ساختمان داده‌ها (۲۲۸۲۲)

مدرس: حسین بومری

[بهار ۱۴۰۰]

جلسه ۲۴: ترای و گراف

نگارنده: سهیل کشاورز

در این جلسه در ادامه‌ی مباحث مطرح شده در مورد کار با رشته‌ها، به داده‌ساختار ترای [1] پرداخته‌شد و سپس مبحث گراف آغاز شد.

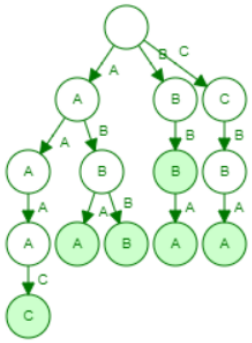
۱ ترای Trie

در جلسه‌ی قبل کاری که کردیم این بود که الگوریتم kmp را بررسی کردیم که خطی بود و در آن پترنی بدست می‌آمد که اردر جستجوی رشته برایش در $O(n)$ می‌بود. حالا یک ساختمان داده پیشنهاد می‌دهیم که جستجو در آن هم خطی است کارهای دیگری هم می‌تواند بکند. در این ساختمان به ازای یک دیکشنری از کلمه‌ها، درختی به نام ترای ساخته می‌شود.

این درخت یک نود ریشه دارد که کل دیکشنری زیردرخت آن است. از این نود ریشه، به ازای هر حرف الفبا، یک نود تعریف می‌شود و برای مثال نود متناظر با a ، کلمه‌هایی را دربر میگیرد که با a شروع می‌شوند. به همین ترتیب، هریک از نود های زیرین یک نود، به حروف الفبا اشاره می‌کنند و اگر کلمه‌ای متناظر با این نود ها نباشد، این نودها null می‌شوند. از طرفی دیگر، وقتی یک سری از کلمات وارد ترای می‌شوند، نود انتهایی آنها دارای یک علامت true می‌شود و ما در پیمایش درخت با دیدن این علامت متوجه تمام شدن کلمه‌ای می‌شویم که از بالا شروع شده و تا نود علامت خورده ادامه دارد.

مثال: میخواهیم یک ترای بسازیم که این کلمات را دارد: aaac, aba, abb, bb, bba, cba

ترای ساخته شده به شکل زیر است: (نود های null نشان داده نشده‌اند)



☒

یک فایده‌ای که ترای می‌تواند داشته باشد این است که در نودهایی که true اند، معنی کلمات را ذخیره کنیم و بدین صورت یک دیکشنری در ترای داشته‌باشیم؛ که در آن در اردر خواندن کلمات ورودی، معنی آن ها هم بدست بیاید، یعنی کمترین اردر ممکن. اگر کلمات بصورت عادی نگه‌داشته شده بودند، چجوری این کار انجام میشد؟ n مقایسه یا در بهترین حالت (سورت شده و در باینری سرچ) $\log n$ مقایسه صورت میگرفت که زمان هر مقایسه هم بستگی به طول رشته می‌داشت.

دیگه چیکار میشه کرد؟

- اگر به ما متنی با کلمات به هم چسبیده بدهند،می‌توان کلمه‌های متن را شکست، یا آن که بصورت اتوماتیک و بدون هزینه اضافی آن را ترجمه کرد. در واقع اینجا دیگر سرچ نداریم و هزینه، اندازه‌ی خواندن کلمه است.

[1] Trie

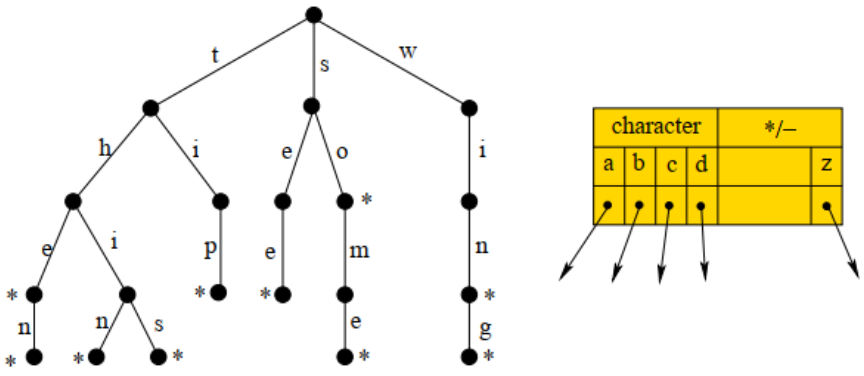
- کلمات با یک پیشوند خاص را پیدا کنیم.
- فهمیدن تعداد رخدادهای کلمه در یک متن (مشابه استرینگ سرچ) به این‌صورت که به ازای هر حرفی که خوانده می‌شود، یک اشاره‌گر جدید روی ریشه شروع به پیمایش می‌کند و اشاره‌گرهای قبلی نیز اگر بتوانند به جلو می‌روند. مثلا در مثال قبل، متن aacab را به ما می‌دهند. وقتی c خوانده می‌شود، یک اشاره‌گر جدید به c سمت راست ریشه می‌رود و دو اشاره‌گر روی فرزند a ریشه و فرزند a اش، چون هیچکدام فرزند c ندارند از بین می‌روند. نهایتا هم کلمه‌ای پیدا نمی‌شود و دو اشاره‌گر روی b فرزند a و b فرزند ریشه خواهیم‌داشت.
- فهمیدن اینکه چند کلمه برعکسشون هم در دیکشنری هستند. بدین صورت که یک‌بار هم از راست به چپ کلمات خوانده‌شوند.
- (خارج درس) مشابه kmp هم میشه کاری کرد که اگر اشاره‌گر به خطا بخورد، به یک جد مشخص برگشت داده شود. در واقع یک درخت برای kmp داریم.(Aho-Corasick)
- suggestion های گوگل

اردر ساخته شدن ترای چقدر است؟ به اندازه‌ی طول کلمات در بدترین حالت! اگر رشته‌ها بصورت سورت‌شده باشند از این هم کمتر می‌شوند. بنابراین از نظر ساخته شدن ترای کاملا بهینه بنظر می‌رسد.

نقطه ضعف ترای در کجاست؟ امتیازی که داشتیم این بود که در $O(1)$ از یک نود به نود بعدی برویم ولی اینکار ابخاطر ایندکس‌کردن (مثل count sort) حافظه زیادی می‌گیرد. مثلا فرض کنید الفبای بزرگی داریم (مثل حروف اسکی) و از یک نود که فقط b را در ادامه‌ی خود می‌بیند می‌خواهیم به b برویم. در این حالت در نودی که قرار داریم، تعداد زیادی (به‌اندازه‌ي تعداد حروف الفبا منهای یک) اشاره‌گر به null داریم و فقط یک اشاره‌گر مورد استفاده است پس حافظه بهینه‌ای مصرف نمی‌شود. بنابر این می‌توان حالتی را متصور شد که به اندازه‌ي الفبا برابر اندازه ورودی حافظه مصرف شده. البته وقتی الفبا بزرگ نباشد و کلمات زیادی هم در پیشوندهای خود مشترک باشند، این ساختار خوب است چرا که برای مجموع حروف کلمات نود ساخته نمی‌شود و مثلا یک نود در زیر خود چندین کلمه را پوشش می‌دهد.

راه حل ضعف در حافظه؟ نگه داری لیست پیوندی از حروفی که از هرکدام از نود ها خارج می‌شوند. در آن صورت حافظه اپتیمال می‌شود اما ترنزیشن بین دو نود به اندازه‌ی الفبا می‌تواند شود. اگر بجای لیست از ددج استفاده شود، این ترنزیشن‌ها در زمان لگاریتم اندازه الفبا انجام می‌شود

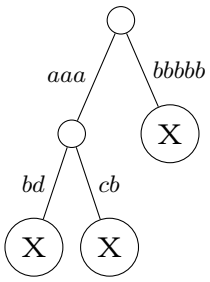
پس در ترای یا حافظه‌ی زیادی گرفته می‌شود و ترنزیشن ها سریع می‌شوند یا اینکه با حافظه‌ی بهینه ترنزیشن‌ها به اندازه‌ي الفبا طول می‌کشند. یک ویژگی مثبتی هم که ترای دارد، داینامیک بودن آن است و اینکه به راحتی می‌توان یک کلمه به آن اضافه کرد.



یک تِرای برای کلمات wing، win، so، some، see، tip، this، thin، then، و the.

در حالت ایندکس کردن و استفاده از آرایه، با کمپرس کردن می‌توان در حافظه صرفه‌جویی کرد. در این حالت، وقتی یک گره، تنها یک فرزند داشته باشد، تا جایی که به بیشتر از یک شاخه برخورد کند با فرزندانش ادغام می‌شود و در خود یک رشته (بجای یک کاراکتر) نگه می‌دارد. برای مثال برای ذخیره‌ی کلمات aaacb, aaabd, bbbbb، از این بهینه‌سازی در حافظه می‌توان استفاده کرد.

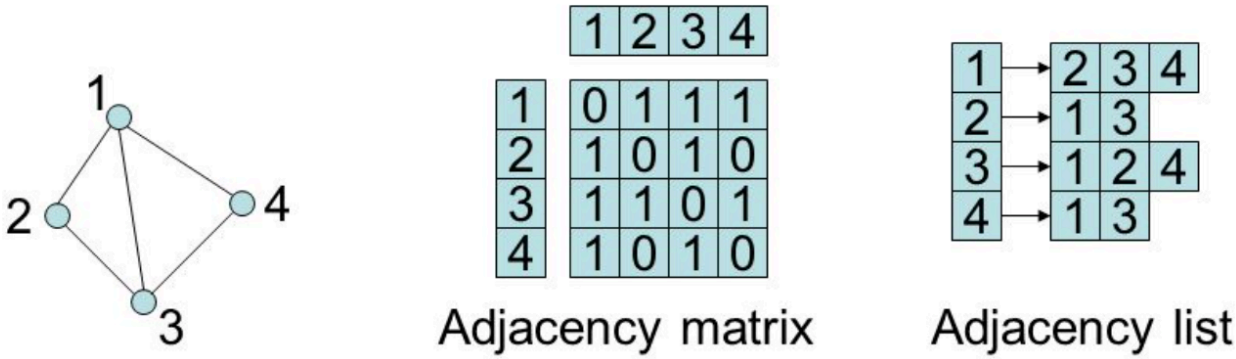
در پیاده‌سازی کاراکتر ها و رشته‌ها را در یال‌ها هم نگه داشت و یک پیاده‌سازی می‌تواند به اینصورت باشد که یال و گره دو فرزند کلاس Element باشند و یال هم دو نوع از جنس کاراکتر و از جنس رشته داشته‌باشد.



گراف

گراف G شامل یک مجموعه راس V و یک مجموعه از دوگانه‌های بین رئوس $E \subseteq V^2$ (یال‌ها) می‌شود. هر راس یک اسم دارد و هر یال هم می‌تواند جهت‌دار، وزن‌دار یا طوقه باشد. برای نگهداری گراف دو حالت داریم:

- (لیست مجاورت) در هر نود، یک لیست پیوندی داشته باشیم که نشان بدهد که به چه نود هایی وصل هست. خوبی این حالت این است که حافظه‌ی کمی می‌گیرد (مثلا اگر گراف بدون جهت باشد، به‌اندازه‌ی V+2E حافظه می‌گیرد). نقطه ضعف آن هم این است که اگر بخواهیم بفهمیم بین دو راس یالی هست یا نه، ضعیف کار می‌کنه و باید لیست را پیمایش کرد.
- (ماتریس مجاورت) در هر درایه a_{ij} مشخص کنیم که آیا از i به j یالی وجود دارد یا خیر. در ماتریس بدون وزن و جهت، وجود یال را با ۱ و عدم وجود را با ۰ نشان می‌دهیم و ماتریس متقارن است. طوقه‌ها را هم در قطر این ماتریس می‌توان پیدا کرد. در این ماتریس بر خلاف نگهداری گراف با لیست، وجود یال بین دو راس را به سرعت (در O(۱)) می‌توان فهمید اما مشکل آن، حافظه‌ی زیادی است که به‌کار می‌گیرد(این مشکل در گراف‌های با تعداد یال $O(V^2)$ مشکلی نیست اما در ماتریس‌های تنک [۲] و گراف‌های مسطح که تعداد یال ها هم اردر تعداد راس هاست،اختلاف زیادی با حد مطلوب دارد و حافظه اختصاص یافته، بهینه نیست). برای ساخت این گراف، باید رئوس گراف را از قبل داشت و نسبت به نگهداری با لیست مجاورت، سخت است که بتوان به گراف یک راس جدید اضافه کرد.



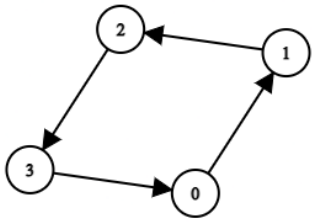
در روش ماتریس مجاورت، ضعف دیگر آن است که نمی‌توان یال چند گانه نگه‌داشت. راجع به به‌دست اوردن ترانهاده ماتریس هم، اگر ماتریس بدون جهت باشد، ماتریس ترانهاده خودش هم هست و اگر ماتریس جهت‌دار باشد، کافیست در پیاده‌سازی عملیات خود، جای سطر و ستون را عوض کنیم. بنابراین ترانهاده در پیاده‌سازی ماتریسی به سادگی قابل دسترسی است. اما در پیاده‌سازی با لیست‌مجاورت، باید گراف را از اول پیمایش کنیم و دوباره آن را بسازیم که نسبت به ماتریس، هزینه‌بر است.

یکی از کارهایی که در گراف می‌افتد، **انقباض** است. یعنی اینکه دو راس را با هم ادغام کنیم و راس جدیدی بسازیم. اگر یالی از این دو خارج شود، از راس جدید یالی به مقصد قبلی ایجاد می‌شود و اگر یالی به یکی از دو راس پیشین وارد شود هم، یال جدیدی به مقصد راس جدید ایجاد می‌شود. یال بین این دو راس هم طوقه‌ی راس جدید است. پیاده‌سازی خوب برای انقباض، لیست مجاورت است که در آن چسباندن لیست خروجی‌ها و زدن یک برچسب به راس قدیم و اشاره به اسم راس جدید در O(۱) انجام می‌شود. برچسب زدن در ماتریس خیلی کمکی نمی‌کند و کاری که قبلا در یک ستون یا یک سطر انجام می‌شد، حالا باید در دو ستون یا دو سطر انجام شود. اگر بخواهیم ماتریس مانند حالت عادی خود کار کند هم باید از اول ماتریس ساخته شود.

[2] Sparse Matrix

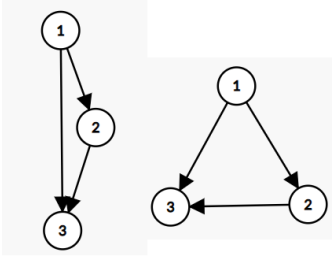
دقت داریم که وجود یال هایی با مقصد یکسان در لیست جدید یک راس، مشکلی بوجود نمی‌آورد چرا که در انقباض، یال‌های چندگانه هم ممکن است بوجود آیند. راجع به کاربرد این یال‌ها، ممکن است در جایی وزن این یال‌ها متفاوت باشد یا در جای دیگری، فارغ از وزن دار بودن یال ها به مسئله‌ی connectivity برخورد کنیم.

Connectivity یعنی حداقل تعداد گروه‌های توی گراف که هر گروه که به آن نگاه می‌کنیم، همه‌ی راس‌های آن گروه به هم مسیر دارند(تعداد مولفه‌های همبندی ـ شکل ۱). در گراف‌های جهت دار، هر دو مسیری که به هم وصل باشند، در یک دسته‌ی connectivity قرار دارند و در یک مولفه‌ی همبندی اند. در گراف‌های جهت‌دار، اگر بین دو راس یک مسیر از راس یک به راس دو و یک مسیر هم از راس دو به راس یک وجود داشته‌باشند، این دو راس در یک مولفه‌ي قویا همبند قرار دارند. در پیاده‌سازی با لیست، پیدا کردن این مولفه‌ها به کمک منقبض کردن، راحت تر و بهتر از حالت ماتریسی است. یک تعریف دیگر که برای connectivity داریم و بیشتر مورد استفاده قرار می‌گیرد، این است که تعداد یال‌هایی که می‌توان حذف کرد (مثلا سیم‌های یک شبکه کامپیوتری) و گراف همچنان connected بماند. در این تعریف است که یال چندگانه موضوعیت پیدا می‌کند.



شکل ۱: در این گراف، connectivity برابر چهار است. اگر یال‌ها جهت دار نبودند این مولفه برابر یک می‌شد

همچنین گراف‌های جهت داری داریم که دور ندارند و به DAG مشهورند. در این گراف‌ها اگر از راس یک به راس دو مسیری وجود داشته باشد، قطعا از دومی به اولی مسیری نداریم. در رسم این گراف،راس‌هایی که یالی به آنها وارد نمی‌شود را بالا می‌کشیم و سایر راس‌ها را زیر آنها و شبیه‌به درخت می‌کشیم. می‌توان DAG را با سطح‌بندی هم کشید به این‌صورت که در برای رسم سطح‌های زیر یک سطح، در نظر می‌گیریم که با حذف سطح بالایی، کدام بچه‌ها ورودی‌های کمتری دارند و بدین صورت بچه‌های با ورودی‌های کمتر در سطح بالاتر قرار می‌گیرند (شکل ۲). این روش خوبی‌اش این است که هم گرافیکی جهت‌یال ها را می‌بینیم و نیازی به نشان دادن صریح جهت یال نیست (انگار گراف را آویزان کرده‌ایم) و هم اینکه عمق هر راس مشخص میشه یعنی آنکه وقتی از بالا شروع کنیم، چه زمانی به آن راس می‌رسیم.



شکل ۲: تفاوت در ظاهر DAG با اعمال سطح‌بندی