فصل

۱

معرفی

هدف اصلی این کتاب ارائه‌ی مبانی نظری مورد نیاز برای کسب مهارت لازم در «حل مسئله[۱]» به‌کمک کامپیوتر است. یک کارشناس «علم و مهندسی کامپیوتر[۲]» در طول تحصیل خود باید به توانایی حل مسئله‌های مختلف به‌کمک یک یا چند زبان برنامه‌نویسی مجهز شود. در این فصل با ذکر مثالی، مراحل مختلف حل یک مسئله را تشریح می‌کنیم و با مفاهیم اولیه‌ی هر مرحله آشنا می‌شویم. برخی از این مفاهیم را ابتدا به‌صورت شهودی مطرح می‌کنیم، بدون آن‌که وارد جزئیات شویم. انتظار می‌رود خواننده‌ای که با این عناوین آشنا نیست، ابتدا تصویر مناسبی از این مفاهیم دریافت کند و از ارتباط بین آن‌ها مطلع شود. این برداشت اولیه کمک می‌کند تا با خواندن جزئیات این مفاهیم در فصل‌های بعد، عمق دانش کسب شده‌ی او بیش‌تر شود.[۳]

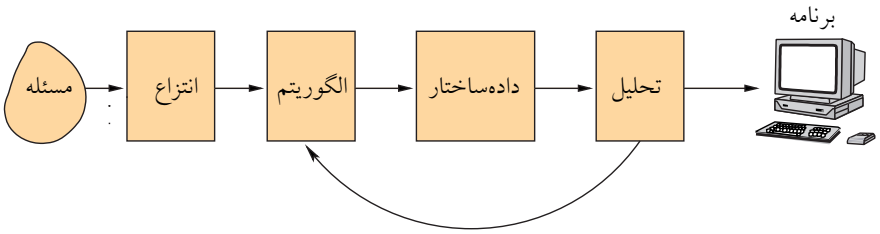
شکل ۱–۱ نمایی کلی از مراحل مختلف حل یک مسئله را نشان می‌دهد. یک مسئله‌ی واقعی حاوی جنبه‌ها و ویژگی‌های مختلف و اغلب متعددی است که به برخی از آن‌ها در طراحی راه‌حل و پیاده‌سازی چندان نیازی نیست. گام اول در حل هر مسئله، شناسایی کامل و تفکیک این ویژگی‌هاست، به‌طوری‌که پس از پالایش، از آن بتوان یک مدل ساده‌ی قابل

---

[۱] problem solving

[۲] computer science and engineering

[۳] به این روش آموزش سطح‌ـاول (breadth-first) گفته می‌شود که برنامه‌های پیش‌نهادی انجمن‌های معتبر اِی‌سی‌اِم و آی‌تریپل‌ای برای رشته‌ی کامپیوتر آن‌را توصیه می‌کنند.



شکل ۱-۱ نمایی کلی از مراحل مختلف حل یک مسئله.

حل ساخت. به گونه‌ی پالایش‌شده‌ی مسئله، مدل «انتزاعی[۴]» می‌گوییم. این عمل انتزاع، یا پالایش ابعاد و ویژگی‌های مسئله، یکی از مفاهیم مهم در علم کامپیوتر است که در بسیاری از مراحل حل مسئله‌ها به‌کار می‌رود. برای یک مسئله می‌توان سطوح مختلفی از انتزاع را تصور کرد که یک سطح آن تبدیل مسئله‌ی واقعی به مدلی ریاضی است که ممکن است چندان شباهتی به مسئله‌ی اصلی نداشته باشد. یافتن یک مدل انتزاعی خوب از مسئله‌ی مورد نظر، که در مورد آن قبلاً مطالعه شده باشد، بخش مهمی از هنر حل مسئله است. فهم مناسب از استقرا، ترکیبیات و ریاضی گسسته به استخراج مدل انتزاعیِ قابل حل کمک شایان می‌کند.

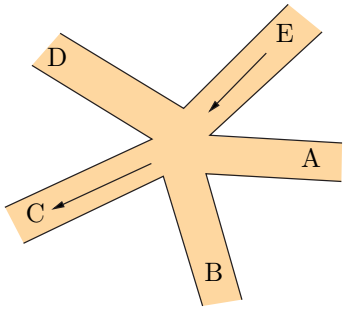
پس از تبدیل مسئله به یک مدل انتزاعی قابل حل با کامپیوتر، الگوریتم حل مسئله‌ی ساده‌شده طراحی می‌شود. در ابتدا، همه‌ی جزئیات این الگوریتم مشخص نیست. اما در همین مرحله، الگوریتم طراحی شده مشخص می‌کند که داده‌های مسئله باید به‌وسیله‌ی چه ساختارهایی عرضه شود و بر روی هر قلم داده‌ی آن چه اَعمالی انجام می‌گیرد. مجموعه‌ی این اطلاعات کمک می‌کند تا برای ذخیره‌ی داده، «داده‌ساختار[۵]»های مناسب انتخاب یا طراحی شوند. الگوریتم طراحی شده را با توجه به داده‌ساختارهای انتخابی می‌توان در همین مرحله تحلیل کرد و میزان زمان اجرا و حافظه‌ی مصرفی الگوریتم را برای اندازه‌های مختلف ورودی حدس زد. اگر این الگوریتم پس از تحلیل راضی‌کننده نبود، لازم است مراحل طراحی الگوریتم و انتخاب داده‌ساختارهای مورد نیاز تکرار شود. توجه کنید که این مراحل معمولاً مستقل از یک زبان برنامه‌نویسی خاص است. پس از نهایی شدن الگوریتم، می‌توان با استفاده از یک زبان برنامه‌نویسی مناسب، الگوریتم مطلوب را پیاده‌سازی کرد و در عمل مورد آزمون قرار داد.

---

[۴] abstract
[۵] data structure

۱-۱ یک مثال: برنامه‌ریزی چراغ‌های راهنما

فرض کنید می‌خواهیم برنامه‌ریزی چراغ‌های راهنمای محل تقاطعی در نزدیکی دانشگاه را که در شکل ۱-۲ نشان داده شده است به‌عهده بگیریم. تعدادی از خیابان‌های منتهی به این تقاطع یک‌طرفه هستند که با پیکان نمایش داده شده‌اند. باید مشخص کنیم که چراغ راهنمای این تقاطع چندزمانه است و در هر بازه‌ی زمانی آن، در چه مسیرهایی مجاز به حرکت و در کدام مسیرها مجبور به توقف پشت چراغ قرمز هستند. این‌کار را می‌خواهیم طوری انجام دهیم که تعداد زمان‌های مختلف چراغ راهنمایی کمینه باشد، و نیز در هر زمان، تا جایی که ممکن است حرکت‌ها در مسیرهای مجاز به عبور باشند.



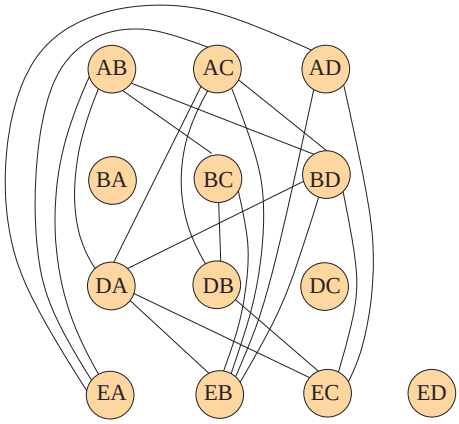
شکل ۱-۲ یک تقاطع از خیابان‌ها.

در این مسئله بار ترافیکی خیابان‌ها مهم نیستند یا مساوی در نظر گرفته می‌شوند. در واقع اطلاعات ترافیکی مانند میزان ترافیک، عرض و تعداد مسیرها در هر خیابان نمونه‌هایی از ویژگی‌های مسئله هستند که تأثیری در راه‌حل ندارند و در مرحله‌ی انتزاع نادیده گرفته می‌شوند. شما هم می‌توانید اطلاعات مشابه دیگری را نام ببرید.

توجه کنید که هدف ما، حل این مسئله برای هر ورودی دل‌خواه است که می‌تواند شامل تعداد نامشخص و شاید زیادی خیابان باشد. البته برای یک مسئله‌ی خاص مانند شکل داده شده، می‌توان از روش‌های دیگر مسئله را حل کرد.

در مدل انتزاعی این مسئله آن‌چه مهم است «گردش[۶]»های مختلف این تقاطع و تلاقی این گردش‌هاست. بدیهی است که امکان حرکت گردش‌های متلاقی در یک بازه‌ی زمانی وجود ندارد. اگر گردش از خیابان X به خیابان Y را XY بنامیم، خود گردش‌ها و تلاقی بین

[۶] turn



شکل ۱–۳ گراف تقاطع مثال گفته‌شده در شکل ۱–۲.

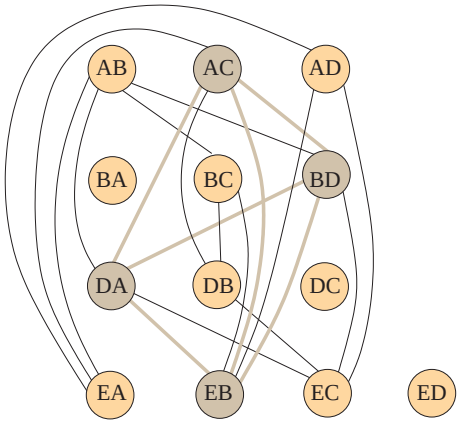
آن‌ها را می‌توان با یک گراف نشان داد که آن‌را «گراف تقاطع[۷]» می‌نامیم. در این گراف، هر گردش یک رأس است و بین دو گردش متلاقی یک یال قرار داده‌ایم. شکل ۱–۳ گراف تقاطع مثال شکل ۱–۲ را نشان می‌دهد. این همان مدل انتزاعی مسئله است که هیچ شباهتی به مسئله‌ی اصلی ندارد. این مدل از این نظر مناسب است که مباحث زیادی برای نظریه و الگوریتم‌های گراف موجود است.

حال باید خود چراغ راهنمایی و بازه‌های زمانی آن‌را نیز مدل کنیم. چراغ راهنمایی یک تقاطع اصطلاحاً چندزمانه است و تعداد زمان‌های آن با توجه به جهت‌های حرکت (گردش به راست، چپ، و غیره) مشخص می‌شود. برای سادگی کار فرض می‌کنیم چراغ مورد نظر k زمان دارد و با رنگ‌های ۱ تا k مشخص شده است (مقدار k را الگوریتم تعیین می‌کند). همچنین در ابتدای هر خیابان محل تقاطع، تابلویی نصب شده است که نشان می‌دهد هر گردش فقط در چه رنگ‌هایی از چراغ مجاز به حرکت است. البته چنان‌چه می‌دانید این کار در عمل به‌صورت دیگری انجام می‌شود.

در مدل گراف، مسئله انتساب یک یا چند رنگ به هر رأس گراف تقاطع است به‌طوری که دو رأسی که دو انتهای یک یال هستند رنگ یک‌سان نداشته باشند. تعداد این رنگ‌ها همان k است و ما می‌خواهیم کم‌ترین مقدار آن‌را به‌دست بیاوریم.

این یک مسئله‌ی شناخته‌شده در نظریه‌ی گراف‌هاست که «رنگ‌آمیزی گراف[۸]» نامیده می‌شود. در این مسئله، یک گراف G با مجموعه‌ی رأس‌های V و یال‌های E داده شده است و هدف، پیداکردن کمینه‌ی تعداد رنگ‌هاست که رأس‌های گراف را بتوان با آن رنگ

[۷] conflict graph
[۸] graph coloring



شکل ۱-۴ یک خوشه‌ی چهار رأسی در گراف تقاطع.

کرد به‌طوری‌که دو رأس همسایه (مجاور) یک‌رنگ نباشند. در مورد این مسئله‌ی انتزاعی اطلاعات زیادی وجود دارد، مثلاً می‌دانیم که این مسئله «ان‌پی‌ـسخت[۹]» است.

روشن است که اگر گراف G یک زیرگراف G' با r رأس و به شکل خوشه[۱۰] داشته باشد، برای رنگ‌آمیزی G' و در نتیجه G به دست‌کم r رنگ نیاز است. شکل ۱-۴ یک خوشه‌ی ۴ رأسی در این گراف را نشان می‌دهد. بنابراین اگر بتوانیم در این گراف بزرگ‌ترین زیرگراف خوشه‌ای‌شکل با اندازه‌ی r پیدا کنیم، می‌توان کل گراف را با دست‌کم r رنگ، رنگ‌آمیزی کرد. مشکل این‌جاست که مسئله‌ی پیدا کردن بزرگ‌ترین زیرگراف خوشه‌ای‌شکل در یک گراف نیز یک مسئله‌ی ان‌پی‌ـسخت است.

از این آگاهی که مسئله‌ی مورد نظر ان‌پی‌ـسخت است، در می‌یابیم که نباید در حالت کلی به‌دنبال راه‌حل سریع برای حل مسئله باشیم، چرا که یک راه‌حل بهینه برای مسئله‌های بزرگ شامل گردش‌های زیاد نمی‌تواند در زمان معقولی جواب بهینه را تولید کند. ممکن است رضایت دهیم که یک الگوریتم سریع، تعداد رنگ‌های «نزدیک به بهینه[۱۱]» پیدا کند.

روش‌های مختلفی برای حل سریع ولی نزدیک به بهینه برای یک مسئله‌ی ان‌پی‌ـسخت

[۹] NP-Hard: مجموعه‌ی ان‌پی‌ـسخت شامل چند هزار مسئله‌ی مختلف با کاربردهای فراوان است که تاکنون برای آن‌ها راه‌حل «سریع» و قابل انجام در زمان معقول پیدا نشده است و به‌احتمال زیاد در آینده نیز یافت نخواهد شد؛ این‌که راه‌حل سریعی برای آن‌ها وجود ندارد هم اثبات نشده است. البته ثابت شده است که اگر فقط برای یکی از این مسئله‌ها راه‌حل سریعی پیدا شود، این راه‌حل موجب حل سریع بقیه‌ی مسئله‌ها خواهد شد. البته احتمال پیدا شدن چنین الگوریتمی ضعیف است. منظور از راه‌حل سریع آن است که زمان اجرای آن با اندازه‌ی ورودی مسئله به‌صورت چندجمله‌ای رابطه داشته باشد.

[۱۰] clique: یک گراف کامل که هر رأس آن به بقیه وصل باشد یک خوشه نامیده می‌شود.

[۱۱] near optimal

وجود دارد:

- راه‌حل تقریبی قابل اثبات[۱۲]: که در آن یک الگوریتم سریع برای حل مسئله ارائه می‌شود ولی اثبات می‌شود که اندازه‌ی خروجی (مثلاً تعداد رنگ‌ها در این مسئله) ضریبی از اندازه‌ی خروجی بهینه‌ی مسئله است.
- الگوریتم‌های «مکاشفه‌ای[۱۳]»: با این‌که الگوریتم‌هایی سریع هستند و به‌صورت تقریبی جواب را به‌دست می‌آورند، اما در مورد ضریب تقریب یا میزان خوبی الگوریتم اثباتی وجود ندارد. بسیاری از این الگوریتم‌ها به‌صورت تجربی آزمایش می‌شوند. برخی از این الگوریتم‌ها از روش «حریصانه[۱۴]» برای حل استفاده می‌کنند.

۱-۱-۱ یک راه‌حل حریصانه برای مسئله

یک راه‌حل حریصانه معمولاً ساده است و در هر مرحله از اجرا، تنها بر اساس اطلاعات محلی بخشی از جواب را پیدا می‌کند، به امید این‌که در انتها به جواب بهینه نزدیک شود. البته برخی الگوریتم‌های حریصانه راه‌حل بهینه‌ی مسئله‌های مهمی را به‌دست می‌آورند.[۱۵]

برای مسئله‌ی زمان‌بندی چراغ راهنما یک راه‌حل حریصانه ارائه می‌کنیم، اما در مورد میزان نزدیکی جواب به مقدار بهینه ادعایی نداریم. در این الگوریتم رأس‌های گراف به‌ترتیب دل‌خواهی شماره‌گذاری می‌شوند. در ابتدا رنگ شماره‌ی ۱ را برمی‌داریم و رأس شماره‌ی ۱ را با آن رنگ می‌کنیم. سپس بقیه‌ی رأس‌ها را به‌ترتیب شماره‌شان نگاه می‌کنیم و آن‌هایی را که امکان دارد (یعنی آن‌هایی که رأس مجاور همرنگ نداشته باشند) با رنگ ۱ رنگ می‌کنیم. در مرحله‌ی بعد، رنگ شماره‌ی ۲ را برمی‌داریم و اولین رأسی را که رنگ نشده است و سپس هر رأس دیگری را که بتوانیم با آن رنگ می‌کنیم. این کار را ادامه می‌دهیم تا همه‌ی رأس‌ها رنگ شوند.

کلیات این الگوریتم در رَویّه‌ی[۱۶] CROSS-SOLUTION نگارش شده است. ورودی آن گراف تقاطع G است که رأس‌ها و یال‌های آن به‌ترتیبی دل‌خواه شماره‌گذاری شده‌اند. در این الگوریتم از یک آرایه به‌نام $[]VerticesWithColor$ استفاده می‌شود که درایه‌ی i ام آن

[۱۲] provable approximation algorithm
[۱۳] heuristic
[۱۴] greedy
[۱۵] برخی از این الگوریتم‌ها به دلیل اهمیت‌شان به‌نام طراح‌شان ثبت شده‌اند، مانند الگوریتم‌های هافمن، دایکسترا، پریم، کروسکال و ....
[۱۶] routine

حاوی مجموعه‌ای از رأس‌هاست که با رنگ i، رنگ می‌شوند و برای به‌دست آوردن این مجموعه رَویّه‌ی GREEDYSOLUTION فراخوانده می‌شود.

```
CROSS-SOLUTION (G)
    ▷ Input: G گراف
    ▷ Output: مجموعه‌ای از رأس‌ها برای هر رنگ؛ رأس‌هایی که آن رنگ را دارند
    ▷ متغیرهای محلی: ColorNo و آرایه‌ی VerticesWithColor
1   MAKEGRAPH(G)
2   ColorNo← 0
3   while رأس رنگ نشده در G وجود دارد
4      do ColorNo← ColorNo+1
           ▷ همه‌ی رأس‌هایی را که بتوان با ColorNo رنگ کرد به‌دست آور
5          VerticesWithColor[ColorNo]← GREEDYSOLUTION (G)
6   return VerticesWithColor


GREEDYSOLUTION (G)
1   MAKEEMPTYSET (newcolor) ▷ مجموعه‌ی تهی newcolor را ایجاد کن
2   for G در v هر رأس
3     do if v رنگ نشده است
4           then if v مجاور هیچ رأسی در newcolor نباشد
5                   then به v برچسب «رنگ‌شده» بزن
6                        v را به مجموعه‌ی newcolor اضافه کن
7   return newcolor
```

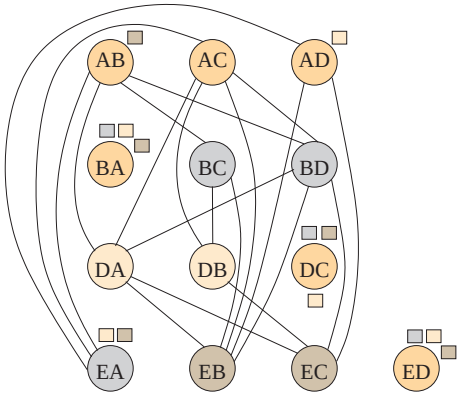
در رویه‌ی GREEDYSOLUTION فرض می‌شود که ممکن است تعدادی از رأس‌های گراف G قبلاً رنگ شده باشند. رأس‌های رنگ‌شده دارای برچسب «رنگ‌شده» (colored) هستند. این ویژگی‌ها به رأس‌های گراف منتسب می‌شوند و قابل خواندن هستند. این الگوریتم رأس‌ها را به‌ترتیب شماره‌شان بررسی می‌کند و اگر رأس v قبلاً رنگ نشده بود آن‌را «رنگ‌شده» ثبت می‌کند و به مجموعه‌ی $newcolor$ اضافه می‌کند. به‌سادگی می‌توان در این الگوریتم کاری کرد که یک رأس بیش از یک رنگ داشته باشد. این مشخص می‌کند که هر گردش در چه رنگ‌هایی از چراغ راهنما می‌تواند حرکت کند.

دقت کنید که رویه‌های فوق به‌صورت «شبه‌کد[۱۷]» نوشته شده‌اند که یک زبان

[۱۷] pseudo-code

برنامه‌نویسی نیست و ما می‌توانیم در آن از هر دستوری استفاده کنیم. بدیهی است که هنگام پیاده‌سازی واقعی، باید محدودیت‌های زبان برنامه‌نویسی را رعایت کنیم.

اگر رأس‌های گراف شکل ۱–۳ را از بالا به پایین و از چپ به راست شماره‌گذاری کنیم، حاصل الگوریتم فوق بر روی این گراف در شکل ۱–۵ نشان داده شده است که در آن رأس‌هایی که با چند رنگ بتوان رنگ کرد نیز مشخص شده‌اند. مثلاً گردش‌های BA، DC و ED در هر زمانِ چراغ راهنما مجاز به حرکت‌اند و مزاحم گردش دیگری نیستند.

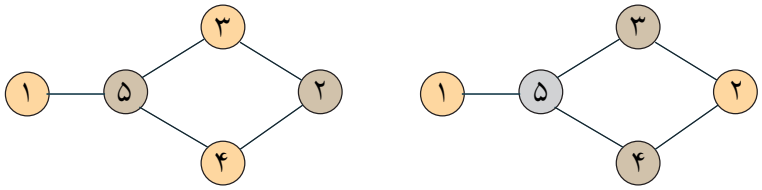


شکل ۱–۵ نتیجه‌ی انجام الگوریتم حریصانه.

این الگوریتم برای این مسئله ۴ رنگ به‌دست می‌آورد که با توجه به وجود بزرگ‌ترین خوشه‌ی ۴ رأسی در این گراف، این تعداد رنگ بهینه است. البته این جواب کاملاً وابسته به ترتیب بررسی رأس‌های گراف است و به‌دست آوردن جواب بهینه در این مثال خاص یک استثنا است. مثلاً اگر همین الگوریتم بر روی گراف شکل ۱–۶ و با ترتیبی از رأس‌ها که مشخص شده است اجرا شود، سه رنگ به‌دست می‌آورد (شکل ۱–۶ (الف))، در حالی که آن‌را با دو رنگ می‌توان رنگ کرد (شکل ۱–۶ (ب)).

۱–۱–۲ داده‌های مسئله

برای پیاده‌سازی الگوریتم حریصانه باید داده‌های مختلف آن‌را به‌صورتی ذخیره کنیم تا انجام اَعمال مختلف بر روی آن‌ها در زمان کوتاهی انجام شود. داده‌های اصلی این الگوریتم «گراف» و «مجموعه» است. قبل از مشخص کردن نحوه‌ی پیاده‌سازی آن‌ها، اَعمال مختلفی



(ب) (الف)

شکل ۱-۶ (الف) الگوریتم حریصانه این گراف را با ۳ رنگ، رنگ‌آمیزی می‌کند. (ب) رنگ‌آمیزی بهینه‌ی این گراف.

را که این الگوریتم بر روی این داده‌ها انجام می‌دهد فهرست می‌کنیم.

الگوریتم، اَعمال زیر را بر روی گراف G انجام می‌دهد:

- MakeEmpty (G): ایجاد یک گراف تهی،
- InsertVertex(G, v) و InsertEdge(G, u, v): درج یک رأس یا یک یال در G،
- NumVertices(G)، NumEdges(G): تعداد رأس‌ها و یال‌های G را برمی‌گرداند،
- ListofVertices(G): ترتیبی از رأس‌ها را تولید می‌کند،
- AdjacentVertices(G, v): رأس‌های مجاور یک رأس v را تولید می‌کند،
- ColorVertex(G, v): رأس v را رنگ می‌کند،
- IsColored(G, v): آیا رأس v رنگ شده است؟ و
- IsAdjacent(G, u, v): آیا رأس u مجاور رأس v است؟

و نیز بر روی مجموعه‌ی Set اَعمال زیر را انجام می‌دهد:

- MakeEmpty(S): ایجاد مجموعه‌ی تهی S،
- InsertSet(S, e): درج عنصر e در S، و
- ListofElements(S): در اختیار گذاردن لیست تک‌تک عناصر موجود در S.

علاوه بر این دو ساختار، متغیرهای محلی دیگری مانند آرایه‌ای از مجموعه‌ها و متغیرهای ساده‌ی دیگر مورد نیاز است.

در این مرحله، از داده‌ساختارهای مناسب برای پیاده‌سازی داده‌های مسئله، یعنی «گراف» و «مجموعه» استفاده می‌کنیم. داده‌ساختاری مناسب و کاراست که مجموعه‌ی اَعمال گفته‌شده را در زمان کوتاهی انجام دهد. داده‌ساختارهای شناخته‌شده‌ای برای هر کدام از این‌دو وجود دارند که آگاهی از آن‌ها و نیز نحوه‌ی تعریف و ساخت داده‌ساختارهای جدید، از مطالب مهمی است که بخش مهمی از این کتاب را تشکیل می‌دهد.

با انتخاب هر داده‌ساختار، زمان اجرای (یا هزینه‌ی انجام) هر یک از اَعمال گفته‌شده را برحسب اندازه‌ی داده‌ی ورودی تحلیل می‌کنیم. سپس، با روش‌هایی که در این کتاب خواهیم گفت زمان اجرای کل برنامه را در هر مرحله تخمین زده و در نهایت الگوریتم و داده‌ساختار کارایی را به‌عنوان راه‌حل انتخاب می‌کنیم. در انتها نیز، با استفاده از یک زبان برنامه‌نویسی مناسب، الگوریتم را پیاده‌سازی می‌کنیم.

۱-۲ گونه‌های مختلف داده

در یک برنامه، داده‌ها از طریق متغیرهای مختلفی در اختیار کاربر قرار می‌گیرند و این متغیرها خود به‌گونه‌های مختلفی پیاده‌سازی می‌شوند که به هر نوع آن یک «داده‌گونه[۱۸]» می‌گوییم. در زبان‌های «ساخت‌یافته[۱۹]» مانند پاسکال[۲۰] و سی[۲۱]، داده‌گونه‌ها به دسته‌های مختلف تقسیم می‌شوند: «داده‌گونه‌های ساده[۲۲]» که هر زبان برنامه‌نویسی آن‌را در ساده‌ترین شکل در اختیار کاربر قرار می‌دهد (مانند «عدد صحیح»، «عدد حقیقی»، «اشاره‌گر»، شاید «رشته» و نظایر آن). «داده‌گونه‌های مرکب[۲۳]» ترکیبی از داده‌گونه‌های ساده یا داده‌گونه‌های مرکبِ ساده‌تر است. مثلاً، «رکورد[۲۴]» از مؤلفه[۲۵]های مختلفی تشکیل شده است که هر یک از داده‌گونه‌ی دیگری است، یا «آرایه[۲۶]» از چندین درایه[۲۷] از یک داده‌گونه‌ی مشخص

---

[۱۸] data type
[۱۹] structured
[۲۰] Pascal
[۲۱] C
[۲۲] simple data types
[۲۳] compound data types
[۲۴] record
[۲۵] field
[۲۶] array
[۲۷] array entry

تشکیل شده است و به هر درایه‌ی آن می‌توان به‌طور مستقیم دست‌رسی داشت.

۱-۲-۱ داده‌گونه‌ی انتزاعی

در طراحی یک برنامه با زبان‌های ساخت‌یافته، تعریف داده به‌صورت انتزاعی انجام می‌شود که به آن «داده‌گونه‌ی انتزاعی[۲۸]» می‌گوییم. این نوع سازمان‌دهی داده از دو اصل پیروی می‌کند:

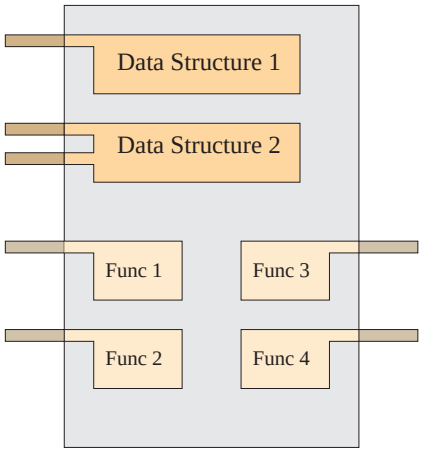
- انتزاع[۲۹]: پالایش داده و استخراج ویژگی‌های با اهمیت و تفکیک آن‌ها از بخش‌های کم اهمیت و جزیی، و
- بسته‌بندی[۳۰]: مخفی‌کردن جزییات پیاده‌سازی از دید کاربر؛ جزییات داده‌ساختارها و الگوریتم‌های استفاده‌شده در پیاده‌سازیِ اَعمال مختلف بر روی داده‌گونه‌ها، در اختیار کاربر قرار نمی‌گیرد؛ او تنها از عناوینِ داده‌گونه‌ها برای دست‌رسی به داده‌ی مورد نظر و نیز از عناوین رَویّه‌ها و پارامترهای هریک برای فراخوانی مطلع است.

یک داده‌گونه‌ی انتزاعی متشکل از دو قسمت است: داده‌ساختارهای مورد استفاده و نیز اَعمال[۳۱] مختلفی که بر روی آن انجام می‌شوند. شکل ۱-۷ یک داده‌گونه‌ی انتزاعی را نشان می‌دهد که از دو داده‌ساختار و چهار عمل بر روی این داده‌ساختارها تشکیل شده است؛ این داده‌گونه مانند یک آی‌سی است که فقط از طریق ۶ درگاه نشان داده شده می‌توان به آن دست‌رسی داشت.

۱-۲-۲ داده‌ها در زبان‌های شیءگرا

در زبان‌های «شیءگرا[۳۲]» برخلاف زبان‌های ساخت‌یافته، مجموعه‌ای از داده‌های مرتبط به‌هم و نیز اَعمال مربوط به آن‌ها در یک قالب و به اسم «شیء[۳۳]» تعریف می‌شوند. به بیانی دیگر، هر شیء مجموعه‌ای از داده‌ها و اَعمال روی آن‌هاست.

[۲۸] Abstract Data Type (ADT)
[۲۹] abstraction
[۳۰] encapsulation
[۳۱] operations
[۳۲] object oriented
[۳۳] object



شکل ۱–۷ داده‌گونه‌ی انتزاعی.

یک داده‌ساختار برای سهولت در انجام اَعمال روی مجموعه‌ای از داده طراحی می‌شود، پس به کمک شیءگرایی می‌توان مستقل از نحوه‌ی پیاده‌سازی داده‌ساختار، مجموعه‌ی رفتارها را به‌صورت یک «میانا[۳۴]» تعریف کرد و با روش پیاده‌سازی، که وابسته به نحوه‌ی نگه‌داری داده‌هاست، اَعمال را تعریف و از آن‌ها در پیاده‌سازی الگوریتم‌ها استفاده کرد.

به‌عنوان مثال، در زبان «جاوا[۳۵]» که یک زبان برنامه‌نویسی شیءگرا است میانایی به نام Set وجود دارد که تمام اَعمال مورد نظر روی «مجموعه» در آن تعریف شده است ضمن آن‌که الزامی در روش پیاده‌سازی این اَعمال ندارد. از طرفی، در این زبان «رده[۳۶]»های متعددی از شیءها وجود دارند که هرکدام این میانا را به روش خاصی پیاده‌سازی کرده است؛ از جمله رده‌ی LinkedList که یک مجموعه را با استفاده از داده‌ساختار «لیست پیوندی[۳۷]» پیاده‌سازی می‌کند و رده‌ی ArrayList که همان میانا را با استفاده از آرایه پشتیبانی می‌کند یا رده‌های HashMap و HashSet که با استفاده از «جدول درهم‌سازی[۳۸]» عملیات مربوط به «مجموعه» در آن پیاده‌سازی شده است.

این قابلیت زبان‌های شیءگرا از این نظر قابل توجه است که می‌توان صرف‌نظر از روش پیاده‌سازی داده‌ساختار و فقط به‌کمک میانای تعریف شده برای آن، الگوریتم مورد نظر را

---

[۳۴] interface یا واسط
[۳۵] java
[۳۶] class
[۳۷] linked list
[۳۸] hash table

طراحی کرد و سپس با توجه به معیارهای کارایی، روش پیاده‌سازی دل‌خواه را انتخاب نمود. مثلاً چنان‌چه حافظه‌ی مصرفی و پیمایش ترتیبی معیار انتخاب باشد بهتر است از LinkedList استفاده شود و اگر سرعت دست‌رسی و جست‌وجو مد نظر باشد انتخاب HashSet یا HashMap مناسب‌تر است.

اگر بخواهیم برای مسئله‌ی برنامه‌ریزی چراغ راهنما داده‌ساختارهایی در یک زبان شیءگرا مانند جاوا تعریف کنیم، می‌توان به‌صورت زیر عمل کرد: هر رأس، یال و گراف را یک شیء در نظر می‌گیریم که دارای مشخصات (داده‌ها) و رفتار (اَعمال) خاص خود است. یک رأس دارای داده‌ساختار ساده‌ای برای نگه‌داری شماره، رنگ و وضعیت آن (رنگ‌شده، رنگ‌نشده) و نیز رفتارهایی برای انجام عملیات ایجاد، تعیین و گرفتن هر کدام از مشخصات آن و نیز مقایسه‌ی دو رأس است.

هر یال هم دارای داده‌ساختارهایی برای نگه‌داری رأس‌های دو سر یال و نیز دستورهایی برای ایجاد یال، تعیین و گرفتن هر کدام از رأس‌های دو سر یال و مقایسه‌ی دو یال است. برای هر گراف نیز داده‌ساختارهای بزرگ‌تری برای نگه‌داری مجموعه‌ی رأس‌ها و یال‌های گراف نیاز است که باید عملیات مربوط به درج و حذف رأس و یال، گرفتن مجموعه‌ی تمام رأس‌ها و یال‌ها و نیز رأس‌های مجاور یک رأس خاص قابل انجام باشند. این عملیات مستقل از روش نگه‌داری مجموعه‌ی رأس‌ها و یال‌ها در گراف وجود دارند و برحسب روش نگه‌داری داده‌ساختارها، پیاده‌سازی می‌شوند.

در پیوست ۱، یک روش پیاده‌سازی از این شیءها و نیز پیاده‌سازی الگوریتم مورد نظر ارائه شده است.

۱–۳ زبان برنامه‌نویسی استفاده‌شده در این کتاب

این کتاب مبتنی بر زبان برنامه‌نویسی خاصی نیست. برای بیان برنامه و رَویّه‌ها از همان شبه‌کدی (زبان سطح بالایی) که در کتاب معروف [۵] به‌کار رفته استفاده می‌شود. این کتاب در اکثر دانشگاه‌های دنیا به‌عنوان کتاب اول درس «داده‌ساختارها و الگوریتم‌ها» استفاده می‌شود و از اعتبار زیادی برخوردار است. به این زبان CLRS می‌گوییم که مخفف حرف اول نویسندگان این کتاب است.

تمرین‌های فصل ۱

سعی کنید مسئله‌های زیر را مدل کنید. ارائه‌ی حل کامل ضروری نیست و شاید از عهده‌ی شما خارج باشد. ایده‌هایی را مطرح کنید که مسئله به چه مدلی تبدیل می‌شود.

۱.۱ می‌خواهیم برنامه‌ی زمان‌بندی یک دوره بازی‌ها با شرکت n تیم با شماره‌های ۱ تا n را به‌دست آوریم. برنامه‌ی بازی‌ها به‌صورت جدولی است که سطر i ام آن برنامه‌ی کامل هفته‌ی i ام را نشان می‌دهد که در آن مشخص می‌شود هر تیم با چه تیمی بازی خواهد کرد. می‌خواهیم جدول کامل بازی‌ها را طوری به‌دست آوریم تا هر تیم با بقیه‌ی تیم‌ها بازی کند، در هر هفته یک تیم حداکثر یک بازی انجام دهد و تعداد هفته‌های بازی‌ها کمینه شود.

الف) این مسئله را برای هر n دل‌خواه حل کنید. فرض کنید که در ابتدا هیچ بازی انجام نشده است.

ب) فرض کنید تعدادی از بازی‌ها از قبل انجام شده و نباید در برنامه‌ی جدید تکرار شوند. همین مسئله را با این فرض حل کنید.

★★ **۲.۱** گروهی از دانش‌جویان «زرنگ» رشته‌ی کامپیوتر می‌خواهند ضمن یادگیری کلیه‌ی مطالب درس، شرکت خود در کلاس‌های دانشکده را به حداقل برسانند با این شرط که در هریک از کلاس‌هایی که با هم دارند فقط یک نفر از آن‌ها شرکت کند. فرض بر این است که دیگر اعضای این گروه با استفاده از یادداشت‌های فرد شرکت کننده و بحث با او می‌توانند درس را به‌خوبی فرا بگیرند و نیازی به شرکت در کلاس ندارند.

فرض کنید همه‌ی این دانش‌جویان در تعدادی درس که در هفته n کلاس c_1 تا c_n دارد ثبت‌نام کرده‌اند. کلاس c_i از زمان a_i آغاز و در زمان b_i تمام می‌شود (زمان از ساعت صفر روز شنبه تا ۲۴ روز پنج‌شنبه به‌صورت یک عدد صحیح سراسری بیان می‌شود). مدت زمان لازم برای رفتن از کلاس c_i به کلاس c_j برابر r_{ij} است. بدیهی است که یک نفر نمی‌تواند در دو کلاس که زمان برگزاری آن‌ها مشترک است شرکت کند و نیز یک فرد مأمور می‌شود که به کلاس خاصی برود مشروط بر آن‌که بتواند از ابتدا تا پایان کلاس در آن شرکت کند.

ورودی این مسئله n، زمان‌های a_i و b_i برای هر کلاس درس c_i، زمان r_{ij} برای هر زوج کلاس c_i و c_j است. همه‌ی اعداد صحیح هستند. خروجی، کم‌ترین تعداد دانش‌جویان مورد نیاز در این گروه است و این‌که هر یک از اعضای گروه در کدام کلاس حتماً باید شرکت کند.[۳۹]

[۳۹] سؤال المپیاد دانش‌جویی کامپیوتر در تابستان ۸۲.

پروژه‌های برنامه‌نویسی فصل ۱

رنگ‌آمیزی گراف

همان‌طور که بیان شد، حل رده‌ی وسیعی از مسئله‌ها منجر به حل مسئله‌ی رنگ‌آمیزی گراف می‌شود که هدف آن انتساب یک رنگ از مجموعه‌ی رنگ‌ها به هر رأس گراف است، به‌طوری‌که هیچ دو رأس مجاوری هم‌رنگ نباشند. این یک مسئله‌ی ان‌پی‌ـسخت است، بنابراین یافتن جواب بهینه، که شامل تعداد کمینه‌ی رنگ‌ها است، برای گراف‌های بزرگ امکان‌پذیر نیست، و در نتیجه دست‌یابی به یک رنگ‌آمیزی نزدیک‌به‌بهینه ولی سریع، ارزش‌مند است.

در این فصل، یک روش تقریبی حریصانه برای حل این مسئله بیان شد که در این‌جا آن‌را approx1 می‌نامیم. یک راه‌حل تقریبی دیگر، انتخاب یک رأس با بیش‌ترین تعداد همسایه و رنگ‌آمیزی آن با یک رنگ مجاز و ادامه‌ی این کار به‌صورت بازگشتی بر روی بقیه‌ی گراف است. ما این روش تقریبی دوم را approx2 نام نهاده‌ایم.

به‌عنوان تمرین برنامه‌نویسی، از شما خواسته می‌شود که الگوریتم بهینه (یا exact) و دو الگوریتم تقریبی فوق را برای مسئله‌ی رنگ‌آمیزی گراف پیاده‌سازی کنید و نتایج آن‌ها را از نظر زمان اجرا و تعداد رنگ‌های مصرفی با هم مقایسه کنید. مقایسه‌ی زمان‌های اجرا را به‌صورت یک منحنی برحسب اندازه‌های مختلف ورودی نشان دهید.

ورودی: از ورودی استاندارد بخوانید. در سطر اول ورودی، به‌ترتیب n (تعداد رأس‌های گراف) و e (تعداد یال‌های گراف) با یک فاصله از هم آمده‌اند. در هر یک از e سطر بعد، دو عدد با فاصله از هم هستند که دو سر یک یال از گراف می‌باشند. رأس‌ها از ۱ تا n شماره‌گذاری شده‌اند. می‌دانیم که گراف داده شده هم‌بند است.

خروجی: در خروجی استاندارد بنویسید. در سطر اول، تعداد رنگ‌های لازم و در سطر دوم، n عدد با یک فاصله از هم بنویسید که عدد i ام رنگ رأس i ام است. رنگ‌ها را از ۱ شماره‌گذاری کنید.

راهنمایی: راه‌حل‌های تقریبی را می‌توان با زمان اجرایی متناسب با n^2 نوشت. اما در مورد پیاده‌سازی الگوریتم بهینه، رنگ‌آمیزی را به صورت بازگشتی انجام دهید. در هر سطح از اجرای تابع، یک رأس را انتخاب و با رنگ‌های مختلف رنگ کنید و تابع سطح بعدی را فراخوانی کنید. سعی کنید هرجا که می‌توانید از ادامه‌ی رنگ‌آمیزی صرف‌نظر کنید تا زمان اجرای برنامه کوتاه‌تر شود. مثلاً، ممکن است رنگ‌آمیزی فعلی نامعتبر شده باشد، یا این‌که تعداد رنگ‌هایی که تاکنون برای رنگ‌آمیزی استفاده شده بیش‌تر از تعداد رنگ‌های بهترین جوابی باشد که تا کنون پیدا شده است. یک ایده‌ی مهم دیگر برای بهبود زمان اجرا، این است که سعی کنید در هر مرحله رأسی را انتخاب کنید که با شماره‌ی رنگ کم‌تری بتوان آن را رنگ کرد (طوری که رنگ‌آمیزی معتبر بماند). این کار فضای جستجو را به طرز

چشم‌گیری کاهش می‌دهد. این نکته را هم به‌خاطر داشته باشید که هر گراف G، دارای رنگ‌آمیزی معتبری با $\Delta(G)+1$ رنگ است.[۴۰]

نمونه‌ی ورودی و خروجی

input	outputs		
	approx1	approx2	exact
8 12 1 2 1 3 1 4 2 3 2 4 3 4 4 5 5 6 6 7 6 8 4 8 5 7	4 4 3 2 1 2 1 3 2	4 1 2 3 4 1 2 3 1	4 4 3 2 1 2 1 3 2
8 7 1 5 2 5 3 5 3 6 4 6 4 7 4 8	3 2 2 3 1 1 2 2 2	2 1 1 1 1 2 2 2 2	2 2 2 2 2 1 1 1 1
4 3 1 3 3 4 4 2	2 1 2 2 1	3 1 1 2 3	2 1 2 2 1

[۴۰] $\Delta(G)$ بیشترین درجه در گراف G است.