мярка на случайността на таблицата: максималната честота, разделена на средната. Колкото по-малко е частното, толкова по-малко случайна е таблицата. Честотата на escape се пресмята с помощта на формулата:

*честота =(256 -#срещнати символи)\* #срещнати \_символи/(256\* най\_висока\_честота) ако (честотата <1) то честотата = 1*

Възможно е компресирането да става с постепенно увеличаване на реда на модела. Започва се с модел от нулев ред, като към модел от първи ред се преминава, едва след натрупване на достатъчно статистики. Аналогично към модел от втори ред се преминава, едва след някаква степен на заоълване на таблиците от втори ред и т.н. това намалява значително броя на генерираните escape символи.

**4.Речниково кодиране**

Речниковото кодиране се основава на изграждането и поддържането на таблица, съдържаща някои от думите от входното съобщение. Алгоритъмът се изпълнява на две стъпки и предполага кодиране на крайни входни последователности. На първата стъпка входното съобщение се изчита и анализира, за да се определят кодовите думи. На втората стъпка то се преглежда отново, като думите от речника се заменят с индекса си в него. При подходящ избор те ще се окажат по-къси от индекса си, т.е. ще се постигне компресиране.За съжаление, не съществува алгоритъм, който да ни дава оптималния речник. Различните методи решават по различен начин този проблем, като никой от тях не гарантира оптималност. Ясни са две неща:

1. в речника трябва да влязат най-често срещаните думи
2. при по-дълги думи следва да се очаква по-добра степен на компресия.

Обикновено размерът на речника е 2ⁿ , тъй като тогава индексът има размер n бита. Някои варианти на алгоритъма изискват фиксиран размер на думите от речника. Променливият размер по принцип осигурява по-висока ефективност. В случай, че някоя дума липсва в речника, тя ще постъпи на изхода без изменения. Получава се смесване на кодирани и некодирани думи, което явно може да доведе до проблеми при декодирането. Един възможен подход за решаване на проблема е, преди кода на думата да се поставя специален разделител, указващ дали следващите битове следва да се възприемат като дума или като индекс в речника. Използването на отделни такива разделители може да се окаже неефективно, поради което те често се групират. Най-често за разделителите се отделя цял байт. Ако всички негови битове са 0, след него следва индекс от речника. В противен случай следва дума с дължина, указана от числото, записано в байта.

**4.1.Ентропия**

През 1940.г Клод Шенън въвежда понятието количество информация. Понятието информация, съдържаща се в едно съобщение, може да се разглежда както синтактично, така и семантично. Очевидно, за целите на компресирането смисълът на съобщението е напълно ненужен. Можем да компресираме произволно съобщение, без да ни е необходимо да знаек какво точно означава, т.е. при кодирането ние гледаме на него строго синтактично и само като на последователност от букви. За да се избегнат подобни недоразумения, свързани със смесване на двете интерпретации на понятието информация, Шенън въвежда термина ентропия. Опитвайки се по някакъв начин да го дефинира, той забелязва, че то зависи изцяло от вероятностите за поява на отделните букви във входното съобщение. Поставяйки и някои допълнителни естествени изисквания, Шенън достига до идеята за дефиниция на ентропията като функция Н(p1,p2,………..pn) на вероятностите p1, p2……pn за поява на всяка от буквите, със следните свойства:

1. неотрицателност: Н≥0;
2. непрекъснатост: Малките изменения на вероятностите да водят до малки изменения на ентрпията;
3. симетричност: Н (
4. кохерентност:

**4.2. Компресиране в реално време**

Комресирането и декомпресирането на цял диск или части от него в реално време има безпорни предимства. Главното е увеличаването размера на свободното дисково пространство, като обикновено се постига степен на компресия между 1,7:1 и 2,1:1 в зависимост от типа данни, съхранявани на диска. Методът се оказва изключително ефективен при големи дискове, съдържащи множество малки файлове. Повечето файлови системи изискват всеки файл да заема цяло число сектори, като всеки сектор принадлежи на най-много един файл. При големи дискове размерът на секторите нараства, при което някои достатъчно малки файлове могат да заемат незначителна част от сектора, например 5-10%, което води до големи разхищения. Подобен проблем се среща често и при по-големи файлове, при които обикновено последният сектор е почти незапълнен. Програми като Stacker и DoubleSpase решават проблема, като съхраняват всички файлове в единствен компресиран файл.

Реализирането на такъв метод е сложен проблем по няколко причини. Първо, той рядко започва да компресира добре от самото начало, поради използването на адаптивни схеми, изискващи натрупването на съответни статистики. Алтернативната възможност за използване на фиксирани статични речници е по-лоша в перспектива. Основният проблем обаче е, че за разлика от модемите, лентовите устройства и последователните файлове твърдите дискове са устройства с произволен достъп. Това означава, че трябва да се гарантира на програмите начин за пряк достъп до произволна позиция на всеки файл, което поражда редица проблеми. Декомпресирането на целия файл, последвано от даване на достъп до исканата позиция, е крайно неефективен подход, особено при големи файлове. Повечето компресиращи програми решават проблема, компресирайки на ниво сектор, като при всеки нов сектор моделът се инициализира наново. Тъй като данните, така или иначе физически се четат и записват на сектори, такъв подход се оказва изключително ефективен.

Компресирането на данни в реално време има и недостатъци. Един от основните се отнася до сигурността на данните в случай на грешки, породени от прекъсване на тока, вирус и др. Вторият проблем е свързан с евентуално забавяне на дисковите операции, тъй като всяка такава операция е свързана с процес на компресиране или декомпресиране. На практика се оказва, че това е несъстоятелно, тъй като в повечето случаи вместо забавяне, се наблюдава ускоряване. Причината е следната: компресирането означава, че реално на твърдия диск се записват и съответно прочитат по-малко данни. И тъй като времето, необходимо за компресирането/декомпресирането, е по-малко от спестеното в резултат на намаленото време за четене/писане от твърдия диск, в крайна сметка се получава ускорение.

**5.Компресиране със загуба**

**5.1. Изразяване и квантифициране**

Най прост метод за компресиране със загуба е прякото изрязване на някои от битовете и/или квантифициране. Да предположим, че искаме да компресираме графично изображение, цветът на всяка точка от което се определя от 24-бита. Ако “изрежем” 3 бита по подходящ начин, качеството на изображението почти няма да се промени. Същевременно, ще сме успели да намалим размера на файла с изображението с 1/8. Такива техники са приложими не само при графични изображения, но и при звук, видео и др. Изобщо, навсякъде където се работи с неточни данни от вълнов характер, произтичащи от заобикалящата ни действителност.