دانشگاه صنعتی شریف
دانشکده علوم ریاضی

گزارش پروژه درس
ساختمان داده

فشرده سازی متن با الگوریتم هافمن

نگارش
مسعود علیپور
نیما کلیدری
محمدرضا پورنادر

استاد درس
دکتر رضوانی

۳۱ تیر۱۴۰۱

فشرده‌سازی فایل متنی با الگوریتم هافمن

چکیده

فشرده سازی داده ها یکی از کلیدی ترین مفاهیم در تئوری اطلاعات است. انواع مختلف رسانه با حجم بالا را میتوان با فشرده سازی به فایل های با حجم کمتر تبدیل کرد. این فایل های کم حجم تر برای انتقال داده از طریق اینترنت بسیار کاربردی تر هستند و به پهنا باند کمتر و فضای ذخیره سازی کمتری نیاز دارند. یکی از انواع این رسانه ها فایل های متنی هستند که با کمک الگوریتم هافمن و کدگذاری مناسب، می توان حجم فایل را به مقدار قابل توجهی کاهش داد. همچنین با استفاده از روش های ابداعی، با متحمل شدن مقداری خسارت از دست دادن داده، در عوض حجم بسیار کمتری از حجم اصلی فایل را با داده های نسبتا سالم، در اختیار داشته باشیم. در این پروژه، نیما کلیدری برنامه نویس جاوا و نقش پیاده سازی شاکله اصلی کد و نوشتن بخشی از گزارش پروژه است. مسعود علیپور و محمدرضا پورنادر نیز ایده پرداز برای مسیر پروژه، عیب‌یاب کد و نویسنده گزارش هستند.

کلمات کلیدی: الگوریتم هافمن، فشرده سازی، ساختمان داده، فشرده سازی متن، فشرده سازی متن با از دست رفتن داده، فشرده سازی متن با استفاده از درخت، استفاده از درخت در الگوریتم هافمن، تغییر درخت در الگوریتم هافمن.

فهرست مطالب

فهرست تصاویر

۱ مقدمه

فشرده سازی یکی از مفاهیم کلیدی در تئوری اطلاعات است. فایل های مختلف مانند تصاویر یا فایل های متنی به صورت خام حاوی جزئیات زیادی هستند و حجم زیادی اشغال می‌کنند.
با کاهش حجم و فشرده سازی فایل ها، فضای ذخیره سازی کمتری اشغال می‌شود و انتقال داده ها نیز با سرعت بیشتر و پهنای باند کمتری صورت می‌گیرد. این امر برای شرکت های بزرگی مانند نتفلیکس، یوتیوب و اسپاتیفای که حجم زیادی از اطلاعات را در هر لحظه انتقال می‌دهند بسیار حیاتی است و به بهبود عملکرد آن ها منجر می‌شود. فشرده سازی بدون اتلاف فقط با کدگذاری محتویات فایل صورت می‌گیرد اما در فشرده سازی با اتلاف، علاوه بر کدگذاری، برخی از جزئیات غیرضروری و جزئی فایل از دست می‌روند.
هدف از انجام این پروژه کاهش حجم فایل های متنی ابتدا به صورت lossless و سپس به صورت lossly است. ابتدا فایل متنی را با الگوریتم هافمن کدگذاری می‌کنیم که فایل فشرده شده بدون اتلاف را به ما می‌دهد. سپس با توجه به ضریب اتلاف، برگ های با بیشترین ارتفاع را به ترتیب حذف می‌کنیم که منجر به فشرده سازی با اتلاف می‌شود. نحوه کار کد به شرح در قسمت روش بیان می‌شود. در نهایت در قسمت نتایج به مقایسه عملکرد برنامه به ازای ورودی های متفاوت می‌پردازیم و نمودارها را تحلیل می‌کنیم.

۲ روش

برای فشرده سازی فایل ها از زبان برنامه نویسی جاوا ۱٫۸ استفاده شده است.
کد شامل دو بخش اصلی کدگذاری و کدگشایی است که به شرح هرکدام می‌پردازیم.

۱.۲ کدگذاری

در این قسمت فایل متن انگلیسی شامل کاراکتر های ASCII را به عنوان ورودی می‌گیریم و خروجی نهایی، فایل کدگذاری شده و فشرده شده به همراه دیکشنری در ابتدای فایل (به عنوان متادیتا) برای کدگشایی آن است. ورودی دوم برنامه ضریب lossly است. اگر مقدار صفر به عنوان ورودی داده شود، فشرده سازی ما lossless خواهد بود. ابتدا متن انگلیسی ورودی را کاراکتر به کاراکتر خوانده و تعداد دفعاتی که هر نماد به کار رفته است را می‌شماریم و ذخیره می‌کنیم. برای ذخیره سازی داده ساختاری ساخته‌ایم که کد اسکی نماد، تعداد دفعات مشاهده شدن و کد هافمن آن را ذخیره می‌کند. مقدار اولیه کد هافمن "" است و در مراحل بعدی مقداردهی می‌شود. اکنون این آرایه از شی های این داده ساختار ما را برحسب تعداد دفعات تکرار متناظر هر شی به کمک الگوریتم مرتب سازی ادغامی (merge sort) مرتب می‌کنیم.

در ابتدای شروع برنامه، درصد اتلاف که بین ۰ تا ۱۰۰ است، از کاربر دریافت میشود. این درصد، درصد تعداد نمادهایی که از متن حذف شده و به جای آنها نماد قرار می‌گیرد را نشان می‌دهد. این نماد اتلاف اطلاعات در اثر فشرده سازی را در آن جایگاه از متن نشان می‌دهد. قابل ذکر است از آنجا که اگر یک نماد حذف شده و بجایش مد قرار بگیرد، تغییری در فایل رخ نخواهد داد. لذا همواره اگر این مقدار برابر ۰ نباشد، یک نماد بیشتر از مقدار بدست آمده از متن حذف میشود. این حذف باعث تغییر در ساختار درخت میشود و علاوه بر اینکه متن مقدار کمتری برای ذخیره دیکشنری نیاز دارد، باعث میشود برادرش تک فرزند شده و یک جا برای طول کد کمتر برای یک کاراکتر پر تکرار تر باز بشود. با شکل دادن دوباره درخت، کاراکتر پر تکرار تر در ردیف پدر کاراکتر حذف شده مینشیند و طول کلی متن نیز کمتر میشود. (بنابر الگوریتم هافمن، اثبات میشود فرزند با بیشترین عمق، همواره برادر دارد، چون اگر ندارد میتواند به جای پدرش قرار بگیرد.)

نمادهای مربوط به خط بعدی، مد و فاصله از این قاعده مستثنی هستند، بنابراین مقدار حداکثر ضریب اتلاف کمتر از تعداد نمادهای متفاوت استفاده شده در متن است که این مقدار حداکثری در هنگام دریافت ضریب از کاربر به آن اطلاع داده می‌شود. پس از حذف هر کارکتر یکبار دیگر متن بررسی شده و هیستوگرام جدید برای حروف متن بدست می‌آید و سپس کاراکتر بعدی به همین ترتیب جایگزین میشود. این روند تا آنجا پیش میرود که به مقدار مورد نظر برای حذف حروف برسیم.

پس از بدست آوردن هیستوگرام جدید برای حروف و متن جدید، درخت هافمن را می‌سازیم. یک گره خالی می‌سازیم. در لیست مرتب شده بر حسب تکرار، شی با کمترین تکرار را به عنوان فرزند چپ و دومین شی با کمترین تکرار را به عنوان فرزند راست آن گره قرار می‌دهیم. مجموع دیتای تکرار دو فرزند را به گره پدر نسبت می‌دهیم. دو گره فرزند را از لیست حذف و گره پدر را به لیست اضافه می‌کنیم بدون اینکه نظم لیست بهم بخورد. پس لیست همچنان مرتب است. مراحل بیان شده را مجددا انجام می‌دهیم و آن قدر تکرار می‌کنیم که در نهایت لیست ما فقط ۳ گره داشته باشد. برای این ۳ گره نیز مراحل بیان شده را انجام می‌دهیم و درخت ما کامل می‌شود.

در این درخت، برگ ها، شی های متناظر با هر نماد هستند و گره های میانی مجموع دفعات تکرار برگ های زیر درخت های چپ و راست آن را نشان می‌دهند. بنابراین برگ با کمترین ارتفاع، نماد با بیشترین تکرار را نشان می‌دهد و برگ با بیشترین ارتفاع، نماد با کمترین تکرار را نشان می‌هد. ارتفاع این درخت برابر با تعداد نمادهای بکار رفته در متن است و ارتفاع درخت به اندازه ضریب اتلاف کاهش می‌یابد. در این مرحله، می‌خواهیم کد هافمن متناظر با نماد را پیدا کنیم. برای اینکار، در درخت هافمن، به هر یالی که پدر را به فرزند چپ‌اش متصل می‌کند عدد صفر و به یالی که گره پدر را به فرزند راست متصل می‌کند، عدد یک را نسبت می‌دهیم. کدهافمن هر برگ، مسیری از یال‌ها از ریشه به آن برگ است. این کد دودویی را برای هر برگ بصورت بازگشتی بدست می‌آوریم و مقادیر کد هافمن را در داده ساختارمان آپدیت می‌کنیم.

وقتی پیمایش از ریشه به هر برگ یعنی کاراکتر رسید رشته ای که در هر یال اپدیت میشد بعنوان کد هافمن آن کاراکتر در شی داده ساختار کاراکتر ذخیره میشود. به این ترتیب با کدگذاری برای همه کاراکتر های استفاده شده در متن، الگوریتم هافمن به پایان میرسد و نوبت به ساخت دیکشنری میرسد . برای ساخت دیکشنری از یک رشته آرایه به طول ۱۲۷ استفاده میکنیم به طوری که شماره هر خانه آرایه متناظر با کد اسکی کاراکتر مورد نظر باشد و در ان کد هافمن آن نماد ذخیره شده باشد. در قسمت های بعدی هنگام ساختن فایل هایی فشرده شده که در ابتدای آن دیکشنری قرار دارد از این آرایه استفاده میشود، به طوریکه کد اسکی هر نماد بصورت باینری در آورده و همراه با کد هافمن مربوط در ابتدای فایل بصورت پیوسته ذخیره میشود.

برای ذخیره سازی دیکشنری، ابتدا یک متن خالی بوجود می‌آوریم که بصورت ۰ و ۱ به آن کاراکتر اضافه

میکنیم. این کاراکتر ها در ذخیره دیکشنری بصورت بلوک های ۷ تایی ۰ و ۱ به متن اضافه شده و سپس متن کد گذاری شده به انتهای همین رشته افزوده . میشود (اما نه بصورت بلوک ۷ تایی) برای ذخیره دیکشنری ، ابتدا طول دیکشنری را بدست آورده و مقدار آن را بعنوان یک کاراکتر ۷ بیتی در ابتدای متن قرار میدهیم. سپس کد اسکی حرف را نیز به همین صورت تبدیل به یک بلوک ۰ و ۱ هفت تایی کرده و آن را در موقعیت n قرار میدهیم. در ادامه یک بلوک برای اندازه ها میسازیم که ۴ بیت اول آن تعداد بلوک ها و ۳ بیت بعدی، مقدار حروف Valid آخرین بلوک را نشان میدهد. با قرار دادن این بلوک های ۳ و ۴ بیتی در کنار هم، یک بلوک ۷ تایی بوجود می‌آید. سپس این بلوک ۷ تایی را نیز پس از بلوک ۷ تایی حرف دیکشنری قرار میدهیم. در ادامه به اندازه ذکر شده در بلوک دوم، بلوک ۷ تایی ساخته و به ادامه این قطعه می‌افزاییم. در بلوک آخر نیز همین کار را میکنیم اما با داشتن مقدار ،Valid در هنگام رمزگشایی، تنها بخش معتبر این بلوک را برمیداریم. همین کار را نیز به دفعات اندازه دیکشنری تکرار میکنیم. و این بخش متن باینری بدست می‌آید.

حال که کدهافمن هر کاراکتر را داریم، کدگذاری متن براحتی قابل انجام است. برای این‌کار از دیکشنری آماده شده که بر اساس ترتیب جدول اسکی مرتب شده است استفاده میکنیم. مقدار عددی هر کاراکتر، کد اسکی آن را نشان میدهد که با پیمایش در متن برای هر کاراکتر، کد هافمن ذخیره شده در خانه متناظر در آرایه دیکشنری را در یک رشته بصورت باینری ذخیره میکنیم و به متن باینری قبلی اضافه میکنیم تا به انتهای متن اصلی برسیم. در این متن باینری هر ۰ و ۱ مانند یک بیت هستند و می‌توانید هر دنباله ۷تایی از آن را یک بایت درنظر بگیریم و با تبدیل این رشته های ۷تایی به کد اسکی، طول آن را ۱/۷ ام برابر کنیم و متن کوتاه تر و فشرده شود. علت انتخاب عدد ۷ به این دلیل است که جاوا بیت ۸ام را برای خود ذخیره میکند و دنباله های ۸تایی در ۲بایت فشرده می‌شوند که مطلوب نیست. دنباله های ۷تایی به کد اسکی ۰ تا ۱۲۷ تبدیل می‌شوند.

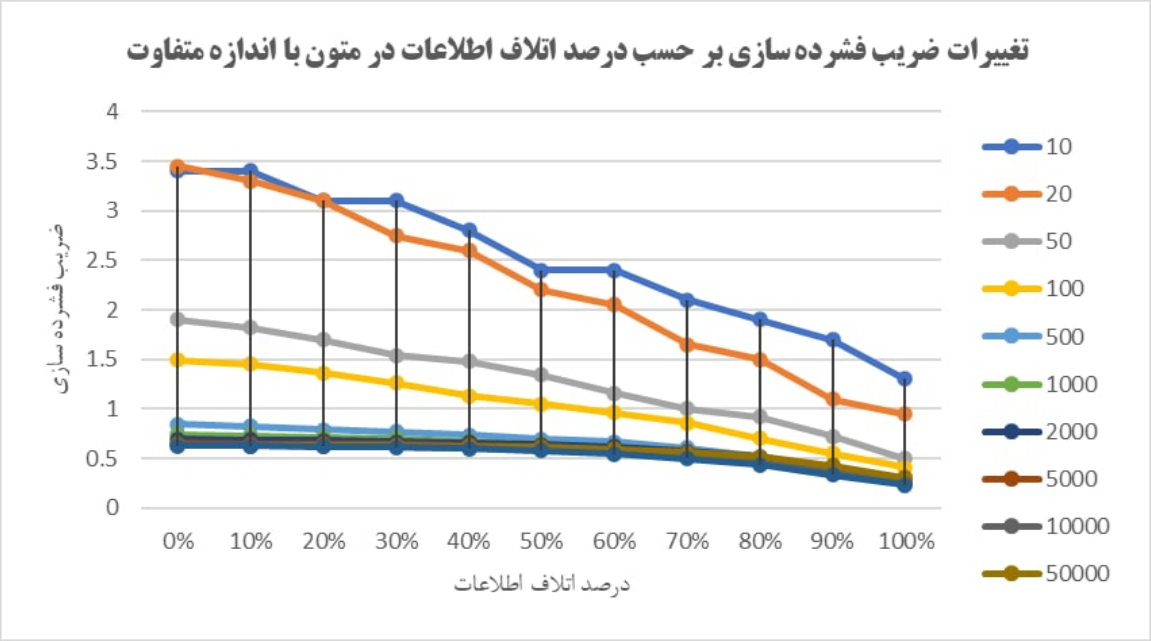
کد اسکی ۱۰ و ۱۳ به ترتیب مربوط به پایان خط و پایان متن است، بنابراین برای جلوگیری از مشکلات در رمزگشایی فایل، این مقادیر را به کدهای ۱۲۸ و ۱۲۹ تغییر می‌دهیم. نکته ای که برای پایان این کد وجود دارد، این است که کد بدست آمده، احتمالا مضرب ۷ نیست و برای ذخیره بیت های باقی مانده، نیاز به یک بلوک کامل داریم. اما امکان دارد در سمت چپ کد دودویی، کاراکتر ۰ باشد و از آنجا که برای کامل کردن یک بلوک غیر ۷ تایی نیاز به افزودن تعداد مناسبی ۰ به ابتدای بلوک داریم، این باعث رخ دادن ناتوانی در جدا کردن ۰ های معتبر از غیر معتبر میشود. برای حل این مشکل، به عنوان کاراکتر آخر متن، تعداد بیت های معتبر یکی مانده به آخر میگذاریم. بدین ترتیب، مشکل بوجود آمده برای بلوک آخر حل شده و متن کدگذاری شده با دستورالعمل مناسب، میتواند کدگشایی شود.

۲.۲ کدگشایی

در تابع کدگشایی، تنها لازم است که محل قرار گیری فایل کد شده را دریافت کنیم. سپس در این تابع ابتدا کل متن درون فایل را میگیریم و تنظیمات لازم را روی آن انجام میدهیم. برای مثال از کاراکتر اول اندازه دیکشنری را استخراج میکند و تمام کاراکتر هایی که برابر ۱۲۸ و ۱۲۹ هستند را به ۱۳ و ۱۰ بازگرداند. بعد از این لازم است حلقه ای اجرا کنیم که به اندازه دیکشنری اجرا بشود سپس دیکشنری را به همان صورتی که کدگذاری کردیم، استخراج میکنیم. به این صورت که در هر حلقه کاراکتر اول را به عنوان حرف در دیکشنری دریافت میکنیم؛ کاراکتر دوم را به دو بخش ۴ و ۳ بیتی تقسیم میکنیم و بخش اول که ۴ بیتی است را بعنوان تعداد بلوک ها و بخش دوم که ۳ بیتی است را تعداد کاراکتر های Valid بلوک آخر را نشان میدهد. برای بدست آوردن کد متناظر حرف، در ابتدا به تعداد بلوک ها منهای یک پیش میرویم و تمام این بلوک ها را با اضافه کردن صفر به اولشان ۷ بیتی کرده و به هم میچسبانیم.

سپس س در بلوک آخر، پس از تبدیل به یک رشته ۷ بیتی، تنها n کارکتر Valid آخرش را درنظر میگیریم و این بخش را نیز به بلوک های قبلی چسبانده و کد نهایی برای آن حرف بدست می‌آید. پس از کدگشایی یک حرف دیکشنری و ذخیره آن نیز به دنبال پیدا کردن حرف بعدی دیکشنری میرویم. سپس اینکار را آنقدر تکرار میکنیم تا به پایان دیکشنری برسیم. اکنون دیکشنری و دیتا با موفقیت از هم جدا شده‌اند. در مرحله بعد، آنقدر روی دیتا پیشروی میکنیم که به آخر آن برسیم. کاراکتر آخر متن نیز همانطور که در کدگذاری اشاره شد، تعداد کاراکتر های Valid کاراکتر یکی مانده به آخر را به ما میدهد. به استفاده از آن مقدار و جدا کردن بخش Valid کاراکتر یکی مانده به آخر، کل رشته باینری دیتا بدست می‌آید. حال با استفاده از پیشروی در این رشته از ابتدا به انتها و جستجو در دیکشنری برای حرف بیتی که اضافه میشود، و همچنین پیشروی پوینتر در هنگام یافتن حرف متناظر، میتوانیم به راحتی متن کد شده را بدست بیاوریم.

۳ نتیجه گیری

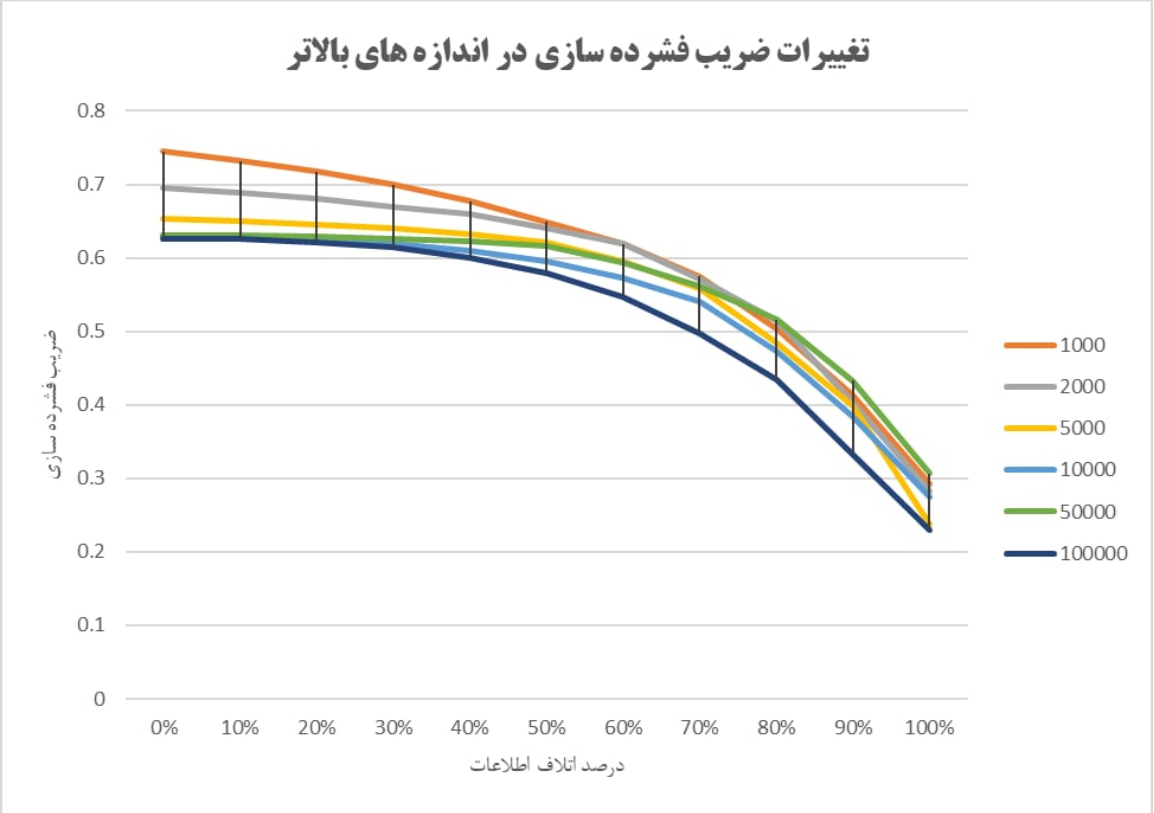


شکل ۱.۳ : متون با حجم متفاوت

پس از اتمام و راه اندازی، با دادن ورودی های متنوع، سعی شد با آنالیز نتایج، میزان کارایی و بهینگی روش بدست بیاید. دو مدل آزمایش کلی برای اینکار انجام شد که در یکی درصد فشرده سازی برای متن هایی نرمال با اندازه های مختلف بر حسب درصد اتلاف های مختلف، بررسی شده است. در حالت دیگر، درصد فشرده سازی های مختلف برای متن هایی بررسی شده که سایز یکسانی دارند، اما تنوع کاراکتر در آنها متفاوت است.

۱.۳ حجم های متفاوت

در این قسمت به برنامه متونی جدا شده از ادبیات انگلیسی، با تعداد کاراکتهای متفاوت داده شده است. در متن های کوتاه و با تعداد کاراکتر کم، مشاهده می‌شود که حجم فایل کد شده از فایل اولیه بیشتر است که



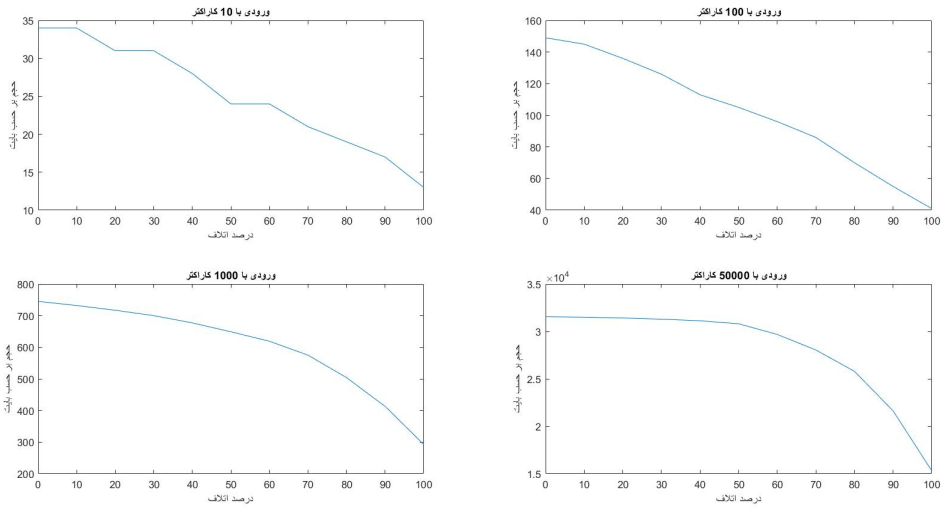
شکل ۲.۳: متون با حجم های متفاوت (تعداد کاراکتر بیشتر)

نامطلوب بودن برنامه را در متون کوتاه نشان می‌دهد. علت این موضوع در قسمت ۳.۳ که آنالیز دیکشنری است توضیح داده شده است.

با بیشتر شدن تعداد کاراکترها، ضریب فشردگی کوچکتر می‌شود و در متن های با ۵۰۰ کاراکتر یا بیشتر، ضریب فشردگی lossless از ۱ کوچکتر می‌شود که نتیجه مطلوب ما است. از نزدیک بودن مقادیر ضریب فشردگی در فشردگی بدون اتلاف به ازای متون ۵ هزار تا ۱۰۰ هزار کاراکتری، می‌توان نتیجه گرفتن که ضریب فشردگی در بی‌نهایت، تقریبا به ۰.۶ میل می‌کند و با این الگوریتم به مقادیر بهینه تر از آن نمی‌توان دست یافت.

در فشردگی با اتلاف ۱۰۰% هم مشاهده می‌شود که ضرایب فشردگی در حال همگرا شدن به مقداری بین ۱.۰ و ۲.۰ است. در این حالت تمام کاراکترها با مد جایگزین شده‌اند. بنابراین حجم دیکشنری کاهش یافته است و چون تنوع کاراکترها نیز از بین رفته است، پس هر بایت در متن عملا به یک بیت تبدیل شده است و از آنجا که هر ۷ بیت در اینجا تبدیل به یک بایت شده است، طول متن در این حالت به مقدار حدی ۱۴۳.۰ میل میکند.

با بررسی روند تغییر ضریب و حجم با افزایش درصد اتلاف در متن های کوتاه که توزین کاراکترها



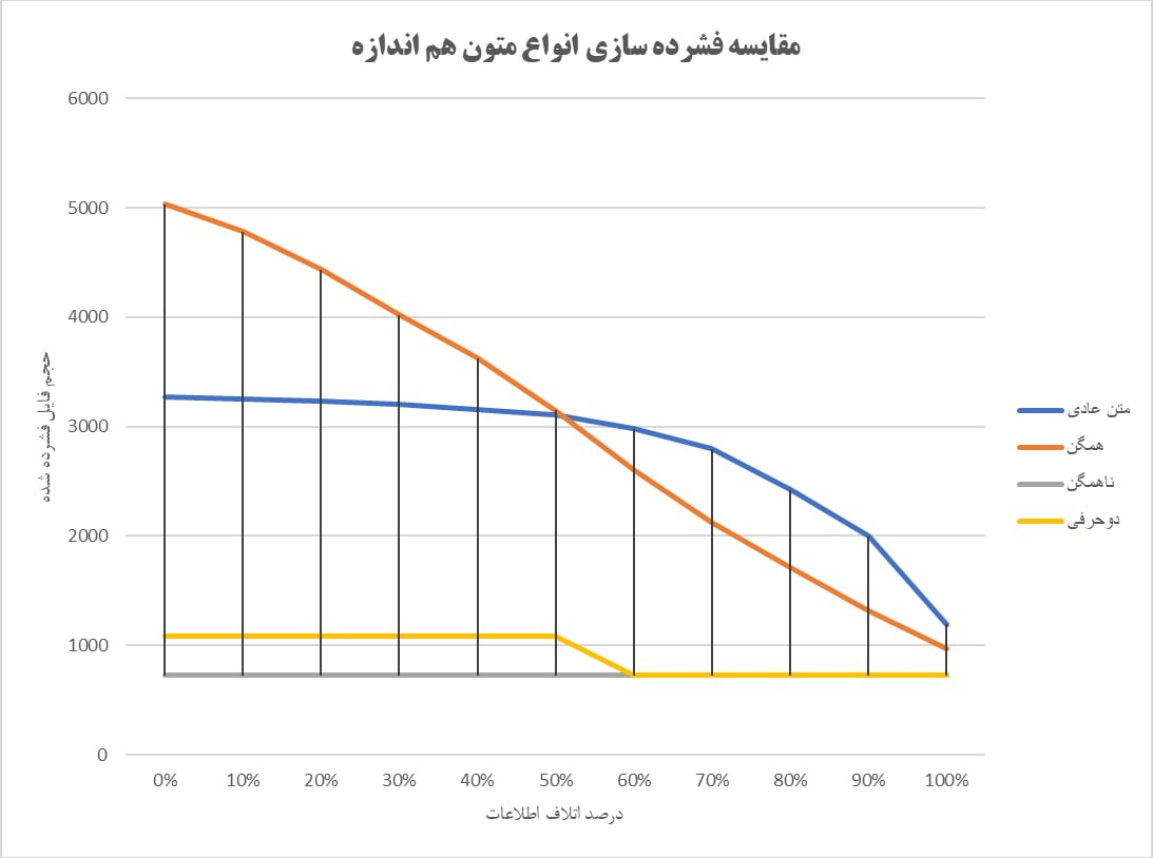
شکل ۳.۳: حجم بر حسب درصد اتلاف برای چهار حجم اولیه کلیدی

تقیبا یکسان است، شیب نمودا تقریبا بصورت خطی کاهش می‌یابد. اما در متون طولانی و با تعداد کاراکتر بالا، برخی از کاراکترها مانند حروف صدادار که کاربرد بیشتری دارند، بسیار بیشتر از سای کاراکترها تکرار می‌شوند. بنابراین در ابتدا، با اتلاف اطلاعات، ضریب فشردگی تغییر محسوسی نمی‌کند و شیب خط کم است. با گذشتن درصد اتلاف از مرز ۵۰ % شیب نمودار بیشتر می‌شود و نمودار تقریبا سهمی شکل می‌شود. این شیب در نمودار ۲.۳ قابل ملاحظه است. علت این افزایش تغییرات شیب، استفاده زیاد یک کاراکتر در متن است که با حذف آن، تغییر قابل توجهی در متن ایجاد می‌شود. با حذف یک کاراکتر که مکررا در متن استفاده شده است، ارتفاع درخت یک واحد کم می‌شود و تعداد زیادی از کاراکترها با کد هافمن کوتاه‌تری متناظر می‌شوند.

۲.۳ توزیع کاراکتر متفاوت

در این قسمت چندین متن با تعداد کاراکتر یکسان ولی با تنوع کارکتری متفاوت استفاده شده است. در شکل ۳.۳ نمودار حجم خروجی بر حسب درصد فشرده سازی برای ۴ متن با حجم ۵۰۰۰۰ کاراکتر و تنوع کاراکتری متفاوت نشان داده شده است.

برای متن عادی یک متن عادی انگلیسی که از کتاب های ادبیات انگلیسی جدا شده است، قرار داده شده است. در این متن تنوع کاراکتری نرمال و مانند متون عادی است؛ برای مثال تعداد فاصله ها و حروف صدادار از سایر حروف بیشتر است و بسیاری از کاراکترهای ASCII نیز استفاده نشده اند. در این

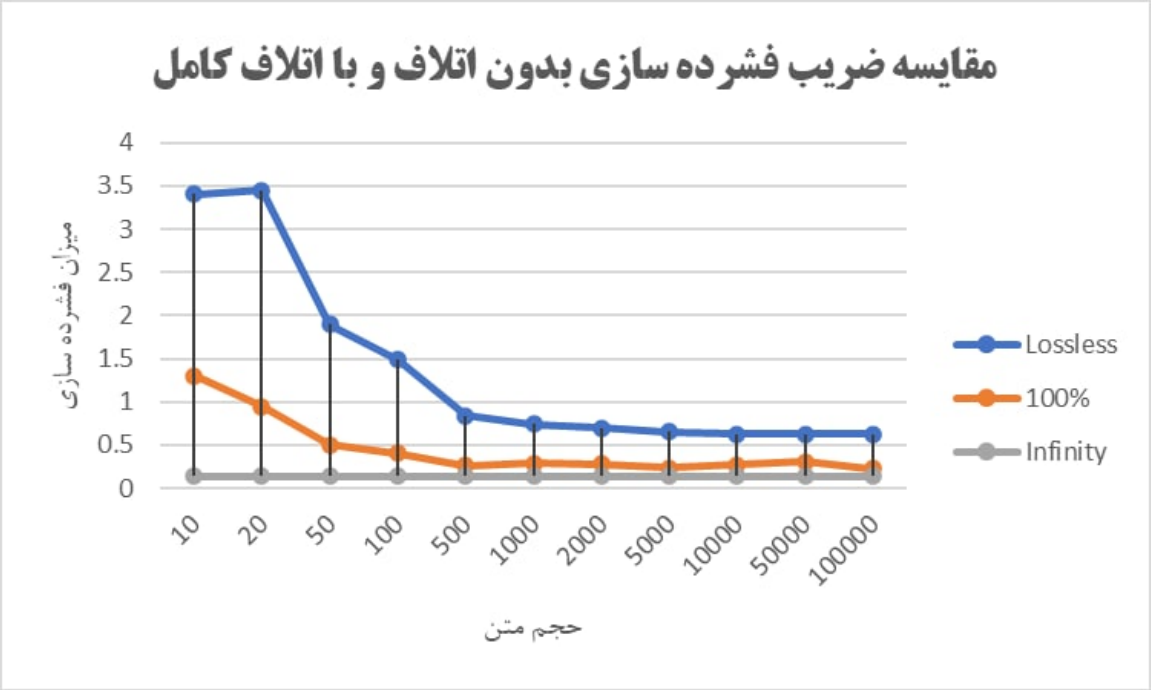


شکل ۴.۳: متون با اندازه یکسان و تنوع کاراکتر متفاوت

حالت در ابتدا که با بالارفتن درصد اتلاف حجم خروجی با شیب کمی تغییر میکند چراکه تعداد تکرار کاراکترهای حذفی در متن کم است و با حذف آنها تغییر محسوسی دیده نمیشود؛ اما با بیشتر شدن درصد اتلاف کاراکترهای پرتکرار نیز حذف میشوند که حذف آنها شیب کاهش حجم خروجی را افزایش میدهد.

در متن همگن کارکترهای موجود تقریبا تعداد یکسانی دارند؛ طول کل متن مانند قبل ۵۰۰۰۰ کاراکتر است؛ در اینجا مشاهده میکنیم که الگوریتم حالت بهینه ای ندارد و در حالت بدون اتلاف نیز حتی با وجود طولانی بودن متن از ورودی حجم بیشتری دارد و عملا ناکارآمد است. اما این در حالیست که با زیاد کردن درصد اتلاف شاهد کاهش خطی و مناسب حجم هستیم چرا که با حذف هر کاراکتر تقریبا تعداد یکسانی از کاراکترهای قابل خواندن کم میشود.

در متن ناهمگن تنها ۵۰۰۰۰ از یک کاراکتر خاص (مانند a) استفاده شده است. در اینجا مشاهده میکنیم که با بالا بردن درصد اتلاف عملا هیچ کاراکتری به جز کاراکتر موجود حذف نمیشود لذا صرفا



شکل ۵.۳: مقایسه حالت بدون اتلاف و با اتلاف کامل با حالت حدی یک هفتم

تمام خروجی ها بازای درصد اتلاف های مختلف حجم یکسانی دارند. در این حالت الگوریتم بسیار بهینه است چون که صرفا با یک کد تک بیتی تمام دیکشنری ساخته میشود. لذا هم حجم دیکشنری کم است و هم حجم هر کاراکتر از ۷ بیت به ۱ بیت کاهش میابد؛ در این بخش حدودا حجم حاصل کمی بیشتر از یک هفتم حجم اولیه است. مقدار اضافه شده به دلیل حجم دیکشنری است. از آنجا که حجم دیکشنری ثابت است یا رشد بسیار کند تری از متن دارد، اثبات میشود در طول بسیار زیاد متن با متن ناهمگن، حجم حاصل به یک هفتم حجم اولیه میل میکند.

در متن دو حرفی هم تنها از دو حرف با پراکندگی یکسان استفاده شده است که در ۵۰ الی ۶۰ درصد اتلاف، یکی از کاراکتر ها به همراه نشانه ی خط بعد تبدیل به شده و تنها با دو بیت میتوان متن را کدگذاری کرد. اما به دلیل حجم دیکشنری بالاتر، حجم با ضریب اتلاف بالا در این بخش، کمی از حجم در بخش قبل بیشتر است.

۳.۳ حجم دیکشنری

در شکل ۵.۳، اثر حجم دیکشنری بر حجم نهایی در دو حالت بدون اتلاف و با اتلاف کامل نشان داده شده است. منحنی خاکستری، منحنی یک هفتم است که در حالت حدی، حجم متن خروجی در حالت نا همگن

به این مقدار میل میکند. از آنجا که حالت اتلاف حداکثری حدودا همان حالت تک کاراکتری میشود، انتظار داریم حالت با اتلاف حداکثری نیز به همان یک هفتم حجم میل کند؛ اما چیزی که جلوی این مورد را میگیرد، حجم دیکشنری است. همانطور که در پیاده سازی دیدیم، ذخیره هر کاراکتر در دیکشنری، حداقل ۳ و حداکثر ۱۸ بایت را اشغال میکند (که البته در متون عادی غیر ممکن است این مقدار از ۴ بیشتر شود). اما ذخیره همین ۳ یا ۴ بایت ها، در یک متن عادی شامل ۳۰ تا ۳۵ حرف، خود باعث افزایش شدید حجم فایل خروجی میشود. البته قابل ذکر است معمولا روند رشد حجم دیکشنری بسیار آرام است و از جایی به بعد تمام میشود.

تاثیرگذاری حجم دیکشنری، معمولا در فایل های با حجم کوچک دیده میشود. چراکه در یه متن ۲۰ بایتی حتی اگر ۳ کاراکتر هم کلا داشته باشیم، ۱۸ بایت صرف ذخیره دیکشنری میشود که با حجم اصلی حدودا برابری میکند. یک سری کاراکتر ثابت دیگر هم بنابر روش پیاده سازی داریم که البته حجم زیادی نمی برند. بنابراین مهمترین دلیلی که نمیگذارد حجم فایل های که در ابتدا حجم کم دارند حتی به یک هفتم حجم خودشان نزدیک هم شوند، حجم نسبی زیاد دیکشنری است که با توجه به شکل ۵.۳ ، این اثر در حجم های بیشتر بسیار کمرنگ میشود.

۴ بحث

سوال: چرا در هنگام فشرده سازی متن باینری به کد اسکی، به جای رشته های ۸بیتی، رشته های ۷بیتی در نظر گرفتیم؟ درست است که یک بایت، ۸ بیت است اما جاوا یک بیت را برای خود رزرو می‌کند و اگر رشته های ۸بیتی انتخاب کنیم، در ۲ بایت ذخیره می‌شوند که بهینه نیست و به کاهش حجم فایل کمک نمی‌کند. پس رشته های ۷ بیتی که در یک بایت ذخیره می‌شوند، برای الگوریتم ما مناسب تر هستند.

چرا کدهای اسکی ۱۰ و ۱۳ را به ۱۲۸ و ۱۲۹ تغییر داده می‌شود؟ اگر تغییر داده نشوند، برای خوانده شدن توسط کدگشا غیر ممکن میشود. برای مثال امکان دارد کدگذار به ۱۳ برسد در حین خواندن متن و آن را بعنوان پایان متن درنظر گرفته و خواندن را تمام کند. هنگامی که به کاراکتر ۱۰ میرسد، لازم است خواند را تمام کند و خط بعد را شروع کند که این خود باعث ایجاد کاراکتر های اضافی میشود. بنابراین منطقی تر است که ضرر تبدیل این کاراکتر هارا به کاراکتر های ۲ بایتی را متحمل شویم ولی متن به صورت دقیق کدگذاری و کدگشایی شود.

کتاب‌نامه

[۱] Huffman Text Compression Technique Article, *September 2016*. International Journal of Advanced Trends in Computer Science and Engineering. 3(8):103-108.

[۲] *Mohammad Ghodsi* . Introduction to Algorithms and Data structures Fatemi publications, 1399

[۳] Thomas H. Cormen Charles E. Leiserson Ronald L. Rivest Clifford Stein. *Introduction to Algorithms*, 1990 .

[۴] Huffman Implementing *https://www.programiz.com/dsa/huffman-coding*

[۵] Text generator for test *https://www.blindtextgenerator.com/lorem-ipsum*