دانشکده‌ی علوم ریاضی

آنالیز الگوریتم‌ها — ۲۷ فروردین ۹۲

جلسه‌ی ۱۳: برنامه ریزی پویا

مدرّس: *دکتر شهرام خزائی* — نگارنده: *امیر کیال و حمید پورربیع رودسری*

۱ مقدمه

برنامه ریزی پویا[۱] روشی بهینه برای تبدیل مسایل پیچیده به مجموعه‌ای از مسایل ساده‌تر است. برنامه‌ریزی پویا یک قالب کلی برای حل دسته‌ی گسترده‌ای از مسایل است و بر خلاف برنامه‌ریزی خطی، چارچوبی استاندارد برای طبقه‌بندی مسائل مربوط به آن وجود ندارد و معمولا برای حل این دسته از مسایل به خلاقیت نیاز است. در واقع، آن‌چه برنامه‌ریزی پویا انجام می‌دهد ارائه روش برخورد کلی جهت حل دسته‌ای از مسایل است. در اینجا سعی می‌کنیم با حل چند مسأله از این طریق روش دیدی کلی نسبت به روش حل مسایل با برنامه‌ریزی پویا ایجاد کنیم.

۲ سنگین‌ترین مجموعه مستقل در گراف مسیری

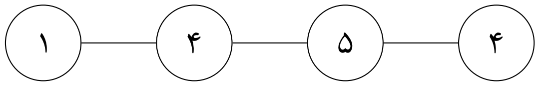
تعریف ۱ *(مجموعه مستقل[۲]) زیرمجموعه $S \subset V$ یک مجموعه مستقل برای گراف $G = (V, E)$ نامیده می‌شود هرگاه G هیچ یالی که دو سر آن در S باشد نداشته باشد.*

تعریف ۲ *(مسأله سنگین‌ترین مجموعه مستقل[۳]) گراف $G = (V, E)$ که رأس‌های آن وزن‌دار هستند داده شده است و هدف پیدا کردن مجموعه‌ای از رئوس مستقل این گراف است به گونه‌ای که مجموع وزن آن‌ها بیشینه باشد.*

تعریف ۳ *(گراف مسیری[۴]) گراف $G = (V, E)$ که $V = \{v_1, \ldots, v_n\}$ و $E = \{v_iv_{i+1} \mid i = 1, \ldots, n-1\}$ یک گراف مسیری نامیده می‌شود.*

ما به‌دنبال الگوریتمی برای حل مسأله سنگین‌ترین مجموعه مستقل در گراف مسیری G با وزن w_i برای رأس v_i هستیم.

مثال ۱ *سنگین‌ترین مجموعه مستقل در گراف زیر کدام مجموعه است؟ اعداد درون گره‌ها بیانگر وزن آنها هستند.*



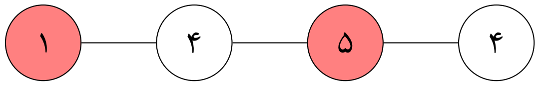
[۱] Dynamic Programming
[۲] Independent Set
[۳] Maximum Weight Independent Set
[۴] Path Graph

۱.۲ جواب بدیهی

اگر بخواهیم با جستجوی کامل[۵] به جواب برسیم باید ۲^n عملیات انجام بدهیم که برای مسائل بزرگ، به هیچ عنوان قابل استفاده نیست.

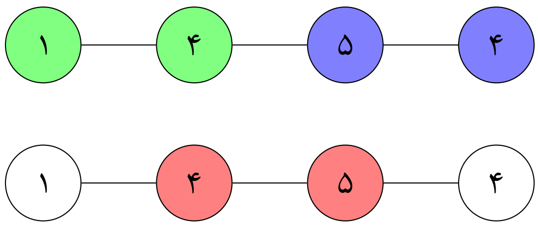
۲.۲ حل به روش الگوریتم حریصانه

در الگوریتم حریصانه ابتدا رأس با بیشترین وزن انتخاب می‌شود که در اینجا یعنی رأس با وزن ۵. پس از انتخاب این رأس الگوریتم اجازه ندارد رئوس مجاور را انتخاب کند زیرا در غیر این صورت مجموعه مستقل نخواهد بود. پس در این‌جا رأس‌های مجاور ۵ با وزن ۴ انتخاب نخواهند شد. در این صورت الگوریتم حریصانه مجبور به انتخاب رأس با وزن ۱ می‌شود. یعنی در نهایت جواب نهایی الگوریتم حریصانه دو رأس با وزن‌های ۱ و ۵ خواهد بود که مجموع وزن آن‌ها ۶ خواهد شد در حالی که اگر رأس‌های با وزن ۴ را انتخاب می‌کردیم مجموع وزن‌ها ۸ می‌شد. در نتیجه می‌توان به این نکته پی برد که الگوریتم حریصانه روشی مناسب برای حل این سوال نیست.



۳.۲ حل به روش الگوریتم تقسیم وحل

برای استفاده از روش تقسیم و حل ابتدا گراف را باید به دو زیرگراف افراز کرده و سعی کنیم که مسأله را برای هر زیرگراف حل کنیم. در مثال داده شده گراف را به صورت زیر به دو زیرگراف تقسیم می‌کنیم. در زیرگراف آبی رأس با وزن ۵ سنگین‌ترین مجموعه‌ی مستقل و در گراف سبز رأس با وزن ۴ سنگین‌ترین مجموعه‌ی مستقل است. پس در نهایت دو رأس با وزن‌های ۴ و ۵ انتخاب خواهند شد که مجاور نیستند. یعنی روش تقسیم و حل نیز در این سوال کار نمی‌کند.



۴.۲ حل با یک راهکار جدید

کلید اصلی در حل این مسأله استفاده از این نکته است که برای یک رأس در گراف یا خودش در سنگین‌ترین مجموعه‌ی مستقل حضور دارد و یا همسایه‌هایش. یعنی به طور همزمان یک رأس و همسایه‌هایش در مجموعه‌ی مستقل حضور ندارند. اثبات این مسأله هم از طریق تعریف سنگین‌ترین مجموعه‌ی مستقل بدیهی است زیرا اگر به طور همزمان یک

[۵] Brute-force-search

رأس و همسایه‌هایش در مجموعه باشند دیگر آن مجموعه مستقل نخواهد بود.
در این‌جا گراف مطرح شده یک گراف مسیری است. با در نظر گرفتن نکته‌ی بالا برای رأس انتهایی مسیر مسأله به دو زیر مسأله کوچکتر تقسیم می‌شود:

- حالتی که رأس انتهایی در مجموعه وجود دارد که در این صورت رأس همسایه نمی‌تواند در مجموعه باشد. در نتیجه مسأله را باید برای گرافی که شامل $n-۲$ رأس دیگر است حل کنیم و در نهایت رأس انتهایی را به جواب اضافه کنیم.

- حالتی که رأس انتهایی در جواب نیست. در این صورت باید مسأله را برای گرافی با $n-۱$ رأس دیگر حل کنیم.

در انتها برای پیدا کردن جواب نهایی می‌توانیم ماکزیمم را از بین دو جواب بالا پیدا کنیم و به عنوان جواب در نظر بگیریم. پس الگوریتم بالا به صورت زیر خواهد بود:

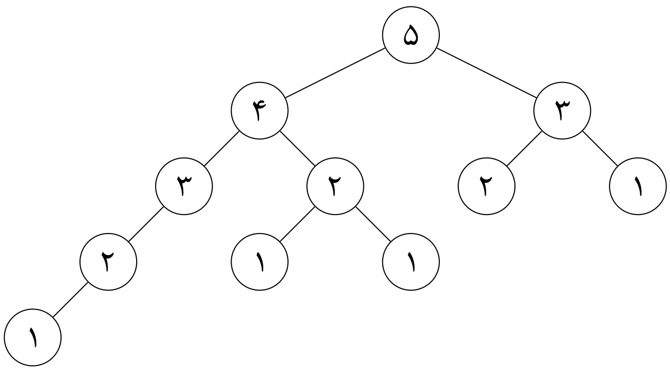
Algorithm 1 Algorithm: MAXIMUM WEIGHT INDEPENDENT SET

```
function MWIS(graph G, weights w_1, ..., w_n)
    [assumes G is weighted path graph]
    G' ← G without last vertex
    G'' ← G without the last two vertices
    S' ← MWIS(G', w_1, ..., w_{n-1})
    S'' ← MWIS(G'', w_1, ..., w_{n-2})
    return S' or S'' ∪ {last vertex of G}, whichever is heavier
```

پیچیدگی الگوریتم بالا از رابطه‌ی بازگشتی زیر محاسبه می‌شود:

$$T(n) = T(n-۱) + T(n-۲) + O(۱)$$

در نتیجه پیچیدگی $O(۲^n)$ خواهد بود.
مراحل اجرای این الگوریتم را بر روی گرافی با سایز ۵ بررسی می‌کنیم:



این الگوریتم برای پردازش گراف با سایز ۵، یک بار بر روی زیرگراف با سایز ۴، دو بار بر روی زیرگراف با سایز ۳، سه بار بر روی زیرگراف با سایز ۲ و چهار بار بر روی زیرگراف با سایز ۱ اجرا شده است. در حالی که تنها لازم بود یک بار بر روی هر کدام از این زیرگراف‌ها اجرا شود. برای جلوگیری از این موضوع به این صورت عمل می‌کنیم که از رأس

ابتدایی شروع کرده و در هر مرحله رأس همسایه را به گراف اضافه می‌کنیم و سوال را برای زیرگراف به دست آمده حل کرده و در آرایه‌ای ذخیره می‌کنیم. حال در صورتی که بخواهیم سوال را برای گراف با سایز n حل کنیم فقط کافی است که مسأله را برای $n-۱$ زیر گراف آن حل کنیم و در آرایه‌ای ذخیره کنیم. در این صورت مسأله به سادگی برای گراف حل خواهد شد. الگوریتم فوق به صورت زیر خواهد بود.

Algorithm 2 Algorithm: Maximum Weight Independent Set

```
function MWIS(graph G, weights w_1, ..., w_n)
    [assumes G is weighted path graph]
    A[0] ← 0
    A[1] ← v_1
    for i = 2 to n do
        A[i] ← max{A[i−1], A[i−2] + w_i}
    S ← ∅
    i ← n
    while i ≥ 1 do
        if A[i−1] ≥ A[i−2] + w_i then
            i ← i − 1
        else
            Add v_i to S
            i ← i − 2
    return S, A[n]
```

واضح است که پیچیدگی الگوریتم فوق $O(n^۲)$ است.
برای اثبات صحت الگوریتم می‌توان از استقرا روی سایز گراف استفاده کرد.

۳ راهکارهای کلی برنامه ریزی پویا

با حل مسأله‌ی بالا تا حدی با روش استفاده از برنامه‌ریزی پویا آشنا شدید. به صورت خلاصه می‌توان راهکارهای برنامه‌ریزی پویا را به صورت زیر طبقه بندی کرد:

۱. تعیین زیرمسأله‌های مرتبط

۲. حل سوال برای زیرمسأله‌های بزرگتر با استفاده از زیرمسأله‌های کوچکتر

۳. به دست آوردن جواب نهایی مسأله با استفاده از جواب زیرمسأله‌ها (اکثراً، مسأله اصلی یک از زیرمسأله‌ها است).

۴ مسأله کوله پشتی

مسأله کوله‌پشتی[۶] یک مسأله معروف در علوم کامپیوتر است که می‌توان آنرا بدین صورت مطرح کرد. طی یک سرقت، سارقی بیشتر از آنچه که انتظار داشته اموال مسروقه پیدا می‌کند و باید بین آن‌ها انتخاب کند. کوله پشتی وی حداکثر وزن W را تحمل می‌کند و n جسم مختلف با وزن‌های $w_۱$ تا w_n (که $w_i \leq W$) و ارزش‌هایی $v_۱$ تا v_n موجود است. دزد کدام یک از این اشیا را انتخاب کند تا بیشترین ارزش را داشته باشد؟
این مسأله به دو صورت مختلف قابل بیان است.

- کوله پشتی با تکرار: در این حالت می‌توان از هر جسم بیش از یک‌بار استفاده کرد.

[۶] Knapsack Problem

- کوله پشتی بدون تکرار: در این صورت از هر جسم فقط یکی موجود است و در نتیجه نمی‌توان بیش از یکبار از یک جسم استفاده کرد.

با کمک برنامه ریزی پویا می‌توان هر دو نوع این مسأله را در $O(nW)$ حل کرد.

۱.۴ کوله پشتی با تکرار

اولین سوالی که باید برای حل با کمک برنامه‌ریزی پویا از خود بپرسیم این است که زیرمسأله‌ها کدامند؟
این مسأله را به دو صورت می‌توان تقسیم کرد. کوله‌پشتی با حجم کمتر و تعداد اشیا ثابت و یا تعداد اشیا کمتر و کوله پشتی با حجم ثابت. در اینجا ما از روش کوله پشتی با حجم کمتر استفاده خواهیم کرد:
$K(w)$ را به صورت زیر تعریف می‌کنیم:

$$K(w) = \text{maximum value achievable with a knapsack of capacity } w\ .$$

حال برای به دست آوردن $K(w)$ از روی مقادیر کوچکتر w می‌توان به صورت زیر عمل کرد:

$$K(w) = \max_{i:w_i \leq w} \{K(w - w_i) + v_i\}$$

البته اینجا باید در نظر داشت که ماکزیمم بر روی مجموعه‌ی تهی برابر ۰ است.

Algorithm 3 Algorithm: KNAPSACK WITH REPETITION

---

function KNAPSACKWITHREPETITION
  for $w = 1$ **to** W **do**
    $K(w) \leftarrow \max_{i:w_i \leq w}\{K(w - w_i) + v_i\}$
  return $K(W)$

پیچیدگی الگوریتم فوق به وضوح $O(nW)$ است.

۲.۴ کوله پشتی بدون تکرار

برای حل این سوال نمی‌توانیم از الگوریتم ارائه شده در بالا استفاده کنیم. فرض کنید برای محاسبه‌ی $K(w)$ بدانیم مقدار $K(w - w_n)$ بسیار زیاد است. این موضوع کمک چندانی به ما نخواهد کرد زیرا نمی‌دانیم که برای محاسبه‌ی $K(w - w_n)$ از v_n استفاده شده است یا خیر. پس نمی‌توانیم مقدار $K(w)$ را از روش بالا به دست آوریم. پس باید زیر مسأله‌ها را به گونه‌ای تغییر دهیم که این اطلاعات را نیز در خود ذخیره کند. یعنی به جای $K(w)$ ، از $K(w,j)$ که به صورت زیر تعریف شده است، استفاده می‌کنیم.

$$K(w, j) = \text{maximum value achievable using a knapsack of capacity } w \text{ and items } 1, \ldots, j\ .$$

برای به دست آوردن $K(w,j)$ از روی مقادیر قبلی به این صورت عمل می‌کنیم که جسم jام یا توسط دزد برداشته می‌شود و یا خیر. در نتیجه $K(w,j)$ به صورت زیر محاسبه می‌شود:

$$K(w, j) = \max\{K(w - w_j, j - ۱) + v_i, K(w, j - ۱)\}$$

در این صورت حل مسأله تبدیل می‌شود به پر کردن یک جدول با $W + ۱$ سطر و $n + ۱$ ستون. پس مسأله از $O(nW)$ خواهد بود.

Algorithm 4 Algorithm: Knapsack without repetition

function KnapsackWithoutRepetition
  Set $K(0,j) \leftarrow 0$ for $j = 0, \dots, n$
  Set $K(w,0) \leftarrow 0$ for $w = 0, \dots, W$
  for $j = 1$ **to** n **do**
    for $w = 1$ **to** W: **do**
      $K(w,j) \leftarrow \max\{K(w,j-1), K(w-w_j, j-1) + v_j\}$
  return $K(W,n)$

۵ مسأله فاصله ویرایشی

وقتی یک نرم‌افزار غلط‌یاب املایی[۷] با کلمه‌ای که ممکن است اشتباه نوشته شده باشد مواجه می‌گردد در لغت‌نامه به دنبال گزینه‌های نزدیک به لغت وارد شده می‌گردد. سوالی که به وجود می‌آید این است که روش مناسب برای تعریف نزدیکی دو لغت چیست؟ چگونه باید فاصله‌ی بین دو کلمه را حساب کرد؟
یک روش مناسب برای تعریف فاصله استفاده از هم‌ترازسازی[۸] لغات است. هم‌تراز سازی یعنی نوشتن کلمات بالا و پایین یکدیگر. برای مثال دو هم‌تراز سازی برای دو کلمه‌ی $SUNNY$ و $SNOWY$ در ادامه آورده شده است.

$$\begin{array}{cccccc} S & U & N & N & - & Y \\ S & - & N & O & W & Y \end{array}$$

$$\begin{array}{ccccccc} S & U & N & - & - & N & Y \\ - & S & N & O & W & - & Y \end{array}$$

هزینه را تعداد ستون‌هایی می‌نامیم که با هم برابر نیستند که در مثال سمت اول برابر ۳ و در مثال دوم برابر ۵ است. حال کمترین هزینه‌ی ممکن را فاصله‌ی ویرایشی[۹] می‌نامیم. این مقدار از این رو با نام فاصله‌ی ویرایشی شناخته می‌شود که کمترین تعداد ویرایش‌های_حذف، جایگذاری و اضافه‌کردن_لازم بر روی یک کلمه است تا به کلمه‌ی دیگر تبدیل شود.
برای حل این مسأله از روش برنامه‌ریزی پویا، باید به این سوال جواب بدهیم که زیرمسأله‌ها کدامند؟
زیرمسأله‌ها را پیشوندهای رشته‌های ورودی که با x و y نشان می‌دهیم درنظر می‌گیریم. یعنی $E(i,j)$ را برابر فاصله‌ی ویرایشی رشته‌های $x[۰:i]$ و $y[۰:j]$ در نظر خواهیم گرفت که $x[۰:i]$ پیشوند به طول i از رشته x است. در صورتی که روش استفاده شده درست باشد باید بتوان روشی برای محاسبه‌ی $E(i,j)$ از روی زیرمسأله‌های دیگر به دست آورد. برای اینکار به آخرین ستون در هم‌ترازسازی نگاه می‌کنیم. این ستون می‌تواند یکی از سه حالت زیر باشد:

$$\begin{array}{c} x[i] \\ - \end{array} \quad \text{یا} \quad \begin{array}{c} - \\ y[j] \end{array} \quad \text{یا} \quad \begin{array}{c} x[i] \\ y[j] \end{array}$$

پس برای محاسبه‌ی $E(i,j)$ می‌توانیم به صورت زیر عمل کنیم:

$$E(i,j) = \min\{۱ + E(i-۱, j), ۱ + E(i, j-۱), \mathrm{diff}(i,j) + E(i-۱, j-۱)\}$$

[۷] Spell checher
[۸] Alignment
[۹] Edit distance

که در آن $\mathrm{diff}(i,j)$ در صورتی که $x[i]$ برابر $y[j]$ باشد، صفر و در غیر این صورت برابر یک است.
از آنجایی که توانستیم $E(i,j)$ را با استفاده از زیرمسأله‌های دیگر به دست آوریم، می‌توان نتیجه گرفت که زیرمسأله‌ها به درستی انتخاب شده‌اند. قدم بعدی برای حل مسأله پیدا کردن ترتیبی مناسب برای به دست آوردن $E(i,j)$ها است.
در هنگام به دست آوردن $E(i,j)$ها تنها کافی است که آن‌ها را به گونه‌ای حساب کنیم که در هنگام محاسبه‌ی $E(i,j)$ مقادیر $E(i-۱,j)$ و $E(i,j-۱)$ حساب شده باشند. برای اینکار می‌توان جدول را به صورت سطر به سطر از چپ به راست و یا ستون به ستون از بالا به پایین پر کرد. تنها نکته‌ای که تا نهایی شدن حل سوال باقی مانده است به دست آوردن مقادیر $E(۰,\cdot)$ و $E(\cdot,۰)$ است. زیرا از طریق رابطه‌ی بیان‌شده در بالا قابل محاسبه نیستند. از تعریف فاصله‌ی نگارشی بدیهی است که $E(i,۰)=i$ و $E(۰,j)=j$. پس سوال حل شد.
الگوریتم بیان‌شده در بالا را می‌توان به صورت زیر نوشت:

Algorithm 5 Algorithm: EDIT DISTANCE

function EDITDISTANCE($x_1 \dots x_m, y_1, \dots, y_n$)
    Set $E(i,0) \leftarrow i$ for $i = 0, \dots, m$
    Set $E(0,j) \leftarrow j$ for $j = 0, \dots, n$
    for $i = 1$ **to** m **do**
        for $j = 1$ **to** n **do**
            $E(i,j) = \min\{E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1)+\mathrm{diff}(i,j)\}$
    return $E(m,n)$