ALGORITHM DESIGN

Shahrokh Amani

JUNE 5, 2026
EMPEROR-DEV
Toronto, ON, Canada

فهرست مطالب

فصل 1 1

پیچیدگی اجرایی 1

فصل 2 9

رابطه بازگشتی 9

فصل 3 19

روش تقسیم و حل 19

(Divide & Conquer) 19

فصل 4 32

روش حریصانه 32

(Greedy) 32

فصل 5 47

روش عقبگرد 47

(Backtracking) 47

فصل 6 60

گراف 60

فصل 7 104

مسائل P و NP 104

فصل 1

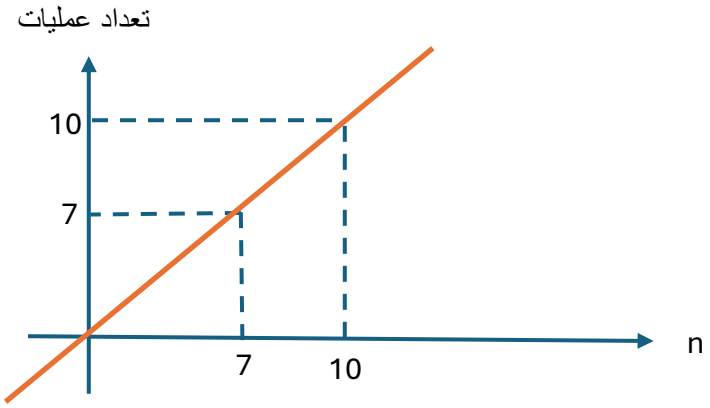
پیچیدگی اجرایی

پیچیدگی اجرائی: تابعی است که مدت زمان اجرای یک الگوریتم را بر حسب تعداد ورودی ها اندازه می گیرد.

مثال: در یک آرایه با 7 عنصر به صورت زیر داریم:

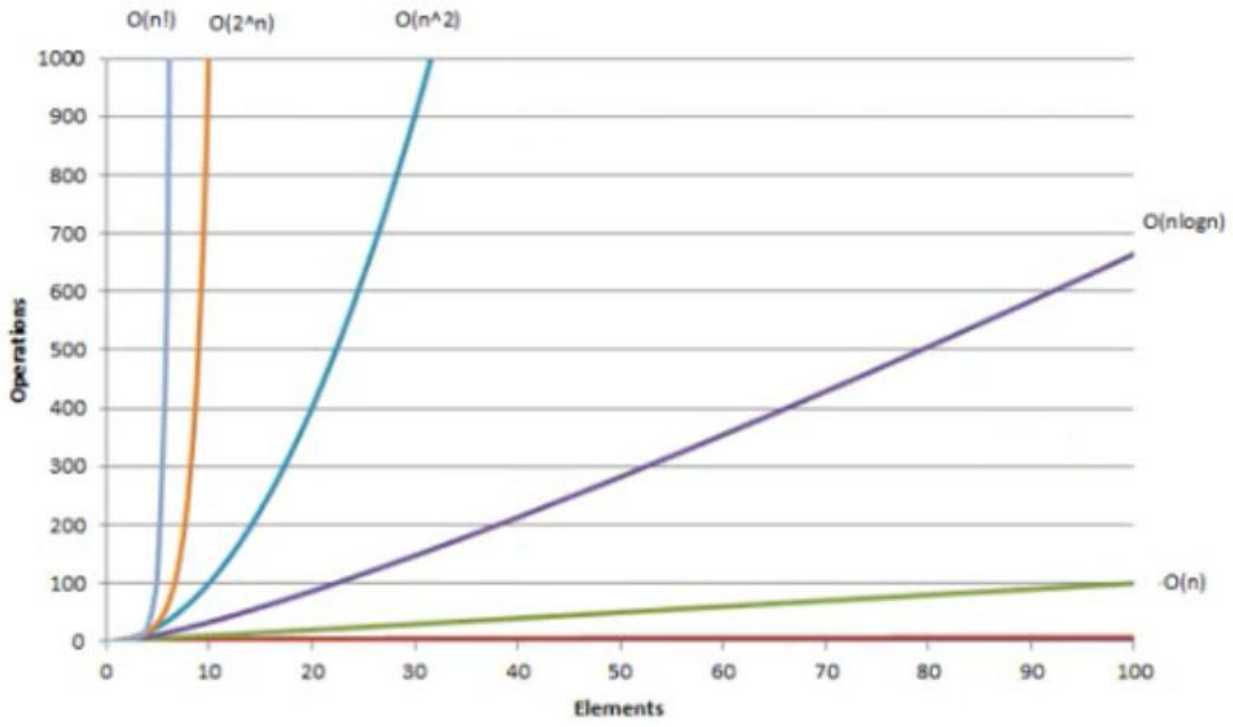
5	1	17	16	7	13	20
0	1	2	3	4	5	6

در این آرایه بنا داریم با استفاده از الگوریتم جستجوی خطی مقدار Key=13 را پیدا کنیم. برای این کار لازم است تا مقدار Key با همه درایه ها مقایسه شود تا محل مورد نظر را پیدا کنیم. در بد ترین حالت مقدار مورد نظر می تواند در خانه آخر(n) قرار داشته باشد که در این نوع الگوریتم می گوئیم پیچیدگی اجرایی از درجه n (O(n)) می باشد.



انواع پیچیدگی ها :

$$O(1) < O(\lg n) < O(n) < O(n \lg n) < O(n^2) < O(2^n) < O(n!) < O(n^n)$$



مرتبه اجرایی یک حلقه

for(int i = a; i ≤ b; i+= k)

$Statements;$

$O(n)= \frac{b-a+1}{k}$

مثال1:

for(int i = 3; i < = n; i+= 2)

$Statements;$

$= \frac{n-3+1}{2} == \frac{n-2}{2} = \frac{n}{2} - 1 => O(n)$

مثال 2: ضرب و تقسیم در گام حلقه دارای فرمول یکسان می باشد.

$\text{for(int i} = a; \text{i} <= \text{b; i} *= \text{k)}$ $\qquad\qquad$ $log_k^b - log_k^a + 1$

$Statements;$

$\text{for(int i} = 1; \text{i} <= \text{n; i} *= 2)$ $\qquad\qquad$ $log_2^n - log_2^1 + 1 = O(\log n)$

$Statements;$

مرتبه اجرایی در حلقه های تو در تو:

$\text{for(int i} = 1; \text{i} <= \text{n; i} + +)$

$Statements;$ $\qquad$ =➔ $\qquad$ O(max (m, n)) OR O(m, n)

$\text{for(int } j = 1; \text{j} <= \text{m; j} + +)$

$Statements;$

مثال:

1-

$\text{for(int i} = 1; \text{i} <= \text{n; i} + +)$

$\text{for(int } j = 1; \text{j} <= \text{n; j} + +)$ ➔ $O(n^2)$

$Statements;$

2-

$\text{for(int i} = 2; \text{i} <= \text{n}; \text{i+=} 4)$

$\text{for(int } j = n; \text{j} > 3; \text{j} = \text{j} - 2)$ ➔ $\frac{n-2+1}{4} * \frac{n-4+1}{2} = O(n^2)$

$Statements;$

3-

$\text{for(int i} = 1; \text{i} <= \text{n}; \text{i} *= 2)$

$\text{for(int } j = 1; \text{j} < \text{n}; \text{j} + +)$ ➔ $(log_2^n - log_2^1 + 1)$*n=O(nlog n)

$Statements;$

مرتبه اجرائی حلقه تو در تو وابسته

$\text{for(int i} = 1; \text{i} <= \text{n}; \text{i} + +)$

$\text{for(int } j = 1; \text{j} <= \text{i}; \text{j} + +)$

$Statements;$

در این مثال ابتدا می توانیم با فرمول ریاضی تعداد دفعات اجرای Statement را به صورت زیر محاسبه کرد:

1+2+3+...+n=>$\frac{n(n+1)}{2}$

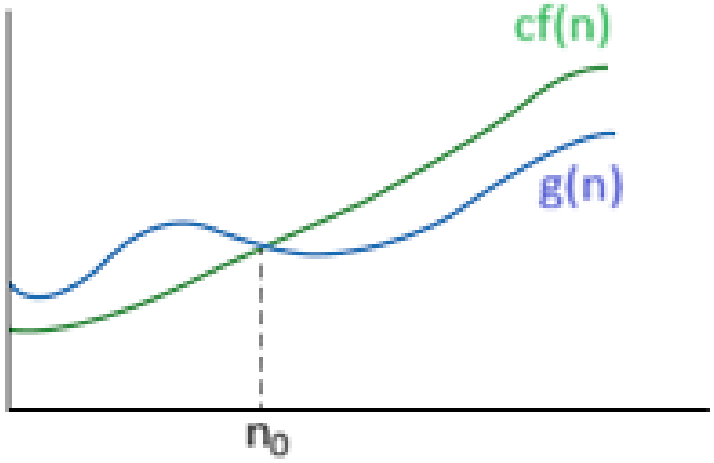
از این فرمول همچنین می توان نتیجه گرفت که مرتبه اجرایی آن O(n^2) می باشد.

نمادهای پیچیدگی اجرائی

Big O -1

عبارت $g(n) \in O(f(n))$

یعنی: برای تابع پیچیدگی مفروض f(n) ، O(f(n)) به مجموعه ای از توابع اشاره دارد که برای آنها ثابتهای c و n_0 وجود دارند، بطوریکه برای همه $n \geq n_0$ داریم: $g(n) \leq cf(n)$

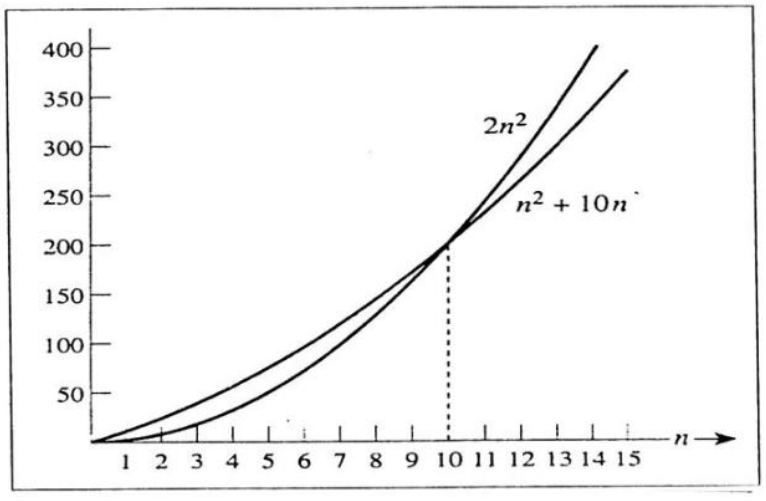


مثال:

$$n^2 + 10n \in O(n^2)$$

$$n^2 + 10n \leq 2n^2$$

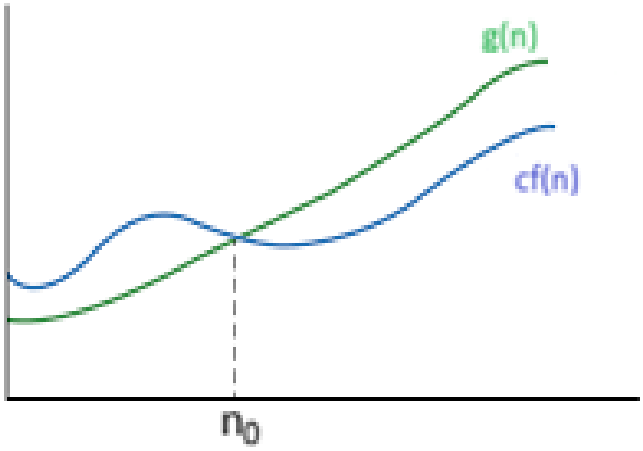
$$n^2 + 10n = 2n^2$$
$$\Rightarrow 10n = n^2$$
$$\Rightarrow n = 10$$



Big Ω -2

عبارت $g(n) \in \Omega(f(n))$

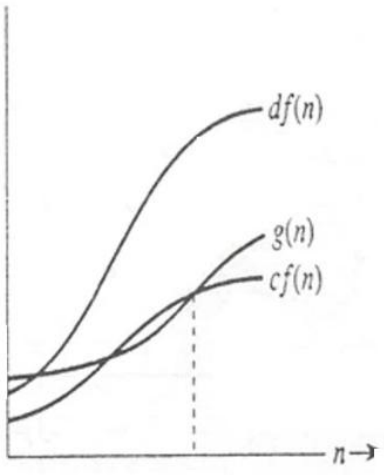
یعنی: برای تابع پیچیدگی مفروض f(n) ، $\Omega(f(n))$ به مجموعه ای از توابع اشاره دارد که برای آنها ثابتهای c و n_0 وجود دارند، بطوریکه برای همه $n \geq n_0$ داریم: $g(n) \geq cf(n)$



3- تتا Θ:

عبارت $g(n) \in \theta(f(n))$

یعنی : $g(n) \in O(f(n))$ و $g(n) \in \Omega(f(n))$



مثال کلی:

$$O(n^2) = \{\lg n, n, 7n^2 + 4, n\lg n, n^2, 5n^2 - 6, \ldots\}$$

$$\Omega(n^2) = \{n^2, 5n^2 - 6, n^2 \lg n, n^3, n^6, 7n^2 + 4, \ldots\}$$

$$\theta(n^2) = \{n^2, 5n^2 - 6, 7n^2 + 4, \ldots\}$$

فصل 2

رابطه بازگشتی

توابع بازگشتی، توابعی هستند که در آن تابع، خود تابع فراخوانی می شود.

```
int Fact(int n)
{
    if (n == 0)
        return 1;
    else
        return n*Fact(n-1);
}
```

رابطه بازگشتی مربوط به این تابع در اصل تابعی است که تعداد ضربها را محاسبه می کند و به صورت زیر عنوان می شود:

$$\begin{cases} 0 & n = 0 \\ 1 + T(n-1) & n \geq 1 \end{cases}$$

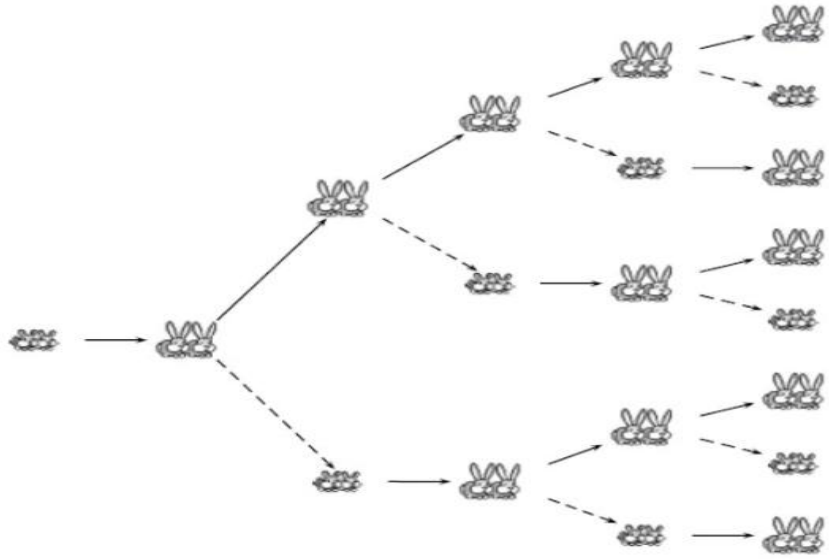
اگر n=0 باشد هیچ عمل ضربی انجام نمی شود. در صورتی که بخواهیم فاکتوریل n را محاسبه کنیم، 1+(n-1)! نیاز می باشد.

مسئله خرگوش ها

در یک جزیره یک جفت خرگوش نر و ماده داریم. مدل رشد جمعیت خرگوش ها به صورت زیر است:

1- خرگوشها یک ماه پس از تولد به سن بلوغ می رسند.
2- یک ماه پس از رسیدن بلوغ و از آن به بعد همه ماهه هر جفت خرگوش یک جفت خرگوش، یک جفت خرگوش دیگر تواید می کنند.
3- خرگوش ها هرگز نمی میرند.

رابطه بازگشتی برای تعداد خرگوش ها در هر ماه به شکل زیر است:

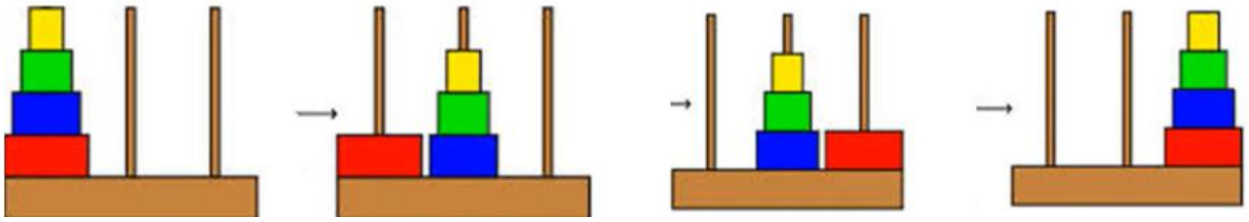


$$\begin{cases} f(1) = 1, f(2) = 1 \\ f(n) = f(n-1) + f(n-2) \qquad n \geq 3 \end{cases}$$

برج هانوی

در تعریف برج Hanoi می‌توان گفت: برج هانوی یک پازل ریاضی است که در آن سه میله و nحلقه‌داریم .این حلقه‌ها اندازه‌های مختلفی دارند و به صورت صعودی روی‌هم قرار می‌گیرند، یعنی حلقه کوچک‌تر روی حلقه بزرگ‌تر قرار می‌گیرد. انواع دیگری از پازل وجود دارد که در آن تعداد حلقه‌ها افزایش می‌یابد، اما تعداد برج‌ها ثابت می‌ماند. هدف از این بازی این است که کل حلقه‌ها را به میله دیگری با رعایت قوانین ساده زیر منتقل کنید:

1. فقط یک دیسک را می‌توان در یک زمان جابجا کرد.
2. هر حرکت شامل برداشتن دیسک بالایی از یکی از پشته‌ها و قرار دادن آن در بالای پشته دیگر است، یعنی یک دیسک تنها در صورتی می‌تواند جابجا شود که بالاترین دیسک روی یک پشته باشد.
3. هیچ دیسکی را نمی‌توان روی دیسک کوچک‌تر قرارداد



$$a_n = 2(a_{n-1}) + 1$$

روش های حل رابطه های بازگشتی

1- تکرار با جایگذاری:

در این روش با جایگذاری مرحله به مرحله متغیر ها می توانیم به رابطه ای برسیم که در نهایت با تخصیص مقدار مشخصی از ورودی تعداد دفعات اجرا را مقایسه کنیم و رابطه بازگشتی را به یک رابطه قابل حل تبدیل نمائیم.

$$T(n) = n + T(n-1)$$

$$T(1) = 1$$

$$\begin{aligned}
T(n) &= n + T(n-1) \\
&= n + (n-1) + T(n-2) \\
&= n + (n-1) + (n-2) + T(n-3) \\
&\dots \\
&= n + (n-1) + (n-2) + \dots + 2 + T(1) \\
&= n + (n-1) + (n-2) + \dots + 2 + 1 \\
&= \sum_{i=1}^{n} i = \frac{n(n+1)}{2}
\end{aligned}$$

2- درخت بازگشت

در روش درخت بازگشت (Recursion Tree) می توانیم رابطه بازگشتی را یا حل کنیم و یا حدس بزنیم. در هر سطح نحوه قرار گرفتن یک عبارت بازگشتی و مقدار ثابت نمایش داده می شود. همچنین با جمع کردن مقادیر ثابت جواب بدست می آید.

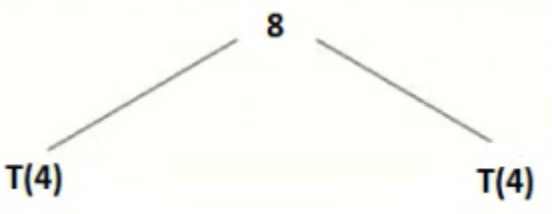
مثال: در رابطه بازگشتی زیر درخت بازگشت T(8) را رسم و هزینه را محاسبه کنید.

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$T(1) = 1$$

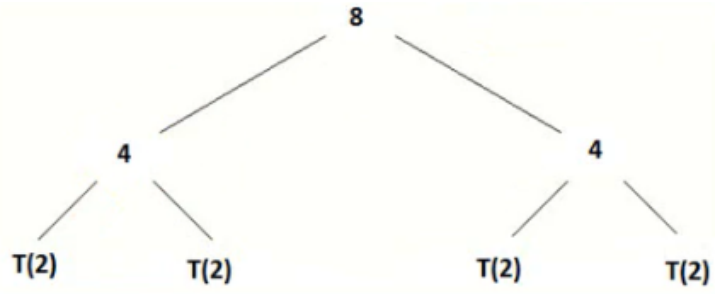
برای شروع مقدار 8 را در رابطه قرار می دهیم و سپس درخت آن ترسیم می شود.

$$T(8) = 2T(4) + 8$$



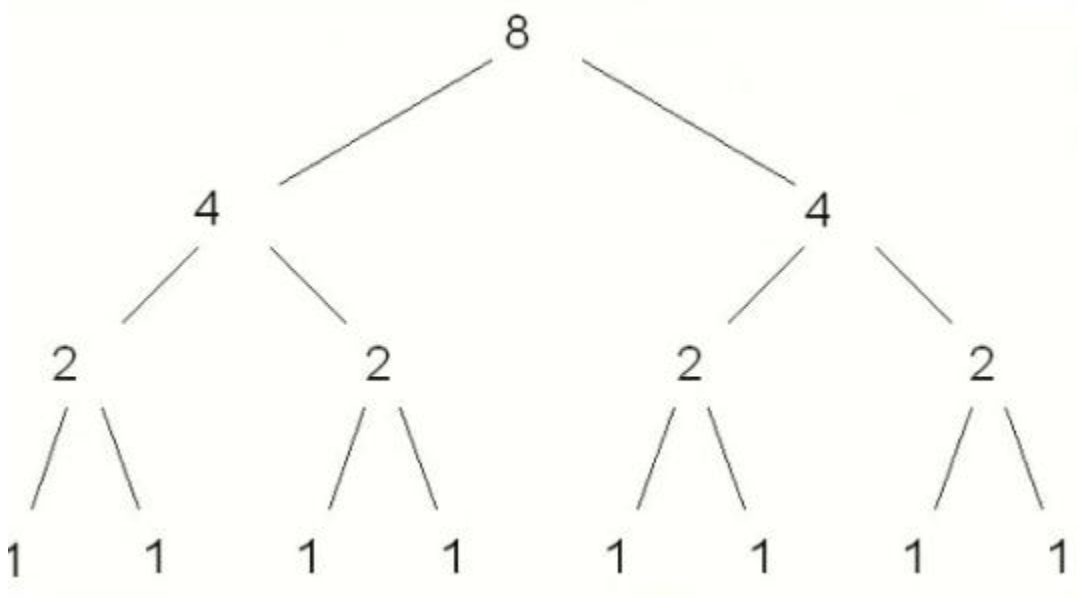
همانطور که در شکل مشخص می باشد، عدد ثابت 8 در ریشه قرار می گیرد و به تعداد ضریب تابع بازگشتی که مقدار 2 می باشد یال خواهیم داشت و هر یال $T(4)$ خواهد شد. در مرحله بعد روشن است که باید رابطه $T(4)$ و درخت مربوط به آن باید به درخت اصلی اضافه شود.

$$T(4) = 2T(2) + 4$$



در مرحله بعد باز این روال ادامه دارد و باید رابطه $T(2)$ در درخت اضافه شود.

$$T(2) = 2T(1) + 2$$



در درخت نهایی چون داریم $T(1) = 1$ در نتیجه مستقیم مقدار 1 در درخت قرار گرفت که با این کار عمل رسم درخت بازگشت به پایان می رسد. حال برای محاسبه هزینه کافیست که این اعداد با هم جمع شوند.

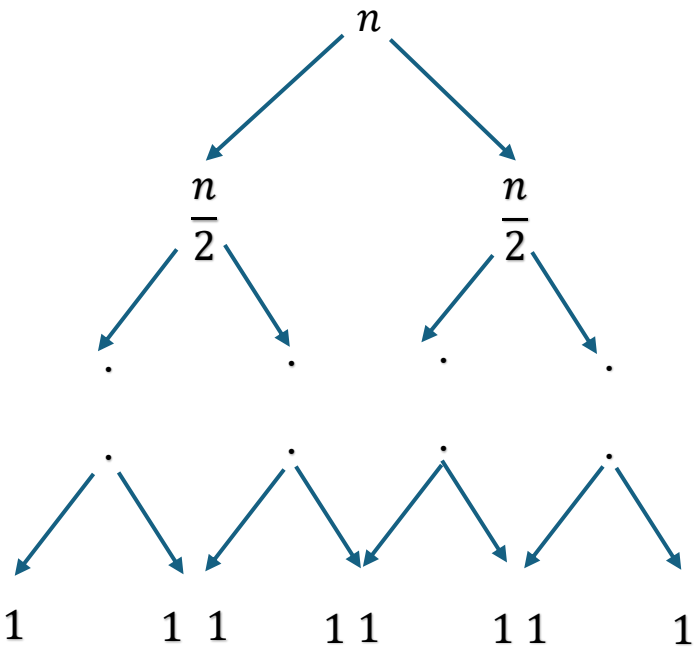
$$8 + (4 + 4) + (2 + 2 + 2 + 2) + (1 + 1 + 1 + 1 + 1 + 1 + 1 + 1) = 4 * 8 = 32$$

نکته: در این مثال مشخص می باشد که درخت ما دارای 4 $(logn + 1)$سطح می باشد و هزینه هر سطح 8 که همان مقدار n می باشد.

پس در حالت کلی داریم:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$T(1) = 1$$



$$(logn + 1) * n = nlogn + n$$

3- قضیه اصلی

در این روش داریم:

اگر داشته باشیم $a \geq 1, b > 1$

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

آنگاه داریم:

$$\begin{cases} \theta\left(n^{log_b^a}\right) & f(n) < n^{log_b^a} \\ \theta(f(n) * lgn) & f(n) = n^{log_b^a} \\ \theta\left(f(n)\right) & f(n) > n^{log_b^a} \end{cases}$$

مثال:

$$T(n) = 4T\left(\frac{n}{2}\right) + logn$$

در این مثال $a = 4$ و $b = 2$ و $f(n) = logn$ می باشد. در ابتدا باید مقدار $n^{log_b^a}$ را حساب کنیم.

$$n^{log_b^a} = n^{log_2^4} = n^2$$

در نتیجه:

$$logn < n^2 => \theta(n^2)$$

مثال:

$$T(n) = 3T\left(\frac{n}{4}\right) + nlogn$$

$$n^{log_b^a} = n^{log_4^3} < nlogn => \theta(nlogn)$$

مثال:

$$T(n) = T\left(\frac{2n}{3}\right) + 1$$

داریم: $a = 1\,, b = \frac{3}{2}\,, f(n) = 1$

$$1 = n^{log_{\frac{3}{2}}^{1}} => \theta(f(n) * logn) = \theta(logn)$$

فصل 3

روش تقسیم و حل

(Divide & Conquer)

الگوریتم‌های تقسیم و حل از تکنیک توابع بازگشتی برای حل مسائل مشخص شده استفاده می‌کنند. الگوریتم‌های تقسیم و حل مسائل پیچیده را به بخش‌های کوچکتری تجزیه می‌کنند. یعنی جایی که راه حل تعریف شده به صورت بازگشتی بر روی هر کدام از بخش‌های تجزیه شده اعمال می‌شود. هر کدام از این بخش‌های کوچک زیر مجموعه مربوط به مسئله اصلی و بزرگ به صورت جداگانه حل می‌شوند. سپس راه حل‌های آماده به صورت منظم با یکدیگر ترکیب شده و مسئله اصلی را حل می‌کنند.

روش تقسیم و حل دارای 3 مرحله می باشد:

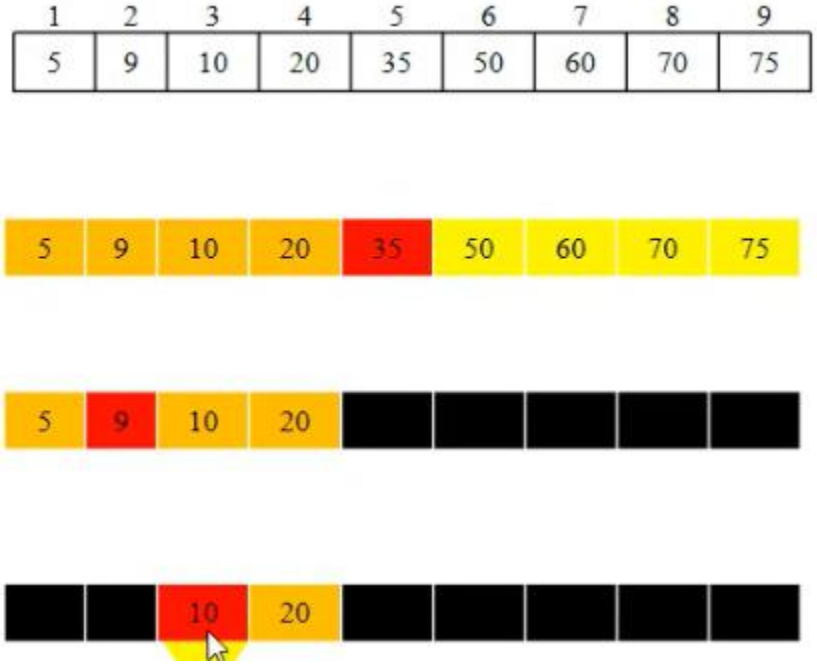
1- تجزیه مسئله بزرگتر به زیرمسائل کوچکتر (تقسیم)
2- استفاده بازگشتی از توابع برای حل کردن هر کدام از زیرمسائل کوچکتر (حل)
3- ترکیب پاسخ زیر مسئله ها برای رسیدن به جواب مسئله اصلی

اگر یک مسئله با n ورودی داشته باشیم و آن را بتوانیم به دو مسئله تقریبا مساوی تقسیم کنیم، آنگاه زیر مسئله ها را بصورت بازگشتی حل می کنیم و سپس با هزینه خطی حاصل این دو را ترکیب می کنیم تا جواب مسئله بدست بیاید. رابطه بازگشتی به صورت زیر می باشد.

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) => T(n) = O(nlogn)$$

مثال 1: الگوریتم جستجوی دودویی(*Binary Search*)

در این الگوریتم می دانیم که یک آرایه با n عنصر داریم که به صورت صعودی(نزولی) مرتب می باشد. در مثال زیر می خواهیم محل قرار گرفتن مقدار 10 را داخل آرایه پیدا کنیم.



همانطور که در شکل مشخص می باشد، ابتدا وسط آرایه پیدا می شود، عدد 10 با مقدار موجود در این محل(35) مقایسه می شود اگر برابر بود که محل مورد نظر پیدا شده است. در غیر اینصورت در این حالت چون از مقدار میانی کوچکتر است پس در نیمه اول آرایه جستجو ادامه پیدا می کند. این فرایند آنقدر ادامه پیدا میکند تا عدد پیدا شود و یا موجود نباشد.

در این الگوریتم همواره ما شاهد نصف شدن آرایه هستیم. در تصویر زیر تکه کد پیاده سازی این الگوریتم نمایش داده شده است.

```
static int BinarySearch(int[] numbers, int key, int low, int high)
{
    if (high < low)
        return -1;

    int mid = low + (high - low) / 2;

    if (numbers[mid] > key)
    {
        return BinarySearch(numbers, key, low, mid - 1);
    }
    else if (numbers[mid] < key)
    {
        return BinarySearch(numbers, key, mid + 1, high);
    }
    else
    {
        return mid;
    }
}
```

روشن است رابطه بازگشتی این الگوریتم به صورت زیر نوشته می شود و مرتبه اجرائی الگوریتم به کمک روش قضیه اصلی بدست می آید.

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

$$n^{log_b^a} = n^{log_2^1} = 1 =>$$

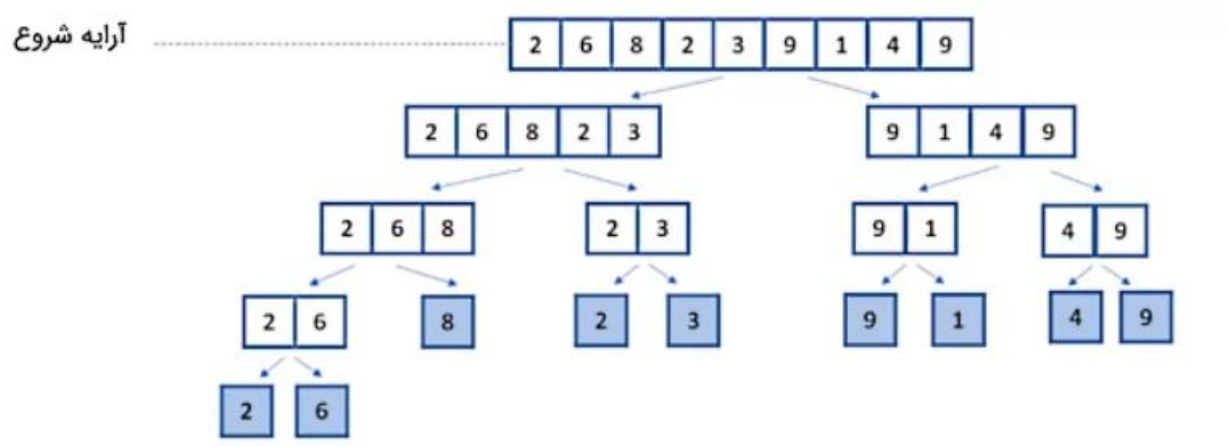
$$f(n) = 1 => O(logn)$$

مثال 2: مرتب سازی ادغامی (*Merge Sort*)

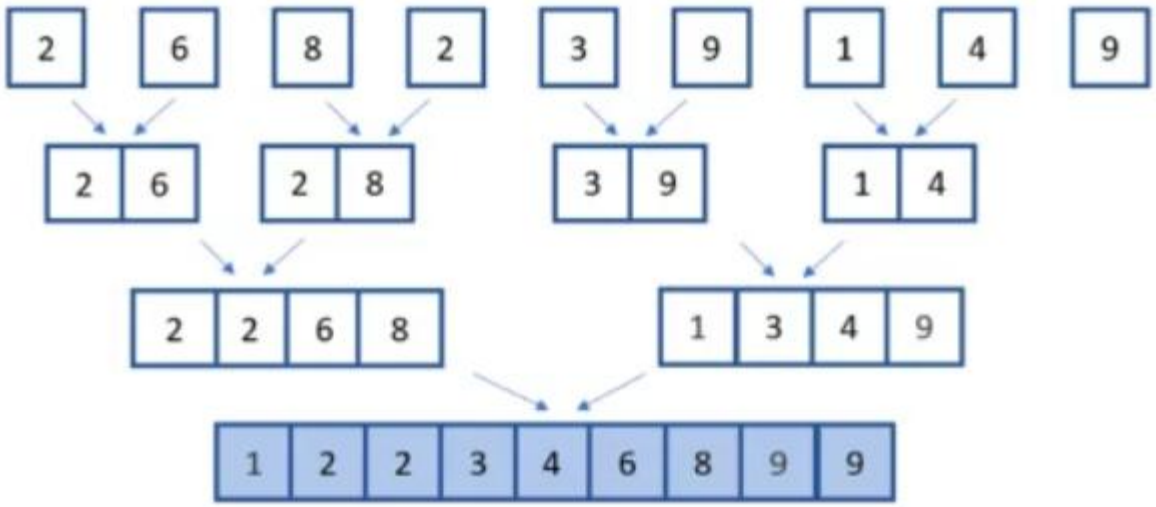
در ساده‌ترین شکل ممکن، الگوریتم مرتب‌سازی ادغامی، لیست نامرتبی را به تعداد n عدد زیرلیست تقسیم می‌کند. هر کدام از این زیرلیست‌ها در ساده‌ترین شکل ممکن خود و فقط شامل یک عنصر هستند. بعد به صورت تکراری زیرلیست‌ها را در یکدیگر ادغام می‌کند تا زیرلیست‌های جدید مرتب شده‌ای ایجاد شوند. این عملیات را تا زمانی می‌توان انجام داد که فقط یک زیرلیست باقی مانده باشد.

دو عملیات اصلی این الگوریتم شامل رفتارهای تقسیم و حل می‌شود. مرحله تقسیم مربوط به زمانی می‌شود که آرایه به دو نیمه تقسیم شده است و تا موقعی که به عناصر غیر تقسیم برسیم ادامه خواهد یافت(بصورت بازگشتی). در حالی که مرحله حل مربوط به زمانی است که نیمه‌های مرتب شده به صورت مجزا از یکدیگر را در کنار هم قرار می‌دهیم.

در تصویر زیر، می‌توانید تمام عملیات تقسیم، مقایسه و ترکیب را ببینید. این قدم‌ها به ترتیب در الگوریتم مرتب سازی ادغامی شامل می‌شوند.



تصویر بالا مربوط به مرحله اول و تقسیم لیست داده شده در مسئله است. اما تصویر پایین فرایند مقایسه و ترکیب را برای ساخت لیست نهایی جواب نشان می‌دهد.



مرحله مرتب‌سازی و ترکیب زیرآرایه‌ها

در شکل زیر پیاده سازی کلی این الگوریتم نمایش داده شده است.

```
static void MergeSort(int[] numbers, int left, int right)
{
    if (left < right)
    {
        int mid = (right + left) / 2;

        MergeSort(numbers, left, mid);
        MergeSort(numbers, mid + 1, right);
        Merge(numbers, left, mid, right);
    }
}
```

می توان رابطه بازگشتی این الگوریتم را به صورت زیر بیان کرد و با توجه به روش قضیه اصلی به نتیجه مورد نظر رسید.

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1$$

$$=> T(n) = nlogn - (n-1) \in \theta(nlogn)$$

مثال 3: الگوریتم مرتب سازی سریع (Quick Sort)

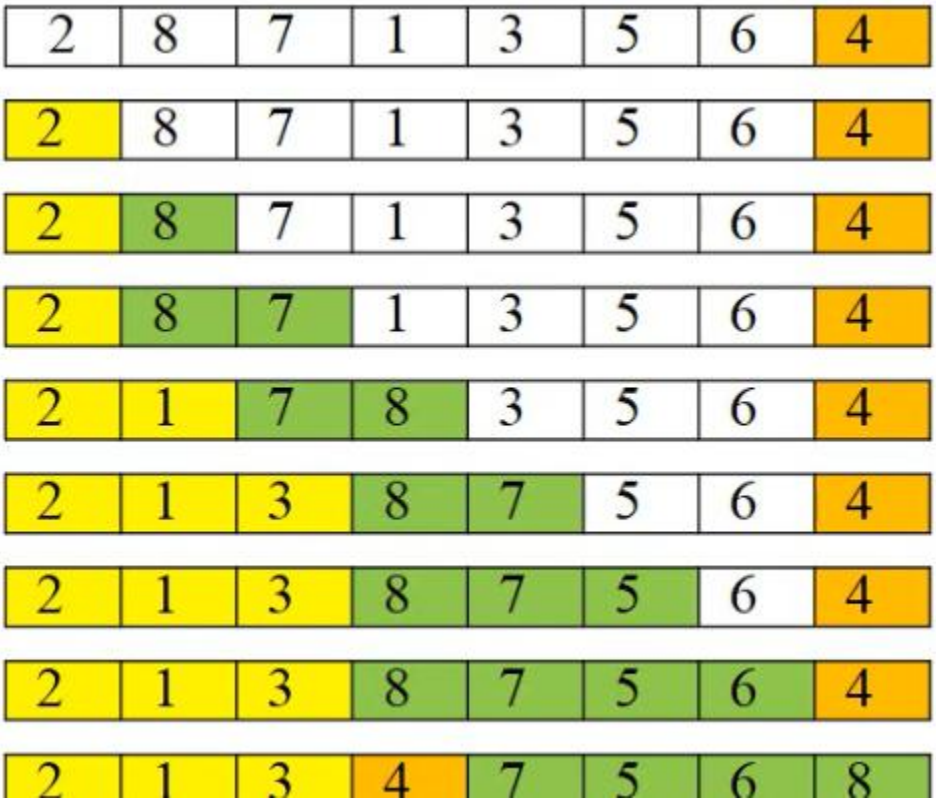
به‌طور کلی در این الگوریتم، یکی از عناصر آرایه به‌عنوان عنصر محور(Pivot) ، انتخاب شده و آرایه از همان نقطه به ۲ زیرآرایه تقسیم می‌شود. سپس، عناصری که از عنصر *Pivot* کوچکتر هستند به زیرآرایه سمت چپ منتقل می‌شوند و همچنین عناصر بزرگتر از عنصر *Pivot* در زیرآرایه سمت راست قرار می‌گیرند. به‌همین ترتیب، هر یک از این زیرآرایه‌ها دوباره با یک عنصر *Pivot*به ۲ زیرآرایه تقسیم شده و این روند به‌صورت بازگشتی ادامه پیدا می‌کند. در نهایت به نقطه‌ای خواهیم رسید که تمامی زیرآرایه‌ها تنها یک عنصر دارند. اکنون می‌توانیم با ترکیب زیرآرایه‌ها در قالب یک لیست، به داده‌های مرتب شده خود دست پیدا کنیم.

```
static int[] QuickSort(int[] a, int p, int r)
{
    if (p<r)
    {
        int partition = Partition(a, p, r);
        QuickSort(a, p, partition-1);
        QuickSort(a, partition+1, r);
    }
    return a;
}
```

در شکل بالا پیاده سازی کلی این الگوریتم را مشاهده می کنید.

پس از اعمال تابع *Partition* تمام مقادیر کوچکتر در سمت چپ و بزرگتر ها در سمت راست محور قرار می گیرند.

مثال:



در این مثال عنصر آخر(4) به عنوان محور انتخاب شده است. تمام عناصر با محور مقایسه می شود، اگر کوچک تر باشد با رنگ زرد و در صورت بزرگتر بودن با رنگ سبز نشان داده می شوند. در نهایت عنصر محور با اولین مقدار لیست سبز جابجا می شود.

پیاده سازی تابع Partition به صورت زیر می باشد. که در این پیاده سازی عنصر اول آرایه به عنوان محور انتخاب شده است.

```
static int Partition(int[] a,int p,int r)
{
    int m = a[p],i=p+1,j=r;
    do
    {
        while (a[i] < m) i++;
        while (a[j] > m) j--;
        if (i < j) swap(a[i], a[j]);
    } while (i < j);
    swap(a[p], a[j]);
    return m;
}
```

نحوه انتخاب عنصر محور:

- می‌توانیم همیشه اولین عنصر از داده‌های خود را به‌عنوان *Pivot* انتخاب کنیم.
- یا اینکه عنصر انتهایی از مجموعه داده‌های خود را به‌عنوان *Pivot* انتخاب کنیم.
- می‌توانیم عنصر *Pivot* را به‌صورت رندوم یا تصادفی انتخاب کنیم که در این‌صورت هریک از عناصر مجموعه، شانس *Pivot* شدن را خواهند داشت.
- همچنین، امکان انتخاب عنصر میانی مجموعه، به‌عنوان *Pivot* نیز وجود دارد.

در بدترین حالت این الگوریتم دارای پیچیدگی اجرایی $O(n^2)$ و در بهترین حالت و حالت میانگین دارای پیچیدگی اجرایی $O(nlogn)$ می باشد.

ضرب دو عدد صحیح بزرگ

اعدادی وجود دارند که از حداکثر مقدار بزرگترین نوع داده موجود در زبان برنامه نویسی بزرگتر هستند و نمی توان آنها را در حافظه نگهداری کرد و وقتی در محاسبات نیز شرکت می کنند ممکن است بزرگتر هم شوند و نگهداری آنها به صورت عادی غیر ممکن است. یکی از راه هایی که می توان برای محاسبات اعداد بزرگ از آن استفاده کرد، روش تقسیم و حل می باشد.

در این روش، اگر عدد u یک عدد صحیح با n رقم باشد آن را به دو عدد صحیح تقسیم می کنیم.

1- عددی صحیح با $\left\lfloor \frac{n}{2} \right\rfloor$ رقم و آن را x می نامیم.

2- عددی صحیح با $\left\lfloor \frac{n}{2} \right\rfloor$ رقم و آن را y می نامیم.

حال عدد را به صورت زیر می توان نوشت که در این رابطه $m = \left\lfloor \frac{n}{2} \right\rfloor$ می باشد.

$$u = x * 10^m + y$$

مثال: عدد 96584236581 مفروض است. در این عدد تعداد ارقام n=11 می باشد.

$$x = \left\lfloor \frac{11}{2} \right\rfloor = 6 \qquad y = \left\lfloor \frac{11}{2} \right\rfloor = 5 \qquad m = \left\lfloor \frac{11}{2} \right\rfloor = 5$$

در نتیجه عدد مورد نظر به صورت زیر قابل نوشتن می باشد

$$u = 965842 * 10^5 + 36581$$

حال اگر دو عدد بزرگ u و v داشته باشیم و بخواهیم آنها را در هم ضرب کنیم، مانند مثال کلی زیر قابل محاسبه می باشد.

$$uv = (x * 10^m + y)(w * 10^k + z)$$
$$= xw * 10^{km} + xz * 10^m + yw * 10^k + yz$$

مثال:

U= $123456 = 123 * 10^3$+*456*

V = $45612 = 456 * 10^2$+*12*

UV = $123 * 456 * 10^6$+(123*12* 10^3)+(456*456* 10^2)+(12*456)

در نهایت با در نظر نگرفتن عمل ضرب مقادیر پایه 10 تعداد 4 ضرب خواهیم داشت. لذا جهت محاسبه پیچیدگی اجرایی با روش قضیه اصلی داریم:

T(n)=4T($\frac{n}{2}$)+cn ➔ $\theta\left(n^{\log_2^4}\right) = \theta(n^2)$

انتخاب i امین کوچکترین عنصر

یکی دیگر از کاربرد های روش تقسیم و حل می باشد. روشن است اولین کوچک ترین عنصر در یک لیست را کوچکترین مقدار Min و آخرین آن بزرگترین مقدار Max می باشد. آرایه زیر با 8 عنصر مفروض می باشد.

6	9	13	4	8	1	2	12

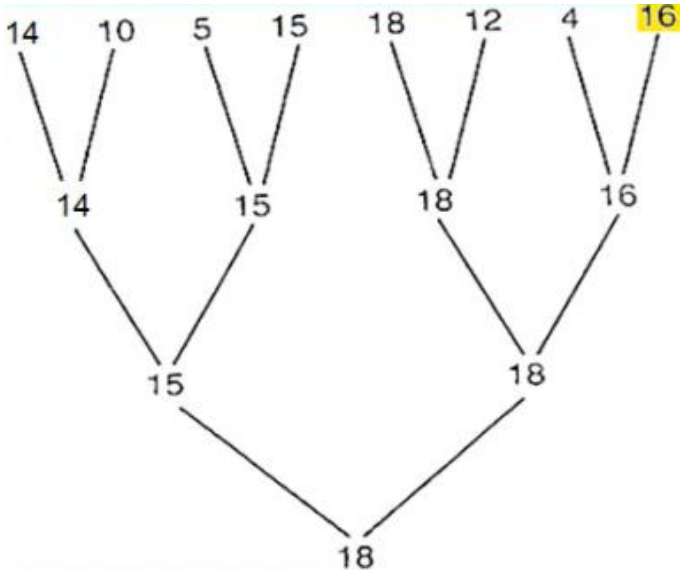
در این آرایه قصد داریم 7 امین کوچکترین عنصر(i=7) را پیدا کنیم که عدد 12 می باشد. در این آرایه عنصر اول را به عنوان محور انتخاب می کنیم و عمل پارتیشن بندی(در توضیح جستجوی سریع در مورد آن توضیح داده شده است) را بر روی لیست اعمال می کنیم. در نتیجه تمام عناصر کوچکتر از محور در سمت چپ و عناصر بزرگتر در سمت راست محور قرار می گیرند.

1	2	4	6	8	9	13	12

پس از پارتیشن بندی محور در عنصر k=4 قرار می گیرد. و چون 12 از محور بزرگتر می باشد، محدوده جستجو به سمت چپ لیست محدود میشود و حال نیاز داریم که سومین کوچکترین عنصر (i-k=7-4=3) را در این لیست پیدا کنیم. روشن است با پیدا کردن این روش در هر مرحله لیست جستجو کاهش می یابد.

یافتن بزرگترین کلید دوم

مجموعه اعدادی را داریم که در یک ساختار درخت دودویی با هم به رقابت می پردازند و برنده این تورنمنت به عنوان اولین بزرگترین کلید انتخاب می شود. به ازای n عنصر n-1 مقایسه خواهیم داشت.



در انتها مشاهده می شود که عدد 18 عنوان اولین بزرگترین کلید انتخاب می شود. حال کلیدهایی که یک مقایسه را به به 18 واگذار کرده اند به لیست اضافه می شوند[12,16,15]. حال در این لیست با داشتن $logn$ عنصر مشخص است که $logn-1$ مقایسه خواهیم داشت. در نتیجه فرمول محاسبه تعداد مقایسه ها برای یافتن بزرگترین کلید دوم به شکل زیر می باشد.

$$(n-1)+(logn-1)=n+logn-2$$

همچنین رابطه کلی زیر برای یافتن k امین بزرگترین کلید در مجموعه ای با n عنصر موجود می باشد.

$$n+(k-1)\left\lceil \log\left(\frac{n}{k-1}\right)\right\rceil - k \qquad => \ \theta(n)$$

$$k>1$$

فصل 4

روش حریصانه

(Greedy)

در رویکرد حریصانه، از روی ویژگی افراد خسیس، پر طمع و حریص الگو برداری شده است. این افراد هرگز به گذشته و یا آینده فکر نمی کنند، بلکه تمام فکر آنها جمع آوری سرمایه در حال حاضر است.

بطور کلی در این رویکرد برای تصمیم گیری بدون توجه به انتخاب های قبلی و بعدی بهترین انتخاب بر اساس معیارهای کنونی صورت می گیرد. روش های حریصانه برای تولید جواب، دنباله ای از عناصر انتخابی استفاده می کنند که هر کدام از آنها در آن لحظه، بهترین انتخاب بنظر می رسند. به عبارت دقیق تر هر انتخاب، بطور محلی بهینه است.

الگوریتم Greedy ، یک الگوریتم ساده است که در مسائل بهینه سازی استفاده می شود. این الگوریتم در هر مرحله، انتخاب بهینه را انجام می دهد تا بتواند راه بهینه برای حل کل مسئله را بیابد. این الگوریتم در برخی مشکلات کاملاً موفق می باشد، با این حال، در بسیاری از مشکلات، یک استراتژی Greedy راه حل بهینه را پیدا نمی کند.

روش حریصانه

در حل مسائل با شیوه حریصانه با یک مجموعه تهی شروع می کنیم و هر تکرار از سه بخش تشکیل می شود:

1- روال انتخاب[1]: در این قسمت در بین عناصر باقیمانده بهترین عنصر انتخاب می شود.
2- امکان سنجی[2]: عنصر انتخاب شده را در صورت امکان به مجموعه جواب اضافه می کنیم.
3- بررسی راه حل[3]: اگر مجموعه جواب بیانگر یک راه حل است، آن را برمی گردانیم در غیر اینصورت مراحل بالا تکرار می شود.

[1] Selection Procedure
[2] Feasibility Check
[3] Solution Check

- **مسئله خرد کردن پول**

فرض کنید به فروشگاهی رفته اید و پس از خرید و پراخت هزینه، فروشنده باید باقی پول شما را به صورت صحیح و با درشت ترین سکه و یا اسکناس ممکن برگرداند. ابتدا فروشنده بزرگترین مقدار پول از نظر ارزش را انتخاب می کند، در اینجا ملاک فروشنده ارزش پول است(روال انتخاب). سپس فروشنده باید حساب کند که با اضافه کردن این پول به بقیه پول ها، جمع کل آنها از چیزی که باید باشد بیشتر می شود یا نمی شود(امکان سنجی). اگر با اضافه شدن این پول مقدار کل از مقدار لازم بیشتر نشد، به مجموعه اضافه می شود. سپس کنترل می کند که مقدار پول با مقدار لازم برابر است یا برابر نیست(بررسی راه حل). **اگر برابر نبود باید پول دیگری را انتخاب کند و فرآیند را تکرار کند**. او چندین بار این کار را انجام می دهد تا پول لازم را جور کند و به صاحبش تحویل دهد و یا دیگر پولی برایش باقی نماند که قادر به بازگرداند مابقی پول مشتری نمی باشد.

مثال :

در فروشگاهی سکه های 1، 5، 10، 25 سنتی و نیم دلاری از هر کدام حداقل یک سکه موجود می باشد. نحوه بازگرداندن مبلغ 36 سنت به مشتری را بررسی کنید.

1- پرداخت یک سکه 25 سنتی
2- پرداخت یک سکه 10 سنتی
3- پرداخت یک سکه 10 سنتی با توجه به اینکه از پول لازم بیشتر می شود لغو می شود.
4- پرداخت یک سکه 5 سنتی با توجه به اینکه از پول لازم بیشتر می شود لغو می شود.
5- پرداخت یک سکه 1 سنتی

خواهیم دید با پرداخت یک سکه 25، یک سکه 10 و سکه ای یک سنتی می توان پول مشتری را بازگرداند.

مثال:

در فروشگاهی سکه های 1، 5، 10، 12، 25 سنتی و نیم دلاری از هر کدام حداقل یک سکه موجود می باشد. نحوه بازگرداندن مبلغ 16 سنت به مشتری را بررسی کنید.

1- پرداخت سکه 12 سنتی
2- پرداخت یک سکه 10 سنتی با توجه به اینکه از پول لازم بیشتر می شود لغو می شود.
3- پرداخت یک سکه 5 سنتی با توجه به اینکه از پول لازم بیشتر می شود لغو می شود.
4- پرداخت 4 سکه یک سنتی

دیده می شود که با پرداخت پنج سکه می توان مابقی پول مشتری را بازگرداند.

اما با کمی دقت بیشتر می توان دریافت که طور دیگری هم می شد عمل کرد.

تنها با سه سکه به جای پنج سکه، پرداخت یک سکه 10، یک سکه 5 و یک سکه 1 سنتی که حالت بهینه می باشد.

در اینجا مشاهده می شود مه روش حریصانه نتوانسته است به پاسخ بهینه دست پیدا کند.

- **مسئله زمانبندی**

بیان این مسئله را با یک مثال پیش می بریم. فرض کنید سه کار داریم که قرار است به ترتیب انجام شود. هزینه زمانی هر کار j1=5، j2=10 و j3= 4 می باشد. در محاسبه زمان کل، زمان کار اول با حاصل جمع زمان کار دوم و زمان انتظار(زمان همان کار اول) جمع شده و سپس با مجموع کار سوم و زمان انتظار(زمان کار اول و دوم)، جمع می شوند. حال ترکیب های مختلف انجام کارها را در جدول زیر می بینیم.

زمانبندی	زمان کل
J1, j2, j3	5+(5+10) +(5+10+4) =39
J1 , j3 , j2	5+(5+4)+(5+4+10)=33
J2 , j1 , j3	10+(10+5)+(10+5+4)=44
J2 , j3 , j1	10+(10+4)+(10+4+5)=43
J3 , j1, j2	4+(4+5)+(4+5+10)=32
J3 , j2 , j1	4+(4+10)+(4+10+5)=37

با بررسی زمانبندی ها متوجه می شویم که اگر کارهای یا زمان کمتر را زودتر انجام دهیم زمانبندی بهینه خواهیم داشت. پس می توانیم با مرتب کردن کارها به صورت صعودی بر اساس زمان کار و اعمال روش حریصانه در این مسئله، به پاسخ بهینه دست یافت.

زمان الگوریتمی کلی در این مسئله به صورت فاکتوریل می باشد و n! حالت برای همه زمانبندی های ممکن وجود دارد.

پیچیدگی اجرایی زمانبندی الگوریتم با توجه به اینکه تنها عملی که انجام می شود، مرتب سازی زمان کار ها است $\theta(nlogn)$ می باشد.

- **مسئله زمانبندی با مهلت معین**

در این زمانبندی، کار ها دارای زمان و سود مشخص می باشند. اگر کاری در زمان مشخص و تعیین شده به پایان برسد، سود آن حاصل می شود. هدف، زمانبندی کار ها می باشد که در نهایت بیشترین سود را داشته باشد و نیازی هم نیست که همه کارها زمانبندی شوند.
روش حریصانه برای این نوع زمانبندی به اینصورت است که کار ها بر اساس سود به صورت صعودی مرتب شوند و بررسی بر روی آنها انجام و در صورت امکان به زمانبندی اضافه شوند.

مثال:

جدول کارها و مشخصات آنها به صورت زیر مفروض می باشد.

کار	مهلت	سود
1	2	30
2	1	35
3	2	25
4	1	40

** منظور از مهلت این است که اگر این مقدار 2 باشد، می تواند هم به عنوان کار اول و هم به عنوان کار دوم وارد شود. ولی اگر این مقدار یک باشد، تنها می تواند به عنوان اولین کار وارد زمانبندی شود.

در ذیل ترکیب زمانبندی های ممکن به همراه سود آنها را مشاهده می کنیم.

زمانبندی	سود حاصل
[1,3]	30+25=55
[2,1]	35+30=65
[2,3]	35+25=60
[3,1]	25+30=55
[4,1]	40+30=70
[4,3]	40+25=65

- مشاهده می شود که در این زمانبندی، فقط 2 کار زمانبندی می شود چون بیشترین مقدار موجود در ستون مهلت عدد 2 می باشد.
- ترکیب *[4,1]* بهترین ترکیب با بیشترین سود می باشد. ولی ترکیب *[1,4]* یک ترکیب مجاز نمی باشد. زیرا کار 4 با مهلت یک می باشد و تنها می تواند به عنوان کار اول وارد زمانبندی شود.

مثال : مطلوب است یک زمانبندی معتبر برای کار های لیست شده

کار	1	2	3	4	5	6	7
مهلت	3	1	1	3	1	3	2
سود	40	35	30	25	20	15	10

نکته1: با توجه به اینکه بیشترین مقدار در جمع مقادیر موجود در مهلت، عدد 3 می باشد، این یعنی ما می توانیم تنها 3 کار را زمانبندی کنیم

نکته 2: کار ها بر اساس سود به صورت نزولی مرتب شده اند.

مراحل زمانبندی:

1- یک مجموعه تهی به نام *R* به منظور قرار گرفتن کارهای زمانبندی شده تعریف می کنیم.
2- کار 1 به مجموعه *R* اضافه می شود زیرا با توجه به مهلتی که دارد می تواند در هر ترکیب سه تایی قرار گیرد.
3- کار 2 نیز در مجموعه *R* قرار میگیرد، زیرا ترکیب *[2,1]* مجاز می باشد.
4- کار 3 نمی تواند به مجموعه *R* اضافه شود. زیرا با وجود کار 2 که می تواند به عنوان اولین کار در مجموعه می باشد، این کار نمی تواند در ترکیب قرار بگیرد.
5- کار 4 وارد مجموعه *R* می شود، زیرا ترکیب *[2,1,4]* یک ترکیب مجاز می باشد.
6- قرار گرفتن مابقی کارها در مجموعه *R* مجاز نمی باشد، زیرا هیچ ترکیب مجازی بر اساس مهلت آنها حاصل نمی شود.

در پایان مجموعه *[2,1,4]* با سود 100 بهترین زمانبندی در این مثال می باشد.

در ذیل الگوریتم پیشنهادی برای این زمانبندی ارائه شده است.

```
schedule (n , deadline[ ] , J )
{
  sequence_of_integer  K;
  J =  [1];
  for (i = 2 ;  i <= n ; i++ ) {
        K = J  with  i  added according to Descending values of deadline[i]
        if  ( K  is  feasible)
              J = K;
   }
}
```

کار ها در زمان $\theta(nlogn)$ مرتب می شوند و در اختیار الگوریتم قرار می گیرند. در هر دور تکرار، حداکثر $i-1$ مقایسه برای اضافه کردن i به k انجام می شود و حداکثر i مقایسه نیز برای بررسی امکان پذیر بودن k لازم می باشد.

در بدترین حالت داریم:

$$\sum_{i=2}^{n}[i-1]+i=n^2-1\ \in\ \theta(n^2)$$

- **کد هافمن**[1]

کد هافمن نوع خاصی از کدهای پیشوندی[2] بهینه است که اغلب برای فشرده‌سازی بی‌اتلاف اطلاعات مورد استفاده قرار می‌گیرد. فرایند پیدا کردن یا استفاده از این کد به وسیله کدگذاری هافمن، با بهره‌گیری از الگوریتمی انجام می‌شود که توسط دیوید آ هافمن[3] توسعه داده شده است.

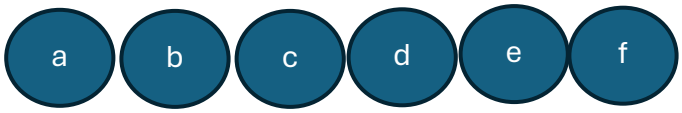
کدهای پیشوندی نوعی از کدها (توالی بیت‌ها) هستند که در آن‌ها کد اختصاص داده شده به یک کاراکتر، پیشوند کد تخصیص داده شده به هیچ کاراکتر دیگری نیست. این، روشی است که کدگذاری هافمن با استفاده از آن اطمینان حاصل می‌کند که هیچ ابهامی هنگام رمزگشایی توالی بیت‌های (جریان بیت) تولید شده وجود نخواهد داشت. در ادامه، برای درک بهتر موضوع، مثالی ارائه شده است .فرض می‌شود که چهار کاراکتر c ، b، aو d موجود هستند و کدهای طول متغیر متناظر با آن‌ها به ترتیب ۰۰، ۰۱، ۰ و ۱ است. این کدگذاری موجب ابهام می‌شود زیرا کد تخصیص یافته بهc ، پیشوند کدهای تخصیص یافته به a و b است .اگر جریان رشته فشرده شده ۰۰۰۱ است، خروجی که از حالت فشرده خارج شود امکان دارد $cccd$ یا ccb یا acd یا ab باشد. همچنین این نحوه کدگذاری کاهش حجم کد گذاری را تضمین می کند.

[1] Huffman Code
[2] Prefix Code
[3] David A. Huffman

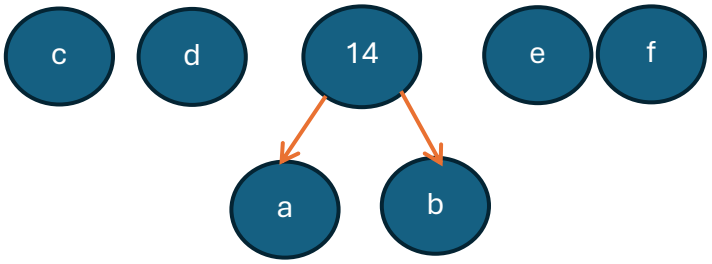
مراحل ساخت

برای توضیح نحوه عملکرد این الگوریتم از یک مثال استفاده می کنیم. برای همین لیست کاراکترها و تعداد تکرار های آن مرتب شده بر اساس تعداد تکرار به شرح زیر را فرض می کنیم.

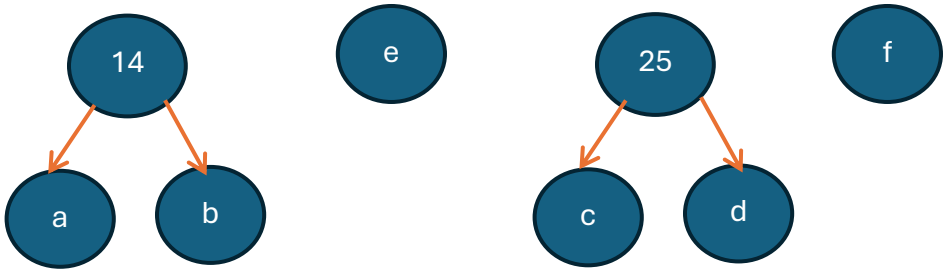
a	b	c	d	e	f
5	9	12	13	16	45



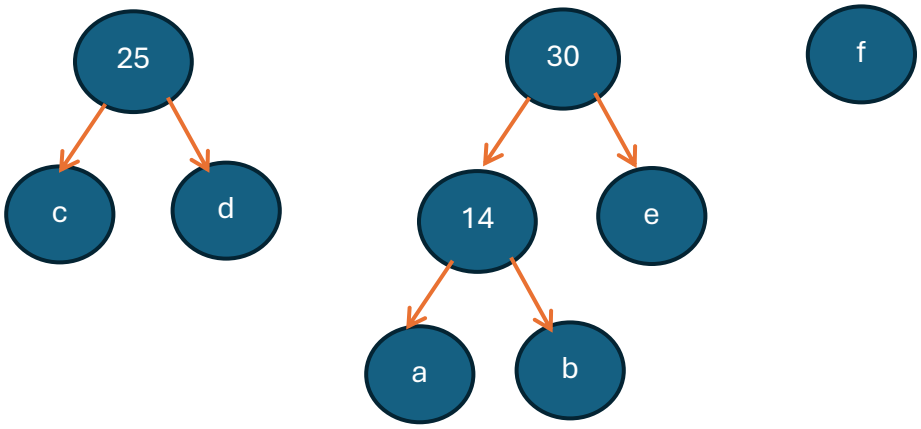
در مرحله بعد دو عنصر که تعداد تکرار کمتر را دارند (a,b) تشکیل یک درخت را می دهند که ریشه این درخت حاصل جمع تکرار ها می باشد و در محل مناسب قرار می گیرد که ترتیب صعودی عناصر حفظ شود.



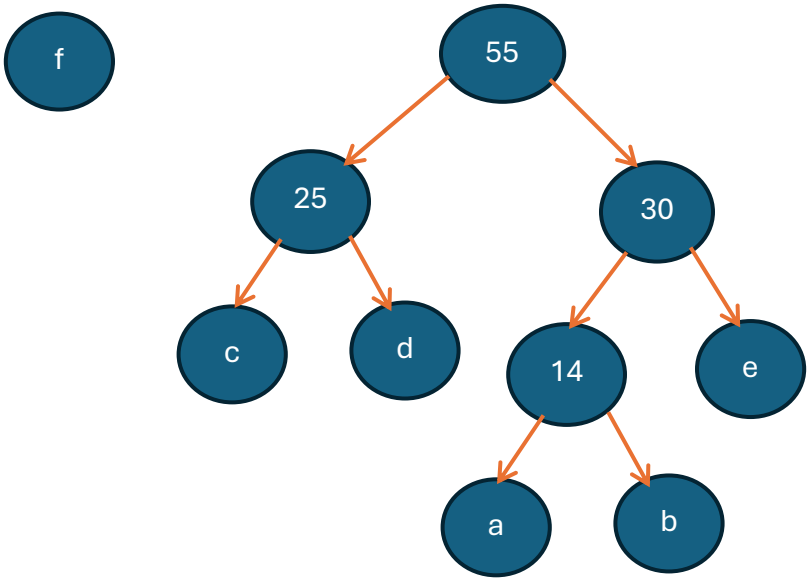
در مرحله دیگر باز دو عنصر با تعداد تکرار کمتر انتخاب می شود (c,d) و همان عملیات انجام می شود.



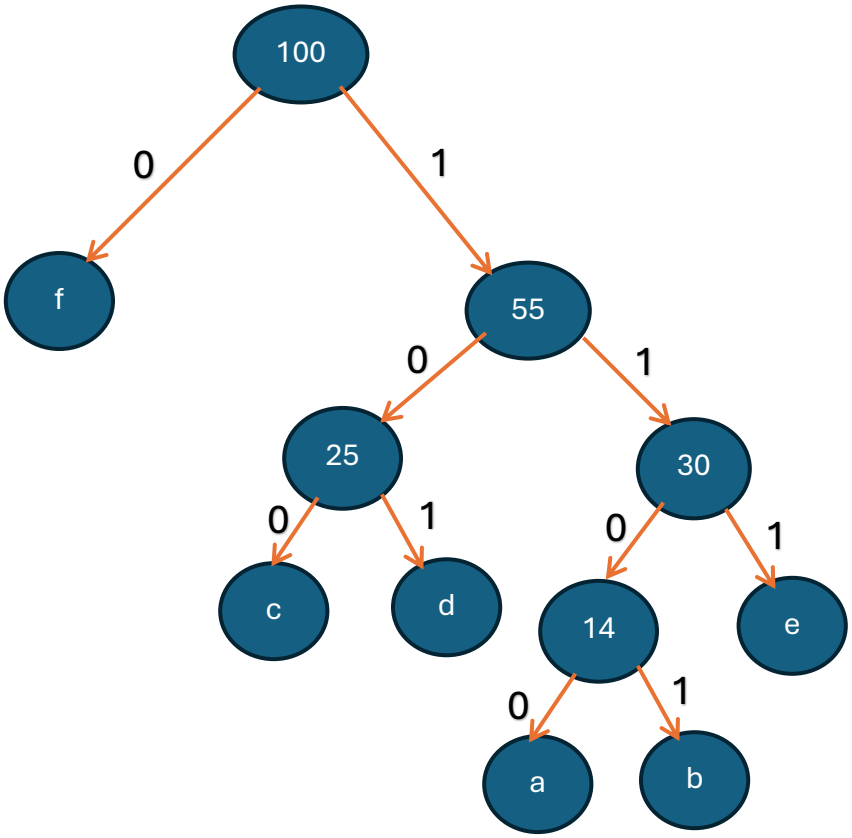
همین مراحل باز تکرا می شود و 14 و e اینبار انتخاب می شود و در جای مناسب قرار می گیرد.



در مرحله بعد 30 و 25 انتخاب می شوند و در جای مناسب قرار می گیرد.



در نهایت با انتخاب 55 و f ساخت درخت مورد نظر چون در هرم یک گره داریم به پایان می رسد.



پس از ترسیم درخت، حال باید برای یالهای درخت برچسب بزنیم. به تمام یال های سمت چپ برچسب 0 و به یال های سمت راست برچسب 1 میزنیم.

حال برای نوشتن کد های هر کاراکتر، از ریشه حرکت می کنیم و به سمت کاراکتر مورد نظر می رویم و مقادیر یالها را کنار هم می نویسیم.

a	b	c	d	e	f
1100	*1101*	*100*	*101*	*111*	*0*

مشاهده می شود که کاراکتر های با تکرار بیشتر تعداد بیتهای کمتری دارند تا مجموع بیتهای کد گذاری کاهش یابد.

- **مسئله کوله پشتی**

مسئله کوله پشتی به این صورت می باشد که یک کوله پشتی داریم که برای حمل گنجایش مشخصی دارد، همچنین اجسامی را هم داریم که می خواهیم با این کوله پشتی حمل کنیم که این اجسام دارای وزن (w) و ارزش(p) می باشند. هدف برداشتن اجسامی می باشد که بیشترین ارزش و سود را دارند. به منظور انتخاب اجسام، رابطه $\frac{p}{w}$ را برای هر کدام از اجسام محاسبه می کنیم تا ارزش هر کیلو از آن نوع مشخص گردد. سپس برای قرار دادن اجسام در کوله از بیشترین مقدار این رابطه شروع می کنیم و داخل کوله قرار می دهیم تا کوله پر شود و یا دیگر جسم دیگری در کوله جا نشود.

مسئله کوله پشتی با دو رویکرد ارائه می شود که در ذیل به آنها می پردازیم.

1- کوله پشتی 0 و 1

در این مسئله برای قرار دادن یک شئ در کوله یا باید همه آن در داخل کوله قرار بگیرد و یا اگر در کوله جا نمی شود نمی توان آن جسم را داخل کوله قرار داد.

مثال : کوله پشتی با گنجایش 30 کیلوگرم داریم. با توجه جدول زیر ارزش قطعاتی را که می توانیم در کوله قرار دهیم را محاسبه کنید.

قطعه	ارزش	وزن
A	50	5
B	60	10
C	140	20

برای این منظور ابتدا برای قطعات رابطه $\frac{p}{w}$ را حساب می کنیم.

$$A: \frac{50}{5} = 10 \ , \ B: \frac{60}{10} = 6 \ , \ C: \frac{140}{20} = 7$$

حال قطعه A که بیشترین مقدار در این بین را دارد را در کوله قرار می دهیم. با این کار 5 کیلو از گنجایش کوله پر می شود(25=5-30). در مرحله بعد باید قطعه C را برداریم که با این کار 20 کیلو دیگر از گنجایش کوله پر می شود(5=20-5-30). قطعه بعدی B می باشد که 10 کیلو وزن دارد و نمی تواند داخل کوله با گنجایش باقی مانده 5 کیلو قرار بگیرد. حال سود حاصل با جمع سود قطعات A و C بدست می آید.

$$50 + 140 = 190$$

2- کوله پشتی کسری

تنها تفاوت این مسئله با مسئله قبلی این است که در این روش می توان قسمتی از یک قطعه را نیز برداشت و در کوله پشتی قرار داد.

مثال: همان اعمالی را که در مثال قبل انجام دادیم را انجام می دهیم، فقط در اینجا می توانیم 5 کیلو از 10 کیلو از قطعه B را هم در کوله قرار دهیم. سود حاصل به صورت زیر محاسبه می شود.

$$50 + 140 + \frac{5}{10} * 60 = 220$$

این کسر می گوید که 5 کیلو از کوله خالی است و وزن قطعه B عدد 10 می باشد که این کسر از سود 60 را با قرار دادن 5 کیلو از قطعه B در کوله موفق شدیم کسب کنیم.

فصل 5

روش عقبگرد

(Backtracking)

این روش حل مسائل مانند بازی هزار تو می باشد که از یک درب وارد بازی می شویم و با مسیر های متعددی روبرو هستیم تا بتوانیم از درب خروج خارج شویم و به پایان بازی برسیم. مسیرهای پیش روی ما ممکن است که بن بست باشد، که اگر چنین مسیر هایی را انتخاب کنیم به ناچار مجبوریم برگردیم و مسیر دیگری را برای ادامه بازی انتخاب کنیم. این تلاش ها، انتخاب ها و بازگشت ها و در نهایت اصلاح مسیر ها ما را به مقصد خواهند رساند. روشن است اگر در هنگام مسیر و یا حتی بهتر از آن قبل از انتخاب مسیر تابلوی راهنمایی آنجا باشد و اطلاعات مناسبی را از مسیر به ما بدهد(مثلا این مسیر مسدود می باشد.) روشن است ما از تکرار و انتخاب مسیر های مسدود رها می شویم و راههای مناسب را انتخاب می کنیم و مسیر های بدون سرانجام کمتر می شود.

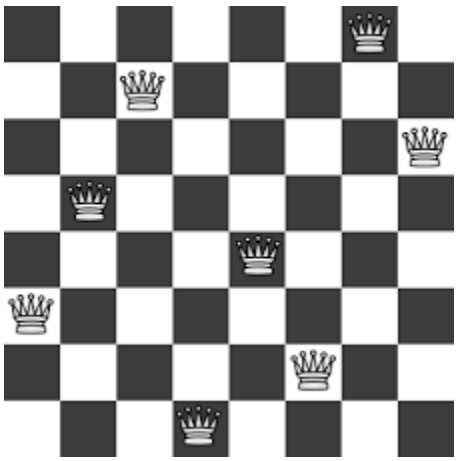
روش حل عقبگرد با همین رویکرد برای حل مسائل استفاده می کند، یک مسیر را می رود و با برخورد با بن بست و راه حل اشتباه به عقب بازگشته و مسیر را اصلاح می کند تا مجموعه جواب مناسب حاصل شود.

- **مسئله n وزیر**

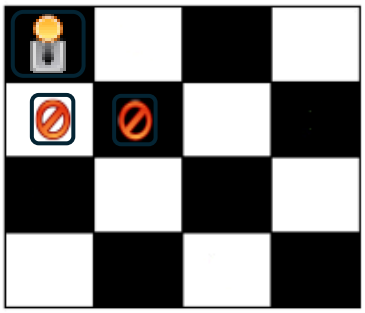
هدف از این مسئله این است که ما بتوانیم n وزیر را در یک صفحه شطرنج n*n طوری قرار دهیم که هیچ کدام از وزیر ها هم دیگر را تهدید(گارد) نکنند. این بدین معناست که هیچ دو وزیری در یک سطر، ستون یا قطر یکسانی نباشند.

مجموعه جواب: n موقعیت که وزیر ها در آن قرار گرفته اند.

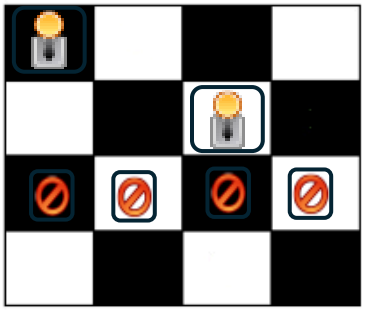
** مسئله n وزیر نمونه کلی از محیط واقعی شطرنج می باشد که در آن n=8 می باشد.



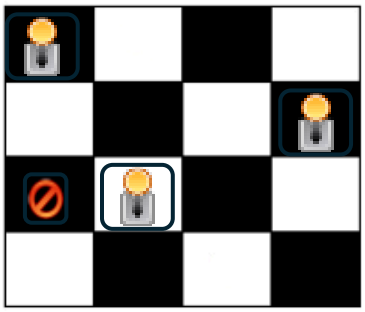
به منظور تشریح روش عقبگرد برای این مسئله از نمونه کوچکتر n=4 استفاده می کنیم که بتوانیم در مراحل کمتری به جواب برسیم. پس بررسی می کنیم حالتی را که یک صفحه شطرنج 4*4 داریم و می خواهیم 4 وزیر را طوری در این صفحه قرار دهیم که همدیگر را تهدید نکنند.



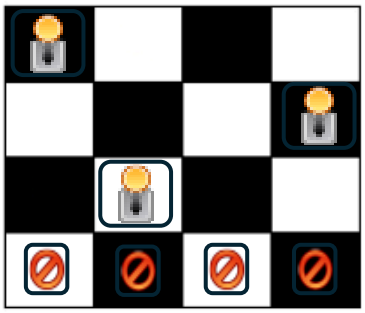
در اولین گام وزیر را در خانه ای در سطر اول و ستون اول قرار می دهیم و روشن است که وزیر دیگری در این سطر و ستون نمی تواند قرار بگیرد. همچنین خانه موجود در سطر دوم و ستون دوم هم انتخاب درستی نمی باشد. در نتیجه وزیر بعدی در خانه با سطر دوم و ستون سوم قرار می گیرد



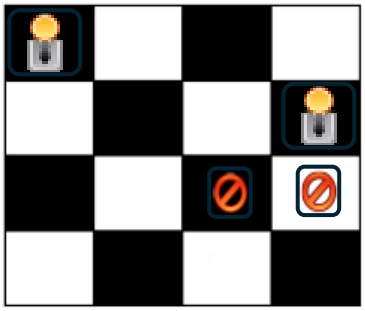
با این انتخاب مشاهده می شود که برای قرار دادن وزیر بعدی در سطر سوم تمام خانه ها نا معتبر می باشد و در واقع این مسیر اشتباه می باشدو برای همین یک مرحله به عقب بر میگردیم و وزیر موجود در سطر دوم را یک خانه به جلو حرکت می دهیم.



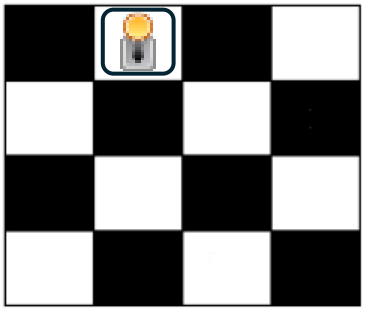
در این مرحله می بینیم که برای وزیر سوم در سطر سوم ستون دوم مناسب می باشد. اما در مرحله بعد و برای وزیر چهارم باز محل معتبری یافت نمی شود.



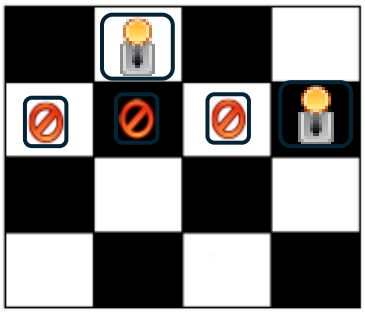
وقتی به این مشکل برخورد می کنیم باید برگشت به عقب کنیم و جای وزیر در سطر سوم را تغییر دهیم ولی مشاهده می شود که محل مناسب دیگری وجود ندارد.



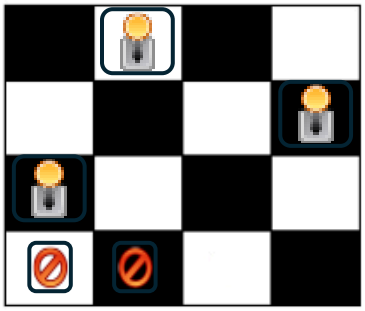
در اینجا متوجه می شویم که مشکل از سطر دوم نمی باشد و باید وزیر سطر اول جابجا شود، در نتیجه بازگشت به عقب کرده و وزیر موجود در سطر اول را یک ستون به جلو می بریم.



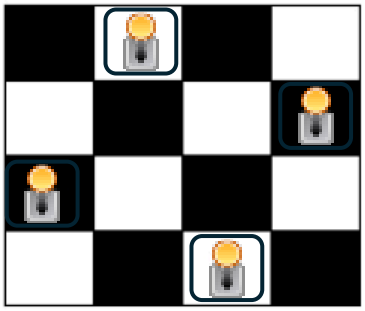
حال باید انتخاب ها را از ابتدا شروع کنیم. و با توجه به محدودیتهای موجود ستون آخر از سطر دوم جای مناسب برای وزیر دوم می باشد.



در گام بعدی وزیر بعدی می تواند در ستون اول از سطر سوم قرار بگیرد و مشکلی هم بوجود نمی آید و کاملا معتبر می باشد.

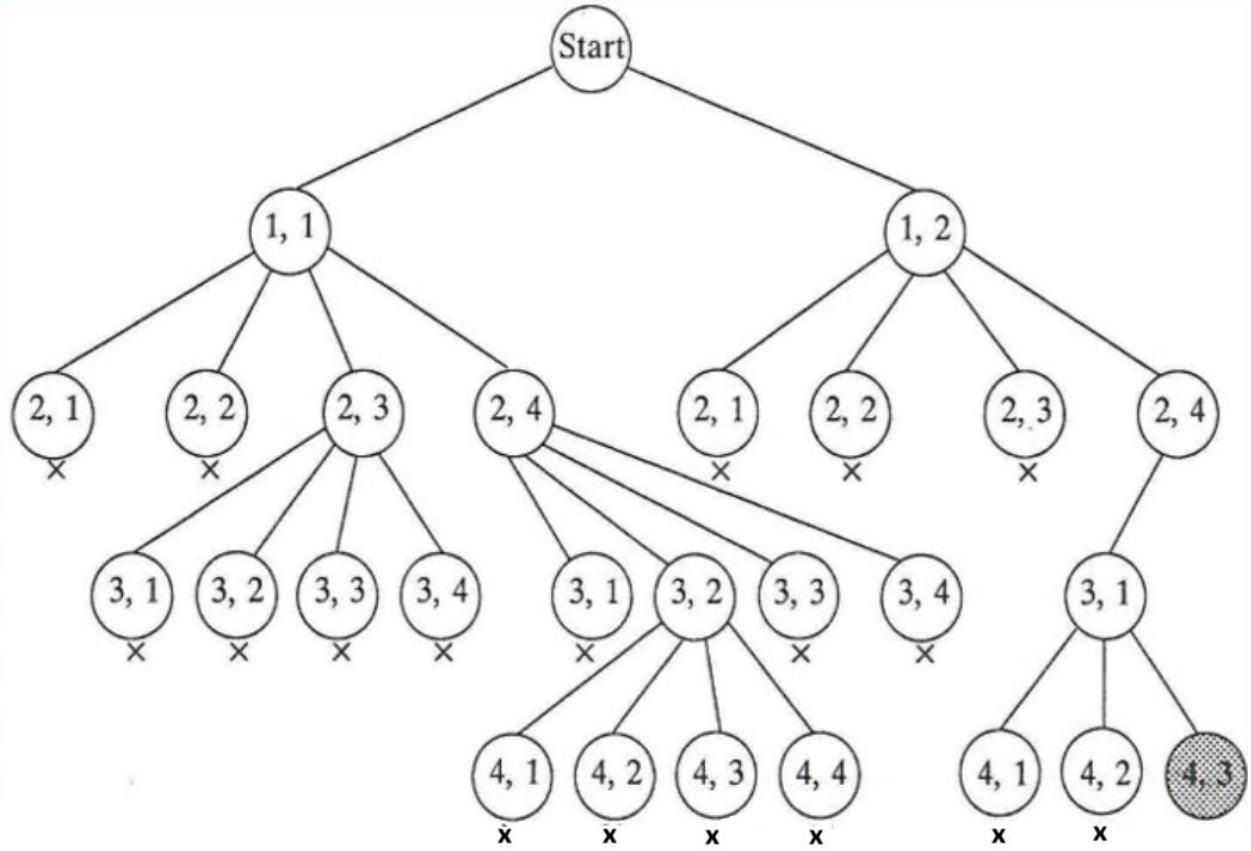


در مرحله بعدی و در تعیین وزیر آخر و وجود محدودیتهای مشخص شده، وزیر چهارم می تواند در ستون سوم از سطر چهارم قرار بگیرد و در نتیجه به پاسخ مورد نظر خواهیم رسید.



می توان با تکرار این روش حل در این نمونه یافت که آیا چیدمان دیگری وجود دارد یا این تنها راه حل می باشد.

نمایش درخت هرس شده برای حل مسئله 4 وزیر به روش عقبگرد



این درخت دو حالت وقتی وزیر اول در خانه های [1,1] و [1,2] قرار می گیرد را بررسی می کند و گره هایی که برچسب x خورده اند یعنی به نتیجه نمی رسند و ادامه آن بی نتیجه است.

بررسی شرایط تهدید در مسئله n وزیر

اولین حالت این است که وزیر جدید در سطر و یا ستونی قرار بگیرد که وزیری قبلا در آن سطر و ستون وجود داشته باشد. فرض کنید یک وزیر در سطر i=3 و ستون k=3 قبلا وجود دارد. حال برای وزیر دیگر می خواهیم سطر j =3 و ستون l=4 را انتخاب کنیم در اینجا چون i=j می باشد یعنی دو وزیر در یک سطر هستند و این شرط تهدید است. همچنین اگر مقدار k=l باشد نیز دو وزیر در یک ستون هستند و این نیز شرط تهدید است و غیر مجاز است.

Row: Queen $Q[i,k]$ conflicts with Queen $Q[j,l] \iff i = j.$

Column: Queen $Q[i,k]$ conflicts with Queen $Q[j,l] \iff k = l.$

حالت دیگر که تهدید فرض می شود این است که دو وزیر بر روی قطر های اصلی قرار بگیرند. فرض کنید وزیری در سطر i=4 و ستون k=2 قرار دارد و می خواهیم وزیر دیگری را در سطر j=8 و ستون l=6 قرار دهیم. چون شرط i-k=j-l بر قرار است این دو وزیر در یک قطر قرار دارند و همدیگر را تهدید می کنند. در زیر تمام حالات تهدید در قطر های اصلی آمده است.

Diagonal

	1	2	3	4	5	6	7	8
1	0	-1	-2	-3	-4	-5	-6	-7
2	1	0	-1	-2	-3	-4	-5	-6
3	2	1	0	-1	-2	-3	-4	-5
4	3	2	1	0	-1	-2	-3	-4
5	4	3	2	1	0	-1	-2	-3
6	5	4	3	2	1	0	-1	-2
7	6	5	4	3	2	1	0	-1
8	7	6	5	4	3	2	1	0

$Q[i,k]$ conflicts with $Q[j,l]$

$$\Updownarrow$$

$$i - k = j - l$$

حالت دیگر که تهدید فرض می شود این است که دو وزیر بر روی قطر های فرعی قرار بگیرند. فرض کنید وزیری در سطر i=5 و ستون k=3 قرار دارد و می خواهیم وزیر دیگری را در سطر j=2 و ستون l=6 قرار دهیم. چون شرط i+k=j+l بر قرار است این دو وزیر در یک قطر قرار دارند و همدیگر را تهدید می کنند. در زیر تمام حالات تهدید در قطر های فرعی آمده است.

Back-Diagonal

	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	9
2	3	4	5	6	7	8	9	10
3	4	5	6	7	8	9	10	11
4	5	6	7	8	9	10	11	12
5	6	7	8	9	10	11	12	13
6	7	8	9	10	11	12	13	14
7	8	9	10	11	12	13	14	15
8	9	10	11	12	13	14	15	16

$Q[i,k]$ conflicts with $Q[j,l]$

$$\Updownarrow$$

$$i+k=j+l$$

تعداد کل گره ها در درخت فضای حالت در مسئله n وزیر

$$1+n+n^2+n^3+\ldots+n^n=\frac{n^{n+1}-1}{n-1} \quad \Rightarrow \quad \frac{8^{8+1}-1}{8-1}=19173961$$

1 گره در سطح صفر، n گره در سطح 1، n^2 گره در سطح 2 و ... و n^n گره در سطح n است.

مسئله حاصل جمع زیر مجموعه ها

در این مسئله قصد داریم همه زیر مجموعه ها از n عدد را بنویسیم که حاصل جمع آنها برابر با مقدار مشخص w باشد. در اینجا داریم n=5 و w=21 و اعداد به صورت زیر آمده اند.

$$w_1 = 5\ , w_2 = 6\ , w_3 = 10\ , w_4 = 11\ , w_5 = 16$$

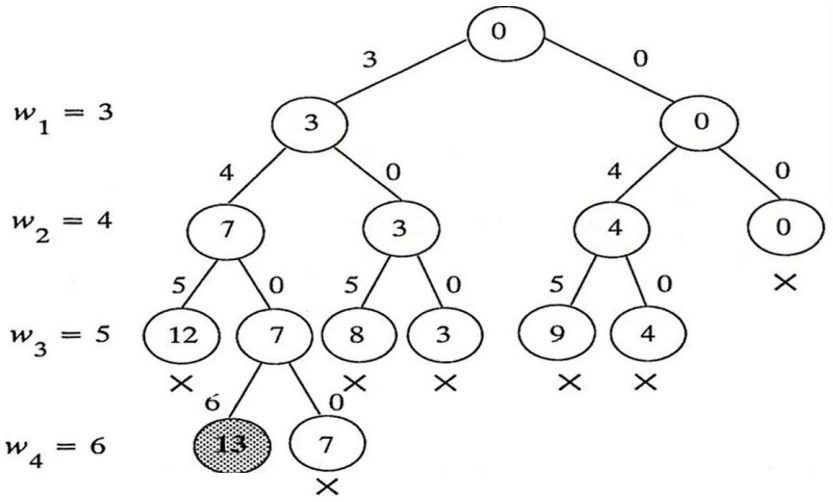
مجموعه جواب زیر را خواهیم داشت:

$$\{w_1, w_5\}, \{w_3, w_4\}, \{w_1, w_2, w_3\}$$

مثال :

اگر در بین n=4 عضو بخواهیم مجموعه هایی بیابیم که حاصل جمع آنها w=13 باشد درخت فضای حالت و ترکیب یا ترکیبهای مناسب را بیابید.

$$w_1 = 5\ , w_2 = 6\ , w_3 = 10\ , w_4 = 11\ , w_5 = 16$$



در این مثال هر بار دو انتخاب داریم که یکی برداشتن عنصر می باشد و دیگری برنداشتن آن، اگر عدد برداشته شود حاصل جمع آن با نتیجه گره بالاتر در گره جاری قرار می گیرد و روی یال نوشته

می شود. در صورتی هم که برداشته نشود گره حاصل بدون تغییر می ماند و روی یال مقدار 0 نوشته می شود و در هر جای مسیر اگر ادامه آن با توجه به مقادیر اعضا بی فایده باشد با بر چسب زدن انتهای آن مسیر مشخص می گردد. در این مثال مشخص است با برداشتن عنصر های $w_1 = 5\ , w_2 = 6\ \ , w_4 = 11$ و بر نداشتن $w_3 = 10$ به جواب می رسیم و بقیه مسیر ها به بن بست خواهد خورد.

$\{w_1, w_2, w_4\}$

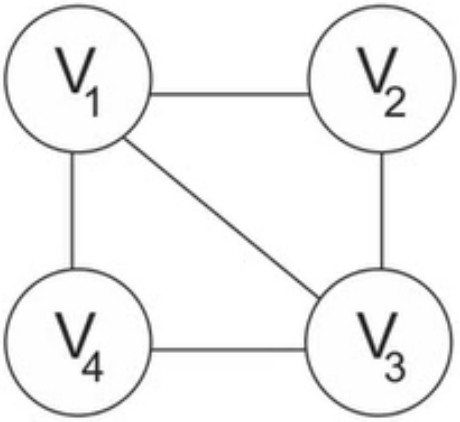
تعداد گره های فضای حالت از فرمول زیر بدست می آید.

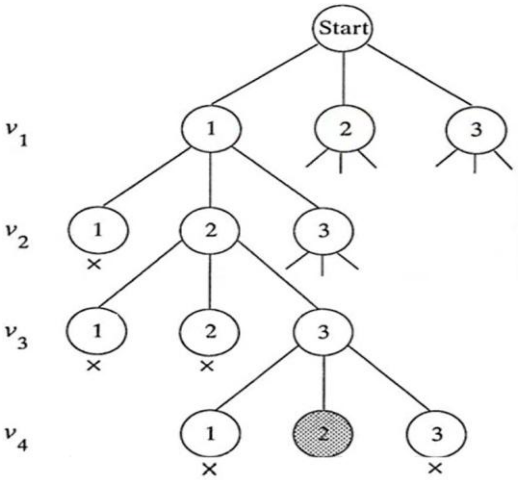
$$1 + 2 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$$

رنگ آمیزی گراف

هدف آن است که با تعداد m رنگ داده شده، راس‌های گراف به گونه‌ای رنگ شوند که هیچ دو راس مجاوری دارای رنگ مشابه نباشند.

مثال با 3 رنگ می خواهیم گراف زیر را رنگ آمیزی کنیم. درخت فضای حالت آن را ترسیم کنید.



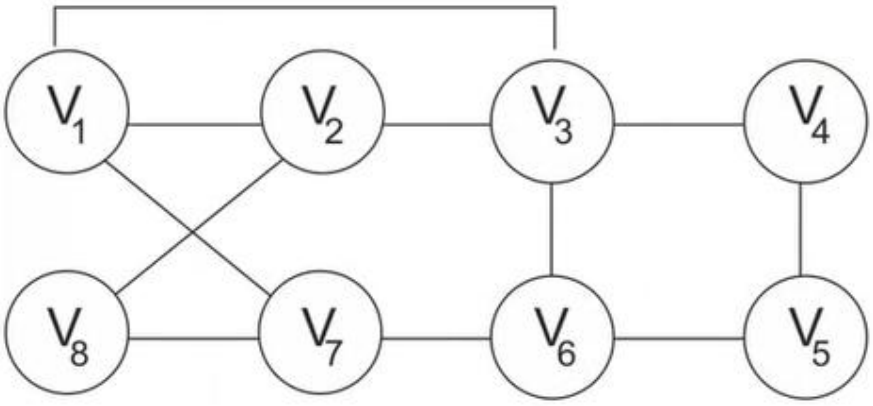


تعداد گره های فضای حالت از فرمول زیر بدست می آید.

$$1 + m + m^2 + \ldots + m^n = \frac{m^{n+1} - 1}{m - 1}$$

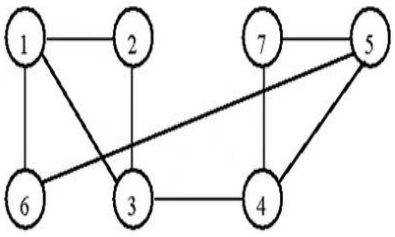
مسئله مدارهای همیلتونی

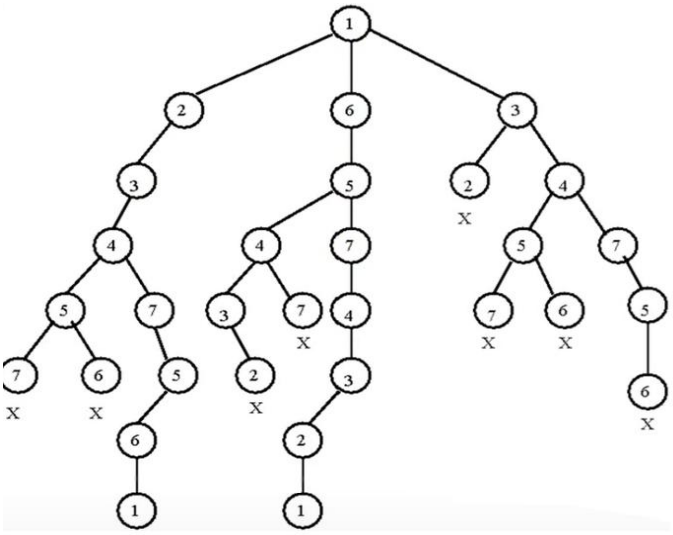
یک گراف همیلتونی گرافی است که دارای یک "دور همیلتونی (Hamiltonian Cycle) "باشد. دور همیلتونی مسیری بسته در گراف است که دقیقا یک بار از هر راس گراف عبور می کند و در نهایت به راس شروع بازمی گردد.



[v1,v2,v8,v7,v6,v5,v4,v3,v1]

مثال:



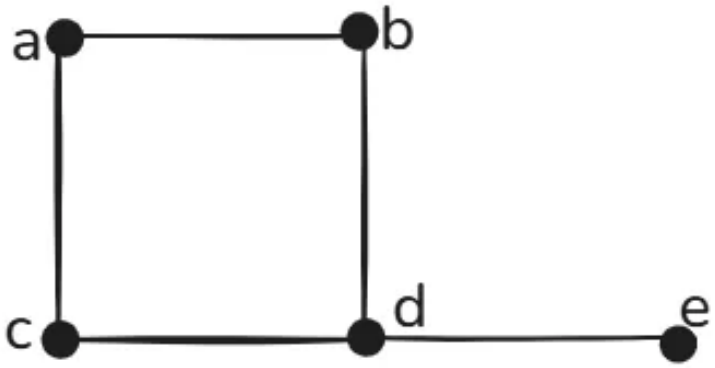


فصل 6

گراف

(Graph)

یک گراف، نمایشی تصویری از مجموعه اشیائی است که با هم ارتباط دارند. هر یک از این اشیا را راس یا گره می‌نامند. راس‌ها نیز از طریق یالها یا لبه‌ها با هم مرتبط هستند. اگر بخواهیم کمی رسمی‌تر بنویسیم، یک گراف را با (V,E) تعریف می‌کنیم که در آن، V مجموعه راس‌ها و E مجموعه یال‌ها است. شکل زیر، یک گراف را نشان می‌دهد. احتمالاً این شکل، تا حدودی شبیه همان چیزی است که تصور کرده‌اید.



در شکل بالا، Vو E به‌صورت زیر هستند:

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

نمایش گراف

برای نماش گراف به دو صورت می شود کمل کردو

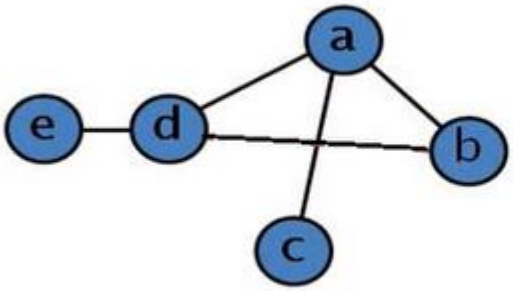
1- ماتریس همجواری[1]

2- لیست همجواری[2]

ماتریس همجواری

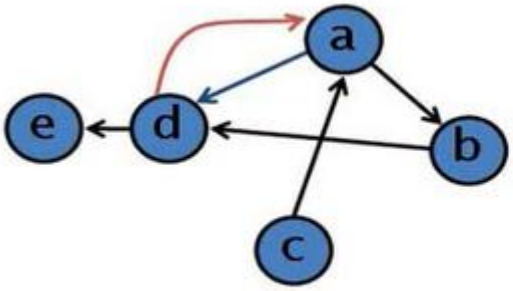
ماتریس همجواری برای گرافهای غیر جهت دار، به منظور مشخص کردن ارتباط بین گره های یک گراف ایجاد می شود و در اصل پیاده سازی گراف در حافظه می باشد. با اینجاد این گراف تمام رابطه های بین گره ها در گراف مشخص می شود. برای تشکیل این ماتریس، ماتریسی با تعداد سطر و ستون برابر و به تعداد گره های گراف ایجاد می شود. اندیس های این آرایه مقدار 0 یا 1 را خواهند داشت. سطر اول این ماتریس مربوط به گره a می باشد که مشخص می کند که گره a با چه گره هایی در ارتباط است. ستون اول ارتباط a با خودش می باشد، چون یالی از a به خودش وجود ندارد مقدار 0 در درایه 0,0 قرار می گیرد. در ستون بعد ارتباط بین a و b مشخص می شود. چون مسیر در بین این دو گره وجود دارد پس درایه 0,1 برابر 1 می شود. ارتباط بین a و c و d نیز وجود دارد پس مقدار درایه های 0,2 و 0,3 نیز 1 می شود. ولی ارتباطی بین a و e وجود ندارد پس درایه مربوط به آن برابر 0 قرار میگیرد. با این عمل سطر اول به پایان می رسد و همین اعمال در سطر های بعدی و برای گره های دیگر انجام می شود تا ماتریس کامل گردد.

[1] Adjacency Matrices
[2] Adjacency List



	a	b	c	d	e
a	0	1	1	1	0
b	1	0	0	1	0
c	1	0	0	0	0
d	1	1	0	0	1
e	0	0	0	1	0

در تشکیل ماتریس همجواری برای گرافهای جهت دار دقیقا مانند بالا عمل می شود، فقط باید دقت شود که یالهای جهت دار ارتباط را محدود می کنند. مثلا در گراف زیر، گره a به d راه دارد ولی برعکس آن وجود ندارد. این مهم در ترسیم ماتریس همجواری باید مدنظر قرار گیرد.



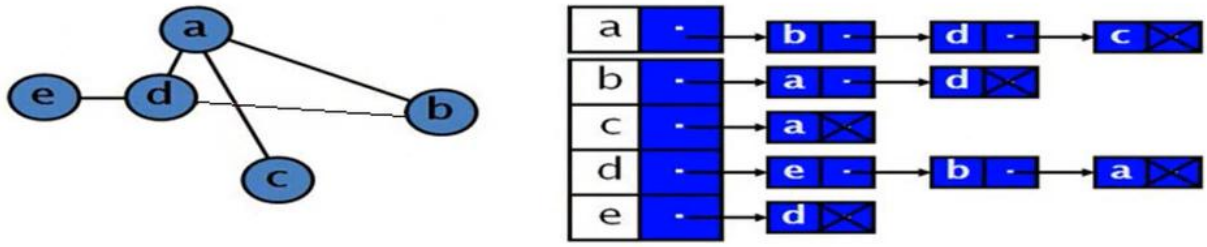
	a	b	c	d	e
a	0	1	0	1	0
b	0	0	0	1	0
c	1	0	0	0	0
d	1	0	0	0	1
e	0	0	0	0	0

لیست همجواری

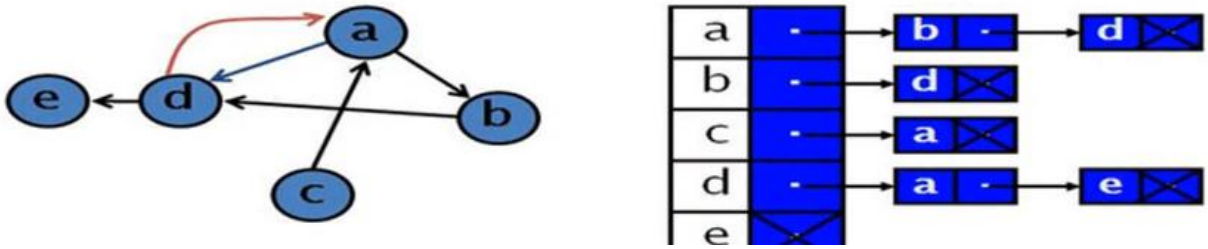
برای تشکیل لیست همجواری از لیست های پیوندی استفاده می شود. روش کار به اینصورت است که برای هر گره با کمک لیست های پیوندی تمام گره های همجوار مشخص می گردد. به عنوان مثال، برای راس a لیست پیوندی ایجاد می شود، که در داده لیست نام گره(a) قرار میگیرد و قسمت اشاره گر آن به گره دیگری اشاره می کند که گره مجاور گره a می باشد(مثلا گره b).

در گره مربوط به b و در قسمت اشاره گر آن به گره دیگر مجاور a یعنی d اشاره می شود و این فرایند ادامه پیدا می کند تا تمام گره های مجاور a در لیست قرار بگیرد.

این فرایند برای تمام گره ها و در لیست های پیوندی مجزا تکرار می شود.

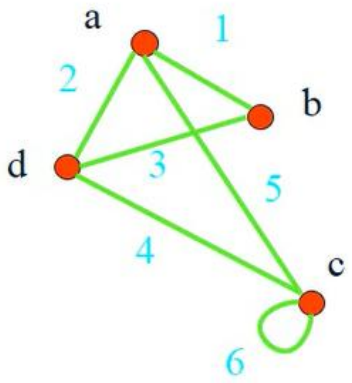


در تشکیل لیست همجواری برای گرافهای جهت دار دقیقا مانند بالا عمل می شود، فقط باید دقت شود که یالهای جهت دار ارتباط را محدود می کنند. مثلا در گراف زیر، گره a به d راه دارد ولی برعکس آن وجود ندارد. این مهم در ترسیم ماتریس همجواری باید مدنظر قرار گیرد.



ماتریس برخورد

ماتریس برخورد، ماتریسی برای یک گراف بدون جهت با |v| سطر و |E| ستون می باشد. اگر یال i گره j را لمس کند، آنگاه درایه واقع در سطر i و ستون j برابر 1 می شود.



$$M = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

پیمایش گراف

گراف ها را می توان با دو روش زیر پیمایش کرد.

1- سطحی (BFS: Breadth First Search)

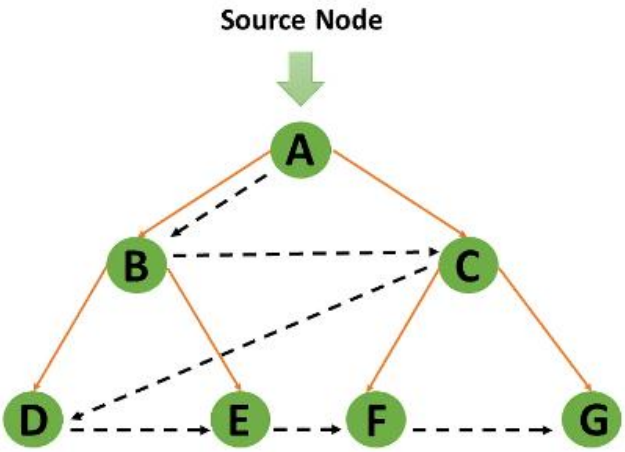
2- عمقی (DFS: Depth First Search)

پیمایش سطحی (BFS: Breadth First Search)

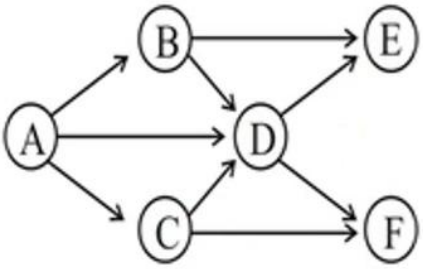
الگوریتم BFS ، جزو پرکاربردترین الگوریتم‌ها به‌حساب می یادBFS ، یک رویکرد پیمایش گراف و درخت می باشد که می‌توانیم ابتدا الگوریتم رو با گره مبدأ (منبع) آغاز کنیم و بعدش لایه به لایه، گره‌هایی رو که مستقیماً با گره مبدأ ارتباط دارن، تحلیل کنیم. در مرحله بعد پیمایشBFS ، باید گره‌های مجاور سطح بعد رو مشاهده کند.

بر اساسBFS ، شما باید گراف و با درخت را در جهت عرضی پیمایش کنین:

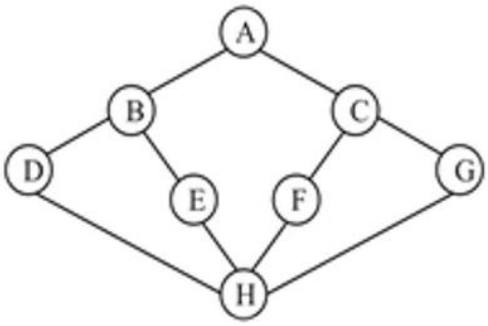
- برای شروع، به‌صورت افقی حرکت می کنیم و از کل گره‌های لایه موجود رو بازدید می کنیم
- سپس به لایه بعدی حرکت می کنیم.



مثال :



A B C D E F



A B C D E F G H

جستجوی اول سطح از ساختار داده صف برای نگهداری گره و بازدید شدن آن تا زمان بازدید از کل رئوس مجاور که ارتباط مستقیمی با اون گره دارن، استفاده می‌کنه. صف هم بر اساس اصل فایفو شکل می‌گیره و اولین گره نمایش داده شده، اون گرهی هست که اول از همه، قرار داده شده. پیمایش الگوریتم BFS در مورد گره‌ها به دو شکل انجام می‌شه:

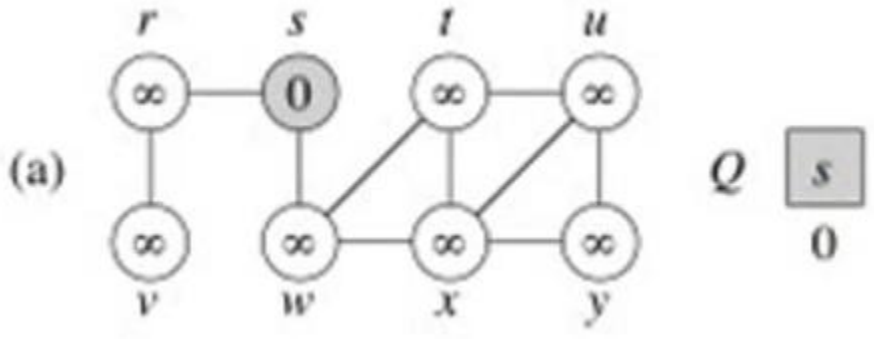
- گره بازدید شده
- گره بازدید نشده.

ترتیب مراحل اجرای الگوریتم جستجوی اول سطح

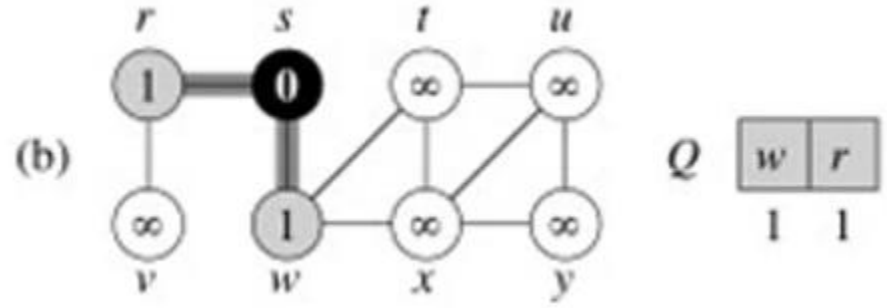
- شروع با گره مبدأ (منبع)
- اضافه کردن گره در جلوی صف نسبت به فهرست بازدید شده
- ایجاد فهرستی از گره‌های بازدید شده که در فاصله نزدیکی به آن رأس قرار دارند .
- خارج کردن گره‌های بازدید شده از صف
- تکرار مراحل تا زمان خالی شدن صف

مثال :

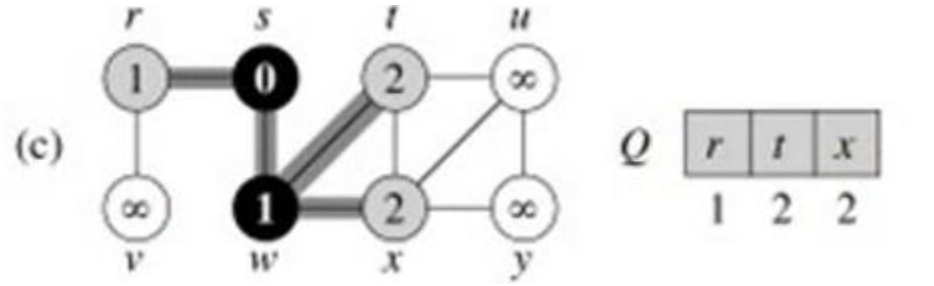
در مثال زیر s گره ریشه فرض می شود و در شروع با رنگ طوسی مشخص شده است و داخل صف Q قرار میگیرد. رنگ طوسی نشان می دهد که گره در صف می باشد و هنوز ملاقات(خوانده) نشده است. یک اندیس در صف در نظر گرفته شده است که با مراحل بالا این اندیس به عنوان برچسب ریشه انتخاب شده است که می تواند سطح گره را مشخص کند. در بقیه گره ها می بینید که برچسب گره ها نامشخص می باشد که به مرور اصلاح می شود.



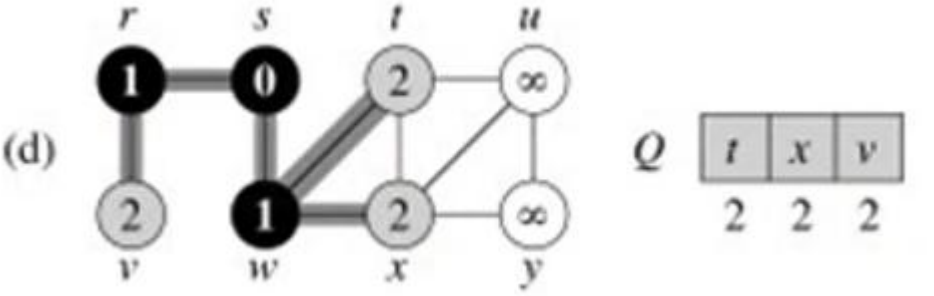
با ملاقات s و خارج شدن آن از صف رنگ گره مشکلی می شود و گره های همجوار آن(r,W) با برچسب جدید نشانه گذاری می شوند، رنگ جدید می گیرند و داخل صف قرار می گیرند. در اینجا فرض می شود که گره w در جلوی صف قرار گرفته است.



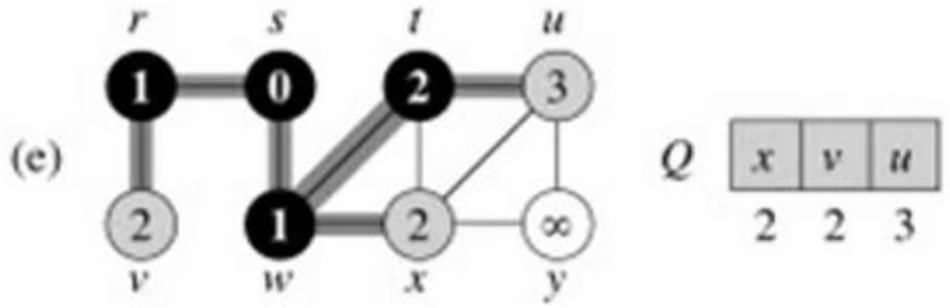
در مرحله بعد، چون w در ابتدای صف بود، ملاقات شده(رنگ آن مشکی می شود و از ابتدای صف خارج می شود) و گره های مجاور آن (t,x) با مقدار جدید در صف قرار میگیرند.



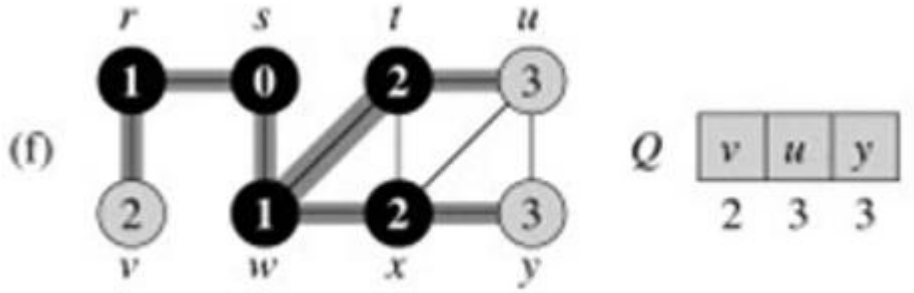
در گام بعد، r از ابتدای صف خارج شده و ملاقات می شود(رنگ آن مشکی می شود و از ابتدای صف خارج می شود). در نتیجه گره هم جوار آن(v) با برچسب جدید به انتهای صف منتقل می گردد.



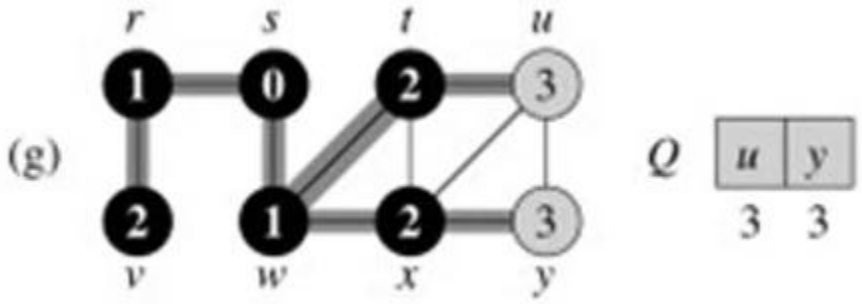
در این مرحله نوبت گره t می باشد که ملاقات شود. در نتیجه گره همجوار با آن(u) با برچسب جدید در انتهای صف قرار می گیرد.



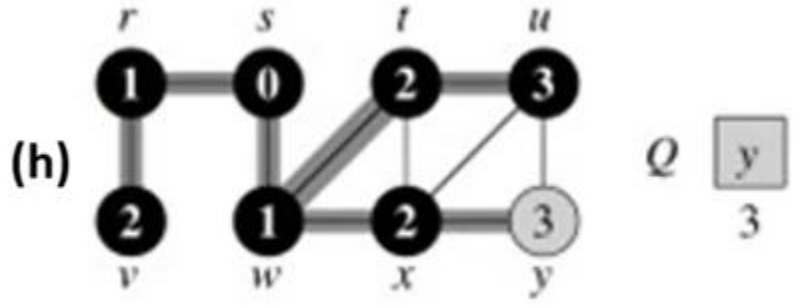
در مرحله بعد، نوبت گره x می باشد که ملاقات شود. در نتیجه گره همجوار با آن(y) با برچسب جدید در انتهای صف قرار می گیرد.



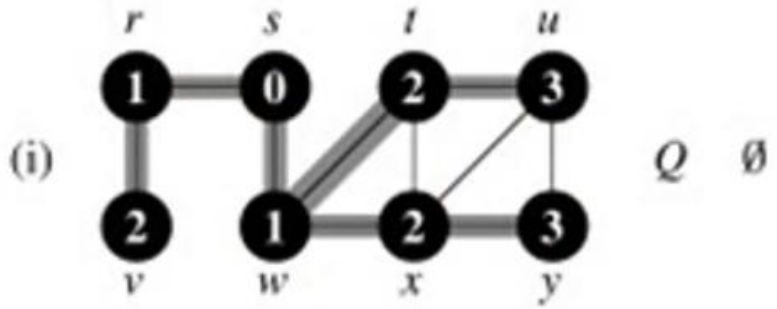
در گام بعد، نوبت گره v میرسد تا ملاقات شود و چون گره همجوار ندارد، هیچ گره ای به صف اضافه نمی شود.



در گام ماقبل آخر، نوبت گره u می باشد تا ملاقات شود و بدلیل اینکه گره همجوار آن نیز در صف می باشد و یا ملاقات شده اند، گره ای به صف وارد نمی شود.

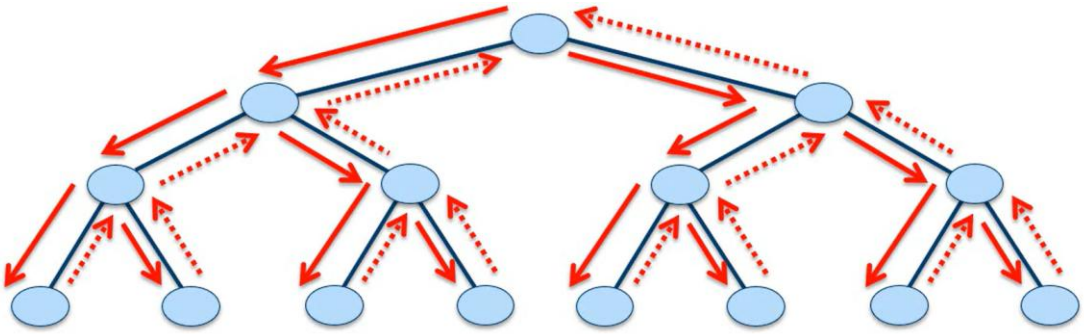


در نهایت تنها گره موجود در صف (y) ملاقات شده و با تهی شدن صف مراحل پیمایش به اتمام می رسد.

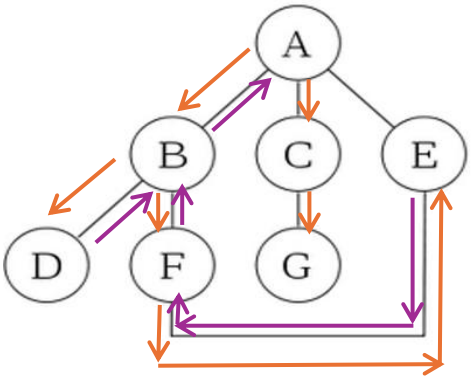


پیمایش عمقی (DFS: Depth First Search)

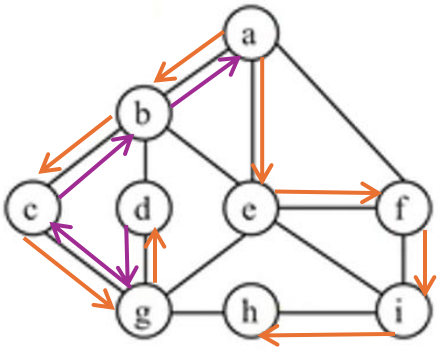
در این نوع از الگوریتم‌ها عمل پیمایش را از گره اول شروع می‌کنیم. در ابتدا قبل از عقب نشینی از هر شاخه، همه گره‌های آن شاخه را به ترتیب تا عمیق‌ترین سطح ممکن بازدید می‌کنیم. سپس گره‌های همه شاخه‌های دیگر نیز طبق همین روش باید بازدید شوند.



مثال:



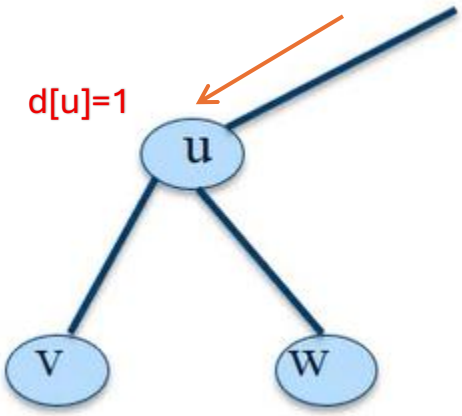
A B D F E C G



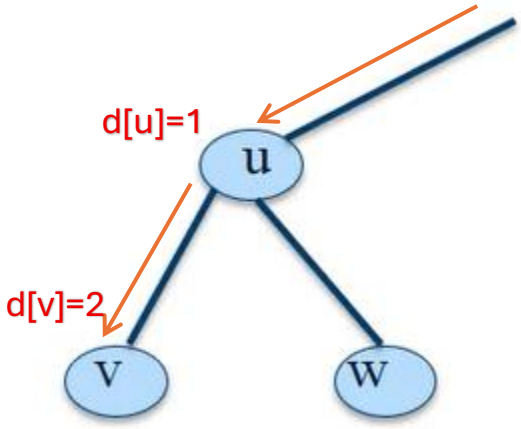
a b c g d e f i h

تعیین زمان کشف و انتظار

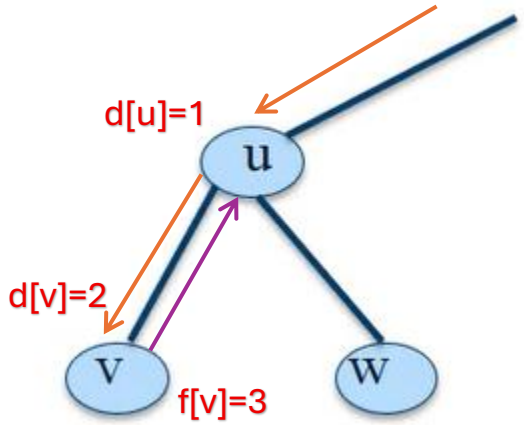
به منظور تعیین این دو زمان دو متغیر d[u] برای ملاقات یا کشف گره u و f[u] برای ترک آن در نظر می گیریم. توجه شود که این دو متغیر برای همه گره ها در نظر گرفته می شود.



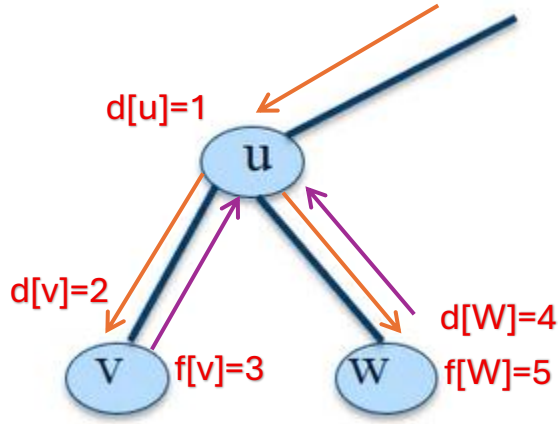
در مرحله اول وقتی گره u ملاقات می شود مقدار d[u] برابر 1 می شود که زمان اول می باشد.



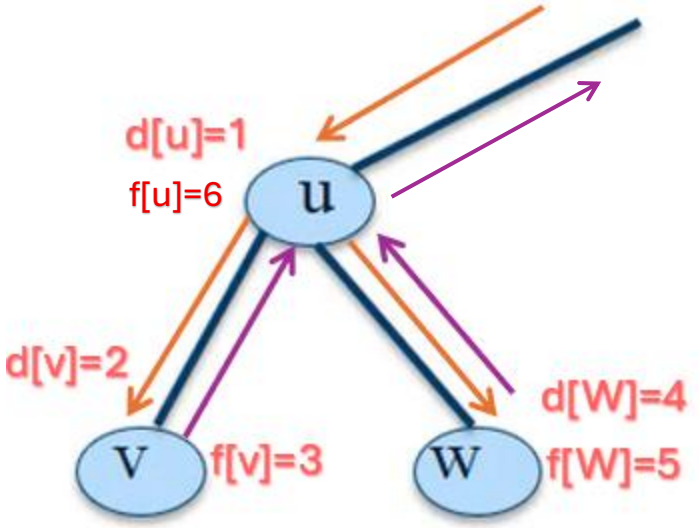
در مرحله بعد گره v ملاقات می شود و مقدار d[v] برابر 2 می شود که منظور در زمان 2 می باشد. پس تا اینجا گره u در زمان 1 و گره v در زمان 2 ملاقات شده است.



در این مرحله چون v گره همجواری ندارد ترک می شود و مقدار f[v] برابر 3 قرار می گیرد.



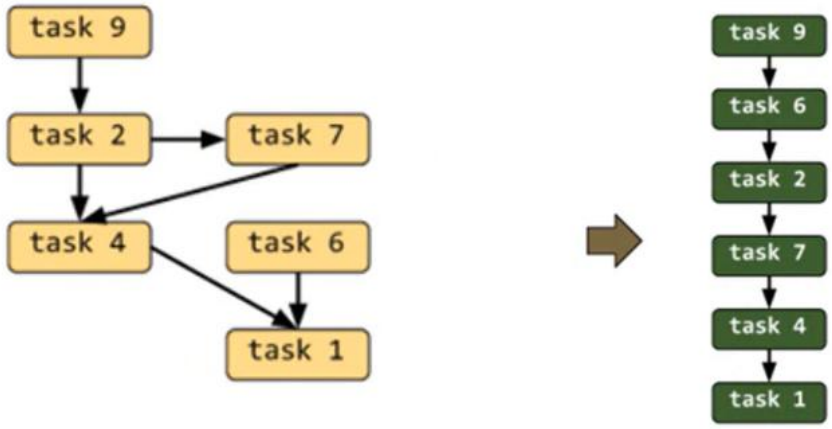
در مرحله بعد گره w ملاقات می شود و مقدار d[w] برابر4 می شود. در این مرحله چون W گره همجواری ندارد ترک می شود و مقدار f[W] برابر 5 قرار می گیرد و به گره u باز می گردیم.



در مرحله آخر چون همه گره های مجاور u بازدید شده و به پایان رسیده اند، u را نیز ترک می کنیم و f[u] را برابر 6 قرار می دهیم. مشاهده می شود که برای تمام گره ها زمان ملاقات و اتمام محاسبه شده است.

مرتب سازی موضعی[1]

یک مرتب‌سازی موضعی یا ترتیب موضعی یک گراف بدون دور جهت دار[2]، یک ترتیب خطی از همه رئوس آن است به‌طوری‌که هر گره قبل از همه گره‌هایی می‌آید که از آن به آن‌ها یال خارج شده‌است. هر درخت بدون دور جهت دار یک یا چند مرتب‌سازی موضعی دارد. کاربرد اصلی مرتب‌سازی موضعی (ترتیب موضعی) در زمانبندی یک سلسله‌ای از کارها یا وظایف است.

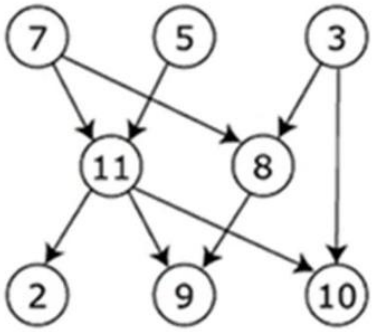


در شکل بالا که یک گراف از کارهایی که باید انجام شود داریم. کار های 9 و 6 که یال ورودی ندارند در خروجی قرار می گیرند که ترتیب آنها اهمیت ندارد. سپس چون کار 9 انجام شده است می توانیم کار 2 را در خروجی بیاوریم. حال نوبت به کار 7 می رسد که به خروجی برود. حال کار 4 می تواند در گراف ترتیب مد نظر اضافه شود و در نهایت به کار 1 می رسیم و در لیست قرار می دهیم.

در ترتیب سمت چپ کارها طوری مرتب شده اند که کار های وابسته به دیگران بعد آنها آمده است. به عنوان مثال کار 4 لازم است که بعد انجام کار 2 و 7 انجام شود. مشاهده می شود که در ترتیب کارها این مهم اتفاق افتاده و کار 4 بعد کارهای 2 و 7 آمده است.

[1] Topological Sort
[2] Directed Acyclic Graph(DAG)

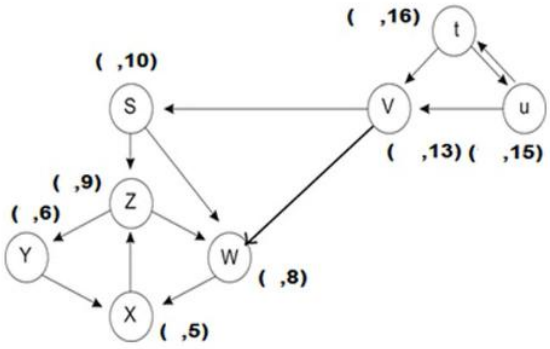
یک راه برای مرتب سازی توپولوژیکی این است که گره هایی که یال ورودی ندارند را ملاقات کنیم. پس از ملاقات آنها گره یال های خارج شده از آن را حذف کنیم و این کار را ادامه دهیم تا تمام گره ها ملاقات شوند. در تصویر زیر حالت های مختلف گراف سمت چپ را در سمت راست نشان داده شده است.

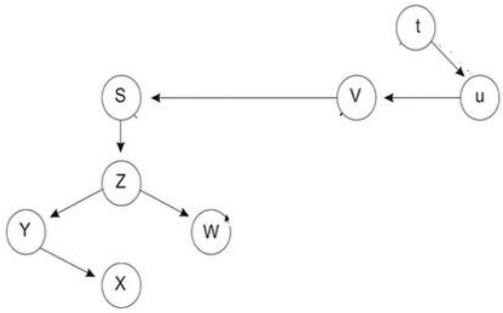


7, 5, 3, 11, 8, 2, 9, 10
3, 5, 7, 8, 11, 2, 9, 10
5, 7, 3, 8, 11, 10, 9, 2
7, 5, 11, 3, 10, 8, 9, 2
7, 5, 11, 2, 3, 8, 9, 10
3, 7, 8, 5, 11, 10, 2, 9

مرتب سازی توپولوژیکی بر اساس زمان خاتمه

گراف را به صورت عمقی پیمایش می کنیم و زمان خاتمه گره ها را بدست می آوریم. در مرحله بعد گره ها را به صورت نزولی بر اساس زمان خاتمه ملاقات می کنیم.





t u v s z w y x

مسئله کوتاه ترین مسیر[1]

مسئله یافتن کوتاه‌ترین مسیر در واقع مسئله یافتن مسیری بین دو رأس (یا گره) است به گونه‌ای که مجموع وزن یال‌های تشکیل دهنده آن کمینه شود. برای مثال می‌توان مسئله یافتن سریع‌ترین راه برای رفتن از یک مکان به مکان دیگر روی نقشه را، در نظر گرفت؛ در این حالت رأس‌ها نشان دهنده مکان‌ها و یال‌ها نشان دهنده بخش‌های مسیر هستند که برحسب زمانِ لازم برای طی کردن آن‌ها وزن گذاری شده‌اند.

اگر مسیری مانند p داشته باشیم:

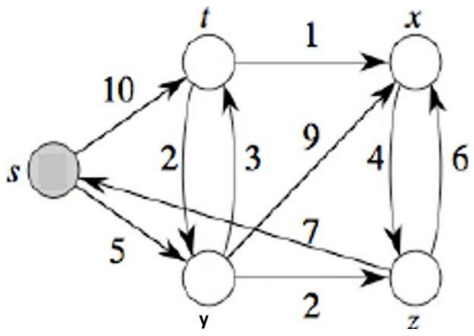
$$p = \{v_0, v_2 \dots v_k\}$$

آنگاه برای بدست آوردن وزن این مسیر داریم

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, w_i)$$

برای محاسبه کوتاهترین مسیر بین دو گره لازم است وزن کلیه مسیرهای ممکن بین آن دو گره را محاسبه کرد و کمترین وزن مسیر در این بین کوتاه ترین مسیر بین آن دو گره می باشد.

$$\delta(u,v) = \begin{cases} \min\{w(p): u \to v\} & if\ there\ is\ a\ path\ from\ u\ to\ v \\ \infty & otherwise \end{cases}$$



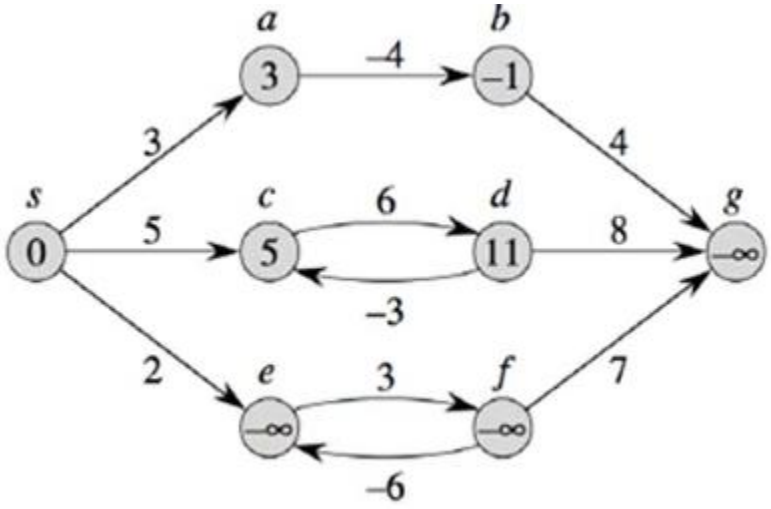
$p_1 = \{s, t, x\}$ $w(p1) = 10 + 1 = 11$
$p_2 = \{s, y, z, x\}$ $w(p_2) = 5 + 2 + 6 = 13$
$p_3 = \{s, y, x\}$ $w(p_3) = 5 + 9 = 14$

$$\delta(s, x) = W(p_1) = 11$$

[1] Shortest Paths

یال با وزن منفی

در گرافی بصورت $G = (V, E)$ اگر چرخه ای با وزن منفی از گره شروع تا تمام گره های موجود در مسیر تا مقصد وجود نداشته باشد، کوتاه ترین مسیر را خوش تعریف[1] توصیف می کنیم. در صورتی که این چرخه منفی موجود باشد خوش تعریف نمی باشد و $\delta(s, v) = \infty$ می شود.



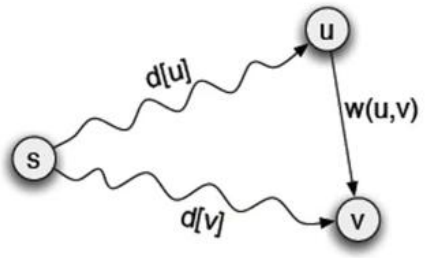
باید توجه داشته باشیم که کوتاه ترین مسیر دارای سیکل نمی باشد.

[1] Well defined

آرامش(Relaxation)

این اصطلاح به این معنی می باشد که، اگر مسیری داشته باشیم از s به v (d[v])، و وزن این گره از وزن مسیر s به u و u به v بیشتر باشد، آنگاه مسیر با وزن کمتر جایگزین مسیر پر هزینه تر می شود و تسریع می شود که گره پدر v ($\pi[v]$) برابر با u می شود.

```
RELAX(u, v, w)
1   if d[v] > d[u] + w(u, v)
2       then d[v] ← d[u] + w(u, v)
3            π[v] ← u
```



دو رویکرد در این مسئله وجود دارد:

1- رویکرد یک گره به مقصد های مختلف

- Dijkstra
- Belman-Ford

2- رویکرد همه گره ها نسبت به هم

- Matrix Multiplication
- Floyd-Marshall

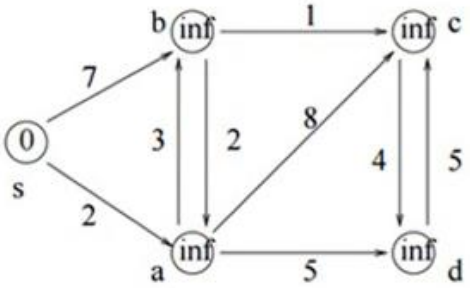
الگوریتم Dijkstra

الگوریتم دیکسترا **(Dijkstra's Algorithm)** یا اولین الگوریتم کوتاهترین مسیر دیکسترا (Dijkstra's Shortest Path First) از جمله الگوریتم‌هایی است که برای پیدا کردن کوتاهترین مسیر از گره آغازین تا گره هدف در **گراف** وزن دار (گرافی هست که یال‌های آن دارای وزن هستند) استفاده می‌شود.

الگوریتم دیکسترا یا دایجسترا الگوریتمی برای پیدا کردن کوتاهترین مسیر بین گره مبدا و دیگر گره ها در یک گراف است که البته از آن برای پیدا کردن کوتاهترین مسیر بین دو گره هم استفاده می‌شود. در ادامه، گام‌های مورد استفاده در الگوریتم دایجسترا به منظور یافتن کوتاه‌ترین مسیر از یک راس مبدا مجرد به دیگر راس‌ها در گراف داده شده به صورت مشروح بیان شده‌اند.

روش کار الگوریتم دایجسترا

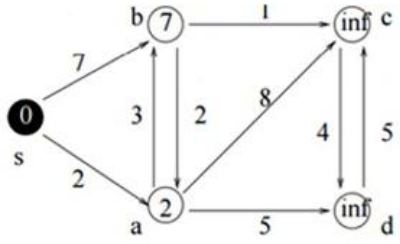
برای شروع در گراف گره ریشه را s قرار می دهیم و مقدار 0 در آن قرار می دهیم. همچنین مابقی گره های گراف با مقدار inf یا null برچسب گذاری می شود. سپس یک صف اولویت ایجاد می کنیم. در این صف گره ها و وزن آنها قرار می گیرند که در مرحله نخست بخیر از گره s همه inf می باشند و گره شروع نیز 0 می باشد.



Priority Queue:

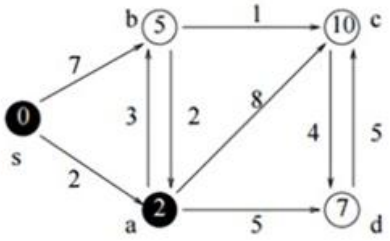
v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞

برای شروع الگوریتم، کوچکترین مقدار وزن در صف خوانده می شود که در این مرحله 0 می باشد و از صف خارج می شود و رنگ گره مشکی می شود که نشانه ملاقات گره می باشد. سپس گره های همجوار با وزن یالهای خود برچسب می خورند و داخل صف و زیر نام خود قرار می گیرند.



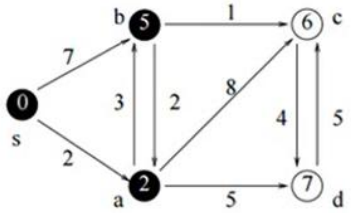
v	a	b	c	d
$d[v]$	2	7	∞	∞

در گام بعد، باز کمترین وزن انتخاب می شود و ار صف خارج می شود و رنگ آن گره تغییر می کند. با این کار مسیر های جدید باز می شود. همچنین مقدار گره b نیز از 7 به 5 بروز می گردد زیرا مسیر رسیدن به گره b از a کم هزینه تر می باشد. با باز شدن مسیر های جدید بر چسب های گره های مجاور بروز رسانی می گردد.



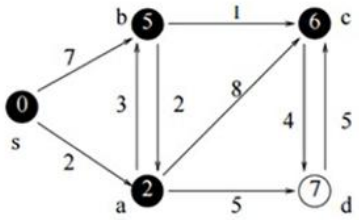
v	b	c	d
$d[v]$	5	10	7

در گام بعد، باز کمترین مقدار انتخاب شده، رنگ گره تغییر می کند و با موقعیت جدید مقدار گره c بروز رسانی می شود.



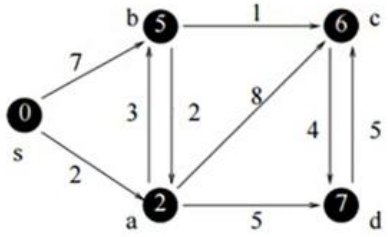
v	c	d
d[v]	6	7

از آنجایی که مسیر جدیدی در این مرحله باز نشده است، در صف کوچکترین مقدار خارج می شود و گره مربوط به آن (c) ملاقات می شود.



v	d
$d[v]$	7

در آخرین مرحله، آخرین مقدار خارج می شود، گره مربوط به آن ملاقات می شود و صف تهی می گردد که این نشانه پایان کار می باشد.



Priority Queue: $Q = \emptyset$.

مشاهده می شود که در گراف ایجاد شده برچسب هر گره هزینه آن گره از ریشه می باشد و هم هزینه ترین مسیر برای این گره برای گره های مختلف محاسبه شده است. در زیر الگوریتم این رفتار به منظور بهره برداری آمد است. شایان ذکر است مرتبه اجرائی این الگوریتم حریصانه $\boldsymbol{\theta(n^2)}$ می باشد.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5       do u ← EXTRACT-MIN(Q)
6          S ← S ∪ {u}
7          for each vertex v ∈ Adj[u]
8               do RELAX(u, v, w)
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ V[G]
2       do d[v] ← ∞
3          π[v] ← NIL
4  d[s] ← 0
```

الگوریتم Belman-Ford

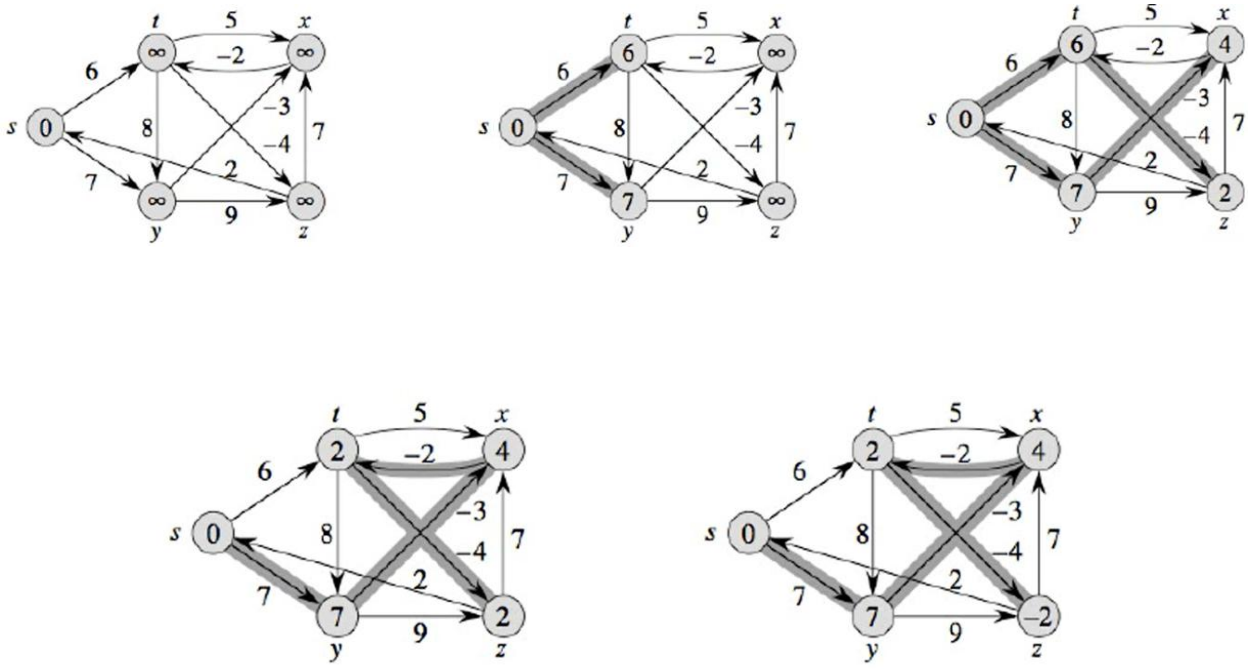
این الگوریتم به ما کمک می کند تا کوتاه ترین مسیر رو در گراف های وزن دار و جهت دار پیدا کنیم. نکته پر اهمیت این است که وقتی با حلقه های منفی مواجه می شود، به خوبی عمل کرده و حلقه منفی را شناسائی می کند.

الگوریتم Bellman-Ford یک راهکار کارآمد برای پیدا کردن کوتاه ترین مسیر در گراف های وزن دار به حساب می آید. این الگوریتم به ویژه زمانی که گراف شامل لبه های با وزن منفی باشد، خیلی کاربردی تر عمل می کنند. برعکس الگوریتم Dijkstra که فقط برای گراف های بدون لبه های منفی مناسب است، Bellman-Ford توانایی شناسایی این نوع لبه ها رو دارد و می تواند حلقه های منفی رو هم تشخیص دهد.

روش کار این الگوریتم بدین صورت هست که برای هر راس در گراف، فاصله آن تا راس مبدا رو محاسبه می کند. در ابتدای کار، فاصله راس مبدا صفر در نظر گرفته می شود و فاصله سایر راس ها بی نهایت فرض می شوند. سپس، الگوریتم به تعداد راس ها منهای یک بار، تمامی لبه های گراف رو بررسی می کند و با استفاده از روابط بازگشتی، فاصله بهینه رو به روز رسانی می نماید. این پروسه تا وقتی که دیگه هیچ تغییری در فاصله ها ایجاد نشود ادامه پیدا می کند.

مراحل اجرای الگوریتم Belman-Ford

تفاوت این الگوریتم با الگوریتم دایجسترا این است که این الگوریتم به جای توجه به گره ها بر روی یال ها تمرکز دارد و در صف یال های با وزن کمتر را قرار می دهد و در آن صف یال با کمترین هزینه را انتخاب می کند.



در زیر الگوریتم مربوطه ارائه می گردد.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i ← 1 to |V[G]| − 1
3       do for each edge (u, v) ∈ E[G]
4            do RELAX(u, v, w)
5  for each edge (u, v) ∈ E[G]
6       do if d[v] > d[u] + w(u, v)
7            then return FALSE
8  return TRUE
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ V[G]
2       do d[v] ← ∞
3          π[v] ← NIL
4  d[s] ← 0
```

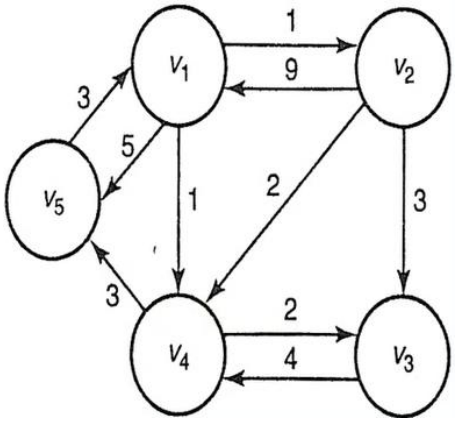
پیچیدگی اجرائی این الگوریتم $O(|V| * |E|)$ می باشد.

الگوریتم فلوید

این الگوریتم برای بدست آوردن کوتاه ترین مسیر بین دو گره استفاده می شود. برای این کار یک ماتریس n*n تشکیل می دهیم و نام آن را $D^{(0)}$ می گذاریم. این ماتریس همان ماتریس همجواری گراف می باشد. سپس با استفاده از رابطه زیر $D^{(1)}$ تا $D^{(n)}$ را محاسبه می کنیم که در نهایت $D^{(n)}$ پاسخ می باشد. همچنین k تعداد گره های گراف می باشد.

$$\boldsymbol{D^{(k)}[i][j] = \min\left(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j]\right)}$$

مثال: تعیین کوتاه ترین مسیر بین گره ها در گراف زیر



با توجه به این گراف ماتریس همجواری را به صورت زیر تشکیل می دهیم. توجه کنید که منظور از درایه های با مقدار ∞ این است که مسیر مستقیمی بین آن دو گره وجود ندارد.

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

حال مراحل لازم برای محاسبه کوتاه ترین مسیر بین دو گره 2 و 5 به شکل زیر می باشد.

$$\boldsymbol{D^{(k)}[i][j] = \min\left(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j]\right)}$$

$$\mathbf{D^{(1)}[2][5] = \min(D^{(0)}[2][5], D^{(0)}[2][1] + D^{(0)}[1][5]) = \min(\infty, 9+5) = 14}$$

$$\mathbf{D^{(2)}[2][5] = \min(D^{(1)}[2][5], D^{(1)}[2][2] + D^{(1)}[2][5]) = \min(14, 0+14) = 14}$$

$$\mathbf{D^{(3)}[2][5] = \min(D^{(2)}[2][5], D^{(2)}[2][3] + D^{(2)}[3][5]) = \min(14, 3+\infty) = 14}$$

$$\mathbf{D^{(4)}[2][5] = \min(D^{(3)}[2][5], D^{(3)}[2][4] + D^{(3)}[4][5]) = \min(14, 2+3) = 5}$$

$$\mathbf{D^{(5)}[2][5] = \min(D^{(4)}[2][5], D^{(4)}[2][5] + D^{(4)}[5][5]) = \min(5, 5+0) = 5}$$

در نهایت مشاهده می شود که کوتاه ترین مسیر بین دو گره 2 و 5 برابر 5 می شود. این مراحل می بایست برای دو زوج گره محاسبه شود و در نتیجه ماتریس نهایی جواب مورد انتظار می باشد.

```
floyd ( n , W[ ] [ ] , D[ ] [ ] , P[ ] [ ] ){
   for( i=1; i<= n ; i++)
      for( j=1; j<= n ; j++)
            P[i][j]=0;
    D=W;
   for( k=1; k<= n ; k++)
      for( i=1; i<= n ; i++)
         for( j=1; j<=n ; j++)
           if (  D[i][k] + D[k][j] < D[i][j] )
           {
                P[i][j]=k;
                D[i][i]=D[i][k]+D[k][j];
            }
}
```

$$T(n) \in \theta(n^3)$$

الگوریتم Matrix Multiplication

این الگوریتم برای بدست آوردن کوتاه ترین مسیر بین همه جفت گره های موجود در گراف مورد استفاده قرار می گیرد. برای محاسبه هزینه کوتاه ترین مسیر بین دو گره i و j در که در این مسیر m یال وجود دارد. در این صورت داریم

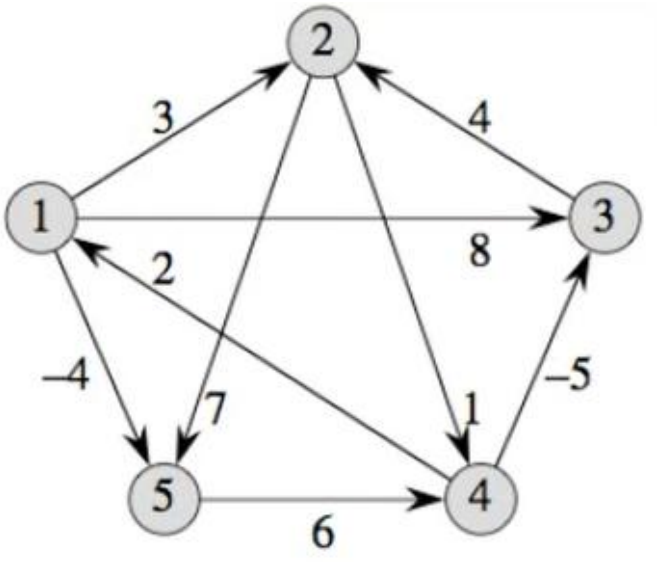
اگر m=0 باشد:

$$l_{i,j}^{(0)} = \begin{cases} 0 & if\ i = j \\ \infty & if\ i \neq j \end{cases}$$

و در صورتی که m>0 رابطه زیر برای محاسبه استفاده می شود.

$$l_{ij}^{(m)} = \min_{1 \le k \le n} \{ l_{ik}^{(m-1)} + w_{kj} \}$$

مثال: مطلوب است یافتن کوتاه ترین مسیر بین همه جفت گره های ماتریس زیر



برای شروع باید ماتریس وزن این گراف را ترسیم کنیم.

$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

در این ماتریس درایه ها نشان دهنده هزینه بین گره ها می باشند. مثلا مقدار موجود در سطر اول و ستون سوم هزینه مسیر بین گره اول و گره سوم می باشد.

مقادیر ∞ در گراف یعنی بین آن دو گره مسیر مستقیم وجود ندارد. مثلا در سطر 3 و ستون 4 این علامت وجود دارد که نشان می دهد از گره 3 به 4 مسیر مستقیم وجود ندارد. برای ادامه کار باید ماتریس زیر را از روی ماتریس بالا بدست آوریم

$$L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

نحوه بدست آوردن این ماتریس شبیه ضرب ماتریس ها می باشد.

$$L^{(2)} = L^{(1)} * L^{(1)}$$

برای بدست آوردن مقدار 2 در سطر اول و ستون 3 که در شکل مشخص شده است به صورت زیر عمل می کنیم.

$$\begin{pmatrix} 0 & 3 & 8 & \infty & -4 \end{pmatrix} \begin{pmatrix} \infty \\ 1 \\ \infty \\ 0 \\ 6 \end{pmatrix} \Rightarrow \begin{pmatrix} \infty \\ 4 \\ \infty \\ \infty \\ 2 \end{pmatrix}$$

سطر اول و ستون 4 ماتریس $L^{(1)}$ باید در محاسبه شرکت کنند و بجای ضرب عناصر آنها را جمع می کنیم و در خروجی می نویسیم. در ماتریس خروجی به جای اینکه مقادیر را جمع کنیم کافیست

از آن مجموعه min بگیریم که همان مقدار 2 می شود. برای تمام درایه ها باید این مراحل را انجام داد تا ماتریس $L^{(2)}$ کامل شود.

حال باید با $L^{(1)} * L^{(2)}$ سراغ ساختن $L^{(3)}$ برویم.

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

برای همچنین باید با $L^{(1)} * L^{(3)}$ سراغ ساختن $L^{(4)}$ برویم.

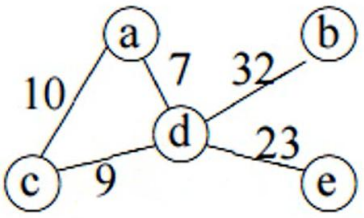
$$L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

چون تعداد گره ها 5 می باشد تا $L^{(4)}$ ادامه می دهیم. که این ماتریس جواب مورد نظر ما می باشد. مرتبه اجرائی این الگوریتم $O(n^3 logn)$

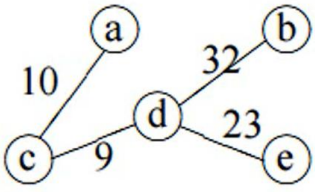
```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
1  n ← rows[W]
2  L(1) ← W
3  m ← 1
4  while m < n − 1
5       do L(2m) ← EXTEND-SHORTEST-PATHS(L(m), L(m))
6          m ← 2m
7  Return L
```

درخت پوشای کمینه (Minimum Spanning Tree)

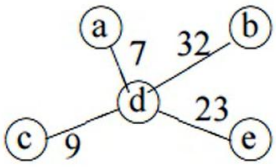
فرض کنید گراف یک گراف همبند باشد (یعنی بین هردو رأس متمایز آن یک مسیر وجود داشته باشد) منظور از یک درخت پوشا از این گراف درختی است که شامل همه رئوس این گراف باشد ولی فقط بعضی از یال‌های آن را دربر گیرد. منظور از درخت پوشای مینیمم (برای گراف همبند وزن دار) درختی است که بین درخت‌های پوشای آن گراف، مجموع وزن یال‌های آن، کمترین مقدار ممکن باشد.



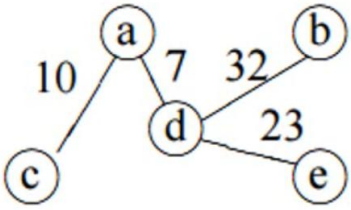
weighted graph



Tree 1. w=74



Tree 2, w=71
Minimum spanning tree



Tree 3, w=72

برای به دست آوردن درخت پوشای بهینه یک گراف جهت دار متصل می‌توان از الگوریتم‌های زیر استفاده کنیم:

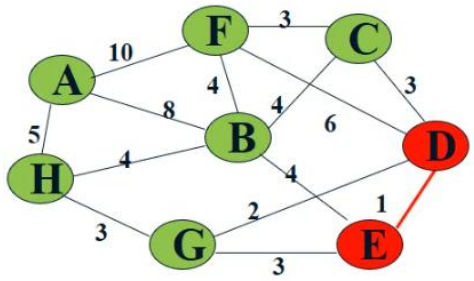
1- کروسکال(Kruskal)

2- پریم(Prim)

الگوریتم کروسکال(Kruskal)

این الگوریتم برای یافتن درخت پوشای کمینه یک گراف به کارمی رود. دراین الگوریتم ابتدا یال‌ها از کمترین وزن به بیشترین وزن مرتب می‌گردند سپس یال هابه ترتیب انتخاب شده و اگر یالی ایجاد حلقه کند، کنار گذاشته می‌شود. عملیات هنگامی خاتمه می‌یابد که تمام رأس هابه هم وصل شوند یا اینکه تعداد یال‌های موجود در F برابر n-1 شود که n تعداد رأس‌های گراف می باشد.

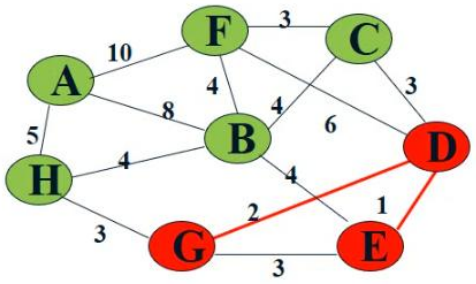
مثال: مطلوب است درخت پوشای کمینه در گراف زیر



edge	d_v	
(D,E)	1	√
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

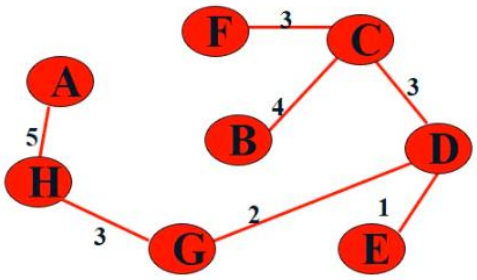
همانطور که مشاهده می کنید، یال های گراف بر اساس هزینه آنها در جدولی به صورت صعودی مرتب می شود و در انتخاب اول، کم هزینه ترین یال انتخاب می شود.



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

در قسمت بعد، یال با وزن کمتر انتخاب می شود. ولی انتخاب بعدی (E,G) ایجاد حلقه می کند پس انتخاب نمی شود. این مراحل تکرار می شود تا تمام گره ها انتخاب شوند.



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

not considered

Total Cost = 21

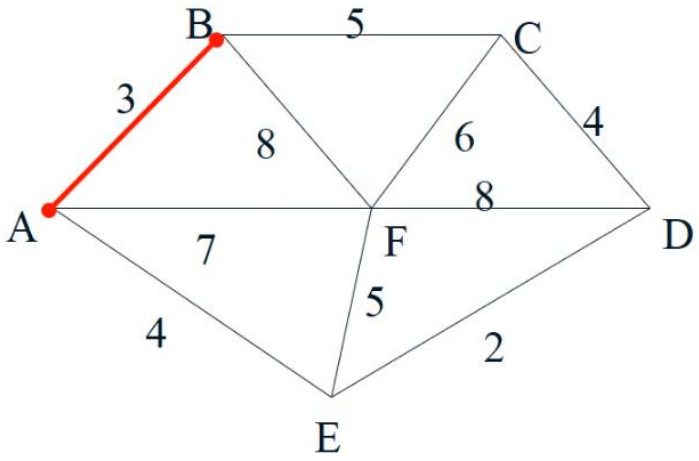
MST-Kruskal(G, w) $O(ElogE)$

```
1  A ← ∅
2  for each vertex v ∈ V[G]
3        do MAKE-SET(v)
4  sort the edges of E into nondecreasing order by weight w
5  for each edge (u, v) ∈ E, taken in nondecreasing order by weight
6        do if FIND-SET(u) ≠ FIND-SET(v)
7              then A ← A ∪ {(u, v)}
8                   UNION(u, v)
9  return A
```

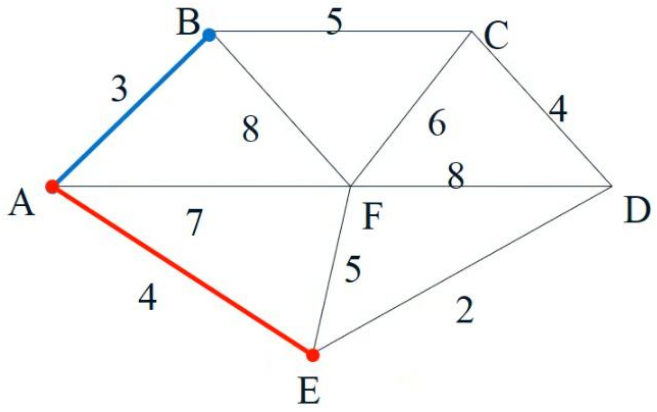
الگوریتم پریم(Prim)

در این روش از یک رأس شروع می‌کنیم و کمترین یال (یال با کمترین وزن) که از آن می‌گذرد را انتخاب می‌کنیم. در مرحله بعد یالی انتخاب می‌شود که کمترین وزن را در بین یال‌هایی که از دو گره موجود می‌گذرد داشته باشیم. به همین ترتیب در مرحله بعد یالی انتخاب می‌گردد که کمترین وزن را در بین یالهایی که از سه گره موجود می‌گذرد داشته باشد. این روال را آنقدر تکرار می‌کنیم تا درخت پوشای بهینه حاصل شود. باید توجه کرد که یال انتخابی در هر مرحله در صورتی انتخاب می‌شود که در گراف دور ایجاد نکند. تفاوت روش پریم با روش کراسکال در این است که گراف حاصل در مراحل میانی تشکیل درخت پوشای بهینه در روش پریم همیشه متصل است ولی در الگوریتم کراسکال در آخرین مرحله قطعاً متصل است.

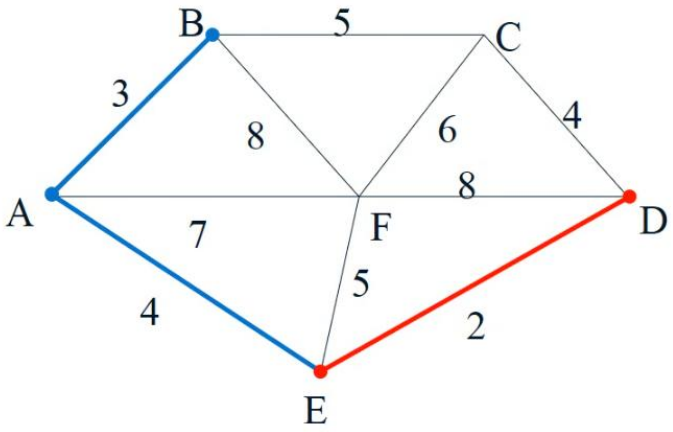
مثال: مطلوب است درخت پوشای کمینه برای گراف زیر



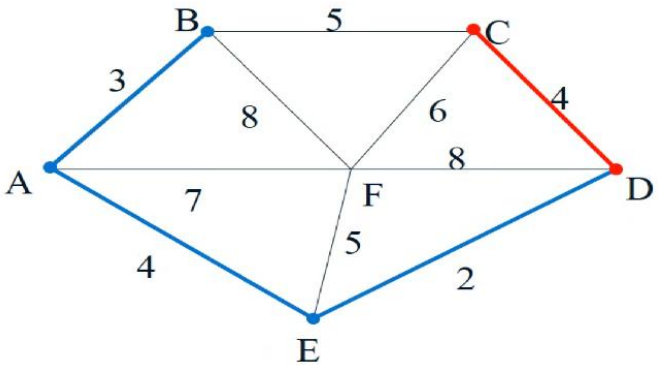
در این الگوریتم، یک گره به دلخواه انتخاب می شود که در اینجا گره A انتخاب شده است. مسیر های گره های همجوار گره انتخاب شده بررسی می شود و کم هزینه ترین مسیر برگزیده می شود. در اینجا یال متصل به B کمترین هزینه را دارد و انتخاب می شود.



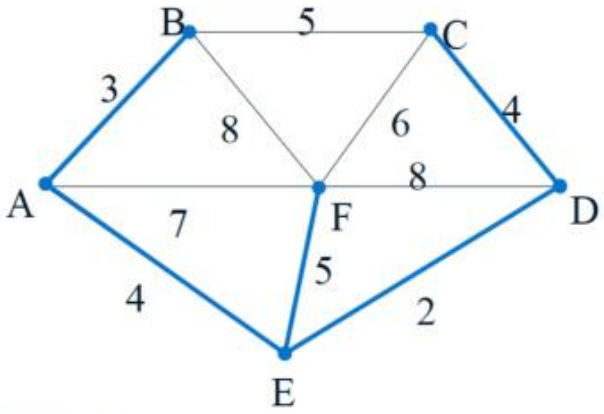
در گام بعد، باید کلیه مسیر ها بین گره های همجوار با A و B بررسی شود و کم هزینه ترین مسیر انتخاب شود. در اینجا مسیر A,E با هزینه 4 انتخاب شده است.



در این مرحله، گره ها و مسیرهای مربوط به A,B,E بررسی می شود که در این بین مسیر E,D به دلیل هزینه کمتر انتخاب می شود.



در این مرحله، مسیر ها و گره های مربوط به گره های A,B,E,D بررسی می شود و مسیر D,C انتخاب می گردد.



Total weight of tree: 18

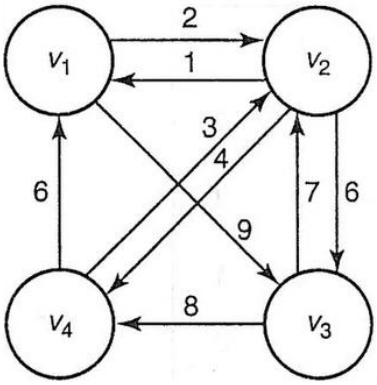
در انتها وقتی در بین مسیر های موجود می خواهیم کم هزینه ترین مسیر را انتخاب کنیم، مسیر E,F انتخاب می شود و در نهایت درخت پوشای کمینه با هزینه 18 بدست می آید.

```
MST-Prim(G, w, r)
1   for each u ∈ V[G]
2         do key[u] ← ∞
3            π[u] ← NIL
4   key[r] ← 0
5   Q ← V[G]
6   while Q ≠ ∅
7         do u ← EXTRACT-MIN(Q)
8            for each v ∈ Adj[u]
9                 do if v ∈ Q and w(u, v) < key[v]
10                      then π[v] ← u
11                           key[v] ← w(u, v)
```

مسئله فروشنده دوره گرد[1] (TSP)

فروشنده ای می خواهد به سفری برود که در آن بازدید از چند شهر در برنامه قرار دارد. هر شهر مسیر های مختلفی به شهر های دیگر دارد که هر مسیر دارای هزینه مخصوص به خود است. در این مسئله باید کوتاه ترین مسیر را از شهر محل سکونت فروشنده ارائه دهد که فروشنده بتواند هر شهر را یک بار بازدید کند و در نهایت به شهر خود بازگردد. این مسئله یافتن یک تور بهینه در گرافی جهت دار و وزن دار می باشد.

برای شروع این الگوریتم لازم است که ماتریس همجواری گراف را ترسیم کنیم.



	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

برای ادامه دادن رابطه بازگشتی زیر را باید داشته باشیم.

$$D[i][A] = \min_{j:vj \in A} (W[i][j] + D[j]\,[A - \{v_j\}].$$

D[i][Ø]=W[i][1]

در این رابطه سطر ها (i) تعداد شهر ها(گره) می باشد. همچنین A مجموعه مسیر های در طول مسیر می باشد که برای رفتن از مبدا به مقصد وجود دارد و تعداد آن برابر با 2 به توان n-1 که در آن n تعداد گره ها می باشد.

[1] Traveling Salesman Problem (TSP)

مراحل انجام کار:

ماتریس D برای پیشبرد الگوریتم به صورت زیر می باشد.

	$\{\varphi\}$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_2,v_3\}$	$\{v_2,v_4\}$	$\{v_3,v_4\}$	$\{v_2,v_3,v_4\}$
1								
2								
3								
4								

تعداد سطر ها در این ماتریس به تعداد گره ها می باشد و ستونها ($2^{N-1} = 2^3 = 8$) مسیر های موجود در نظر گرفته شده است.

تعیین ستون اول:

در تعیین ستون اول این ماتریس از فرمول D[i][Ø]=W[i][1] استفاده می شود چون ستون اول مربوط به Ø می باشد. همانطور که میدانید i سطر می باشد. در اینجا چون سطر 1 مربوط به گره آغازین می باشد همیشه خالی می ماند.

D[2][Ø]=W[2][1]=1

در اینجا منظور از w[2][1] سطر دوم و ستون اول ماتریس همجواری است و بقیه رابطه ها هم از روی ماتریس همجواری نوشته می شوند.

D[3][Ø]=W[3][1]=∞

D[4][Ø]=W[4][1]=6

	{φ}	{v2}	{v3}	{v4}	{v2,v3}	{v2,v4}	{v3,v4}	{v2,v3,v4}
1								
2	1							
3	∞							
4	6							

مجموعه های تک عنصری

در مرحله بعد شروع به پر کردن ستون دوم می کنیم. در سطر 2 و ستون 2 چون مربوط به گره دوم می باشد و مسیری از گره 2 به خودش نیست مقداری قرار نمی گیرد. برای محاسبه خانه ها از فرمول زیر استفاده می شود که قبلا گفته شد.

$$D[i][A] = \min_{j:vj\in A}(W[i][j] + D[j]\,[A - \{v_j\}].$$

در این این مرحله $A = v_j$ می باشد که اگر در فرمول بالا قرار بگیرد به فرمول زیر بدست می آید.

$$D[i]\left[v_j\right] = w[i][j] + D[j][\emptyset]$$

حال با این فرمول و جایگزاری مقادیر به عناصری که باید در ماتریس قرار دهیم خواهیم رسید. باید بدانید که منظور از این فرمول این است که مثلا از گره i شروع شود و حتما از گره v_j عبور کند.

$$D[2][\{v_3\}] = w[2][3] + D[3][\emptyset] = 6 + \infty = \infty$$

$$D[2][\{v_4\}] = w[2][4] + D[4][\emptyset] = 4 + 6 = 10$$

$$D[3][\{v_2\}] = w[3][2] + D[2][\emptyset] = 7 + 1 = 8$$

$$D[3][\{v_4\}] = w[3][4] + D[4][\emptyset] = 8 + 6 = 14$$

$$D[4][\{v_2\}] = w[4][2] + D[2][\emptyset] = 3 + 1 = 4$$

$$D[4][\{v_3\}] = w[3][3] + D[3][\emptyset] = 0 + \infty = \infty$$

حال مقادیر را در محل خودشان در جدول قرار می دهیم.

	$\{\varphi\}$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_2,v_3\}$	$\{v_2,v_4\}$	$\{v_3,v_4\}$	$\{v_2,v_3,v_4\}$
1								
2	1		∞	10				
3	∞	8		14				
4	6	4	∞					

مجموعه های دو عنصری

برای بدست آوردن مقادیر ستون دو عنصری باز از همان فرمول کلی استفاده می شود.

$$D[i][A] = \min_{j:vj\in A}(W[i][j] + D[j]\,[A - \{v_j\}].$$

نکته : اعضای مجموعه A اعضایی هستند که با j نمایش داده می شوند و j و i نباید مساوی باشند زیرا از یک گره به خود آن مسیری وجود ندارد. حال به محاسبه مقادیر می پردازیم. منظور از این فرمول این است که مثلا از گره 2 شروع می کنیم و حتما باید از گره های v_3 و v_4 باید عبور کنیم.

$$D[2][\{v_3,v_4\}] = \begin{cases} w[2][3] + D[3][\{v_4\} = 6 + 14 = 20] \\ w[2][4] + D[4][\{v_3\} = 4 + \infty = \infty\ ] \end{cases}$$

$$D[3][\{v_2,v_4\}] = \begin{cases} w[3][2] + D[2][\{v_4\} = 7 + 10 = 17] \\ w[3][4] + D[4][\{v_2\} = 8 + 4 = 12\ ] \end{cases}$$

$$D[4][\{v_2,v_3\}] = \begin{cases} w[4][2] + D[2][\{v_3\} = 3 + \infty = \infty] \\ w[4][3] + D[3][\{v_2\} = \infty + 8 = \infty\ ] \end{cases}$$

در نهایت از هر جفت عدد هر دسته طبق فرمول کمترین(min) را انتخاب می کنیم.

	{φ}	{v2}	{v3}	{v4}	{v2,v3}	{v2,v4}	{v3,v4}	{v2,v3,v4}
1								
2	1		∞	10			20	
3	∞	8		14		12		
4	6	4	∞		∞			

<u>جواب نهایی:</u>

برای بدست آوردن جواب نهایی به اطلاعات موجود در جدول نیاز می باشد و از همان فرمول کلی هم استفاده می شود.

$$D[i][A] = \min_{j:vj\in A}(W[i][j] + D[j]\,[A - \{v_j\}].$$

در گام آخر، از گره 1 شروع می کنیم و باید از هر سه گره v_2 ، v_3 و v_4 عبور کنیم.

$$D[1][\{v_2,v_3,v_4\}] = \begin{cases} w[1][2] + D[2][\{v_3,v_4\}] = 2 + 20 = 22 \\ w[1][3] + D[3][\{v_2,v_4\}] = 9 + 12 = 21 \\ w[1][4] + D[4][\{v_2,v_3\}] = \infty + \infty = \infty \end{cases}$$

از این نتایج باید min بگیریم تا جواب حاصل بدست بیاید.

	{φ}	{v2}	{v3}	{v4}	{v2,v3}	{v2,v4}	{v3,v4}	{v2,v3,v4}
1								21
2	1		∞	10			20	
3	∞	8		14		12		
4	6	4	∞		∞			

در اینجا مشخص می شود که هزینه کوتاه ترین مسیر 21 می باشد.

تعیین مسیر

اما برای تعیین مسیر که از گره v_1 شروع می شود نیز باید یک ماتریس مشابه ترسیم کنیم تا از روی آن ترتیب قرار گرفتن گره ها را مشخص کنیم.

	$\{\varphi\}$	$\{v_2\}$	$\{v_3\}$	$\{v_4\}$	$\{v_2,v_3\}$	$\{v_2,v_4\}$	$\{v_3,v_4\}$	$\{v_2,v_3,v_4\}$
1								3
2				4			3	
3		2		4		4		
4		2						

برای پر کردن این ماتریس باید به محاسبات مراجعه کنیم و در رابطه هایی که در آن کمینه گیری کردیم مقدار j رابطه ای که در کمینه گیری انتخاب شده است در این ماتریس قرار میگیرد.

عدد 3 در سطر 1 وستون $\{v_2, v_3, v_4\}$ مشخص می کند که از گره v_1 باید به گره v_3 برویم. حال به سطر 3 و ستون $\{v_2, v_4\}$ می رویم. مقدار 4 در آن نشان می دهد که از گره v_3 باید به گره v_4 برویم. سپس به ستون 4 و سطر $\{v_2\}$ که نگاه کنیم متوجه می شویم که باید از گره v_4 باید به گره v_2 برویم.

$$[v_1, v_3, v_4, v_2, v_1]$$

مرتبه اجرائی الگوریتم فروشنده دوره گرد $\boldsymbol{\theta(n^2 2^n)}$ می باشد.

فصل 7

مسائل P و NP

مسائل بطور کلی به دو دسته تقسیم می شوند.

1- مسائل قابل حل[1]

- مسائلی که در زمان قابل قبولی حل می شوند[2]. (مرتب سازی، جستجو ....)
- مسائلی که در زمان قابل قبول حل نمی شوند و زمان بر هستند[3]. (دور همیلتونی)

2- مسائل غیر قابل حل[4] : مسائلی هستند که برای آنها الگوریتمی وجود ندارد.

تقسیم بندی الگوریتم ها

الگوریتم ها بطور کلی به دو دسته اصلی تقسیم می شوند.

1- الگوریتم های قطعی[5]

الگوریتمی است که در شرایط غیررسمی رفتاری قابل پیش‌بینی دارد، با دادن یک ورودی خاص، همیشه خروجی خاصی را ایجاد کند و ماشین اساسی آن همیشه از مسیری یکسان برای دنباله‌های وضعیت مشابه عبور کند. الگوریتم قطعی تا حد زیادی مورد مطالعه قرارگرفته و آشناترین نوع الگوریتم است، همچنین یکی از عملی‌ترین الگوریتم‌ها محسوب می‌شود، چرا که آن‌ها را می‌توان به‌طور کارآمدی بر روی ماشین‌های واقعی اجرا کرد . به‌طور رسمی یک الگوریتم قطعی یک تابع ریاضی را محاسبه می‌کند. یک تابع برای هر ورودی یک مقدار منحصربه‌فرد دارد، و الگوریتم فرایندی است که این مقدار منحصربه‌فرد را به عنوان خروجی را تولید می‌کند.

[1] Solvable
[2] Tractable
[3] Untractable
[4] Unsolvable
[5] Deterministic Algorithm

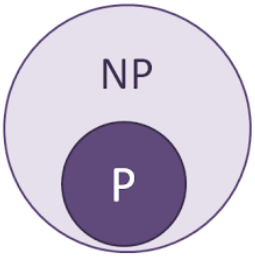
2- الگوریتم های غیر قطعی[1]

عوامل و متغیرهای زیادی می‌توانند باعث شوند که یک الگوریتم به‌صورت تصادفی یا غیرقطعی رفتار کند مانند:

- زمانی که ماشین از حالت‌های خارجی به‌غیر از ورودی استفاده کند، مانند ورودی کاربر، متغیر جهانی، مقدار زمان‌سنج سخت‌افزار، مقدار اتفاقی یا داده ذخیره شده روی دیسک.
- زمانی که ماشین حساس به زمان عمل می‌کند برای مثال زمانی که می‌خواهید چند عمل نوشتن روی قسمتی از حافظه را به‌صورت همزمان انجام دهید، در این حالت مرتبهٔ قیمتی که هر پردازنده برای نوشتن داده‌اش می‌پردازد، نتیجه را تحت تأثیر قرار می‌دهد.
- زمانی که یک خطای سخت‌افزاری باعث شود تا ماشین حالت خود را به یک مسیر غیرمنتظره تغییر دهد.

تعریف

اگر الگوریتمی داشته باشیم که بصورت قطعی و در زمان چند جمله ای جواب را ارائه دهد می گوئیم این الگوریتم در کلاس P[2] می باشد. همچنین اگر الگوریتمی داشته باشیم که در زمان چند جمله ای جواب غیر قطعی به ما ارائه دهد می گوئیم این الگوریتم در کلاس NP[3] می باشد.



در حال حاضر این مسئله وجود دارد که p زیر مجموعه np می باشد و این دو با هم برابر نیستند.

[1] Nondeterministic Algorithm
[2] Polynomial-time
[3] Nondeterministic Polynomial-time

بیان الگوریتم های غیر قطعی

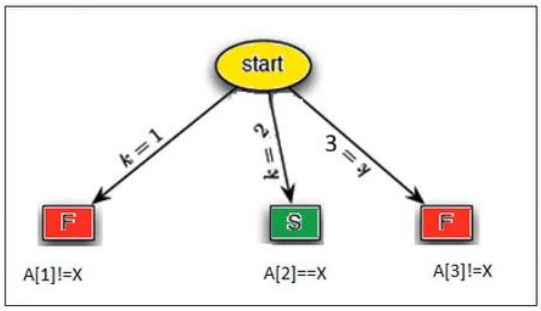
برای بیان الگوریتم های غیر قطعی می توان 3 تابع را بیان کرد.

1- Choose(S) : این تابع همزمان S کپی از یک ماشین را ایجاد می کند و هر آیتم از مسئله را به یک ماشین اختصاص می دهد.
2- Success () : این تابع بررسی می کند که جوابی برای مسئله وجود داشته عمل می کند.
3- Failure() : این تابع اگر جوابی برای مسئله پیدا نکند عمل می کند.

مثال:

```
1: NSearch (A[1···n], x)
2: k ← choose({1, 2, ··· , n}) ;
3: if A[k] = x then
4:    success() ;
5: else
6:    failure() ;
7: end if
8: end.
```

A[1..3]={7,5,9}
x=5



در این مثال یک آرایه با طول 3 در نظر گرفته شده است، 3 کپی از آن ایجاد شده و در اختیار ماشین های *k=1,k=2* و *k=3* قرار می گیرد. به هر ماشین گفته می شود که یک درایه را بررسی کند که آیا مقدار *x=5* با مقدار آن درایه برابر است یا برابر نمی باشد. ماشین *k=1* بررسی را در درایه اول انجام میدهد و مقدار برابر نمی باشد پس تابع *Failure* اجرا می شود. ماشین *k=2* با بررسی مقدار و برابر بودن آنها تابع *Success* را اجرا می کند و همچنین ماشین *k=3* نیز با شکست مواجه می شود. دیده می شود این مسئله که بصورت غیر قطعی حل شده است از مرتبه اجرائی *O(1)* می باشد و سریعتر اجرا میشود. البته در واقعیت در حال حاضر امکان پذیر نیست.

بهینه سازی[1]-تصمیم گیری[2]

هر مسئله بهینه سازی می تواند به یک مسئله تصمیم گیری تبدیل شود در صورتی که یک مقدار مرزی را در نظر بگیریم. مسائل تصمیم گیری دارای خروجی منطقی می باشند.

مثال :

در مسئله کوله پشتی که یک مسئله بهینه سازی بود ما تلاش می کردیم که کوله پشتی را به نحوی پر کنیم تا بیشترین سود را کسب کنیم. در مسئله تصمیم گیری می گوئیم آیا راه دیگری وجود دارد که جواب بهتری از جواب کنونی داشته باشد یا خیر.

:Reduction

فرض می کنیم مسائل A و B مسائل تصمیم گیری باشند. می گوئیم A، Reduce شده به B

$A \leqslant_p P$

اگر الگوریتم چند جمله ای p وجود داشته با شد که:

1- هر نمونه دلخواه از A را به نمونه ای از B تبدیل کند.

2- جواب ها معادل باشند.(اگر جواب A برابر yes باشد برای B هم yes باشد و بالعکس)

در اینجا می گوئیم سختی A از سختی B کم تر است($A \leqslant_p P$).

اگر حل A سخت باشد ، سپس حل B نیز سخت است.

اگر حل B ساده باشد ، سپس حل A نیز ساده است.

[1] Optimization
[2] Decision

اگر حل A ساده باشد ، در رابطه با B نمی توان نظر داد.

اگر حل B سخت باشد ، در رابطه با A نمی توان نظر داد.

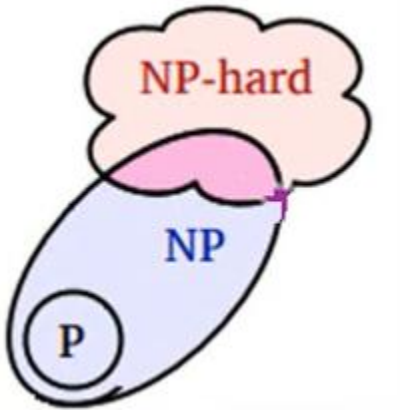
برای اینکه نشان دهیم B سخت است، کافی است یک مسئله شناخت شده سخت به نام A را انتخاب کرده و به B ، Reduction بزنیم.

مسائل کلاس NP-Hard

به مسائلی که از مسائل P سخت تر باشند را NP-hard می گوئیم.

مسئله L را NP-hard می گوئیم اگر:

$$L \in NP - hard \iff \forall\, L' \in NP : L' \precsim_p L$$



در تصویر مشاهده می شود که همه مسائل *NP-hard* از کلاس *P* نمی باشند.

اگر یک مسئله هم *NP* باشد و هم *NP-hard* آنگاه می گوئیم مسئله از کلاس *NP-complete* می باشد.