روش حریصانه
Greedy Method

مطالب مورد بحث

- مقدمه
- بررسی چند مساله نمونه
  - کوله پشتی غیر صفر و یک
  - زمانبندی بهینه اجرای برنامه ها
  - درخت پوشای مینیمم
    - الگوریتم Prim
    - الگوریتم Kruskal
  - زمانبندی بهینه فرایندهای دارای ضرب الاجل
  - فشرده سازی با استفاده از کد Huffman

مقدمه

- در مسائلی که به این روش حل می شوند، معمولا:
  - ورودی مساله شامل مجموعه ای از عناصر است.
  - خروجی مساله شامل مجموعه ای از عناصر است که بیشتر مواقع، زیرمجموعه ورودی مساله می باشد ← مساله انتخاب
    - علیرغم استفاده از لفظ مجموعه، ممکن است ترتیب عناصر مجموعه جواب، مهم باشد.
  - موضوع بهینگی مطرح است.
    - یعنی حل مساله، مستلزم انتخاب زیرمجموعه ای از مجموعه عناصر ورودی می باشد که تابع هدف مساله را بهینه می نمایند.
  - جواب را می توان بصورت مرحله به مرحله بدست آورد. (در هر مرحله یک مولفه جواب را بدست آورد)
- مثال: زمانبندی بهینه اجرای برنامه ها

مقدمه

- کلیات روش حریصانه
  - در ابتدا مجموعه جواب را تهی در نظر می گیریم.
  - بطور مرحله به مرحله عمل می کنیم. در هر مرحله:
    - از بین عناصر ممکن که می توانیم بعنوان عنصر بعدی مجموعه جواب انتخاب کنیم، بهترین عنصر را در نظر می گیریم. (selection)
    - بررسی می کنیم آیا با انتخاب آن مولفه، امکان رسیدن به جواب وجود دارد یا خیر (feasibility check).
      - اگر بله: آن مولفه را به مجموعه جواب اضافه می کنیم.
      - اگر خیر: آن مولفه را به مجموعه جواب اضافه نمی کنیم و آن را برای همیشه کنار می گذاریم.
    - با افزودن هر عنصر به مجموعه جواب، بررسی می کنیم اگر جواب مساله حاصل شده است، جواب بدست آمده بهینه خواهد بود و کار تمام است. (solution check)
      - بنابراین با بدست آوردن اولین جواب کار تمام می شود.

مطالب مورد بحث

- کوله پشتی غیر صفر و یک
- زمانبندی بهینه اجرای برنامه ها
- درخت پوشای مینیمم
  - الگوریتم Prim
  - الگوریتم Kruskal
- زمانبندی بهینه فرایندهای دارای ضرب الاجل
- فشرده سازی با استفاده از کد Huffman

کوله پشتی غیر صفر و یک

- تعدادی شیء و یک کوله پشتی داریم.
- شیء i دارای وزن w_i و ارزش p_i می باشد.
- ظرفیت وزنی کوله پشتی برابر M می باشد.
- هدف: اشیا را برای قرار دادن در داخل کوله پشتی انتخاب کنیم، بگونه ای که ضمن رعایت ظرفیت وزنی کوله پشتی، مجموع ارزش اشیای انتخاب شده، ماکزیمم گردد.
- امکان انتخاب کسری از هر شیء نیز وجود دارد. بنابراین از هر شیء i به میزان x_i واحد انتخاب می کنیم. $(0 \leq xi \leq 1)$
  - اگر اینطور نباشد، یعنی یا باید یک شیء را بطور کامل انتخاب کنیم یا اصلا آن را انتخاب نکنیم. در اینصورت، مساله به مساله کوله پشتی صفر و یک تبدیل خواهد شد.

کوله پشتی غیر صفر و یک

- بیان ریاضی
- تعداد اشیا: n
- $0 \le x_i \le 1$

$$\text{maximize} \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \le M$$

کوله پشتی غیر صفر و یک

- حدس ۱:
  - چون ارزش اشیا تاثیر مثبت دارد و می خواهیم مجموع ارزش اشیای انتخاب شده را ماکزیمم کنیم، پس بهتر است اشیا را به ترتیب بیشترین ارزش انتخاب کنیم.
  - اشیا را برحسب ارزش بطور غیر صعودی مرتب کنیم
  - از ابتدا اشیا را به ترتیب بررسی می کنیم.
    - اگر وزن شیء از ظرفیت باقیمانده کوله کمتر بود آن را انتخاب می کنیم.
    - اگر وزن شیء از ظرفیت باقیمانده کوله بیشتر بود، کل ظرفیت باقیمانده را با کسری از شیء پر می کنیم و کار تمام است.

کوله پشتی غیر صفر و یک

- حدس ۲:
  - چون وزن اشیا تاثیر منفی دارد و می خواهیم کوله پشتی دیرتر پر شود، پس بهتر است اشیا را به ترتیب کمترین وزن انتخاب کنیم.
  - اشیا را برحسب وزن بطور غیر نزولی مرتب کنیم
  - از ابتدا اشیا را به ترتیب بررسی می کنیم.
    - اگر وزن شیء از ظرفیت باقیمانده کوله کمتر بود آن را انتخاب می کنیم.
    - اگر وزن شیء از ظرفیت باقیمانده کوله بیشتر بود، کل ظرفیت باقیمانده را با کسری از شیء پر می کنیم و کار تمام است.

کوله پشتی غیر صفر و یک

- حدس ۳:
  - چون ارزش اشیا تاثیر مثبت و وزن اشیا تاثیر منفی دارد، بهتر است اشیا را به ترتیب بیشترین pi/wi انتخاب کنیم.
  - اشیا را برحسب نسبت ارزش به وزن بطور غیر صعودی مرتب کنیم
  - از ابتدا اشیا را به ترتیب بررسی می کنیم.
    - اگر وزن شیء از ظرفیت باقیمانده کوله کمتر بود آن را انتخاب می کنیم.
    - اگر وزن شیء از ظرفیت باقیمانده کوله بیشتر بود، کل ظرفیت باقیمانده را با کسری از شیء پر می کنیم و کار تمام است.

کوله پشتی غیر صفر و یک

- مثال: M=20 و n=3

$(p_1 , p_2 , p_3) = (25, 24, 15)$

$(w_1 , w_2 , w_3) = (18, 15, 10)$

حدس مورد استفاده	جواب	مجموع وزن اشیای انتخاب شده	مجموع ارزش اشیای انتخاب شده
حدس اول	(1, 2/15, 0)	20	28.2
حدس دوم	(0, 2/3, 1)	20	31
حدس سوم	(0, 1, 1/2)	20	31.5

کوله پشتی غیر صفر و یک

- فقط حدس سوم درست است و با استفاده از آن، الگوریتمی منطبق بر روش حریصانه ارائه خواهیم نمود.
- درستی این حدس را باید اثبات کنیم.

کوله پشتی غیر صفر و یک

```
void knapsack(float P[], float W[], float M, float X[], int n) {
    pwSort(P, W, n);  //اشیا را بر حسب نسبت ارزش به وزن، نزولی مرتب کن
    for(int i=0; i<n; i++)          X[i] = 0; // مقداردهی اولیه
    rc = M; // rc= remained capacity ظرفیت باقیمانده کوله پشتی
    for(int i=0; i<n; i++) {
        if (W[i] ≤ rc)   {
            X[i] = 1;          rc=rc – W[i];
        } else {
            X[i] = rc/W[i];   rc=0;   break;                    }
    }
}
```

کوله پشتی غیر صفر و یک

- محاسبه مرتبه زمانی
- ورودی: لیست وزن و ارزش اشیای ورودی و مقدار M
- اندازه ورودی: تعداد اشیای ورودی یعنی n
- عمل کلیدی: مقایسه W[i] با rc
- تابع پیچیدگی:

- $W(n) = n + W_{pwSort}(n)$
- $W_{pwSort}(n) = O(n \log^n)$

$$W(n) = O(n \log^n)$$

کوله پشتی غیر صفر و یک

- الگوریتم منطبق بر الگوی روش حریصانه می باشد.
- اثبات درستی

مطالب مورد بحث

- کوله پشتی غیر صفر و یک
- **زمانبندی بهینه اجرای برنامه ها**
- درخت پوشای مینیمم
  - الگوریتم Prim
  - الگوریتم Kruskal
- زمانبندی بهینه فرایندهای دارای ضرب الاجل
- فشرده سازی با استفاده از کد Huffman

زمانبندی بهینه اجرای برنامه ها

- تعدادی برنامه با زمان اجرای معلوم موجود است. می خواهیم برنامه ها را به ترتیبی اجرا کنیم که زمان متوسط برگشت آنها مینیمم گردد.
- زمان برگشت (turnaround time)، مدت سپری شده از زمان تحویل برنامه به کامپیوتر تا پایان اجرای آن می باشد.
- ویژگیهای مساله
  - ورودی مجموعه ای از عناصر است (مجموعه زمان اجرای برنامه ها)
  - خروجی مجموعه ای از عناصر است. در این مثال تعداد عناصر مجموعه خروجی با تعداد عناصر مجموعه ورودی برابر است و فقط ترتیب عناصر ممکن است متفاوت باشد.
  - مساله بهینه سازی مطرح است: کمینه کردن متوسط زمان برگشت

زمانبندی بهینه اجرای برنامه ها

- احتمالا می توانیم در هر مرحله یکی از مولفه های مجموعه جواب را تعیین کنیم.

- این ویژگیها ← مساله به احتمال زیاد به روش حریصانه قابل حل است.

زمانبندی بهینه اجرای برنامه ها

- مثال : سه برنامه P_1 و P_2 و P_3 با زمان های اجرای ۸ و ۴ و ۶

ترتیب اجرا	زمان برگشت برنامه اول	زمان برگشت برنامه دوم	زمان برگشت برنامه سوم	زمان متوسط برگشت
$P_1 P_2 P_3$	۸	۸ + ۴	۸ + ۴ + ۶	۱۲.۷
$P_1 P_3 P_2$	۸	۸ + ۶	۸ + ۶ + ۴	۱۳.۳
$P_2 P_1 P_3$	۴	۴ + ۸	۴ + ۸ + ۶	۱۱.۳
$P_2 P_3 P_1$	۴	۴ + ۶	۴ + ۶ + ۸	۱۰.۷
$P_3 P_1 P_2$	۶	۶ + ۸	۶ + ۸ + ۴	۱۲.۷
$P_3 P_2 P_1$	۶	۶ + ۴	۶ + ۴ + ۸	۱۱.۳

زمانبندی بهینه اجرای برنامه ها

ترتیب اجرا	زمان برگشت برنامه اول	زمان برگشت برنامه دوم	زمان برگشت برنامه سوم	مجموع زمان برگشت
$P_x P_y P_z$	x	x + y	x + y + z	3x + 2y +z

- از آنجا که می خواهیم میانگین را مینیمم کنیم، پس باید مجموع زمانهای برگشت (sum) را مینیمم کنیم.
- زمان اجرای برنامه اول ۳ بار در sum ظاهر می شود و زمان اجرای برنامه دوم، ۲ بار و زمان اجرای برنامه سوم، ۱ بار.
- حدس: برنامه ها را به ترتیب زمان اجرا، بطور غیر نزولی مرتب کرده و به همان ترتیب اجرا نماییم.
- بدین ترتیب برنامه ای که بیشترین تاثیر را در sum دارد، برنامه ای خواهد بود که کمترین زمان اجرا را دارد و ....

زمانبندی بهینه اجرای برنامه ها

- درستی این حدس را هم باید اثبات کرد.

زمانبندی بهینه اجرای برنامه ها

- الگوریتم منطبق بر الگوی روش حریصانه

```
void optOrder(float P[], float C[], int n) {
                                        //آرایه P زمان اجرای برنامه ها را شامل می شود
                                        //آرایه C خروجی برنامه است که ترتیب اجرای برنامه ها را مشخص می کند.
    for (int i=0; i<n ; i++) {
        min = P[i];         minIndex = i;
        for( int j=i+1; j<n; j++) {
            if (P[j] < min) {
                min = P[j];         minIndex = j;
            }
        }
        swap P[minIndex] and P[i]
        C[i] = P[i];
    }
}
```

زمانبندی بهینه اجرای برنامه ها

- محاسبه مرتبه زمانی الگوریتم optOrder
- عمل کلیدی: عمل مقایسه P[j] با min

$$T(n) = (n-1) + (n-2) + (n-3) + \ldots + 2 + 1$$

$$T(n) = \frac{(n-1)*n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

$$\Rightarrow T(n) = O(n^2)$$

زمانبندی بهینه اجرای برنامه ها

- الگوریتم دیگر با مرتبه زمانی بهتر (منطبق بر الگوی روش حریصانه نیست)

```
void optOrder2( float P[], float C[], int n) {
    C = descendingSort(P);
}
```

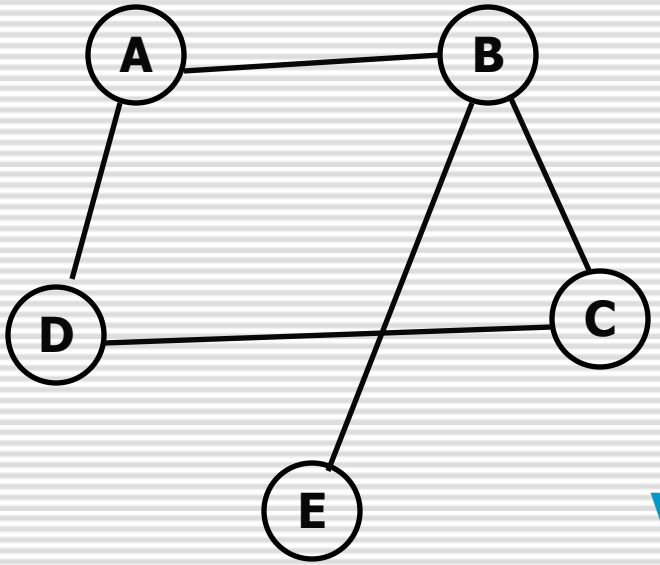
- از آنجا که الگوریتم هایی برای مرتب سازی با مرتبه زمانی O(nlogn) وجود دارد
  - پس الگوریتم optOrder2 نیز مرتبه زمانی اش O(n logn) می باشد.
- نکته: الگوریتم قبلی (optOrder) را می توان یک الگوریتم مرتب سازی مبتنی بر روش حریصانه به حساب آورد.

مطالب مورد بحث

- کوله پشتی غیر صفر و یک
- زمانبندی بهینه اجرای برنامه ها
- درخت پوشای مینیمم
  - الگوریتم Prim
  - الگوریتم Kruskal
- زمانبندی بهینه فرایندهای دارای ضرب الاجل
- فشرده سازی با استفاده از کد Huffman

درخت پوشای مینیمم

- گراف: G=(V, E)
- V: مجموعه گره ها
- E: مجموعه یالها

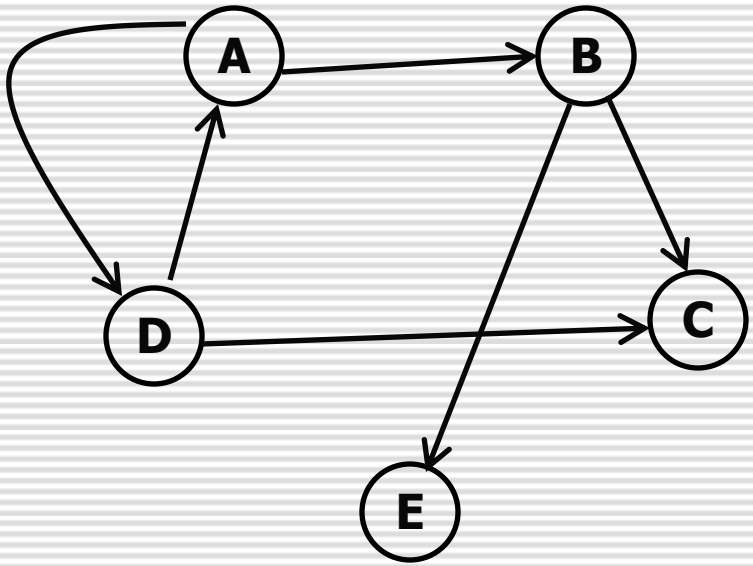


V = { A, B, C, D, E }

E = { (A,B), (A,D), (B,C), (B,E), (C, D) }

درخت پوشای مینیمم

- گراف جهت دار



V = { A, B, C, D, E }

E = { <A,B>, <D,A>, <D,A>, <B,C>, <B,E>, <D,C> }

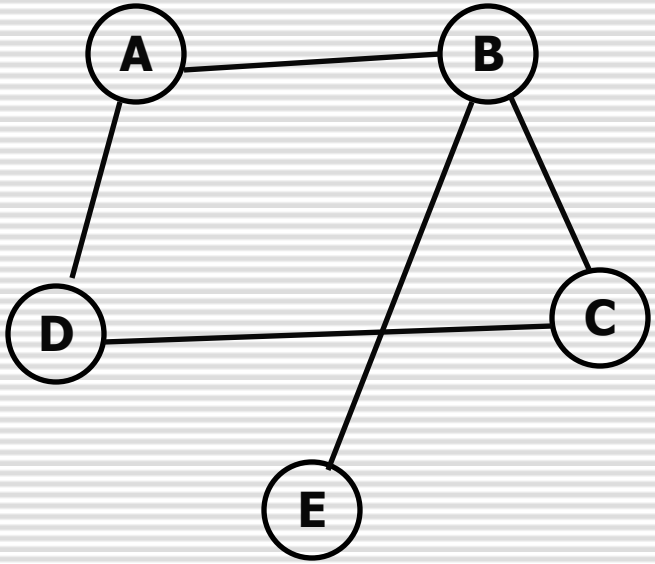
درخت پوشای مینیمم

- گراف وزن دار: گرافی که یالهای آن دارای وزن می باشند. مفهوم وزن یالها، به اینکه گراف در چه کاربردی استفاده شده باشد، بستگی دارد.
  - گراف راههای ارتباطی بین شهرها
    - گره ها: شهرها
    - یالها: جاده ها
    - وزن یالها: طول جاده ها
- بین دو گره متمایز u و v مسیری وجود دارد اگر
  - یک یال بین دو گره u و v موجود باشد. یا
  - یک گره z موجود باشد که با یک یال به گره u متصل باشد و ضمنا مسیری بین دو گره z و v موجود باشد.

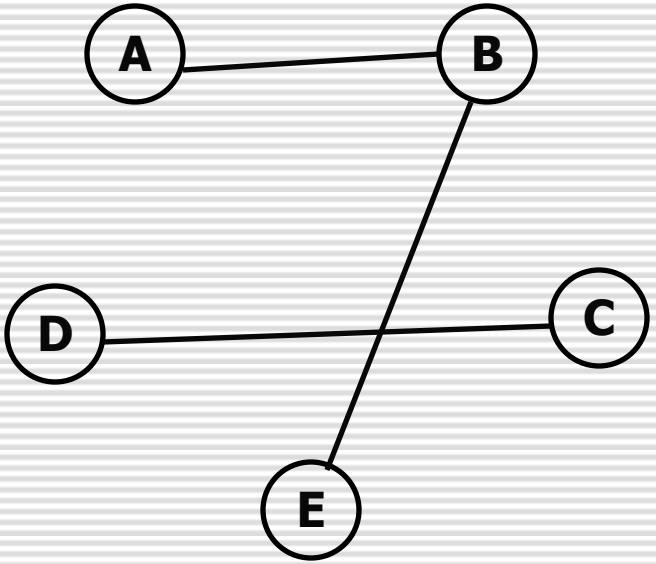
درخت پوشای مینیمم

- چرخه یا حلقه (cycle): اگر در یک گراف مسیری از یک گره به خودش وجود داشته باشد، آن گراف دارای حلقه خواهد بود.
- گراف متصل یا همبند (connected): گرافی که بین هر دو گره دلخواه آن حداقل یک مسیر وجود داشته باشد.
- درخت: گراف همبند بدون جهت بدون حلقه
- زیرگراف: گراف G2=(V2, E2)، زیرگراف گراف G1=(V1,E1) خواهد بود اگر
  - V2 زیرمجموعه V1 باشد.
  - E2 زیرمجموعه E1 باشد.

درخت پوشای مینیمم

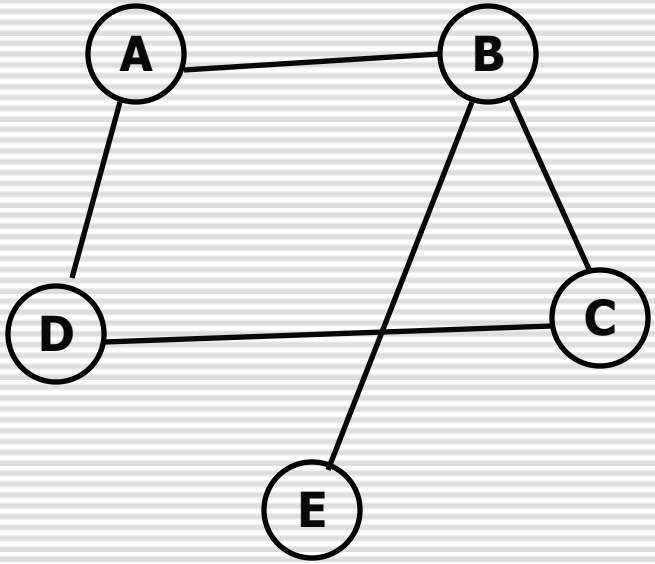


گراف G

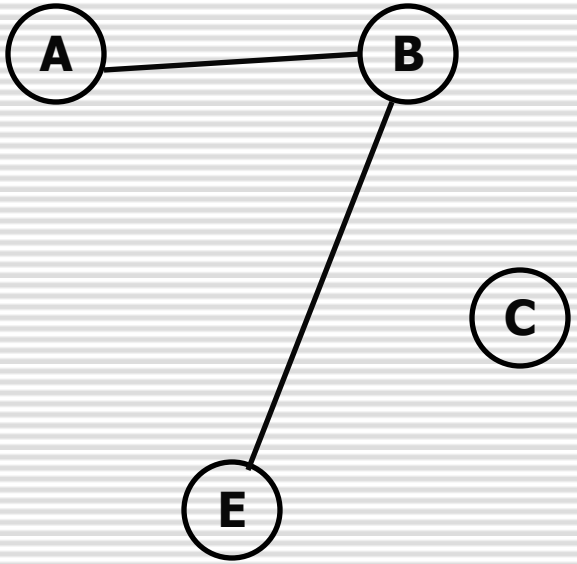


یک زیرگراف از گراف G

درخت پوشای مینیمم



گراف G

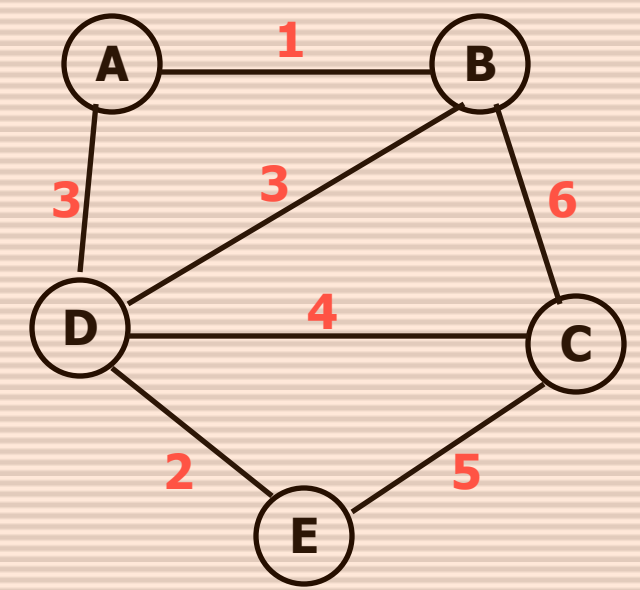


یک زیرگراف از گراف G

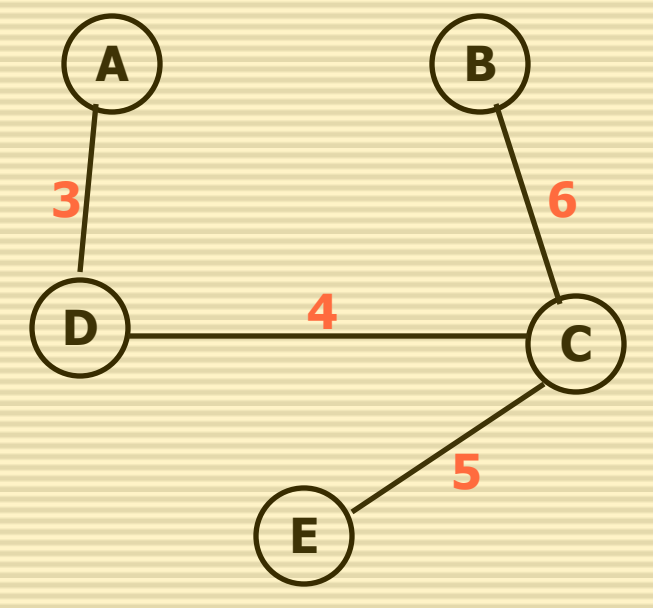
درخت پوشای مینیمم

- درخت پوشا
  - اگر G یک گراف همبند بدون جهت باشد، T یک درخت پوشا برای G است اگر
    - T یک زیرگراف از گراف G باشد که مجموعه گره هایش برابر مجموعه گره های G باشد و
    - T یک درخت باشد.
- درخت پوشای مینیمم (MST: Minimum Spanning Tree)
  - اگر G یک گراف همبند بدون جهت وزن دار باشد، T یک درخت پوشای مینیمم برای G است اگر
    - T یک درخت پوشا برای G باشد و
    - هزینه T از هزینه تمام درخت های پوشای گراف G کمتر باشد.

درخت پوشای مینیمم

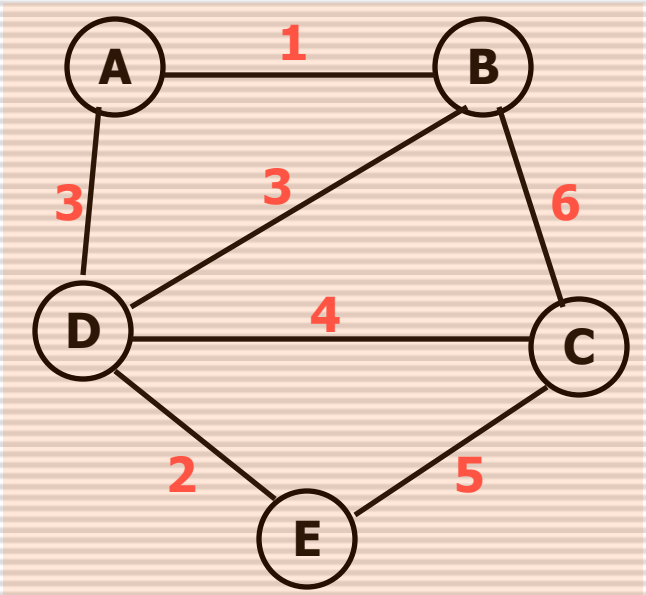


گراف G

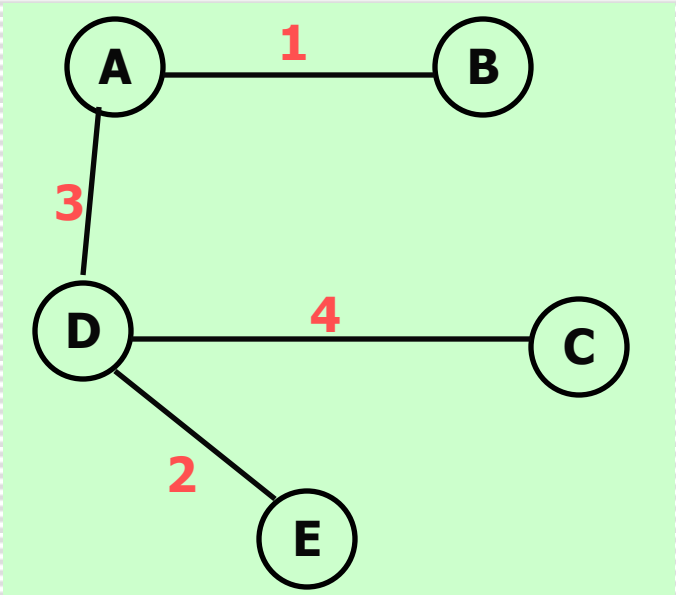


یک درخت پوشای غیر
مینیمم برای گراف G

درخت پوشای مینیمم

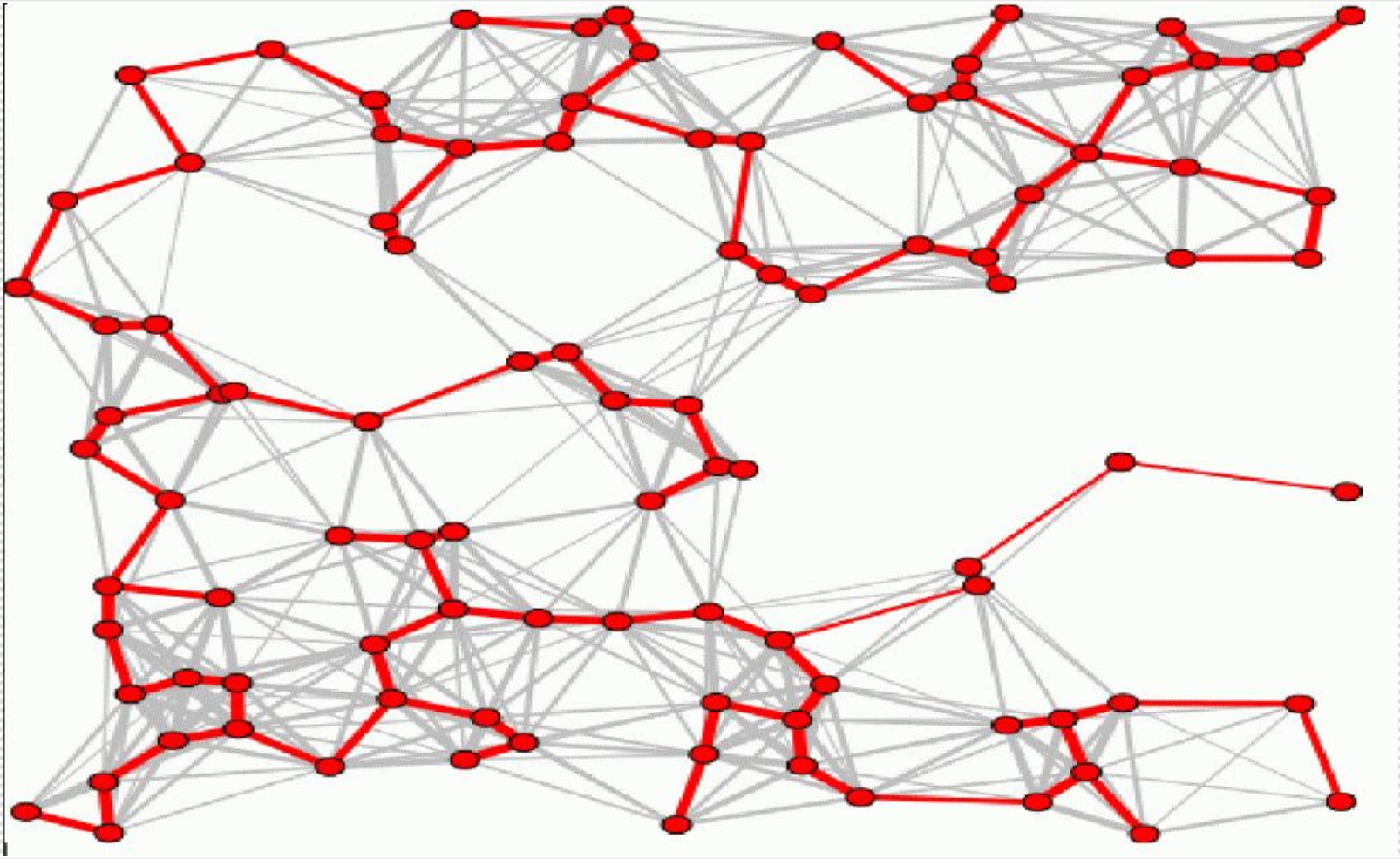


گراف G

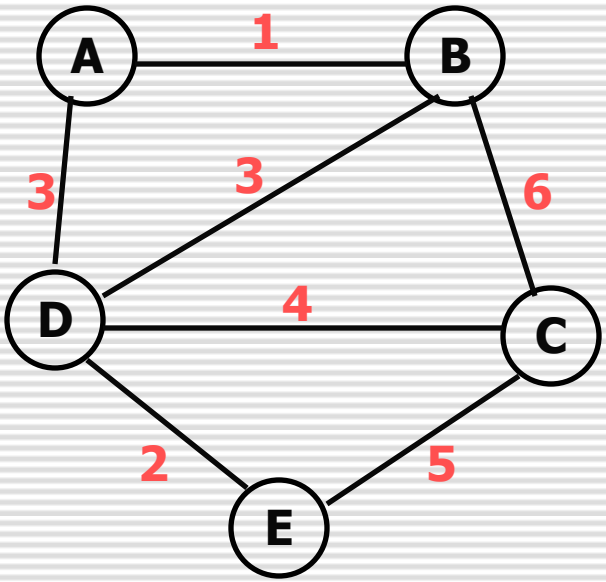


درخت پوشای مینیمم گراف G

درخت پوشای مینیمم



درخت پوشای مینیمم



- یک روش نمایش گراف: ماتریس مجاورت

	A	B	C	D	E
A	0	1	3	∞	∞
B	1	0	6	3	∞
C	∞	6	0	4	5
D	3	3	4	0	2
E	∞	∞	5	2	0

- با $O(1)$ می توان هزینه یال بین دو گره را بدست آورد.

درخت پوشای مینیمم

- مساله: الگوریتمی که گراف همبند بدون جهت وزن دار G را گرفته و درخت پوشای مینیمم آن را بدست آورد.
- الگوریتم Prim و همچنین الگوریتم Kruskal مبتنی بر روش حریصانه برای حل این مساله طراحی شده اند.

درخت پوشای مینیمم / الگوریتم Prim

- روش کلی
  - از آنجا که تمام گره ها باید در MST باشند، ابتدا یک گره را به دلخواه بعنوان عضو اول مجموعه جواب انتخاب می کنیم. سپس در هر مرحله یک گره انتخاب نشده را به عنوان عضو بعدی مجموعه جواب انتخاب می کنیم.
  - در هر مرحله
    - گره ای را انتخاب می کنیم که با کمترین هزینه (وزن یال) به گره های مجموعه جواب متصل می شود.
    - این گره را به مجموعه جواب اضافه می کنیم.

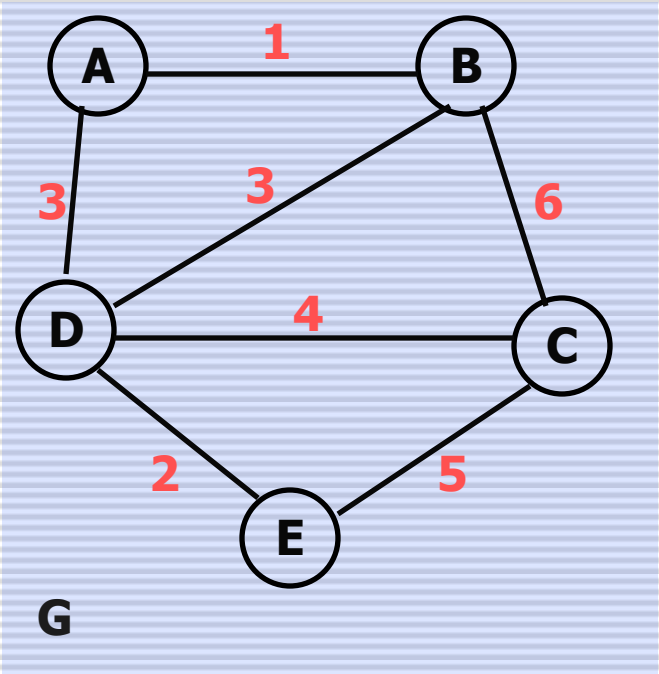
درخت پوشای مینیمم / الگوریتم Prim

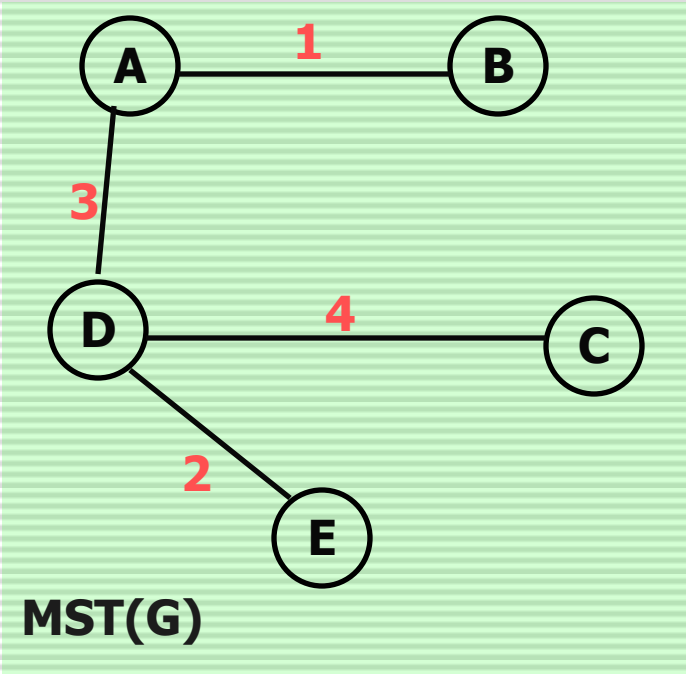
- قضیه ۱: افزودن یک یال (بین گره های موجود) به یک درخت موجب ایجاد حلقه می شود.
- اثبات: درخت T با n گره موجود است. یال e را به آن می افزاییم که گره های u و v را به هم متصل می کند. از آنجا که T در ابتدا، درخت بود بنابراین مسیری بین دو گره u و v موجود بوده است (مسیر p1). با افزودن یال e که u را به v متصل می کند، مسیر دیگری بین u و v بوجود خواهد آمد (خود یال e در واقع یک مسیر جدید است: p2). ترکیب دو مسیر p1 و p2، یک حلقه می باشد.
- واضح است که یال افزوده شده در حلقه قرار دارد.

درخت پوشای مینیمم / الگوریتم Prim

- $G=(V,E)$ یک گراف همبند، بدون جهت و وزن دار
- تعریف: مجموعه F زیرمجموعه E، را امیدوارکننده (promising) گوییم اگر F این قابلیت را داشته باشد که به $MST(G)$ منتهی شود.
  - یعنی بتوان با افزودن تعدادی از یالهای G به آن، به درخت پوشای مینیمم G رسید.
  - یعنی F شامل هیچ یال مزاحمی نباشد.

درخت پوشای مینیمم / الگوریتم Prim





F={ (B,C), (A,B) } **is not** promising

F={ (A,B), (A,D) } **is** promising

درخت پوشای مینیمم / الگوریتم Prim

- قضیه ۲: $G=(V,E)$ گراف همبند بدون جهت و وزن دار است.
  - فرض: E' یک زیرمجموعه امیدوار کننده از E، و V' مجموعه گره هایی که توسط یالهای E' بهم متصل می شوند.
  - حکم: اگر e کم وزن ترین یالی باشد که یک گره V' را به گره ای در مجموعه $V-V'$ متصل می کند، آنگاه $E' \cup \{e\}$ نیز امیدوارکننده خواهد بود.
    - یعنی با افزودن یال e به E' باز هم E' این قابلیت را دارد که به $MST(G)$ منتهی شود.
- اثبات

درخت پوشای مینیمم / الگوریتم Prim

- الگوریتم prim بر اساس قضیه ۲ عمل می کند.
- ابتدا Y تنها شامل یک گره می باشد (به دلخواه انتخاب می شود)
  - درنتیجه F تهی می باشد. (واضح است که F امیدوارکننده است)
- در هر گام،
  - ابتدا گره های موجود در V-Y را بررسی می کنیم و برای هر گره i موجود در V-Y، کم هزینه ترین یالی را که آن گره را به گره ای در Y متصل می کند مشخص می کنیم، وزن این یال را با dist(i) نمایش می دهیم.
  - سپس گره ای را که دارای کمترین dist می باشد را انتخاب کرده و آن را به مجموعه V-Y منتقل می کنیم. همچنین یال مربوط به آن را به F اضافه می کنیم.
- ادامه می دهیم تا Y شامل تمام گره های G شود.

درخت پوشای مینیمم / الگوریتم Prim

ساختار مورد استفاده برای انجام عملیات الگوریتم

nodes	Y	near	minDist
0			
1			
...			
n			

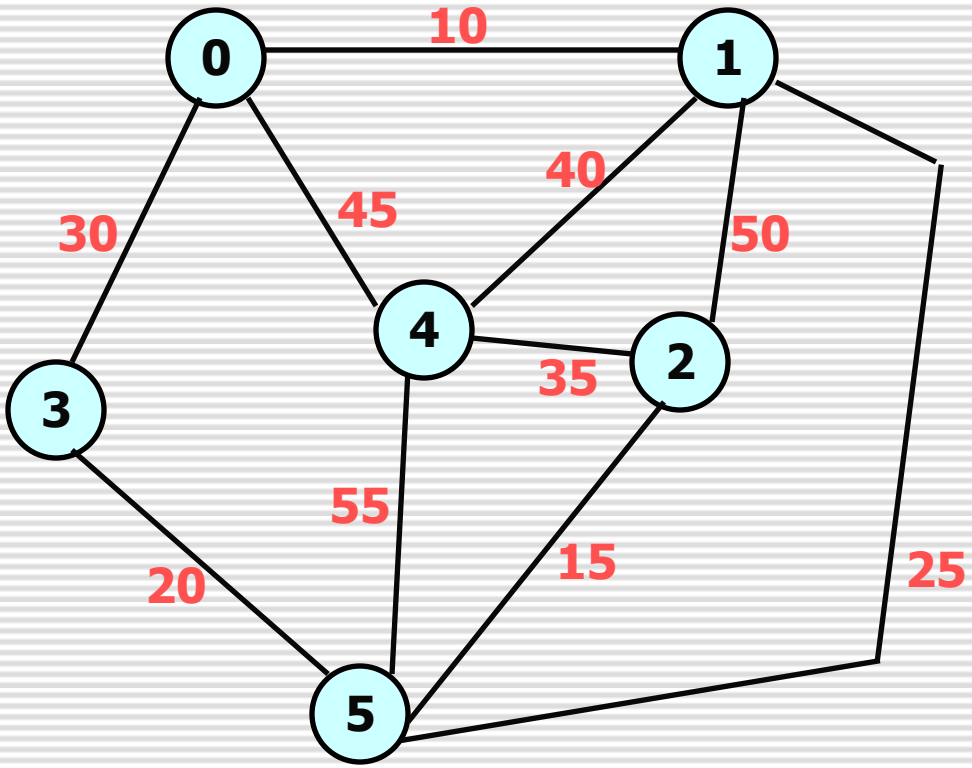
مشخص می کند که آیا گره انتخاب شده یا نه. ۱ یعنی انتخاب شده و ۰ یعنی انتخاب نشده

برای هر گره انتخاب نشده، شماره نزدیکترین گره انتخاب شده را مشخص می کند. برای گره های انتخاب شده مقدار این ستون ۱- می باشد.

برای هر گره انتخاب نشده، فاصله نزدیکترین گره انتخاب شده را مشخص می کند. برای گره های انتخاب شده مقدار این ستون ∞ می باشد.

درخت پوشای مینیمم / الگوریتم Prim

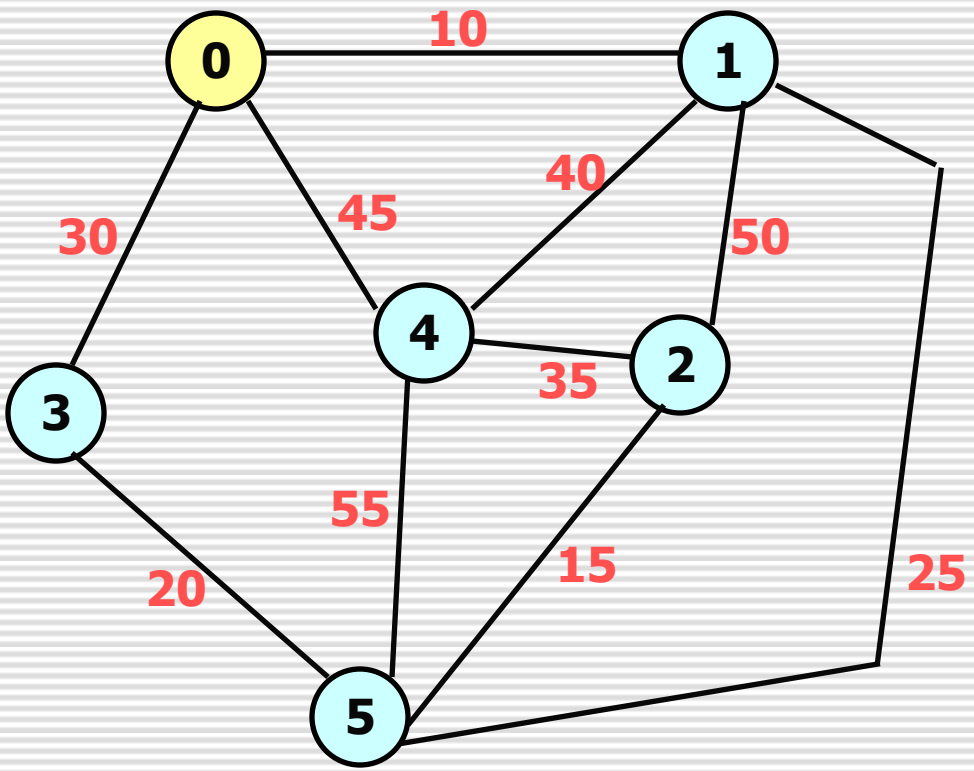
- مثال



درخت پوشای مینیمم / الگوریتم Prim

nodes	Y	near	minDist
0	1	-1	∞
1	0	0	10
2	0	0	∞
3	0	0	30
4	0	0	45
5	0	0	∞

Cost=0

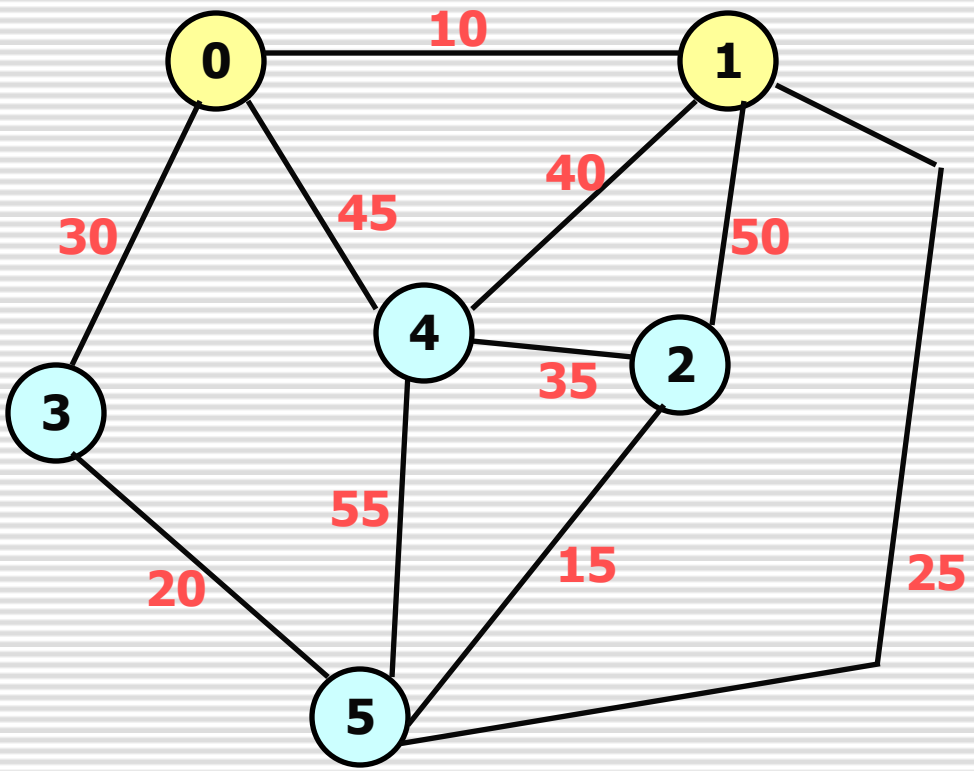


گره های زرد مشخص کننده گره های انتخاب شده هستند (گره های عضو مجموعه **Y**)

درخت پوشای مینیمم / الگوریتم Prim

nodes	Y	near	minDist
0	1	-1	∞
1	1	-1	∞
2	0	1	50
3	0	0	30
4	0	1	40
5	0	1	25

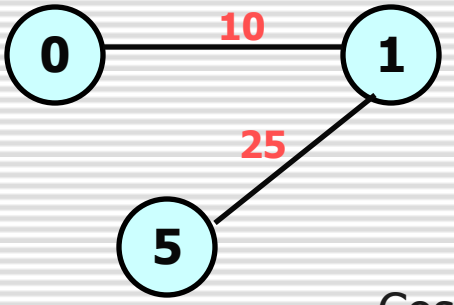
Cost=10



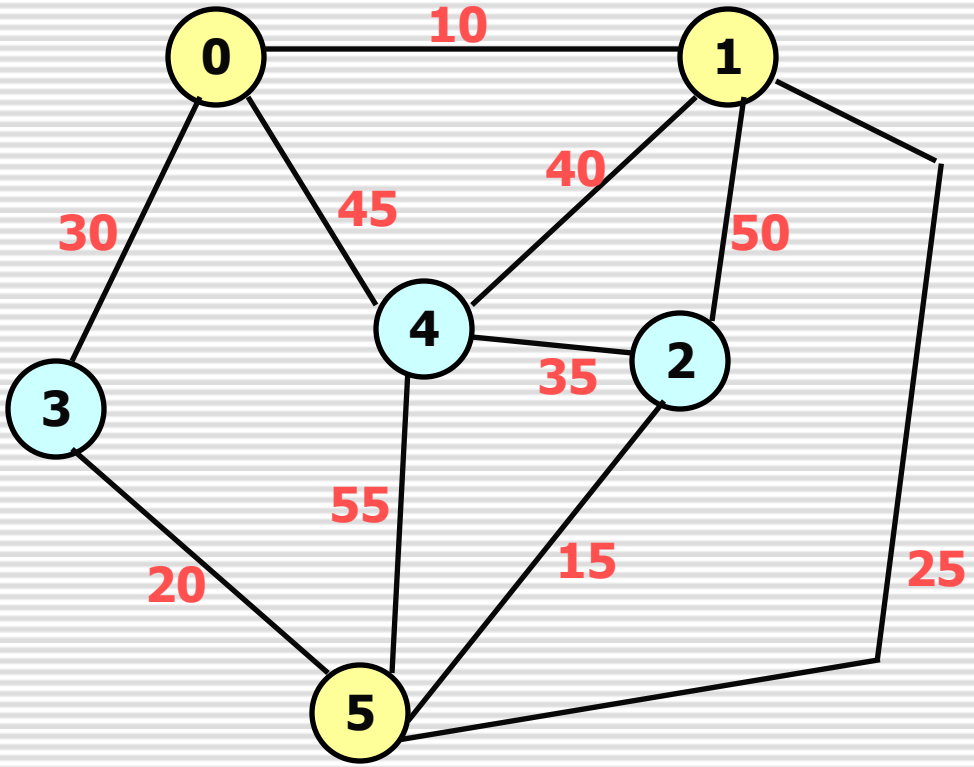
گره های زرد مشخص کننده گره های انتخاب شده هستند (گره های عضو مجموعه **Y**)

درخت پوشای مینیمم / الگوریتم Prim

nodes	Y	near	minDist
0	1	-1	∞
1	1	-1	∞
2	0	5	15
3	0	5	20
4	0	1	40
5	1	-1	∞



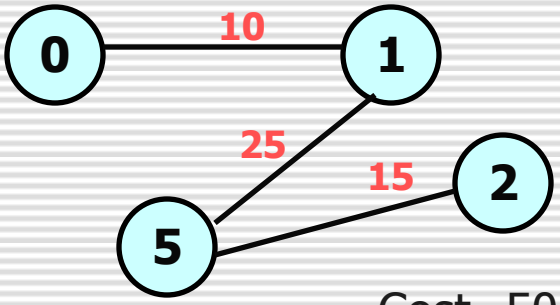
Cost=35



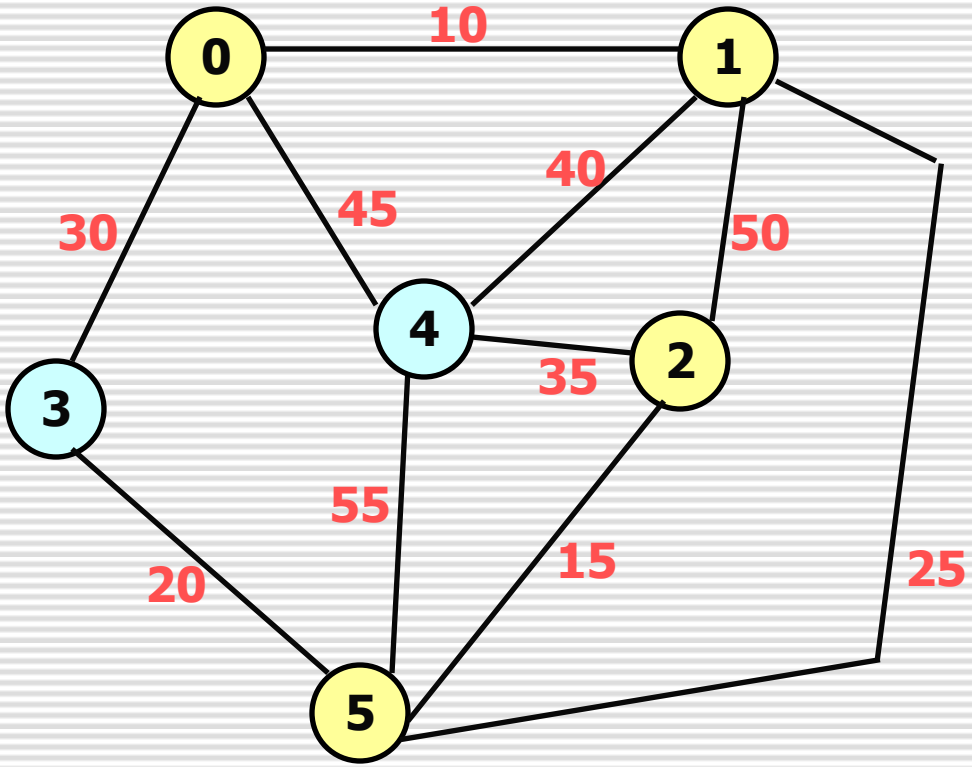
گره های زرد مشخص کننده گره های انتخاب شده هستند (گره های عضو مجموعه **Y**)

درخت پوشای مینیمم / الگوریتم Prim

nodes	Y	near	minDist
0	1	-1	∞
1	1	-1	∞
2	1	-1	∞
3	0	5	20
4	0	2	35
5	1	-1	∞



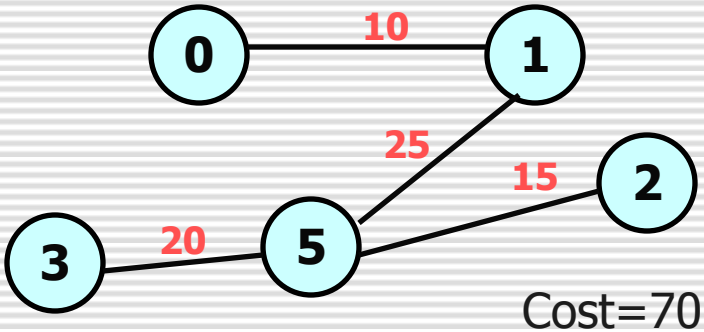
Cost=50



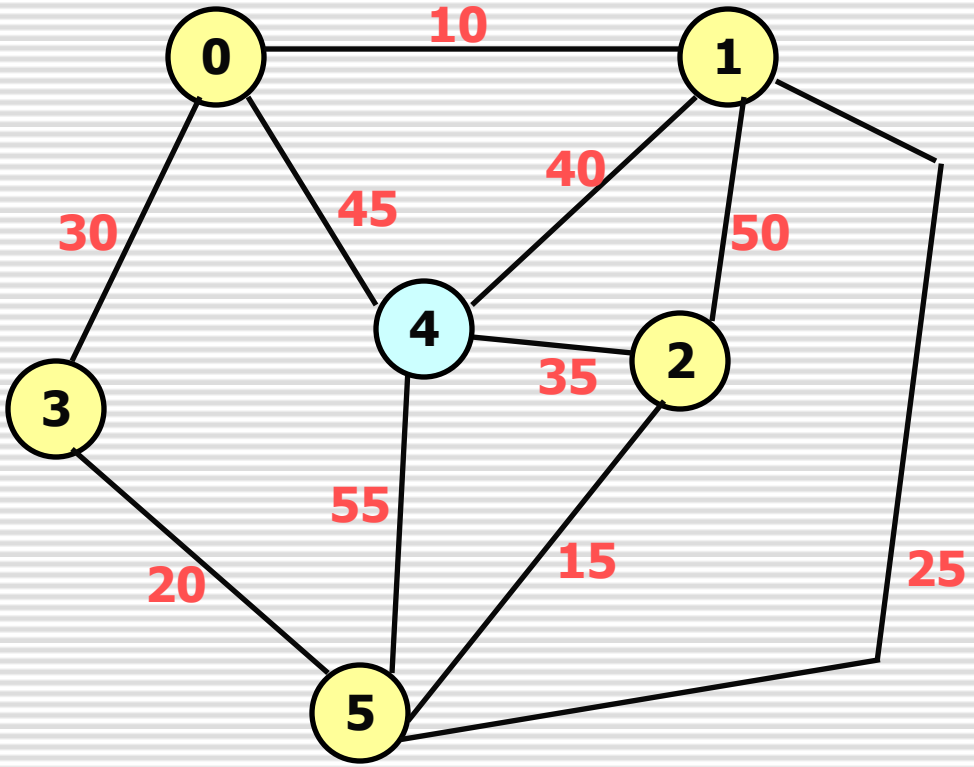
گره های زرد مشخص کننده گره های انتخاب شده هستند (گره های عضو مجموعه **Y**)

درخت پوشای مینیمم / الگوریتم Prim

nodes	Y	near	minDist
0	1	-1	∞
1	1	-1	∞
2	1	-1	∞
3	1	-1	∞
4	0	2	35
5	1	-1	∞



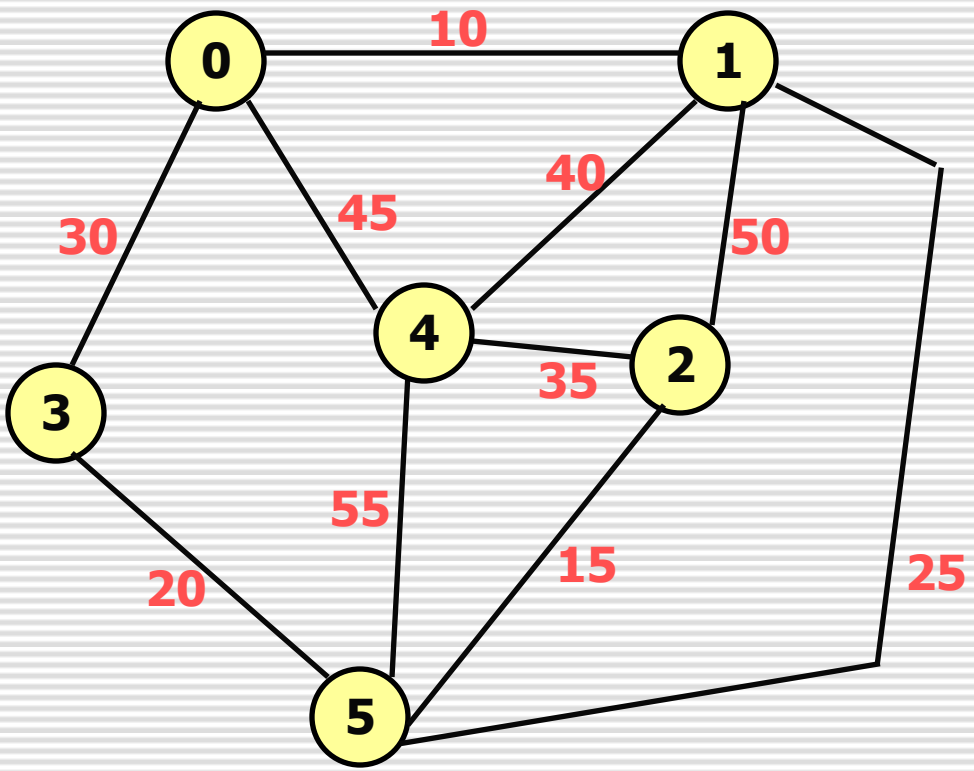
Cost=70

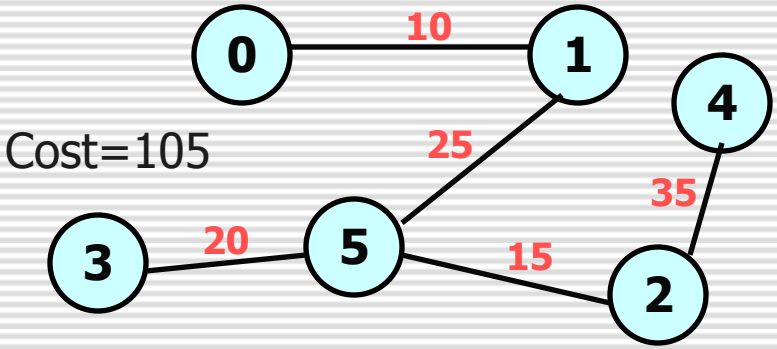


گره های زرد مشخص کننده گره های انتخاب شده هستند (گره های عضو مجموعه **Y**)

درخت پوشای مینیمم / الگوریتم Prim

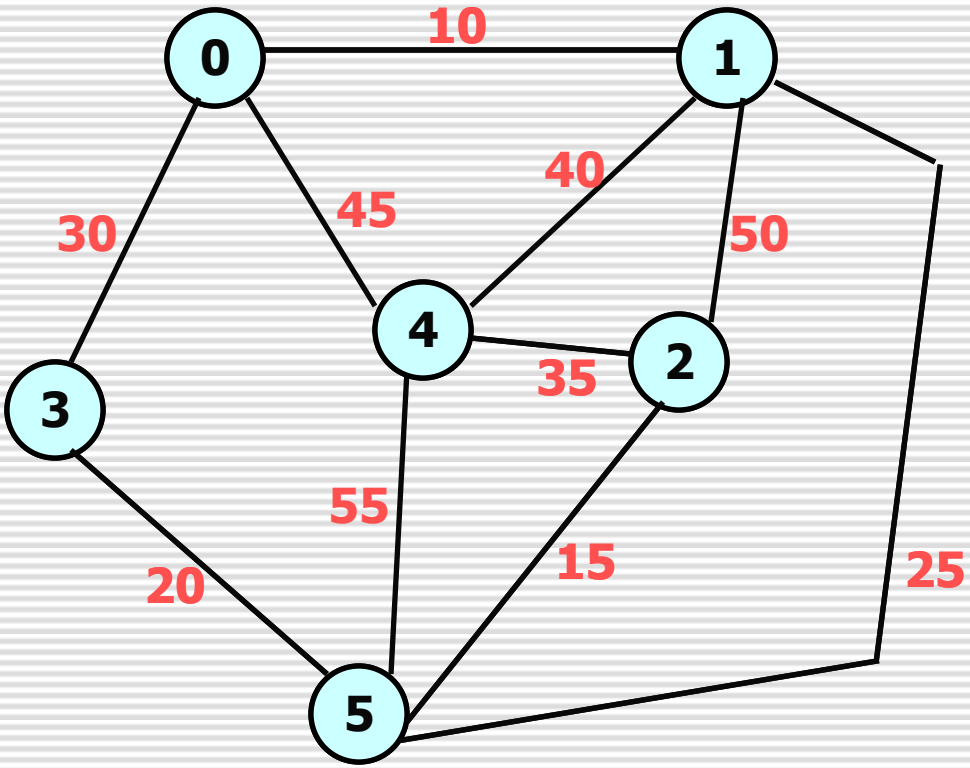
nodes	Y	near	minDist
0	1	-1	∞
1	1	-1	∞
2	1	-1	∞
3	1	-1	∞
4	1	-1	∞
5	1	-1	∞





گره های زرد مشخص کننده گره های انتخاب شده هستند (گره های عضو مجموعه **Y**)

درخت پوشای مینیمم / الگوریتم Prim



G

MST(G)

درخت پوشای مینیمم / الگوریتم Prim

- محاسبه مرتبه زمانی به روش ذهنی (n : تعداد گره های گراف)
- الگوریتم شامل ۲ فاز است:
- فاز ۱ (initialization): یک گره (V1) را بعنوان گره شروع (به دلخواه) انتخاب می کنیم و برای هر یک از n سطر جدول
  - مقدار ستون Y را تعیین می کنیم. برای گره V1 برابر ۱ و برای بقیه گره ها برابر -۱ ← O(1)
  - مقدار ستون near را تعیین می کنیم. برای گره V1 برابر 1- و برای بقیه گره ها برابر V1 ← O(1)
  - مقدار ستون minDist را تعیین می کنیم. با یک lookup در ماتریس مجاورت. ← O(1)
- پس فاز ۱ ← O(n)

درخت پوشای مینیمم / الگوریتم Prim

- فاز ۲: n-1 مرتبه روال زیر را تکرار می کنیم:
  - ابتدا گره با کمترین مقدار minDist را انتخاب کرده (این کار با O(n) قابل انجام است) و مقدار ستون Y و ستون near و ستون minDist آن را اصلاح می کنیم (این کار با O(1) قابل انجام است).
  - سپس از ابتدای جدول، سطر به سطر پیمایش کرده و برای گره های انتخاب نشده، در صورت لزوم مقدار ستون های آن سطر را update می کنیم. این کار با O(n) قابل انجام است.
    - بررسی اینکه آیا یک سطر نیاز به Update دارد یا خیر با O(1) قابل انجام است.
- پس فاز ۲ با $O(n^2)$ قابل انجام است.

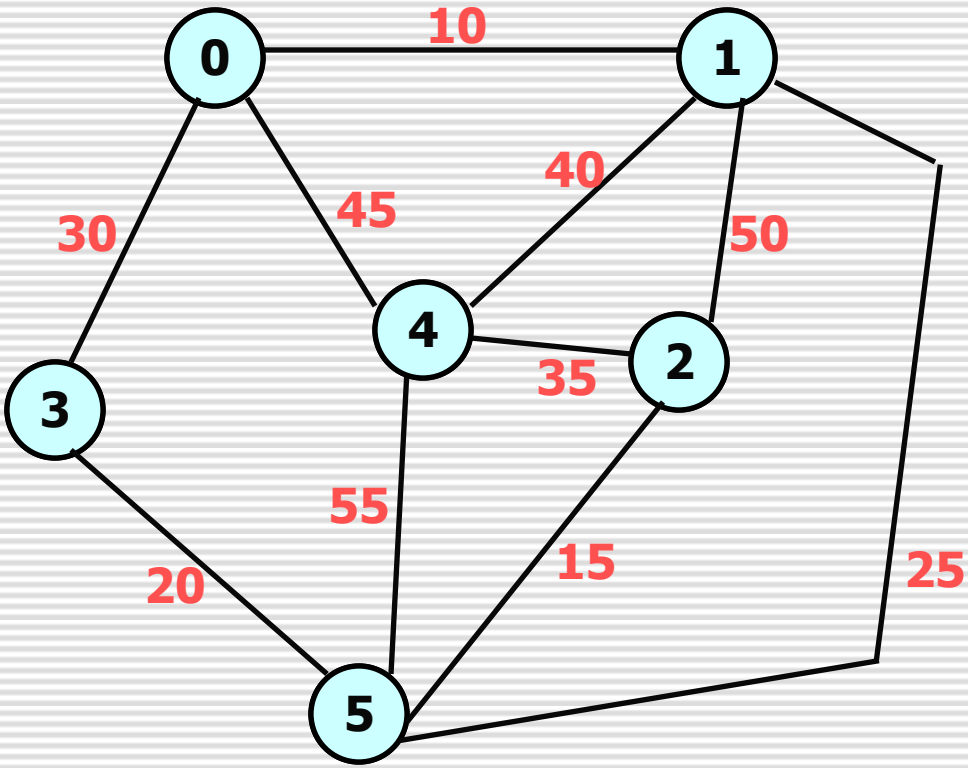
درخت پوشای مینیمم / الگوریتم Prim

- پس بطور کلی مرتبه زمانی الگوریم prim را $O(n^2)$ می باشد.

درخت پوشای مینیمم / الگوریتم Kruskal

- روش کلی
  - برای گراف دارای n گره، MST شامل n-1 یال خواهد بود (چون درخت است).
  - یالها را بر حسب وزنشان بطور غیرنزولی مرتب می کنیم.
  - ابتدا گراف را در قالب n مجموعه که هر یک دارای یک گره می باشند در نظر می گیریم.
  - یالها را از ابتدای لیست مرتب، یکی یکی بررسی می کنیم. در هر مرحله
    - اگر یال مورد بررسی، دو مجموعه جدا از هم را به هم متصل می کند، آن را بعنوان یک یال موجود در MST در نظر می گیریم و دو مجموعه مربوط به ابتدا و انتهای یال را به هم متصل می کنیم.
  - این کار را ادامه می دهیم تا n-1 یال انتخاب شوند.

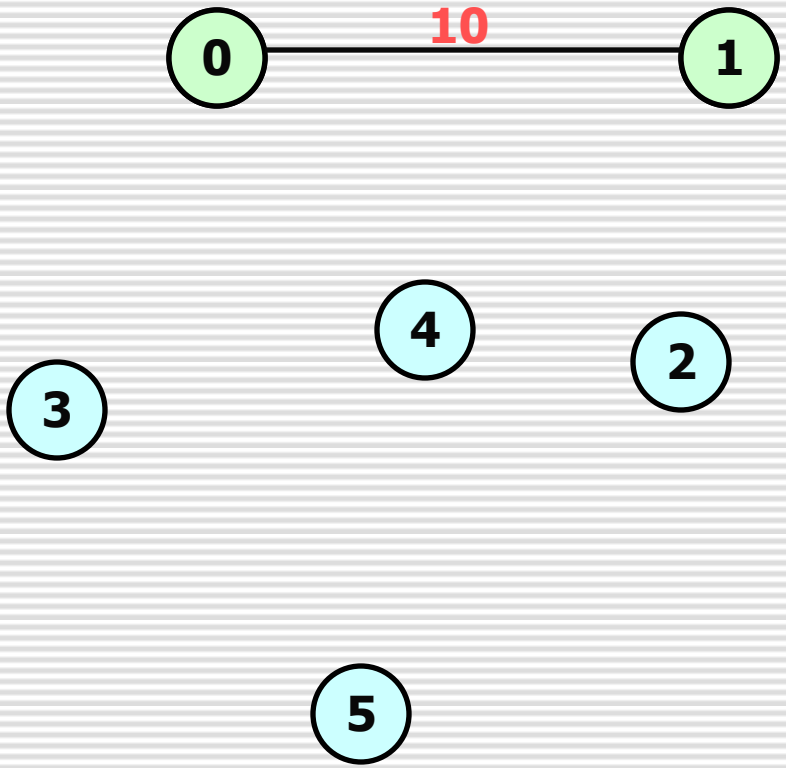
درخت پوشای مینیمم / الگوریتم Kruskal



Sorted list of edges

edge	cost
(0,1)	10
(2,5)	15
(3,5)	20
(1,5)	25
(0,3)	30
(2,4)	35
(1,4)	40
(0,4)	45
(1,2)	50
(4,5)	55

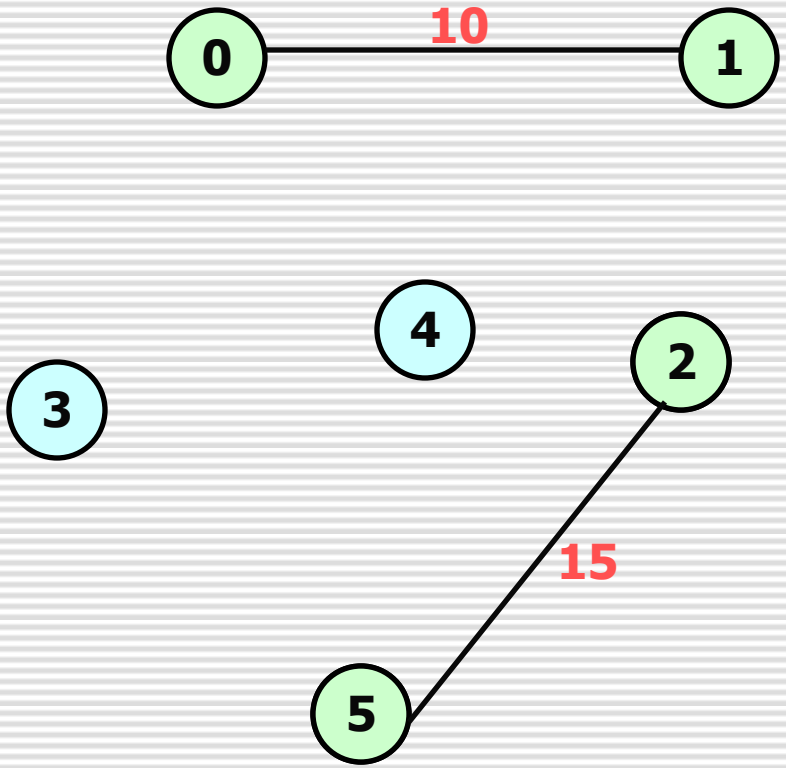
درخت پوشای مینیمم / الگوریتم Kruskal



Sorted list of edges

edge	cost
(0,1)	10
(2,5)	15
(3,5)	20
(1,5)	25
(0,3)	30
(2,4)	35
(1,4)	40
(0,4)	45
(1,2)	50
(4,5)	55

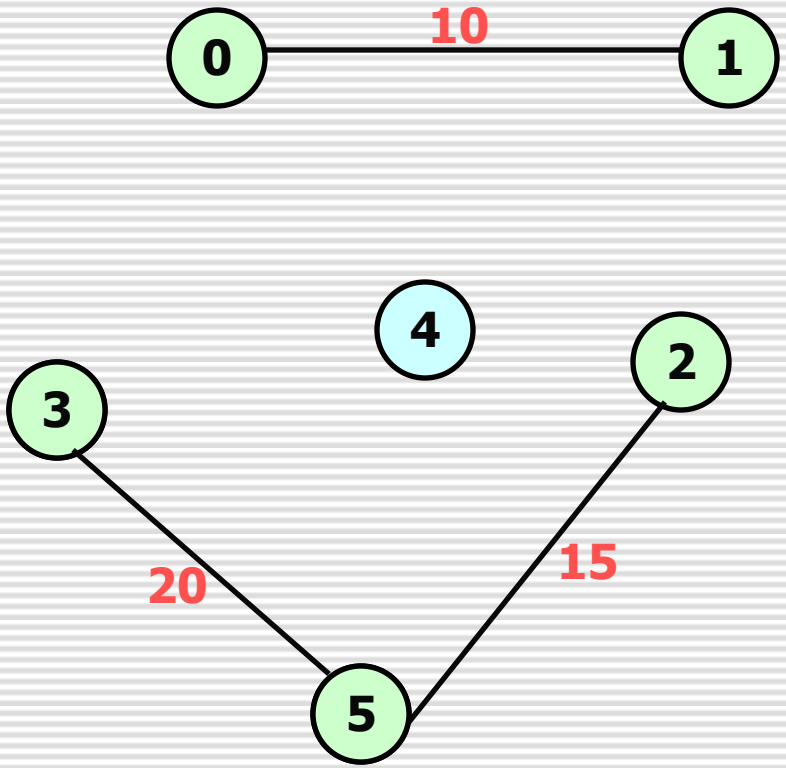
درخت پوشای مینیمم / الگوریتم Kruskal



Sorted list of edges

edge	cost
(0,1)	10
(2,5)	15
(3,5)	20
(1,5)	25
(0,3)	30
(2,4)	35
(1,4)	40
(0,4)	45
(1,2)	50
(4,5)	55

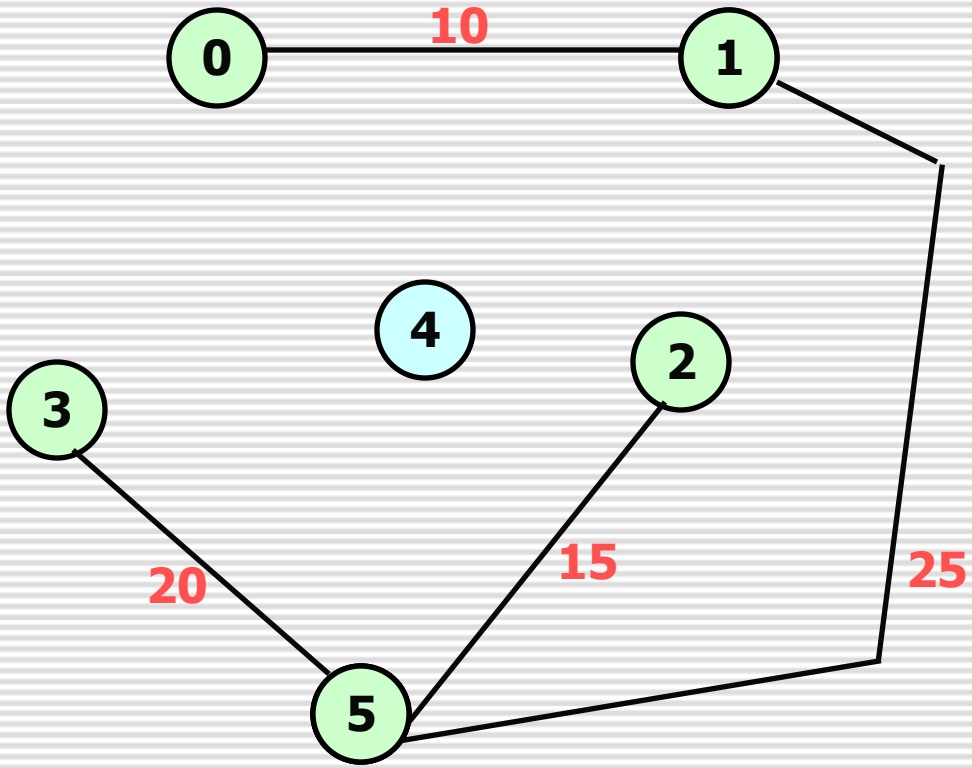
درخت پوشای مینیمم / الگوریتم Kruskal



Sorted list of edges

edge	cost
(0,1)	10
(2,5)	15
(3,5)	20
(1,5)	25
(0,3)	30
(2,4)	35
(1,4)	40
(0,4)	45
(1,2)	50
(4,5)	55

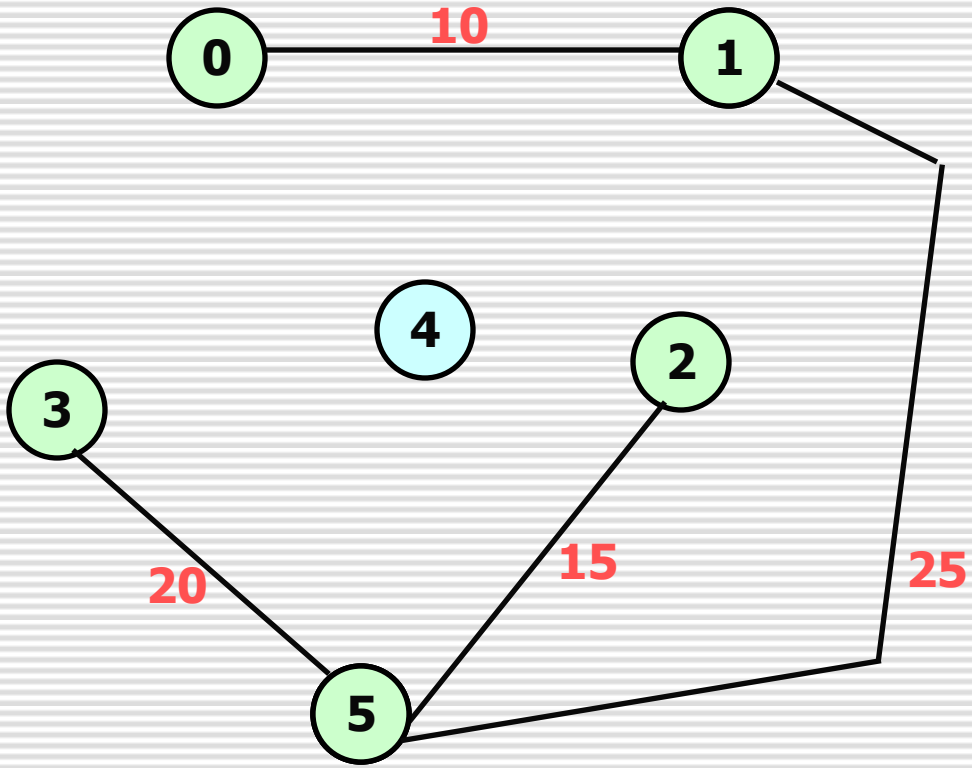
درخت پوشای مینیمم / الگوریتم Kruskal



Sorted list of edges

edge	cost
(0,1)	10
(2,5)	15
(3,5)	20
(1,5)	25
(0,3)	30
(2,4)	35
(1,4)	40
(0,4)	45
(1,2)	50
(4,5)	55

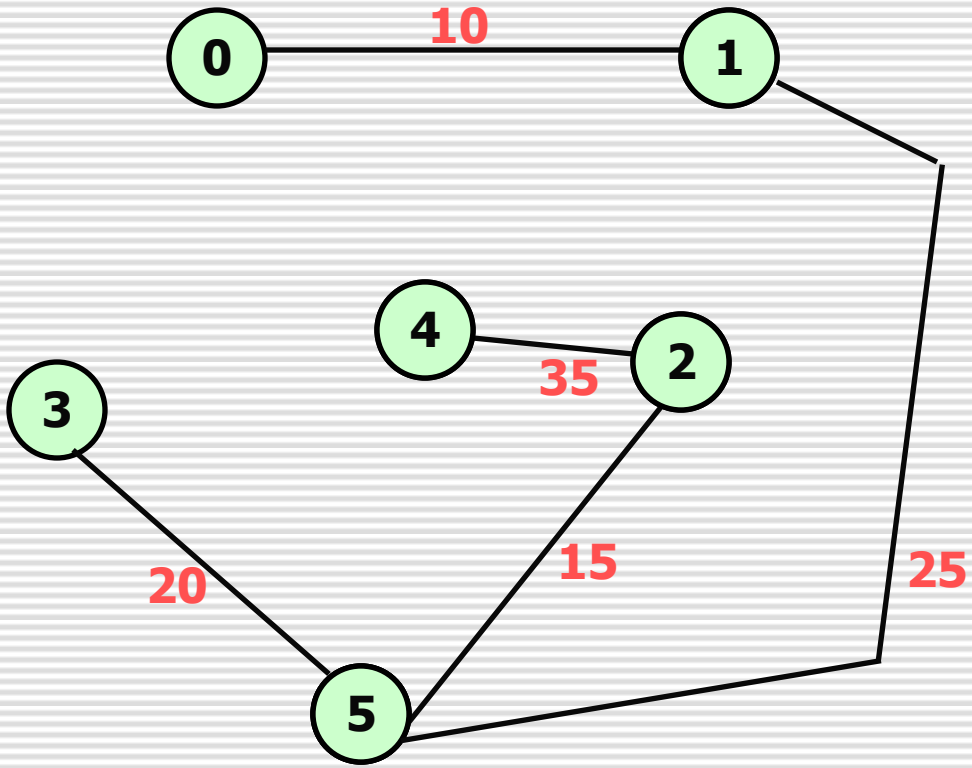
درخت پوشای مینیمم / الگوریتم Kruskal



Sorted list of edges

edge	cost
(0,1)	10
(2,5)	15
(3,5)	20
(1,5)	25
(0,3)	30
(2,4)	35
(1,4)	40
(0,4)	45
(1,2)	50
(4,5)	55

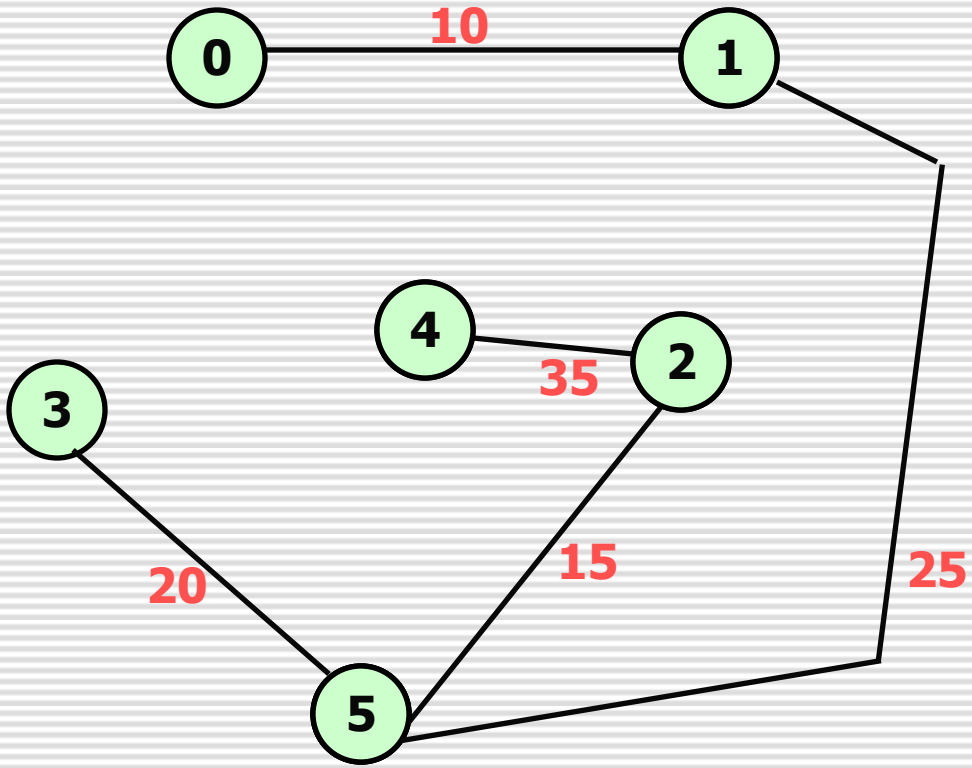
درخت پوشای مینیمم / الگوریتم Kruskal



Sorted list of edges

edge	cost
(0,1)	10
(2,5)	15
(3,5)	20
(1,5)	25
(0,3)	30
(2,4)	35
(1,4)	40
(0,4)	45
(1,2)	50
(4,5)	55

درخت پوشای مینیمم / الگوریتم Kruskal



Cost of MST = 105

Sorted list of edges

edge	cost
(0,1)	10
(2,5)	15
(3,5)	20
(1,5)	25
(0,3)	30
(2,4)	35
(1,4)	40
(0,4)	45
(1,2)	50
(4,5)	55

درخت پوشای مینیمم / الگوریتم Kruskal

```
void kruskal(int n, int m) {
  edges = sortEdges(m);
  init_sets();
  selectedEdges = 0;     k=0;     MST=null;
  while(selectedEdges < n-1) {
      e = edges[ k++ ];
      i , j = source and dest of edge e;
      p = findSet(i);          q = findSet(j);
      if ( p != q ) {
            merge(p, q);
            add e to MST;    selectedEdges++;
}}}
```

- شبه کد الگوریتم
  - m: تعداد یالها
  - n: تعداد گره ها

درخت پوشای مینیمم / الگوریتم Kruskal

```
void kruskal(int n, int m) {
  edges = sortEdges(m);          O(m log m)
  init_sets();                   O(n)
  selectedEdges = 0;     k=0;     MST=null;
  while(selectedEdges < n-1) {
      e = edges[ k++ ];
      i , j = source and dest of edge e;
      p = findSet(i);          q = findSet(j);
      if ( p != q ) {                          O(log n)
           merge(p, q);          O(1)
           add e to MST;    selectedEdges++;
}}}
```

O(n log n)

- مرتبه زمانی بر حسب n و m

درخت پوشای مینیمم / الگوریتم Kruskal

$W(m,n) = O(n) + O(n \log n) + O(m \log m)$

- از آنجا که برای گراف همبند $m>=n-1$ پس

$W(m,n) = O(m \log m)$

- ضمنا از آنجا که $n-1 \le m \le n(n-1)/2$ می توان گفت
  - اگر تعداد یالها به حد پایین یعنی $n-1$ نزدیک باشد مرتبه زمانی به سمت $O(n \log n)$ میل می کند و اگر تعداد یالها به سمت حد بالا میل کند مرتبه زمانی به سمت $O(n^2 \log n)$ میل می کند.

- با توجه به اینکه مرتبه زمانی الگوریتم prim، $O(n^2)$ بود
  - هر چه گراف خلوت تر (sparse) باشد الگوریتم kruskal بهتر است و هر چه شلوغتر و متصل تر باشد، الگوریتم prim بهتر است.

درخت پوشای مینیمم / الگوریتم Kruskal

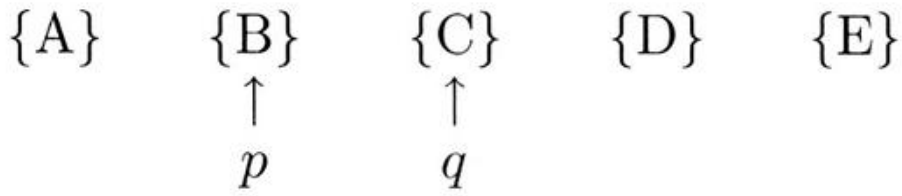
- مقایسه الگوریتم Kruskal و Prim
  - حاصل الگوریتم prim در هر مرحله یک درخت است ولی حاصل الگوریتم kruskal در هر مرحله یک جنگل است.

درخت پوشای مینیمم / الگوریتم Kruskal

- چرا مرتبه زمانی findSet را لگاریتمی و merge را O(1) در نظر گرفتیم؟
  - با استفاده از ساختمان داده خاصی با نام Disjoint Sets می توان اعمال فوق را طوری پیاده سازی نمود که چنین مرتبه زمانی هایی داشته باشند.

Disjoint Set

(a) There are five disjoint sets. We have executed $p = find(\mathrm{B})$ and $q = find(\mathrm{C})$.



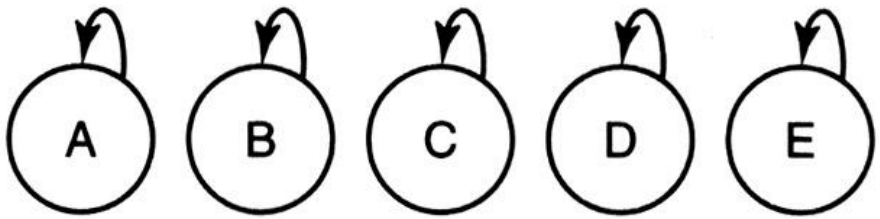
(b) There are four disjoint sets after {B} and {C} are merged.

(c) We have executed $p = find(\mathrm{B})$.

Disjoint Set

- یک روش مناسب برای نمایش: استفاده از شکل خاصی از درخت
  - گره های غیر ریشه، به والد خود اشاره دارند
  - گره ریشه، به خودش

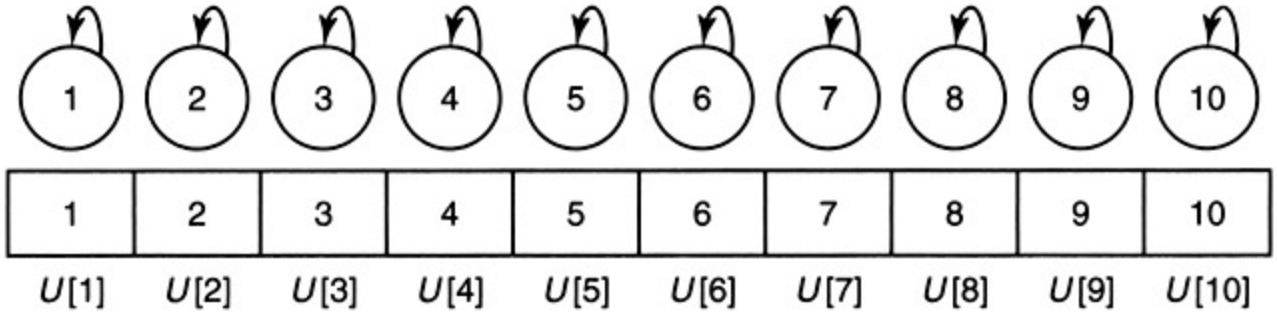
(a) Five disjoint sets represented by inverted trees.



Disjoint Set

- یک روش مناسب برای پردازش: استفاده از آرایه
- مقدار عنصر i ام آرایه، والد عنصر i ام را مشخص می کند

(a) The inverted trees and array implementation for 10 disjoint sets.

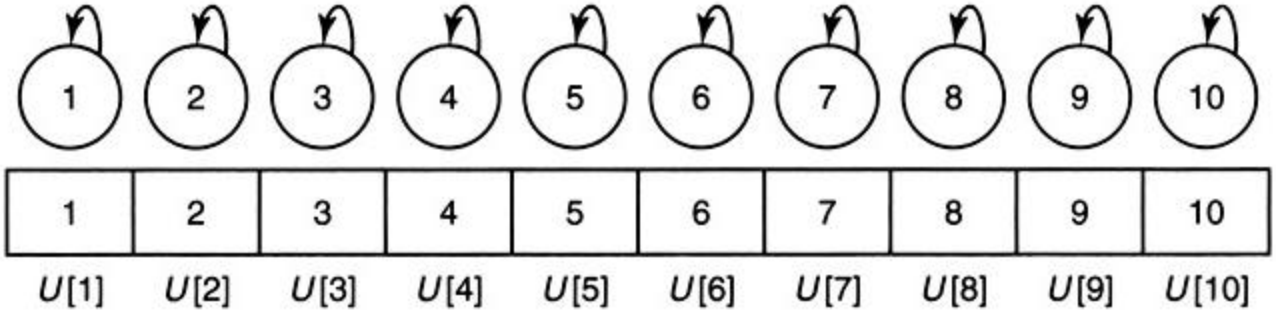


Disjoint Set

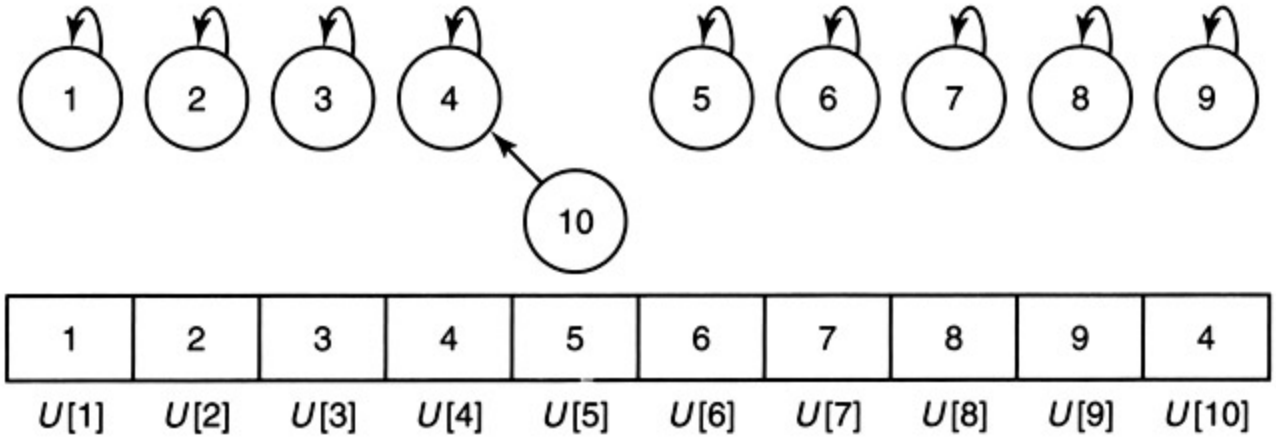
- روش اول برای ادغام دو مجموعه
  - مجموعه ای که مقدار ریشه اش از ریشه مجموعه دیگر بزرگتر است را مشخص کن
  - ریشه این مجموعه را بعنوان فرزند ریشه مجموعه دیگر قرار بده

Disjoint Set

(a) The inverted trees and array implementation for 10 disjoint sets.



(b) The sets {4} and {10} in part (a) have been merged. The 10th array slot now has value 4.



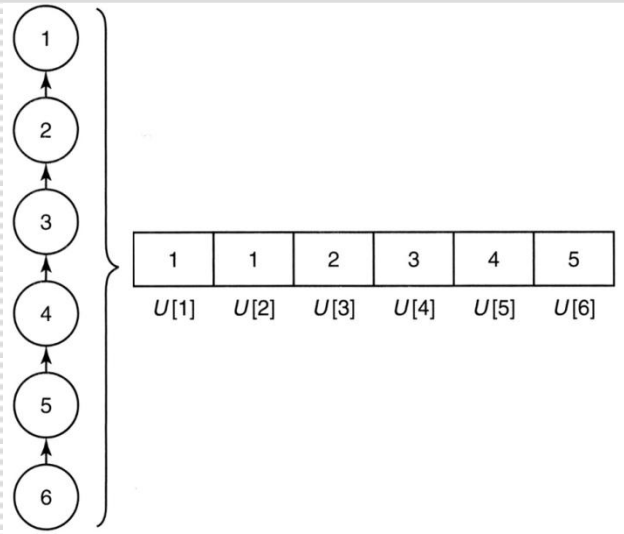
Disjoint Set

- اگر برای انجام عمل ادغام همواره ریشه دو مجموعه مورد نظر را به تابع merge ارسال کنیم، واضح است که عمل merge با O(1) و فقط با اصلاح یک عنصر آرایه انجام می شود.

- بنابراین هر بار که میخواهیم مجموعه های مربوط به دو عنصر x و y را با هم ادغام کنیم، ابتدا ریشه مجموعه ای که x در آن قرار دارد، و ریشه مجموعه ای که y در آن قرار دارد را پیدا کرده (توسط تابع find) و سپس ریشه ها را به merge پاس می دهیم.

Disjoint Set

- روال find(i) ریشه مجموعه ای که شامل عنصر i است را برمیگرداند
- مثالی برای بدترین حالت
  - {1} {2} {3} {4} {5} {6}
  - merge({5}, {6});
  - merge({4}, {5, 6});
  - merge({3}, {4, 5, 6});
  - merge({2}, {3, 4, 5, 6});
  - merge({1}, {2, 3, 4, 5, 6});

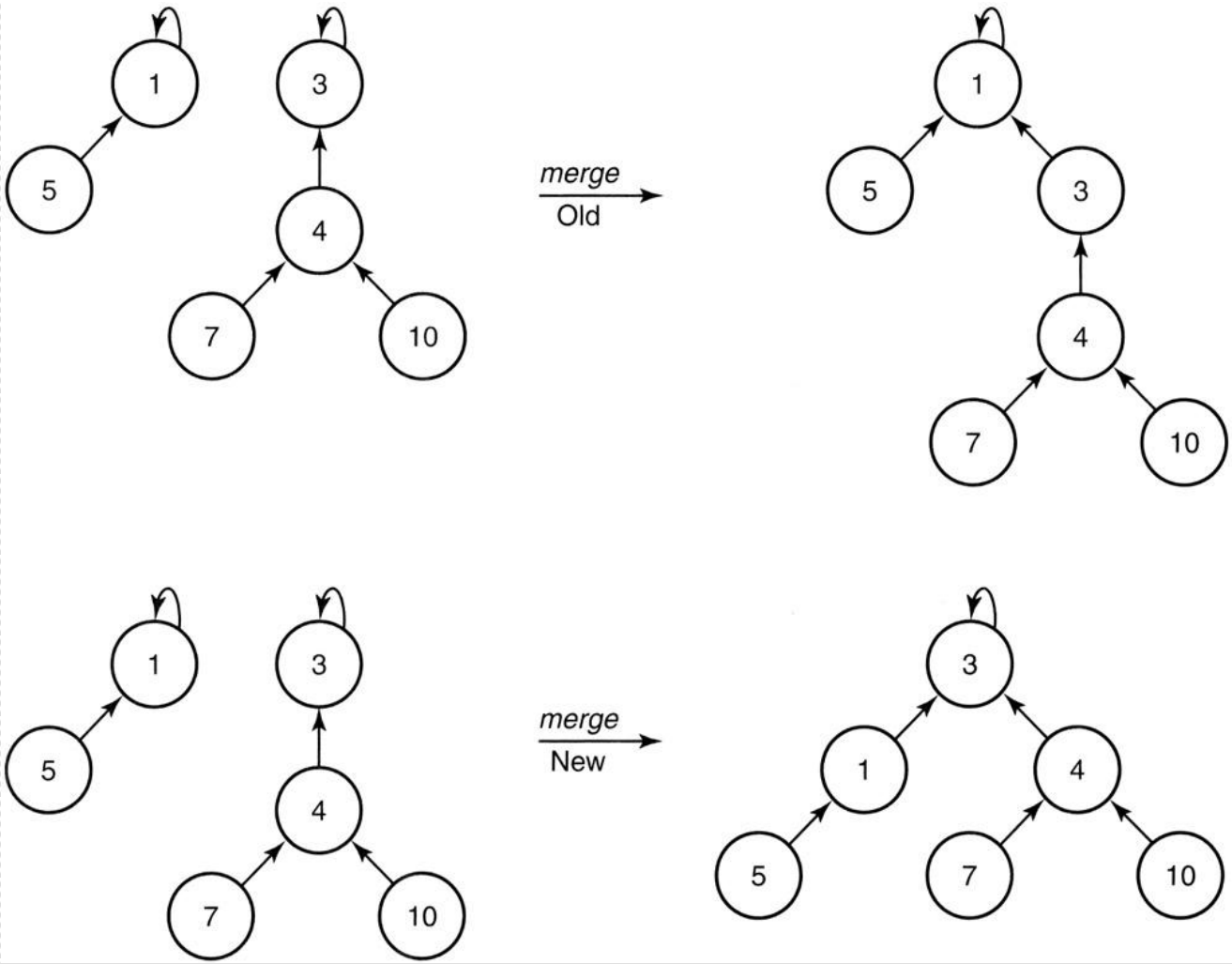


- در بدترین حالت برای پیدا کردن ریشه مجموعه شامل یک عنصر: O(n)

Disjoint Set

- روش دوم برای ادغام دو مجموعه
  - همواره برای هر مجموعه (هر درخت) عمق آن را نیز ذخیره کن
  - مجموعه ای که عمق کمتری دارد را مشخص کن.
  - ریشه این مجموعه را بعنوان فرزند ریشه مجموعه دیگر قرار بده

Disjoint Set



Disjoint Set

- با این تغییر مرتبه زمانی تابع find بهبود می یابد
- در بدترین حالت برای پیدا کردن ریشه مجموعه شامل یک عنصر: O(log n)

مطالب مورد بحث

- کوله پشتی غیر صفر و یک
- زمانبندی بهینه اجرای برنامه ها
- درخت پوشای مینیمم
  - الگوریتم Prim
  - الگوریتم Kruskal
- **زمانبندی بهینه فرآیندهای دارای ضرب الاجل**
- فشرده سازی با استفاده از کد Huffman

زمانبندی بهینه اجرای فرآیندهای دارای ضرب الاجل

- تعدادی برنامه داریم.
- اجرای هر برنامه ۱ واحد زمانی طول می کشد.
- هر برنامه دارای یک ضرب الاجل است.
  - وقتی می گوییم برنامه ای دارای ضرب الاجل ۲ است، یعنی آن برنامه فقط می تواند در واحد زمانی ۱ یا ۲ اجرا شود و نه دیرتر.
- اجرای هر برنامه دارای یک میزان سود است ، یعنی اگر آن برنامه را در ضرب الاجلش اجرا کنیم، آن سود را کسب خواهیم کرد.
- هدف: برنامه ها را طوری زمانبندی کنیم که اجرای آنها بر طبق آن زمانبندی، حداکثر سود را عایدمان کند.

زمانبندی بهینه اجرای فرآیندهای دارای ضرب الاجل

- مثال:
- زمانبندی های ممکن: (با رعایت ضرب الاجلها)
- [1,3] سود: 55
- [2,1] سود: 65
- [2,3] سود: 60
- [3,1] سود: 55
- [4,1] سود: 70
- [4,3] سود: 65

Job	Deadline	Profit
1	2	30
2	1	35
3	2	25
4	1	40

زمانبندی بهینه اجرای فرآیندهای دارای ضرب الاجل

- یک راه حل: بررسی کلیه زمانبندی های ممکن و انتخاب بهترین
  - قابل قبول نیست، چون مرتبه زمانی فاکتوریلی خواهد بود.
- راه حل دیگر: مبتنی بر روش حریصانه
  - آیا ویژگی های مسائل مشابه را دارد؟

زمانبندی بهینه اجرای فرآیندهای دارای ضرب الاجل

Job	Deadline	Profit
1	2	30
2	1	35
3	2	25
4	1	40

- تعریف
  - Feasible sequence: یک دنباله از بر
    ترتیب اجرا کنیم، ضرب الاجل تمام آنها ر
  - Feasible set: یک مجموعه از برنامه ها
    feasible sequence موجود باشد.
  - مثال: مجموعه {1,2} یک feasible set است چون [2,1] یک feasible sequence است.
  - مجموعه {2,4} feasible set نیست، چون هیچ feasible sequence ای برای آن موجود نیست (یعنی به هیچ ترتیب نمی توان دو برنامه ۲ و ۴ را اجرا کرد که ضرب الاجل هر دو رعایت شود.)

زمانبندی بهینه اجرای فرآیندهای دارای ضرب الاجل

- حدس: ابتدا برنامه ها را بر حسب ارزش، بطور غیرصعودی مرتب می کنیم.
- مجموعه جواب، ابتدا تهی است.
- سپس برنامه ها را از ابتدای لیست مرتب، تا انتها، یک به یک بررسی می کنیم
  - آیا با افزودن برنامه جاری به مجموعه جواب، حاصل یک feasible set است؟ اگر بله، برنامه را (به شکل خاصی که بعدا گفته می شود) به مجموعه جواب بیافزا.
- درستی این حدس را باید اثبات نمود.

زمانبندی بهینه اجرای فرآیندهای دارای ضرب الاجل

- مثال ۲: برنامه ها بر حسب ارزش مرتب شده اند (غیر صعودی)
  - جواب: تهی
  - آیا {1} یک feasible set است؟ بله
  - آیا {1, 2} یک feasible set است؟ بله
  - آیا {1,2,3} یک feasible set است؟ خیر
  - آیا {1,2,4} یک feasible set است؟ بله
  - آیا {1,2,4,5} یک feasible set است؟ خیر
  - آیا {1,2,4,6} یک feasible set است؟ خیر
  - آیا {1,2,4,7} یک feasible set است؟ خیر
  - جواب: [2,1,4] بر اساس مجموعه {1,2,4}

Job	Deadline	Profit
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

زمانبندی بهینه اجرای فرآیندهای دارای ضرب الاجل

- سوال: چرا برخی حالات را بررسی نمی کنیم؟
  - مثلا آیا بعد از اینکه دیدیم {1,2} یک feasible set است، نیاز به بررسی {1,3} نیست؟
  - چون ماکزیمم کردن سود هدف است، و چون {1,2} یک feasible set است، واضح است که حتی اگر {1,3} هم feasible set باشد، {1,2} برای ما اولویت دارد، چون سود حاصل از آن، بیشتر از {1,3} است.
  - سوال: خوب درست است که ارزش ۲ از ارزش ۳ بیشتر است، اما آیا ممکن نیست با انتخاب ۳ بجای ۲، در آینده بتوانیم برنامه های بیشتری را انتخاب کنیم و به ارزش بیشتری برسیم و تفاوت فوق از بین برود؟ اگر اینطور است پس باید حالتی مانند {1,3,4} را هم در نظر بگیریم. در حالیکه اینکار را نکردیم، آیا مشکلی پیش نمی آید؟

زمانبندی بهینه اجرای فرآیندهای دارای ضرب الاجل

- چطور بررسی می کنیم که آیا یک مجموعه، feasible set است یا خیر؟
  - یک راه حل: کلیه ترتیب های مختلف برنامه های آن مجموعه را در نظر بگیریم اگر حداقل یکی از آنها قابل قبول بود، جواب مثبت است.
  - این راه حل قابل قبول نیست چون مرتبه زمانی فاکتوریلی دارد.
- راه حل مناسب:
  - برنامه های مجموعه مورد بررسی را بر حسب ضرب الاجل بطور غیر نزولی مرتب می کنیم. سپس از ابتدای لیست مرتب بررسی می کنیم، اگر ضرب الاجل برنامه های لیست رعایت می شد، مجموعه قابل قبول است و در غیر اینصورت خیر (درستی این مطلب را باید اثبات نمود)

زمانبندی بهینه اجرای فرآیندهای دارای ضرب الاجل

- سوال: آیا هر بار که می خواهیم feasible بودن مجموعه جواب را بررسی کنیم، باید تمام عناصر آن را مرتب کنیم؟
- جواب: خیر. طوری عمل می کنیم که مجموعه جواب فعلی همواره مرتب باشد.
  - در ابتدا که مجموعه جواب تهی است، واضح است که مرتب است.
  - هر بار که می خواهیم عنصری را به مجموعه جواب (که مرتب است) بیافزاییم، فقط کافی است آن عنصر را در مکان مناسبش درج کنیم. مجموعه جواب باز هم مرتب خواهد بود.
- این کار چه ارزشی دارد؟

زمانبندی بهینه اجرای فرآیندهای دارای ضرب الاجل

```
solution Schedule(int n, int deadlines[], int profits[]) {
    sort jobs in non-increasing order of profit
    solution = null;
    for (int i=0; i<n; i++) {
        temp = insert job_i to solution based on non-
                                    decreasing order of deadlines
        if ( feasible(temp) )
            solution = temp;
    }
    return solution;
}
```

زمانبندی بهینه اجرای فرآیندهای دارای ضرب الاجل

- محاسبه مرتبه زمانی
- مرتبه زمانی مرتب سازی بر حسب ارزش: O(n logn)
- یک حلقه داریم با n بار تکرار. در مرحله i ام این حلقه،
  - برنامه i ام به solution افزوده می شود بطوری که برنامه های موجود در solution برحسب ضرب الاجل مرتب باشند. در نتیجه برای درج برنامه i ام در solution، حداکثر به i-1 مقایسه نیاز است تا جای مناسب برنامه i ام مشخص شود.
  - همچنین برای بررسی feasible بودن solution باید عناصر solution را که حداکثر i تا هستند، پیمایش کنیم.
- در نتیجه

زمانبندی بهینه اجرای فرآیندهای دارای ضرب الاجل

$$W(n) = T_{sort}(n) + \sum_{i=1}^{n} ((i-1)+i)$$

$$= T_{sort}(n) + n^2$$

$$W(n) = O(n^2)$$

زمانبندی بهینه اجرای فرآیندهای دارای ضرب الاجل

- برای درستی الگوریتم باید دو مطلب را اثبات کنیم:
  - روال بررسی feasible set درست عمل می کند.
  - جواب بدست آمده از الگوریتم بهینه است.
    - ارتباط این مساله با مساله کوله پشتی غیر صفر و یک

مطالب مورد بحث

- کوله پشتی غیر صفر و یک
- زمانبندی بهینه اجرای برنامه ها
- درخت پوشای مینیمم
  - الگوریتم prim
  - الگوریتم Kruskal
- زمانبندی بهینه فرآیندهای دارای ضرب الاجل
- **فشرده سازی با استفاده از کد Huffman**

فشرده سازی با استفاده از کد هافمن

- مفهوم فشرده سازی (compression)
- به یک روش encoding و یک روش decoding نیاز داریم.
  - با encryption و decryption اشتباه نشود.
- مزیت: صرفه جویی در هزینه ذخیره و ارسال
  - قالبهای zip.
  - سیستم فایل NTFS
  - Fax و Modem
  - Multimedia: JPEG، MPEG، GIF

فشرده سازی با استفاده از کد هافمن

- دو روش مختلف encoding
  - انکدینگ با طول ثابت: کد مربوط به تمام نویسه ها از نظر طول یکسان می باشند.
    - در سیستم ASCII برای هر کاراکتر ۸ بیت در نظر گرفته شده است.
  - انکدینگ با طول متغیر: کد مربوط به نویسه های مختلف ممکن است از نظر طول با هم یکسان نباشند.
    - کد Morse
- مزیت: می توان به نویسه هایی که تکرار بیشتری دارند، کد کوتاهتری نسبت داد تا در مجموع میزان فشرده سازی افزایش یابد.
- عیب: نیاز به درج جداکننده در زمان انکدینگ برای اینکه عمل دیکدینگ دچار ابهام و مشکل نگردد.

فشرده سازی با استفاده از کد هافمن

- روش هافمن یک روش انکدینگ با طول متغیر است که مشکل فوق را حل کرده است یعنی نیازی به درج جداکننده ندارد.
  - ۱۹۵۲
  - امروزه بعنوان یک روش انکدینگ back-end بطور ترکیبی با دیگر متدهای فشرده سازی (نظیر JPEG و MPEG) استفاده می شود.
  - کدهای بهینه تولید می کند.
  - سرعت encode و decode خوب است.

فشرده سازی با استفاده از کد هافمن

- مثال: متن اولیه شامل 8100 نویسه
- حجم متن در سیستم اسکی
  - برای هر نویسه 8 بیت ← 64800 بیت
- حجم متن در یک سیستم انکدینگ با طول ثابت
  - برای 6 نویسه مختلف ، کدی با طول 3 بیت کافی است
  - برای هر نویسه 3 بیت ← 24300 بیت
- حجم متن با استفاده از روش هافمن

(2000*2) + (1000*3) + (500*4) + (500*4) +
(2000*2) + (2100*2) = 19100 bits

جدول فراوانی حروف

نویسه	تعداد تکرار	کد مربوطه
a	2000	00
b	1000	101
c	500	1000
d	500	1001
e	2000	01
f	2100	11

codebook

نویسه	کد مربوطه

فشرده سازی با استفاده از کد هافمن

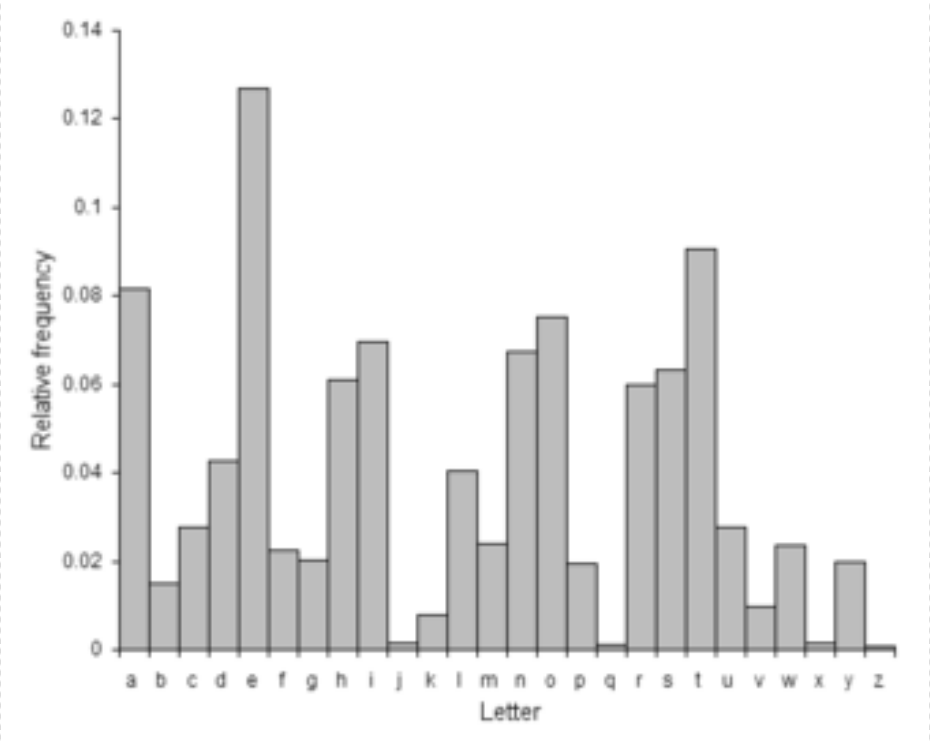
- در روش هافمن کد هیچ نویسه ای، پیشوند کد نویسه دیگری نمی باشد. در نتیجه نیازی به جدا کننده نمی باشد.
- اصطلاحا کدها پیشوند-آزاد می باشند (prefix-free code)
- مثلا با استفاده از codebook صفحه قبل می توان متن زیر را دیکد کرد:

نویسه	کد مربوطه
a	00
b	101
c	1000
d	1001
e	01
f	11

- Input: 010010001001
- Output: eacd

فشرده سازی با استفاده از کد هافمن

- روش هافمن برای بدست آوردن کد نویسه ها
  - محاسبه تعداد تکرار یا بسامدتمام نویسه های موجود در متن
    - یا استفاده از یک برآورد معتبر



فشرده سازی با استفاده از کد هافمن

- هر نویسه را بهمراه بسامد آن بعنوان ریشه یک درخت در نظر می گیریم.
  - پس در حالت اولیه یک جنگل خواهیم داشت.
- در هر مرحله
  - دو درختی که دارای کمترین بسامد در ریشه می باشند را انتخاب کرده
  - آن دو درخت را با هم ادغام می کنیم و درخت جدیدی می سازیم که بسامد ریشه آن، مجموع بسامد ریشه های دو درخت می باشد و درختی که بسامد ریشه اش کمتر بوده را زیردرخت چپ و درختی را که بسامد ریشه اش بیشتر بوده را زیردرخت راست قرار می دهیم. درخت جدید را جایگزین دو درخت اولیه می کنیم.
- این کار را تا زمانیکه تنها یک درخت باقی بماند ادامه می دهیم.

فشرده سازی با استفاده از کد هافمن

- مثال:
- متن ورودی شامل: a:30 b:10 c:7 d:8 e:40 f:14

فشرده سازی با استفاده از کد هافمن

- در درخت حاصل
  - نویسه ها در برگها قرار می گیرند (نشانه پیشوند آزاد بودن).
  - کد هر نویسه در مسیر ریشه تا آن نویسه نهفته است
    - حرکت به سمت چپ را معادل 0 و حرکت به راست را معادل 1 در نظر می گیریم.
- برای دیکدینگ یک رشته:
  - از ریشه حرکت می کنیم و با مشاهده هر 0 در ورودی به سمت چپ و با مشاهده هر 1 در ورودی به سمت راست حرکت می کنیم تا به برگ برسیم و نویسه موجود در آن برگ را به خروجی اضافه می کنیم و دوباره به ریشه رفته و کار را از آنجا ادامه می دهیم.

فشرده سازی با استفاده از کد هافمن

- مرتبه زمانی ایجاد درخت، با فرض داشتن بسامد کاراکترها
  - n: تعداد کاراکتر های مختلف متن (مثلا برای متن aaaabb مقدار n برابر ۲ است)
  - ایجاد یک درخت به ازای هر کاراکتر: O(1)
    - n کاراکتر داریم ← O(n)
  - مرتب کردن n درخت موجود برحسب بسامد ریشه، O(n log n)
  - یک حلقه که n-1 مرتبه اجرا می شود. در داخل حلقه:
    - انتخاب دو عنصر ابتدای لیست مرتب درختها، O(1)
    - ادغام دو درخت مذکور، O(1)
    - درج درخت حاصل در لیست، بطوریکه لیست باز هم مرتب باشد، O(n)
  - حلقه: $O(n^2)$
- نتیجه نهایی: $O(n^2)$

فشرده سازی با استفاده از کد هافمن

- مرتبه زمانی
  - در محاسبه مرتبه زمانی، فرض کردیم برای ذخیره درختها، از لیست یا آرایه استفاده می کنیم. در نتیجه مرتبه زمانی $O(n^2)$ بدست آمد.
  - اگر از ساختمان داده heap استفاده کنیم، می توانیم اعمال درج و حذف را با مرتبه زمانی لگاریتمی انجام دهیم و در نتیجه، مرتبه زمانی الگوریتم $O(n \log n)$ می شود.