درس‌نامه الگوریتم‌های گراف

نویسنده: محمد جواد عبدالهی

با نظارت: دکتر بهناز عمومی

۲۳ تیر ۱۴۰۴

فهرست مطالب

پیشگفتار

درس «الگوریتم‌های گراف» یکی از زیباترین و کاربردی‌ترین مباحث در علوم ریاضی و کامپیوتر است. گراف‌ها به عنوان یک مدل ریاضی قدرتمند، در همه جا حضور دارند؛ از شبکه‌های اجتماعی و مسیریابی آنلاین گرفته تا تحلیل‌های زیستی و بهینه‌سازی فرآیندها. با این حال، این درس اغلب به عنوان یک مبحث صرفاً تئوری و انتزاعی شناخته می‌شود. هدف اصلی از تهیه این مجموعه، ایجاد پلی میان دنیای تئوری و کاربرد عملی این علم است.

برای رسیدن به این هدف، هر الگوریتم در یک ساختار استاندارد و ده‌بخشی ارائه شده است تا یک مسیر یادگیری کامل و شفاف را فراهم کند:

۱. **توضیح الگوریتم:** برای درک مفهومی و شهودی ایده اصلی.

۲. **اثبات درستی:** برای تضمین صحت عملکرد و درک عمیق منطق حاکم بر الگوریتم.

۳. **شبه‌کد:** به عنوان یک نقشه راه دقیق برای پیاده‌سازی.

۴. **پیچیدگی:** بررسی پیچیدگی زمانی و حافظه الگوریتم.

۵. **مثال:** برای مشاهده عملی روند اجرای الگوریتم روی یک نمونه ساده.

۶. **انیمیشن و ابزارهای پایتون:** برای مشاهده زنده و تعاملی الگوریتم و ساختمان داده‌های مرتبط با آن.

۷. **کاربردها:** برای درک اهمیت الگوریتم در دنیای واقعی. مطالعه این بخش می‌تواند الهام‌بخش شما برای پروژه‌های آینده و مطالعات بیشتر باشد.

۸. **تمرین:** برای درک بهتر، تمرین‌های چالشی طراحی شده‌اند که به شما در رسیدن به فهم عمیق از مطالب کمک می‌کنند.

۹. **منابع برای مطالعه بیشتر:** برای هدایت شما به سمت منابع معتبر جهت عمیق‌تر شدن در مباحث.

۱۰. **اسکریپت های پایتون:** در کنار جزوه برای هر الگوریتم یک اسکریپت پایتون تهیه شده است و با تغییر در گرافی که ابتدای اسکریپت تعریف شده می‌توانید انیمیشن الگوریتم روی گراف مد نظر خود را مشاهده کنید.

آخر هر فصل مجموعه‌ای از سوالات حل‌شده و تمارین بیشتر در سه سطح(آسان_متوسط_سخت) برای تسلط بیشتر و یادگیری مطالب فراتر از جزوه جمع‌آوری شده است.

این مجموعه یک پروژه زنده و در حال رشد است. پیشنهادات و انتقادات شما برای بهبود آن بسیار ارزشمند است. می‌توانید از طریق بخش‌های Issues و Discussions در صفحه گیت‌هاب پروژه یا راه‌های ارتباطی گفته شده در صفحه گیت‌هاب با بنده در ارتباط باشید. همچنین اگر این مجموعه برای شما مفید بوده است، باعث دلگرمی من خواهد بود که با ستاره دادن (Starring) به ریپازیتوری پروژه در گیت‌هاب، از این کار حمایت کنید.

در پایان، لازم می‌دانم از راهنمایی‌ها و نظارت‌های استاد گرانقدر، سرکار خانم دکتر بهناز عمومی، که در شکل‌گیری این اثر نقش بسزایی داشتند، صمیمانه سپاسگزاری کنم.

با آرزوی موفقیت برای شما در این سفر یادگیری،
محمد جواد عبدالهی
تابستان ۱۴۰۴

فصل ۰

مفاهیم پایه و تعاریف از دروس پیش‌نیاز

در این فصل، مفاهیم و ابزارهای اولیه‌ای را که از دروس دیگر به آن‌ها نیاز داریم، به صورت موضوعی مرور می‌کنیم.

۱.۰) مفاهیم درس نظریه گراف

در این بخش، به تعریف ریاضیاتی گراف و روش‌های نمایش آن که پایه و اساس تمام مباحث ماست، می‌پردازیم.

۱.۱.۰) تعریف رسمی گراف

به طور رسمی، یک **گراف** G یک سه‌تایی مرتب $G = (V, E, \psi)$ است که در آن:

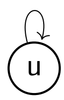
- V یک مجموعه متناهی و ناتهی از عناصر به نام **رئوس** (Vertices) است.
- E یک مجموعه متناهی (و مجزا از V) از عناصر به نام **یال‌ها** (Edges) است.
- ψ یک **تابع وقوع** (incidence function) است که هر یال را به یک زوج (نامرتب) از رئوس نگاشت می‌دهد.

۲.۱.۰) انواع گراف‌ها و یال‌ها

در ادامه به بررسی چند مفهوم کلیدی که انواع گراف‌ها را مشخص می‌کنند، می‌پردازیم.

۱.۲.۱.۰) طوقه (Loop)

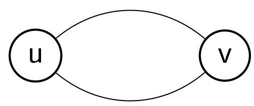
یالی که هر دو سر آن به یک رأس یکسان متصل است.



شکل ۱: نمایش یک طوقه روی رأس u.

۲.۲.۱.۰) یال‌های موازی (Parallel Edges)

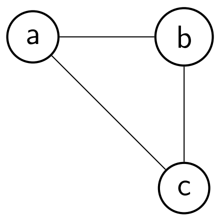
دو یا چند یال که دقیقاً دو رأس یکسان را به هم متصل می‌کنند.



شکل ۲: نمایش دو یال موازی بین رئوس u و v.

۳.۲.۱.۰) گراف ساده (Simple Graph)

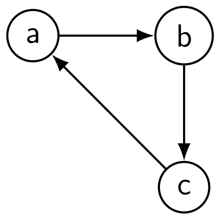
گرافی است که هیچ طوقه یا یال موازی ندارد. در اکثر الگوریتم‌ها، فرض پیش‌فرض ما کار با گراف‌های ساده است.



شکل ۳: نمایش یک گراف ساده.

۴.۲.۱.۰) گراف جهت‌دار (Directed Graph)

گرافی است که یال‌های آن دارای جهت هستند. در این حالت، تابع وقوع هر یال را به یک **زوج مرتب** از رئوس نگاشت می‌دهد.



شکل ۴: نمایش یک گراف جهت‌دار (Digraph).

۵.۲.۱.۰) مسیر (Path) و همبندی

یک **مسیر** در گراف، دنباله‌ای از رئوس به صورت $P = (v_0, v_1, \ldots, v_k)$ است به طوری که برای هر i از ۱ تا k، یال (v_{i-1}, v_i) در گراف وجود داشته باشد. **طول** این مسیر برابر k (**تعداد یال‌ها**) است.

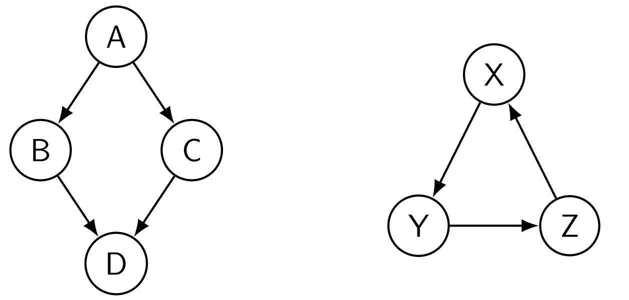
- اگر بین دو رأس u و v مسیری وجود داشته باشد، می‌گوییم v از u **قابل دسترس (reachable)** است.
- یک گراف بی‌جهت **همبند (Connected)** است اگر بین هر دو رأس دلخواه آن مسیری وجود داشته باشد.

۶.۲.۱.۰) دور (Cycle) و گراف غیرمدور

اگر یک مسیر از یک رأس شروع شده و به همان رأس ختم شود، یک **دور (Cycle)** نامیده می‌شود. گرافی که هیچ دوری نداشته باشد، **غیرمدور (Acyclic)** است.

۷.۲.۱.۰) گراف جهت‌دار غیرمدور (DAG)

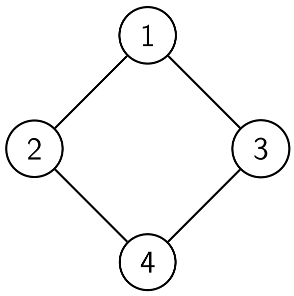
یک **گراف جهت‌دار غیرمدور** (Directed Acyclic Graph) که به اختصار DAG نامیده می‌شود، یک گراف جهت‌دار است که هیچ دور جهت‌داری در آن وجود ندارد. این نوع گراف‌ها برای مدل‌سازی وظایفی که دارای پیش‌نیازی هستند (مانند ترتیب دروس دانشگاهی یا مراحل کامپایل یک پروژه) بسیار پرکاربرد هستند.



شکل ۵: مقایسه یک DAG با یک گراف دارای دور.

۳.۱.۰) ماتریس مجاورت (Adjacency Matrix):

در این روش، گراف با یک ماتریس $n \times n$ که ($n = |V|$) به نام A نمایش داده می‌شود. درایه A_{ij} برابر ۱ است اگر یالی بین رأس i و رأس j وجود داشته باشد و در غیر این صورت برابر ۰ است. این روش برای گراف‌های متراکم (پر یال) کارآمد است اما برای گراف‌های خلوت، حافظه زیادی مصرف می‌کند



A	۱	۲	۳	۴
۱	۰	۱	۱	۰
۲	۱	۰	۰	۱
۳	۱	۰	۰	۱
۴	۰	۱	۱	۰

شکل ۶: یک گراف نمونه و ماتریس مجاورت متناظر با آن.

۱.۳.۱.۰) گشت، گذر، مسیر و دور

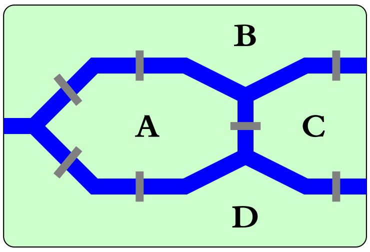
برای توصیف حرکت روی یک گراف، از این اصطلاحات دقیق استفاده می‌کنیم:

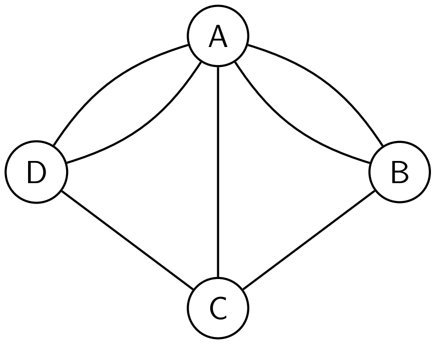
- **گشت (Walk):** دنباله‌ای متناوب از رئوس و یال‌هاست $(v_0, e_1, v_1, \ldots, e_k, v_k)$. در یک گشت، تکرار رئوس و یال‌ها مجاز است.
- **گذر (Trail):** یک گشت است که در آن **هیچ یالی تکرار نمی‌شود**.
- **مسیر (Path):** یک گذر است که در آن **هیچ رأسی تکرار نمی‌شود** (به جز رأس شروع و پایان در مسیر بسته).
- **دور (Cycle/Circuit):** یک مسیر بسته است که طول آن بزرگ‌تر از صفر است.

- **تور (Tour)**: یک گذر بسته است (تکرار رأس مجاز است).

۴.۱.۰) مسئله پل‌های کونیگسبرگ: تولد نظریه گراف

در قرن هجدهم، اهالی شهر کونیگسبرگ با یک معما روبرو بودند: آیا می‌توان از خانه‌ای شروع کرد، از تمام هفت پل شهر دقیقاً یک بار عبور کرد و به نقطه شروع بازگشت؟ این مسئله که به نظر یک سرگرمی می‌آمد، توسط لئونارد اویلر در سال ۱۷۳۶ به صورت یک مسئله ریاضی مدل‌سازی شد و سنگ بنای نظریه گراف را گذاشت. اویلر شهر را به یک گراف تبدیل کرد: هر خشکی (دو جزیره و دو ساحل) به یک رأس و هر پل به یک یال تبدیل شد. مسئله اکنون به این تبدیل شد که آیا می‌توان در این گراف، یک «تور اویلری» پیدا کرد؟

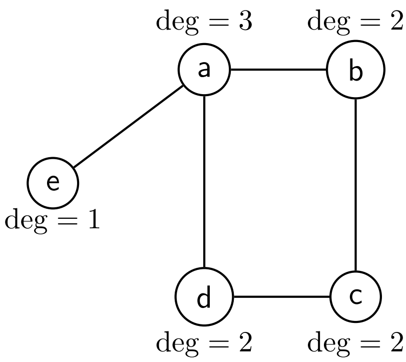




شکل ۷: نقشه شهر کونیگسبرگ (چپ) و مدل گراف متناظر با آن (راست).

۱.۴.۱.۰) درجه و همسایگی رأس

- **همسایگی (Neighborhood)**: مجموعه همسایگان یک رأس v، که با $N(v)$ نمایش داده می‌شود، مجموعه‌ای از تمام رئوسی است که با یک یال به v متصل هستند.
- **درجه (Degree)**: درجه یک رأس v، که با $\deg(v)$ یا $d(v)$ نمایش داده می‌شود، تعداد یال‌هایی است که به آن متصل هستند (طوقه‌ها معمولاً دو بار شمرده می‌شوند). برای گراف‌های ساده، $\deg(v) = |N(v)|$.
- **درجه کمینه (Minimum Degree)**: کمترین درجه در میان تمام رئوس یک گراف G را درجه کمینه آن می‌نامند و با $\delta(G)$ نمایش می‌دهند. $\delta(G) = \min_{v\in V} \deg(v)$.
- **درجه بیشینه (Maximum Degree)**: بیشترین درجه در میان تمام رئوس یک گراف G را درجه بیشینه آن می‌نامند و با $\Delta(G)$ نمایش می‌دهند. $\Delta(G) = \max_{v\in V} \deg(v)$.

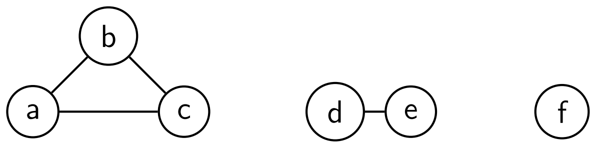


شکل ۸: یک گراف نمونه با درجه هر رأس. در این گراف، $\delta(G) = 1$ و $\Delta(G) = 3$.

۲.۴.۱.۰) همبندی و مؤلفه‌های همبند

یک **مسیر (Path)** در گراف، دنباله‌ای از رئوس به صورت $P = (v_0, v_1, \dots, v_k)$ است به طوری که برای هر i از ۱ تا k، یال (v_{i-1}, v_i) در گراف وجود داشته باشد.

- اگر بین دو رأس u و v مسیری وجود داشته باشد، می‌گوییم v از u **قابل دسترس (reachable)** است.
- یک گراف بی‌جهت **همبند (Connected)** است اگر بین هر دو رأس دلخواه آن مسیری وجود داشته باشد.
- یک **مؤلفه همبند (Connected Component)** از یک گراف، یک زیرگراف همبند بیشینه (maximal) است. به زبان ساده‌تر، مجموعه‌ای از رئوس است که همگی به یکدیگر مسیر دارند.
- تعداد مؤلفه‌های همبند در یک گراف G را با نماد $\omega(G)$ نمایش می‌دهیم.



شکل ۹: یک گراف ناهمبند با سه مؤلفه همبند. برای این گراف، $\omega(G) = 3$.

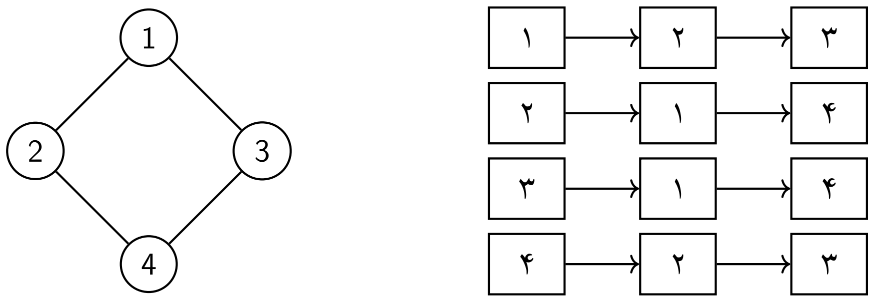
۲.۰) مفاهیم درس ساختمان داده

۱.۲.۰) نمایش گراف در کامپیوتر

برای پیاده‌سازی الگوریتم‌ها، باید گراف را در حافظه ذخیره کنیم.

۱.۱.۲.۰) لیست مجاورت (Adjacency List)

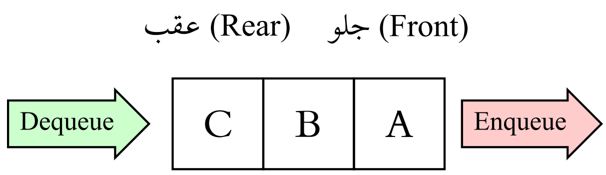
در این روش، یک آرایه از لیست‌ها به اندازه n ایجاد می‌شود. خانه iــام این آرایه، به لیستی از همسایگان رأس i اشاره می‌کند. این روش برای گراف‌های خلوت (کم یال) بسیار بهینه و متداول است.



شکل ۱۰: همان گراف نمونه شکل ۶ و لیست مجاورت متناظر با آن.

۲.۲.۰) صف (Queue)

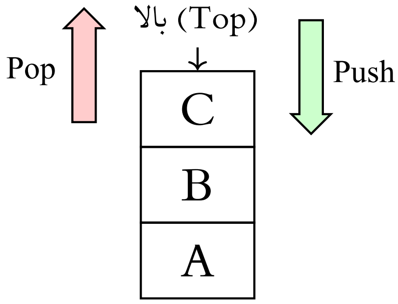
یک **صف** از اصل **ورودی اول، خروجی اول** (First-In, First-Out - FIFO) پیروی می‌کند. عملیات اصلی آن Enqueue (افزودن به انتها) و Dequeue (حذف از ابتدا) است و اساس کار الگوریتم BFS می‌باشد.



شکل ۱۱: نمایش گرافیکی یک صف.

۳.۲.۰) پشته (Stack)

یک **پشته** از اصل **ورودی آخر، خروجی اول** (Last-In, First-Out - LIFO) پیروی می‌کند. عملیات اصلی آن Push (افزودن به بالا) و Pop (حذف از بالا) است و اساس کار الگوریتم DFS می‌باشد.



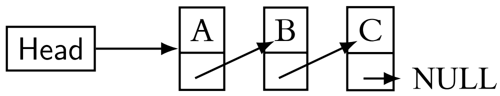
شکل ۱۲: نمایش گرافیکی یک پشته.

۴.۲.۰) لیست پیوندی (Linked List)

یک **لیست پیوندی** یک ساختمان داده خطی است که در آن عناصر در حافظه به صورت پراکنده ذخیره می‌شوند. هر عنصر، که یک **گره (Node)** نامیده می‌شود، از دو بخش تشکیل شده است:

- **داده (Data):** مقداری که گره نگهداری می‌کند.
- **اشاره‌گر (Pointer/Next):** آدرس گره بعدی در توالی را ذخیره می‌کند. اشاره‌گر آخرین گره به یک مقدار تهی (NULL) اشاره دارد.

مزیت اصلی لیست پیوندی، افزودن و حذف آسان عناصر (به خصوص از ابتدا) بدون نیاز به جابجایی سایر عناصر است. این همان دلیلی است که در الگوریتم مرتب‌سازی توپولوژیک مبتنی بر DFS از آن استفاده شد.



شکل ۱۳: نمایش گرافیکی یک لیست پیوندی ساده با سه گره.

۳.۰) مقدمه‌ای بر الگوریتم‌ها

۱.۳.۰) تحلیل کارایی و پیچیدگی الگوریتم‌ها

وقتی یک مسئله را با الگوریتمی حل می‌کنیم، دو سوال مهم مطرح می‌شود: این الگوریتم چقدر **سریع** است و چقدر **حافظه** مصرف می‌کند؟ تحلیل پیچیدگی، یک روش استاندارد برای پاسخ به این سوالات به صورت مستقل از سخت‌افزار و زبان برنامه‌نویسی است. هدف ما اندازه‌گیری زمان به ثانیه نیست، بلکه بررسی **نرخ رشد** منابع مورد نیاز (زمان یا حافظه) با افزایش **اندازه ورودی** است.

۱.۱.۳.۰) پیچیدگی زمانی (Time Complexity)

پیچیدگی زمانی، معیاری برای سنجش تعداد عملیات پایه‌ای است که یک الگوریتم برای ورودی با اندازه n انجام می‌دهد. ما به دنبال یک رابطه بین اندازه ورودی (n) و تعداد مراحل اجرای الگوریتم هستیم. برای مثال، اگر ورودی ما یک گراف باشد، اندازه ورودی معمولاً با تعداد رئوس ($|V|$) و تعداد یال‌ها ($|E|$) تعریف می‌شود.

۲.۱.۳.۰) پیچیدگی حافظه (Space Complexity)

پیچیدگی حافظه، معیاری برای سنجش مقدار حافظه اضافی (Auxiliary Space) است که یک الگوریتم برای ورودی با اندازه n نیاز دارد. منظور از حافظه اضافی، فضایی است که خود الگوریتم برای متغیرها، ساختمان داده‌های کمکی (مانند صف یا پشته) و پشته فراخوانی بازگشتی استفاده می‌کند و شامل فضایی که خود ورودی اشغال کرده، نمی‌شود.

۳.۱.۳.۰) نماد O بزرگ (Big O Notation)

نماد O بزرگ، زبان مشترک ما برای بیان پیچیدگی است. این نماد یک **کران بالا (Upper Bound)** برای نرخ رشد تابع هزینه در **بدترین حالت (Worst Case)** ارائه می‌دهد. ایده اصلی O بزرگ این است که روی رفتار الگوریتم برای ورودی‌های بسیار بزرگ ($n \to \infty$) تمرکز می‌کند و از جزئیات کم‌اهمیت صرف نظر می‌کند. به طور مشخص:

- **ضرایب ثابت نادیده گرفته می‌شوند:** $O(2n)$ و $O(n)$ هر دو یکسان هستند، چون ضریب ۲ در مقیاس‌های بزرگ اهمیت ندارد.
- **جملات با رشد کمتر حذف می‌شوند:** در عبارتی مانند $n^2 + 100n + 50000$، با بزرگ شدن n، جمله n^2 بر بقیه جملات غالب می‌شود. بنابراین پیچیدگی آن $O(n^2)$ است.

۴.۱.۳.۰) کلاس‌های رایج پیچیدگی

در زیر چند کلاس معروف پیچیدگی از سریع‌ترین به کندترین، به همراه یک مثال شهودی آمده است:

$O(1)$ – **پیچیدگی ثابت:** زمان اجرا به اندازه ورودی بستگی ندارد. مثال: دسترسی به عنصر سوم یک آرایه.

$O(\log n)$ – **لگاریتمی:** بسیار سریع. با هر مرحله، اندازه مسئله به طور قابل توجهی کاهش می‌یابد.
مثال: الگوریتم جستجوی دودویی (Binary Search).

$O(n)$ – **خطی:** زمان اجرا به صورت خطی با اندازه ورودی افزایش می‌یابد.
مثال: پیدا کردن بزرگترین عنصر در یک لیست نامرتب.

$O(n \log n)$ – **خطی-لگاریتمی:** پیچیدگی بسیاری از الگوریتم‌های مرتب‌سازی بهینه مانند Merge Sort.

$O(n^2)$ – **مربعی:** معمولاً در الگوریتم‌هایی با دو حلقه تودرتو دیده می‌شود.

مثال: مقایسه هر زوج از عناصر در یک لیست.

$O(2^n)$ – **نمایی:** بسیار کند و فقط برای ورودی‌های کوچک قابل استفاده است.

مثال: پیدا کردن تمام زیرمجموعه‌های ممکن از یک مجموعه.

$O(n!)$ – **فاکتوریل:** کندترین حالت ممکن که با بررسی تمام جایگشت‌های ممکن از ورودی به وجود می‌آید.

مثال: حل مسئله فروشنده دوره‌گرد با روش brute-force.

در تحلیل الگوریتم‌های گراف، معمولاً با پیچیدگی‌هایی مانند $O(|V| + |E|)$ یا $O(|V|^2)$ روبرو می‌شویم که نشان‌دهنده وابستگی الگوریتم به تعداد رئوس و یال‌هاست.

۲.۳.۰) کلاس پیچیدگی P (مسائل آسان)

کلاس P شامل تمام **مسائل تصمیم‌گیری (Decision Problems)** است که می‌توان آن‌ها را در **زمان چندجمله‌ای (Polynomial Time)** حل کرد. به عبارت دیگر، اگر اندازه ورودی یک مسئله n باشد، الگوریتمی وجود دارد که آن را در زمان $O(n^k)$ برای یک مقدار ثابت k حل می‌کند. به طور شهودی، مسائل کلاس P مسائلی هستند که ما برای آن‌ها راه‌حل‌های «سریع» و «کارآمد» می‌شناسیم.

- **مثال:** مسئله «آیا در یک گراف وزن‌دار، مسیری با وزن کمتر از k بین دو رأس u و v وجود دارد؟». این مسئله با الگوریتم دایکسترا در زمان چندجمله‌ای قابل حل است، پس در کلاس P قرار دارد.

۳.۳.۰) کلاس پیچیدگی NP (مسائلی که امتحان کردنشان آسان است)

کلاس NP (Nondeterministic Polynomial time) شامل تمام مسائل تصمیم‌گیری است که اگر یک «گواه» یا «راه‌حل پیشنهادی» برای آن‌ها به ما داده شود، می‌توانیم درستی آن را در **زمان چندجمله‌ای** بررسی کنیم. به عبارت دیگر، حل کردن مسئله ممکن است بسیار سخت باشد، اما **تایید کردن یک جواب پیشنهادی** برای آن آسان است.

- **مثال:** مسئله «آیا یک گراف، دور هامیلتونی دارد؟». پیدا کردن خودِ دور هامیلتونی بسیار سخت است. اما اگر کسی یک دنباله از رئوس را به عنوان یک دور هامیلتونی به ما بدهد، ما می‌توانیم به سادگی و در زمان چندجمله‌ای بررسی کنیم که آیا این دنباله واقعاً یک دور است و از تمام رئوس عبور می‌کند یا خیر. پس این مسئله در کلاس NP قرار دارد.

نکته مهم: هر مسئله‌ای که در کلاس P باشد، قطعاً در کلاس NP نیز هست. چرا؟ چون اگر بتوانیم مسئله‌ای را در زمان چندجمله‌ای حل کنیم، قطعاً می‌توانیم جواب پیشنهادی آن را نیز (با نادیده گرفتن جواب و حل دوباره مسئله) در زمان چندجمله‌ای تایید کنیم. بنابراین، $P \subseteq NP$.

۴.۳.۰) مسئله P در مقابل NP

یکی از بزرگترین و مهم‌ترین سوالات حل‌نشده در علوم کامپیوتر این است که آیا $P = NP$ است یا خیر؟ به عبارت دیگر، آیا هر مسئله‌ای که جوابش به سرعت قابل تایید است، به سرعت نیز قابل حل است؟ شهود عمومی بر این است که این دو کلاس با هم برابر نیستند، اما این موضوع هنوز اثبات نشده است.

۵.۳.۰) مسائل NP-Hard و NP-Complete

برای درک این دو مفهوم، ابتدا باید مفهوم «کاهش پذیری» را بشناسیم. می‌گوییم مسئله A به مسئله B «کاهش‌پذیر» است، اگر بتوانیم هر نمونه از مسئله A را با یک الگوریتم زمان چندجمله‌ای، به نمونه‌ای از مسئله B تبدیل کنیم، به طوری که جواب هر دو نمونه یکسان باشد.

- **کلاس NP-Complete:** این کلاس شامل **سخت‌ترین مسائل** در کلاس NP است. یک مسئله، NP-Complete است اگر:

  ۱. خود مسئله در کلاس NP باشد.

  ۲. هر مسئله دیگری در کلاس NP به آن کاهش‌پذیر باشد.

  اگر بتوانید فقط یکی از این مسائل را در زمان چندجمله‌ای حل کنید، در واقع تمام مسائل NP را حل کرده‌اید و ثابت کرده‌اید که $P = NP$.

  – **مثال:** مسئله دور هامیلتونی و مسئله صدق‌پذیری بولی (SAT).

- **کلاس NP-Hard:** این کلاس شامل مسائلی است که **حداقل به اندازه سخت‌ترین مسائل NP**، سخت هستند. یک مسئله NP-Hard است اگر هر مسئله‌ای در NP به آن کاهش‌پذیر باشد. تفاوت کلیدی این است که یک مسئله NP-Hard لزوماً نباید در کلاس NP باشد.

  – **مثال:** مسئله فروشنده دوره‌گرد (TSP) در حالت بهینه‌سازی (پیدا کردن کوتاه‌ترین تور) یک مسئله NP-Hard است.

۶.۳.۰) پارادایم‌های طراحی الگوریتم

بسیاری از الگوریتم‌های کارآمد، از یک استراتژی یا «پارادایم» طراحی مشخص پیروی می‌کنند. آشنایی با این پارادایم‌ها به ما کمک می‌کند تا برای مسائل جدید، راه‌حل‌های بهینه طراحی کنیم. در ادامه سه پارادایم اصلی را معرفی می‌کنیم.

۱.۶.۳.۰) الگوریتم‌های حریصانه (Greedy Algorithms)

ایده اصلی: در هر مرحله از حل مسئله، انتخابی را انجام بده که در همان لحظه، بهترین گزینه به نظر می‌رسد، به این امید که توالی این انتخاب‌های «محلی بهینه»، در نهایت به یک جواب «بهینه کلی» منجر شود.

- **مثال شهودی:** فرض کنید می‌خواهید با استفاده از کمترین تعداد سکه، مبلغ مشخصی را بپردازید. استراتژی حریصانه این است که همیشه بزرگترین سکه ممکن که از مبلغ باقی‌مانده کمتر است را انتخاب کنید. این روش برای سکه‌های رایج به درستی کار می‌کند.

- **نکته کلیدی:** الگوریتم‌های حریصانه بسیار سریع هستند، اما تضمینی برای رسیدن به جواب بهینه کلی در تمام مسائل وجود ندارد. درستی آن‌ها باید برای هر مسئله به صورت جداگانه اثبات شود.

- **مثال در الگوریتم‌های گراف:** الگوریتم‌های دایکسترا، پریم و کراسکال (که در فصل‌های آینده خواهیم دید) نمونه‌های معروفی از الگوریتم‌های حریصانه هستند که به جواب بهینه می‌رسند.

۲.۶.۳.۰) برنامه‌نویسی پویا (Dynamic Programming)

ایده اصلی: یک مسئله بزرگ را به زیرمسائل کوچکتر و **همپوشان (overlapping)** تقسیم کن، هر زیرمسئله را فقط **یک بار** حل کرده و جواب آن را در یک جدول ذخیره کن تا در آینده اگر دوباره به همان زیرمسئله برخورد کردی، به جای محاسبه مجدد، از جواب آماده استفاده کنی.

- **مثال شهودی:** محاسبه جمله nـام دنباله فیبوناچی. برای محاسبه $fib(5)$، به $fib(4)$ و $fib(3)$ نیاز داریم. برای محاسبه $fib(4)$ نیز دوباره به $fib(3)$ نیاز پیدا می‌کنیم. برنامه‌نویسی پویا $fib(3)$ را یک بار حساب کرده و جوابش را ذخیره می‌کند.
- **ویژگی‌های لازم:** این روش برای مسائلی مناسب است که دارای ویژگی «زیرمسائل همپوشان» و «ساختار بهینه» باشند.
- **مثال در الگوریتم‌های گراف:** الگوریتم فلوید–وارشال برای یافتن کوتاه‌ترین مسیر بین تمام زوج رئوس و الگوریتم Held-Karp برای مسئله فروشنده دوره‌گرد، نمونه‌های قدرتمندی از برنامه‌نویسی پویا هستند.

۳.۶.۳.۰) تقسیم و حل (Divide and Conquer)

ایده اصلی: این پارادایم شامل سه مرحله است:

۱. **تقسیم (Divide):** مسئله اصلی به چند زیرمسئله کوچکتر و **مستقل (independent)** از همان نوع تقسیم می‌شود.
۲. **حل (Conquer):** زیرمسئله‌ها به صورت بازگشتی حل می‌شوند. اگر به اندازه کافی کوچک بودند، مستقیماً حل می‌شوند.
۳. **ترکیب (Combine):** راه‌حل‌های به دست آمده از زیرمسئله‌ها با هم ترکیب شده تا راه‌حل مسئله اصلی ساخته شود.

- **تفاوت کلیدی با برنامه‌نویسی پویا:** در این روش، زیرمسئله‌ها مستقل از هم هستند و همپوشانی ندارند، بنابراین نیازی به ذخیره کردن جواب آن‌ها نیست.
- **مثال کلاسیک:** الگوریتم مرتب‌سازی ادغامی (Merge Sort) که در آن لیست به دو نیم تقسیم شده، هر نیمه به صورت بازگشتی مرتب شده و سپس دو نیمه مرتب‌شده با هم ادغام می‌شوند.

فصل ۱

تکنیک‌های اولیه پیمایش گراف

پس از آشنایی با تعاریف رسمی گراف در فصل قبل، در این فصل به سراغ دو روش بنیادین برای پیمایش رئوس و یال‌های گراف می‌رویم:

جستجوی اول سطح (BFS) و جستجوی اول عمق (DFS). این الگوریتم‌ها سنگ بنای بسیاری از الگوریتم‌های پیشرفته‌تر گراف هستند. ما در این بخش، این الگوریتم‌ها را عمدتاً روی **گراف‌های ساده** توضیح می‌دهیم، اما منطق آن‌ها با استفاده از نمایش **لیست مجاورت** (Adjacency List) (بخش ۱.۱.۲.۰)، برای گراف‌های دارای یال موازی نیز قابل تعمیم است.

۱.۱) جستجوی اول سطح (Breadth-First Search - BFS)

۱.۱.۱) توضیح الگوریتم

جستجوی اول سطح (BFS)، یک الگوریتم پیمایش گراف است که از یک رأس مبدأ (source) مانند s شروع کرده و به صورت لایه‌به‌لایه یا موجی، رئوس گراف را ملاقات می‌کند. به این معنی که ابتدا خود رأس s، سپس تمام همسایگان مستقیم آن (لایه اول)، سپس تمام همسایگانِ همسایگان (لایه دوم) و الی آخر پیمایش می‌شوند. این رفتار موجی، با استفاده از یک ساختمان داده **صف (Queue)** (بخش ۲.۲.۰) پیاده‌سازی می‌شود که از منطق "هر که زودتر وارد شود، زودتر خارج می‌شود" پیروی می‌کند.

۲.۱.۱) اثبات درستی الگوریتم

در این بخش، ویژگی بنیادین الگوریتم BFS، یعنی توانایی آن در پیمایش تمام رئوس قابل دسترس از مبدأ را اثبات می‌کنیم.

حکم: اگر رأسی مانند v از رأس مبدأ s قابل دسترس باشد، آنگاه اجرای BFS از مبدأ s، حتماً رأس v را ملاقات خواهد کرد.

اثبات (با برهان خلف): فرض کنید حکم نادرست باشد. یعنی رأسی مانند v وجود دارد که از s قابل دسترس است، اما BFS پس از شروع از s، آن را ملاقات نمی‌کند. از آنجایی که v از s قابل دسترس است، حداقل یک مسیر از s به v وجود دارد. بیایید کوتاه‌ترین مسیر بین آن‌ها را در نظر بگیریم. طول این مسیر را k می‌نامیم.

- اگر $k = 0$، آنگاه $v = s$ است. BFS همیشه با ملاقات رأس مبدأ شروع می‌کند، پس v قطعاً ملاقات می‌شود.

- اگر $k > 0$، یک رأس u را در نظر بگیرید که در مسیر کوتاه‌ترین از s به v، دقیقاً قبل از v قرار دارد. فاصله کوتاه‌ترین مسیر از s به u برابر $k - 1$ است.

- چون مسیر s به u از مسیر s به v کوتاه‌تر است، طبق فرض ما (که v اولین رأس قابل دسترسی است که ملاقات نشده)، رأس u باید حتماً ملاقات شده باشد.
- هنگامی که الگوریتم رأس u را ملاقات می‌کند، آن را به صف اضافه می‌کند. از آنجایی که صف از منطق FIFO پیروی می‌کند، رأس u در نهایت از صف خارج خواهد شد.
- در لحظه‌ای که الگوریتم رأس u را پردازش می‌کند، تمام همسایگان آن را بررسی می‌کند. از آنجا که v همسایه u است و طبق فرض ما ملاقات نشده، الگوریتم قطعاً v را ملاقات کرده و به صف اضافه می‌کند.
- این نتیجه با فرض ما مبنی بر اینکه v هرگز ملاقات نشده، در **تناقض** است.

بنابراین فرض اولیه ما غلط بوده و حکم اثبات می‌شود. ■

۳.۱.۱) شبه‌کد الگوریتم

الگوریتم ۱ BFS

```
procedure BFS(G, s)
    For each vertex u ∈ V[G], set u.visited = false
    s.visited = true
    Q ← ∅
    Enqueue(Q, s)
    while Q is not empty do
        u ← Dequeue(Q)
        for each neighbor v of u do
            if v.visited == false then
                v.visited = true
                Enqueue(Q, v)
```

۴.۱.۱) تحلیل پیچیدگی زمانی و حافظه

الگوریتم BFS در صورتی که گراف با **لیست مجاورت** نمایش داده شود، یک الگوریتم بسیار بهینه با پیچیدگی خطی است.

جدول ۱.۱: پیچیدگی الگوریتم BFS

نوع پیچیدگی	مقدار
پیچیدگی زمانی	$O(\|V\| + \|E\|)$
پیچیدگی حافظه	$O(\|V\|)$

توضیح پیچیدگی الگوریتم:

- **پیچیدگی زمانی** ($O(|V| + |E|)$):

۱. هر رأس در گراف دقیقاً **یک بار** وارد صف (Enqueue) و دقیقاً **یک بار** از آن خارج (Dequeue) می‌شود. این بخش از عملیات در مجموع $O(|V|)$ هزینه دارد.

۲. هنگامی که یک رأس از صف خارج می‌شود، ما لیست مجاورت آن را برای پیدا کردن همسایگانش پیمایش می‌کنیم. در کل اجرای الگوریتم، لیست مجاورت هر رأس دقیقاً یک بار پیمایش می‌شود. مجموع طول تمام لیست‌های مجاورت در یک گراف برابر با $O(|E|)$ است.

بنابراین، هزینه کل الگوریتم از جمع این دو بخش، یعنی $O(|V| + |E|)$، به دست می‌آید.

- **پیچیدگی حافظه ($O(|V|)$):** حافظه اضافی مورد نیاز الگوریتم برای موارد زیر است:

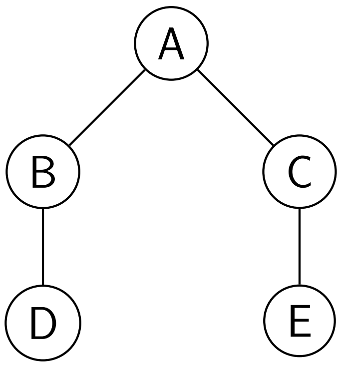
– یک آرایه برای نگهداری وضعیت ملاقات هر رأس (یا فاصله آن از مبدأ) که $O(|V|)$ فضا نیاز دارد.

– خودِ صف. در بدترین حالت، ممکن است تمام رئوس یک لایه از گراف به صورت همزمان در صف قرار بگیرند. برای مثال در یک گراف ستاره‌ای، پس از پردازش رأس مرکزی، تمام $V - 1$ رأس دیگر وارد صف می‌شوند. بنابراین، صف به $O(|V|)$ فضا نیاز دارد.

در نتیجه، پیچیدگی حافظه الگوریتم $O(|V|)$ است.

۵.۱.۱) مثال

گراف ساده و بدون جهت زیر را در نظر بگیرید. پیمایش BFS را از رأس A شروع می‌کنیم.



شکل ۱.۱: یک گراف ساده و بدون جهت برای پیمایش BFS از رأس A.

پیمایش به صورت لایه‌به‌لایه خواهد بود: ابتدا رأس A (لایه ۰)، سپس همسایگان آن B و C (لایه ۱)، و در نهایت رئوس D و E (لایه ۲). بنابراین یک ترتیب ملاقات ممکن **A, B, C, D, E** است.

۶.۱.۱) انیمیشن و ابزارهای پایتون

اسکریپت پایتون bfs_animation.py (موجود در پوشه chapter-1) یک انیمیشن دو قسمتی تولید می‌کند که در یک سمت، پیمایش گراف و در سمت دیگر، وضعیت لحظه‌ای **صف (Queue)** را نمایش می‌دهد.

۷.۱.۱) کاربردهای الگوریتم

ویژگی پیمایش لایه‌به‌لایه، BFS را به ابزاری قدرتمند برای کاربردهای زیر تبدیل کرده است که در ادامه درس‌نامه به آن‌ها خواهیم پرداخت:

- **یافتن کوتاه‌ترین مسیر:** در گراف‌های بدون وزن، مسیر با کمترین تعداد یال را پیدا می‌کند.
- **تست دوبخشی بودن گراف (Bipartite Testing):** با رنگ‌آمیزی متناوب لایه‌ها.
- **یافتن مؤلفه‌های همبندی (Connected Components):** در گراف‌های بی‌جهت.

۸.۱.۱ تمرین

۱. **مسیر شوالیه:** یک صفحه شطرنج 8×8 را در نظر بگیرید. یک اسب (شوالیه) در یک خانه مشخص قرار دارد. برنامه‌ای بنویسید که با مدل‌سازی این صفحه به عنوان یک گراف، کوتاه‌ترین مسیر (کمترین تعداد حرکت) برای رسیدن اسب به یک خانه مقصد را پیدا کند. (راهنمایی: هر خانه یک رأس و هر حرکت ممکن اسب یک یال است).

۲. **قطر درخت:** قطر یک درخت، طولانی‌ترین مسیر بین هر دو رأس دلخواه در آن درخت است. الگوریتمی ارائه دهید که با دو بار اجرای BFS، قطر یک درخت را پیدا کند. درستی الگوریتم خود را اثبات کنید. (راهنمایی: ابتدا BFS را از یک رأس دلخواه اجرا کنید...)

۳. **اثبات ویژگی گراف دوبخشی:** همانطور که در کاربردها اشاره شد، BFS برای تشخیص دوبخشی بودن گراف استفاده می‌شود. اثبات کنید که یک گراف، دوبخشی است اگر و تنها اگر هیچ دور با طول فرد نداشته باشد.

۹.۱.۱ منابع برای مطالعه بیشتر

- **Algorithm Design (KT):** Chapter 3, Section 3.2

۲.۱) جستجوی اول عمق (Depth-First Search - DFS)

۱.۲.۱) توضیح الگوریتم

جستجوی اول عمق (DFS)، یک روش پیمایش گراف است که استراتژی آن، حرکت در عمق یک مسیر تا رسیدن به بن‌بست و سپس بازگشت به عقب (Backtracking) برای کاوش مسیرهای دیگر است. این رفتار بازگشتی، به طور طبیعی با استفاده از یک ساختمان داده **پشته (Stack)** (بخش ۳.۲.۰) پیاده‌سازی می‌شود. در نسخه بازگشتی الگوریتم، پشته همان پشته فراخوانی سیستم است. برای درک دقیق‌تر عملکرد، می‌توان رئوس را در سه حالت در نظر گرفت: **سفید** (ملاقات نشده)، **خاکستری** (در حال پیمایش) و **سیاه** (پیمایش شده).

۲.۲.۱) اثبات درستی الگوریتم

در این بخش، ویژگی بنیادین الگوریتم DFS یعنی توانایی آن در پیمایش تمام گراف را اثبات می‌کنیم.

حکم: اگر رأسی مانند v از رأس u قابل دسترس باشد، آنگاه اجرای DFS از مبدأ u، حتماً رأس v را ملاقات خواهد کرد.

اثبات (با برهان خلف): فرض کنید حکم نادرست باشد. یعنی رأسی مانند v وجود دارد که از u قابل دسترس است، اما DFS پس از شروع از u، آن را ملاقات نمی‌کند (سفید باقی می‌ماند). از آنجایی که v از u قابل دسترس است، حداقل یک مسیر از u به v وجود دارد. بیایید این مسیر را $P = (u, p_1, p_2, \ldots, p_k, v)$ در نظر بگیریم. در این مسیر، رأس u قطعاً ملاقات شده است (چون مبدأ است) و رأس v ملاقات نشده است. بنابراین، باید اولین رأسی در این مسیر وجود داشته باشد که ملاقات نشده است؛ آن را p_i می‌نامیم. پس رأس قبلی آن، p_{i-1}، ملاقات شده است. اما طبق تعریف الگوریتم DFS، هنگامی که پیمایش به رأس p_{i-1} می‌رسد، الگوریتم تمام همسایگان «سفید» آن را بررسی و به صورت بازگشتی ملاقات می‌کند. از آنجایی که p_i همسایه p_{i-1} است و طبق فرض ما سفید است، الگوریتم باید DFS-Visit(G, p_i) را فراخوانی کند. این فراخوانی باعث می‌شود p_i ملاقات شود. این نتیجه با فرض ما مبنی بر اینکه p_i هرگز ملاقات نشده، در **تناقض** است. بنابراین فرض اولیه ما غلط بوده و حکم اثبات می‌شود. ■

۳.۲.۱) شبه‌کد الگوریتم

الگوریتم ۲ DFS – نسخه بازگشتی

```
procedure DFS-Visit(G, u)
    Mark u as visited (e.g., set color to gray)
    Process u
    for each neighbor v of u do
        if v has not been visited then
            DFS-Visit(G, v)
    Mark u as finished (e.g., set color to black)
```

۴.۲.۱) تحلیل پیچیدگی زمانی و حافظه

همانند BFS، الگوریتم DFS نیز در صورت استفاده از نمایش **لیست مجاورت**، یک الگوریتم بسیار کارآمد با پیچیدگی زمانی خطی است.

جدول ۲.۱: پیچیدگی الگوریتم DFS

نوع پیچیدگی	مقدار
پیچیدگی زمانی	$O(\|V\| + \|E\|)$
پیچیدگی حافظه	$O(\|V\|)$

توضیح شهودی پیچیدگی‌ها:

- **پیچیدگی زمانی** ($O(|V| + |E|)$):

۱. روال اصلی الگوریتم، یک حلقه روی تمام رئوس گراف دارد. روال کمکی DFS-Visit برای هر رأس دقیقاً **یک بار** فراخوانی می‌شود، زیرا پس از اولین فراخوانی، آن رأس به عنوان ملاقات‌شده علامت‌گذاری می‌شود. این بخش در مجموع $O(|V|)$ هزینه دارد.

۲. در تمام فراخوانی‌های DFS-Visit، حلقه for روی لیست مجاورت هر رأس دقیقاً یک بار پیمایش می‌شود. مجموع طول تمام لیست‌های مجاورت در یک گراف برابر با $O(|E|)$ است.

بنابراین، هزینه کل الگوریتم از جمع این دو بخش، یعنی $O(|V| + |E|)$، به دست می‌آید.

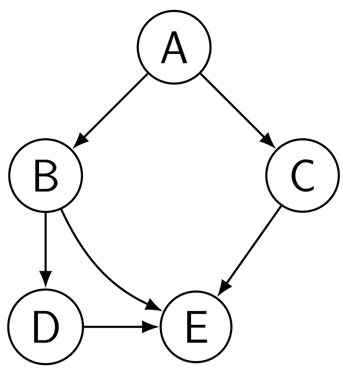
- **پیچیدگی حافظه** ($O(|V|)$): حافظه اضافی مورد نیاز الگوریتم برای موارد زیر است:

  - یک آرایه برای نگهداری وضعیت ملاقات هر رأس که $O(|V|)$ فضا نیاز دارد.
  - **پشته فراخوانی بازگشتی.** در بدترین حالت، برای یک گراف که شبیه به یک زنجیره بلند (لیست پیوندی) است، عمق بازگشتی می‌تواند تا $O(|V|)$ برسد. در این حالت، تمام رئوس به صورت همزمان در پشته فراخوانی سیستم قرار می‌گیرند.

در نتیجه، پیچیدگی حافظه الگوریتم در بدترین حالت، $O(|V|)$ است.

۵.۲.۱) مثال

گراف جهت‌دار زیر را در نظر بگیرید. پیمایش DFS را از رأس A شروع می‌کنیم.



شکل ۲.۱: یک گراف جهت‌دار برای پیمایش DFS.

یک ترتیب ملاقات ممکن (بسته به ترتیب پیمایش همسایه‌ها): **A, B, D, E, C**. در این پیمایش، الگوریتم ابتدا به عمق مسیر A-B-D-E می‌رود، سپس به عقب برگشته و مسیر A-C را کاوش می‌کند.

۶.۲.۱) انیمیشن و ابزارهای پایتون

اسکریپت پایتون dfs_animation.py (موجود در پوشه chapter-1) یک انیمیشن دو قسمتی تولید می‌کند که در یک سمت، پیمایش گراف و در سمت دیگر، وضعیت لحظه‌ای **پشته (Stack)** را نمایش می‌دهد. این انیمیشن به درک عمیق نحوه عملکرد بازگشتی و عقب‌گرد الگوریتم کمک می‌کند.

۷.۲.۱) کاربردهای الگوریتم

پیمایش DFS زیربنای الگوریتم‌های مهمی است. این الگوریتم به تنهایی یک پیمایش است، اما با کمی تغییرات، کاربردهای قدرتمندی پیدا می‌کند که در ادامه درس‌نامه به آن‌ها خواهیم پرداخت:

- **تشخیص دور (Cycle Detection):** با بررسی وجود یال‌های برگشتی.
- **مرتب‌سازی توپولوژیک (Topological Sorting):** با استفاده از زمان پایان رئوس.
- **یافتن مؤلفه‌های قویاً همبند (Strongly Connected Components)**

۸.۲.۱) تمرین

۱. **تشخیص دور در گراف بی‌جهت:** الگوریتم DFS را طوری تغییر دهید که بتواند وجود دور در یک گراف **بی‌جهت** و همبند را تشخیص دهد. (راهنمایی: صرفاً پیدا کردن یک یال که به یک رأس ملاقات‌شده (visited) اشاره دارد کافی نیست. چرا؟ چه تفاوتی با رأس پدر در درخت DFS دارد؟)

۲. **رئوس مفصلی (Articulation Points):** یک رأس مفصلی، رأسی است که حذف آن (به همراه یال‌های متصل به آن) تعداد مؤلفه‌های همبندی گراف را افزایش می‌دهد. الگوریتمی مبتنی بر DFS و زمان‌های کشف/پایان ارائه دهید که تمام رئوس مفصلی یک گراف بی‌جهت را در زمان خطی پیدا کند.

۳. **تعداد مسیرها در DAG:** الگوریتمی بنویسید که تعداد کل مسیرهای مختلف بین دو رأس مشخص s و t را در یک گراف جهت‌دار غیرمدور (DAG) محاسبه کند. (راهنمایی: از یک روش بازگشتی مشابه DFS به همراه برنامه‌نویسی پویا استفاده کنید).

۹.۲.۱) منابع برای مطالعه بیشتر

- **Algorithm Design (KT):** Chapter 3, Section 3.2

۳.۱) مرتب‌سازی توپولوژیک (Topological Sorting)

۱.۳.۱) توضیح الگوریتم

مرتب‌سازی توپولوژیک یک الگوریتم بنیادی برای **گراف‌های جهت‌دار غیرمدور (DAGs)** است. خروجی این الگوریتم، یک **ترتیب خطی** از تمام رئوس گراف است، به طوری که برای هر یال جهت‌دار از رأس u به رأس v، رأس u در این ترتیب، قبل از رأس v آمده باشد.
این الگوریتم برای مدل‌سازی هر نوع وابستگی‌ای که دارای ترتیب است، مانند پیش‌نیازهای درسی، مراحل یک پروژه ساختمانی یا ترتیب کامپایل شدن فایل‌ها، کاربرد حیاتی دارد. شرط لازم برای وجود چنین ترتیبی این است که گراف هیچ دور جهت‌داری نداشته باشد؛ زیرا وجود یک دور (مثلاً $A \to B \to A$) منجر به یک پارادوکس منطقی می‌شود که در آن A باید هم قبل و هم بعد از B قرار بگیرد.
دو رویکرد اصلی برای این مرتب‌سازی وجود دارد: یکی مبتنی بر حذف متوالی رئوس با درجه ورودی صفر (الگوریتم Kahn) و دیگری مبتنی بر پیمایش DFS و ترتیب معکوس زمان پایان رئوس.

۲.۳.۱) اثبات درستی

در این بخش، قضیه بنیادینی را اثبات می‌کنیم که وجود مرتب‌سازی توپولوژیک را به ساختار خود گراف مرتبط می‌کند.
قضیه: یک گراف جهت‌دار G دارای مرتب‌سازی توپولوژیک است، **اگر و تنها اگر** G یک گراف جهت‌دار غیرمدور (DAG) باشد.

اثبات بخش اول ($\Leftarrow$): (اگر گراف مرتب‌سازی توپولوژیک داشته باشد، آنگاه غیرمدور است)

اثبات (با برهان خلف): فرض کنید گراف G هم دارای یک مرتب‌سازی توپولوژیک است و هم دارای یک دور $C : v_1 \to v_2 \to \cdots \to v_k \to v_1$. طبق تعریف مرتب‌سازی توپولوژیک، باید v_1 قبل از v_2 بیاید، v_2 قبل از v_3 و در نهایت v_{k-1} قبل از v_k. این زنجیره ایجاب می‌کند که v_1 باید قبل از v_k بیاید. اما یال (v_k, v_1) نیز در دور وجود دارد که ایجاب می‌کند v_k باید قبل از v_1 بیاید. این دو نتیجه با یکدیگر در **تناقض** هستند. پس حکم اثبات می‌شود. ■

اثبات بخش دوم ($\Rightarrow$): (اگر گراف غیرمدور باشد، آنگاه مرتب‌سازی توپولوژیک دارد)

اثبات (به صورت سازنده): برای اثبات، ابتدا یک لم (قضیه کمکی) کلیدی را ثابت می‌کنیم.
لم: هر گراف جهت‌دار غیرمدور (DAG) متناهی، حداقل یک رأس با درجه ورودی صفر (in-degree = 0) دارد.
اثبات لم (با ایده طولانی‌ترین مسیر): فرض کنید P یکی از طولانی‌ترین مسیرها در گراف G باشد (چون گراف غیرمدور و متناهی است، چنین مسیری وجود دارد). رأس شروع این مسیر را u می‌نامیم. ادعا می‌کنیم درجه ورودی u صفر است. با برهان خلف، فرض کنید درجه ورودی u صفر نباشد. پس رأسی مانند v وجود دارد که یال (v, u) در گراف است. چون گراف DAG است، v نمی‌تواند در مسیر P باشد (وگرنه دور ایجاد می‌شد). بنابراین می‌توانیم مسیر P را با یال (v, u) گسترش دهیم و مسیر جدید و طولانی‌تری بسازیم که این با فرض ما مبنی بر اینکه P طولانی‌ترین مسیر بوده، در **تناقض** است. پس رأس u باید درجه ورودی صفر داشته باشد. ■
حال با استفاده از این لم، یک ترتیب توپولوژیک می‌سازیم:

۱. طبق لم، یک رأس با درجه ورودی صفر مانند u پیدا کرده و آن را به عنوان عنصر بعدی در ترتیب قرار می‌دهیم.

۲. رأس u و یال‌های خروجی از آن را از گراف حذف می‌کنیم. گراف باقی‌مانده همچنان یک DAG است.

۳. این فرآیند را تا زمانی که گرافی باقی نماند، تکرار می‌کنیم.

چون در هر مرحله یک رأس حذف می‌شود، این فرآیند پس از n مرحله به پایان رسیده و یک ترتیب کامل تولید می‌کند. ■

۳.۳.۱) شبه‌کد الگوریتم‌ها

در این بخش، شبه‌کد هر دو رویکرد اصلی برای یافتن ترتیب توپولوژیک را بررسی می‌کنیم.

۱.۳.۳.۱) الگوریتم اول: مبتنی بر درجه ورودی (الگوریتم Kahn)

این الگوریتم به صورت مستقیم از لم «وجود رأس با درجه ورودی صفر» استفاده می‌کند و یک روش شهودی و تکراری است.

الگوریتم ۳ مرتب‌سازی توپولوژیک - الگوریتم Kahn

```
procedure Kahn-Topological-Sort(G)
    Let L be an empty list that will contain the sorted elements.
    Let S be a queue of all nodes with no incoming edge.

    while S is not empty do
        Remove a node u from the front of S.
        Add u to the end of L.

        for each node v with an edge e from u to v do
            Remove edge e from the graph (conceptually).
            if v has no other incoming edges then
                Enqueue v into S.

    if graph has edges then
        return error: graph has at least one cycle.
    else
        return L (A topologically sorted order).
```

توضیح شبه‌کد: الگوریتم با یک صف از تمام رئوسی که درجه ورودی صفر دارند شروع می‌شود. در هر مرحله، یک رأس از صف برداشته، به لیست خروجی اضافه می‌شود و تمام یال‌های خروجی از آن به صورت مفهومی حذف می‌شوند. اگر حذف این یال‌ها باعث شود درجه ورودی همسایگان آن رأس صفر شود، آن همسایگان نیز به صف اضافه می‌شوند. اگر در پایان، تعداد رئوس لیست خروجی کمتر از تعداد کل رئوس گراف باشد، یعنی گراف دارای دور بوده است.

۲.۳.۳.۱) الگوریتم دوم: مبتنی بر جستجوی اول عمق (DFS)

این رویکرد از ویژگی پیمایش DFS استفاده می‌کند. ترتیب معکوس زمان پایان یافتن رئوس در DFS، یک ترتیب توپولوژیک معتبر است.

الگوریتم ۴ مرتب‌سازی توپولوژیک - مبتنی بر DFS

```
procedure TOPOLOGICAL-SORT-DFS(G)
    Let L be an empty linked list that will contain the sorted elements.
    Mark all vertices as unvisited.

    for each vertex u in G do
        if u is unvisited then
            DFS-Visit-Topo(u, L)

    return L
```

الگوریتم ۵ روال کمکی: DFS-Visit-Topo

```
procedure DFS-VISIT-TOPO(u, L)
    Mark u as visited.

    for each neighbor v of u do
        if v is unvisited then
            DFS-Visit-Topo(v, L)

    Prepend u to the list L.                    ▷ Add u to the beginning of the list
```

توضیح شبه‌کد: الگوریتم اصلی، یک پیمایش DFS را روی کل گراف اجرا می‌کند. نکته کلیدی در روال کمکی DFS-Visit-Topo نهفته است: پس از آنکه پیمایش از یک رأس u و تمام نوادگان آن به طور کامل به پایان رسید (یعنی درست قبل از خروج از تابع بازگشتی)، آن رأس به **ابتدای** یک لیست پیوندی اضافه می‌شود. این کار تضمین می‌کند رأسی که دیرتر از همه تمام می‌شود، در ابتدای لیست قرار گیرد و ترتیب توپولوژیک به درستی ساخته شود.

۴.۳.۱ تحلیل پیچیدگی زمانی و حافظه

هر دو الگوریتم ارائه شده برای مرتب‌سازی توپولوژیک، در صورتی که گراف با **لیست مجاورت** نمایش داده شود، بسیار کارآمد هستند و پیچیدگی زمانی خطی دارند. در جدول زیر، پیچیدگی هر دو الگوریتم را مشاهده می‌کنید.

جدول ۳.۱: مقایسه پیچیدگی الگوریتم‌های مرتب‌سازی توپولوژیک

الگوریتم	**پیچیدگی زمانی**	**پیچیدگی حافظه**
مبتنی بر درجه ورودی (Kahn)	$O(\|V\|+\|E\|)$	$O(\|V\|)$
مبتنی بر DFS	$O(\|V\|+\|E\|)$	$O(\|V\|)$

توضیح پیچیدگی الگوریتم

الگوریتم Kahn:

- **پیچیدگی زمانی** ($O(|V|+|E|)$):

۱. **محاسبه درجه‌های ورودی:** برای این کار کافی است یک بار تمام یال‌های گراف را پیمایش کنیم. این کار $O(|E|)$ هزینه دارد.

۲. **یافتن رئوس اولیه:** یک بار تمام رئوس را پیمایش می‌کنیم تا آن‌هایی که درجه ورودی صفر دارند را پیدا کرده و به صف اضافه کنیم. این کار $O(|V|)$ هزینه دارد.

۳. **حلقه اصلی:** حلقه while دقیقاً به ازای هر رأس یک بار اجرا می‌شود (هر رأس فقط یک بار وارد صف و از آن خارج می‌شود). داخل حلقه، به ازای هر رأس u، ما همسایگان آن را پیمایش می‌کنیم. در مجموع کل اجرای حلقه، هر یال گراف دقیقاً یک بار بررسی می‌شود. پس هزینه کل این بخش $O(|V| + |E|)$ خواهد بود.

بنابراین، هزینه کل الگوریتم از جمع این مراحل به دست می‌آید که برابر با $O(|V| + |E|)$ است.

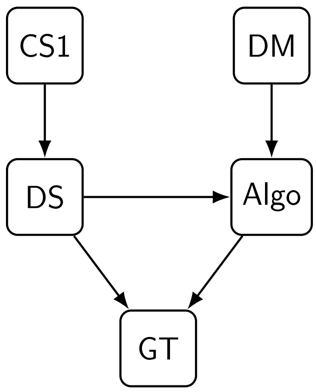
- **پیچیدگی حافظه ($O(|V|)$):** ما برای نگهداری درجه ورودی هر رأس به یک آرایه به اندازه $O(|V|)$ نیاز داریم. صف نیز در بدترین حالت تمام رئوس را در خود جای می‌دهد که $O(|V|)$ فضا می‌گیرد. لیست خروجی نیز $O(|V|)$ فضا نیاز دارد.

الگوریتم مبتنی بر DFS:

- **پیچیدگی زمانی ($O(|V|+|E|)$):** این الگوریتم در واقع یک پیمایش کامل DFS روی گراف است. می‌دانیم که هزینه پیمایش DFS با نمایش لیست مجاورت، $O(|V|+|E|)$ است، زیرا هر رأس و هر یال دقیقاً یک بار ملاقات می‌شوند. تنها عملیات اضافه، افزودن هر رأس به ابتدای یک لیست پیوندی است که این عمل هزینه ثابت $O(1)$ دارد. بنابراین هزینه کلی تغییری نمی‌کند.
- **پیچیدگی حافظه ($O(|V|)$):** حافظه مورد نیاز شامل آرایه‌ای برای علامت‌گذاری رئوس ملاقات‌شده ($O(|V|)$)، پشته بازگشتی سیستم که در بدترین حالت (یک گراف زنجیره‌ای) می‌تواند تا عمق $O(|V|)$ برود، و لیست پیوندی خروجی ($O(|V|)$) است.

۵.۳.۱) مثال

فرض کنید گراف زیر، پیش‌نیازهای تعدادی از دروس دانشگاهی را نشان می‌دهد. می‌خواهیم با استفاده از الگوریتم Kahn، یک ترتیب معتبر برای گذراندن این دروس پیدا کنیم.



شکل ۳.۱: گراف وابستگی دروس به عنوان یک DAG.

حل با الگوریتم Kahn: این الگوریتم با پیدا کردن رئوس با درجه ورودی صفر کار می‌کند. ما از یک صف (Queue) برای نگهداری رئوس با درجه ورودی صفر استفاده می‌کنیم. لیست مرتب‌شده نهایی را نیز L می‌نامیم.

۱. **مرحله اولیه:** ابتدا درجه ورودی تمام رئوس را محاسبه می‌کنیم:

- in-degree(CS1) = 0
- in-degree(DM) = 0

- in-degree(DS) = 1
- in-degree(Algo) = 2
- in-degree(GT) = 2

رئوسی که درجه ورودی صفر دارند (CS1 و DM) را به صف اضافه می‌کنیم.

- صف $S = [\text{CS1, DM}]$
- لیست مرتب $L = []$

۲. **گام ۱:** رأس CS1 را از صف خارج کرده و به لیست L اضافه می‌کنیم. یال خروجی آن به DS را حذف می‌کنیم. درجه ورودی DS از ۱ به ۰ کاهش می‌یابد. چون درجه ورودی DS صفر شده، آن را به صف اضافه می‌کنیم.

- صف $S = [\text{DM, DS}]$
- لیست مرتب $L = [\text{CS1}]$

۳. **گام ۲:** رأس DM را از صف خارج کرده و به L اضافه می‌کنیم. یال خروجی آن به Algo را حذف می‌کنیم. درجه ورودی Algo از ۲ به ۱ کاهش می‌یابد.

- صف $S = [\text{DS}]$
- لیست مرتب $L = [\text{CS1, DM}]$

۴. **گام ۳:** رأس DS را از صف خارج و به L اضافه می‌کنیم. یال‌های خروجی آن به Algo و GT را حذف می‌کنیم.

- درجه ورودی Algo از ۱ به ۰ کاهش می‌یابد. پس Algo به صف اضافه می‌شود.
- درجه ورودی GT از ۲ به ۱ کاهش می‌یابد.

- صف $S = [\text{Algo}]$
- لیست مرتب $L = [\text{CS1, DM, DS}]$

۵. **گام ۴:** رأس Algo را از صف خارج و به L اضافه می‌کنیم. یال خروجی آن به GT را حذف می‌کنیم. درجه ورودی GT از ۱ به ۰ کاهش می‌یابد. پس GT به صف اضافه می‌شود.

- صف $S = [\text{GT}]$
- لیست مرتب $L = [\text{CS1, DM, DS, Algo}]$

۶. **گام ۵:** رأس GT را از صف خارج و به L اضافه می‌کنیم.

- صف $S = []$ (صف خالی شد)
- لیست مرتب $L = [\text{CS1, DM, DS, Algo, GT}]$

صف خالی است و الگوریتم پایان می‌یابد. یک ترتیب توپولوژیک معتبر برای دروس، **CS1, DM, DS, Algo, GT** است.

حل با الگوریتم مبتنی بر DFS: در این روش، ترتیب معکوس زمان پایان یافتن رئوس، نتیجه را مشخص می‌کند. فرض کنید حلقه اصلی الگوریتم، رئوس را به ترتیب الفبایی (Algo, CS1, DM, DS, GT) بررسی می‌کند.

۱. **گام ۱:** اولین رأس ملاقات‌نشده Algo است. DFS-Visit(Algo) فراخوانی می‌شود.

- Algo همسایه GT را می‌بیند. DFS-Visit(GT) فراخوانی می‌شود.
- GT همسایه ملاقات‌نشده‌ای ندارد. کارش تمام شده و به ابتدای لیست L اضافه می‌شود.
- وضعیت فعلی: لیست $L = [\text{GT}]$

۲. **گام ۲:** پیمایش به Algo بازمی‌گردد. Algo همسایه دیگری ندارد. کارش تمام شده و به ابتدای لیست L اضافه می‌شود.

- وضعیت فعلی: لیست $L = [\text{Algo, GT}]$

۳. **گام ۳:** حلقه اصلی به کار خود ادامه می‌دهد. رأس ملاقات‌نشده بعدی CS1 است. DFS-Visit(CS1) فراخوانی می‌شود.

- CS1 همسایه DS را می‌بیند. DFS-Visit(DS) فراخوانی می‌شود.
- DS همسایگان Algo و GT را می‌بیند که هر دو قبلاً ملاقات شده‌اند. پس کار DS تمام شده و به ابتدای لیست L اضافه می‌شود.
- وضعیت فعلی: لیست $L = [\text{DS, Algo, GT}]$

۴. **گام ۴:** پیمایش به CS1 بازمی‌گردد. CS1 همسایه دیگری ندارد. کارش تمام شده و به ابتدای لیست L اضافه می‌شود.

- وضعیت فعلی: لیست $L = [\text{CS1, DS, Algo, GT}]$

۵. **گام ۵:** حلقه اصلی به کار خود ادامه می‌دهد. رأس ملاقات‌نشده بعدی DM است. DFS-Visit(DM) فراخوانی می‌شود.

- DM همسایه Algo را می‌بیند که قبلاً ملاقات شده است. کار DM تمام شده و به ابتدای لیست L اضافه می‌شود.
- وضعیت فعلی: لیست $L = [\text{DM, CS1, DS, Algo, GT}]$

تمام رئوس ملاقات شده‌اند و الگوریتم پایان می‌یابد. ترتیب نهایی که در لیست L به دست آمد، یک ترتیب توپولوژیک معتبر دیگر است: **DM, CS1, DS, Algo, GT**.

۶.۳.۱) انیمیشن و ابزارهای پایتون

برای درک عمیق‌تر و بصری تفاوت عملکرد دو رویکرد مرتب‌سازی توپولوژیک، دو اسکریپت انیمیشن مجزا در پوشه chapter-1 پروژه گیت‌هاب قرار داده شده است.

- **kahn_animation.py:** این اسکریپت، الگوریتم Kahn را به صورت گام به گام نمایش می‌دهد. انیمیشن به صورت دو قسمتی است: در یک سمت، وضعیت گراف و در سمت دیگر، وضعیت لحظه‌ای **صف رئوس با درجه ورودی صفر** و همچنین **لیست مرتب‌شده نهایی** که به تدریج ساخته می‌شود را نشان می‌دهد. این انیمیشن به شما کمک می‌کند تا فرآیند حذف مفهومی یال‌ها و به‌روزرسانی درجه‌های ورودی را به صورت شهودی درک کنید.

- **topo_dfs_animation.py:** این اسکریپت، رویکرد مبتنی بر DFS را نمایش می‌دهد. انیمیشن آن نیز دو قسمتی بوده و در یک سمت، پیمایش عمیق DFS و در سمت دیگر، نحوه ساخته شدن لیست مرتب‌شده نهایی با اضافه شدن هر رأس به **ابتدای لیست** پس از پایان یافتن کار پیمایش آن را نشان می‌دهد.

۷.۳.۱) کاربردهای الگوریتم

مرتب‌سازی توپولوژیک یکی از پرکاربردترین الگوریتم‌های گراف در دنیای واقعی است، زیرا هر فرآیندی که شامل وابستگی و ترتیب باشد را می‌توان با آن مدل‌سازی کرد. برخی از کاربردهای کلیدی آن عبارتند از:

- **زمان‌بندی وظایف و پروژه‌ها (Task Scheduling):** این کلاسیک‌ترین کاربرد است. در یک پروژه بزرگ، وظایف زیادی وجود دارند که هر کدام پیش‌نیازهایی دارند (مثلاً قبل از رنگ‌آمیزی دیوار، باید بتونه‌کاری تمام شده باشد). با مدل‌سازی وظایف به عنوان رئوس و پیش‌نیازها به عنوان یال‌ها، مرتب‌سازی توپولوژیک یک ترتیب معتبر برای انجام تمام کارها ارائه می‌دهد. مثال دیگر، ترتیب گذراندن دروس دانشگاهی است.

- **سیستم‌های ساخت (Build Systems) و کامپایلرها:** در پروژه‌های نرم‌افزاری، فایل‌ها به یکدیگر وابسته‌اند. برای مثال، فایل A.c برای کامپایل شدن ممکن است به B.h نیاز داشته باشد. ابزارهایی مانند Make یک گراف وابستگی از فایل‌ها می‌سازند و با اجرای مرتب‌سازی توپولوژیک، ترتیب صحیح کامپایل شدن فایل‌ها را پیدا می‌کنند تا هیچ فایلی قبل از پیش‌نیازش کامپایل نشود.
- **محاسبات صفحات گسترده (Spreadsheets):** در نرم‌افزارهایی مانند اکسل، هر سلول می‌تواند به مقادیر سلول‌های دیگر وابسته باشد (مثلاً C1 = A1 + B1). وقتی مقدار یک سلول تغییر می‌کند، تمام سلول‌های وابسته به آن باید دوباره محاسبه شوند. گراف وابستگی سلول‌ها یک DAG است و مرتب‌سازی توپولوژیک یک ترتیب بهینه و صحیح برای به‌روزرسانی تمام مقادیر ارائه می‌دهد.

۸.۳.۱) تمرین

۱. **فرهنگ لغت بیگانه (Alien Dictionary):** لیستی از کلمات یک زبان بیگانه به شما داده شده است که بر اساس قوانین الفبای آن زبان، به صورت دیکشنری مرتب شده‌اند. الگوریتمی بنویسید که ترتیب صحیح حروف الفبای این زبان را پیدا کند. (راهنمایی: با مقایسه کلمات متوالی، گراف وابستگی بین حروف را بسازید و سپس آن را مرتب‌سازی توپولوژیک کنید. برای مثال، اگر کلمات “xzy” و “xyw” داده شوند، می‌توان نتیجه گرفت که حرف “z” باید قبل از “y” بیاید).

۲. **طولانی‌ترین مسیر در DAG:** پیدا کردن طولانی‌ترین مسیر در یک گراف عمومی یک مسئله NP-Hard است، اما این مسئله در گراف‌های جهت‌دار غیرمدور (DAG) در زمان خطی قابل حل است. الگوریتمی ارائه دهید که با استفاده از مرتب‌سازی توپولوژیک، طولانی‌ترین مسیر بین دو رأس داده شده s و t را در یک DAG وزن‌دار پیدا کند.

۳. **یکتایی مرتب‌سازی توپولوژیک:** الگوریتمی طراحی کنید که تشخیص دهد آیا یک DAG داده‌شده، دارای **تنها یک** مرتب‌سازی توپولوژیک منحصربه‌فرد است یا خیر. (راهنمایی: به شرایط صف در الگوریتم Kahn در هر مرحله دقت کنید. چه شرطی یکتایی را تضمین می‌کند؟)

۹.۳.۱) منابع برای مطالعه بیشتر

برای درک عمیق‌تر و مطالعه جزئیات بیشتر در مورد مرتب‌سازی توپولوژیک و اثبات‌های مربوط به آن، می‌توانید به منابع زیر مراجعه کنید:

- **Algorithm Design (KT):** Chapter 3, Section 3.6
- **Graphs, Algorithms, and Optimization (Kocay & Kreher):** Chapter 6, Section 6.5

۴.۱) تورهای اویلری (Eulerian Tours)

۱.۴.۱) توضیح الگوریتم

یک تور اویلری، یک تور (گذر بسته) در گراف است که از هر یال **دقیقاً یک بار** عبور می‌کند. این مسئله از معمای «پل‌های کونیگسبرگ» نشأت گرفت و توسط لئونارد اویلر حل شد. شرط لازم و کافی برای وجود چنین توری در یک گراف بی‌جهت و همبند این است که درجه تمام رئوس آن زوج باشد. الگوریتم استاندارد برای پیدا کردن این تور، الگوریتم هیرهولزر (Hierholzer) است که با پیدا کردن دورهای متوالی و ادغام آن‌ها کار می‌کند.

۲.۴.۱) اثبات درستی

قضیه: یک گراف بی‌جهت و همبند G، یک تور اویلری دارد **اگر و تنها اگر** درجه تمام رئوس آن زوج باشد.

اثبات بخش اول: (وجود تور $\Rightarrow$ زوج بودن درجه‌ها) فرض کنید گراف یک تور اویلری دارد. هر بار که این تور از یک رأس میانی v عبور می‌کند، با یک یال وارد و با یک یال دیگر خارج می‌شود که دو واحد به درجه v اضافه می‌کند. از آنجایی که تور تمام یال‌ها را پوشش می‌دهد، تمام یال‌های متصل به هر رأس به این شکل جفت می‌شوند، لذا درجه هر رأس باید زوج باشد. ■

اثبات بخش دوم: (زوج بودن درجه‌ها $\Rightarrow$ وجود تور) این بخش با روش سازنده الگوریتم هیرهولزر اثبات می‌شود. از یک رأس دلخواه u شروع کرده و یک گذر را دنبال می‌کنیم. چون درجه هر رأس زوج است، هرگز در یک رأس میانی گیر نمی‌افتیم و این گذر نهایتاً باید به u بازگردد و یک دور C را تشکیل دهد. اگر C تمام یال‌ها را پوشش نداده باشد، چون گراف همبند است، رأسی مانند v روی C وجود دارد که به یال‌های دیگری متصل است. گراف باقی‌مانده از حذف یال‌های C نیز شرط درجه زوج را ارضا می‌کند. طبق فرض استقرا، می‌توان در آن مؤلفه نیز یک تور اویلری پیدا کرد و آن را به تور C در نقطه v «ادغام» کرد. با تکرار این فرآیند، تمام یال‌ها پوشش داده می‌شوند. ■

۳.۴.۱) شبه‌کد الگوریتم

الگوریتم ۶ الگوریتم هیرهولزر برای یافتن تور اویلری

```
procedure FIND-EULERIAN-TOUR(G, u)
    Let T be an empty list (the tour).
    Let S be a stack and push u onto it.
    while S is not empty do
        Let curr ← the node at the top of stack S.
        if curr has an unvisited edge (curr, v) then
            Push v onto stack S.
            Mark edge (curr, v) as visited.
        else
            Pop curr from S and prepend it to list T.
    return T.
```

۴.۴.۱) تحلیل پیچیدگی زمانی و حافظه

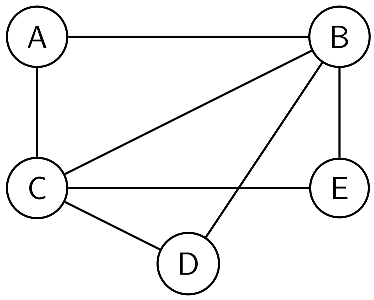
جدول ۴.۱: پیچیدگی الگوریتم هیرهولزر

نوع پیچیدگی	مقدار
پیچیدگی زمانی	$O(\|V\| + \|E\|)$
پیچیدگی حافظه	$O(\|V\| + \|E\|)$

توضیح: الگوریتم هر یال را دقیقاً یک بار پیمایش می‌کند که هزینه آن $O(|E|)$ است. هزینه کلی با احتساب پیمایش رئوس، خطی و برابر با $O(|V|+|E|)$ است. حافظه مورد نیاز نیز برای ذخیره گراف، پشته و لیست تور است که در مجموع $O(|V|+|E|)$ فضا نیاز دارد.

۵.۴.۱) مثال

یک تور اویلری در گراف زیر از رأس A پیدا می‌کنیم. الگوریتم ممکن است ابتدا دور A-C-B-A را پیدا کند. سپس از رأس C در این دور، دور جدید C-D-B-E-C را پیدا کرده و این دو را با هم ادغام می‌کند.



شکل ۴.۱: یک گراف اویلری که تمام رئوس آن درجه زوج دارند.

یک تور اویلری ممکن: **A-B-C-E-B-D-C-A**.

۶.۴.۱) انیمیشن و ابزارهای پایتون

یک اسکریپت پایتون با نام eulerian_animation.py در پوشه chapter-1 قرار دارد که اجرای الگوریتم هیرهولزر را به صورت بصری نمایش می‌دهد.

۷.۴.۱) کاربردها

- **مسیریابی بهینه:** مانند تمیز کردن خیابان‌ها یا بازرسی خطوط لوله.
- **ترسیم یکپارچه:** بررسی اینکه آیا یک شکل را می‌توان بدون برداشتن قلم از روی کاغذ کشید.
- **زیست‌شناسی محاسباتی:** در بازسازی توالی DNA از گراف‌های دی‌بروین اویلری استفاده می‌شود.

۸.۴.۱) تمرین

۱. **پیاده‌سازی هیرهولزر:** الگوریتم هیرهولزر را با استفاده از یک پشته صریح پیاده‌سازی کنید.

۲. **مسئله دومینوها:** مجموعه‌ای از دومینوها داده شده است. آیا می‌توان تمام آن‌ها را در یک حلقه چید به طوری که اعداد روی نیمه‌های مجاور یکسان باشند؟ این مسئله را به یافتن تور اویلری در یک گراف مناسب مدل کنید.

۳. **مسیر اویلری:** ثابت کنید یک گراف همبند دارای مسیر اویلری (گشتی که از هر یال یکبار عبور کند ولی بازگشتی نباشد) است اگر و تنها اگر دقیقاً دو رأس با درجه فرد داشته باشد.

۱.۴.۹) منابع برای مطالعه بیشتر

برای مطالعه دقیق‌تر و فنی‌تر در مورد تورهای اویلری، اثبات‌های مختلف و تاریخچه آن، می‌توانید به منابع زیر مراجعه کنید:

- **Graph Theory (J. A. Bondy, U. S. R. Murty):** Chapter 3, Section 3.3 (Euler Tours)
- **Graphs, Algorithms, and Optimization (Kocay & Kreher):** Chapter 3, Section 3.4 (Euler Tours)

۵.۱) تورهای هامیلتونی (Hamiltonian Tours)

۱.۵.۱) توضیح الگوریتم

یک **مسیر هامیلتونی (Hamiltonian Path)**، مسیری در یک گراف است که از هر **رأس** دقیقاً یک بار عبور می‌کند. اگر این مسیر یک دور باشد (یعنی رأس شروع و پایان آن یکی باشد و به هم یال داشته باشند)، به آن یک **دور یا تور هامیلتونی (Hamiltonian Cycle/Tour)** می‌گویند. گرافی که چنین دوری داشته باشد، **گراف هامیلتونی** نامیده می‌شود.

تفاوت کلیدی با تور اویلری: نکته بسیار مهمی که باید به آن توجه کرد، تفاوت بنیادین این مفهوم با تور اویلری است. تور اویلری از هر **یال** دقیقاً یک بار عبور می‌کرد، در حالی که تور هامیلتونی از هر **رأس** دقیقاً یک بار عبور می‌کند. این تفاوت ظریف، مسئله را از یک مسئله قابل حل در زمان چندجمله‌ای (کلاس P) به یک مسئله بسیار سخت تبدیل می‌کند.

پیچیدگی و دشواری مسئله: برخلاف مسئله تور اویلری، مسئله پیدا کردن تور هامیلتونی یکی از مشهورترین مسائل **NP-Complete** در علوم کامپیوتر است. این یعنی هیچ الگوریتم کارآمدی (با زمان چندجمله‌ای) برای تشخیص وجود یا پیدا کردن تور هامیلتونی در یک گراف عمومی شناخته نشده است و تمام الگوریتم‌های شناخته‌شده برای حل قطعی این مسئله، در بدترین حالت زمان اجرای نمایی دارند.

۲.۵.۱) قضایای مرتبط

از آنجایی که مسئله NP-Complete است، یک شرط لازم و کافی ساده مانند مسئله اویلر برای آن وجود ندارد. در عوض، قضایایی وجود دارند که شرایطی (کافی یا لازم) را برای وجود دور هامیلتونی بیان می‌کنند.

- **قضیه دیراک (Dirac's Theorem) - یک شرط کافی:** اگر G یک گراف ساده با $n \geq 3$ رأس باشد و درجه هر رأس آن حداقل $n/2$ باشد (یعنی $\delta(G) \geq n/2$)، آنگاه G حتماً هامیلتونی است. این قضیه به ما نمی‌گوید که اگر شرط برقرار نباشد، دور هامیلتونی وجود ندارد؛ بلکه فقط تضمین می‌کند که اگر شرط برقرار باشد، حتماً چنین دوری وجود دارد.

- **یک شرط لازم:** اگر گراف G دارای دور هامیلتونی باشد، آنگاه برای هر زیرمجموعه ناتهی و محض S از رئوس، با حذف رئوس S از گراف، تعداد مؤلفه‌های همبندی گراف باقی‌مانده ($\omega(G-S)$)، کمتر یا مساوی تعداد رئوس حذف شده ($|S|$) خواهد بود. یعنی $\omega(G-S) \leq |S|$. اگر بتوانیم یک مجموعه S پیدا کنیم که این شرط را نقض کند، می‌توانیم با اطمینان بگوییم که گراف هامیلتونی نیست.

۳.۵.۱) شبه‌کد الگوریتم

شبه‌کد زیر، الگوریتم هیرهولزر را با استفاده از یک رویکرد تکراری و یک پشته صریح برای پیدا کردن تور اویلری نشان می‌دهد.

الگوریتم ۷ الگوریتم هیرهولزر برای یافتن تور اویلری

```
procedure FIND-EULERIAN-TOUR(G, u)
    Let T be an empty list (the tour).
    Let S be a stack and push u onto it.
    while S is not empty do
        Let curr ← the node at the top of stack S.
        if curr has an unvisited edge (curr, v) then
            Push v onto stack S.
            Mark edge (curr, v) as visited.
        else
            Pop curr from S and prepend it to list T.
    return T.
```

توضیح شبه‌کد: الگوریتم با یک پشته که شامل رأس شروع است، آغاز می‌شود. در هر مرحله، به رأس بالای پشته ($curr$) نگاه می‌کنیم. اگر این رأس یال ملاقات‌نشده‌ای داشته باشد، به یکی از همسایگانش می‌رویم (آن همسایه را به بالای پشته اضافه می‌کنیم) و یال طی شده را علامت می‌زنیم. اما اگر رأس بالای پشته هیچ یال خروجی ملاقات‌نشده‌ای نداشته باشد، یعنی کار ما در این «زیرتور» تمام شده است. در این لحظه، آن رأس را از پشته برداشته و به **ابتدای** لیست تور نهایی (T) اضافه می‌کنیم. این فرآیند «افزودن به ابتدا پس از اتمام کار» تضمین می‌کند که دورهای تو در تو به درستی به یکدیگر متصل شوند.

۴.۵.۱) تحلیل پیچیدگی زمانی و حافظه

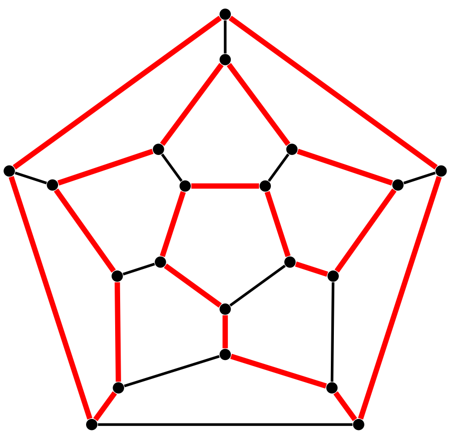
جدول ۱.۵: پیچیدگی الگوریتم Backtracking

نوع پیچیدگی	مقدار
پیچیدگی زمانی	$O(n!)$
پیچیدگی حافظه	$O(n)$

توضیح: الگوریتم در بدترین حالت، تمام مسیرهای ممکن را که شبیه به جایگشت‌های رئوس هستند، بررسی می‌کند. به همین دلیل پیچیدگی زمانی آن نمایی و در حدود فاکتوریل است. پیچیدگی حافظه نیز به دلیل عمق پشته فراخوانی‌های بازگشتی است که حداکثر برابر با تعداد رئوس (n) خواهد بود.

۵.۵.۱) مثال

در گراف دوازده‌وجهی (dodecahedron) زیر، یک دور هامیلتونی با رنگ قرمز مشخص شده است. پیدا کردن این دور به صورت چشمی بسیار دشوار است و نشان‌دهنده سختی این مسئله است.



شکل ۱.۵: یک دور هامیلتونی (قرمز) در گراف دوازده‌وجهی.

۶.۵.۱) انیمیشن و ابزارهای پایتون

از آنجایی که پیدا کردن دور هامیلتونی یک مسئله سخت است، انیمیشن آن نیز متفاوت خواهد بود.

اسکریپت hamiltonian_animation.py در پوشه chapter-1، به جای پیدا کردن قطعی دور، روند جستجوی **عقب‌گرد (Backtracking)** را به تصویر می‌کشد. در این انیمیشن، شما مشاهده می‌کنید که الگوریتم چگونه یک مسیر را می‌سازد، به بن‌بست می‌رسد، و با قرمز کردن یال اشتباه، به عقب برمی‌گردد تا مسیر دیگری را امتحان کند.

۷.۵.۱) کاربردها

- **لجستیک و حمل و نقل:** پایه‌ای برای مسئله فروشنده دوره‌گرد (TSP) که به دنبال پیدا کردن کوتاه‌ترین تور هامیلتونی است.
- **توالی‌یابی DNA:** در ژنومیک، برای چیدن قطعات کوتاه DNA و ساختن یک ژنوم کامل.
- **طراحی مدارها و شبکه‌ها:** پیدا کردن مسیری برای بازدید از تمام اجزا روی یک برد مدار چاپی.

۸.۵.۱) تمرین

۱. **تور اسب (Knight's Tour):** آیا یک اسب شطرنج می‌تواند از یک خانه شروع کرده، از تمام خانه‌های صفحه شطرنج دقیقاً یک بار عبور کند و به خانه اول بازگردد؟ این مسئله را به صورت پیدا کردن دور هامیلتونی در یک گراف مدل کنید.

۲. **بررسی قضیه دیراک:** یک گراف با ۸ رأس بسازید که در آن درجه هر رأس برابر ۴ باشد. آیا طبق قضیه دیراک، این گراف الزاماً هامیلتونی است؟ یک دور هامیلتونی در آن پیدا کنید.

۳. **توضیح NP-Completeness:** توضیح دهید چرا مسئله «تصمیم‌گیری دور هامیلتونی» (آیا گراف دور هامیلتونی دارد یا خیر؟) در کلاس NP قرار دارد.

۹.۵.۱) منابع برای مطالعه بیشتر

- **Introduction to Algorithms (CLRS):** Chapter 34, Section 34.5.3 (به عنوان یک مسئله NP-Complete)
- **Graph Theory (J. A. Bondy, U. S. R. Murty):** Chapter 18 (Hamilton Cycles)
- **Graphs, Algorithms, and Optimization (Kocay & Kreher):** Chapter 9 (Hamilton Cycles)

۶.۱) مسائل حل شده فصل ۱

در این بخش، به بررسی عمیق‌تر چند مسئله کلاسیک می‌پردازیم که بر پایه مفاهیم پیمایش بنا شده‌اند.

۱.۶.۱) مسئله فروشنده دوره‌گرد (Traveling Salesman Problem - TSP)

۱.۱.۶.۱) توضیح و تعریف مسئله

صورت مسئله: یک فروشنده را در نظر بگیرید که لیستی از n شهر را در اختیار دارد و باید به تمام آن‌ها سفر کند. او از شهر مبدأ خود حرکت کرده، از هر شهر دیگر دقیقاً یک بار بازدید می‌کند و در نهایت به شهر مبدأ باز می‌گردد. اگر هزینه (یا مسافت) سفر بین هر دو شهر مشخص باشد، مسئله این است: **کوتاه‌ترین تور ممکن که از تمام شهرها عبور کند، کدام است؟**

به زبان نظریه گراف، مسئله به این صورت تعریف می‌شود:

> در یک گراف کامل و وزن‌دار، یک **دور هامیلتونی با کمترین وزن کل** پیدا کنید.

۲.۱.۶.۱) پیچیدگی و دشواری مسئله

مسئله فروشنده دوره‌گرد، نمونه بارز یک مسئله بهینه‌سازی **NP-Hard** است. این به آن معناست که:

- هیچ الگوریتم کارآمدی (با زمان چندجمله‌ای) برای پیدا کردن جواب دقیق و بهینه این مسئله شناخته نشده است.
- برای پیدا کردن جواب قطعی، در بدترین حالت باید تعداد بسیار زیادی از مسیرهای ممکن را بررسی کنیم که با افزایش تعداد شهرها، این کار به سرعت غیرممکن می‌شود.

به همین دلیل، رویکردهای حل TSP به دو دسته اصلی تقسیم می‌شوند: **الگوریتم‌های دقیق** که جواب بهینه را تضمین می‌کنند اما کند هستند، و **الگوریتم‌های تقریبی** که جوابی نزدیک به بهینه را در زمانی معقول پیدا می‌کنند.

۳.۱.۶.۱) رویکرد دقیق: برنامه‌نویسی پویا (الگوریتم Held-Karp)

این الگوریتم که در اوایل دهه ۱۹۶۰ میلادی به صورت مستقل توسط مایکل هلد و ریچارد کارپ ابداع شد، از روش **برنامه‌نویسی پویا (Dynamic Programming)** برای حل مسئله استفاده می‌کند. ایده اصلی، شکستن مسئله به زیرمسائل کوچکتر و همپوشان است.

تعریف زیرمسئله: برای پیاده‌سازی این ایده، یک زیرمسئله را به شکل زیر تعریف می‌کنیم: فرض کنید رئوس گراف را با اعداد $1, 2, \ldots, n$ شماره‌گذاری کرده‌ایم و رأس ۱ را به عنوان مبدأ انتخاب می‌کنیم.

> $C(S, j)$ را برابر با هزینه کوتاه‌ترین مسیر از رأس مبدأ (۱) در نظر بگیرید که از تمام رئوس مجموعه $S \subseteq \{1, \ldots, n\}$ دقیقاً یک بار عبور کرده و در نهایت به رأس $j \in S$ ختم می‌شود.

رابطه بازگشتی: با تعریف بالا، می‌توانیم یک رابطه بازگشتی برای محاسبه $C(S, j)$ بنویسیم:

- **حالت پایه:** اگر اندازه مجموعه S برابر ۲ باشد، یعنی $S = \{1, j\}$، آنگاه مسیر فقط شامل یک یال از ۱ به j است. بنابراین: $C(\{1, j\}, j) = \text{dist}(1, j)$
- **گام بازگشتی:** برای محاسبه $C(S, j)$، فرض می‌کنیم که قبل از رسیدن به رأس j، در رأس دیگری مانند k بوده‌ایم. این رأس k باید عضوی از S و مخالف j باشد. مسیر بهینه تا رسیدن به k، باید از تمام رئوس $S - \{j\}$ عبور کرده باشد. بنابراین، برای یافتن کوتاه‌ترین مسیر تا j، باید تمام انتخاب‌های ممکن برای رأس قبلی (k) را امتحان کنیم و بهترین گزینه را انتخاب نماییم.

$$C(S, j) = \min_{k \in S, k \neq j} \{C(S - \{j\}, k) + \text{dist}(k, j)\}$$

با حل این رابطه از زیرمسائل کوچک (مجموعه‌هایی با اندازه کم) به سمت زیرمسائل بزرگتر، در نهایت می‌توانیم هزینه کوتاه‌ترین تور کامل را پیدا کنیم. هزینه تور نهایی برابر است با هزینه رفتن به آخرین رأس و سپس بازگشت از آن به مبدأ:

$$\text{Cost(Tour)} = \min_{j \neq 1} \{C(\{1, \ldots, n\}, j) + \text{dist}(j, 1)\}$$

۴.۱.۶.۱) شبه‌کد الگوریتم Held-Karp

الگوریتم ۸ الگوریتم Held-Karp برای TSP

procedure HELD-KARP-TSP(G)
  Let C be a 2D array for memoization, initialized to ∞.
  For $k \leftarrow 2$ to n: $C[\{1, k\}][k] \leftarrow \text{dist}(1, k)$.

  for $s \leftarrow 3$ to n **do** ▷ s is the subset size
    for each subset S of $\{1, \ldots, n\}$ of size s that contains 1 **do**
      for each $j \in S$ where $j \neq 1$ **do**
        $C[S][j] \leftarrow \min_{k \in S, k \neq j} \{C[S - \{j\}][k] + \text{dist}(k, j)\}$

  Let $V_{all} \leftarrow \{1, \ldots, n\}$.
  opt $\leftarrow \min_{j \in V_{all}, j \neq 1} \{C[V_{all}][j] + \text{dist}(j, 1)\}$.
  return opt.

۵.۱.۶.۱) تحلیل دقیق پیچیدگی

پیچیدگی زمانی $(O(n^2 \cdot 2^n))$:

- قلب الگوریتم، سه حلقه تودرتو است. بیایید تعداد تکرار هر کدام را تحلیل کنیم:
- **حلقه اول (اندازه زیرمجموعه s):** این حلقه از ۳ تا n اجرا می‌شود، پس $O(n)$ تکرار دارد.
- **حلقه دوم (انتخاب زیرمجموعه S):** برای هر اندازه s، ما باید تمام زیرمجموعه‌هایی به آن اندازه را که شامل رأس ۱ هستند، انتخاب کنیم. تعداد این زیرمجموعه‌ها برابر است با انتخاب $s - 1$ عضو از بین $n - 1$ رأس باقی‌مانده، یعنی $\binom{n-1}{s-1}$.
- **حلقه سوم (انتخاب رأس پایانی j):** برای هر زیرمجموعه S به اندازه s، $s - 1$ انتخاب برای j وجود دارد.
- **حلقه داخلی (انتخاب رأس قبلی k):** برای هر j، $s - 2$ انتخاب برای k وجود دارد.
- اگر تمام این موارد را در هم ضرب کرده و روی تمام s ها جمع ببندیم، به یک رابطه پیچیده می‌رسیم. اما یک راه ساده‌تر برای تحلیل این است: تعداد کل حالات ممکن برای (S, j) چقدر است؟ S یک زیرمجموعه از رئوس است (2^n حالت) و j یک رأس است (n حالت). پس تقریباً $n \cdot 2^n$ حالت داریم. برای هر حالت، یک حلقه روی k داریم که $O(n)$ هزینه دارد. ضرب این‌ها در هم، پیچیدگی زمانی $O(n^2 \cdot 2^n)$ را به ما می‌دهد.

پیچیدگی حافظه $(O(n \cdot 2^n))$:

- حافظه اصلی مورد نیاز برای نگهداری جدول برنامه‌نویسی پویا، یعنی آرایه C است.

- این آرایه دو بعد دارد: بعد اول، زیرمجموعه S است. تعداد زیرمجموعه‌های ممکن از n رأس، 2^n است. بعد دوم، رأس پایانی j است که n حالت دارد.
- بنابراین، اندازه این جدول از مرتبه $O(n \cdot 2^n)$ خواهد بود.

۶.۱.۶.۱) رویکردهای تقریبی (Approximation Algorithms)

از آنجایی که الگوریتم‌های دقیق برای TSP تنها برای تعداد شهرهای کم قابل استفاده هستند، در عمل از الگوریتم‌های تقریبی استفاده می‌شود. این الگوریتم‌ها جواب بهینه را تضمین نمی‌کنند، اما در زمان معقولی، یک جواب «به اندازه کافی خوب» تولید می‌کنند. کیفیت یک الگوریتم تقریبی معمولاً با **ضریب تقریب (approximation ratio)** سنجیده می‌شود که نشان می‌دهد جواب الگوریتم حداکثر چند برابر بدتر از جواب بهینه است.

۱. الگوریتم حریصانه: نزدیک‌ترین همسایه (Nearest Neighbor) این روش، یک الگوریتم حریصانه و بسیار ساده است که درک و پیاده‌سازی آن آسان است.

- **ایده اصلی:** در هر مرحله، از شهری که در آن هستیم، به نزدیک‌ترین شهری که هنوز ملاقات نکرده‌ایم، سفر می‌کنیم.
- **مراحل اجرا:**
  ۱. از یک شهر دلخواه شروع کنید و آن را به تور اضافه کنید.
  ۲. در حالی که هنوز شهرهای ملاقات‌نشده وجود دارند، از شهر فعلی به نزدیک‌ترین شهر ملاقات‌نشده بروید و آن را به تور اضافه کنید.
  ۳. پس از ملاقات تمام شهرها، از آخرین شهر به شهر شروع بازگردید تا تور کامل شود.
- **تحلیل عملکرد:** با وجود سادگی، این الگوریتم می‌تواند بسیار بد عمل کند و هیچ ضریب تقریب ثابتی ندارد. یعنی در برخی موارد، جوابی که تولید می‌کند می‌تواند به صورت دلخواه از جواب بهینه بدتر باشد. انتخاب شهر شروع نیز می‌تواند به شدت روی کیفیت جواب نهایی تأثیر بگذارد.

۲. الگوریتم کریستوفیدس (Christofides Algorithm) این الگوریتم یکی از بهترین الگوریتم‌های تقریبی برای حالتی از TSP است که وزن یال‌ها **نامساوی مثلثی** را ارضا می‌کنند (یعنی مسافت مستقیم بین دو شهر همیشه کوتاه‌تر یا مساوی رفتن از یک شهر به شهر دیگر از طریق یک شهر سوم است: $[\text{dist}(u, v) \leq \text{dist}(u, w) + \text{dist}(w, v)]$). این الگوریتم تضمین می‌کند که هزینه تور پیدا شده، حداکثر **۵.۱ برابر** هزینه تور بهینه است (1.5-approximation).

- **ایده اصلی:** ساختن یک تور اویلری در یک گراف کمکی و سپس تبدیل آن به یک دور هامیلتونی.
- **مراحل اجرا:**

۱. **ساخت اسکلت اولیه:** یک **درخت فراگیر کمینه (Minimum Spanning Tree - MST)** از گراف شهرها بسازید. هزینه این درخت (C_{MST}) حتماً کمتر از هزینه تور بهینه است، زیرا با حذف یک یال از هر دوری، یک درخت فراگیر به دست می‌آید.
۲. **شناسایی رئوس مشکل‌ساز:** در درخت MST به دست آمده، تمام رئوسی که درجه فرد دارند را پیدا کنید. می‌دانیم که تعداد این رئوس همیشه زوج است. این رئوس مانع از داشتن یک تور اویلری هستند.
۳. **رفع مشکل درجه فرد:** برای زوج کردن درجه این رئوس، باید آن‌ها را با یال‌های جدیدی به هم وصل کنیم. برای اینکه کمترین هزینه ممکن را اضافه کنیم، یک **تطابق کامل با وزن کمینه (Minimum-weight perfect matching)** روی این زیرگراف از رئوس درجه فرد پیدا می‌کنیم.
۴. **ساخت گراف اویلری:** یال‌های تطابق پیدا شده را به درخت MST اضافه می‌کنیم. گراف چندگانه جدیدی که به دست می‌آید، همبند است و تمام رئوس آن درجه زوج دارند، بنابراین حتماً یک تور اویلری در آن وجود دارد.
۵. **پیدا کردن تور اویلری:** یک تور اویلری در گراف جدید پیدا می‌کنیم.

۶. **تبدیل به تور هامیلتونی:** تور اویلری ممکن است برخی رئوس را چند بار ملاقات کند. با استفاده از یک تکنیک به نام **میان‌بُر زدن (shortcutting)**، این تور را به یک دور هامیلتونی تبدیل می‌کنیم. به این صورت که در حین پیمایش تور اویلری، اگر قرار است به رأسی برویم که قبلاً در مسیر هامیلتونی ما بوده، از آن صرف نظر کرده و مستقیماً به رأس بعدی در تور اویلری می‌رویم. به لطف نامساوی مثلثی، این میان‌بر هرگز طول مسیر را افزایش نمی‌دهد.

ما در فصل‌های آینده، با مفاهیمی مانند درخت فراگیر کمینه و تطابق آشنا خواهیم شد و این الگوریتم را بهتر درک خواهیم کرد.

۲.۶.۱) مسئله پست‌چی چینی (Chinese Postman Problem - CPP)

۱.۲.۶.۱) توضیح و تعریف مسئله

صورت مسئله: یک پست‌چی را در نظر بگیرید که باید از اداره پست شروع کرده، تمام خیابان‌های یک منطقه را برای تحویل نامه طی کند و در نهایت به اداره پست بازگردد. او می‌خواهد کوتاه‌ترین مسیر ممکن را طی کند. تفاوت کلیدی این مسئله با تور اویلری این است که در اینجا، پست‌چی مجبور نیست از هر خیابان فقط یک بار عبور کند؛ او می‌تواند برخی خیابان‌ها را برای رسیدن به بخش‌های دیگر شهر، دوباره طی کند.

به زبان نظریه گراف، مسئله به این صورت تعریف می‌شود:

> در یک گراف همبند و وزن‌دار، یک **تور (گذر بسته) با کمترین وزن کل** پیدا کنید که از **هر یال حداقل یک بار** عبور کند.

۲.۲.۶.۱) پیچیدگی و قابلیت حل

برخلاف مسئله فروشنده دوره‌گرد که NP-Hard است، مسئله پست‌چی چینی به طرز شگفت‌انگیزی **قابل حل در زمان چندجمله‌ای** است و در کلاس P قرار دارد. این مسئله به زیبایی مفاهیم تور اویلری، کوتاه‌ترین مسیر و تطابق در گراف‌ها را به هم پیوند می‌زند.

۳.۲.۶.۱) رویکرد حل مسئله

ایده اصلی حل این مسئله، تبدیل هوشمندانه گراف ورودی به یک گراف اویلری است.

- اگر گراف ورودی از ابتدا اویلری باشد (یعنی همبند بوده و تمام رئوس آن درجه زوج داشته باشند)، آنگاه هر تور اویلری در آن، یک راه‌حل بهینه برای مسئله پست‌چی چینی است. در این حالت، هزینه کل برابر با مجموع وزن تمام یال‌های گراف است.

- چالش اصلی زمانی است که گراف، اویلری نیست. این یعنی حداقل دو رأس با درجه فرد در آن وجود دارد (می‌دانیم که تعداد رئوس درجه فرد در هر گرافی همیشه زوج است). برای اینکه بتوانیم یک تور اویلری بسازیم، باید درجه تمام این رئوس را زوج کنیم.

الگوریتم حل مسئله به صورت زیر است:

۱. **شناسایی رئوس مشکل‌ساز:** تمام رئوسی که درجه فرد دارند را در یک مجموعه به نام O قرار دهید.

۲. **محاسبه هزینه‌های اتصال:** برای هر زوج از رئوس در مجموعه O، کوتاه‌ترین مسیر بین آن دو را در گراف اصلی پیدا کنید. برای این کار می‌توان از الگوریتم‌هایی مانند دایکسترا یا فلوید–وارشال استفاده کرد.

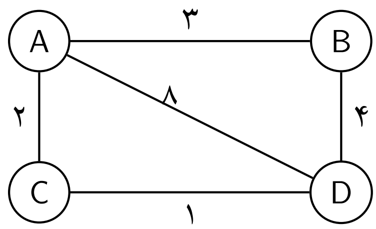
۳. **پیدا کردن بهینه‌ترین جفت‌ها:** یک گراف کامل جدید $K_{|O|}$ بسازید که رئوس آن همان رئوس مجموعه O هستند. وزن یال بین هر دو رأس در این گراف جدید، برابر با هزینه کوتاه‌ترین مسیری است که در مرحله قبل محاسبه کردیم. در این گراف کامل، یک **تطابق کامل با وزن کمینه (Minimum-weight perfect matching)** پیدا کنید. این تطابق، رئوس درجه فرد را به بهینه‌ترین شکل ممکن دو به دو جفت می‌کند.

۴. **تبدیل به گراف اویلری:** به گراف اصلی بازگردید. به ازای هر یال که در تطابق مرحله قبل پیدا شد، یال‌های مسیر کوتاه‌ترین متناظر با آن را در گراف اصلی «تکرار» کنید (یعنی یک نسخه کپی از آن‌ها به گراف اضافه کنید). گراف چندگانه جدیدی که به دست می‌آید، تضمیناً همبند است و تمام رئوس آن درجه زوج دارند، پس یک گراف اویلری است.

۵. **پیدا کردن تور نهایی:** در گراف جدید و اویلری شده، یک تور اویلری پیدا کنید. این تور، کوتاه‌ترین مسیر ممکن است که از تمام یال‌های گراف اصلی حداقل یک بار عبور می‌کند و جواب بهینه مسئله پست‌چی چینی است.

۴.۲.۶.۱) مثال

گراف وزن‌دار زیر را در نظر بگیرید. رئوس A و D درجه فرد (۳) و رئوس B و C درجه زوج (۲) دارند.



شکل ۶.۱: یک گراف غیر اویلری برای مسئله پست‌چی چینی.

۱. رئوس درجه فرد: $O = \{A, D\}$.

۲. کوتاه‌ترین مسیر بین A و D را پیدا می‌کنیم. مسیر مستقیم وزنی برابر ۸ دارد. اما مسیر A-C-D وزنی برابر $2 + 1 = 3$ دارد که کوتاه‌تر است.

۳. چون فقط دو رأس درجه فرد داریم، تطابق کمینه بدیهی است: A به D وصل می‌شود.

۴. یال‌های مسیر A-C-D را در گراف تکرار می‌کنیم. گراف جدید دارای یال‌های (A,C) و (C,D) به صورت تکراری است.

۵. در گراف جدید، یک تور اویلری پیدا می‌کنیم، مثلاً: A-B-D-C-A-D-C-A. این تور جواب بهینه مسئله است.

۷.۱) تمرینات تکمیلی فصل ۱

در این بخش، مجموعه‌ای از تمرین‌ها در سطوح مختلف برای مرور و تسلط بر مفاهیم ارائه شده در این فصل، آورده شده است.

۱.۷.۱) تمرین‌های آسان

این تمرین‌ها برای مرور تعاریف و الگوریتم‌های پایه طراحی شده‌اند.

۱. **قضیه دست‌دادن:** قضیه دست‌دادن (Handshaking Lemma) را ثابت کنید: مجموع درجات تمام رئوس در یک گراف برابر با دو برابر تعداد یال‌هاست ($\sum_{v \in V} \deg(v) = 2|E|$). از این قضیه نتیجه بگیرید که تعداد رئوس با درجه فرد، همیشه زوج است.

۲. **حداکثر یال‌ها:** حداکثر تعداد یال‌ها در یک گراف ساده با n رأس چقدر است؟

۳. **گراف با دنباله درجات:** یک گراف با ۶ رأس و دنباله درجات (۵، ۵، ۵، ۵، ۵، ۵) را رسم کنید. آیا این گراف هامیلتونی است؟ آیا اویلری است؟

۴. **گراف کامل K_5:** گراف کامل K_5 (گرافی با ۵ رأس که هر رأس به تمام رئوس دیگر متصل است) را در نظر بگیرید. آیا این گراف اویلری است؟ آیا هامیلتونی است؟

۵. **تعداد مرتب‌سازی‌های توپولوژیک:** یک گراف جهت‌دار غیرمدور (DAG) با ۴ رأس مثال بزنید که دقیقاً ۴ مرتب‌سازی توپولوژیک متفاوت داشته باشد.

۶. **گراف اویلری و هامیلتونی:** آیا یک گراف می‌تواند همزمان هم اویلری و هم هامیلتونی باشد؟ یک مثال با حداقل ۵ رأس بیاورید.

۷. **پیمایش گراف کامل:** ترتیب پیمایش BFS و DFS را برای یک گراف کامل K_4 از یک رأس دلخواه بنویسید.

۸. **گراف ناهمبند:** آیا یک گراف ناهمبند می‌تواند تور اویلری داشته باشد؟ چرا؟

۲.۷.۱) تمرین‌های متوسط

این تمرین‌ها نیاز به ترکیب مفاهیم و تسلط بر الگوریتم‌ها دارند.

۱. **گراف دوبخشی:** ثابت کنید یک گراف، دوبخشی است اگر و تنها اگر هیچ دور با طول فرد نداشته باشد. سپس الگوریتمی مبتنی بر BFS برای بررسی این ویژگی ارائه دهید.

۲. **تشخیص دور در گراف بی‌جهت:** الگوریتم DFS را طوری تغییر دهید که بتواند وجود دور در یک گراف **بی‌جهت** و همبند را تشخیص دهد. (راهنمایی: صرفاً پیدا کردن یک یال که به یک رأس ملاقات‌شده (visited) اشاره دارد کافی نیست).

۳. **مؤلفه‌های همبندی:** با استفاده از پیمایش DFS، تعداد مؤلفه‌های همبند یک گراف بی‌جهت را در زمان $O(|V|+|E|)$ محاسبه کنید.

۴. **مسیر با وزن خاص:** یک گراف جهت‌دار و وزن‌دار G داده شده است. الگوریتمی ارائه دهید که تشخیص دهد آیا یک مسیر از رأس s به t وجود دارد که وزن آن دقیقاً برابر W باشد یا خیر.

۵. **یکتایی مرتب‌سازی توپولوژیک:** ثابت کنید که یک DAG دارای یک مرتب‌سازی توپولوژیک یکتاست، اگر و تنها اگر یک مسیر هامیلتونی داشته باشد.

۶. **قطر درخت:** الگوریتمی ارائه دهید که با دو بار اجرای BFS، قطر یک درخت (طولانی‌ترین مسیر بین هر دو رأس) را پیدا کند و درستی آن را اثبات کنید.

۷. **طول مسیر در گراف همبند:** ثابت کنید که در هر گراف ساده همبند با $n \geq 3$ رأس، یک مسیر با طول حداقل $\min(n-1, 2\delta(G))$ وجود دارد.

۸. **دور هامیلتونی در گراف شبکه‌ای:** یک گراف شبکه‌ای $m \times n$ را در نظر بگیرید. شرایطی را برای m و n پیدا کنید که این گراف دارای دور هامیلتونی باشد.

۳.۷.۱) تمرین‌های سخت و مفهومی

این تمرین‌ها به درک عمیق‌تر نظریه و کاربردهای پیشرفته‌تر الگوریتم‌ها نیاز دارند.

۱. **رئوس مفصلی (Articulation Points):** الگوریتمی مبتنی بر DFS و زمان‌های کشف/پایان ارائه دهید که تمام رئوس مفصلی (رئوسی که حذف آن‌ها تعداد مؤلفه‌های همبندی را افزایش می‌دهد) را در یک گراف بی‌جهت در زمان خطی پیدا کند.

۲. **پل‌ها (Bridges):** الگوریتمی مبتنی بر DFS ارائه دهید که تمام پل‌ها (یال‌هایی که حذف آن‌ها تعداد مؤلفه‌های همبندی را افزایش می‌دهد) را در یک گراف بی‌جهت در زمان خطی پیدا کند.

۳. **مؤلفه‌های قویاً همبند (SCC):** الگوریتم کوساراجو (Kosaraju) یا تارژان (Tarjan) را برای یافتن تمام مؤلفه‌های قویاً همبند در یک گراف جهت‌دار در زمان خطی پیاده‌سازی و تحلیل کنید.

۴. **طولانی‌ترین مسیر در DAG:** الگوریتمی در زمان خطی برای پیدا کردن طولانی‌ترین مسیر در یک گراف جهت‌دار غیرمدور و وزن‌دار ارائه دهید. چرا این مسئله برای گراف‌های عمومی (که ممکن است دور داشته باشند) بسیار سخت‌تر است؟

۵. **تور اسب (Knight's Tour):** الگوریتمی با استفاده از عقب‌گرد بنویسید که یک مسیر هامیلتونی برای حرکت اسب در یک صفحه شطرنج $n \times n$ پیدا کند.

۶. **اثبات قضیه دیراک:** قضیه دیراک برای وجود دور هامیلتونی را اثبات کنید(مبانی نظریه گراف).

۷. **اثبات NP-Complete بودن دور هامیلتونی:** توضیح دهید چرا مسئله «تصمیم‌گیری دور هامیلتونی» در کلاس NP قرار دارد، سپس ثابت کنید که این مسئله NP-Complete است (می‌توانید از کاهش مسئله 3-SAT یا Vertex Cover استفاده کنید).

۸. **مسئله پست‌چی چینی (حالت کلی):** الگوریتم کامل مسئله پست‌چی چینی را پیاده‌سازی کنید. این الگوریتم نیازمند ترکیب الگوریتم‌های یافتن کوتاه‌ترین مسیر و پیدا کردن تطابق کامل با وزن کمینه در یک گراف است.

۹. **تعداد مرتب‌سازی‌های توپولوژیک:** الگوریتمی برای شمارش تعداد کل مرتب‌سازی‌های توپولوژیک ممکن در یک DAG ارائه دهید. (این مسئله در حالت کلی P-Complete و بسیار سخت است).

۱۰. **گراف‌های دی‌بروین (De Bruijn Graphs):** تحقیق کنید که چگونه از تورهای اویلری در گراف‌های دی‌بروین برای بازسازی توالی ژنوم استفاده می‌شود. یک مثال کوچک از این فرآیند را توضیح دهید.