فصل ۷

الگوریتم‌های گراف

راه میان‌بر، طولانی‌ترین راه بین دو نقطه است.

ناشناس

۷-۱ آشنایی

در فصل پیش، الگوریتم‌هایی را برای کار با مجموعه‌ها یا دنباله‌هایی از اشیاء بررسی کردیم و به رابطه‌هایی مانند ترتیب، تکرار و هم‌پوشانی برخوردیم. در این فصل، با رابطه‌های بیش‌تری بین اشیاء آشنا خواهیم شد و گراف‌ها را برای مدل‌سازی این رابطه‌ها به کار خواهیم گرفت. گراف‌ها می‌توانند وضعیت‌های بسیار گوناگونی را مدل کنند و در زمینه‌های بسیاری از باستان‌شناسی گرفته تا روان‌شناسی اجتماعی کاربرد دارند. در این فصل، برای کار با گراف‌ها و محاسبه‌ی ویژگی‌های مشخصی از آن‌ها، چندین الگوریتم مهم و پایه‌ای را به شما معرفی خواهیم کرد.

نخست، به نمونه‌هایی از مدل‌سازی با گراف نگاهی می‌اندازیم:

۱- یافتن مسیری خوب از خانه تا رستورانی در شهر؛ خیابان‌ها متناظر با یال‌ها (اگر خیابان‌ها یک‌طرفه باشند، یال‌های جهت‌دار) و تقاطع خیابان‌ها متناظر با رأس‌ها هستند. برای هر رأس و هر یال (هر تکه از خیابان که بین دو تقاطع است) زمان تأخیری مورد انتظار است و مسأله، یافتن «سریع‌ترین» مسیر بین دو رأس است.

۲- برخی برنامه‌های رایانه‌ای را می‌توان به وضعیت‌های گوناگون تقسیم کرد. ممکن است در هر یک از این وضعیت‌ها، حالت‌های مختلفی برای پیش‌روی وجود داشته باشد و برخی از این وضعیت‌ها نیز ممکن است نامطلوب باشند. یافتن وضعیت‌هایی که به یک حالت نامطلوب منجر می‌شوند، مسأله‌ی دیگری از نظریه‌ی گراف است که در آن هر وضعیت، متناظر با یک رأس و هر یال متناظر با امکان پیش‌روی از یک وضعیت به وضعیت دیگر خواهد بود.

۳- به مسأله‌ی برنامه‌ریزی کلاس‌های یک دانشگاه می‌توان به صورت مسأله‌ای از نظریه‌ی گراف نگریست. رأس‌ها بیانگر کلاس‌ها هستند و اگر دانش‌آموزی بخواهد دو کلاس را با هم بگیرد، یا استادی بخواهد در هر دو کلاس تدریس کند، آنگاه آن دو کلاس را به یکدیگر

متصل می‌کنیم. مسأله، برنامه‌ریزی کلاس‌هاست، به گونه‌ای که تعداد تداخل کلاس‌ها کمینه گردد. این مسأله دشوار است و به راحتی نمی‌توان راه‌حل مناسبی برای آن یافت.

۴- یک سامانه‌ی رایانه‌ای با چندین حساب کاربری در نظر بگیرید که در آن هر کاربر برای دست‌رسی به حساب خود، مجوز یا امتیازی امنیتی دارد. ممکن است کاربران بخواهند با یکدیگر همکاری کنند و اجازه‌ی دست‌رسی به حساب خود را در اختیار کاربر دیگری هم بگذارند. از سویی، اگر کاربر A به حساب کاربر B و کاربر B به حساب کاربر C دست‌رسی داشته باشد، آنگاه A نیز به حساب کاربر C دست‌رسی خواهد داشت. مسأله‌ی تعیین دست‌رسی کاربران به حساب‌های یکدیگر، مسأله‌ای دیگر از نظریه‌ی گراف است. در اینجا، کاربران با رأس‌ها متناظر می‌شوند و اگر کاربر A اجازه‌ی دست‌رسی به حساب خود را به کاربر B بدهد، یالی جهت‌دار از A به B وجود خواهد داشت.

کتاب‌های درسی بسیاری درباره‌ی نظریه‌ی گراف و کاربردهای فروان آن وجود دارند (مراجع پایان فصل را ببینید).

چند ساختمان‌داده که برای نگه‌داری گراف مناسبند، پیش‌تر در بخش ۴-۶ بررسی شدند. در این کتاب، برای نگه‌داری گراف‌ها، بیش‌تر، لیست همسایگی را به کار می‌گیریم که در گراف‌های تنک یا خلوت (گراف‌های کم‌یال) از دیگر روش‌ها کارآمدتر است. نخست، با واژه‌های رایج آشنا می‌شویم. گراف G=(V,E) از مجموعه‌ی رأس‌های (یا گره‌های) V و مجموعه‌ی یال‌های E تشکیل می‌شود. هر یال، با زوجی متمایز از دو رأس متناظر است. (گاهی طوقه هم، یعنی یالی از یک رأس به خودش مجاز است، ولی ما فرض می‌کنیم به کار بردن طوقه مجاز نباشد.) یک گراف ممکن است «جهت‌دار» یا «بدون‌جهت» باشد. یال‌های گراف جهت‌دار، زوج‌هایی مرتبند؛ یعنی ترتیب دو رأسی که یک یال آن‌ها را به هم متصل می‌کند، بااهمیت است. در گراف جهت‌دار یال را به صورت پیکانی از یک رأس (دم) به رأس دیگر (سر) می‌کشیم. یال‌های گراف بدون‌جهت، زوج‌هایی از رأس‌ها بدون توجه به ترتیب آن‌هاست و آن‌ها را به راحتی با خطی بین دو رأس نشان می‌دهیم. گراف چندگانه گرافی است که در آن بین هر دو زوج از رأس‌ها ممکن است چندین یال وجود داشته باشد (یعنی E در این گراف، یک مجموعه‌ی چندگانه است). گاهی به گرافی که چندگانه نیست، گراف ساده می‌گویند. فرض می‌کنیم گراف‌هایی که با آن‌ها کار می‌کنیم، همگی ساده‌اند مگر آن که خلافش گفته شود. برای رأس v، درجه‌ی $d(v)$، تعداد یال‌های متصل به آن است. در گراف‌های جهت‌دار بین درجه‌ی ورودی و درجه‌ی خروجی فرق می‌گذاریم؛ اولی، تعداد یال‌هایی است که به v وارد می‌شوند و دومی، تعداد یال‌هایی است که از v خارج می‌شوند.

مسیری از رأس v_1 به رأس v_k دنباله‌ای از رأس‌های v_1، v_2، ... و v_k است که به ترتیب با یال‌های (v_1,v_2)، (v_2,v_3)، ... و (v_{k-1},v_k) به یکدیگر متصل شده‌اند. (معمولاً یال‌ها نیز بخشی از مسیر در نظر گرفته می‌شوند.) مسیر را ساده گویند، اگر هر رأس حداکثر یک بار در آن دیده شود. اگر

مسیری (بنا به نوع گراف، جهت‌دار یا بدون‌جهت) از رأس v به رأس u وجود داشته باشد، می‌گوییم u از v دسترس‌پذیر است (یا v به u دسترسی دارد). مدار، مسیری است که رأس‌های آغاز و پایانش یکسانند. مدار را ساده گویند، اگر در آن به جز نخستین رأس (یا همان آخرین رأس) رأس دیگری بیش از یک بار ظاهر نشود. به مدار ساده، دور نیز می‌گویند. (گاهی به مدارهای غیرساده نیز دور می‌گویند، ولی ما فرض می‌کنیم که «دور»ها همگی ساده‌اند.) منظور از شکل بدون‌جهتِ یک گراف جهت‌دار، خود آن گراف است بدون در نظر گرفتن جهت یال‌هایش. گراف را همبند گویند، اگر در شکل بدون‌جهت آن، از هر رأس مسیری به هر رأس دیگر وجود داشته باشد. جنگل، گرافی است که در شکل بدون‌جهت آن دور وجود نداشته باشد. درخت، جنگلی همبند است. درخت ریشه‌دار (که به آن arborescence نیز می‌گویند) درختی جهت‌دار است با یک رأس خاص به نام «ریشه» به گونه‌ای که همه‌ی یال‌ها از آن دور می‌شوند.

زیرگرافی از گراف G=(V,E)، هر گرافی همچون H=(U,F) است چنان که $U \subseteq V$ و $F \subseteq E$. درخت پوشا یا پوششیِ گراف بدون‌جهت G، زیرگرافی از G است که هم درخت باشد و هم تمام رأس‌های G را در بر گیرد. جنگل پوشا یا پوششی گراف بدون‌جهت G، زیرگرافی از G است که هم جنگل باشد و هم تمام رأس‌های G را در بر گیرد. در گراف G=(V,E)، هر زیرگراف القاشده با رأس‌ها، زیرگرافی مانند H=(U,F) است که $U \subseteq V$ و F تمام یال‌هایی از E را در بر گیرد که هر دو رأس آن‌ها متعلق به U باشد. معمولاً به زیرگراف القاشده با رأس‌ها، زیرگراف القایی می‌گویند. اگر گراف G=(V,E) همبند نباشد، می‌توان آن را به طور یکتا به زیرگراف‌هایی همبند افراز کرد. این زیرگراف‌ها را مؤلفه‌های همبند G می‌گویند. یک مؤلفه‌ی همبند G زیرگرافی همبند از آن است، به گونه‌ای که زیرگراف همبند دیگری از G آن را در بر نگرفته باشد؛ به عبارت دیگر، مؤلفه‌های همبند، بزرگ‌ترین زیرگراف‌های همبند هستند. گراف دوبخشی، گرافی است که می‌توان رأس‌هایش را به دو مجموعه تقسیم کرد، به گونه‌ای که هر یال گراف، رأسی از یک مجموعه را به رأسی از مجموعه‌ی دیگر متصل کند. گراف وزن‌دار، گرافی است که یال‌هایش وزن (یا هزینه، یا طول) داشته باشند.

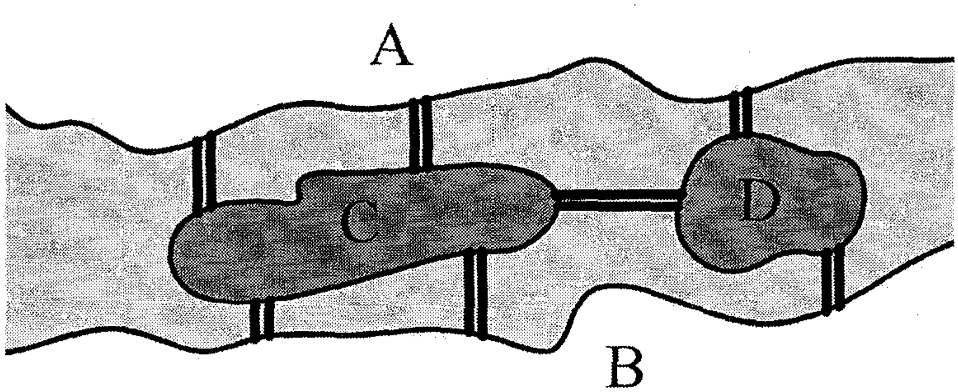
صرف‌نظر از چند مورد روشن، بسیاری از تعریف‌ها در گراف‌های جهت‌دار و گراف‌های بدون‌جهت به یکدیگر شبیه هستند؛ برای مثال، مسیرهای جهت‌دار و مسیرهای بدون‌جهت دقیقاً مانند یکدیگر تعریف می‌شوند، اما روشن است که جهت یال‌ها در مسیرهای جهت‌دار مشخص است. هرگاه درباره‌ی یکی از این دو دسته‌ی کلی گراف‌ها سخن می‌گوییم، نمادی متفاوت برای آن به کار نخواهیم برد. پس، برای مثال، اگر در مبحث گراف‌های جهت‌دار سخنی درباره‌ی مسیرها می‌گوییم، منظورمان مسیرهای جهت‌دار است.

کار را با مثالی ساده آغاز می‌کنیم که نخستین مسأله از نظریه‌ی گراف محسوب می‌شود: عبور از پل‌های شهر Königsberg. سپس درباره‌ی چگونگی کارهایی مانند پیمایش گراف، ترتیب‌دهی به رأس‌های گراف (یعنی یافتن یک رابطه‌ی ترتیب بین رأس‌های گراف – مترجمان)، یافتن کوتاه‌ترین

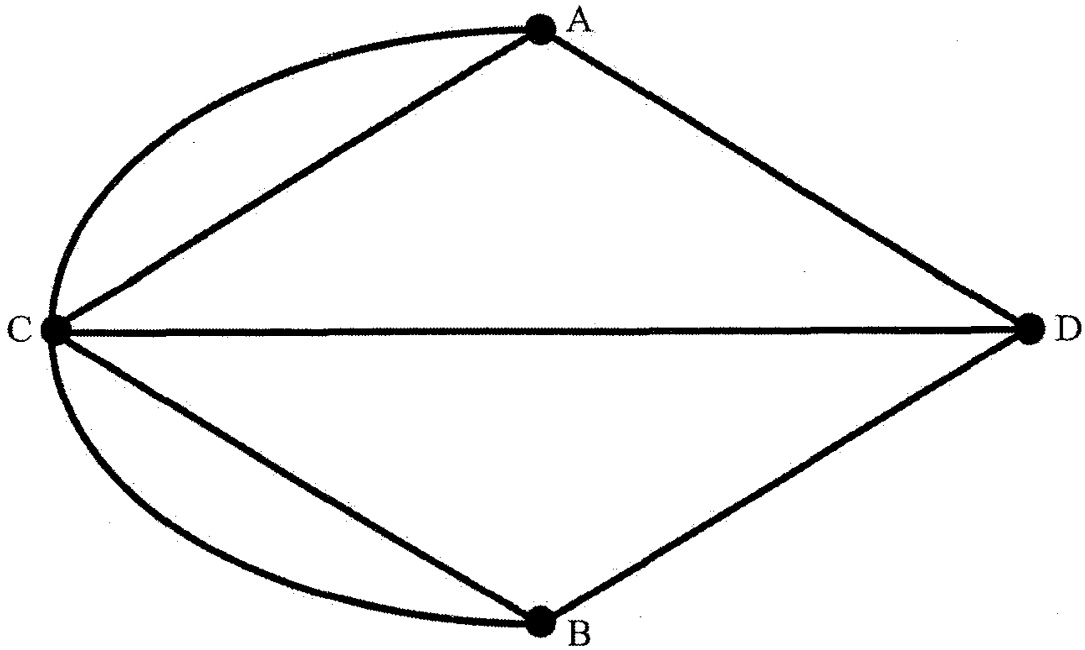
مسیرها در گراف و افراز گراف به بخش‌هایی برای برآوردن ویژگی‌های مشخص بحث می‌کنیم. در فصل ۱۰ نیز مبحثی درباره‌ی رابطه‌ی بین الگوریتم‌های گراف و الگوریتم‌های ماتریس آمده و چند الگوریتم دیگر گراف هم در آنجا ارائه شده است.

۷-۲ گراف‌های اویلری

گراف‌های اویلری از نخستین مسأله‌های طرح و حل‌شده در نظریه‌ی گراف هستند. سال ۱۷۳۶ میلادی، ریاضی‌دان سوئیسی، Leonhard Euler با این معما روبه‌رو شد. شهر Königsberg (که امروزه Kaliningrad نامیده می‌شود) بر دو کرانه‌ی رودخانه‌ی Pregel قرار داشت و دو جزیره‌ی درون رودخانه را نیز در بر می‌گرفت (شکل ۷-۱). بخش‌های مختلف شهر با هفت پل به یکدیگر متصل شده بودند. بسیاری از شهروندان برای حل این معما کوشیده بودند: «آیا می‌توان از جایی در شهر قدم زدن را آغاز کرد و پس از دقیقاً یک بار عبور از روی همه‌ی پل‌ها به همان نقطه‌ی شروع بازگشت؟» راه‌حل مسأله با تجرید آن به دست آمد. گراف شکل ۷-۲ معادل با مسأله‌ی شکل ۷-۱ است. از دیدگاه نظریه‌ی گراف، معما، یافتن مداری در گراف (در صوت وجود) است که هر یال گراف دقیقاً یک بار در آن دیده شود. راه دیگر طرح این معما چنین است: «آیا می‌توانیم گراف شکل ۷-۲ را بدون برداشتن مداد به گونه‌ای رسم کنیم که نقطه‌ی پایان ترسیم همان نقطه‌ی شروع آن باشد و از هیچ یالی بیش از یک بار نگذریم؟» اویلر این مسأله را حل کرد. او ثابت کرد چنین پیمایشی شدنی است، اگر و تنها اگر گراف مسأله، همبند و درجه‌ی همه‌ی رأس‌هایش زوج باشد. چنین گراف‌هایی گراف‌های اویلری نامیده می‌شوند. از آنجا که گراف شکل ۷-۲، رأس‌هایی با درجه‌ی فرد دارد، در نتیجه، مسأله‌ی پل‌های Königsberg حل‌شدنی نیست. برای این قضیه، برهانی بر پایه‌ی استقرا ارائه می‌شود که به الگوریتمی کارآمد، برای یافتن مسیر بسته‌ی مورد نظر منجر می‌گردد.



شکل ۷-۱ مسأله‌ی پل‌های Königsberg



شکل ۷-۲ گراف متناظر با مسأله‌ی پل‌های Königsberg

مسأله: گراف همبندی ((G=(V,E)) را به شما داده‌اند که درجه‌ی همه‌ی رأس‌هایش زوج است. مسیر بسته‌ی P را چنان بیابید که هر یال E دقیقاً یک بار در این مسیر بسته ظاهر شود.

به آسانی می‌توان ثابت کرد برای آن که چنین مسیر بسته‌ای وجود داشته باشد، باید درجه‌ی همه‌ی رأس‌ها زوج باشد: هنگام پیمایش یک مسیر بسته تعداد دفعات ورود به یک رأس با تعداد دفعات خروج از آن برابر است. از آنجا که قرار است از هر یال دقیقاً یک بار بگذریم، پس تعداد یال‌های گذرنده از هر رأس باید زوج باشد. برای آن که با استقرا ثابت کنیم چنین شرطی کافی است، نخست باید روشن سازیم استقرا روی کدام پارامتر بنا می‌شود. در آغاز می‌کوشیم بدون تغییر دادن مسأله، اندازه‌ی آن را کاهش دهیم. اگر یک رأس یا یک یال را حذف کنیم، ممکن است دیگر، درجه‌ی رأس‌های گراف حاصل زوج نباشد. باید مجموعه‌ی یال‌های S را به گونه‌ای برای حذف برگزینیم که تعداد یال‌هایی از S که از هر رأس v می‌گذرند، زوج باشد (توجه کنید که صفر هم زوج است). هر مدار، این شرط را برآورده می‌کند؛ پس پرسش این است که آیا یک گراف اویلری، همواره مدار دارد یا نه. فرض کنید با آغاز از رأس دل‌خواه v بدون آن که از یالی بیش از یک بار بگذریم، به ترتیبی دل‌خواه، شروع به پیمایش گراف کنیم. ادعا می‌کنیم این پیمایش سرانجام به v باز خواهد گشت، چراکه هر بار که وارد رأسی شویم، درجه‌ی آن رأس یک واحد کاهش یافته، عددی فرد می‌گردد؛ پس بی هیچ مشکلی می‌توانیم از آن خارج شویم. (توجه کنید که شاید این مدار همه‌ی یال‌ها را در بر نگیرد.)

اینک آماده‌ایم تا با بیان فرض استقرا، قضیه را ثابت کنیم. (با آن که استقرا روی مسیرهای بسته انجام می‌شود، بیان فرض روی تعداد یال‌ها از بیان آن روی تعداد مسیرها آسان‌تر است.)

فرض استقرا: یک گراف همبند با تعداد یال‌هایی کم‌تر از m که درجه‌ی همه‌ی رأس‌هایش زوج باشد، مسیری بسته دارد که در این مسیر هر یال دقیقاً یک بار آمده است و ما می‌دانیم چگونه این مسیر بسته را بیابیم.

گراف G=(V,E) را که m یال دارد، در نظر گرفته، فرض کنید P مسیری بسته در این گراف باشد. گرافی را که با حذف تمام یال‌های P از G به دست می‌آید، G' بنامید. درجه‌ی همه‌ی رأس‌های G' باید زوج باشد، چراکه تعداد یال‌های حذف‌شده‌ی گذرنده از هر رأس زوج است، اما بازهم نمی‌توان از فرض استقرا سود جست، چراکه شاید G' همبند نباشد. G'_1، G'_2، ... و G'_k را مؤلفه‌های همبند G' بگیرید. درجه‌ی تک‌تک رأس‌های هر مؤلفه‌ی همبند زوج است. همچنین، تعداد یال‌های هر مؤلفه (و بی‌شک تمام مؤلفه‌ها با همدیگر) از m کوچک‌تر است. به این ترتیب، اِعمال فرض استقرا به هر مؤلفه امکان‌پذیر می‌شود؛ یعنی هر مؤلفه، مسیری بسته دارد که هر یال آن مؤلفه دقیقاً یک بار در آن مسیر آمده است و ما می‌دانیم چگونه این مسیرهای بسته را پیدا کنیم. این k مسیر بسته را با P_1، P_2، ... و P_k نشان می‌دهیم. حال، لازم است همه‌ی این مسیرها را در قالب یک مسیر بسته چنان با هم ادغام کنیم که کل گراف را در بر گیرد. کار را با پیمایش رأسی دل‌خواه از P آغاز می‌کنیم تا آن که به نخستین رأسی برسیم که متعلق به یکی از مؤلفه‌های همبند باشد. این رأس را v_j و مؤلفه‌ی مربوط به آن را G'_j بگیرید. در اینجا، مسیر P_j را پیموده، دوباره به v_j باز می‌گردیم. می‌توان با پی‌گیری این شیوه، نخستین بار که با رأسی از یکی از این مؤلفه‌ها روبه‌رو می‌شویم، مسیر آن مؤلفه را بپیماییم. سرانجام به رأس آغازین P باز خواهیم گشت و بنابراین از همه‌ی یال‌های گراف دقیقاً یک بار گذاشته‌ایم. به این مسیر بسته، یک مدار اویلری می‌گویند؛ اما هنوز الگوریتم کامل نشده است، زیرا لازم است هم شیوه‌ای کارآمد برای یافتن این مؤلفه‌های همبند و هم روشی کارآمد برای پیمایش گراف بیابیم. اندکی بعد درباره‌ی این دو موضوع بحث خواهد شد. پیاده‌سازی الگوریتم مدار اویلری را به عنوان تمرین به خواننده واگذار می‌کنیم.

۷-۳ روش‌های پیمایش گراف

هنگام طراحی الگوریتم‌های گراف با این پرسش روبه‌رو می‌شویم که چگونه باید به ورودی بنگریم. این مسأله در فصل پیش به علت تک‌بعدی بودن ورودی، مسأله‌ای ساده و سرراست بود؛ چراکه به آسانی می‌توان دنباله‌ها و مجموعه‌ها را به ترتیب خطی پویش کرد، ولی پویش یک گراف که آن را پیمایش گراف هم می‌گوییم، به این سادگی نیست. دو الگوریتم برای پیمایش گراف ارائه می‌کنیم: جست‌وجوی نخست‌-ژرفا (DFS) و جست‌وجوی نخست‌-پهنا (BFS). (در بیش‌تر کتاب‌ها، DFS را جست‌وجوی عمق‌-اول و BFS را جست‌وجوی سطح‌-اول ترجمه کرده‌اند - مترجمان) بیش‌تر الگوریتم‌های این فصل احتمالاً به گونه‌ای به یکی از این دو شیوه برمی‌گردند.

۷-۳-۱ جست‌وجوی نخست-ژرفا

انجام جست‌وجوی نخست-ژرفا در گراف‌های جهت‌دار و گراف‌های بدون‌جهت تقریباً یکسان است، اما چون می‌خواهیم چندین ویژگی را هم در این دو دسته گراف بررسی کنیم و این ویژگی‌ها در هر دسته متفاوت از دیگری است، پس این بحث را برای هر یک از این دو دسته گراف به طور جداگانه ارائه می‌کنیم.

گراف‌های بدون‌جهت

فرض کنید گراف بدون‌جهت G=(V,E) را از روی یک نمایشگاه آثار هنری (شامل چند راهرو که نقاشی‌هایی به دیوارهای راهروهای آن آویزان است) ساخته باشیم و بخواهیم از نمایشگاه به گونه‌ای بگذریم که تمام نقاشی‌ها را تماشا کنیم. (فرض می‌کنیم جهت حرکت هر چه باشد، هنگام عبور از هر راهرو نقاشی‌های هر دو سوی آن را می‌بینیم.) هنگامی که گراف اویلری باشد، می‌توانیم طوری در نمایشگاه قدم بزنیم که از هر راهرو دقیقاً یک بار بگذریم، اما فعلاً فرض نکرده‌ایم که گراف G اویلری است؛ پس اجازه داریم از هر یالی هر چند بار که می‌خواهیم بگذریم (هنگامی که بحث به نتیجه برسد، خواهید دید که هر یال دقیقاً دو بار پیموده شده است). ایده‌ی الگوریتم جست‌وجوی نخست-ژرفا چنین است: درون نمایشگاه قدم می‌زنیم و به محض آن که توانستیم، وارد راهروی تازه‌ای می‌شویم. نخستین باری که به یک تقاطع می‌رسیم، در آنجا یک سنگ‌ریزه می‌گذاریم و وارد راهروی دیگری می‌شویم (مگر آن که راهروی تازه بن‌بست باشد) اما اگر به تقاطعی رسیدیم که در آنجا سنگ‌ریزه وجود داشت، به همان راهرویی که در آن بودیم، بازمی‌گردیم و می‌کوشیم راهرویی بیابیم که تا به حال وارد آن نشده‌ایم. هنگامی که تمام مسیرهای خارج‌شونده از یک تقاطع را دیدیم، سنگ‌ریزه را از آن تقاطع برمی‌داریم و به راهرویی وارد می‌شویم که در ابتدا از آنجا آمده بودیم؛ یعنی دوباره وارد این تقاطع نمی‌شویم. (منظور از برداشتن سنگ‌ریزه تنها تمیز کردن نمایشگاه است، پس این کار جزئی ضروری از الگوریتم نیست.) هر بار کوشیدیم راهروی تازه‌ای را بررسی کنیم. پس از بررسی تمام راهروهای هر تقاطع نیز از همان راهرویی که از آن وارد تقاطع شده بودیم، بازگشتیم. این شیوه را از این رو جست‌وجوی نخست-ژرفا می‌گویند که هر بار کوشیدیم به یک راهروی تازه برویم؛ یعنی در نمایشگاه به ژرفای بیش‌تری نفوذ کنیم. سودمندی اصلی روش DFS در شیوه‌ی آن برای تقسیم گراف، به همراه قابلیت اجرای بازگشتی آن روی بخش‌های تقسیم‌شده است.

DFS را به شکل قدم زدن و علامت‌گذاری با سنگ‌ریزه توضیح دادیم. حال، ببینیم چگونه DFS برای گراف بدون‌جهتِ داده‌شده با یک لیست همسایگی پیاده‌سازی می‌شود. پیمایش گراف را از رأس دل‌خواهی همچون r آغاز می‌کنیم و آن را ریشه‌ی DFS می‌نامیم. به ریشه علامت «مشاهده‌شده»

می‌زنیم. یک رأس دل‌خواه از رأس‌های علامت‌نخورده‌ی متصل به r برمی‌گزینیم و آن را r_1 می‌نامیم. حال، عمل DFS را به صورت بازگشتی با شروع از r_1 انجام می‌دهیم. بازگشت‌ها زمانی متوقف می‌شوند که به رأسی همچون v برسیم که تمام همسایگان آن علامت‌خورده باشند. اگر پس از آن که DFS روی r_1 به پایان رسید، تمام رأس‌های همسایه‌ی r علامت‌خورده باشند، DFS روی r نیز به پایان می‌رسد؛ در غیر این صورت، رأس علامت‌نخورده‌ی دل‌خواهی همچون r_2 را از رأس‌های متصل به r برمی‌گزینیم و DFS را با شروع از r_2 انجام می‌دهیم و به همین ترتیب کار را ادامه می‌دهیم تا هنگامی که همه‌ی رأس‌ها مشاهده شوند.

معمولاً از پیمایش گراف هدفی داریم؛ یعنی الگوریتم DFS انجام می‌شود و گراف را می‌پیماییم تا کاری را روی رأس‌ها یا یال‌های آن انجام دهیم. پیمایش «پیش‌ترتیب» گراف یعنی کار مورد نظر را (هر چه که باشد) هنگام رسیدن به یک رأس یا یال و علامت‌گذاری آن انجام می‌دهیم و پیمایش «پس‌ترتیب»، یعنی عمل مورد نظر پس از بازگشت از یک یال، یا پس از آن انجام می‌شود که دریافتیم یال به یک رأس مشاهده‌شده می‌رسد. برگزیدن روش پیش‌ترتیب یا روش پس‌ترتیب به مسأله‌ای که DFS را برای آن به کار می‌بریم، بستگی دارد. با یاری این دو اصطلاح جای اعمال را در کاربردهای گوناگون به صورت پیش‌ترتیب یا پس‌ترتیب مشخص می‌کنیم. الگوریتم DFS در شکل ۷-۳ داده شده است. در فراخوانی بازگشتی الگوریتم این شکل، v رأس آغاز است. برای سادگی در ابتدا گراف را همبند در نظر می‌گیریم. در شکل ۷-۴ مثالی از اجرای الگوریتم DFS آورده شده است که در آن اعداد روی رأس‌ها نشان‌دهنده‌ی ترتیب پیمایش آن‌هاست.

الگوریتم: Depth_First_Search(G,v)

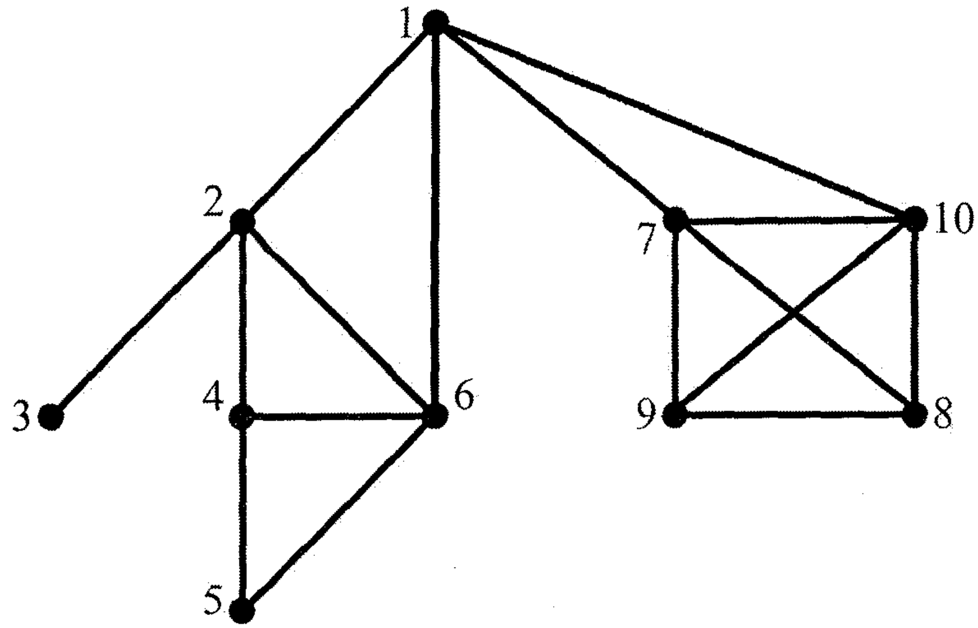
ورودی: G=(V,E) (یک گراف همبند بدون‌جهت) و v (رأسی از G)

خروجی: وابسته به کاربرد مورد نظر است.

```
begin
    v; را علامت بزن
    انجام کار پیش‌ترتیب مورد نظر روی v ;        {کار مورد نظر وابسته به هدفی
                                                 است که از پیمایش گراف داریم.}
    for تمام یال‌های (v,w) do
        if w علامت‌نخورده است then Depth_First_Search(G,w);
        انجام کار پس‌ترتیب مورد نظر روی یال (v,w) ;

        {انجام کار مورد نظر وابسته به هدفی است که از پیمایش گراف داریم؛ گاهی این کار
         تنها روی یال‌هایی انجام می‌شود که به رأس‌های تازه علامت‌خورده می‌روند.}
end
```

شکل ۷-۳ الگوریتم Depth_First_Search (در متن اصلی کتاب، شماره‌ی این شکل به اشتباه ۷-۴ درج شده است - مترجمان)



شکل ۷-۴ اجرای DFS بر روی یک گراف بدون‌جهت

لم ۷-۱

اگر G همبند باشد، با اجرای الگوریتم Depth_First_Search تمام رأس‌ها علامت‌گذاری می‌شوند و در طی اجرای الگورتیم نیز هر یال دست‌کم یک بار دیده خواهد شد.

برهان: فرض کنید این‌گونه نباشد و U را مجموعه‌ی رأس‌های علامت‌نخورده بگیرید. از آنجا که G همبند است دست‌کم یکی از رأس‌های U باید به یک یا بیش از یک رأس از رأس‌های علامت‌خورده متصل باشد، اما چنین چیزی ممکن نیست، چون هر بار که به یک رأس می‌رسیم و آن را علامت‌گذاری می‌کنیم، به سراغ تمام رأس‌های همسایه‌ی آن هم می‌رویم و آن‌ها را نیز علامت‌گذاری می‌کنیم؛ بدین ترتیب تمام رأس‌ها پیموده خواهند شد و از آنجا که پس از دیدن هر رأس، تمام یال‌های آن را هم بررسی می‌کنیم، پس تمام یال‌های گراف نیز پیموده خواهند شد.

□

اگر گراف ورودی یعنی G=(V,E) همبند نباشد، باید DFS را اندکی تغییر دهیم. هرگاه تمام رأس‌ها در دور نخست علامت بخورند، گراف همبند است و کار به پایان می‌رسد؛ در غیر این صورت، باید دوباره کار را از یک رأس دل‌خواه علامت‌نخورده آغاز کنیم و یک DFS دیگر انجام دهیم و این کار را تا پیمایش کامل گراف ادامه دهیم. بنابراین می‌توانیم DFS را برای فهمیدن این که گراف همبند است یا نه و یافتن مؤلفه‌های همبند آن به کار ببریم. این الگوریتم در شکل ۷-۵ آورده شده است. ما عموماً با گراف‌های همبند کار خواهیم کرد، چون اگر گراف همبند نباشد، می‌توانیم هر یک از مؤلفه‌های همبند آن را جداگانه در نظر بگیریم. پس DFS را همان‌گونه که در شکل ۷-۳ آمده است، به کار می‌بریم؛ بدون آن که به صراحت بگوییم این الگوریتم ممکن است چند دور اجرا گردد.

الگوریتم: Connected_Components(G)

ورودی: G=(V,E) (یک گراف بدون‌جهت)

خروجی: برای هر رأس v، v.Component نشان‌دهنده‌ی شماره‌ی مؤلفه‌ی همبندی از گراف خواهد شد که دربردارنده‌ی رأس v است.

```
begin
      Component_Number := 1;
      while رأس علامت‌نخورده‌ای همچون v وجود دارد do
            Depth_First_Search(G,v);
                                    {این عمل پیش‌ترتیب انجام شود:
            {v.Component_Number:=Component_Number;
            Component_Number := Component_Number + 1
end
```

شکل ۷-۵ الگوریتم Connected_Components

پیچیدگی: روشن است که هر یال دقیقاً یک بار از هر سوی خود (یعنی در مجموع دو بار) پیموده می‌شود. پس زمان اجرای الگوریتم متناسب با تعداد یال‌هاست. به علاوه ممکن است گراف تعدادی رأس هم داشته باشد که به هیچ جا متصل نباشند (و تمام این رأس‌ها نیز باید بررسی شوند). پس باید $O(|V|)$ را هم به عبارت زمان اجرا بیفزاییم. بنابراین کل زمان اجرا از $O(|V|+|E|)$ خواهد بود.

ساخت درخت DFS

حالا دو کاربرد ساده‌ی DFS را ارائه می‌کنیم: شماره‌گذاری یال‌ها با اعداد DFS و ساخت یک درخت پیمایش خاص که آن را درخت DFS می‌نامیم. اعداد درخت DFS ویژگی خاصی دارند که حتا اگر درخت را به طور صریح هم نسازیم، در بسیاری الگوریتم‌ها سودمند خواهد بود. با در نظر گرفتن این اعداد درک بسیاری از الگوریتم‌ها آسان‌تر می‌شود. برای توصیف این الگوریتم‌ها تنها لازم است تعیین کنیم که شیوه‌ی پیش‌ترتیب را به کار برده‌ایم یا شیوه‌ی پس‌ترتیب را. الگوریتم شماره‌گذاری رأس‌ها با اعداد DFS در شکل ۷-۶ و الگوریتم ساخت درخت DFS در شکل ۷-۷ آمده است. لازم نیست این دو الگوریتم جدا از هم اجرا شوند.

الگوریتم: DFS_Numbering(G,v)

ورودی: G=(V,E) (یک گراف بدون‌جهت) و v (یک رأس از G)

خروجی: به ازای هر رأس v، v.DFS شماره‌ی DFS برای v خواهد بود.

DFS_Number := 1; {مقداردهی اولیه}

این اعمال را در DFS به صورت پیش‌ترتیب به کار ببرید:

```
v.DFS := DFS_Number;
DFS_Number := DFS_Number + 1;
```

شکل ۷-۶ الگوریتم DFS_Numbering

الگوریتم: Build_DFS_Tree(G,v)

ورودی: G=(V,E) (یک گراف بدون‌جهت) و v (یک رأس از G)

خروجی: T (یک درخت DFS از G که در آغاز تهی است.)

این عمل را در DFS به صورت پس‌ترتیب به کار ببرید:

if w علامت‌نخورده است then یال (v,w) را به T بیفزا;

{این دستور را می‌توان به فرمان if، از خط ۴ الگوریتم Depth_First_Search افزود.}

شکل ۷-۷ الگوریتم Build_DFS_Tree

رأسی همچون v را «بالادست» رأس w در درخت T با ریشه‌ی r گوییم، اگر v روی مسیر یکتای موجود از w به r در T باشد. اگر v بالادست w باشد، w را «پایین‌دست» v گوییم.

☐ **لم ۷-۲** (ویژگی اصلی درخت‌های DFS برای گراف‌های بدون‌جهت)

اگر G=(V,E) یک گراف همبند بدون‌جهت و T=(V,F) یکی از درخت‌های DFS در G باشد که با الگوریتم Build_DFS_Tree ساخته شده است، آنگاه هر یال مانند e ($e \in E$) یا متعلق به T است (یعنی $e \in F$) و یا دو رأس از G را به هم متصل می‌کند که یکی از آن‌ها در T بالادست دیگری است.

برهان: اگر (v,u) یک یال G باشد، فرض کنید با الگوریتم DFS، v پیش از u دیده شود. پس از آن که v را علامت زدیم، DFS را از رأس‌های علامت نخورده‌ی همسایه‌ی v آغاز می‌کنیم. چون u همسایه‌ی v است، پس یا DFS از u آغاز شده است که در این حالت یال (v,u) جزو درخت T است و یا DFS پیش از عقب‌گرد از رأس v، u را دیده است که در این حالت u در درخت T پایین‌دست v است.

☐

به عبارت دیگر DFS به سراغ یال‌های جانبی (یعنی یال‌هایی که بین رأس‌ها علاوه بر مسیرهای درخت، مسیرهای جانبی به وجود می‌آورند) نمی‌رود و چنان که بعداً خواهیم دید، پرهیز از این‌گونه یال‌ها در روال بازگشتی‌ای که روی گراف اعمال می‌گردد، اهمیت دارد.

از آنجا که DFS الگوریتمی پراهمیت است، نسخه‌ای غیربازگشتی هم از آن ارائه می‌دهیم. ابزار اصلی برای پیاده‌سازی بازگشتی برنامه‌ها پشته است که اطلاعات لازم برای برگشت از فراخوانی‌های بازگشتی تودرتو را در خود نگه می‌دارد. کامپایلر، تمام داده‌های محلیِ مربوط به هر فراخوانی از هر روال بازگشتی را روی پشته نگه می‌دارد. پس هرگاه یکی از فراخوانی‌های بازگشتی به پایان برسد، می‌توانیم (بدون کوچک‌ترین تغییری در اطلاعات) دقیقاً به همان نقطه‌ی فراخوانی در روال صدازننده بازگردیم (که ممکن است فراخوانی دیگری از همین روال بازگشتی باشد). یکی از دلایل کاراتر بودن روال‌های غیربازگشتی این است که بیش‌تر اوقات لازم نیست تمام داده‌های محلی روی پشته نگه‌داری شوند. نسخه‌ی غیربازگشتی‌ای که بعداً ارائه می‌دهیم، نمونه‌ی خوبی از تبدیل یک برنامه‌ی بازگشتی به یک برنامه‌ی غیربازگشتی است.

یک مشکل عمده در تبدیل روال بازگشتی به نسخه‌ای غیربازگشتی از آن، لزوم نگه‌داری صریح محل بازگشت است. در داخل یک حلقه‌ی for، DFS را به طور بازگشتی فراخوانی می‌کنیم و از برنامه انتظار داریم محلی را به خاطر بسپارد که پس از پایان فراخوانی بازگشتی، اجرا باید از آنجا ادامه یابد. در گونه‌ی غیربازگشتی روال، خودمان باید این اطلاعات را نگه‌داری کنیم. فرض می‌کنیم هر رأس، لیستی پیوندی (به ترتیبی معین) از یال‌های گذرنده از خود دارد. (الگوریتم DFS هم به همین ترتیب به پیمایش گراف می‌پردازد.) v.First به ابتدای این لیست اشاره می‌کند و هر عنصر لیست، رکوردی شامل دو متغیر است که یکی از آن‌ها (Vertex) نشان‌دهنده‌ی رأس دیگر یال است و دیگری (Next) به عنصر بعدی لیست اشاره می‌کند. در آخرین یال هم مقدار مؤلفه‌ی Next را nil می‌گذاریم. DFS مانند پیش، تا جایی که نتواند رأس تازه‌ای بیابد، به پیمایش درخت می‌پردازد. در طی جست‌وجو، در یک پشته، رأس‌هایی را که روی مسیر ریشه به رأس فعلی قرار دارند، به ترتیب نگه می‌دارد. پس برای هر دو رأس Parent و Child در پشته اشاره‌گری به یک یال از Parent نگه‌داری می‌شود که این یال در پیمایش با DFS، هنگام عقب‌گرد از Child، یال بعدی است. نسخه‌ی غیربازگشتی DFS در شکل ۷–۸ آمده است.

الگوریتم: Nonrecursive_Depth_First_Search(G,v)

ورودی: G=(V,E) (یک گراف همبند بدون‌جهت) و v (یک رأس از G)

خروجی: وابسته به کاربرد است.

{در اینجا برخلاف بقیه‌ی فصل، نماد اشاره‌گر زبان پاسکال، یعنی ^ را به صورت صریح به کار می‌بریم.}

```
begin
      while رأس علامت‌نخورده‌ای همچون v وجود دارد do
            v را علامت بزن;
            روی رأس v کارهای پیش‌ترتیب مورد نظر را انجام بده;
            Edge := v.First;
            v و Edge را به سر پشته وارد کن;
            Parent := v;
            {تا اینجا مقداردهی اولیه انجام شد؛ حال به حلقه‌ی اصلی بازگشت می‌پردازیم.}
            while پشته تهی نیست do
                  سر پشته را بردار و مقدار آن را در v بریز;
                  while Edge ≠ nil do
                        Child := Edge^.Vertex;
                        if Child علامت‌نخورده است then
                           Child را علامت بزن;
                           روی Child کارهای پیش‌ترتیب مورد نظر را انجام بده;
                           Edge^.Next را بالای پشته قرار بده;
                           {این عمل برای این بود که پس از انجام کارهای مورد
                           نظر روی Child بتوانیم به یال بعدی بازگردیم.}
                           Edge := Child.First;
                           Parent := Child;
                           Parent را به سر پشته وارد کن;
                        else   {یعنی Edge یک یال عقب‌رو است.}
                           روی (Parent,Child) کارهای پس‌ترتیب دل‌خواه را انجام بده;
                           {اگر کارهای پس‌ترتیب تنها برای یال‌های درختی لازم باشد،
                           Edge := Edge^.Next;   از این تکه چشم‌پوشی می‌کنیم.}
                  سر پشته را بردار و مقدار آن را در Child قرار بده;
                  if پشته خالی نیست then
                  {پشته هنگامی خالی می‌شود که Child ریشه باشد.}
                    دو مقدار بالای پشته را (بدون حذف) به ترتیب در Edge و Parent بریز;
                    روی (Parent,Child) کارهای پس‌ترتیب مورد نظر را انجام بده
end
```

شکل ۷-۸ الگوریتم Nonrecursive_Depth_First_Search