دانشکدهٔ مهندسی کامپیوتر

ارائه و پیاده سازی یک تبدیل برای فشرده‌سازی مجدد فایل‌ها

نگارش:

امیر اعوانی

شهاب تشرفی

استاد راهنما:

دکتر محسن شریفی

پایان‌نامهٔ کارشناسی

در رشتهٔ

مهندسی کامپیوتر- نرم‌افزار

تابستان ۱۳۸۳

فهرست مطالب

عنوان **شماره صفحه**

فصل اول: مقدمه ۱

۱-۱- روشهای فشرده سازی چگونه عمل می نمایند؟ ۳

۱-۲- انواع مختلف فشرده سازی ۶

۱-۳- ساختار پایان نامه ۹

فصل دوم: کاربردهای فشرده سازی و عوامل مهم در بررسی روش های فشرده سازی ۱۱

۲-۱- سرعت فشرده سازی ۱۳

۲-۲- سرعت استخراج داده ها ۱۵

۲-۳- سادگی روش فشرده سازی ۱۷

۲-۴- قابلیت دستیابی تصادفی به داده ها ۱۸

۲-۵- قابلیت بازیابی دقیق داده ها ۲۰

۲-۶- قابلیت انتقال در محیطهای دارای نویز ۲۱

۲-۷- قابلیت به کارگیری در کاربردهای بلادرنگ ۲۳

۲-۸- قابلیت کار در حجم بالا یا پایین داده ها ۲۶

۲-۹- قابلیت ویرایش داده های فشرده شده ۲۶

۲-۱۰- آیا نیاز به داده های مرجعی برای استخراج داده های اولیه داریم؟ ۲۷

فصل سوم: فشرده سازی مبتنی بر کد کردن سمبولها ۲۹

۳-۱- الگوریتم Shannon-Fano ۲۹

۳-۱-۱- الگوریتم ساختن درخت کدینگ Shannon-Fano ۳۱

۳-۲- الگوریتم Huffman ۳۲

۳-۳- الگوریتم هافمن تطبیق پذیر ۳۶

۳-۳-۱- به روز رسانی درخت هافمن ۳۹

۳-۳-۲- یک بهبود دیگر در الگوریتم تطبیق پذیر هافمن ۴۴

عنوان **شماره صفحه**

فصل چهارم: روشهای فشرده سازی با استفاده از درخت ها و Trie ها ... ۴۷
۴-۱- درخت ها و Tries ... ۴۷
۴-۱-۱- خصوصیات ویژه Trie ها برای نگه داری داده ها ... ۴۹
۴-۲- درخت های پسوندی ... ۵۱
۴-۳- فشرده سازی با استفاده از روش شاخص گذاری پنجره های لغزنده ... ۵۲
۴-۳-۱ ایجاد درخت پسوندی ... ۵۴
۴-۳-۲- لغزاندن پنجره ها ... ۵۸
۴-۳-۲-۱- حفظ خاصیت مسیر فشرده بودن درخت ... ۶۱
۴-۳-۲-۱- حفظ خاصیت مسیر فشرده بودن درخت ... ۶۱

فصل پنجم: روش های فشرده سازی با استفاده از دیکشنری ها ... ۶۴
۵-۱- ایده اصلی روش های فشرده سازی با استفاده از دیکشنری ... ۶۵
۵-۱-۱ یک مثال ... ۶۶
۵-۲- دیکشنری ایستا در مقابل دیکشنری تطبیق پذیر ... ۶۷.
۵-۳- معرفی چند روش ... ۷۲
۵-۳-۱- روش LZ77 ... ۷۲
۵-۳-۲- مشکل کد کننده در LZ77 ... ۷۷
۵-۳-۳- روش LZ78 ... ۷۸
۵-۳-۳-۱- جزئیات الگوریتم LZ78 ... ۸۱
۵-۳-۳-۲- چگونگی پیاده سازی LZ78 ... ۸۵

فصل ششم: پیشنهاد یک روش برای فشرده سازی دوباره فایل های فشرده ... ۹۰
۶-۱- تعریف تصادفی بودن ... ۹۰
۶- ۲- تئوری Shannon ... ۹۳
۶-۳- نمایش گرافیکی فرآیندهای مارکف ... ۹۷
۶-۴- آشفتگی منبع اطلاعات ... ۹۹

عنوان **شماره صفحه**

فصل هفتم: مدل عمومی فشرده سازی ... ۱۰۷
۷-۱- توصیف سیستم ... ۱۰۷
۷-۲- سیستم کدگشایی ... ۱۱۱
۷-۳- آنالیز کارایی روش عمومی ... ۱۱۲

فصل هشتم: پیاده سازی نمونه از روش عمومی ارائه شده ... ۱۱۵
۸-۱- توابع پیش بینی کننده ... ۱۱۵
۸-۱-۱- گرامرهای L-System ... ۱۱۵
۸-۱-۲- نحوه استفاده ما از گرامرهای L-System ... ۱۱۸
۸-۱-۳- تولید درخت اشتقاق ... ۱۲۱
۸-۱-۴- نگاشت از ورودی به نودهای درخت ... ۱۲۲
۸-۲- تابع محاسبه گر تفاوت ... ۱۲۳
۸-۳- نگاهی بر بازگشت پذیری روش ... ۱۲۵
۸-۴- مقدار تصادفی بودن روش ... ۱۲۶

فصل نهم: نتایج آماری و تحلیلی ... ۱۲۸
۹-۱- معیارها ... ۱۲۸
۹-۲- نتایج ابتدایی ... ۱۳۱
۹-۳- انتخاب گرامرها بر اساس بهترین P% ... ۱۳۲
۹-۴- انتخاب گرامرها بر اساس بهترین U% ... ۱۳۲
۹-۵- انتخاب گرامرها بر اساس بهترین AEI ... ۱۳۳
۹-۶- نتایج ریزتر برای بهترین پیکربندی ... ۱۳۳

فصل دهم: نتیجه گیری و کارهای آتی ... ۱۳۵
۱۰-۱- نتیجه گیری ... ۱۳۵
۱۰-۲- کارهای آتی ... ۱۳۶

عنوان	شماره صفحه

فهرست منابع: ۱۳۷

چکیده

امروزه، روش‌های فشرده‌سازی کاربردهای بسیاری در جنبه‌های مختلف کامپیوتر و ارتباطات پیدا کرده‌اند. هر سیستم اطلاعاتی امروزی، به گونه‌ای (مستقیم و یا غیرمستقیم) از مزایا و منافع روش‌های فشرده‌سازی، بهره می‌برد. پایان‌نامه‌ای که در مقابل شماست، تلاشی است برای ارائه روشی جدید برای فشرده‌سازی بدون نقصان[1] داده‌ها بر روی فایل‌هایی که یک بار فشرده شده‌اند و رفتاری شبه‌تصادفی دارند. در روش ارائه‌شده،از گرامرهای L-System استفاده کرده ایم که قبل از این در ساخت فرکتال‌ها استفاده می‌شدند. از آن‌جا که دامنه تعریف گرامرهای L-System بسیار وسیع است نیاز به تعریف معیارهایی برای انتخاب یک گرامر مناسب از بین گرامر های مختلف احساس می‌شد. در این پایان‌نامه سه معیار اساسی برای انتخاب و ارزیابی کارآیی گرامرها پیشنهاد شده است. در نهایت با استفاده از معیارهای ارائه‌شده، در بهترین حالت به طور متوسط کاهش حجم دست یافته‌شده، ۲ درصد می‌باشد که در مقایسه با الگوریتم ریاضی بدون این بهینه‌سازی و الگوریتم ریاضی با این بهینه‌سازی به دست آمده است.

کارهای مرتبط با این پروژه که می توان آن‌ها را ادامه این پروژه دانست عبارتند از:

- بررسی جهت تعیین معیارهای ارزیابی دقیق‌تر
- بررسی جهت پیدا کردن روش‌های دیگری برای پیش‌بینی ورودی

[1] Lossless

فصل اول

مقدمه

فشرده سازی داده ها یک نوع عمل کدینگ است که در آن، داده های ورودی به طریقی کد می شوند که فضای کمتری را اشغال نموده و نیز، بتوانند دوباره در هر زمان دلخواه بازیابی شده و داده اصلی را برای ما بازگردانند[16].

همان طور که از معنای این عبارت بر می آید، هدف اصلی در فشرده سازی، کوچک کردن حجم داده‌ها می باشد. شاید سال‌ها پیش، همزمان با ایجاد مبحث فشرده سازی، این هدف بسیار بزرگ می نمود. چرا که منابع ذخیره داده‌ها حجم محدودی داشتند و بسیار هم گران‌قیمت بودند. اما اکنون به نظر می رسد که آن زمان گذشته است و دیگر نیازی به این همه تلاش برای بایتی یا کیلو بایتی کمتر فایده ای ندارد. حال آنکه وضعیت کاملا متفاوت است و امروزه، نیاز به فشرده سازی نه تنها کمتر نشده است، بلکه به صورت روز افزونی هم بیشتر می شود و شاید تنها تفاوتی که وجود دارد، در نحوه اعمال روش‌های فشرده سازی باشد. در گذشته، اکثرا، افراد به صورت دستی اقدام به فشرده نمودن داده های خود می کردند. ولی امروز، همه این کارها در پشت صحنه انجام می شود و

چون افراد از انجام آنها اطلاع کمتری دارند، بعضا این فکر به وجود می آید که شاید نیاز به این روش‌ها از بین رفته است و این روش‌ها دیگر انجام نمی شوند. اجازه دهید این نیاز روز افزون را با چند مثال توجیه نماییم:

- هرکدام از ما، به طور قطع، در دیسک سخت کامپیوتر خود، یک یا چند فیلم را ذخیره نموده ایم. آیا هیچ می دانید که اگر این فیلم‌ها فشرده نشده بودند، شاید حتی داشتن یکی از آنها بر روی دیسک سخت شما غیر ممکن بود. شاید به نظر غیر منطقی برسد، لیکن با یک مقایسه ساده بین قالب های فایل avi.(حالت قدیمی) و mpg. می توان به راحتی این موضوع را ثابت کرد. ۲ دقیقه فیلم در قالب‌های avi. در حدود ۲۰۰ مگا بایت فضا نیاز دارد. با یک ضرب ساده، می-توان دریافت که یک فیلم ۱۲۰ دقیقه ای، به ۱۲ گیگا بایت فضا نیاز خواهد داشت و جالب اینجاست که قالب avi.، از نوعی فشرده سازی ساده نیز بهره می برد و با این وجود، فیلم ۲ ساعته به ۱۲ گیگابایت فضا نیاز دارد.

- آیا باور می کنید که اگر فشرده سازی وجود نداشت، هیچ یک از تحقیقات فضایی کنونی با ماهواره‌های بدون سرنشین ممکن نبود؟ دلیل این گفته را در بخش کاربردهای روش‌های فشرده سازی توضیح خواهیم داد.

- آیا می دانید که بدون وجود فشرده سازی، اکنون با انفجار اطلاعات مواجه بودیم و هیچ یک از سرویس‌های بزرگ اینترنتی همچون Google و Yahoo نمی توانستند، داده‌های بیش از حد زیاد خود را کنترل کنند. دلیل این موضوع، در وابسته بودن بسیار زیاد بانک‌های اطلاعاتی به روش‌های فشرده سازی نهفته است. این بانک‌ها نه تنها، برای کم کردن فضای مورد نیاز خود از

این روش‌ها استفاده می کنند، بلکه، بدون این روش‌ها، بازیابی این اطلاعات نیز اگر غیر ممکن نباشد، مطمئنا بسیار سخت خواهد بود. این موضوع نیز به تفصیل در بخش کاربردهای روشهای فشرده سازی بررسی خواهد شد.

- آیا می دانید که اگر فشرده سازی موجود نبود، ارتباط شما با اینترنت هم نمی توانست با این سرعت انجام گیرد؟ چرا که همه مودم‌ها از پروتکل‌های فشرده سازی ساده ای که به صورت سخت افزاری در داخل آن‌ها تعبیه شده است استفاده می نمایند و به این ترتیب، سرعت انتقال داده‌ها را تا حداقل دو برابر افزایش می دهند.

در ادامه به توضیح نحوه کلی عملکرد روش‌های فشرده سازی و دسته بندی های موجود برای این روشها خواهیم پرداخت. سپس ذکر مفصلی از عواملی که باید در هنگام انتخاب روش فشرده سازی مناسب در نظر بگیرید، خواهیم داشت.

۱-۱- روش‌های فشرده سازی چگونه عمل می نمایند؟

شاید مهمترین سوالی که درباره فشرده سازی، مطرح است، چگونگی کارکرد این روش‌ها باشد. آیا واقعا تمام داده‌ها را می توان فشرده نمود؟ اگر اینطور است، چرا از داده‌های غیر فشرده استفاده می نماییم؟ روش‌های فشرده سازی چگونه بدون از دست دادن هیچ اطلاعاتی، داده‌ها را فشرده می نمایند؟

حقیقت این است که تعداد غیر قابل تصوری از داده‌هایی که می توانند موجود باشند، با هیچ یک از روش‌های موجود فشرده سازی، قابل فشرده نمودن نیستند. ولی نکته اصلی در اینجاست که این داده‌ها اصلا وجود خارجی ندارند. تعداد داده‌هایی که انسانها در حجم‌های بالا استفاده می کنند، بسیار بسیار بسیار کمتر از تعداد واقعی داده‌هایی است که در همان حجم می توانند موجود باشند.

اجازه دهید با یک مثال عددی، این موضوع را حل کنیم. فرض کنید همه انسانهای کره زمین از کامپیوتر استفاده می کنند و هر کدام از آنها، یک میلیارد فایل با حجم بالای ۱ کیلو بایت دارد که منحصر به خود اوست و هیچ کس دیگر فایل مشابه آن را ندارد. با یک ضرب ساده می توان دریافت که تعداد کل فایل‌ها در این سیستم که سیستمی کاملا غیر واقعی و بیش از حد بزرگ شده است، ۶ میلیارد میلیارد فایل است. این عدد از عدد ۲ به توان ۶۸ کوچکتر است. حال آنکه می دانیم تعداد کل فایل‌های با حجم فقط یک کیلو بایت، ۲ به توان ۸۰۹۷ منهای یک است. یعنی بیش از ۲ به توان ۸۰۲۸ برابر تعداد حداکثر فایل‌های موجود بر روی کره زمین. بزرگی این عدد را تنها با محاسبه آن می توان دریافت. حال اگر تعداد واقعی فایل‌های موجود بر روی کره زمین و تعداد واقعی حالات را در نظر بگیریم، درخواهیم یافت که بخشی از فایل‌ها که ما در حال استفاده از آن هستیم، بخش غیر قابل تصور کوچکی از تمام حالات ممکن است.

در حقیقت اگر روشی موجود بود که بتواند به اطلاعات مورد استفاده ما، کدهای پشت سر هم تخصیص دهد، تعداد این شماره‌ها آنقدر کم بود که حتی در انتهای عمر بشریت هم به بیش از ۱۲۸ بیت برای هر فایل نیاز نداشتیم.

با دیدی که اکنون از فشرده‌سازی پیدا کردیم، این سوال مطرح می شود که چرا طـول فایـل-هایی که اکنون استفاده می شود، اینقدر زیاد است و طول‌های در حد مگابایت در حال حاضر عـادی ترین مقادیر ممکن هستند؟ و فایل‌های با حجم کمتر از مگابایت، کوچک ارزیابی می شوند؟ جـواب این سوال با نگاهی گذرا به متن فایل‌هایی که ما استفاده می کنیم، داده خواهد شد. تقریبا تمام فایـل-های متنی موجود، از تعداد بسیار محدودی از لعات و عبارات تکراری تشکیل شده است که بـا قـرار گرفتن در پشت سر یکدیگر، عبارات و مفهومهای جدیدی را ساخته اند.

کاری که روش‌های فشرده سازی انجام می دهند، پیدا نمودن و حذف ایـن تکرارهـای زایـد است. این روش‌ها با پیدا نمودن لغات و عباراتی که بیش از دیگـران تکـرار مـی شـوند و جـایگزین نمودن آنها با علائمی مناسب بـا طـول کمتـر، داده‌هـا را فشـرده تـر از آنچـه کـه بـوده اسـت، مـی نمایند[16].

به عنوان مثال، روش Huffman را در نظر بگیرید. این روش در ابتدا با ایجاد آماری اولیـه از فایل ورودی به روش فشرده‌سازی، درختی به نام درخت Huffman ایجاد می نماید. کاربرد این درخت در تخصیص کد به کاراکترهای فایل ورودی و خصوصیت آن در تخصیص کـد کوتـاهتر بـه کاراکتر با تکرار بیشتر، به صورت بهینه است. سپس در مروری دیگر بر روی فایل، هر کاراکتر با کـد اختصاص داده شده به آن، جایگزین می شود. در انتها نیز با ذخیره نمودن کد هر کـاراکتر، اسـتخراج داده‌ها به داده های ابتدایی، ممکن خواهد شد[17].

درخت Huffman تضمین می نماید که کدهای تخصیص داده شده، کاراترین کدها باشـند و لذا، این روش حرف آخر را در فشرده سازی‌هـای Character-Based مـی زنـد. ولـی انـواع

بسیار متفاوتی از فشرده سازی‌های دیگر مثل فشرده سازی‌های Dictionary-Based وجود دارند که نتایجی بسیار بهتر از روش Huffman را تولید می کنند.

خیلی از ما در هنگام کار با نرم افزارهای فشرده ساز، به مواردی برخورده ایم، که نرم افـزار قادر به فشرده نمودن بعضی از فایل‌ها نیست و حاصل کار نرم افزار، به جای فشرده شدن، چند کیلـو بایت هم بیشتر از فایل اصلی جا می گیرد. شاید این سوال مطرح شود که آیا می تـوان روش فشـرده سازی ای را ابداع نمود که هیچ داده ای را بزرگ ننماید و در ضمن، بعضی از داده ها را نیز کوچـک نماید؟ جواب این سوال منفی است. به صورت ریاضی و با کمی دردسر، می توان ثابت نمود که اگـر روشی، داده ای را به اندازه a بیت کوچک نماید، در بهترین حالت، باید تعدادی دیگـر از داده هـا را مجموعا به مقدار a بیت بزرگ نماید. این قضیه با استفاده از تعداد حالات ممکن ثابت می شود.

بحث کار بر روی روش‌های جدید فشـرده سـازی بـه دلیـل کاربردهـای بسـیار آن در علـم کامپیوتر و ارتباطات، همچنان باز است و هر ساله یا روش‌های جدید تولید می شوند و یـا روشـهای قدیمی بهبود می یابند. کنفرانس سـالانه انجمـن IEEE، بـه نـام DCC(Data Compression Conference) در ماه فوریه هر سال برگزار می گردد و با ارائه روش‌هـای جدیـد، ایـن شـاخه از علم کامپیوتر را گسترش می دهد[18].

۱-۲- انواع مختلف فشرده سازی

در بخش ۱-۱ به قضیه ای اشاره کردیم که بر اساس آن، هیچ روش فشرده سازی‌ای را نمی-توان داشت که همه داده‌ها را کوچکتر از آنچه که بوده‌اند نماید و یا حداقل آنهـا را بـزرگ نکنـد. در

اینجا می خواهیم بگوییم که این قضیه برای همه انواع فشرده سازی‌های موجود درست نیست. این قضیه فقط وقتی مطرح است که روش فشرده سازی ما، به اصطلاح، Loss-less باشد.

مقدمه بالا، باعث شد تا به دو دسته کاملا متفاوت از روش‌های فشرده سازی اشاره نماییم. روش‌های Lossy و روش‌های Loss-less.

روش‌های فشرده‌سازی Lossy، روش‌هایی هستند که داده‌های ورودی آنها، داده‌های عیر حساس به تغییرات کوچک هستند. ولی روش‌های Loss-less، فرض می کنند که حتی یک بیت از داده‌های ابتدایی هم نباید از دست رود.

روش‌های Lossy، برای کاربردهای خاص مطرح می شوند و روش‌های Loss-less، گرچه که می‌توانند برای کاربردهای خاص مطرح شوند، ولی در تئوری، می توانند هر فایلی را در ورودی خود بپذیرند و در خروجی نیز همان فایل را تحویل دهند.

دسته بندی روش‌های Lossy، به دلیل ماهیت این روش‌ها، بیشتر بر اساس کاربردشان انجام می پذیرد، و گاهی نیز بر اساس تکنیک‌هایی که در آنها استفاده شده است. مثل روش‌های فشرده سازی تصویر (تقسیم بندی بر اساس کاربرد)، روش‌های فشرده سازی صوت (تقسیم بندی بر اساس کاربرد) و یا روش‌هایی که از DCT (تبدیل کسینوسی گسسته) استفاده می‌کنند (تقسیم بندی بر اساس تکنیک‌های مورد استفاده).

در مقابل روش‌های Lossy، روشهای Loss-less قرار دارند که به دلیل عمومی‌بودن اکثر آنها، معمولا بر اساس تکنیک‌های مورد استفاده در آنها و گاهی بر اساس کاربرد اختصاصی شان دسته بندی می شوند. به عنوان مثال می توان به روشهای Character Based، روش‌های

Dictionary Based و یا روش‌های فشرده سازی متون اشاره نمود که آخری، بر اساس کاربردش دسته‌بندی شده است.

روش‌های Character Based، روش‌هایی هستند که تعداد بیت دلخواهی از داده‌ها را به عنوان یک کاراکتر معرفی کرده و به هر کدام از آنها به صورت جداگانه و مستقل از دیگر کاراکترها، می نگرند. مهمترین روشی که در این بخش مطرح می شود، روش Huffman و انواع روش‌هایی است که بر مبنای آن توسعه داده شده اند[17].

روش‌های Dictionary Based، روش‌هایی هستند که پس از تعریف هر کاراکتر (همانند آنچه در روش Huffman انجام می‌دهیم)، به کاراکترها به عنوان محموعه‌های مستقل از یکدیگر نمی نگرند و آنها را در ارتباط با هم می بینند. به این ترتیب، کلمات (Wordها) را تشکیل می‌دهند و سپس، با تخصیص کد به هر کلمه (Word)، داده‌های ورودی را فشرده می نمایند[17].

روش‌های Dictionary Based به دو دسته تقسیم می شوند:

دسته اول، دسته ای هستند که ابتدا از یک دیکشنری اولیه و ثابت شروع می کنند و با جلو رفتن در فایل ورودی، دیکشنری خود را غنی می‌نمایند. در نتیجه، در این روش‌ها ممکن است که یک داده یکسان در دو جای متفاوت از یک فایل، دارای کدهای متفارت باشد. بهترین نمونه برای روش‌هایی از این قبیل را می توان فشرده سازی LZW دانست[17].

دسته دوم، دسته‌ای هستند که ابتدا دیکشنری خود را بر اساس کل فایل می‌سازند و سپس، بر اساس این دیکشنری، فایل اولیه را فشرده می نمایند. این دسته از روش‌ها، نیاز دارند که دیکشنری

خود را برای انجام عمل معکوس در فایل مقصد ذخیره نمایند (شبیه آنچه که Huffman برای درخت خود انجام می داد). بهترین نمونه از این نوع روش‌ها را می توان Ray یا XRAY دانست.

البته روش‌های معدودی هم وجود دارند که در این دسته بندی ها از روشهای Loss-less جا نمی‌گیرند. مثل روش RLE (Run Length Encoding) که بیشتر بر اساس کاربرد آن در قالب‌های تصویری قدیمی مثل BMP یا PIC دسته بندی می شود.

۱-۳- ساختار پایان نامه

موضوعات زیر، عناوینی هستند که در این پایان‌نامه بر روی آنها بحث شده است:

۱- فصل اول- مقدمه و آشنایی کلی با نحوه عملکرد روش‌های فشرده‌سازی

۲- فصل دوم- معیارهای مختلف تاثیرگذار بر کارآیی روش‌های فشرده‌سازی

۳- فصل سوم- مروری بر روش‌های فشرده‌سازی مبتنی بر کدکردن سمبول‌ها

۴- فصل چهارم- مروری بر روش‌های فشرده‌سازی با استفاده از درخت‌ها و Trieها

۵- فصل پنجم- مروری بر روشهای فشرده‌سازی با استفاده از دیکشنری‌ها

۶- فصل ششم- توضیحات ما بر روی چگونگی فشرده‌سازی دوباره فایل‌های فشرده شده

۷- فصل هفتم- توصیف مدل عمومی‌ای که ما برای کارمان ارائه داده‌ایم

۸- فصل هشتم- خاص نمودن مدل عمومی فصل قبل و توضیحات پیاده‌سازی انجام شده ما

۹- فصل نهم- نتایج ریز آماری حاصل و نتیجه‌گیری‌های نهایی‌ما برای پیکربندی سیستم

که فصول دوم، سوم، چهارم و پنجم بیشتر به ارائه سیر تحـولات و ایـده هـا در امـر فشـرده سازی می پردازد. در فصول ششم به توضیح امکان فشرده سازی مجدد داده ها پرداخته ایم. در هفتم، هشتم بیشتر بحث در مورد پیاده سازی است. و در فصول نهم و دهم به نتیجه گیری و کارهای آینده پرداخته ایم.

فصل دوم

کاربردهای فشرده‌سازی و

عوامل مهم در بررسی روش‌های فشرده‌سازی

همان‌طور که در اولین نگاه و از نام آن بر می آید، مهمترین عامل در بررسی روش‌های فشرده‌سازی، مقدار فشرده‌نمودن داده‌ها است که معمولا بر حسب bpc (Bit Per Character) محاسبه می‌شود. به عنوان مثال، به جدول زیر که برای مقایسه بین روش‌های معروف فشرده‌سازی آمده است، توجه کنید:

نام روش	WEB-LRG	WEB-SML	WSJ	AP
XRAY	۲,۰۶	۱,۵۷	۲,۰۸	۲,۱۵
COMPRESS	۳,۱۸	۳,۴۱	۳,۴۴	۲,۵۰
GZIP1	۲,۵۵	۲,۷۶	۳,۴۹	۳,۵۱
GZIP9	۲,۲۰	۲,۳۸	۲,۹۷	۲,۹۹
BZIP2	۱,۴۹	۱,۳۵	۲,۱۲	۲,۱۹
HUFFWORD	---	۳,۲۲	۲,۲۳	۲,۳۰

همانگونه که در بالا مشاهده می شود، برای انجام تستها، از چهار نمونه داده استفاده شده است که نام هر یک در اولین سطر از ستون مربوطه آمده است. ضمنا، روشی که با نام XRAY معرفی گشته است، روش معروفی نیست و فقط به دلیل حفظ امانت و از آنجا که در مقاله مرجع موجود بوده، در اینجا ذکر گردیده است[19].

در بالا گفتیم که مقدار فشرده‌سازی مهمترین عامل در بررسی روش‌های فشرده‌سازی و مقایسه آنها با یکدیگر است، ولی آیا این عامل، تنها عامل است یا عوامل دیگری هم در این زمینه دخیلند؟ در ادامه خواهیم دید که بسیاری از عوامل دیگر هم در مقایسه روش‌های فشرده سازی ذخیلند. بعضی از این عوامل به طور غیر مستقیم به دیگر عامل‌ها مربوط می شوند ولی به دلیل

اهمیت بسیار زیادشان به صورت مستقل باید بررسی گردند، و بعضی نیـز بـه کلـی مسـتقل از دیگـر عواملند که در هر مورد به طور خاص شرح خواهیم داد.

۲-۱- سرعت فشرده سازی

اهمیت این عامل را وقتی می توان به درستی درک نمـود، کـه درک درسـتی از کاربردهـای فشرده‌سازی در اختیار داشته باشیم:

یکی از کاربردهای مهم فشرده سازی در انتقال داده‌ها می باشد. با فشرده کـردن داده‌هـا، نـه تنها پهنای باند کمتری برای انتقال داده‌ها نیاز داریم، بلکه به دلیل کمتر بودن داده‌هـای منتقـل شـده، امکان بروز خطا در داده‌ها نیز کمتر شده و تلاش کمتری هم برای کشف و تصحیح خطاها مورد نیاز است.

از سوی دیگر، همانطور که می دانید، بسیاری از سیسـتم‌هـای فایلینـگ هماننـد NTFS، از فشرده سازی داده‌ها در حین ذخیره آنها را پشتیبانی می کنند.

بسیاری از DBMS ها داده‌های خود را قبل از ذخیره در فایل‌ها، فشرده می نمایند.

بسیاری از سخت افزارهای دریافت داده‌های مولتی مدیا از دنیای خارج، خروجـی خـود را به صورت فشرده شده و در قالبهایی مثل MP3 برای صوت، و یا MPEG برای تصـویر، در اختیـار می گذارند.

مواردی که در بالا ذکر شد و بسیاری موارد دیگر که به فشرده‌سازی بلادرنگ و یا بسیار سریع داده‌ها نیازمندند، حاضرند که از کیفیت فشرده سازی بگذرند و سرعت را بیشتر دریابند. معمولا در این روش‌ها گونه‌ای از توازن هملکرد بین سرعت و کیفیت فشرده‌سازی برقرار می شود. به این معنی که تا جایی که محدودیت‌های زمانی اجازه می دهند، فشرده‌سازی را قویتر می نماییم. ولی نه بیشتر از آنچه که محدودیت‌ها اجازه می دهند.

حال با ارائه چند مثال عددی، به مقایسه‌ای بین سرعت روش‌های فشرده سازی مختلف خواهیم پرداخت. این داده‌ها بر حسب مگا بایت بر دقیقه محاسبه شده‌اند و باز هم فقط به دلیل حفظ امانت در مطلب، روش XRAY را بیان کرده ایم[20].

نام روش	WEB-LRG	WEB-SML	WSJ	AP
XRAY	۷۳	۳۲	۵۰	۴۱
COMPRESS	۹۵	۹۶	۹۳	۹۴
GZIP1	۱۴۶	۱۴۹	۱۲۴	۱۲۷
GZIP9	۸۵	۹۰	۶۰	۷۰
BZIP2	۲۷	۲۸	۳۰	۲۹
HUFFWORD	---	۶۳	۶۳	۶۸

۲-۲- سرعت استخراج داده ها

شاید در اولین نگاه این سوال مطرح شود که چرا این مبحث و مبحث بالا در یک سرفصل و با عنوان مثلا سرعت فشرده‌سازی مطرح نشدند. لکن با توضیحاتی که در زیر خواهیم داد، می‌توان دریافت که این دو مبحث باید کاملا جدا از هم بررسی شوند.

مثال مقایسه فیلم‌هایی که در مقدمه آورده شد را به خاطر آورید. هیچ کس نمی تواند انکار کند که بخش بزرگی از کاربرد روش‌های فشرده سازی، در کاربردهای مولتی مدیا می باشد. همه ما با قالب فایل‌های avi. جدید که از فشرده‌سازی DivX استفاده می‌کنند روبرو شده‌ایم و از کیفیت بسیار بالای فیلم‌های آن و نیز حجم بسیار پایین آن لذت برده‌ایم. آیا هرگز فیلمی را از فشرده سازی MPEG به فشرده‌سازی DivX تبدیل کرده‌اید؟ این کار علاوه بر این که بسیار طول می کشد، حجم بسیار بالایی از CPU را نیز مصرف کرده و می تواند شما را کاملا از کاری که کرده‌اید، پشیمان کند. ولی این موضوع در محبوبیت روش فشرده سازی DivX هیچ تاثیری ندارد. دلیل این موضوع در تفاوت بین زمان فشرده‌سازی و زمان استخراج داده‌ها نهفته است. زمان فشرده سازی داده‌ها در روش DivX بسیار بالاست ولی زمان استخراج داده‌های اصلی، در حد معقولی پایین است و حجم پردازش زیادی را نیز نیاز ندارد.

به طور کلی، زمان استخراج داده‌ها در بسیاری از کاربردها، خیلی مهمتر از زمان فشرده-سازی آنها می باشد. زیرا در بسیاری از کاربردها، داده یک بار فشرده می شود و دفعات بسیار زیادی از آن استفاده می شود.

البته این قضیه عمومی نیست و فقط به مواردی که خصوصیت بالا (یک بار فشرده سازی در مقابل تفداد دفعات زیادی از استخراج) را دارند، محدود می شود و از بین آنها هم به مواردی که نباید بلادرنگ باشند محدود می شوند. به عنوان نمونه، مثال مودم را که در مقدمه اشاره کردیم در نظر بگیرید، در اینجا، هر داده یک بار فشرده می شود و یک بار هم باز می شود، پس سرعت فشرده‌سازی و سرعت استخراج داده‌ها، به یک اندازه مهمند. یا به عنوان نمونه‌ای دیگر، در سیستم فایل NTFS و یا پایگاه‌های داده، گرچه که استخراج داده‌ها، به تعداد دفعات بسیار بیشتری از فشرده سازی آنها انجام می شود، ولی به دلیل نیاز به عملیات بلادرنگ، سرعت هر دو روش فشرده-سازی و استخراج داده‌ها، تقریبا به یک اندازه مهمند.

در اینجا هم بد نیست که با ارائه چند مثال عددی به مقایسه سرعت روش‌های مختلف در استخراج داده‌های فشرده سازی شده خود بپردازیم. این داده‌ها بر حسب مگا بایت بر دقیقه اندازی گیری شده‌اند و باز هم به دلیل حفظ امانت، نتایج روش XRAY را شامل کرده‌ایم[20].

نام روش	WEB-LRG	WEB-SML	WSJ	AP
XRAY	۴۹۳	۶۰۸	۴۲۸	۴۱۳
COMPRESS	۴۷۵	۴۵۷	۴۴۹	۴۴۳
GZIP1	۵۷۲	۵۳۵	۴۳۱	۴۳۰
GZIP9	۶۴۴	۵۹۹	۴۹۹	۴۹۶
BZIP2	۱۵۶	۱۵۵	۱۲۶	۱۲۳
HUFFWORD	---	۲۲۷	۲۴۹	۲۴۹

۲-۳- سادگی روش فشرده سازی

شاید جالب باشد که سادگی را جزء عوامل موثر در مقایسه روشهای فشرده‌سازی به حساب می آورند. ولی با توجه به کاربردهای روش‌های فشرده سازی در سخت افزارها، این عامل بسیار حائز اهمیت می شود. چرا که اگر روشی دارای پیاده‌سازی بسیار مشکلی باشد، نه تنها هزینه طراحی بسیار بالایی را بر شرکت سازنده سخت افزار تحمیل می کند، بلکه باعث بالا رفتن بیش از حد هزینه اشکال‌زدایی سیستم‌ها می گردد و در نهایت باعث بالا رفتن هزینه نهایی سیستم سخت افزاری می گردد که بسیار نامطلوب است.

به عنوان نمونه، مثال مودم‌ها را که در مقدمه بیان شد، در نظر بگیرید. در مودم‌ها، به همین دلایل از نمونه‌هایی از روش Huffman استفاده می شود که بسیار ساده شده اند. این روش‌ها گرچه که بسیاری از کارآیی فشرده سازی Huffman را از دست داده‌اند، ولی هنوز هم می توانند به طور متوسط، داده ها را تا دو برابر فشرده کنند. البته در حال حاضر، سیستم‌های سخت افزاری توانایی بسیار بیشتری دارند، ولی به این دلیل که مودم‌ها سخت افزارهای بسیار قدیمی ای هستند، و در زمان تولید آنها محدودیت‌های بسیار بیشتری در سخت افزارها موجود بوده است، روشهایی که اکنون استفاده می شوند نیز با استانداردهای آن زمان به وجود آمده‌اند. البته در مورد این نمونه خاص، مزیت های دیگری هم برای روش Huffman استفاده شده وجود دارد که در بخش "قابلیت به کارگیری در کاربردهای بلادرنگ" ذکر خواهد شد.

به عنوان نمونه ای دیگر که هم اکنون مطرح است، می توان به روش فشرده‌سازی DivX در مقابل روش MPEG اشاره کرد. روش MPEG در حال حاضر در بسیاری از سخت افزارها

پیاده شده است، ولی روش DivX هنوز توانایی پیاده سازی شدن در سخت افزارهای امروزی را ندارد.

۲-۴- قابلیت دستیابی تصادفی به داده ها

دستیابی تصادفی به داده‌ها، به این معنی است که زمان دستیابی به بایت خاصی از داده‌ها به مکان آن بایت در محموعه داده‌ها بستگی نداشته باشد و یا بستگی آن به مکانش در مجموعه داده‌ها آن قدر کم باشد که بتوان از آن صرف نظر کرد.

به صورت کلی وقتی که بتوان در زمانی محدود به هر بخشی از داده‌های فشرده شده، دستیابی پیدا کرد، اصطلاحا می گوییم که دستیابی تصادفی به داده‌ها میسر است.

این قابلیت در چندین جا کاربرد بسیاری دارد که مهمترین آن سیستم های پایگاه داده است. اگر برای هر بار دستیابی به پایگاه داده ها، نیاز به استخراج کل داده‌های ذخیره شده در سیستم می-بود، فشرده‌سازی نه تنها هیچ مشکلی را رفع نکرده بود، بلکه فقط مشکلی را بر مشکلات موجود در سر راه پایگاه‌های داده افزوده است.

کاربرد بسیار مهم دیگر از روش‌های فشرده سازی با دستیابی تصادفی را می توان در قالب-های داده ای مولتی مدیا جستجو کرد. یکی از ویژگی‌های مهمی که در قالبهای مولتی مدیا باید جستجو نمود، قابلیت Feed Forward(FF) یا Play Back است که بعضی از قالب‌ها از آن پشتیبانی می کنند و بعضی دیگر هم نه. در این خصوصیت، فرد می تواند با حرکت دادن

Position Bar به سمت جلو یا عقب و قرار دادن آن در مکان دلخواه، از یعضی از قسمت‌های فایل صرف نظر نماید. مطمئنا در این موارد، خواندن تمام داده ها تا هنگام رسیدن به داده مورد نظر، بسیار احمقانه خواهد بود[21].

خوشبختانه روشهایی از فشرده سازی موجودند که می توان هر بایت و یا گاهی مجموعه ای از بایتهای داده را بدون دانستن دیگر بایت‌های موجود در سیستم، بازیابی نمود. این روش‌ها وقتی که با روش‌های اندیس گذاری مناسب مخلوط می شوند، به روش‌هایی مناسب برای دستیابی تصادفی به داده‌ها تبدیل می گردند.

در بعضی از کاربردها، داشتن خصوصیت دستیابی تصادفی به طور کامل لازم نیست، بلکه عمل Positioning فقط در میان داده‌هایی که حداقل یکبار در طی Session فعلی خوانده شده اند، مورد نیاز است. در این موارد، اندیس را می توان به صورت دینامیک در هنگام خواندن داده ها ساخت و یا به روز درآورد.

همانطور که در بالا ذکر گردید، تنها بعضی از روش‌ها می توانند با همراه شدن با اندیس‌ها، دستیابی تصادفی را پشتیبانی کنند. این روش‌ها، دارای این ویژگی کلی هستند که کد اختصاص داده شده به هر کاراکتر یا مجموعه ای از کاراکترها در طول عمل فشرده‌سازی یکسانند.

به عنوان مثال در روش Huffman، کد اختصاص داده شده به کاراکتر a در تمام طول فایل رشته ای ثابت از بیت‌ها است ولی در روش LZW، درابتدای کار کد ۹ بیتی ۰۰۱۱۰۰۰۰۱ به کاراکتر a اختصاص داده شده و پس از خواندن چندین کاراکتر، کد ۱۰ بیتی 0001100001 را برای همین کاراکتر خواهیم داشت و این کد باز هم در طول اجرای عمل فشرده سازی تغییر

خواهدکرد. البته این روش هم به دلیل داشتن کد ۲۵۶ برای عمل Clear قابل اندیس گذاری به صورت نسبی است ولی به عنوان مثال روش فشرده سازی ریاضی که توسط شرکت IBM ابداع شده است، به دلیل ساختار داخلی این روش نمی تواند با اندیس گذاری تبدیل به روشی مناسب برای دستیابی تصادفی به داده‌ها شود.

۲-۵- قابلیت بازیابی دقیق داده‌ها

داشتن یا نداشتن قابلیت بازیابی دقیق داده‌ها، روش‌های فشرده‌سازی را به دو دسته کاملا متفاوت به نام‌های Lossy compression methods و Loss-less compression methods، تقسیم می‌کند.

روش‌های فشرده سازی Lossy، قادر به بازیابی داده‌های فشرده شده به صورت دقیق نیستند و برای کابردهایی به کار می روند که داده‌ها با حواس انسان سر و کار دارند و چون حواس انسان به تغییرات کوچک در داده‌ها حساس نیست، نیازی هم برای بازیابی دقیق داده‌های اولیه وجود ندارد. فشرده‌سازی‌های Lossy، باعث کاهش غیر قابل تصوری در حجم داده‌ها می گردند و از این رو در کاربردهای اتوماتیکی هم که به تغییرهای کوچک در داده‌ها حساس نیستند استفاده می گردند. از جمله این کاربردها، می توان به پردازش عکس‌های ماهواره ای برای پیش بینی وضعیت هوا و یا تحقیقات فضایی استفاده نمود.

در مقابل روش‌های فشرده سازی Lossy، روش‌های فشرده سازی Loss-less قرار دارند که حتی یک بیت از داده‌های اصلی را از دست نداده و یا تغییر نمی دهند. این روش‌ها، کاربردهای

خاص خود را دارند. به عنوان مثال، اکثر کاربردهای DBMS ها در ارتباط با داده‌هایی است که نباید در اثر فشرده‌سازی تغییر نمایند. فایل‌های اجرایی کامپیوتری، فایل‌های متن، برنامه‌های کامپیوتری، Email ها و بسیاری دیگر از داده‌هایی که بر روی اینترنت منتقل می شوند، در برابر کوچکترین تغییرات در داده اصلی حساسند و از اینرو، نباید کوچکترین تغییری در آنها داده شود.

از سوی دیگر در اکثر روش‌های فشرده سازی Lossy بعد از اعمال یک یا چند تبدیل و حذف داده‌هایی که حواس انسان به آنها حساس نیست، یکی از روش‌های موجود فشرده‌سازی Loss-less بر روی داده‌های باقیمانده اعمال می شود. به عنوان مثال، در قالب فایل JPEG، ترتیب تقریبی کارها به این صورت است: ابتدا از هر ۴ خانه کنار هم که در مجموع ۱۲ داده RGB دارند، ۶ داده که بیشتر به کار می آیند(۴ داده شدت رنگ Y برای هر یک از خانه‌ها، و ۲ داده Cr و Cb برای همه این ۴ خانه با هم) نگاه داشته می شوند و ۶ داده دیگر دور ریخته می‌شوند. سپس، تبدیل کسینوسی گسسته (DCT) بر روی آنها اعمال می شود و فقط مقادیری که بر اثر این تبدیل، مقدار قابل توجهی با صفر تفاوت دارند، نگاه داشته می شوند و بقیه مقادیر، صفر فرض می شوند و در انتها، داده‌های باقیمانده با یکی از روش‌های Extended-Huffman و یا فشرده‌سازی ریاضی، فشرده می شوند.

۲-۶- قابلیت انتقال در محیط‌های دارای نویز

بسیاری از کاربردهای فشرده سازی به انتقال داده‌ها مرتبط می شود. فشرده سازی از یک سو، باعث کاهش حجم داده‌ها و انتقال ساده تر آن از یک دستگاه به دستگاه دیگر می شود (زیرا در

طول انتقال داده‌های با طول کمتر، احتمال وقوع خطا نیز کمتر شده و کوشش مورد نیاز برای کشف و تصحیح آن نیز کمتر خواهد شد) و از سوی دیگر باعث وابستگی تمام یا بخشی از داده‌ها به یکدیگر می شود. از این رو، با بروز کوچکترین خطا، این احتمال وجود دارد که تمام یا بخش بزرگی از داده‌ها از این خطا تاثیر پذیرند. حال آنکه اگر در داده های غیر فشرده شده، خطایی رخ دهد، فقط آن بایت از داده ها که خطا در آن رخ داده تاثیر خواهد پذیرفت و پس از کشف آن، کافی است که فقط همان بخش تصحیح شود.

روشهای LZW یا فشرده‌سازی ریاضی، جزء روش‌هایی هستند که اگر خطایی در آنها رخ دهد، بخش بزرگی از داده‌ها را مورد تاثیر قرار خواهد داد. ولی در فشرده سازی استاندارد Huffman، با ایجاد خطا در یکی از بایت های داده، احتمال انتشار این خطا در مراحل بعد به صورت نمایی پایین است.

علاوه بر این، روش‌هایی مانند بعضی از توسعه‌های Huffman وجود دارند که تصمین می نمایند که اگر خطایی در یکی از داده‌های منتقل شده رخ دهد، به همین داده محدود شود و به هیچ یک از داده‌های پس از آن انتقال نیابد.

استفاده از روش‌هایی مثل اینگونه از توسعه های Huffman، هزینه‌های خاص خود مانند کاهش بسیار زیاد در کیفیت فشرده‌سازی را در بر دارد. این هزینه‌ها باعث شده است که این توسعه های روش Huffman، به انتقال داده در محیط‌های با نویز بسیار بالا محدود شوند. روشهای عملی ای که برای محیطهای معمولی انتقال مثل CD ها استفاده می شوند، بیشتر در قالب فایل‌ها اعمال می‌شوند. به عنوان مثال قالب فایل‌های MPEG به گونه‌ای طراحی گردیده است که اگر بخش

خاصی از CD خراب گردید یا خراش برداشت، فقط بخش کوچکی از فیلم مربوطه دچار اشکال شده و نتواند خوب پخش گردد. بسیاری از روش‌های فشرده سازی هم کاملا این موضوع را فراموش می کنند و آن را به عهده واسطه‌ای که عمل انتقال را انجام می‌دهد می‌گذارند. به عنوان مثال، پروتکل TCP/IP انتقال صحیح اطلاعات را تضمین می‌نماید و بنابراین نیازی نیست که قالب فشرده‌سازی Zip هم به طور جداگانه خود را نگران این موضوع کند.

گرچه که بسیاری از واسطه‌های انتقال اطلاعات امروزی، وظیفه اطمینان از انتقال صحیح اطلاعات را خود بر عهده دارند، لیکن هنوز زمینه‌های بسیاری وجود دارند که نیازمند در نظر گرفتن این ملاحظات در طراحی روش‌های فشرده‌سازی هستند. یکی از این زمینه‌ها، همانا مطالعات فضایی یدون سرنشین است که در یکی از مثال‌های مقدمه ذکر گردید. در این کاربرد، نه تنها باید داده‌ها را فشرده نمود بلکه باید روشی را برای فشرده‌سازی ابداع نمود تا تصحیح خطا در مقصد را پشتیبانی نموده و وابستگی داده‌ها را نیز به حداقل برساند. زیرا ایجاد هر ارتباط برای دریافت داده ای که با خطا روبرو شده، هزینه بسیار زیادی را با خود همراه خواهد کرد. علاوه بر این که ممکن است تا مدت زیادی قادر به ایجاد ارتباط جدیدی نباشیم.

۲-۷- قابلیت به کارگیری در کاربردهای بلادرنگ

گرچه کاربردهای بلادرنگ در بسیاری از مواقع بستگی مستقیم به قابلیت پیاده‌سازی روش-ها با سرعت قابل قبول دارند، ولی همانگونه که خواهیم دید، بعضی از روش‌های فشرده‌سازی بسیار

سریع که قادر به رعایت تمامی Deadline ها هستند، قابلیت پیاده‌سازی بـه صـورت بلادرنـگ را ندارند.

از جمله این روش‌ها، می توان به روش Huffman استاندارد اشاره نمـود. اشـکال بـزرگ این روش، در نیاز آن به مرور اولیه بر روی تمامی‌داده ها، برای ساختن درخـت Huffman اشـاره نمود.

واضح است که در بسیاری از کاربردهای بلادرنگ، به خصوص در کاربردهای بلادرنگی که برای انتقال داده‌ها مطرح می شوند، در هیچ لحظه ای تمام داده‌ها در اختیار ما نیسـتند و امکـان بـافر کردن این داده‌ها نیز وجود ندارد. زیرا تمام deadline ها نقض خواهند شد.

از دیگر روش‌هایی که قابلیت پیـاده سـازی بلادرنـگ را ندارنـد، مـی تـوان بـه بسـیاری از روشهای Dictionary Based اشاره کرد که بر اساس بیشترین دوتایی‌های تکراری، دیکشنری را به صورت دینامیک به وجود می آورند و با استفاده از این دیکشنری، فایل را فشرده می‌نماینـد. ایـن روش‌ها نیز درست همانند روش Huffman، به تمام فایـل در آن واحـد نیازمندنـد و حتـی بـدتر اینکه این روش‌ها، فایل را فقط برای آمارگیری اولیه، نمی‌خواهند، بلکـه کـل روش فشـرده‌سـازی و ساختن دیکشنری، بر اساس داشتن فایل بنا نهاده شده است.

اینکه چطور می توان روشی را که قابل پیاده سازی بلادرنگ نیست، بـه صـورت بلادرنـگ تغییر داد و سپس پیاده‌سازی کرد، دقیقا بستگی به روش مورد بحث دارد و شاید در بعضی از موارد اصلا قابل انجام نباشد. در اینجا با بحث در مورد تغییر دو روش بالا به روش بلادرنگ، سـعی مـی-کنیم که موضوع را شرح دهیم.

برای اینکه Huffman را به روشی بلادرنگ (آنگونه که در مودمها از آن استفاده می‌شود)، باید نیاز آن را به آمارگیری اولیه حذف کنیم. برای این کار، در بسیاری مواقع یک آمار از پیش تعیین‌شده و در حقیقت، یک درخت Huffman آماده در اختیار آن می‌گذاریم. در روشی دیگر که البته برای مودمها نمی‌توان استفاده کرد، نمونه ای کوچک از داده ای را که قرار است فشرده شود، قبل از انجام عمل بلادرنگ در اختیار آن قرار می‌دهیم تا بتواند درخت Huffman را بر اساس آن ساخته و برای باقی‌داده ها نیز از همین درخت استفاده کند.

روشهای Dictionary Based را نیز می توان با در اختیار قرار دادن داده‌های نمونه(Training Data) قبل از شروع اعمال بلادرنگ لازم، به روش‌های بلادرنگ تبدیل نمود و نتیجه به نسبت خوبی هم گرفت.

ولی شاید بهترین کار در طراحی سیستم‌های بلادرنگ، استفاده از روش‌هایی باشد که قابلیت پیاده سازی بلادرنگ را دارند. از نمونه این روش‌ها می توان به روش‌های LZW و فشرده-سازی ریاضی اشاره نمود.

یکی از عامل‌های موثری که در مقایسه روش‌های فشرده‌سازی بلادرنگ می توان استفاده نمود، مقدار فشرده‌سازی روش در مقابل حجم داده فشرده شده در ثانیه است.

۲-۸- قابلیت کار در حجم بالا یا پایین داده ها

این قابلیت را به خودی خود نمی توان نقطه قوت یک روش یا ضعف روش دیگر بیان نمود. بلکه وقتی هدف ما از روش فشرده‌سازی معلوم باشد، می‌توانیم با استفاده از این پارامتر، الگوریتمی را که برای ما مناسب است انتخاب کنیم.

به عنوان مثال، روش XRAY که در این مقاله در جاهای متفاوتی به آن اشاره شده است، روشی است که برای حجم‌های بزرگ داده کارآیی زیادی دارد و برای حجم کوچک داده‌ها نمی-تواند این کارآیی را بروز دهد. ولی روش فشرده‌سازی ای که در مودم‌ها استفاده می شود، باید برای حجم کوچک داده‌ها کارآیی خوبی داشته باشد. چرا که کل داده‌هایی که در یک Session از طریق مودم منتقل می شود، حجمی نزدیک ۱۰ مگا بایت یا شاید کمی بیشتر را دارد.

۲-۹- قابلیت ویرایش داده‌های فشرده شده

در بسیاری از کاربردهای فشرده سازی، مثل پایگاه‌های داده‌ها، روش‌های فشرده‌سازی باید قادر به تغییر داده‌های ذخیره شده باشند.

در روش‌هایی مثل Huffman، و یا فشرده‌سازی‌های Dictionary Based، این عمل می‌تواند بدون صدمه زدن به داده‌های اصلی انجام پذیرد.

ولی در روش‌هایی مانند LZW و فشرده‌سازی ریاضی، این عمل بدون تغییر لااقل بخشی از داده‌ها قابل انجام نیست.

در حالت کلی، روش‌هایی که کدهای داده‌ها را به صورت دینامیک تغییر می دهند، نمی-توانند داده‌ها را به سادگی ویرایش کنند. در مقابل، روش‌هایی که یا کدها را از ابتدا می دانند و یا دیکشنری‌ای برای تخصیص کدها درست کرده و در قایل مقصد ذخیره می کنند، مشکل چندانی برای ویرایش داده‌ها ندارند.

گرچه که قابلیت ویرایش داده‌ها می تواند به صورت ویژگی ای برای بسیاری از روش‌های فشرده سازی ارائه شود، لیکن معمولا تغییر داده‌ها در نرم افزارهای فشرده‌ساز فعلی به صورت حذف فایل تغییر داده شده و اضافه کردن نسخه جدید فایل مربوطه به آرشیو پیاده سازی می‌شود و شاید تنها جایی که این قابلیت به صورت کامل پیاده سازی می شود، پایگاه‌های داده باشد.

۲-۱۰- آیا نیاز به داده‌های مرجعی برای استخراج داده‌های اولیه داریم؟

در بعضی از روش‌های فشرده‌سازی همانند روش Huffman، ابتدا یک سری از داده‌های مرجع، برای کد کردن داده ابتدایی به کار می روند و سپس، تمام داده‌ها با استخراج کدهایی که از این داده مرجع نتیجه می شوند، فشرده خواهند شد.

نیاز به داده‌های مرجع، معمولا ضعف روش فشرده‌سازی را می رساند. زیرا از سویی، ذخیره نمودن این داده‌ها باعث ایجاد حجم بیشتر و نسبت فشرده‌سازی کمتر می شود و از سوی دیگر این داده ها، نقش اساسی ای را در استخراج داده‌های ابتدایی بازی می کنند. به عبارتی، با گم شدن این داده ها، داده های ابتدایی غیر قابل استخراج خواهند شد.

البته نمی توان این خصوصیت را همیشه بد دانست، چرا که در بعضی از موارد، می تـوان از تمام یا بخشی از داده‌های مرجع، به عنوان کلیدی برای استخراج داده‌های اصلی اسـتفاده نمـود و در نتیجه، تنها کسی قادر به استخراج داده‌های ابتدایی است که داده‌های مرجع را در اختیـار دارد. لـیکن در عمل، این کار نه مقدور است و نه به صرفه و در نتیجه، تا کنون هیچ نرم افزار شناخته‌شـده‌ای، از این خصوصیت به این گونه استفاده نکرده است. اکثرا ترجیح با روش‌های معمولی است که مطمئن-تر نیز هستند.

فصل سوم

فشرده سازی مبتنی بر کد کردن سمبولها[1]

۳-۱- الگوریتم Shannon-Fano

اولین متد شناخته شده برای کدینگ موثر سمبولها، کدینگ Shannon-Fano است. آقایان Claude Shannon از آزمایشگاههای بل و R.M. Fano از MIT تقریبا همزمان با هم این متد را توسعه دادند. این متد نیاز داشت تا احتمال وقوع هر سمبول در ورودی را بداند. با دانستن این احتمالات، جدولی از کدها تولید می‌شود که چندین خصوصیت مهم دارد:

- کدهای متفاوت، طولهای متفاوتی دارند.
- کد سمبول‌هایی که احتمال کمتری دارند، طول بیشتری دارد، و کد سمبول‌هایی که احتمال بیشتری دارند، طول بیت کمتری دارد.
- گرچه طول کدها متفاوت است، ولی می‌توانند به صورت یکتا بازیابی شوند.

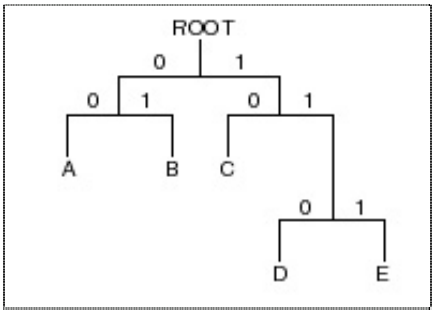
[1] این بخش ترجمه اي است از مرجع شماره ۲۲

دو خصوصیت اول، دست به دست می‌چرخند. توسعه‌دادن کدهای با طول‌های متفاوت، عمل فشرده‌سازی را ممکن می سازد و نشان‌دادن آنها به صورت درخت باینری، دیکد کردن آنها را ممکن می‌سازد.

مثالی از درخت دیکدی که در کدینگ Shannon-Fano استفاده می‌شود، در زیر نشان داده شده است. دیکد کردن یک کد در ورودی به این معنی است که در ریشه این درخت بایستیم و بر اساس بیت فعلی در ورودی، به سمت راست و یا چپ بچرخیم. به محض اینکه به برگی از این درخت رسیدیم، یکی از سمبول‌ها دیکد شده اند.

شکل ۱ نشان‌دهنده درخت Shannon-Fano ای است که برای کد و یا دیکد کردن الفبای پنج حرفی ساده‌ای که فقط از حروف A تا E تشکیل شده است، استفاده می‌شود. با مرور این درخت، جدول کدها به دست می آید.

سمبول	فرکانس تکرار	کد
A	۱۵	۰۰
B	۷	۰۱
C	۶	۱۰
D	۶	۱۱۰
E	۵	۱۱۱



شکل ۱- یک درخت ساده Shannon-Fano

ساختار درختی نشان‌دهنده آن است که چگونه کدها با وجود آنکه تعداد متفاوتی بیت دارند، به صورت یکتا توصیف شده اند. گرچه که به نظر می رسد این ساختار درختی برای کامپیوترها طراحی شده است، ولی باید دانست که آنها برای ساختارهای سوییچی دهه ۱۹۵۰ (مثل ماشین‌های تله‌تایپ) نیز مفید بوده است.

۳-۱-۱- الگوریتم ساختن درخت کدینگ Shannon-Fano:

درخت Shannon-Fano بر اساس نیاز به تعریف جدول کد موثر طراحی شده است. الگوریتم اصلی بسیار ساده است:

- برای لیست مشخصی از داده‌ها، احتمال رخداد آنها در متن را محاسبه نموده و در کنار آنها نگه‌داری کن.
- این لیست را بر اساس بیشتر بودن فرکانس رخداد سمبول‌ها مرتب کن. طوری که بیشترین رخداد در بالا و کمترین رخداد در پایین قرار گیرد.

- این لیست را طوری تقسیم کن که تفاضل جمع فرکانس‌های دو نیمه بالایی و پایینی حداقل باشد.
    - به نیمه بالایی کد صفر و به نیمه پایینی که یک را اختصاص بده.
    - قدمهای ۳ و ۴ را به صورت بازگشتی برای هر کدام از نیمه‌ها انجام بده و گروه‌ها را به دسته‌های کوچکتر بشکن تا بالاخره هر سمبول به یک برگ در این درخت تبدیل شود.

گرچه که الگوریتم Shannon-Fano گام بزرگی به سمت جلو بود، ولی خیلی سریع و با به میان آمدن الگوریتم Huffman کنار گذاشته شد.

۳-۲- الگوریتم Huffman:

الگوریتم کدینگ Huffman در بسیاری از خصوصیات با کدینگ Shannon-Fano مشترک است. این روش نیز، کدهای با طول متغیر تولید می کند که تعداد صحیحی بیت دارند. سمبول‌های با احتمال بالاتر، کدهای کوتاهتر می‌گیرند. کدهای هافمن خصوصیت پیشوندهای یکتا را نیز دارند، یعنی می‌توانند به صورت صحیحی دیکد شوند. این کار معمولا با دنبال کردن یک درخت دیکد باینری تضمین می‌شود.

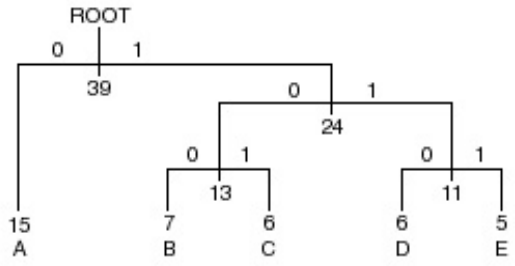
ساخت درخت هافمن با روشی کاملا متفاوت از آنچه در Shannon-Fano انجام می‌شود، انجام می‌گیرد. در حالی که در روش قبلی درخت از بالا به پایین ساخته می‌شد، در این روش، از پایین به بالا ساخته خواهد شد.

روال ساخت درخت ساده و بی‌نظیر است. سمبول‌ها در ردیف پایین به عنوان نودهایی که قرار است توسط درخت باینری ساخته شوند، قرار داده خواهند شد. هر نود وزنی مخصوص به خود دارد که فرکانس تکرار و یا احتمال دیده شدن آن است. سپس درخت با گام‌های زیر ساخته خواهد-شد:

- دو نودی که کمترین وزن را دارند و هنوز استفاده نشده‌اند، مشخص می‌شوند.
- یک نود پدر برای این دو نود ساخته می شود و وزن آن برابر مجموع وزن این دو فرزند قرار می‌گیرد.
- نود پدر به لیست نودها اضافه می گردد و نودهای فرزند از این لیست حذف می گردند.
- یکی از فرزندان، به دلخواه، کد صفر و دیگری کد یک را به خود اختصاص می دهند.
- این گامها تا هنگامی که فقط یک نود باقی مانده باشد، ادامه خواهند یافت. وقتی که تنها این نود وجود داشته باشد، این نود ریشه درخت خواهد بود و الگوریتم خاتمه خواهد-یافت.

این الگوریتم می تواند به سمبول‌هایی که در مثال قبلی به کار رفت، اعمال شود. نتیجه به شکل زیر خواهد بود:

سمبول	فرکانس تکرار	کد
A	۱۵	۰
B	۷	۱۰۰
C	۶	۱۰۱
D	۶	۱۱۰
E	۵	۱۱۱



شکل ۳-۲- درخت هافمن

برای ساختن کد هر سمبول، نیازمند آن هستیم که مسیر از ریشه به برگ مربوط به آن را طی کرده و اعدادی که در مسیر وجود دارند را پشت سر هم بنویسیم.

با دقت در جدول کدینگ دو روش در می یابیم که طول کد سمبول A به یک بیت کاهش یافته، ولی طول کد دو سمبول B و C به سه افزایش یافته است که بر طبق جدول زیر، در کل، ۲ بیت فایل را فشرده‌تر می نماید:

سمبول	تعداد	تعداد بیت در Shannon-Fano	تعداد کل بیتها در Shannon-Fano	تعداد بیتها در Huffman	تعداد کل بیتها در Huffman
A	15	1	30	2	15
B	21	3	14	2	7
C	18	3	12	2	6
D	18	3	18	3	6
E	15	3	15	3	5

در حالت عمومی، کدینگ Shannon-Fano و Huffman از لحاظ بهینگی نزدیک به هم عمل می کنند، ولی کدینگ Huffman همیشه نتیجه‌ای بهتر و یا خداقل مساوی نتیجه Shannon-Fano را خواهد داشت. در نتیجه، این کدینگ به کدینگ غالب این زمینه تبدیل شده است. از آنجا که هر دوی این روش‌ها قدرت پردازشی یکسانی را می طلبند، عقلاتی است که آن را انتخاب کنیم که نتیجه بهتری می دهد. علاوه بر این، هافمن ثابت کرد که روش وی با هیچ روش دیگری که طول بیت‌هایش صحیح باشد، قابل بهینه شدن نیست.

از زمانی که هافمن برای اولین بار در سال ۱۹۵۲ مقاله خود بـا نـام A method for the construction of minimum redundancy codes را منتشـر کـرد، الگـوریتم کـدینگ وی دستخوش هجمه بزرگی از تحقیقات دیگر بوده است. ژورنال‌های تئوری اطلاعات تـا کنـون، تعـداد زیادی مقاله بر روی پیاده‌سازی‌های جنبه‌های مختلف کدهای هافمن منتشر کرده‌اند. کـدینگ هـافمن در برنامه‌های تجاری فشرده‌سازی، ماشین‌های Fax، و حتی استاندارد JPEG استفاده شده است.

۳-۳- الگوریتم هافمن تطبیق پذیر:

در بخش قبل، دیدیم که چگونه کدینگ هافمن مـی توانـد بـا کـاهش افزونگـی در کـدینگ سمبول‌ها به صورت بهینه داده‌ها را فشرده نماید. کدینگ هافمن به خودی خود بیـان نمـی کنـد کـه چگونه محتوای اطلاعاتی هر سمبول را کاهش دهیم. بلکه هر مـدلی کـه قـادر بـه محاسـبه احتمـال رخداد یک سمبول باشد، باید بتواند کدینگ هافمن را برای فشرده نمودن داده‌ها استفاده نماید.

مثال‌های بخش قبل، همگی از مدل‌های درجه صفر استفاده می نمودند که همگی مسـتقل از متن هستند. یعنی آنکه احتمال رخداد هر کاراکتر بدون توجـه بـه کاراکترهـایی کـه قبـل از آن آمـده بودند، محاسبه می شود. برنامه موجود در CD برای الگوریتم هافمن نیز ابتـدا تمـام فایـل ورودی را تحلیل می کند و جدولی از احتمالهای هر سمبول می سازد. هنگامی که این احتمالات از یک توزیـع کاملا ثابت نتیجه شده باشند، ما قادر خواهیم بود که داده‌ها را فشرده نماییم.

یکی از مشکلات کدینگ هافمن، نیازش به دانستن جدول احتمالات است که حداکثر حدود ۲۵۰ بایت را اشغال می کند. این مقدار به خودی خود کوچک است و حتی برای فایل‌هـای کوچـک

هم تغییر زیادی را ایجاد نمی کند. آنچه این مشکل را بزرگ می سازد، آن است که با تلاش ما بـرای افزایش قدرت برنامه‌مان، این سربار به شدت رشد می کند و بیشتر و بیشتر قابل ملاحظـه مـی شـود. اگر از مدل‌های درجه صفر به مدل‌های درجه یک سوییچ کنیم، مجبور خواهیم بود که ۲۵۷ تا از ایـن جدول‌ها را به جای یکی از آن‌ها ذخیره کنیم و تا هنگامی که فایل‌های ما بسیار بـزرگ نشـوند، ایـن پنالتی، هرگونه پیشرفتی را که از افزایش درجه حاصل شده است، از بین خواهد برد.

به نظر می رسد که به نقطه غیر قابل عبوری رسیده ایم. برای فشرده‌سازی بهتر نیاز به داشتن آمار و اطلاعات بیشتری داریم. با داشتن آمار و اطلاعات بیشتر، به فشرده‌سازی‌های بهتری نیز دسـت می‌یابیم، و لی در عوض داده‌های مدل‌سازی بیشتری را نیز نیاز خواهیم داشت که تمام نتایج به دست آمده را از بین خواهد برد.

خوشبختانه، راهی برای خارج شدن از این دو راهی موجود است. کدگذاری تطبیق پذیر، بـه ما این اجازه را می‌دهد که از مدل‌های با درجات بالاتر بدون پرداخـت هزینـه اضـافی بـرای آمـار و اطلاعات استفاده کنیم. این کار با به وجود آوردن درخت هافمن در حین اجرا به دست خواهد آمـد. درخت هافمن تولید شده بر اساس داده‌هایی که تا به حال دیده‌شده‌اند، تولید می‌شود و هـیچ دانشـی درباره آمار داده‌های بعد از آن را ارائه نمی دهد.

کدگذاری تطبیق پذیر، چیزی نیست که فقط با روش هافمن بتواند انجـام شـود. در حقیقـت تقریبا هر روش فشرده‌سازی‌ای می تواند به گونه‌ای تغییر یابد که از خاصیت تطبیق پـذیری اسـتفاده کند. شکل‌های زیر، شبه کدی به زبان C برای عملیات کد کردن و دیکد کردن داده‌هـا در الگـوریتم-های تطبیق پذیر می‌باشند.

```
initialize_model();
do {
     c = getc( input );
     encode( c, output );
     update_model( c );
} while ( c !=  EOF );
```

الگوریتم فشرده‌سازی تطبیق‌پذیر، عملیات کد کردن داده‌ها

```
initialize_model();
while ( ( c = decode( input ) ) ! = EOF ) {
    putc( c, output );
    update_model( c );
}
```

الگوریتم فشرده‌سازی تطبیق پذیر، عملیات دیکد کردن

کدگذاری تطبیق پذیر ازآنجا می تواند به درستی عمل کند که دو تابع Initialize_model() و Update_model() در شبه کدهای بالا دقیقا به یک شکل عمل می‌کنند. اگر یکی از این دو تابع در شبه کد مربوط به فشرده‌سازی، تفاوت هر چند کوچکی با تابع متناظر در شبه کد دیگر داشته باشد، تمام روند کار بر هم خواهد خورد.

این نوع از کدگذاری بسیار ساده است. فشرده ساز و دیکد کننده، با مدل‌های کاملا یکسانی، کار را شروع خواهند کرد. بنابراین، وقتی فشرده ساز، اولین سمبول خود را تولید می کند، دیکد کننده قادر خواهد بود که آن را تفسیر نماید.

هنگامی که فشرده ساز، اولین سمبول را تولید می کند، به پروسه update_model() می رود و در اینجاست که بخش اصلی این گونه روش‌ها انجام می پذیرد. پروسه به‌هنگام‌رسانی مدل، با احتساب کاراکتر جدید، فرکانس رخداد آن کاراکتر و کد اختصاص داده شده به آن (و در صورت

لزوم، بقیه کاراکترها) را تغییر می دهد. در الگوریتم هافمن، این کار به معنای اصافه نمودن شمار یک سمبول خاص و به روز رسانی درخت هافمن می باشد.

۳-۳-۱- به روز رسانی درخت هافمن:

شاید ساده‌ترین کار برای به روز رسانی درخت هافمن، شبه کدی باشد که در زیر آمده است. گرچه که این کد، درست کار خواهد کرد، ولی احتمالا برنامه ما را به کندترین الگوریتم فشرده‌سازی دنیا تبدیل خواهد کرد.

```
update_model( int c )
{
    counts[ c ]++;
    construct_tree( counts );
}
```

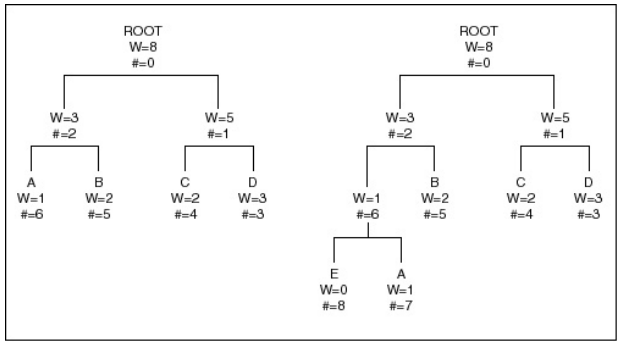
شبه کد ساده برای به روز رسانی درخت هافمن

این الگوریتم، در هر مرحله، درخت هافمن را دوباره بازسازی می کند و به دلیل هزینه مخاسباتی سنگین این کار، انجام دادن آن پس از دریافت هر کاراکتر، کار عقلانی ای نمی‌باشد.

خوشبختانه، راهی وجود دارد که با داشتن یک درخت هافمن و صرف هزینه محاسباتی کمی، بتوان آن را دوباره بازسازی نمود. تمام آنچه برای این کار لازم است، نگاه کمی متفاوت به پروسه ساختن درخت می باشد. این روند، موضوعی را که با نام خاصیت برادری معروف است، به کار می برد. درخت هافمن، درختی دودویی است که به هر گره، چه داخلی و چه برگ، وزنی را نسبت می دهد. هر گره (به غیر از گره ریشه) دارای یک گره برادر (گره ای که والد مشترکی با آن

گره دارد) است. یک درخت وقتی شامل خاصیت برادری است که اولا بتوان گـره هـا را بـه ترتیـب افزایش وزن لیست کرد و ثانیا، هر گره در مجاورت برادر خود ظاهر گردد.

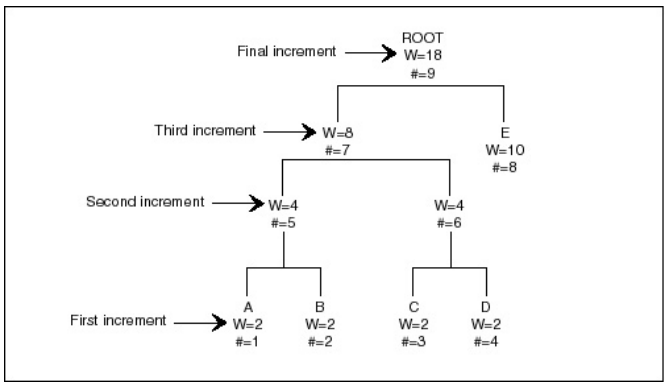
یک درخت دودویی، درخت هافمن است، اگر و فقط اگر، از خاصیت برادری پیـروی کنـد. شکل زیر، نشان دهنده درخت هافمنی است که چگونگی عملکرد این خاصیت را شرح می‌دهـد. در این درخت، به هر گره، عددی نسبت داده شده است، اعداد از ردیف بالا به ردیف پایین و از چپ به راست نسبت داده شده است. این درخت، بر اساس یـک الگـوریتم هـافمن معمـولی و بـا وزن‌هـای A=1، B=2، C=2، D=2، E=0 ساخته شده است.



شکل ۳-۳- یک درخت هافمن

خاصیت برادری از آن جهت در کدینگ تطبیق پذیر هافمن اهمیت دارد که نشان‌دهنده آنچـه برای به روز رسـانی درخـت هـافمن لازم اسـت، مـی باشـد. حفـظ خاصـیت بـرادری در حـین بـه روزرسانی، ما را مطمئن خواهد ساخت که درخت‌های قبـل و بعـد از تنظـیم فرکـانس‌هـای تکـرار، درخت هافمن هستند.

به روز رسانی درخت هافمن از دو عمل ابتدایی تشکیل شده است. عمل اول، اضـافه کـردن شمار داده‌ها است که به سادگی قابل انجام است و در شکل زیر دیده می شود:

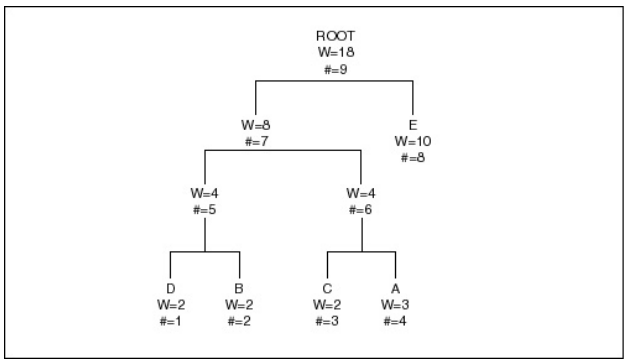


شکل ۳-۴- پروسه افزایش شمار داده‌ها

عمل دومی که برای روند به روزرسانی نیاز است، هنگامی که درخت به دست آمده از قانون برادری پیروی نمی کند، نیاز می‌شود. این موضوع وقتی اتفاق می افتد که گـره افـزایش یافتـه، دارای مقدار مساوی با گره بعدی موجود در لیست باشد. در این حالت اگر عمل افزایش بدون توجـه ادامـه یابد، دیگر درخت هافمنی وجود نخواهد داشت.

وقتی با چنین افزایشی که قانون برادری را بر هم می زند، برخورد کردیم، باید گـره مربوطـه را به محل بالاتری در لیست انتقال دهیم. یعنی اینکه، باید این گره را از پدرش جدا نموده و جای آن را با گره دیگری در لیست عوض کنیم.

شکل زیر، نشان‌دهنده همان درخت هافمن در شکل قبلی اسـت کـه گـره A دوبـاره در آن افزایش یافته است. در پروسه عوض نمودن جای دو گره، این کار آنقدر تکرار می شود، تا گره بعدی در لیست، مقداری کمتر از مقدار گره فعلی لیست داشته باشد.



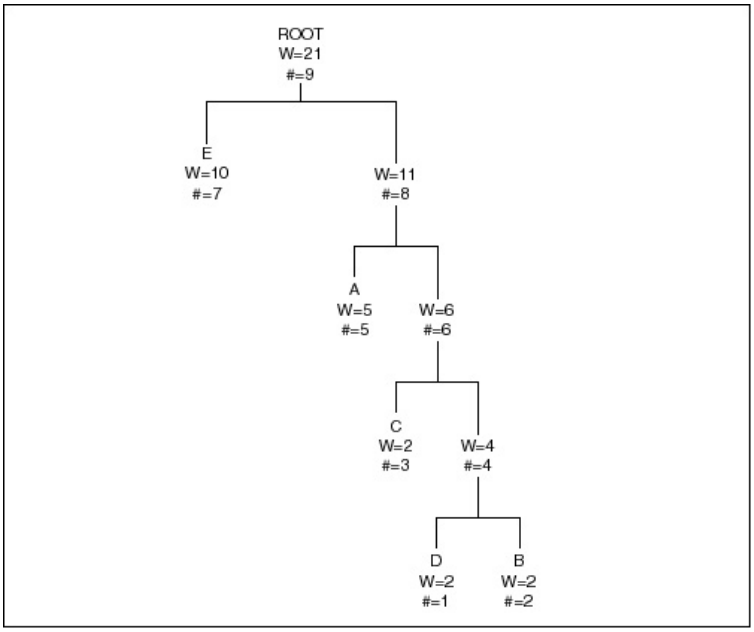
شکل ۳-۵- پس از انجام عمل تعویض (فقط گره A افزایش یافته است)

در این شکل، ابتدا مقدار شمار تکرار گره A از ۲ به ۳ افـزایش داده شـد، از آنجـا کـه گـره بعدی (گره B) دارای شمار تکرار ۲ بود، پس درخت دیگر از قانون برادری پیـروی نمـی کـرد. ایـن موضوع، علامتی است که نیاز به انجام عمل تعویض را مشخص می کند. برای عمـل تعـویض، ابتـدا آخرین گره‌ای که دارای وزن ۲ است، مشخص می‌شود (گره D). سپس گـره‌هـای A و D تعـویض می شوند تا درخت حاصل دارای ترتیب صحیحی باشد.

پس از انجام عمل تعویض، کار به روز رسانی درخت می تواند ادامه یابد. گره بعدی که باید افزایش یابد، والد فعلی گره تعویش شده می باشد. در شکل بالا، این گره، همان گره داخلی با شماره ۶ است. پس از افزایش هر گره، ترتیب صحیح گـره هـا بررسـی مـی شـود و عملیـات تعـویض در صورت نیاز انجام می‌گردد.

آنچه در شکل بالا مشخص نگردید، تاثیر عمل تعویض در کدهای تخصیص داده شده به هر سمبول بود. گرچه جای دو گره A و D عوض شد، ولی طول کدهای تخصیص داده شـده بـه آنهـا، همان ۳ بیت باقی ماند. شکل زیر نشان دهنده آنچه پس از دو بار افزایش دیگر در شمار گـره A رخ خواهد داد، می باشد. همانگونه که در شکل مشخص است، گره A، آنقدر افـزایش یافتـه اسـت کـه

توانسته به سطح بالاتری که با دو بیت نمایش داده می شود، منتقل گردد. درعوض، گره های B و D به سطوح پایین تری که با ۴ بیت قابل نمایش است، منتقل گشته اند. گره C هنوز دست نخـورده در سطح سوم باقی است.



شکل ۳-۶- پس از انجام یک عمل تعویض دیگر

آنچه در شکل زیر آمده است، شبه کدی برای انجام عملیاتی که در بالا توضیح داده شد، می باشد.

```
for ( ; ; ) {
     increment nodes[ node ].count;
     if ( node == ROOT )
         break;
     if ( nodes[ node ].count > nodes[ node + 1 ].count )
         swap_nodes();
     node = nodes[ node ].parent;
}
```

شبه کد لازم برای الگوریتم به روز رسانی درخت هافمن

روال swap_nodes() باید در لیست گره‌ها پیمایش نماید تا به آخـرین گـره ای کـه ایـن مقدار را دارد، برسد. این گره، همان گره‌ای است که باید تعویض روی آن انجام شود. روال مربوطـه، شبیه چیزی که در شبه کد زیر از نظرتان می گذرد، خواهد بود.

```
swap_node = node + 1;
while ( nodes[ swap_node + 1 ].count < nodes[ node ].count )
     swap_node++;
temp = nodes[ swap_node ].parent;
nodes[ swap_node ].parent = nodes[ node ].parent;
nodes[ node ].parent = temp;
```

شبه کد لازم برای عمل swap_node()

۳-۳-۲- یک بهبود دیگر در الگوریتم تطبیق پذیر هافمن

یکی از راههای بهینه نمودن بیشتر الگوریتم هافمن، اطمینان از عدم صـرف فضـای کـدینگ برای داده‌هایی که به کار نمی روند، می باشد. در الگوریتم هافمن معمولی، این کار ساده به نظر مـی-رسید. چرا که ما قبل از انجام هر عملی، آمار اطلاعات درون فایل را جمع می کردیم و در نتیجه، بـه سادگی می‌توانستیم از داده‌هایی که count برابر با صفر دارند، صرف نظر کنـیم و کـدی را بـه آنهـا اختصاص ندهیم.

در یک پروسه تطبیق‌پذیر، ما نمی‌دانیم که چه داده‌هایی در آینـده دیـده خواهنـد شـد و چـه داده‌هایی دیده نخواهند شد. بنابراین، یک راه بدیهی برای مقابله با این روند، مقداردهی اولیه درخـت هافمن با تمام سمبولها و اختصاص فرکانس ۱ به آنها می‌باشد. وقتی که پروسه کدگذاری شروع می-گردد، هر داده ۸ بیت طول خواهد داشت که فضای داده‌ها را به هدر خواهد داد. شاید راه بهتـر ایـن

باشد که درخت اولیه خالی باشد و پس از آن، با دیده شدن هر سمبول جدید، سمبول مربوطه وارد درخت شود. اما این کار در ظاهر با پروسه کدگذاری تناقض دارد. چرا که ما اولین دفعه برخورد با یک سمبول، کد مربوط به آن سمبول را نمی دانیم و در نتیجه نمی توانیم آن را در فایل خروجی وارد کنیم.

جواب معمای بالا، کد فرار (Escape code) است. کد فرار، کدی است که به کدکننده فرستاده می شود تا به آن بگوید که ما در حال فرار از حالت فعلی درخت هافمن هستیم. دیکد کننده با این شرایط خواهد فهمید که کد بعدی در حالت فعلی سیستم (درخت هافمنی که فعلا مورد استفاده است) موجود نیست.

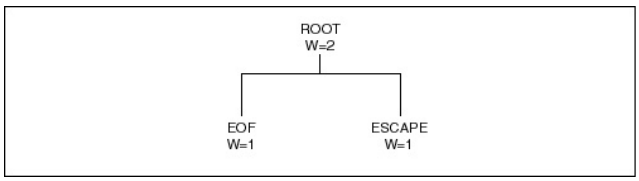
شبه کد زیر که به زبان C نوشته شده است، بیانگر آنچه در روال اصلی کدکننده داده‌ها رخ می دهد، می باشد.

```
encode( char c )
{
     if ( in_tree( c ) )
          transmit_huffman_code( c, out_file );
     else {
          transmit_huffman_code( ESCAPE, out_file );
          putc( c, out_file );
          add_code_to_tree( c );
     }
     update_tree( c );
}
```

روال اصلی کد کننده داده‌ها با ویژگی داشتن کد فرار

روال بالا نشان‌دهنده آن است که کد فرار همانند هر کد دیگری از درخت هافمن در خروجی ظاهر می گردد. بنابراین، این کد باید در درخت هافمن نیز موجود باشد.

شکل زیر نشان‌دهنده درخت ابتدایی هافمن است کـه توسـط روال Initialize_model() ساخته می شود. این درخت در ابتدا شامل تنها ۲ برگ برای کد فرار و کـد انتهـای فایـل (End of file) می باشد. و از آنجا که هر دوی آنها در فایل ظاهر خواهند شد، ما با درخت کوچکی کـه فقـط شامل آنها باشد، شروع خواهیم کرد.



درخت هافمنی که با دو برگ آغازش شده است

فصل چهارم
روش‌های فشرده‌سازی
با استفاده از درخت‌ها و Trieها

۴-۱- درخت‌ها و Tries

ما در این بخش فقط با درخت‌های ریشه دار سروکار داریم. درخت‌ها با قرار دادن راس ریشه در بالاترین نقطه و فرزندهای هر راس، در نزدیک ترین سطح ممکن در زیر آن‌ها نمایش داده می شوند. یک راس با حداقل یک فرزند را راس داخلی و راس‌های بدون فرزند را برگ درخت می‌نامند. عمق یک راس برابر تعداد راس‌های واقع بر مسیر بین ریشه تا آن راس است. بیشترین عمق راس‌های یک درخت را ارتفاع درخت می نامیم.

Trie یک درخت است که در آن رشته‌هایی از سمبل‌ها در مسیر از ریشه تا دیگر راس‌های درخت نمایش داده می شود. هر یال با یک رشته غیر تهی از سمبل‌ها برچسب گذاری شده است و هر راس متناظر با رشته حاصل از به هم چسباندن رشته‌های قرار گرفته روی یال‌ها در مسیر از ریشه تا آن راس می باشد. ریشه نمایانگر رشته لامبدا[۱] است. به ازای هر رشته موجود در Trie، همه پیشوندهای[۲]

---

[۱] Lambda
[۲] Prefix

آن رشته هم در Trie موجود است (گاهی به این ساختار داده درخت دیجیتال[1] هم گفته می شود. در این بحث ما تفاوتی بین Trie و درخت دیجیتال قائل نیستیم).

یک Trie را **مسیر فشرده**[2] می نامیم اگر تمام مسیرهای یک راس با یک فرزند به هم چسبیده باشند. یعنی تمام رئوس داخلی بجز احتمالا ریشه حداقل دو فرزند داشته باشند. یک Trie مسیر فشرده در بین تمام Trieهایی که یک مجموعه از رشتهها را نشان می دهند کمترین تعداد راس‌ها را دارد. در این نوع Trie ،اگر و فقط اگر Trie به ازای دو a و b مختلف، شامل دو رشته αa و αb باشد آنگاه یک رشته مانند α می تواند با یک راس داخلی متناظر شود. طول هر رشته متناظر با یک راس، با عمق آن راس در Trie برابر است.

در ادامه فرض می کنیم که همه Trieهایی که با آن‌ها سر و کار داریم یا مسیر فشرده هستند و یا هر یال آن‌ها با یک رشته به طول یک برچسب گذاری شده است (در نتیجه عمق راس و طول رشته متناظر با آن برابر خواهد بود).

یک **Trieمرتب شده الفبایی**، یک Trie است که در آن رشته‌های نمایش داده شده توسط برگ‌ها، در یک پیمایش میان ترتیب[3] به صورت مرتب ظاهر شوند.**Trie مرتب نشده الفبایی** هیچ تضمینی برای وجود این خاصیت نمی دهد.

---

[1] Digital Tree
[2] Path Compressed
[3] In-Order Traversal

۴-۱-۱- خصوصیات ویژه Trieها برای نگه داری داده‌ها

خصوصیت ویژه ای که ساختار داده ای Trie برای ما فراهم می کند آسان کردن جستجو در بین رشته‌های موجود در آن Trie است. برای یافتن یک رشته در Trie ، از ابتدای رشته و ریشه درخت شروع می کنیم و در هر مرحله اولین سمبل رشته را با یک یال مطابقت داده و آن سمبل را حذف می کنیم و آن یال را می پیماییم و این کار را تا پایان یافتن رشته مورد نظر، یا عدم امکان تطبیق ادامه می دهیم. زمان این جستجو متناسب با طول رشته مورد نظر به اضافه زمان لازم برای انتخاب یال-های مسیر است. انتخاب یال‌ها، دشوارترین قسمت این پروسه است و به مقدار زیادی وابسته به ساختمان داده مورد استفاده برای ذخیره سازی اطلاعات بستگی دارد.

بنابراین هنگام انتخاب روش پیاده سازی Trie باید از نوع پرسش‌های احتمالی مورد استفاده در آن Trie آگاهی کلی داشت. یکی دیگر از اصول مهم در این زمینه، نگه داری مرتب و درست راس‌های Trie است.

به علت خصوصیات مختلف کاربردها، بحث در مورد انواع Trie ضروری نظر می رسد که ما در این جا به چند نمونه از حالات اشاره می کنیم:

- هر راس Trie می تواند به وسیله یک آرایه به طول مشخص k نشان داده شود که در آن رشته ای که آن راس بدان اشاره می کند ذخیره می شود. این روش امکان جستجوی سریع وجود دارد ولی درعوض به فضای زیادی از نظر حافظه نیاز داریم و همچنین مقدار دهی اولیه راس‌های جدید پیچیده خواهد بود.

- هر راس را می توان به وسیله یک لیست پیوندی[۱] یا درخت جستجوی دودویی[۲] نمایش داد. این روش از هدر رفتن حافظه جلوگیری می کند ولی در عوض هزینه جستجو را افزایش خواهد داد.

- رشته نمایش داده شده توسط هر راس را می توان در یک جدول Hash[3] ذخیره کرد.با استفاده از dynamic perfect hashing[4] می توان اطمینان داشت که جستجو با هزینه تقریبا ثابت، بدون توجه به اندازه الفبا، انجام پذیر است. به علاوه این روش می تواند با روش ذکر شده در بالا، در کنار هم مورد استفاده قرار گیرد[1].

یک نکته مهم در مورد Trieها این است که Trieهای مرتب نشده الفبایی را می توان با هزینه کمتر نسبت به Trieهای مرتب شده الفبایی تبدیل کرد. این کار را می- توان با مرتب کردن همه یال‌ها براساس رشته‌هایی که روی آن‌ها قرار دارد و سپس ساخت دوباره درخت بر اساس ترتیب این یال‌ها، انجام داد.

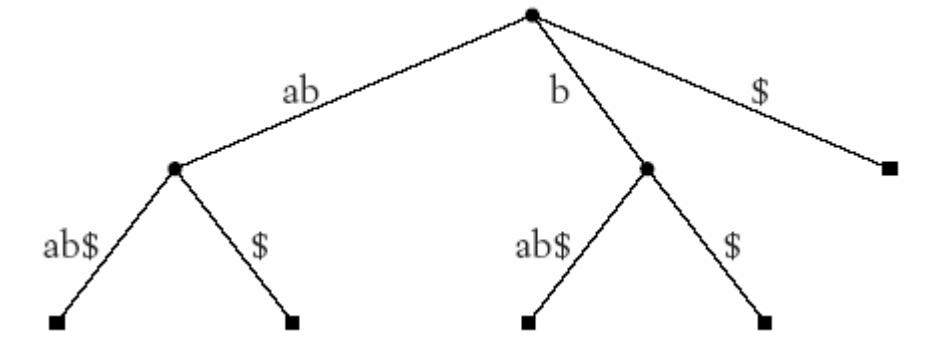
---

۱ Linked List
۲ Binary Search Tree
۳ Hash Table
۴ Dynamic Perfect Hashing

۴-۲- درخت‌های پسوندی[۱]

یک درخت پسوندی (درخت زیرکلمه[۲]) یک رشته عبارت است از یک Trie مسیر فشرده که تمام پسوندهای آن رشته را دارا باشد.

درخت پسوندی یک رشته به طول n، دارای n+1 برگ است که هر برگ برای نمایش دادن یک پسوند که به علامت $ ختم می شود مورد استفاده قرار می گیرد. پس بنابراین، از آن جا که هر راس داخلی حداقل دو فرزند دارد تعداد راس‌های این چنین درختی حداکثر برابر با 2*n+ 1 خواهد بود. در زیر، درخت پسوندی رشته abab$ نشان داده شده است.



شکل ۴-۱- درخت شامل تمام پیشوندهای رشته abab$

درخت‌های پسوندی به عنوان یک شاخص[۳] که اجازه نگهداری و بازیابی زیر رشته‌های یک رشته بزرگ تر را به ما می دهد کاربرد زیادی دارند. در یک درخت پسوندی ساخته شده برای یک رشته خاص می توان بزرگ ترین زیررشته ای که با یک رشته مورد جستجو مطابقت دارد را در زمان

---

۱ Suffix Tree
۲ Subword Tree
۳ Index

مناسبی یافت. در شرایط عادی این بدان معناست که ایجاد یک درخت پسوندی، از نظر زمانی با طول رشته ای که می خواهیم آن را شاخص گذاری کنیم رابطه خطی[1] لازم دارد[2]. همچنین فضای لازم برای ذخیره هم با طول رشته رابطه خطی دارد و زمان لازم برای جستجوی یک رشته ورودی در این نوع درخت‌ها با طول رشته مورد جستجو رابطه خطی دارد[3].

۴-۳- فشرده سازی با استفاده از روش شاخص گذاری پنجره‌های لغزنده

در بسیاری از کاربردها که در آن‌ها، باید زیر- رشته‌های یک رشته با طول زیاد شاخص گذاری شوند، شاخص-گذاری ثابت[2] به تنهایی کافی نیست. به عنوان یکی از نمونه‌های این چنین کاربردهایی می توان به استفاده از شاخص‌ها برای پردازش قسمت‌هایی از یک رشته شاخص گذاری شده قبل از دریافت کامل کل رشته، اشاره کرد. به علاوه، گاهی ما نیازی به نگه داری تمام مسیر تا آغاز ورودی نداریم که در این صورت، می توانیم قسمت‌های قدیمی ورودی را دور بریزیم و در نتیجه به حجم کمتری نیاز داریم.

یکی از زمینه‌های کاربردی که به این نوع شاخص گذاری مورد نیاز است فشرده سازی اطلاعات است. انگیزه اصلی برای حذف اطلاعات قدیمی در مقوله فشرده سازی را می توان در دو زمینه بدست آوردن یک مدل سازگار یا در کاهش فضای مورد استفاده به منظور امکان پذیر کردن پیاده سازی روش‌های پیشرفته برای داده‌های بزرگ دانست.

---

[1] Linear Time
[2] Static Indexing

استفاده از درخت پسوندی برای شاخص گذاری بخش ابتدایی یک رشته، قبل از آن که کل رشته دریافت شده باشد با استفاده از یک الگوریتم ساخت online درخت پسوندی مانند الگوریتم Ukkoneh استفاده شود به طور مستقیم قابل انجام است اما قسمت مبهم کار که همان انتقال نقاط انتهایی شاخص به جلو است باقی می ماند[1] .

در این بخش ما به توضیح چگونگی ارتقا الگوریتم Ukkoneh به یک مکانیزم کامل برای شاخص گذاری پنجره‌های لغزنده که در آن اندازه پنجره متغیر است و در عین حال قدرت و بهینگی درخت پسوندی به طور کامل حفظ شود می پردازیم.

Fiala و Greene علاوه بر ارائه روش فشرده Zip-Lempel که مستقیما به مبحث این بخش مربوط است یک متد برای معتبر نگه داشتن برچسب‌های یال‌های درخت پسوندی هنگام انجام عملیات حذف ارائه داده اند. اما روش آن‌ها برای یک پیاده سازی با پیچیدگی زمانی خطی نسبت به ورودی کافی نبود و همچنین مشکلات زیادی در جا به جا کردن آدرس رشته مورد شاخص گذاری داشته است.

همچنین مساله شاخص گذاری پنجره‌های لغزنده به وسیله یک درخت پسوندی توسط Rodeh ، Pratt و Even مورد بررسی قرار گرفته است. روش آن‌ها مبتنی براجتناب از مسائل مربوط به حذف، با نگه داری همزمان سه درخت پسوندی است. این روش به طور قطع در مقایسه با نگه-داری فقط یک درخت ،حداقل از نظر حافظه مورد نیاز از کارآیی کمتری برخوردار است [4].

۴-۳-۱- ایجاد درخت پسوندی

از آن جا که الگوریتم شاخص گذاری پنجره‌های لغزنده با استفاده از درخت پسوندی نیاز به ارتقا الگوریتم ایجاد درخت‌های پسوندی دارد، در قسمتی که به توضیح الگوریتم پنجره‌های لغزنده می پردازیم به توضیح جزئیات این الگوریتم هم خواهیم پرداخت. ما در این جا یک نسخه از الگوریتم Ukkoneh برای ایجاد درخت پسوندی به صورت همزمان[۱] که مقدار کمی تغییر داده شده را به عنوان پایه کار خود دراین بخش توضیح می دهیم. برای توضیحات دقیق تر می توانید به مقاله Ukkoneh، مرجع [2] مراجعه کنید.

برای دست یابی به نتایج کلی تر، انتهای ورودی را تعریف نشده در نظر می گیریم و فرض می کنیم که طول ورودی بی نهایت باشد. در عوض، برچسب‌های به صورت پویا[۲] نمایش دهنده رشته‌هایی که تا انتهای رشته ورودی مورد شاخص گذاری خواهند بود.

الگوریتم Ukkoneh یک الگوریتم اقزایشی[۳] است. در تکرار iام از این الگوریتم، درخت مربوط به $x_0 \ldots x_i$ را بر اساس درخت مربوط به $x_0 \ldots x_{i-1}$ می سازد. بنابراین در مرحله iام باید به ازای هر پسوند α ازسمبل‌های $x_0 \ldots x_{i-1}$، رشته αx_i که دارای طول i است را به درخت مرحله i-1ام اضافه کنیم. قبل از افزودن رشته αx_i یکی از سه حالت زیر برقرار خواهد بود:

---

۱ Online Suffix Tree Construction Algorithm
۲ Dynamic
۳ Incremental

۱. α دقیقا یک بار در درخت مرحله $i-1$ ام ظاهر شده باشد که بدان معناست که α توسط یکی از برگ‌های درخت مانند s نمایش داده می شود و بنابراین برای افزودن αx_i کافی است به برگ s یک فرزند به نام x_i اضافه کنیم.

۲. α بیش ازا یک بار در درخت مرحله $i-1$ ام ظاهر شده باشد ولی αx_i در آن ظاهر نشده باشد.این بدان معناست که α توسط یکی از راس‌های داخلی درخت نمایش داده می شود و در نتیجه یک برگ جدید باید به درخت مرحله $i-1$ ام اضافه شود تا بتوانیم رشته αx_i را هم در درخت جدید داشته باشیم. علاوه بر این اگر α بوسیله یک راس نمایش داده نشود بلکه در درون یکی از یال‌های درخت قرار داشته باشد (یال‌های درخت با رشته‌هایی با طول بیشتر از یک برچسب گذاری شده باشند) در این حالت باید آن یال را به دو یال و یک راس داخلی جدید تبدیل کرد و αx_i را به عنوان فرزند راس جدید معرفی کنیم.

۳. αx_i در حال حاضر در درخت مرحله $i-1$ ام توسط یک برگ نشان داده می شود.

نکته قابل توجه این است که اگر برای یک x_i مشخص در یک درخت معلوم، برای رشته $\alpha_1 x_i$ حالت ۱ و برای رشته $\alpha_2 x_i$ حالت 2 و برای رشته $\alpha_3 x_i$ حالت ۳ برقرار باشد آنگاه داریم:

$$|\alpha_3| < |\alpha_2| < |\alpha_1|$$

در حالت اول، تنها کاری که باید انجام شود انتخاب یک برچسب مناسب برای یک یال جدید و چسباندن آن یال به برگ درخت مرحله $i-1$ام است و در نتیجه این راس به یک راس داخلی تبدیل می شود.

تابع point (α) را به این صورت تعریف می کنیم که نشان دهنده عمیق ترین نقطه در درخت مرحله قبلی که در مرحله iام نیاز به تغییر پیدا کرده است.به این نقطه، **نقطه فعال**[1] می گوییم. قبل از اولین تکرار، نقطه فعال به صورت (Root, 0, *) است که در آن × می تواند نشان دهنده هر سمبل از زبان باشد. سایر مقادیر تابع Point را می توانیم با استفاده از نقطه فعال مرحله قبل و دنبال کردن لینک‌های آن نقطه به دست آورد.

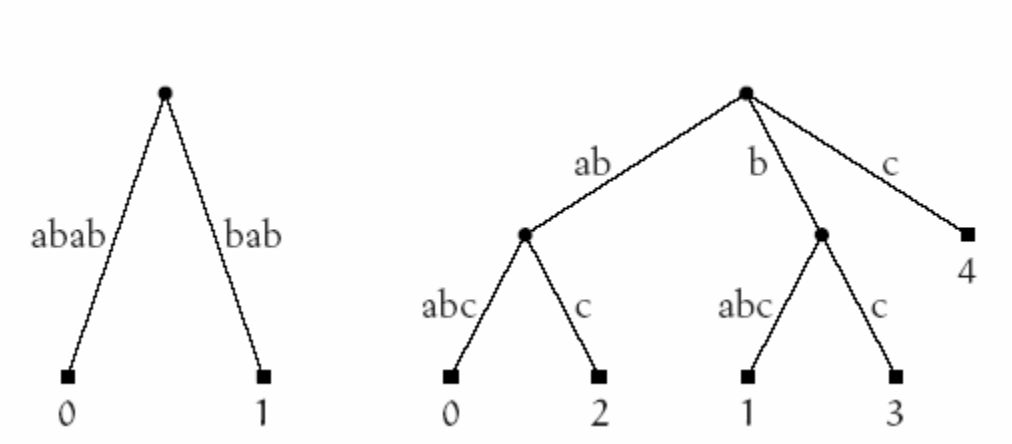
سرانجام در حالت سوم، تمام افزودن‌های[2] مورد نیاز انجام شده است. ما به عمیق ترین نقطه ای را که برای آن شرط ۳ برقرار است نقطه پایانی[3] می نامیم. نقطه فعال جدید برای تکرار بعدی، به راحتی با حرکت دادن نقطه فعال مرحله قبل به سمت پایین به دست می آید.

شکل ۴-۲، یک مثال برای درخت پسوندی قبل و بعد از مرحله ای که رشته مورد شاخص گذاری را از "abab" به "ababc" تغییر می دهد را نشان می دهد. قبل از این مرحله، نقطه فعال (Root, 2, 'a') است این نقطه که مربوط به "ab" است در درون یال منتهی به برگ با شماره صفر قرار دارد. در طول این مرحله، این یال شکسته می شود و نقاط ('a', 2, Root) و ('b', 1, Root) به صورت رئوس جدید ظاهر می- شوند و برگ‌هایی برای نمایش پسوندهای "abc" ، "bc" و "c" ایجاد می شود. نقطه فعال برای مرحله بعد به صورت (*, 0, Root) خواهد بود که نشان دهنده یک رشته خالی است.

---

[1] Active Point
[2] Insertion
[3] Endpoint

در این جا ما یک متغیر جدید به نام Front که نشان دهنده مکان راست ترین عنصر از رشته ورودی که تا کنون در درخت قرار گرفته است می پردازیم. بنابراین اگر درخت ما براساس رشته $x_0..x_{i-1}$ ساخته شده باشد آنگاه داریم Front= i.



شکل ۴-۲- درخت پسوندی مربوط به رشته "ababc" (سمت راست) و "abab" (سمت چپ)

نقطه درج[1] نقطه ای است که راس جدید در آن نقطه درج می شود. برای هر نقطه درج دو متغیر دیگر به نام‌های ins و proj نگه داری می شوند که ins نزدیکترین نقطه در بالای نقطه درج است و proj تعداد سمبل‌های بین نقطه درج و آن نقطه است. پس نقطه درج توسط یک سه تایی به صورت $(ins, proj, X_{front-proj})$ نشان داده می شود.

در شروع هر مرحله، نقطه درج برابر با نقطه فعال در نظر گرفته می شود.

در ادامه مراحل بالا را به صورت شبه کد در شکل ۴-۳ آورده ایم.

[1] Insertion Point

Canonize:

1 While $proj > 0$, repeat steps 2 to 5:
2 $r \leftarrow child(ins, x_{front-proj})$.
3 $d \leftarrow depth(r) - depth(ins)$.
4 If r is a leaf or $proj < d$, then stop and return r,
5 otherwise, decrease *proj* by d, and set $ins \leftarrow r$.
6 Return *nil*.

1 Set $v \leftarrow nil$, and loop through steps 2 to 16:
2 $r \leftarrow$ *Canonize*.
3 If $r = nil$ and $child(ins, x_{front}) \neq nil$, break out of loop to step 17.
4 If $r = nil$ and $child(ins, x_{front}) = nil$, set $u \leftarrow ins$.
5 If r is a leaf, $j \leftarrow spos(r) + depth(ins)$; otherwise $j \leftarrow pos(r)$
6 If $r \neq nil$ and $x_{j+proj} = x_{front}$, break out of loop to step 17.
7 If $r \neq nil$ and $x_{j+proj} \neq x_{front}$, execute steps 8 to 13:
8 Assign u an unused node.
9 $depth(u) \leftarrow depth(ins) + proj$.
10 $pos(u) \leftarrow front - proj$.
11 Delete edge (ins, r).
12 Create edges (ins, u) and (u, r).
13 If r is a leaf, $fsym(r) \leftarrow x_{j+proj}$; otherwise, $pos(r) \leftarrow j + proj$.
14 $s \leftarrow leaf(front - depth(u))$.
15 Create edge (u, s).
16 $suf(v) \leftarrow u$, $v \leftarrow u$, $ins \leftarrow suf(ins)$, and continue from step 2.
17 $suf(v) \leftarrow ins$.
18 $proj \leftarrow proj + 1$, $front \leftarrow front + 1$.

شکل ۴-۳-شبه کد تولید درخت پسوندی همزمان

۴-۳-۲- لغزاندن پنجره‌ها

در بخش قبل ما چگونگی تولید ونگه داری یک درخت پسوندی برای رشته به صورت $X_M=x_{tail}\ldots x_{front-1}$; که در آن tail و front متغیرهای صحیحی هستند که در طول اجرای

الگوریتم در شرط $0\leq x_{tail}\ldots x_{front-1}\leq M$ که در آن M بیشترین طول ممکن رشته ورودی است. برای سادگی ما در اینجا فرض می کنیم که هر یک از دو متغیر front و tail می توانند تا بی نهایت رشد کنند ولی همیشه در شرط بالا صادق هستند. از آن جا که در درخت هیچ ارجاعی به $0\ldots X_{tail-1}$ ندارد ، مکان مورد استفاده برای ذخیره و نمایش این قسمت‌ها می تواند آزاد شود و یا دوباره مورد استفاده قرار گیرد. راحت ترین راه برای این کار، پیدا کردن باقیمانده اندیس‌های در پیمانه[1] M و ذخیره سازی X_M در یک بافر که به صورت حلقوی نگه داری می شود است. در این روش به ازای هر i دربازه $[0..M)$، سمبل‌های x_{i+jM} خانه iام بافر را اشغال می- کند و در نتیجه ما فقط به حافظه ای به طول M برای نگه داری سمبل‌های رشته ورودی نیاز داریم.

هر تکرار از الگوریتم ساخت درخت پسوندی که در شکل ۴-۳ نشان داده شده است را می توان به عنوان یک تابع که در هربار فراخوانی به متغیر front یک واحد می افزاید و سایر پارامترها را برای این مقدار جدید تنظیم می کند در نظر گرفت. به این ترتیب، ما می توانیم از درخت پسوندی، حاصل به عنوان یک شاخص برای پنجره لغزنده با اندازه متغیر که حداکثر اندازه آن M است استفاده کنیم[4].

حذف چپ ترین پسوند از رشته شاخص گذاری شده، معادل حذف بزرگ ترین پسوند X_M (رشته حاصل از M سمبل آخر رشته ورودی) یعنی حذف رشته X_M از درخت است. از آن جا که این رشته، بلند ترین رشته موجود در درخت است پس باید حتما به وسیله یک برگ نمایش داده شود. همان طور که قبلا گفته شد پیدا کردن برگ نمایش دهنده یک رشته در ساختار درختی ما، زمان ثابتی

---

[1] Module

لازم دارد. بنابراین در نگاه اول، به روز در آوردن درخت کار ساده ای به نظر می رسد که شامل پیدا کردن برگ نمایش دهنده رشته X_M در درخت (v) و حذف آن و در نتیجه حذف پسوندی که از سمت چپ ترین نقطه رشته شروع می شود. سپس افزایش متغیر tail به اندازه یک واحد است.

کار ساده ای که در بالا بدان اشاره شد بزرگ ترین پسوند را از درخت حذف می کند وکاملا درست است ولی این کار (فقط حذف بزرگ ترین پسوند) برای نگه داری درست درخت پسوندی برای پنجره لغزنده کافی نیست. بلکه درخت ما پس از حذف برگ مورد نظر باید شرایط زیر را داشته باشد تا بتوانیم در مراحل بعدی از آن استفاده کنیم:

- خاصیت مسیر فشرده بودن درخت باید حفظ شود یعنی اگر حذف یک برگ راس پدر خود را دارای یک فرزند می کند باید راس پدر هم حذف شود.
- فقط باید بلند ترین رشته از درخت حذف شود و بقیه زیر رشته‌ها در درخت باقی بماند. یعنی اگر یال منتهی به راسی که می خواهیم حذف کنیم شامل رشته- ای به طول بیشتر از یک است باید یک راس جدید اضافه شود.
- متغیرهایی که نقطه درج را مشخص می کردند یعنی ins و proj باید پس از حذف برگ مورد نظر همچنان دارای مقادیر معتبر باشند.
- برچسب یال‌ها نباید خارج از پنجره قرار گیرد. یعنی به همان ترتیب که مقدار tail اضافه می شود ما باید اطمینان حاصل کنیم که همواره برای هر راس داخلی $pos(u) \geq tail$ باشد.

بخش بعدی به توضیح روش حذف و چگونگی حفظ این خواص می پردازد.

۴-۳-۲-۱- حفظ خاصیت مسیر فشرده بودن درخت

اگر فقط یک برگ از یک درخت مسیر فشرده حذف شود آنگاه تنها راسی که ممکن است در شرایط مسیر فشرده بودن درخت صدق نکند راس پدر برگ حذف شده است. فرض کنید u= Parent(v). اگر u حداقل دارای دو فرزند دیگر به جز v باشد با حذف v درخت ما باز هم مسیر فشرده است. اگر u دارای فقط یک فرزند دیگر مانند s به غیر از v باشد آنگاه باید راس‌های s و u در هم ادغام شوند به این ترتیب که یال‌های بین ((u,parent (u)) و (u, s) را حذف می کنیم و به جای آن‌ها یک یال بین (parent (u), s) ایجاد می کنیم. حال باید به صورت مناسب برچسب این یال را تولید کنیم. برای ایجاد برچسب مناسب برای یال جدید، کافی است که به ابتدای برچسب یال وارد به راس s برویم و برچسب یال جدید را بسازیم.

۴-۳-۲-۲- جلوگیری از حذف ناخواسته پسوندها

ما راس v را حذف کردیم، حال باید ثابت کنیم که هیچ رشته ای از پسوندهای $x_{tail}\ldots x_{front-1}$ از درخت حاصل حذف نشده اند. این حالت فقط هنگامی اتفاق می افتد که یک پسوند روی یال بین v و parent(v) قرار گرفته باشد. برای مثال در شکل (۴-۴) درخت مربوط به رشته ababcabab نشان داده شده است. حذف v در این درخت بزرگ ترین پسوند رشته را از بین می برد اما پسوند abab نیز چون در روی یال بین دو راس v و parent(v) است حذف می شود.

خوشبختانه یک روش ساده برای جلوگیری از این حالت وجود دارد اما قبل از بیان روش ابتدا به لم زیر توجه کنید:

لم : فرض کنید A و α دو رشته غیر تهی باشند که دارای شرایط زیر هستند:

- α بلند ترین زیر رشته A است که در شرط $A= \delta\alpha= \alpha\theta$ صدق می کند (θ و δ دو رشته غیر تهی باشند.)
- اگر $\alpha\beta$ یک زیر رشته از A باشد آنگاه β یک پیشوند از θ باشد.

آن گاه α بزرگ ترین پسوند رشته A است که به عنوان یک زیر رشته در جایی دیگر از A نیز ظاهر شده است.

فرض کنید حداقل یک پسوند غیر تهی با حذف راس v از درخت، دیگر قابل نمایش نباشد آن گاه این مجموعه یک عضو دارای طول بیشینه دارد که آن را α می نامیم.اگر $A= X_M$, دو شرط موجود در لم بالا برقرار است و در نتیجه α باید در مکان دیگری در درخت وجود داشته باشد.

این مساله باعث می شود که $point(\alpha)$ نقطه فعال در مرحله بعدی باشد. بنابراین ما می توانیم با کنترل کردن این که آیا نقطه فعال در روی یال وارد بر v قرار دارد بفهمیم که آیا رشته ای از درخت ما غیر از رشته مطلوب حذف خواهد شد یا نه. این کار را می توان با فراخوانی تابع Canonize با پارامتر v و کنترل مقدار بازگشتی آن انجام داد. اگر مقدار بازگشتی تابع برابر با v بود, به جای حذف v، آن را با یک برگ جدید که نمایش دهنده رشته α است جایگزین می کنیم. از آن جا که α بزرگ ترین پسوند X_M بود که از بین می رفت و ما با افزودن برگ جدید از گم شدن آن جلوگیری کردیم

پس تمام پسوندهای دیگر X_M که امکان از بین رفتن برای آن‌ها وجود داشت حال، خود پسوندی از α هستند و در نتیجه در درخت وجود دارند[5].

فصل پنجم

روش های فشرده سازی

با استفاده از دیکشنری ها[1]

در فصل قبل روش های فشرده سازی که توضیح داده شد، از یک مدل ایستا برای کد کردن[2] یک سمبل در زبان (داده ورودی) استفاده می کرد. آن روش ها با جایگزینی هرسمبل در داده ورودی با کد معادل آن سمبل سعی در کوچک تر کردن حجم فایل کد شده داشته اند و درآن ها، نرخ فشرده سازی ارتباط مستقیمی با مدل مورد استفاده برای کد کردن داده ها دارد بدین معنی که هر چه مدل دقیق تر باشد به فشرده سازی با نرخ بیشتری دست پیدا می کنیم. این مدل نه تنها باید به صورت دقیق احتمال وقوع هر سمبل را پیش بینی کند بلکه باید بتواند احتمالات انحراف از میانگین را نیز پیش بینی کند. هر چه انحراف از میانگین یا به طبع آن انحراف از معیار یک داده بیشتر باشد داده بیشتر فشرده می شود[6].

---

[1] Dictionary Based Compressions

[2] Encode

اما روش های مبتنی بر دیکشنری، از یک روش کاملا متفاوت برای فشرده سازی استفاده می کنند. این دسته از الگوریتم ها، هر سمبل را به رشته های کد شده با طول های متفاوت تبدیل نمی کنند بلکه در این روش ها، رشته- های با طول متفاوت از سمبل های داده ورودی، به صورت یک علامت[۱] کد می شوند. این علامت ها تشکیل یک شاخص برای دیکشنری حاصل را می دهند. اگر علامت هایی که به جای کلمات قرار می گیرند دارای طول کمتری باشند فشرده سازی صورت می گیرد.

۵-۱- ایده اصلی روش های فشرده سازی با استفاده از دیکشنری

با جستجو در زمینه این نوع فشرده سازی ها، به وضوح این موضوع آشکار خواهد شد که ریشه همه این گونه روش ها و برنامه ها، کارهای انجام شده توسط آقایان Jacob Ziv و Abraham Lempel بوده است. این دو محقق برای اولین بار در دهه ۱۹۷۰، این شاخه از فشرده سازی را به وجود آوردند.

تحقیقات روی فشرده سازی تا سال ۱۹۷۷، در زمینه های تحقیقات روی پراکندگی[۲] داده ها، تعداد تکرار کارکترها و کلمات، و سایر گونه های اطلاعات آماری متمرکز شده بود. و تا آن زمان توجه بسیار کمی به سایر زمینه ها، مانند ماشین های با تعداد حالات متناهی[۳] و مدل های زبان شناسی شده بود و محققان به دنبال روش های جدیدی برای بهبود روش های کد گذاری Huffman بودند[7].

این زمینه از تحقیقات با انتشار مقاله Ziv و Lempel در سال ۱۹۷۷ در IEEE، با عنوان

---

۱ Token
۲ Entropy
۳ Finite State Machine

"A Universal Algorithm for Sequential Data Compression"

شروع شد و سپس با انتشار

"Compression of Individual Sequences via Variable-Rate Coding"

مسیر جدیدی برای محققان گشوده شد.

در این دو مقاله، دو روش جدید فشرده سازی به نام های LZ77 و LZ78 معرفی شدند.LZ77 یک روش مبتنی بر پنجره های لغزنده است که در آن دیکشنری متشکل از یک مجموعه از عبارات با طول ثابت است که در یک پنجره وجود دارند. اندازه پنجره معمولا بین 2K و 16K بایت است و حداکثر طول عبارات بین 16 تا 64 بایت است.LZ78 یک روش کاملا متفاوت برای ساخت دیکشنری در پیش می گیرد. به جای استفاده از عبارات با طول ثابت، LZ78 در هر مرحله یک سمبل به پنجره اضافه می کند تا یک عنصر مانند عنصر مورد جستجو حاصل شود.

۵-۱-۱ یک مثال:

یک مثال خوب از این که روش های فشرده سازی مبتنی بر دیکشنری چگونه کار می کنند را می توان با یک دیکشنری استاندارد توضیح داد. برای مثال، فرض کنید ما برای کد کردن داده ها از فرهنگ واژگان عمید استفاده کردیم. به این ترتیب که به هر کلمه در فرهنگ لغت (دیکشنری) یک کد شامل دو عدد که اولی به شماره صفحه ای که آن لغت در آن وجود دارد و دومی به شماره لغتی که در آن صفحه است اشاره می کند (بدیهی است که این کد به طور یکتا کلمه مورد نظر را مشخص می کند). برای مثال کد های زیر به چهار کلمه اول ، اولین جمله در این پاراگراف اختصاص داده می شود:

1279/3 - 1052/7 - 548/5 - 51/2

این روش فشرده سازی، فقط شامل یک جستجوی ساده در دیکشنری است. دیکشنری ما شامل ۱۲۸۴ صفحه است و هر صفحه حدودا شامل ۳۵ کلمه است.

برای مشاهده کارآیی این روش، فرض کنیم که احتمال ظهور هر کلمه در داده ورودی برابر باشد. بنابراین برای کد کردن اعداد کوچکتر مساوی ۱۲۸۴ نیاز به ۱۱ بیت و برای کد کردن اعداد کوچکتر مساوی ۳۵ به ۶ بیت، یعنی در کل به ۱۷=۱۱+۶ بیت برای کد کردن یک کلمه نیاز داریم. اگر فرض کنیم برای نمایش متنی داده های فارسی از کارکترهای ASCII استفاده کرده باشیم برای نمایش چهار کلمه اول نیاز به ۱۴ بایت یعنی ۱۱۲=۱۴×۸ بیت داریم. در حالی که با روش کد کردن که در بالا بدان اشاره شد فقط به ۶۸=۱۷×۴ بیت نیاز داریم که در حدود نصف حجم داده ورودی است.

البته واضح است که مثال بالا فقط در حد مثال بوده است و به هیچ وجه نمی تواند مبنای یک فشرده ساز واقعی قرار گیرد زیرا این روش فقط برای فایل های با داده های فارسی، که کلمات در آن با کارکتر فاصله(Space)از هم جدا شده اند کار می کند. بدیهی است که می توان این روش را گسترش داد تا دامنه فایل های بیشتری را در بر گیرد.

۵-۲- دیکشنری ایستا[۱] در مقابل دیکشنری تطبیق پذیر[۲]

[۱] Static dictionary
[۲] Adaptive dictionary

همان طور که گفته شد روش های مبتنی بر دیکشنری، عبارات را با علامت ها جایگزین می کنند. اگر تعداد بیت های به کار رفته در علامت ها، کمتر از تعداد بیت های به کار رفته درعبارت باشد، فشرده سازی اتفاق می افتد. اما تا بدین جا، صحبتی از چگونگی ایجاد دیکشنری نکردیم.

گاهی بسته به خصوصیت داده ورودی بهتر است از یک دیکشنری از پیش تعیین شده استفاده کنیم. برای مثال اگر متنی که باید کد شود فقط شامل لیستی از نام شهرهای محل تولد کارمندان یک شرکت باشد می توانیم در ابتدا لیستی از تمامی شهرهای کشور مورد نظر تشکیل دهیم و سپس داده ورودی را با استفاده از دیکشنری ساخته شده کد کنیم. برای کد گشایی[۱] هم می توان از همان دیکشنری استفاده کرد. مثلا به طریقی دیکشنری را به فایل کد شده اضافه کنیم یا به دیکشنری را به صورت عمومی در اختیار همه افراد قرار دهیم.

این گونه دیکشنری ها، که از قبل و بدون توجه به داده ورودی ساخته شده اند و در طول فرآیند فشرده سازی تغییر نمی کنند دیکشنری های ایستا نامیده می شوند. یکی از مهمترین مزایای استفاده از دیکشنری های ایستا، این است که می توان دیکشنری را از قبل برای یک نوع داده معین تنظیم کنیم به این صورت که برای مثال با استفاده از روش کد گذاری Huffman به داده های پر کاربرد تر کد کوچکتری اختصاص دهیم. در مثال ما، چون متولدین تهران بسیار بیشتر از متولدین آستارا هستند می توان به شهر تهران یک کد با طول کمتر نسبت به کد شهر آستارا، اختصاص داد. درمقابل، روش های تطبیق پذیر نمی توانند این قابلیت را داشته باشند که از معایب این روش ها به شمار می رود.

---

[۳] Decoding

از مشکلاتی که روش های ایستا با آن مواجه هستند، مشکل انتقال دیکشنری به همراه فایل است. واضح است که ممکن است سر بار دیکشنری به حدی باشد که مانع فشرده شدن کل فایل خروجی شود. این اشکال مخصوصا در مورد فایل های کوچک، بروز پیدا می کند.البته اگر نیاز به اضافه کردن دیکشنری به فایل کد شده وجود نداشته باشد مثلا از یک دیکشنری عمومی و قابل دسترسی برای هر دو برنامه کد- کننده[1] و کد گشا[2] استفاده شده باشد، تقریبا همواره روش ایستا کاربرد بهتری دارد.

امروزه روش های فشرده سازی با استفاده از دیکشنری ایستا، تقریبا محدود به کاربرد های خاص شده است. بسیاری از روش های فشرده سازی، از دیکشنری های تطبیق- پذیر استفاده می کنند. در این نوع روش ها، به جای شروع کار با یک دیکشنری کامل، از یک خالی یا یک دیکشنری اولیه ثابت استفاده می شود. با شروع و ادامه به کار روند فشرده سازی الگوریتم، عبارات جدیدی را به دیکشنری اضافه می کند.

شبه کد زیر روند کار این نوع الگوریتم ها را نشان می دهد.

```
for ( ; ; ) {
  word = read_word( input_file );
  dictionary_index = look_up( word, dictionary );
  if ( dictionary_index < 0 ) {
    output( word, output_file );
    add_to_dictionary( word, dictionary );
  } else
    output( dictionary_index, output_file );
}
```

[1] Encoder
[2] Decoder

شبه کد الگوریتم فشرده ساز با دیکشنری تطبیق پذیر

در این شبه کد، قسمت های اصلی یک الگوریتم فشرده سازی با دیکشنری تطبیق پذیر نشان داده شده است:

۱. قسمتی که داده های ورودی را تجزیه[۱] کرده و آن را به بخش هایی که باید در دیکشنری جستجو شوند تقسیم می کند.

۲. قسمتی که بخش های ورودی را در دیکشنری جستجو می کند. در این بخش بسته الگوریتم ممکن است همتایی های جزیی هم بازگردانده شود .

۳. قسمتی که عبارت جدید را به دیکشنری اضافه می کند.

۴. قسمت کد کننده اندیس های دیکشنری و متن ساده (کلماتی که در دیکشنری پیدا نشده اند) به طوری که در هنگام کدگشایی قابل تشخیص باشند.

به طور مشابه، برنامه کد گشا هم شامل بخش هایی مشابه کد کننده است با این تفاوت که نیازی به بخش های ۱ و ۲ ندارد و به جای آن باید شامل بخش های زیر باشد:

۱. یک قسمت برای کدگشایی اندیس ها و متن ها

۲. اضافه کردن عبارت های جدید به دیکشنری

---

[۱] Parse

۳. تبدیل اندیس های دیکشنری به عبارات معادل

۴. چاپ کردن عبارات در خروجی

قابلیت انجام، این عملیات با هزینه زمانی نسبتا کم، استفاده از الگوریتم های فشرده سازی مبتنی بر دیکشنری را در ۱۰ سال اخیر پر کاربردتر کرده است [8] .

یک عنوان نمونه از کاربرد های عملی فشرده سازی های مبتنی بر دیکشنری می توان به روش مورد استفاده QIC-122[1] اشاره کرد.QIC-122 به منظور افزایش کارآیی در استفاده از نوارهای مغناطیسی به منظور ذخیره سازی اطلاعات مورد استفاده قرار گرفت.

QIC-122 یک نمونه خوب برای بیان چگونگی عملیات پنجره های لغزنده (که در فصل چهارم بدان اشاره شد) و الگوریتم های فشرده سازی وابسته به دیکشنری اشاره کرد. این روش بر پایه پنجره های لغزان در LZ77 پایه گذاری شده است. هم گام با خوانده شدن, عبارات به انتهای یک پنجره با حجم 2K (۲۰۴۸ کلمه) که دیکشنری ما را تشکیل می دهند، اضافه می شوند. برای کد کردن، برنامه کد کننده، کلمه ورودی را در دیکشنری جستجو می- کند. اگر کلمه در دیکشنری وجود داشته باشد یک علامت که به مکان عبارت در درخت و طول آن اشاره می کند ایجاد می کند و اگر وجود نداشته باشد سمبل بدون کد گذاری عبور می کند. خروجی الگوریتم کد کننده QIC-122 مخلوطی از علامات و سمبل ها است که به طور آزادانه می توانند پس از هم بیایند[8]. قبل از هر

[1] Quarter Inch Cartridge

علامت یا سمبل یک بیت که نوع داده بعدی را مشخص می کند قرار دارد و طبق قرار- داد به صورت زیر اضافه می شود:

Plain text	<1> <eight-bit-symbol>
Dictionary reference	<0> <Window-offset> <Phrase-length>

شکل ۵-۱- بیت های قرار دادی QIC-122

۵-۳- معرفی چند روش

۵-۳-۱- روش LZ77

روش LZ77 برای اولین بار در سال ۱۹۷۷ مطرح شد که به عنوان اولین روش فشرده سازی مبتنی بر دیکشنری از آن یاد می کنند.LZ77 ، از عبارات قبلا دیده شده در متن به عنوان دیکشنری استفاده می کند. این روش کلمات با طول متفاوت در متن ورودی را با یک ساختار اشاره گر با طول ثابت عوض می کند تا به فشرده سازی دست یابد. همان طور که گفته شد، فشرده سازی وقتی حاصل می شود که طول متوسط عبارات در ورودی از طول اشاره گر بیشتر باشد. در مدل LZ77، مقدار نرخ فشرده سازی به متوسط طول عبارات، حجم دیکشنری که باید کلمات قبلا دیده شده را نگه داری کند و همچنین آشفتگی ورودی بستگی دارد.

ایده اصلی به کار رفته در LZ77، استفاده از یک پنجره است که به دو قسمت تقسیم می شود. قسمت اول شامل قسمت هایی از داده ورودی است که تا کنون کد شده اند و معمولا حجم بزرگی را در بر دارد و قسمت دوم که معمولا کوچکتر از قسمت اول است و گاهی به آن بافر پیش

بینی هم می گویند، از عبارت بعدی در رودی که باید کد شود ولی هنوز کد نشده است تشکیل می شود.

حجم معمول پنجره درLZ77 معمولا چندین هزار کارکتر است. قسمت دوم پنجره معمولا بسیار کوچک در حد ده تا صد کارکتر است. الگوریتم تلاش می کند تا محتویات درون قسمت دوم پنجره را درون قسمت اول پنجره بیابد. یک مثال ساده از پنجره در روش LZ77 در شکل ۵-۲ آمده است.

به عنوان مثال در شکل ۵-۲، پنجره مورد استفاده دارای عرض ۶۴ کارکتر است که ۱۶ کارکتر آن به عنوان بافر پیش بینی استفاده شد است. همان طور که گفته شد LZ77 با علائم(Token) کار می کند. در این الگوریتم هر علامت از سه بخش تشکیل می شود:

- نقطه شروع عبارت در در پنجره قسمت اول متن اشاره می کند
- طول عبارت
- اولین سمبل در پنجره پیش بینی که بعد ازآن عبارت آمده است.

به عنوان مثال متن زیر را در نظر بگیرید:

پنجره متن	
قسمت پیش بینی	قسمت اول

i=0; i<MAX-1; i++)\r for (j= i+ 1; j (	j++)\r bc <MAX ;

شکل ۵-۲- مثالی از پنجره متن در LZ77

همان طور که می بینید اولین علامت بعدی در قسمت پیش بینی عبارت <MAX است که در مکان ۷ قرار دارد (البته می توان این مکان را بر حسب تعداد علامت های قبلا خوانده شده نیز بیان کرد) و اولین جایی که عبارت <MAX در قسمت اول اتفاق افتاده است مکان ۸ است و اولین کارکتر بعد از این عبارت " " است. پس علامت مربوط به این عبارت به صورت زیر خواهد بود:

Token= (7, 4, " ")

الگوریتم کد کننده پس از دریافت این علامت، ابتدا این علامت را در فایل خروجی قرار می دهد. سپس پنج کارکتر خوانده شده را به قسمت اول پنجره می افزاید و آن ها را از پنجره پیش بینی حذف می کند و به جای آن ها پنج کارکتر جدید به پنجره پیش بینی اضافه می- کند. عبارت بعدی طبق شرایط بالا "+j ;" است که بر نامه کد- کننده آن را به صورت ("+" ,3 ,35) کد می کند. واضح است که اگر نتوان عبارتی با طول دلخواه از قسمت پیش بینی پنجره متن را انتخاب کرد که با عبارتی از قسمت اول پنجره یکی باشد باید آن عبارت را به عبارت هایی با طول یک شکست و هر عبارت را به صورت یک علامت با طول صفر کد کرد مثلا ("b" ,0 ,0). به این ترتیب می توان مطمئن بود که هر داده ورودی قابل کد شدن است.

الگوریتم کدگشای LZ77، باید علامت ها را از ورودی خوانده و با توجه به عناصر اول و دوم آن عبارت را پیدا کند و سپس عنصر دوم علامت مربوطه را به آن اضافه کند.

یکی از موضوعات جالب در LZ77، توانایی کد کردن و کد گشایی این الگوریتم با استفاده از عباراتی که هنوز کد نشده اند برای کد کردن سایر عبارات است. فرض کنید در داده ورودی ما تعداد زیادی کارکتر "A" متوالی وجود داشته باشد. اولین "A" به صورت سه تایی (0, 0, "A") کد می شود. و علامت بعدی که الگوریتم کد کننده برای این عبارت تولید می کند می تواند به صورت زیر باشد:

(0, 9, "A")

پنجره اولیه:

قسمت پیش بینی پنجره	قسمت اول پنجره
AAAAAAAAAAA….	خالی

شکل ۵-۳- حالت خاصی از کد کردن LZ77

بعد از کد کردن (0, 0, "A")

قسمت پیش بینی پنجره	قسمت اول پنجره
AAAAAAAAAA….	A

شکل ۵-۴-حالت خاصی از کد کردن LZ77- ادامه

بعد از کد کردن (0, 9, "A")

قسمت اول پنجره	قسمت پیش بینی پنجره
AAAAAAAAAAA	...

شکل ۵-۵-حالت خاصی از کد کردن LZ77- ادامه

درهنگام کد گشایی، اگر یک الگوریتم مناسب پیاده- سازی شود می توان داده ورودی را بازیابی کرد. در زیرالگوریتم کد کننده و کد گشای LZ77 و در CD کد پیاده سازی شده این الگوریتم به زبان C++ ارائه شده است[9].

```
1. Find_Win_Len (w: string, Index1, Index2: integer; Len: Integer)
2.              Count= 0
3.               While (Len> 0)
4.                      if (W [Index1]= W [Index2])
5.                              Count = Count+ 1
6.                              Index1= Index1+ 1
7.                              Index2= Index2+ 1
8.                      else
9.                              return (Count);
10.             return (Count);

1.      Match_Pos= 0
2.      Match_Len= 0
3.      for (i= 0; i< Window_Size- Look_Ahead_Size; i++)
4.              Len= Find_Win_Len (Window, i, Look_Ahead,
Look_Ahead_Size)
5.              if Len< Match_Len
6.                      Match_Pos= i
7.                      Match_Len= Len;
8.      Encode (Match_Pos, Match_Len, Window [Look_Ahead+
Match_Len])
9.      UpdateWindow (Window, Match_Len+ 1, Window_Size-
Match_Len);
10.     Goto 1.
```

شبه کد، مربوط به LZ77

با این که LZ77، در نگاه اول بسیار ساده و کارا به نظر می رسد ولی در بحث پیاده سازی عملی با مشکلاتی مواجه است. از جمله این مشکلات، نیاز کد کننده به مقایسه رشته های موجود در قسمت پیش بینی پنجره با تمام رشته های موجود در پنجره است. اگر در LZ77 بخواهیم نرخ فشرده سازی را افزایش دهیم، تنها چاره برای این کار، افزایش اندازه پنجره است و در نتیجه تعداد مقایسه ها و کارایی زمانی برنامه پایین تر خواهد آمد. اما قسمت کد گشای LZ77، با این مشکل روبرو نیست زیرا فقط باید یک کپی از رشته ای معین بگیرد و نیازی به جستجو در رشته ها ندارد.

مشکل فوق را می توان با استفاده از پنجره های لغزنده که در فصل قبل بدان اشاره شد حل کرد.

۵-۳-۲- مشکل کد کننده در LZ77

علاوه بر مشکل کندی کد کننده در LZ77، این روش با مشکل میزان نرخ فشرده سازی نیز روبررو است. در مرحله کدگذاری، LZ77 به راحتی به نرخ فشرده سازی خوبی دست می یابد. حتی اگر در مرحله کد کردن عباراتی از داده ورودی که به صورت کد شده در می آیند کوتاه باشند الگوریتم LZ77، باز هم با نرخ خوبی می تواند فشرده- سازی انجام دهد.

مشکل در LZ77 هنگامی رخ می دهد که عبارت موجود در قسمت پیش بینی پنجره، در دیکشنری موجود نباشد. در این حالت باید برای کد کردن یک کارکتر از یک علامت با همان سه بخش استفاده کنیم. برای درک بهتر این هزینه، هزینه کد کردن یک کارکتر در حالتی که اندازه پنجره ۴۰۹۶ بایت است و قسمت پیش بینی پنجره از ۱۶ بایت تشکیل شده را در نظر بگیرید. برای کد کردن

این کارکتر به فرمت (a, b, c) که در بالا توضیح داده شد نیاز به بیست و چهار بیت فضا داریم (۱۲ بیت برای کد کردن مکان پنجره، ۴ بیت برای کد کردن طول کلمه و ۸ بیت برای کد کردن کارکتر بعدی). در حالتی که کارکتر در دیکشنری موجود نباشد سه تایی ما به صورت (0, 0, c) خواهد بود که در آن از بیست و چهار بیت، برای کد کردن فقط هشت بیت استفاده کردیم. این هزینه بسیار زیاد، باعث می شود که الگوریتم LZ77 برای فایل های کوچک خوب کار نکند. این مشکل در نسخه های بعدی LZ77 مانند LZSS بهبود داده شده است.

مشکل دیگر LZ77 این است که در این روش از یک پنجره با اندازه کوچک استفاده می شود و بنابراین به صورت مداوم با اضافه شدن حجم داده جدید به ناچار باید بخشی از داده قبلی را حذف کند و در نتیجه اطلاعات مربوط به دیکشنری در آن قسمت ها دیگر قابل استفاده نیست.

۵-۳-۳- روش LZ78

همانطور که اشاره شد روش LZ77 شامل نقایص و اشکالاتی بود. یکی از این اشکالات دور ریخته شدن قسمتی از داده های قبلا دیده شده است. مثلا در عبارت "کد کننده" در صفحات قبل به چندین بار مورد استفاده قرار گرفته است. حال فرض کنید بخواهیم این نوشته را با روش LZ77 کد کنیم. اگر اندازه پنجره را در حد معمول روش LZ77 در نظر بگیریم احتمالا هر بار که به عبارت "کد کننده" می رسیم عبارت "کد کننده" قبلی از پنجره به دور انداخته شده است. و در هر

بار باید یک علامت جدید برای آن ایجاد و ذخیره شود که این امر نرخ فشرده سازی را پایین می آورد.

استفاده از پنجره های لغزان در LZ77 باعث می شود که این روش به سمت جذب و ذخیره سازی داده های جدید در پنجره علاقه مند تر شود. همچنین در LZ77 برای طول رشته مورد جستجو در دیکشنری محدودیت طول داریم (طول رشته مورد جستجو نمی تواند از اندازه قسمت پیش بینی کننده پنجره بیشتر باشد). در حالی که بسیاری از رشته ها که در دیکشنری یافت شده اند می توانند با رشته ای با طول بیشتر جور شوند.

برای حل دو مشکل فوق، اولین ایده استفاده، از یک پنجره بزرگ تر، مثلا به جای 4K از یک پنجره با اندازه 64K استفاده کنیم و همچنین طول قسمت پیش بینی کننده پنجره را نیز افزایش دهیم و به جای ۱۶ بیت از ۱۰۲۴ بیت استفاده کنیم. ولی در عوض با انجام این کار و افزایش اندازه پنجره، به ۱۶ بیت به عنوان مولفه اول علامت ها و ۱۰ بیت به عنوان مولفه دوم علامت ها نیاز داریم و در نتیجه اندازه علامت از ۱۷ بیت به ۲۷ بیت افزایش پیدا می کند که این امر ممکن است خود موجب افزایش حجم فایل خروجی شود.

این افزایش در حدود ۵۰ درصدی نسبت شاخص به طول علامت ممکن است موجب اثرات منفی در نرخ فشرده سازی شود. با اعمال این تغییر، کد کردن عبارات ۲ کارکتری هم موجب فشرده سازی نمی شوند زیرا، زیر کلمه های ۲ کارکتری دارای ۱۶ بیت است در حالی که کد ما به ۲۷ بیت فضا نیاز دارد. ممکن است این ایده به ذهن برسد که می توان در این حالت برای کد کردن عبارات با طول کمتر از سه، به جای استفاده از کد، خود کارکتر ها و یا عبارات را قرار دهیم ولی این

کار خود موجب یک سربار جدید مثلا یک بیتی است که مشخص کند که علامت بعدی به صورت کد شده است یا به صورت متن عادی است.

یک مشکل دیگر که در LZ77، وجود دارد این است که با افزایش اندازه پنجره از 4K به 64K، میزان سرعت برنامه کد کننده تقریبا به ۱/۱۶ سرعت قبلی کاهش پیدا می کند زیرا هر عبارت باید با تمام عبارات موجود در پنجره چک شود. همچنین با افزایش اندازه پنجره پیش- بینی از 16 بیت به 1024 بیت، سرعت کد کننده چیزی در حدود ۱/۶۴ مقدار قبلی خواهد بود.

در نتیجه می توان LZ77 را در اعضای از دست دادن سرعت، به کارایی بهتری رساند ولی این کاهش سرعت به قدری زیاد است که نمی توان برای استفاده های عملی از آن استفاده کرد.به همین دلیل Lempel وZiv در سال ۱۹۷۸ در مقاله ای[۱] یک روش جدید برای فشرده سازی های مبتنی بر دیکشنری ارائه دادند که امروزه به روش LZ78 معروف است[10].

در LZ78، ایده استفاده از پنجره به طور کلی کنار گذاشته شد. در LZ77 دیکشنری بر اساس پنجره ای از عبارات قبلا دیده شده که در پنجره قرار دارند ساخته می شد. ولی در LZ78، دیکشنری می تواند لیست تمام عبارات دیده شده قبلا دیده شده در مرحله کد کردن باشد.

LZ78 در بسیاری جهات مشابه LZ77 است. درروش LZ77 کد کننده یک لیست از علامت ها که هر یک از سه بخش تشکیل شده بودند را به خروجی می فرستاد. LZ78 هم یک لیست از علامت ها که اساسا همان معنا را دارد در خروجی چاپ می کند. هر علامت در LZ78 شامل یک کد که نشان دهنده عبارتی است که قبلا توسط کد کننده دیده شده و اولین کارکتر بعد از

[۱] Compression of Individual Sequences via Variable-Rate Coding" in *IEEE Transactions on Information Theory* (September 1978).

عبارت است. بر خلاف LZ77، در LZ78 طول عبارت به کدگشا ارسال نمی شود زیرا کد گشا می- تواند طول عبارت را بازیابی کند.

برخلاف LZ77، LZ78 دارای یک پنجره پر از کلمات که دیکشنری آن را تشکیل می دهند نیست بلکه هر بار که یک کد در خروجی ظاهر می شود عبارت مربوط به آن کد به دیکشنری اضافه می شود و LZ78 می تواند تا انتهای کار (و نه مانند LZ77 که برای چند هزار کارکتر بعدی) از آن استفاده کند[11].

۵-۳-۳-۱- جزئیات الگوریتم LZ78

در الگوریتم LZ78 ، کد کننده و کدگشا کار خود را با یک دیکشنری تقریبا خالی شروع می کنند. طبق قرارداد در لحظه شروع در دیکشنری فقط رشته تهی (یک رشته به طول صفر) قرار دارد. کد کننده خواندن داده ورودی را با یک رشته خالی شروع می کند و هر بار از ورودی کارکتر بعدی را دریافت کرده و آن را به رشته فعلی اضافه می کند و سپس در دیکشنری به دنبال این کلمه جستجو می کند. اگر این کلمه در دیکشنری بود کارکتر بعدی را از ورودی می خواند. اگر با اضافه کردن کارکتر جدید رشته دیگر در دیکشنری وجود نداشته باشد کد کننده یک علامت و یک کارکتر در خروجی چاپ می کند که علامت شامل اندیسی از دیکشنری است که که رشته قبل از اضافه کردن آخرین کارکتر با آن یکی بود. سپس کلمه جدید که متشکل از یک کلمه موجود در دیکشنری به اضافه یک کارکتر است به دیکشنری اضافه می شود. در نتیجه در مراحل بعد اگر همان عبارت دوباره در داده ورودی ظاهر شود می توان با یک علامت، عبارت طولانی تری را بیان کرد.

به عنوان مثال برای توضیح چگونگی عملیات کد کننده LZ78، فرض کنید رشته زیر به عنوان ورودی به کد کننده LZ78 داده شده است.

DAD DADA DADDY DADO

در ابتدای اجرای کد کننده، دیکشنری فقط شامل یک رشته به طول صفر است و رشته ای که تا کنون از ورودی خوانده شده است نیز دارای طول صفر است پس تا کنون رشته در دیکشنری موجود است و کد کننده باید کارکتر بعدی (''D'') را از ورودی دریافت کند. با دریافت کارکتر بعدی رشته به 'D' تبدیل می شود و این رشته دیگر در دیکشنری موجود نیست بنابراین در خروجی یک علامت به صورت (0, ''D'') چاپ می شود و عبارت 'D' نیز در دیکشنری قرار می گیرد. و رشته کد کننده به رشته خالی تبدیل می شود و با خواندن کارکتر بعدی ''A'' علامت (0, ''A'') در خروجی چاپ می شود. با خواندن کارکتر ''D'' از ورودی رشته به 'D' تغییر پیدا می کند و چون این رشته در دیکشنری موجود است کارکتر بعدی به این رشته اضافه می شود و عبارت 'D ' تشکیل می شود که دیگر در دیکشنری وجود ندارد و در نتیجه علامت (1, '' '') در خروجی چاپ می شود و عبارت 'D ' به دیکشنری اضافه می شود. ادامه فرآیند کد شدن رشته در جدول زیر آمده است.

رشته کدشده	کارکتر خروجی	علامت خروجی	اندیس دیکشنری
‘’	–	–	0
‘D’	“D”	۰	1
‘A’	“A”	۰	2
‘D ’	“ ”	۱	3
‘DA’	“A”	۱	4
‘DA ’	“ ”	۴	5
‘DAD’	“D”	۴	6
‘DY’	“Y”	۱	7
‘ ’	“ ”	۰	8
‘DADO’	“O”	۶	9

…	…	…	

جدول ۵-۱- نحوه کد شدن عبارت DAD DADA DADDY DADO

برنامه کد گشا در LZ78، با دریافت هر علامت ، چون این علامت فقط به دیکشنری ساخته شده از کلمات تا قبل ازاین کلمه وابسته است می تواند عبارت ورودی را باز- سازی کند. به عنوان نمونه در زیر چگونگی روند بازسازی رشته ورودی از روی رشته خروجی را در مثال بالا نشان می دهیم.

رشته کدگشایی شده	علامت خروجی	دیکشنری	اندیس دیکشنری
-	-	‘’	۰
‘D’	(0, “D”)	‘D’	۱
‘A’	(0, “A”)	۰	۲
‘D ’	(1, “ ”)	۱	۳
‘DA’	(1, “A”)	۱	۴
‘DA ’	(4, “ ”)	۴	۵
‘DAD’	(4, “D”)	۴	۶
‘DY’	(1, “Y”)	۱	۷
‘ ’	(0, “ ”)	۰	۸

‘DADO’	(6, “O”)	۶	۹
...	...	...	...

جدول ۵-۲- چگونگی عملکرد کد گشا در LZ78

۵-۳-۳-۲- چگونگی پیاده LZ78

LZ78 مانند LZ77 می‌تواند به دلخواه اندازه دیکشنری خود را تنظیم کند. و همان مشکلاتی که در LZ77 در ازای بزرگ کردن دیکشنری با آن روبرو بودیم در LZ78 نیز وجود دارد یعنی با افزایش اندازه دیکشنری تعداد بیت هایی که برای کد کردن یک عبارت نیاز داریم بیشتر می شود و همچنین هزینه جستجو در دیکشنری نیز به طبع زیاد شدن حجم دیکشنری زیاد می شود.

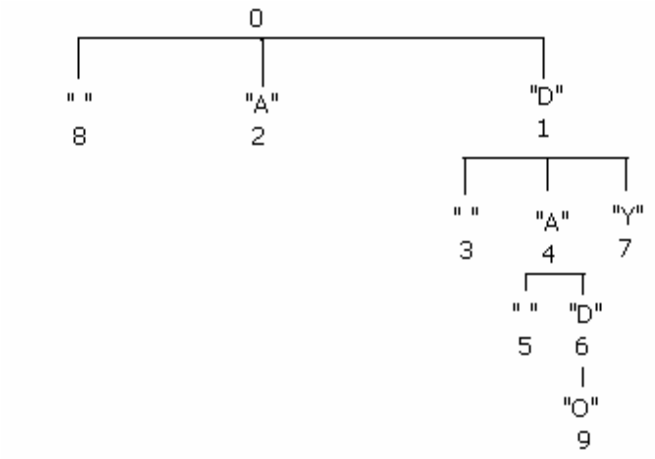
به صورت تئوری، فشرده سازی LZ78 باافزایش اندازه دیکشنری باید بهتر شود ولی این موضوع فقط هنگامی درست است که طول داده ورودی به سمت بی نهایت میل کند و در حقیقت با یک داده نامتناهی روبرو باشیم. در عمل مانند بسیاری از روش های دیگر، فایل های کوچک با افزایش اندازه دیکشنری (و در نتیجه افزایش حجم کد اختصاص داده شده به هر عبارت) از نرخ فشرده سازی کمتری برخوردار خواهند بود[11].

مشکل اصلی LZ78 چگونگی مدیریت و نگه داری دیکشنری است. اگر از شانزده بیت به عنوان کد عبارت استفاده کنیم دردیکشنری می توانیم ۶۵۵۳۶ عبارت مختلف داشته باشیم که شامل

رشته تهی نیز است. اما این عبارات می توانند دارای طول های مختلف باشند و عباراتی با احتمال وقوع بسیار ناچیز در دیکشنری موجود باشند.

معمولا دیکشنری LZ78 به صورت یک درخت نگه داری می- شود که ریشه درخت، رشته تهی (رشته به طول صفر) است. از هر راس از درخت نشان دهنده یک کلمه موجود در دیکشنری است. هنگامی که می خواهیم یک عبارت جدید به طول n به دیکشنری اضافه کنیم به شاخه ای که عبارت متشکل از n-1 کارکتر اول آن عبارت را پیدا می کنیم و سپس برای آن راس یک فرزند جدید ایجاد می کنیم که عبارت جدید را درآن قرار می دهیم. بدیهی است که هر راس می تواند حداکثر به اندازه تعداد کارکتر های موجود در داده ورودی فرزند داشته باشد.

به عنوان نمونه برای مثال قبلی که در آن می خواستیم عبارت DAD DADA DADDY DADO را کد کنیم درخت دیکشنری آن به صورت زیر خواهد بود.



درخت دیکشنری برای رشته
DAD DADA DADDY DADO

همان طور که در بالا بدان اشاره شد به صورت بالقوه، هر راس می تواند به اندازه کل کارکتر های موجود در الفبا فرزند داشته باشد که این موضوع کار نگه داری درخت را مشکل می کند. یعنی اگر این روش را روی یک فایل باینری اعمال کنیم می توانیم انتظار داشته باشیم که هر راس تا ۲۵۶ فرزند داشته باشد. البته برای نگه داری این نوع درخت ها می توان از لینک لیست استفاده کرد که در ازای استفاده از حافظه کمتر سرعت عملیات کمتر خواهد شد.

اگر دیکشنری را با ساختار درختی ذخیره کنیم آنگاه جستجو به دنبال عبارات بسیار ساده خواهد بود و فقط باید از راسی که در آن هستیم با یالی که با آخرین کارکتر خوانده شده مارک شده به پایین برویم. و اگر چنین یالی وجود نداشت باید باید عدم وجود چنین عبارتی در دیکشنری را گزارش دهیم.

در روش LZ78، برنامه کدگشا نیز باید درخت دیکشنری را همگام با کد کننده بسازد زیرا در این روش برخلاف LZ77 که کد هر عبارت مکان آن عبارت در دیکشنری بود ولی در این روش کد هر عبارت شماره یک راس در درخت است.

نکته دیگری که در مورد LZ78، در ابتدا ادعا کردیم این است که در LZ78 می توانیم از کل عبارات دیده شده استفاده کنیم ولی ما به کد هر عبارت تعداد محدودی بیت اختصاص دادیم و در نتیجه درخت دیکشنری ما می تواند تعداد محدودی راس و در نتیجه عبارت را در خود جای دهد. و این مساله به هر حال اجتناب ناپذیر است.

برای مقابله با مشکل پر شدن دیکشنری می توان از راه کارهای زیر استفاده کرد:

- ساده ترین راه حل این است که پس از پرشدن دیکشنری دیگر به دیکشنری چیزی اضافه نکنیم و از دیکشنری حاصل تا آخر کار استفاده کنیم. این روش در مورد فایل ها و داده هایی که از دو یا چند بخش مختلف تشکیل شده اند کارایی خوبی ندارد. زیرا ممکن است هرکدام از بخش ها، عبارات کاملا متفاوتی از هم داشته باشند مثلا یک فایل متنی که شامل یک متن انگلیسی و ترجمه آن است. اگر با این روش روی آن فایل کار کنیم احتمالا دیکشنری ما پس از خواندن متن انگلیسی پر می شود و هنگامی که به متن فارسی می- رسیم به علت این که در دیکشنری هیچ عبارت فارسی وجود ندارد مجبور به کد کردن عبارات فارسی به صورت کارکتر به کارکتر خواهیم بود.

راه دیگر که در فشرده ساز Unix که بر پایه LZ78 است به کار می رود استفاده از آمار نرخ فشرده سازی است. بدین معنی که در صورت پر شدن دیکشنری، تا هنگامی که نرخ فشرده سازی از حد ثابتی پایین تر نیامده است از همان دیکشنری قبلی استفاده شود و در صورت کمتر شدن نرخ از آن حد ثابت دیکشنری قبلی به کلی دور ریخته شود و دیکشنری جدیدی ایجاد شود.

فصل ششم
چگونه فایل های فشرده شده
را دوباره فشرده کنیم

در این بخش به بحث در مورد ادعای مطرح شده در بخش قبلی مبنی بر این که داده ها در فایل های تصادفی از توزیع تصادفی برخوردارند می پردازیم.

۶-۱- تعریف تصادفی بودن

یکی از تعاریف تئوری ، تصادفی بودن این است که داده های تولید شده توسط یک منبع اطلاعات که داده تصادفی تولید می کند را در مجموع نمی توان فشرده کرد. این موضوع با تئوری Shannon هم صادق است و با کمی فکر در آن می توان به درک شهودی آن دست یافت. اما این موضوع به این معنا نیست که نتوان هیچ یک از رشته های تولیدی این منبع را فشرده کرد بلکه به این معناست که اگر داده ای n بیت کوچک می شود داده یا داده های دیگری وجود دارند که حداقل n بیت بزرگ تر شوند.

یکی دیگر از تعاریف تئوری، تصادفی بودن این است که نتوان هیچ فرمول بسته و متناهی برای تولید رشته های منبع تصادفی ارائه کرد و گر نه در صورت ارائه یک فرمول بسته ومتناهی می توان یک تابع برای پیش بینی عنصر بعدی در رشته ارائه داد[13].

برای بررسی پیروی فایل های فشرده از توزیع های تصادفی نیاز به یک معیار برای سنجش این امر داریم. یکی از معیار ها که در اوایل کار به ذهن ما رسید شمارش تعداد صفرها و یک های داده ورودی (فایل های فشرده شده) بود. در صورتی که منبع اطلاعات از توزیع تصادفی پیروی کند باید تعداد این دو مقدار با هم برابر باشند. بدیهی است که این شرط یک شرط لازم برای تصادفی بودن فرآیند است ولی اصلا شرط کافی مناسبی نسیت. زیرا فرض کنید که منبع مورد نظر داده هایی به صورت $(0^n1^n)^*$ تولید کند یعنی دنباله از صفر ها که به دنبال آن ها دنباله ای از یک ها آمده است. که خلاف تعریف دوم از داده های تصادفی است.

به همین منظور معیار خود را به این صورت تغییر دادیم که تعداد دفعاتی که n صفر متوالی در فایل فشرده دیده می شود را به عنوان معیار در نظر گرفتیم. تعداد دفعات دیده شدن n صفر متوالی را با T_n نشان می دهیم. در این معیار، ما فقط هنگامی به T_n یک واحد اضافه می کنیم که دقیقا n صفر متوالی مشاهده شده باشد و قبل و بعد از این n صفر یک بیت با مقدار یک وجود داشته باشد. شبه کد زیر مقادیر T_n را محاسبه می کند.

```
1-    F= Handle of File, n= 0
2-    while not Eof (F) do
3-       c← ReadBit (F)
4-       if c= 0 then
5-          n= n+ 1
6-       else
7-          Tn= Tn+ 1
8-          n = 0
```

شبه کد محاسبه T_n

همان طور که از شبه کد بالا مشخص است برای افزوده شدن یک T_n مشخص باید یک دنباله به صورت 10^n1 اتفاق بیافتد. چون محاسبه احتمالات در ریاضی معمولا ساده تر است بنابراین ما P_n را به صورت زیر تعریف کردیم:

$$P_n=T_n/ F$$

که در آن F تعداد بیت های داده ورودی ماست. و در نتیجه P_n احتمال داده شدن n صفر متوالی در رشته ورودی است.

از نظر تئوری، این احتمال باید برابر با 2^{-n-2} باشد. در شکل ۱ مقادیر مورد انتظار و در شکل ۲ مقادیر بدست آمده روی فایل های فشرده شده را نمایش داده ایم.

N	1	2	3
P	0.125	0.063	0.031
N	4	5	n
P	0.016	0.008	2^{-n-2}

شکل1) مقادیر مورد انتظار

N	1	2	3
P	0.125	0.063	0.031
N	4	5	6
P	0.016	0.008	0.004

شکل 2) مقادیر بدست آمده

همان طور که مشاهده می کنید این داده های این دو جدول تا عمق ۵(۵ صفر متوالی)تا دقت ۳ رقم اعشار با هم برابر هستند. ولی هرچه عمق جستجو را بیشتر کنیم (برای n های بزرگ تر) این اختلاف زیاد می شود. این موضوع نشان می دهد که فرضیه ما مبنی بر پیروی فایل های فشرده شده از توزیع تصادفی تا عمق ۵ رشته درست است و طبق تئوری Shannon نمی توان با هیچ روش کمتر از سطح پنجم، این گونه داده ها را فشرده کرد. ولی احتمالا می- توان آنها را با روش های سطح بالاتر تئوری Shannon فشرده کرد.

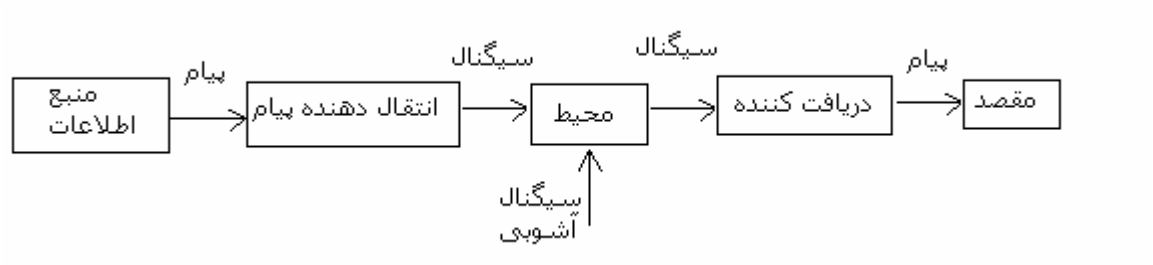
۶- ۲- تئوری Shannon1

در این بخش ما به توضیح تئوری Shannon که در سال ۱۹۴۸ توسط ریاضیدان معروف Claude E Shannon ارائه شد و به صورت ریاضی اثبات می کند که فشرده سازی ها مستقل از روش مورد استفاده و زبان مقصد و ... نمی توانند از مقدار معینی بهتر عمل کنند می پردازیم. این تئوری از این لحاظ مهم است که ممکن است این ابهام نزد خواننده این مقاله پیش بیاید که می توان

[1] Claude E Shannon

با استفاده از روش پیشنهادی ما (که در فصل بعدی به توضیح آن پرداخته- ایم) داده ها را تا بینهایت بار کوچک کرد و حجم داده های نهایی را به صفر رساند.

این تئوری در مورد سیستم ارتباطی[1] است که مدل کلی آن ها در زیر آمده است.



شکل ۶-۳- مدل سیستم ارتباطی در تئوری Shannon

همان طور که در شکل مشخص شده است این مدل از ۵ قسمت تقریبا مجزا تشکیل شده است:

- منبع اطلاعات که پیام یا دنباله ای از پیام ها را تولید و برای مقصد ارسال می کند.پیام یتواند به صورت های گوناگونی باشد:
  - به صورت دنباله ای کارکترهای یک زبان دلخواه مانند پیام های تلگراف
  - یک تابع یک متغیره $f(t)$ از زمان مانند رادیو و تلفن
  - یک تابع زمان و چند متغیر دیگر مانند $f(x, y, t)$ در انتقال تصویرتلویزیون
  - چندین تابع مختلف زمان در سیستم های انتقال چند کاناله
  - چندین تابع مختلف از چندین متغیرکه در فرستنده های تلویزیونی

[1] Communication System

- ترکیبی از موارد بالا مانند تلویزیون

- انتقال دهنده پیام که کار تبدیل پیام به سیگنال های قابل انتقال در محیط را برعهده دارد. در سیستم تلفن این سیستم پیام های صوتی را به سیگنال های الکتریکی تبدیل می کند.
- محیط انتقال که کار انتقال سیگنال ها را برعهده دارد. همان طور که در شکل مشخص شده است ممکن است محیط در سیگنال ها آشوب[1] ایجاد کند. معمولا این آشوب ها در سیستم های کامپیوتری وجود ندارد.
- دریافت کننده پس از دریافت سیگنال، ابتدا سعی در حذف تغییرات احتمالی به سبب آشوب های محیط می کند و سپس با انجام دادن عکس عمل انتقال دهنده سیگنال ها را به پیام تبدیل می کند.
- و در آخر مقصد که وسیله یا شخصی است که پیام برای آن ارسال شده است.

از آن جایی که در میان بخش های سیستم های انتقال اطلاعات توضیح داده شده در بالا، فقط منبع اطلاعات است (که شامل کد کننده ها هستند) که در سیستم های کامپیوتری قابل تغییر است و بقیه جزو سیستم کامپیوتر محسوب می شوند ما در این قسمت به بیان تئوری Shannon در مورد منابع جریان می پردازیم:

در مورد منابع، موضوعاتی که می توانند برای ما جالب باشند چگونگی توصیف ریاضی پیام ومقدار اطلاعات تولیدی برحسب بیت بر ثانیه است. طبق تئوری Shannon، مهمترین عامل برای دست یابی به یک کد گذاری خوب داشتن اطلاعات آماری از احتمال وقوع سمبل های زبان مورد

[1] Noise

استفاده است. Shannon اظهار می دارد که در هر پیامی سمبل ها با یک پیشامد معینی ظاهر می شوند برای مثال در زبان انگلیسی حرف E بیشتراز حرف Q و دنباله TH بیشتر از XP کاربرد دارد. وجود این اطلاعات منبع را قادر می‌سازد تا با استفاده از یک مدل کد گذاری مناسب مثلا با انتصاب کد کوچکتر به E نسبت Q یا .... حجم اطلاعات ارسالی را کمتر کند. این ایده در سیستم کد گذاری ر تلگراف(مورس[1]) نیز به کار رفته است. در سیستم مورس حتی برای برخی از پیام های پر کاربرد هم سمبل هایی پیش بینی شده است.

مدل آماری برای زبان را می توان به چند دسته تقسیم کرد:

- **مدل سطح صفر**:

در این مدل احتمال وقوع سمبل های زبان به صورت مستقل از هم و برابر با هم در نظر گرفته می شود. یعنی برای سمبل E و سمبل Q احتمال وقوع برابردر نظرگرفته می شود.

- **مدل سطح یک**:

در این مدل احتمال وقوع سمبل های زبان به صورت مستقل از هم در نظر گرفته می شود ولی احتمال وقوع هر یک متناسب با کاربرد آن سمبل است. یعنی برای سمبل E احتمالی برابر ۰,۱۲ و سمبل W احتمالی برابر ۰,۰۲ در نظرگرفته می شود.

[1] Morse

- **مدل سطح دو**:

دراین مدل نه تنها احتمال وقوع سمبل ها برابرنیستند بلکه احتمال مجاورت دوسمبل نیزبرابر نخواهد بود و بسته به سمبل قبلی انتخاب شده تعیین می شود. دراین مدل به احتمال وقوع سمبل iام بعد از سمبل jام نیاز داریم که آن را با $P_j(i)$ بیان می کنیم.

- **مدل های سطح بالاتر**:

دراین مدل احتمال وقوع هر سمبل ها بر اساس n-1 سمبل قبلی خود بدست می آید.

همچنین می توان این مدل های آماری را برای عبارات(به جای سمبل ها) محاسبه و مورد استفاده قرار داد[14].

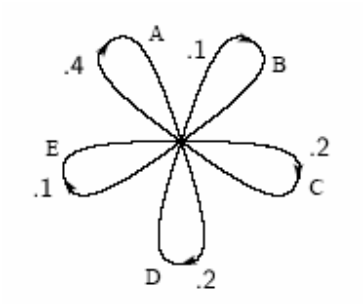
۶-۳- نمایش گرافیکی فرآیندهای مارکف[۱]

سیستم های فیزیکی یا مدل های ریاضی که بتوانند براساس احتمالات داده شده یک دنباله ازسمبل ها راتولید کنند فرآیندهای اتفاقی[۲] نامیده می شوند. در ریاضیات، فرآیندهای اتفاقی ازاین نوع را فرآیندهای گسسته مارکفی[۳] می نامند. حالت کلی این نوع فرآیند ها به صورت زیر است:

۱ Markoff process
۲ Stochastic
۳ Discrete Markoff Process

هر فرآیند از تعداد متناهی حالت [۱] $(S_1,S_2...S_n)$ تشکیل شده است. وازهرحالت با یک احتمال مشخص به حالت دیگرانتقال پیدا می کنیم. برای تبدیل یک مدل مارکف به یک منبع اطلاعات کافی است درازای عبورازهر حالت به حالت بعدی یک سمبل تولید می شود.

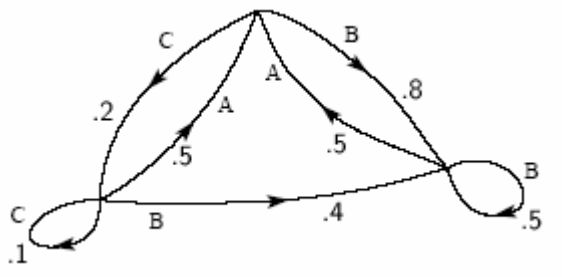
شکل زیرنشان دهنده فرآیند مارکف معادل با مدل سطح یک که در بالا معرفی شد است.



شکل ۶-۴- مدل سطح اول زبان انگلیسی روی سمبل های A,B,C,D,E

در این مدل مارکف فقط یک حالت برای بیان فرآیند کافی است.با هر انتقال از حالت اولیه به حالت ثانویه بسته به یال انتخاب شده برای انتقال سمبل مناسب به دنباله اضافه می شود.

شکل زیر نشان دهنده فرآیند مارکف معادل با مدل سطح دوم که در بالا معرفی شد است.



شکل ۶-۵- مدل سطح دوم زبان انگلیسی روی سمبل های A,B,C

---

[۴] State

در این فرآیند، سه حالت به ازای آخرین سمبل تولید شده وجود دارد و انتقال ازهرحالت به حالت بعدی متناسب با احتمال وقوع سمبل بعدی، پس ازآن سمبل است.

برای مدل سطح سوم با n سمبل، فرآیند مارکف مورد نظرباید شامل n^2 حالت باشد که هرحالت نشان دهنده دو سمبل آخر تولید شده هستند[15].

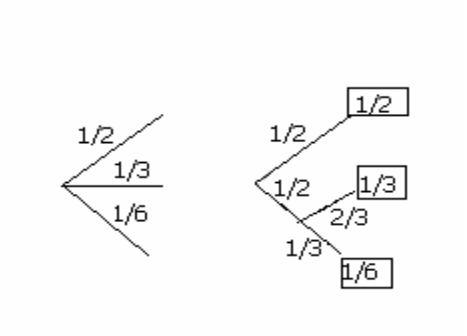
۶-۴- آشفتگی[۱] منبع اطلاعات

فرض کنید یک مجموعه از پیش آمدها با احتمالات $P_1,P_2,..,P_n$ وجود دارد. اگر تابع $H\ (P_1,P_2,..,P_n)$ را به این صورت تعریف کنیم که احتمال وقوع یک دنباله از پیشآمد ها چقدر است آنگاه H باید خواص زیر را داشته باشد:

- H باید نسبت به P_iها پیوسته باشد.
- اگر همه P_iها برابر با $1/n$ باشند H باید تابعی یکنواخت و صعودی برحسب n باشد که نسبت به P_iها متقارن است.
- اگر یک انتخاب به دو انتخاب متوالی تقسیم شد H باید برابربا میانگین وزن دار H های جدید حاصل باشد. به عنوان مثال شکل ۶-۶ را مشاهده کنید.

---

۱ Entropy



شکل ۶-۶

شکل سمت چپ، بیانگر $H(1/2, 1/3, 1/6)$ است و شکل سمت راست ترکیب دو تابع $H(1/2,1/2)$ و $H(1/3,2/3)$ است.

$$H(1/2,1/3,1/6)= H(1/2,1/2)+1/2\ H(1/3,2/3)$$

ضریب ۱/۲ به این علت ظاهر شده است که در ۱/۲ موارد حالت انتخاب بین دو فرآیند با احنمالات ۱/۳ و ۱/۶ پیش می آید.

ثابت شده است که تنها توابعی که این خصوصیات را دارند H هایی به صورت :

$$H= -K\sum P_i \log P_i$$

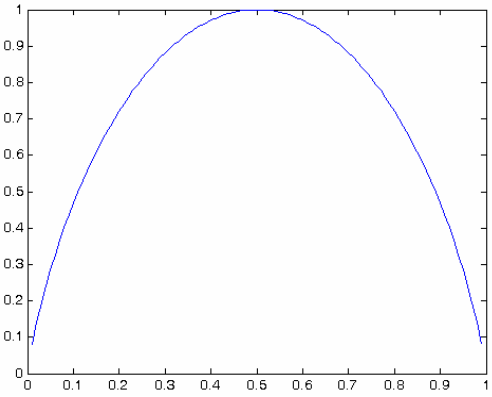
هستند کهK در آن ها عدد ثابت مثبتی است. معمولا K را برابر یک اختیار می کنند.

در حالت خاص اگر $n=2$ باشد می توان تابع H را به صورت زیرتعریف کرد:

$$q= 1-p$$

$$H (p, q) =H (p, 1-p) = - (p\log p+ (1-p) \log (1-p))$$

نمودارآشفتگی برای حالت بالا به صورت زیر خواهد بود.



شکل ۶-۷- نمودار آشفتگی به ازای دو متغیر

توابع H دارای خصوصیات خاصی هستند که دراینجا به چند نمونه ازآنها اشاره می کنیم:

- $H= 0$ اگر و فقط اگر همه P_i ها به جز یکی صفر باشند. یعنی فقط هنگامی که از خروجی فرآیند مطمئن هستیم مقدار H برابر صفر می شود.
- مقدار بیشینه H برابر است با $\log n$ هنگامی که همه P_iها برابر با $1/n$ باشند پیش می آید.
- هر گونه تغییر در جهت برابر نمودن احتمالات P_iها مقدار H را افزایش می دهد.
- هر گونه تغییردرجهت ایجاد اختلاف بین احتمالات P_iها مقدار H را کاهش می دهد.

در این جا می خواهیم به توضیح فرمول محاسبه آشفتگی برای هر یک از مدل های Shannon ارائه دهیم:

- **مدل سطح صفر**

در این مدل همان طور که اشاره شد احتمال وقوع هر سمبل برابر با بقیه سمبل ها در نظر گرفته می شود. و چون مجموع احتمالات باید برابربا ۱ باشد پس احتمال هر سمبل برابر با $1/n$ است که در آن n تعداد سمبل های زبان است. یعنی مقدار آشفتگی در این حالت برابراست با:

$$\sum 1/n * \lg (1/n) = -\sum \log (n)/n$$

- **مدل سطح یک**

در این مدل احتمال وقوع هر سمبل مستقل از سمبل قبلی در نظر گرفته می شودو در صورتی که n سمبل داشته باشیم دنباله ای از احتمالات P_i به ازای هر سمبل وجود خواهد داشت. و مقدار آشفتگی برابر خواهد بود با:

$$-\sum P_i \lg P_i$$

- **مدل سطح دو**

در این مدل احتمال وقوع هر سمبل وابسته به سمبل قبلی در نظر گرفته می شودو در صورتی که n سمبل داشته باشیم دنباله ای از احتمالات P_i به ازای هر سمبل وجود خواهد داشت و به ازای هر دو سمبل دنباله ای از احتمالات $P_{j|i}$ که نشان می دهد احتمال وقوع i پس از j را بیان می کند.مقدار آشفتگی برابر خواهد بود با:

$$-\sum P_i \sum P_{j|i} \lg P_{j|i}$$

- **مدل های سطح بالاتر:**

در این مدل احتمال وقوع هر سمبل وابسته به سمبل های قبلی در نظر گرفته می شود و در صورتی که n سمبل داشته باشیم دنباله ای از احتمالات B_i ها مورد نیاز است که احتمال وقوع تمام رشته هایی که از این n سمبل تسکیل شده است را در بر دارد. نکته قابل توجه در مورد مدل حالت کلی تئوری Shannon این ات که در حالت کلی ، این تئوری برای هیچ طول مشخصی از رشته ها احتمال تعیین نمی کند بلکه تمام زیر رشته های ورودی را در نظر میگیرد و به همین دلیل نمی توان این مدل را به صورت عملی پیاده سازی کرد. مقدار آشفتگی در این مدل برابر خواهد بود با:

$$\lim 1/n\sum P(B_n) \lg P(B_n)$$

از دیگر مواردی که Shannon درمورد فشرده سازی بدان اشاره کرده است این است که نمی توان روش فشرده سازی ارائه کرد که همه فایل های ورودی را فشرده کند و قابل بازیابی هم باشد. اثبات این موضوع ساده است و با استفاده از این موضوع که کد شده هیچ دو فایل ورودی نباید خروجی یکسانی داشته باشند بدست می آید. فرض کنید تمام فایل های باینری به حجم حداکثر n بیت را به عنوان ورودی به الگوریتم داده ایم تعداد این فایل ها برابر با $\sum 2^n = 2^{n+1}-1$ خواهد بود. از آنجا که می خواهیم همه فایل ها حال شده کوچک تر شده باشند پس باید حجم فایل های خروجی حداکثر n-1 باشد یعنی حداکثر می توانیم 2^n-1 فایل داشته باشیم و از آنجا که گفته شد خروجی هیچ دو فایل متفاوتی نباید یکسان باشند پس فایل هایی هستند که نه تنها فشرده نمی شوند بلکه حخمشان

افزایش پیدا می کند. اما درعمل، فایل هایی که یک روش روی آن ها به فشرده سازی دست پیدا می کند فایل هایی هستند که به طور واقعی وجود دارند.

حتی Shannon یک حد پایین برای فشرده سازی ارائه می دهد و اثبات می کند که هیچ روش بدون ازدست – دادن داده ای نمی تواند از این حد پایین بگذرد. این حد پایین که Shannon ادعا دارد که هیچ روشی نمی تواند از آن بهتر عمل کند مقدار آشفتگی داده های (یعنی H) است. بنابراین هرچه مقدار آشفتگی داده ورودی کمتر باشد داده خروجی حجم کمتری خواهد داشت. همان طور که دربالا گفته شد هرچه بین احتمالات اختلاف بیشتری باشد داده کد شده خروجی کوچکتر خواهد بود.

نکته قابل توجه این است که با این که روش Shannon ظاهرا فقط درمورد کدگذاری ها درست است و نمی تواند در مورد فشرده سازی های مبتنی بر دیکشنری چیزی بیان کند ولی Shannon ثابت کرده است که مستقل از روش فشرده سازی مورد استفاده یک حد پایین برای فشرده سازی داده ها ارائه می دهد.

موضوع دیگری که می توان از طریق تئوری Shannon توجیح کرد علت فشرده نشدن دوباره فایل هایی است که قبلا توسط یک الگوریتم فشرده سازی فشرده شده اند. علت این امراین است که تقریبا در خروجی همه الگوریتم های فشرده سازی احتمال P_0 و P_1 تقریبا با هم مساوی هستند ودرنتیجه اگربخواهیم روی این گونه فایل ها از الگوریتم های فشرده سازی دیگری استفاده کنیم به

علت این که هر فشرده سازی به یک سربار[۱] برای بازیابی داده های اولیه نیاز دارد حجم فایل خروجی نسبت به فایل ورودی بیشتر خواهد شد [14].

البته روش Shannon یک حد پایین برای فشرده سازی ارائه می دهد ولی

- این حد پایین فقط هنگامی درست است که به طریقی احتمال متوسط وقوع هر سمبل (که ازاینجا به بعد فقط صفر و یک خواهند بود) را داشته باشیم. ولی این روش Shannon در مورد فایل هایی که از احتمال وقوع سمبل ها در آن ها اطلاعی نداریم نمی تواند حدی ارائه دهد.

- این حد پایین روی طول میانگین داده های خروجی (کد شده) است. بدین معنا که همان طور که گفته شد در ازای فشرده شدن یک داده ورودی، مطمئنا یک یا چند داده ورودی با افزایش حجم روبرو می شوند. Shannon ادعا می کند که بهترین الگوریتم فشرده سازی اگر در بین همه داده های به طول n تولیدی یک منبع اطلاعاتی (که داده هایی هستند که با توجه به فرآیند مارکف معادل آن منبع به وجود آمده اند) الگوریتم را اجرا کنیم و ازاندازه خروجی ها را با توجه به احتمال هرداده ورودی میانگین وزن دار بگیریم این میانگین قطعا از مقدار آشفتگی منبع اطلاعات (H) بیشتر خواهد بود.

ما دراین پروژه سعی کرده ایم روشی برای فشرده سازی مجدد فایل های فشرده شده ارائه دهیم. همان طور که گفته شد علت این که نمی توان فایل های فشرده شده را دوباره فشرده کرد برابر بودن مقادیر P_1، P_0 و در نتیجه حداکثر بودن مقدار آشفتگی در فایل ورودی است. بدیهی است که اگر

---

۱۰ Overhead

بتوان با روشی برگشت پذیر اختلاف بین این احتمالات را زیاد کرد احتمالا می توان به فشرده سازی دست یافت. برای این منظور ما یک روش تبدیل که در بخش بعدی آن را توضیح خواهیم داد را ارائه دادیم و ادعا می کنیم که این روش می تواند فشرده سازی روی فایل های فشرده را به همراه داشته باشد.

فقط یک نکته باید در اینجا روشن شود که اگر بتوانیم فایل های فشرده را دوباره فشرده کنیم هیچ تضادی با تئوری Shannon ایجاد نمی شود. زیرا همان طور که گفته شد تئوری Shannon بیان می کند که هیچ فشرده سازی نمی تواند به صورت میانگین وزن دار به میزان فشرده سازی بیش از آشفتگی منبع اطلاعاتی دست یابد، ولی ممکن است داده هایی از فضای حالت ممکن را به حجمی بسیار کوچک تر از مقدار آشفتگی منبع فشرده کند. در روش ما هم دقیقا همین اتفاق می افتد یعنی فایل هایی که روش ما روی آن ها به فشرده سازی دست می یابند فایل هایی هستند که از توزیع یکنواخت برخوردار باشند که در بخش بعدی به توضیح آن خواهیم پرداخت. البته باید به این نکته توجه کرد که وقتی می گوییم توزیع یکنواخت به این معنی است که فایل ها تا مثلا مدل سطح n ام دارای P_iهای برابری است و بنابراین نمی توان با یک فشرده ساز که با مدل سطح n ام یا کمتر کار می کند به فشرده سازی دست یافت ولی این می توان با فشرده سازی های سطح $n+1$ام به فشرده سازی مطلوبی دست یافت.

فصل هفتم

یک مدل عمومی فشرده سازی

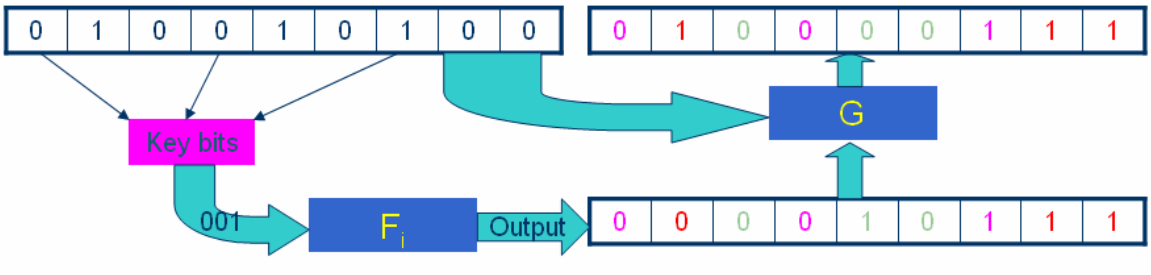
۷-۱- توصیف سیستم:

در این بخش به ارائه یک مدل عمومی از روش های فشرده سازی بدون در نظر گرفتن جزئیات می پردازیم.

همان طور که می دانید، در روش های فشرده سازی، به نوعی سعی در پیش بینی فایل ورودی با استفاده از قسمت هایی از فایل می شود که هر چه قسمت های مورد استفاده کمتر باشد فشرده سازی بهتری حاصل شده است. البته مفهوم پیش بینی در این جا با مفهوم آشنای پیش بینی متفاوت است. زیرا در فشرده سازی بدون از دست دادن اطلاعات، ما هیچ گونه از دست رفتن اطلاعات که به معنی پیش بینی اشتباه داده در فایل ورودی است را نمی –خواهیم. بنابراین ابتدا پیش بینی انجام می

شود و سپس اختلاف پیش بینی را با داده اصلی ذخیره می شود که در ادامه به تفصیل توضیح داده خواهد شد.

در شکل ۷-۱ نمای کلی یک چنین سیستم بیان شده است.



شکل ۷-۱- نمای کلی مدل عمومی فشرده سازی

همان طور که در شکل دیده می شود ابتدا داده ورودی به صورت رشته از بیت ها با طول مشخصی که بستگی به الگوریتم پیش بینی دارد وارد سیستم فشرده ساز می شود(رشته بیت های مشخص شده در سمت چپ در شکل ۱).

برای پیش بینی کل بیت های ورودی، از چند بیت از داده وردی نمونه برداری می شود که همان طور که گفته شد می تواند نسبت به تابع پیش بینی ما بیت های انتخابی متفاوت باشند. سیستم می تواند شامل یک یا چند تابع پیش بینی باشد که در صورت وجود چند تابع پیش بینی می توان، سیستم باید به نوعی (با گذاشتن علامت مشخص کننده ای) تابع پیش بینی کننده را مشخص کند تا درموقع بازیابی بتوان عمل بازیابی را به درستی انجام داد.

توابع پیش بینی کننده باید خواص زیر را دارا باشند:

- تعداد بیت هایی که برای پیش بینی داده ورودی نیاز دارند نسبت به داده مورد پیش بینی به نسبت کم باشد (در مثال بالا داده ورودی ۹ بیت و بیت های مورد نیاز ۳ بیت هستند).
- تابع پیش بینی با دقت خوبی براساس بیت های نمونه برداری شده، بتواند ورودی را حدس بزند. یعنی خروجی تابع شباهت زیادی نسبت به داده ورودی داشته باشد.
- تابع پیش بینی نباید از فرآیند های تصادفی استفاده کند یعنی خروجی این تابع به ازای یک ورودی ثابت باید همواره ثابت باشد.
- تابع پیش بینی باید معکوس پذیر باشد یا حداقل بتوان به وسیله یک تابع دیگر از روی خروجی تابع ورودی را باز سازی کند.زیرا در هنگام بازسازی فایل فشرده شده باید بتوان داده های کد شده را از حالت کد خارج کرد و فایل اولیه را بازسازی کرد.
- تابع پیش بینی کننده باید عمل پیش بینی را در زمان کوتاهی انجام دهد زیرا برای فشرده کردن یک فایل ورودی، بارها این تابع مورد استفاده قرار می گیرد. همچنین تابع باید طوری باشد که تابع معکوس آن (تابع کد گشا) هم بتواند به سرعت کدگشایی را انجام دهد.

ما در شکل توابع پیش بینی کننده را با F_i ها نشان داده ایم که با دریافت بیت های کلیدی (بیت هایی که برای پیش بینی از آن بیت ها استفاده می کند) ورودی را پیش بینی می کند (در شکل ۷-۱، ورودی پیش بینی شده به عنوان Output نشان داده شده است.). تابع پیش بینی درOutput در مکان بیت های کلیدی که در پیش بینی مورد استفاده قرار گرفته بود خود آن بیت ها را قرار می دهد (زیرا از وجود و مکان آن بیت ها را ورودی اطلاع دارد).

ممکن است این پیش بینی با داده ورودی به صورت صد در صد یکی نباشد بلکه فقط مشابه باشد معمولا چنین اتفاقی رخ می دهد. زیرا پیش بینی بر اساس بخش کوچکی از ورودی انجام می شود (مثلا ۳ بیت که ۸ حالت مختلف دارد) در حالی که ورودر دارای بیت های بیشتری نسبت به آن است (در این جا ۹ بیت یعنی ۵۱۲ حالت). برای رفع این مشکل و حفظ ورودی به صورت کامل در خروجی باید به نوعی اختلافات موجود در پیش بینی و داده ورودی را بیان و ذخیره کرد.

برای انجام این کار در مدل ما، از تابع G (که به عنوان تابع تفاضل از آن نام می بریم) استفاده می شود که دو رشته از بیت ها را به عنوان ورودی دریافت می کند و اختلاف آن دو را به عنوان خروجی سیستم فشرده –سازی در خروجی قرار می دهد (البته خروجی باید شامل شماره تابع پیش بینی هم باشد.). خصوصیات تابع تفاضل به صورت زیر است:

- محاسبه این تابع تفاضل باید سریع باشد. چون در طول فشرده کردن یک فایل بارها مورد استفاده قرار می گیرد.
- تابع G باید به تابعی معکوش پذیر یاشد.
- با داشتن G تابع تفاضل و خروجی تابع تفاضل و یکی از دو ورودی آن، باید بتوان داده دیگر را باز سازی کرد.

خروجی مقایسه کننده به اضافه تابع انتخاب شده در فشرده ساز به عنوان خروجی از فشرده ساز خارج می –شود. خروجی باید شامل بیت های کلیدی، اختلافات بین ورودی و داده پیش بینی شده باشد. در این مدل، اندازه خروجی با اندازه ورودی برابر است. و به جای بیت های کلیدی ،

درخروجی مقادیر واقعی آن بیت ها، وبه جای سایر بیت ها مقدار تفاوت محاسبه شده از طریق تابع تفاضل قرار می گیرد.

شاید در نگاه اول به نظر آید که سیستم فوق اصولا نمی تواند هیچ گونه فشرده سازی داشته باشد ولی اگر به این روش به دید یک تبدیل که داده ورودی را به داده ای میانی تبدیل می کند که عمل فشرده سازی روی آن بهتر انجام شود آنگاه با انتخاب یک تابع تفاضل مناسب مانند تابع XOR، می توان امیدوار بود که اگر تابع پیش بینی با دقت خوبی کار کند تمام بیت های خروجی غیر از بیت های کلیدی صفر شوند و بنابراین چون احتمال رخ دادن P_0 از احتمال رخ دادن P_1 بیشتر شده و با توجه به تقریبا برابر احتمال رخ دادن صفرها ویک ها می توان به فشرده سازی دست یافت.

داده خروجی کدگذاربه الگوریتم فشرده سازی مانند فشرده سازی ریاضی که درفصل بعدی آن را توضیح می- دهیم تا یک داده با حجم کوچک تر به دست آید.

۷-۲- سیستم کدگشایی

سیستم کدگشایی نیز به این صورت کار می کند که با دریافت خروجی سیستم فشرده ساز، آن را با استفاده از روش فشرده سازی ریاضی، داده خروجی حاصل از کدگذار را در اختیار سیستم کدگشا قرار می دهد. در گام بعدی با استفاده ازاطلاعات اضافی به همراه داده، تابع مورد استفاده در کدگذار شناسایی می شود و از روی آن بیت های کلیدی مشخص می شود. همان طور که در بالا گفته شد بیت های کلیدی بدون هیچ تغییری در خروجی ظاهر می شود و بنابراین کدگشا می تواند به بیت های کلیدی دست پیدا کند. حال کافی است برنامه کدگشا با اجرای همان تابع پیش کننده که در

کدگذاری مورد استفاده قرار گرفت مقدار پیش بینی که در کدگذار آن را با Output مشخص کرده بودیم تولید کند. حال کدگذار مقدار خروجی تابع تفاضل و یکی از دو مقدار ورودی آن تابع را به در اختیار دارد پس می تواند ورودی دیگر آن تابع را به دست آورد که این مقدار، همان مقدار ورودی برنامه کدگذار است.

۷-۳- آنالیز کارایی روش عمومی:

فرض کنید که تعداد توابع پیش بینی کننده برابر با ۱۲۸ باشد پس برای کد کردن تابع پیش بینی کننده نیاز به ۷ بیت داریم که باید به هر قسمت داده که به کدگذارمی رود اضافه شود. در این جا فرض می کنیم که از بهترین روش فشرده سازی استفاده کنیم پس نسبت فشرده سازی برابراست با:

نرخ فشرده سازی برابر است با اندازه فایل خروجی تقسیم بر اندازه فایل ورودی

اندازه فایل ورودی را F و احتمال رخ دادن ۰ را با P_0 و احتمال رخ دادن ۱ را با P_1 نشان می دهیم. اندازه بخش هایی که ورودی را به آن قسمت ها تقسیم می کنیم را با برابر با Input_Size در نظر بگیریم حجم فایل حاصل از کدگذار پس از فشرده سازی برابر با:

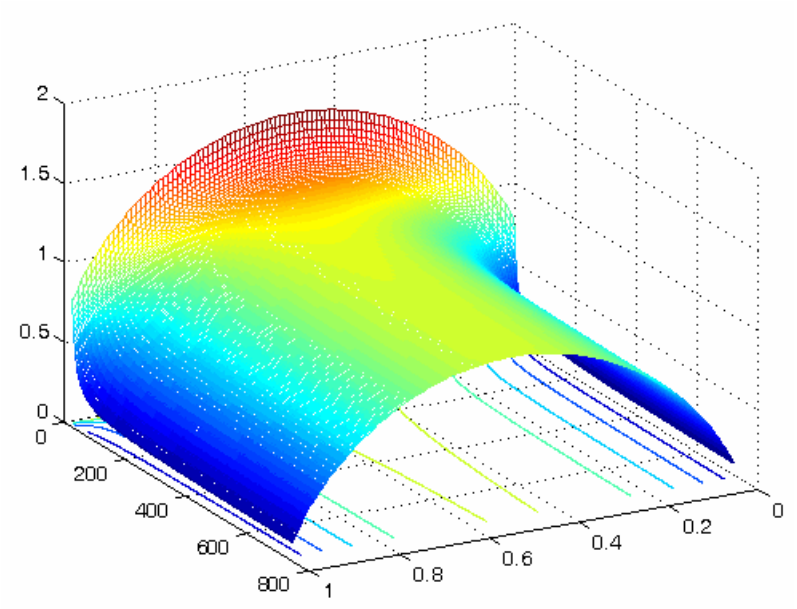
$$Z = \frac{F \times P_0 \times Lg(P_0) + F \times P_1 \times Lg(P_1) + 7 \times F \times Inp\_Size}{F}$$

$$Z = P_0 \times Lg(P_0) + P_1 \times Lg(P_1) + 7 \times Inp\_Size$$

$$P_1 = 1 - P_0$$

که آخرین عبارت یعنی F/ Input_Size *7 به علت این که F/ Input_Size بار تابع کدگذلر فراخوانی می- شود و هر بار ۷ بیت به عنوان سربار به خروجی اضافه می شود.

شکل ۲ نمودار مقدار Z براساس Input_Size و P0 است.



شکل ۷-۲- نمودار نسبت فشرده سازی براساس Input_Size و P0

بدیهی است که با کاهش Input_Size به علت افزایش سربار که در هر مرحله حجم فایل خروجی بیشتر می- شود البته این موضوع با این فرض که کارایی تابع پیش بینی کننده وابسته به Input_Size نیست صورت گرفته است (که البته این موضوع چنان هم درست نیست).

اگر نرخ فشرده سازی عددی بزرگ تر از یک باشد آنگاه به جای فشرده سازی در عمل افزایش حجم فایل اتفاق افتاده است و اگر این مقدار کوچک تر از یک باشد به فشرده سازی دست یافته ایم. با استفاده از روش اشاره شده در بالا، انتظار داریم که P_0 از حدود ۰,۵ جا به جا شده و

مقداری بزرگ تر از ۰,۵ را اختیار کند. در نتیجه با انتخاب یک مقدار مناسب برای Input_Size می توان به فشرده سازی دست یافت.

فصل هشتم

پیاده‌سازی نمونه از روش عمومی ارائه‌شده

۸-۱- توابع پیش‌بینی‌کننده

پیاده‌سازی ما از توابع پیش‌بینی‌کننده وابستگی زیادی به مفهوم قدیمی فرکتالها، ارتباط آنها با پدیده‌های تصادفی و یکی از روش‌های تولید آنها به نام L-System دارد. به همین دلیل، ابتدا به بررسی گرامرهای L-System و نحوه استفاده آنها در تولید فرکتال‌ها خواهیم پرداخت. سپس به بررسی نحوه استفاده ما از این گرامرها و تغییراتی که در تعریف این گرامرها داده‌ایم، خواهیم پرداخت.

۸-۱-۱- گرامرهای L-System[1]

گرامرهای L-System، نوع خاصی از گرامرهای مستقل از متن هستند که:

۱ مطالب این بخش از مرجع ۲۳ استفاده شده اند

- همه متغیرها در جملات آنها به صورت همزمان و با همدیگر گسترش می یابند (اشتقاق از چپ و یا راست در اینجا معنی ندارد).
- هیچ جمله انتهایی‌ای برای آنها متصور نمی‌باشد (تمام سطوح در درخت اشتقاق شامل حداقل یک متغیر می‌باشند).
- هر متغیر فقط یک قانون برای توسعه را شامل می‌شود (تا در هر مرحله از درخت اشتقاق فقط یک جمله موجود باشد).

این گرامرها برای تولید فرکتال‌ها توسعه یافته اند و در این پایان نامه نیز به دلیل ماهیت بازگشتی آنها به کار گرفته شده‌اند. به عنوان نمونه به گرامر زیر که یک گرامر L-System است، توجه کنید:

$$F \longrightarrow F + F - - F + F$$

این فرمول یکی از ساده‌ترین و عمومی‌ترین مثال‌ها در زمینه گرامرهای L-System می‌باشد. همانگونه که می‌بینید، این گرامر هیچ‌گاه به جمله پایانی منجر نخواهد گشت. این موضوع، به خاصیت فرکتالها باز می‌گردد که هیچ‌گاه به پایان نخواهند رسید.

حال فرض کنید که می‌خواهیم از روی گرامر بالا، به تولید فرکتال بپردازیم. جمله اولیه گرامر، جمله F است. با شروع از این جمله، به جملات زیر در هر مرحله خواهیم رسید:

مرحله اول F

مرحله دوم F+F--F+F

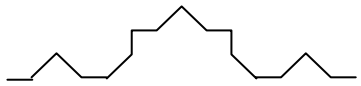
مرحله سوم F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F

برای ساختن شکل فرکتال‌ها نیز به این صورت عمل می کنیم که به جای هر کـدام از Fهـا، خطی به طول واحد می کشیم. به جای هر کدام از علائم مثبت ۶۰ در جه در خلاف جهـت حرکـت عقربه‌های ساعت و به ازای هر کدام از علائم منفی ۶۰ درجه در جهت حرکـت عقربـه‌هـای سـاعت می‌گردیم. نتیجه کار به صورت زیر خواهد بود:

مرحله اول

مرحله دوم

مرحله سوم



شکل ۸-۱- نمونه ای از فرکتال‌های توصیف‌شده با گرامرهای L-System

همانگونه که مشاهده می شود، با ادامه این فرآیند، به برفدانه معروف کخ که از معروف‌تـرین فرکتال‌ها است، خواهیم رسید.نکته جالب در مورد این گرامرها سادگی آن‌هـا و نیـز توانـایی‌شـان در توصیف شکل‌های بسیار پیچیده می باشد. این اشکال طبق تعاریف ریاضی دارای چاوس[۱] می‌باشـند. به این معنی که نقاطی در این اجسام وجود دارد که هیچ نقطه‌ای در همسایگی آنهـا قابـل پـیش‌بینـی نیست. این نقاط خصوصیتهای کاملا تصادفی را از خود بروز می‌دهند.

[۱] Chaos

۸-۱-۲- نحوه استفاده ما از گرامرهای L-System:

حال که با تعریف کلی گرامرهای L-System آشنا شدیم، باید بدانیم که ما چگونه از این گرامرها استفاده کرده‌ایم. بیشترین استفاده ما از این گرامرها، به خاصیت ذاتی آنها در استفاده از بازگشت بازمی‌گردد. همین خاصیت ذاتی است که باعث می‌شود که یک گرامر کوچک بتواند از سویی جملات بسیار بزرگی را ایجاد نماید و از سوی دیگر، خاصیت چاوسی خود را آشکار کند.

علاوه بر آنچه در بالا به عنوان محدودیت‌های گرامرهای L-System نسبت به گرامرهای مستقل از متن مطرح گشت، خصوصیت‌های زیر نیز به دلیل محدودیت‌های مفهومی‌ای که ما با آن روبرو بودیم، مطرح شدند:

- بعضی ازبخش‌های درخت ممکن است به دیگر سیبلینگ[۱]های خود در درخت وابسته باشند. این بخش‌ها نودهای پایانی هر سطح می‌باشند که جای تعریف اپراتورهایی مانند + و یا – در مثال برفدانه کخ را گرفته‌اند. نتایج این بخش مستقل از فایل ورودی و بر اساس نود[۲]های سیبلینگ این نود و عبارتی که در خود این نود جای گرفته است، محاسبه می‌گردند.
- تمام قوانین باید تعداد یکسانی از بخش‌ها را شامل شوند. این محدودیت نیز به دلیل وجود محدودیت قبلی پیش می‌آید.
- جمله آغازین این گرامرها، متغیر A در نظر گرفته می‌شود.

---

۱ Sibling
۲ Node

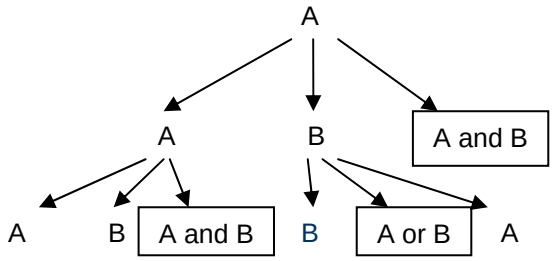
مثال زیر در استفاده از گرامرهای L-System، نحوه استفاده ما از این گرامرها را کاملا روشن می سازد. گرامر زیر را در نظر بگیرید:

$$A \longrightarrow A \sim B \sim (A \,\&\, B)$$

$$B \longrightarrow B \sim (A \,||\, B) \sim A$$

عبارت‌های داخل پرانتز در گرامرهای بالا، همان بخشهایی هستند که به نودهای سیبلینگ خود وابسته‌اند. همان‌گونه که در بالا دیده می شود، این نودها، بر اساس عبارتی که در آن‌ها جای گرفته است، و با داشتن مقادیری برای نودهای سیبلینگ آن‌ها قابل‌محاسبه‌اند.

درخت زیر نمونه‌ای از درخت اشتقاق این گونه از گرامرها می باشد که بر اساس گرامر نمونه بالا تولید شده و توسعه یافته است:



شکل ۸-۲- نمونه‌ای از درخت اشتقاق

بخش‌هایی از شکل بالا که در مستطیل جای گرفته اند، همان بخش‌هایی هستند که به عنوان نود پایانی محسوب گشته‌اند و توسعه داده نخواهند شد. باید دقت داشت که عبارت A and B در سطح یک بر اساس نودهای سیبلینگ خود در سطح یک (یعنی عبارات A و B در سطح یک)

محاسبه می گردند. در حالی که عبارت‌های A and B و A or B در سطح دو بـر اسـاس عبـارت-های سیبلینگ خود در سطح دو محاسبه می شوند.

استفاده از این نوع گرامرها دارای مزایا و معایب خاص خود است کـه در زیـر بـه برخـی از آنها اشاره کرده‌ایم:

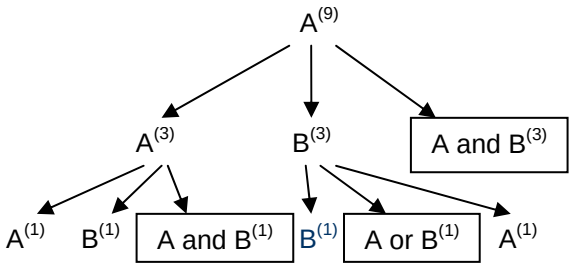
- با وجود آنکه بلادرنگ بودن، از خواص مثبت فشرده‌سازی‌ها بـه شـمار مـی آیـد، ولـی پیاده‌سازی بلادرنگ از این گرامرها در حالت اصلی آن‌ها، نمی تواند متصور باشـد. چـرا که با زیاد شدن سطوح در این گرامرها، سطوح بالای درخت حجم زیادی پیدا خواهنـد کرد و انجام دادن محاسبات بلادرنگ بر روی آنها امکان‌پذیر نخواهد بود. یـک راه حـل ساده برای این مساله، محدود کردن سطوح اشتقاق در درخت است کـه در پیـاده‌سـازی انجام شده نیز در نظر گرفته شده است.
- گرامرهای با حجم کوچک قادر به توصیف فایل‌های بسـیار بـزرگ خواهنـد بـود (ایـن مقوله در روش‌های فشرده سازی مبتنی بر گرامر قبلی امکان پذیر نبود).
- با توجه به خصوصیت چاوسی این گرامرها، می توان نتـایج شـبه تصـادفی خـوبی را از آن‌ها انتظار داشت.

۸-۱-۳- تولید درخت اشتقاق:

همان‌گونه که در بخش قبل بحث شد، تولید درخت اشتقاق برای گرامرهای مورد استفاده در سیستم ما با استفاده از قوانین زیر ممکن می‌گردد:

- نماد آغازین گرامر متغیر A در نظر گرفته می‌شود.
- تمام متغیرهایی که در یک سطح قرار دارند به یکباره و در یک گام گسترش داده می‌شوند.
- طول هر نود برگ (چه مستقل و چه غیر مستقل) از گرامر برابر یک است.
- طول هر نود غیر برگ و مستقل از گرامر، برابر جمع طول فرزندان آن می‌باشد.
- طول هر نود غیر برگ و غیر مستقل از گرامر، برابر طول نودهای سیبلینگ همان نود است.

شکل زیر، نمایان‌گر همان درخت اشتقاق قبلی است. با این تفاوت که اعداد داخل پرانتز در این شکل، نشان دهنده طول هر بخش می‌باشد.



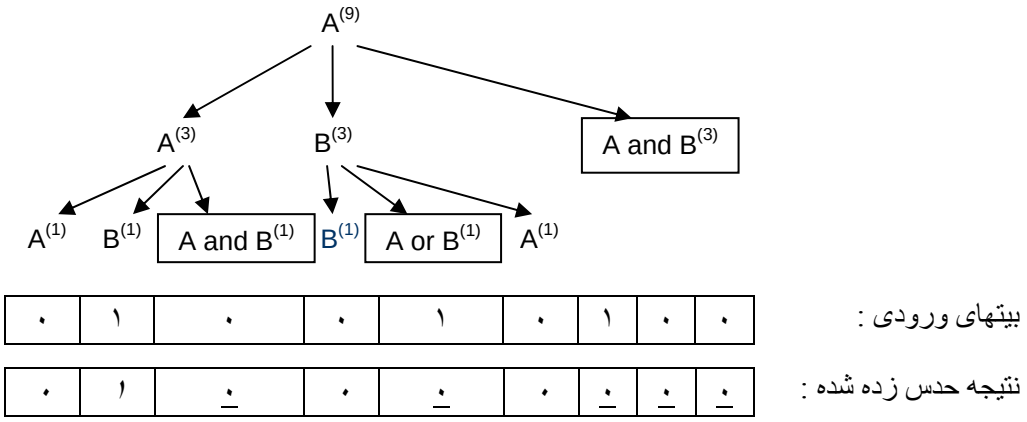
شکل ۸-۳- طول هر بخش در گرامرهای L-System

۸-۱-۴- نگاشت از ورودی به نودهای درخت:

همان‌گونه که در بخشهای قبل ذکر شد، ورودی تابع پیش‌بینی کننده، فایل ورودی و خروجی آن، توصیفی از فایل ورودی بر اساس بیت‌های کلیدی این فایل می باشد. برای اینکه این عمل در پیاده‌سازی نمونه ما رخ دهد، فایل ورودی را به صورت زیر به درخت اشتقاق نگاشت خواهیم نمود:

- نودهای مستقل و برگ، مقدار خود را مستقیما از فایل ورودی دریافت خواهند کرد.
- مقدار متغیرهای مستقل و غیر برگ، با به هم چسباندن مقادیر فرزندانشان محاسبه می‌گردد.
- نودهای غیرمستقل، مقدار خود را بر اساس جایگزینی متغیرها با مقدارشان و محاسبه عبارت مربوطه به دست می آورند.

در شکل زیر، آنچه در بالا بیان گردید، برای درخت نمونه ما به تصویر کشیده‌شده‌است. بیت‌هایی که در بخش "نتیجه حدس زده شده" با خط زیر مشخص گردیده‌اند، بیتهایی می باشند که بر اساس بیت‌های کلیدی مشخص گردیده‌اند و به فایل ورودی مرتبط نیستند. بیت‌های دیگر، که با فونت ایتالیک مشخص گشته‌اند، مستقیما از فایل ورودی کپی شده و بیت‌های کلیدی به شمار می‌آیند.



شکل ۸-۴- نمونه ای از تولید رشته حدس زده شده

۸-۲- تابع محاسبه‌گر تفاوت:

همانگونه که قبلا گفته شد، ما قادر نیستیم که تمام رشته‌ها را به صورت کاملا دقیق پیش‌بینی کنیم. از اینرو، نیاز به تابعی داریم که تفاوت رشته پیش‌بینی شده و رشته واقعی را به صورتی درخور بیان نماید. این تابع به ما اجازه خواهد داد تا به هنگام عمل باز نمودن فایل فشرده‌شده، قادر به بازیافت فایل اصلی (با توجه به تفاوت‌های آن با نمونه پیش‌بینی شده) باشیم.

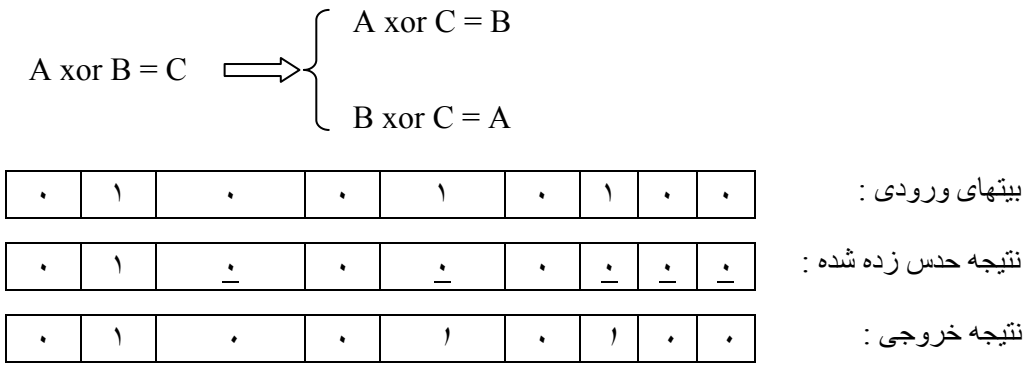
این تابع باید چند خصوصیت اصلی را ارضا کند:

- قابل بازگشت باشد.
- سربار زیادی (از نظر حجم داده) نداشته باشد.

علاوه بر خصوصیات بالا که برای این تابع ضروری هستند، بهتر است که این تابع از خصوصیات زیر نیز پیروی کند:

- سربار محاسباتی کمی داشته باشد.
- ساختار ساده و قابل فهمی را ارائه کند.

تابعی که ما برای این کار در نظر گرفتیم، تابع Xor می باشد. این تابع، اولا دوخاصیت ضروری را حفظ می‌کند و ثانیا دو خاصیت غیر ضروری را هم ارضا می‌نماید.



شکل ۸-۵- تولید خروجی با استفاده از تابع محاسبه‌گر تفاوت

همانگونه که در شکل بالا دیده می شود، استفاده از عملگر Xor تضمین می کند که با داشتن یکی از ورودی‌ها و خروجی حاصله، قادر به دانستن ورودی دیگر باشیم. با توجه به شکل بالا، درمی‌یابیم که عملگر Xor فقط بر روی بیت‌های حدس‌زده‌شده خروجی و بیت‌های متناظرشان در ورودی، عمل کرده است و با بیت‌های کلیدی کاری نداشته است. بنابراین در هنگام بازگشت، با داشتن بیت‌های کلیدی قادر خواهیم بود تا رشته حدس‌زده‌شده را دوباره بسازیم و در نتیجه، با استفاده از خاصیت عملگر Xor قادر به بازسازی داده اولیه هستیم.

علاوه بر این، عملگر Xor، همانگونه که در شکل بالا دیده می شود، از هیچ حافظه اضـافی ای برای نگاه‌داری تفاوت‌ها استفاده نمی کند (به عنوان مثال در شکل بالا، نتیجـه خروجـی همـان ۹ بیتی را مصرف کرده است که نتیجه حدس‌زده‌شده مصرف می‌کرده‌است).

خصوصیت قابل توجه دیگر عملگر Xor، در نتیجه آن نهفته است. این عملگر هنگـامی کـه رشته ورودی درست حدس زده‌شده‌باشد (ی‌عنی هنگامی کـه دو مقـدار متنـاظر در رشـته ورودی و رشته حدس‌زده‌شده، برابر باشند)، مقدار صفر و وقتی کـه ایـن رشـته غلـط حـدس زده شـده باشـد (ی‌عنی هنگامی که دو مقدار متناظر در رشته ورودی و رشته حدس‌زده‌شده، برابر نباشند)، مقدار یک را بازمی‌گرداند. این خاصیت منجر به آن می شود که تعداد صفرها در خروجی زیاد شده، پراکنـدگی داده‌های صفر و یک بیشتر گردد و مقدار فشرده‌سازی نیز بیشتر شود.

۸-۳- نگاهی بر بازگشت‌پذیری روش:

همانگونه که در بخش قبل بحث شد، اگر بدانیم که چه گرامری برای توصیف فایل استفاده-شده‌است، قادر خواهیم بود تا داده ورودی را طبق روندی معکوس آنچه در فشرده‌سازی رخ می داد، بازیابی کنیم. و با توجه به اینکه اندیس گرامر استفاده‌شده، به عنوان سربار در فایل نهایی ذخیره مـی-شود، عمل معکوس فشرده‌سازی به سادگی قابل انجام است.

این روند شامل ساختن نتیجه حـدس‌زده‌شـده از روی نتیجـه خروجـی سیسـتم کـه اکنـون ورودی آن است (با توجه به این نکته که بیت‌های کلیدی به صورت دست‌نخورده در جـای اصـلی-شان باقی مانده‌اند)، و انجام عمل Xor بر روی بیتهایی که کلیدی نیستند می‌باشد.

نتیجه خروجی :	۰	۰	۱	۰	۱	۰	۰	۱	۰
نتیجه حدس زده شده :	۰	۰	۱	۰	۰	۰	۰	۱	۰
بیتهای ورودی :	۰	۰	۱	۰	۱	۰	۰	۱	۰

شکل ۸-۵- پروسه بازیافت فایل اصلی

شکل بالا نمایانگر وقایعی است که در هنگام بازیافت فایل اصلی رخ می دهد. این وقایع، ترتیب عکس آنچه در هنگام فشرده سازی رخ می دهد، هستند.

۸-۴- مقدار تصادفی بودن روش:

همانگونه که در فصل قبل بحث شد، هر روشی که به عنوان پیش‌بینی‌کننده به کار گرفته می-شود، باید دارای خصوصیت شبه تصادفی بودن باشد. به این معنی که نتیجه خروجی این روش، باید از نتایج احتمالی به دست آمده برای یک فرایند تصادفی پیروی کند.

در فصل قبل، برای اندازه‌گیری این معیار روشی را هم ارائه دادیم. این روش، بر مبنای اندازه گیری تعداد دفعات تکرار n صفر متوالی، در یک رشته بی پایان از داده ها بود. سپس، این مقادیر با مقادیر ریاضی محاسبه‌شده مطابقت داده می شدند.

جدول زیر حاوی مقادیر محاسبه‌شده برای این معیار می باشد. این مقادیر طی فرایند شبیه-سازی ورودی این سیستم و بررسی آماری خروجی آن، و با استفاده از مقادیر تصادفی در ورودی سیستم شبیه‌سازی شده است.

۴	۳	۲	۱	مقدار N
1.5625e-2	3.125e-2	6.25e-2	1.25e-1	مقدار مورد انتظار
0.010	0.030	0.495	0.118	مقدار واقعی
۸	۷	۶	۵	مقدار N
8.765625e-4	1.953125e-3	3.90625e-3	7.8125e-3	مقدار مورد انتظار
0.000	0.002	0.003	0.005	مقدار واقعی

همانگونه که از جدول بالا برمی آید، تفاوت مقادیر واقعی با مقادیر مورد انتظار کوچک می-

باشد که نشان دهنده درصد خوب تصادفی بودن روش می‌باشد.

فصل نهم

نتایج آماری و تحلیلی

در این بخش، به نتایج عددی به دست آمده و تحلیل آنها خواهیم پرداخت، بر اساس این تحلیل‌ها، به ارائه چندین روش متفاوت برای انتخاب گرامرها خواهیم پرداخت.

۹-۱- معیارها

در زیر به ارائه تعدادی معیار می‌پردازیم که هر یک برای اندازه‌گیری نسبی عملکرد هر گرامر و یا کل گرامرها بر یک ورودی قابل تعریفند. هیچ یک از این معیارها بدون نقص نیست و هر کدام مزایا و معایب خاص خود را دارد.

- معیار درصد عملکرد مثبت (Positive% و یا P%):

این معیار، بیانگر متوسط افزایشی است که یک گرامر و یـا مجموعـه‌ای از گرامرهـا در تعـداد صفرهای مقدار زیادی داده ایجاد کرده‌اند.

این معیار به چندین حالت متفاوت قابل اندازه‌گیری است که همگی قابل تبدیل به هم هسـتند. در پیاده‌سازی‌های ما، این معیار به صورت تفاوت تعداد صفرهای خروجـی و ورودی انـدازه-گیری شده است.

بزرگتر بودن این معیار برای یک گرامر، نشان‌دهنده خوبتر بودن عملکرد آن گرامر می باشد.

از مشکلات این معیار، در هنگام استفاده آن برای گرامرها، عدم توجه آن به انتخاب شدن و یـا انتخاب نشدن گرامر برای استفاده می‌باشد.

از مزیت‌های این معیار، وابستگی مستقیم آن به عملکـرد متوسـط گرامـر و یـا مجموعـه‌ای از گرامرها می‌باشد.

- معیار درصد استفاده (Use% و یا U%):

این معیار فقط برای یک گرامر و نه مجموعه‌ای از گرامرها قابل تعریـف اسـت و نشـان‌دهنـده تعداد دفعات استفاده یک گرامر، از بین مجموعه‌ای از گرامرها، برای تعداد زیادی داده می‌باشد (استفاده از یک گرامر، منوط به بهترین بودن این گرامر در بین مجموعه انتخاب شده گرامرهـا است).

اندازه‌گیری این معیار با توجه به تعداد دفعاتی که این گرامر استفاده شده است و تعداد دفعاتی که آزمایش انجام گرفته است، قابل محاسبه می‌باشد.

عیب این معیار در عدم توجه آن به مقدار تاثیر یک گرامر در ورودی نهفته است.

مزیت این معیار در توانایی انتخاب گرامرهایی از سوی آن است که در تعداد دفعات بیشتری، ی‌هترین عملکرد را از خود نشان داده‌اند.

- معیار میانگین بهبود انتروپی[1] (Average Entropy Improvement و یا AEI):

این معیار برای یک و یا مجموعه‌ای از گرامرها قابل تعریف است و نشان‌دهنده میانگین افزایشی است که یک و یا مجموعه‌ای از گرامرها در انتروپی سیستم ایجاد کرده‌اند (انتروپی مورد بحث در این بخش شامل سربار نمی‌شود و فقط خروجی بدون سربار را در نظر می-گیرد).

عیب این معیار، همانند معیار P%، در هنگام استفاده آن برای گرامرها و در عدم توجه آن به انتخاب شدن و یا انتخاب نشدن گرامر برای استفاده می باشد.

مزیت این معیار، باز هم همانند P%، وابستگی مستقیم آن به عملکرد متوسط گرامر، در بین مجموعه‌ای از گرامرها می‌باشد.

[1] Entropy

۹-۲- نتایج ابتدایی

این نتایج بر مبنای تمام گرامرهای ۳ تایی ممکن (که به صورت اتوماتیک تولید شده بودند و بیش از ۵۰۰۰ گرامر را شامل می‌شدند) به دست آمده‌است و گرچه که بهترین AEI را تضمین می‌کند، ولی سربار بالایی از نظر فضای اشغالی دارند و اجرای آنها بسیار کند است. نتایج این مجموعه از گرامرها برای سه دسته از ورودی‌ها به صورت زیر می‌باشد:

	P%	AEI	BPC
Sample 1	100%	-0.1	1.1
Sample 2	98%	-0.05	1.05
Sample 3	95%	-0.04	1.04

همانطور که در بالا مشاهده می‌شود، در هیچ کدام از نمونه‌های بررسی شده، معیار BPC که معیار اصلی فشرده‌سازی است و در فصل دوم معرفی شد، به حد قابل قبول (که بتوان نام فشرده‌سازی را بر آن نهاد)، نرسیده است.

مشکل اصلی این روش، گرامرهای مورد استفاده آن هستند. این روش از تمام ۵۰۰۰ گرامر تولیدشده، بدون توجه به خوبی و یا بدی عملکرد این گرامرها، استفاده می‌کند. این کار از سویی سیستم را کند می کند و بدتر از آن اینکه بسیاری از این گرامرها در عمل یا استفاده نمی شوند و یا به ندرت استفاده می شوند. در حالی که بعضی دیگر از آنها، به کرّات استفاده می‌گردند.

از سوی دیگر، برای نشان‌دادن اندیس این گرامرها، به ۱۳ بیت اطلاعات در هر بلوک از داده ها به عنوان سربار نیازمندیم.

همه اینها در حالی است که با استفاده از یک الگوریتم خوب برای انتخاب گرامرها، می توانستیم تعداد ۱۲۸، ۶۴، ۳۲ و یا .... از گرامرها را انتخاب نموده و با تحمل مقدار کمی کاهش در معیار AEI، سربار کار را از ۱۳ بیت برای هر بلوک به ۶ و یا ۷ بیت برای هر بلوک کاهش دهیم.

۹-۳- انتخاب گرامرها بر اساس بهترین P%:

در این بخش، 2^k گرامر را بر اساس معیار P% مورد بررسی و انتخاب قرار می‌دهیم. پس از انتخاب گرامرها، آزمایش بالا را فقط با استفاده از این گرامرها انجام می دهیم. نتایج به شرح زیر است:

K	P%	AEI	BPC
7	91%	0.002	0.998
6	89%	-0.002	1.002
5	80%	-0.003	1.003

۹-۴- انتخاب گرامرها بر اساس بهترین U%:

در این بخش نیز عمل انتخاب گرامرها (با توجه به معیار U%) انجام می شود. پس از انتخاب گرامرها، آزمایش‌ها دوباره با این دسته از گرامرها انجام می‌شود که نتایج آن برای مقادیر مختلف K در زیر آمده است:

K	P%	AEI	BPC
7	99%	0.98	0.02
6	96%	0.995	0.005
5	96%	1.001	-0.001

۹-۵- انتخاب گرامرها بر اساس بهترین AEI:

در این بخش نیز ابتدا گرامرها را با توجه به معیار AEI انتخاب می‌نماییم. سپس گرامرهای انتخاب شده را، دوباره مورد آزمایش قرار می دهیم. نتایج آزمایش‌ها به ازای مقادیر مختلف K در جدول زیر آمده است:

K	P%	AEI	BPC
7	90%	0.999	0.001
6	89%	0.999	0.001
5	80%	1.001	-0.001

۹-۶- نتایج ریزتر برای بهترین پیکربندی:

همانگونه که از بررسی جداول بالا مشخص می‌گردد، بهترین پیکر بندی برای این سیستم در حالتی است که ۱۲۸ گرامر بر اساس معیار U% انتخاب شوند. جدول زیر، نتایج این پیکر بندی را بر

روی همان سه نمونه ای که در بخش نتایج ابتدایی بررسی شده‌انـد، نشـان مـی دهـد. بررسـی ایـن دو جدول، تفاوت فاحش نتایج آنها و بهبود در نتایج کلی را آشکار می‌کند.

	P%	AEI	BPC
Sample 1	100%	0.97	0.03
Sample 2	98%	0.98	0.02
Sample 3	98%	1	0

فصل دهم

نتیجه گیری و کارهای آتی

۱۰-۱- نتیجه گیری

با توجه به آنچه در فصل ۹ صحبت شد، حداکثر فشرده‌سازی به دست‌آمده در روش‌های مورد بررسی، ۲ در صد بهبود نسبت به روش‌های قبلی را نشان می‌دهد. گرچه ۲ درصد به خودی خود عدد بزرگی نیست، ولی باید توجه نمود که این ۲ درصد، درصدی است که روش‌های دیگر نتوانسته‌اند کاری روی آن انجام دهند و در حقیقت بهبودی است که در نتایج روش‌های دیگر داده شده است.

همان‌طور که در فصل قبل مشاهده شد، معیار درصد استفاده (U%) بهترین نتیجه را در بین سه معیار داده شده به دست آورد. دلیل بهتر بودن این معیار نسبت به معیار P%، ضعف معیار P%، در یافتن گرامری است که بهترین افزایش را ایجاد می‌کند. علاوه بر آن، بررسی داده‌های معیار P%، بیانگر آن است که داده‌های این معیار بسیار نزدیک به هم هستند و تفاوت معناداری با یکدیگر ندارند. مزیت

معیار U% بر معیار AEI نیز در وابستگی مستقیم معیار U% به انتخاب شدن یا نشـدن گرامـر نهفتـه است، در حالی که معیار AEI به صورت متوسط بر روی تمام نمونه‌ها (بـدون توجـه بـه آنکـه واقعـا گرامر انتخاب می‌شود و یا نه) اندازه گیری می‌شود.

۱۰-۲- کارهای آتی

کارهایی که در ادامه این پروژه می توان انجام داد، به شرح زیر هستند:

- می توان با استفاده از روش‌های هوش مصنوعی مانند XCS[1] که با اسـتفاده از ترکیـب روش-های یادگیری تقویتی[۲] و روش‌های محاسبات تکاملی[۳] به یادگیری چگونگی انتخـاب بهتـرین گرامرها در کوتاه‌ترین زمان می‌پردازند، به کوتاه نمودن زمان اجـرای الگـوریتم فشـرده‌سـازی کمک کرد. علاوه بر آن، این روش‌ها قابلیت تطبیق با داده‌های خاص کاربران را دارا می‌باشند.
- استفاده از معیارهای دیگر برای جداسازی و انتخاب گرامرها
- استفاده از الگوریتم‌های ژنتیک و یا دیگر روش‌های عمومی حـل مسـاله بـرای یـافتن بهتـرین مجموعه گرامرها
- بافتن دیگر توابع قابل استفاده به عنوان توابع پیش‌بینی کننده

---

۱ eXtended learning Classifier System
۲ Reinforcement Learning
۳ Evolutionary Computing

فهرست منابع

[1] Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan, Dynamic Perfect Hashing: Upper and Lower Bounds, SIAM Journal on Computing 23 (1994).

[2] Esko Ukkonen, On-line Construction of Suffix Trees, Algorithmica 14 (1995).

[3] Peter Weiner, Linear Pattern Matching Algorithms, Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory, (1973).

[4] N. Jesper Larsson, Structures of String Matching and Data Compression (1999).

[5] Edward M. McCreight, A Space-Economical Suffix Tree Construction Algorithm, Journal of the ACM 23 (1976).

[6] Storer, J.A., (1988) Data Compression, Computer Science Press, Rockville, M.

[7] Mark Nelson: The Data Compression Book (2000).

[8] http://www.qic.org

[9] Jacob Ziv and Abraham Lempel, A Universal Algorithm For Sequential Data Compression, IEEE Transactions on Information Theory IT-23 (1977).

[10] Guy E. Blelloch: Introduction to Data Compression (2001).

[11] Jacob Ziv and Abraham Lempel, Compression of Individual Sequences Via Variablerate Coding, IEEE Transactions on Information Theory IT-24 (1978).

[12] Suzanne Bunton, On-line Stochastic Processes in Data Compression, Ph.D. Thesis, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, USA, December (1996).

[13] Matt Powell: Evaluating Lossless Compression Method (2001).

[14] Gjerrit Meinsma :Data compression & Information Theory(2000).

[15] Richard S. Sutton, Andrew G.Barto: Reinforcement Learning, An Introduction (1999).

[16] Mark Nelson: The Data Compression Book (2000), Chapter 1.

[17] Mark Nelson: The Data Compression Book (2000), Chapter 2.

[18] www.ieee.org, DCC Conference
[19] Bernhard Balkenhol, Stefan Kurtz, Yuri M. Shtarkov: Modifications of the Burrows and Wheeler Data Compression Algorithm (1999).
[20] Moses Charikar, etc: Approximating the Smallest Grammar: Kolmogorov Complexity in Natural Models (2002).
[21] J.Gerard Wolf: Neural Mechanisms for Information Compression by Multiple Alignment, Unification and Search (2003).
[22] Mark Nelson: The Data Compression Book (2000), Chapter 3, 4.
[23] Google search for L-System+ Fractals.