به نام خدا

فصل چهارم

روش حریصانه در طراحی الگوریتم

The Greedy Approach

دانشکده مهندسی کامپیوتر - دانشگاه اصفهان
دکتر مرجان کائدی

الگوریتم حریصانه

- الگوریتم حریصانه با انجام یک سری انتخاب که هر یک در لحظه ای خاص بهترین به نظر می رسد، عمل می کند.
- یعنی هر انتخاب در جای خود بهینه به نظر می رسد.
- امید این است که یک حل بهینه سرتاسری یافته شود.
- ولی همیشه چنین نیست.

مثال: پرداخت بقیه پول

- پرداخت بقیه پول با حداقل ممکن تعداد سکه

- در آغاز هیچ سکه ای در مجموعه نداریم.

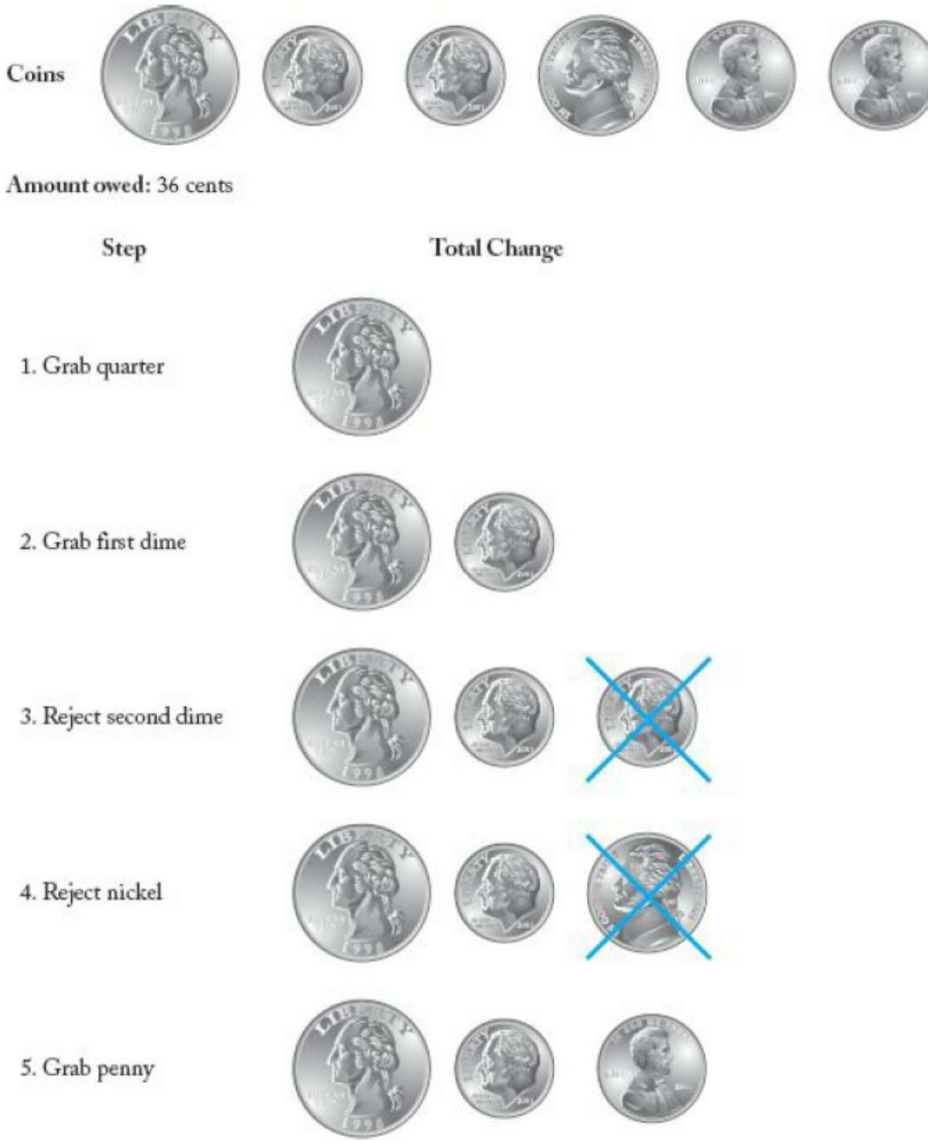
- هر بار بزرگ ترین سکه (از لحاظ ارزش) را انتخاب می کنیم (روال انتخاب)
- کنترل می کنیم که آیا با افزودن سکه به بقیه پول، جمع کل آنها از چیزی که باید باشد بیشتر می شود یا خیر (بررسی امکان سنجی)
- بررسی می کنیم که آیا مقدار جمع سکه ها با مقدار مورد نظر برابر شده است یا خیر؟ (بررسی راه حل)

```
while ( there are more coins and the instance is not solved){
 grab the largest remaining coin;            // selection procedure

 If (adding the coin makes the change exceed the amount owed)  // feasibility check
      reject the coin;
 else
      add the coin to the change;

 If (the total value of the change equals the amount owed)          // solution check
      the instance is solved;
}
```

یک مثال موفق:



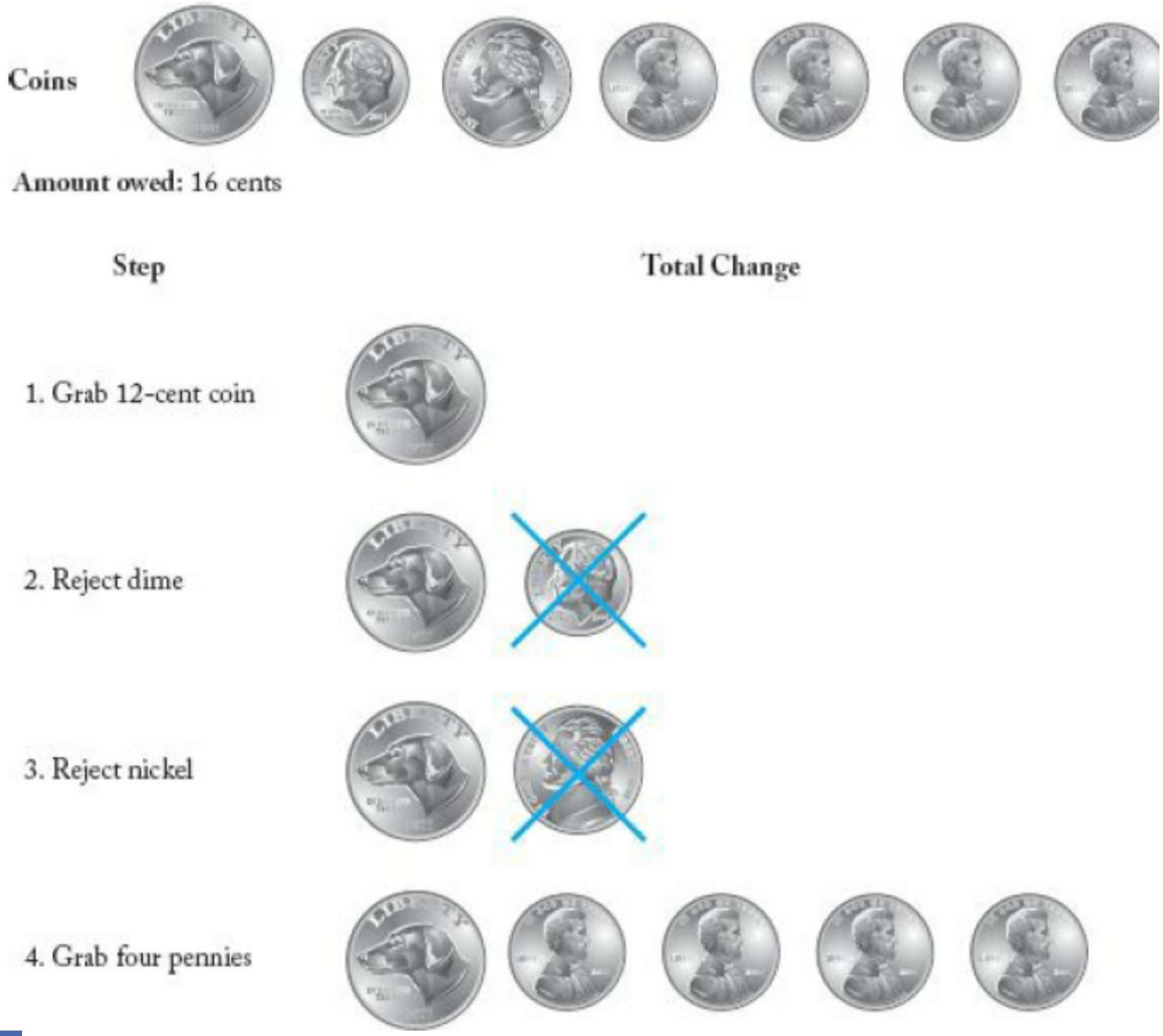
- سکه های موجود:
- ۱ سنتی، ۵ سنتی، ۱۰ سنتی، ۲۵ سنتی، نیم دلاری

- مبلغی که باید پس داده شود: ۳۶ سنت

- برای این سکه ها، همواره الگوریتم حریصانه بهترین راه حل را می دهد

- الگوریتم حریصانه است چون هر بار بزرگ ترین سکه بدون توجه به عواقب آن، انتخاب می شود.

یک مثال ناموفق



- اگر سکه ۱۲ سنتی را هم جزو سکه ها در نظر بگیریم.
- سکه های موجود:
- ۱ سنتی، ۵ سنتی، ۱۰ سنتی، ۱۲ سنتی، نیم دلاری

- الگوریتم به بهترین جواب نمی رسد.
- برای پرداخت ۱۶ سنت:
- بهترین جواب: یک ۱۰ سنتی، یک پنج سنتی و یک ۱ سنتی
- در حالیکه الگوریتم حریصانه پنج تا سکه انتخاب می کند.

مراحل یک الگوریتم حریصانه:

- روال انتخاب (***selection procedure***)
- بررسی امکان سنجی (***feasibility check***)
- بررسی راه حل (***solution check***)

مثال ۱: درخت پوشای کمینه
(minimum spanning tree)

تعریف مساله

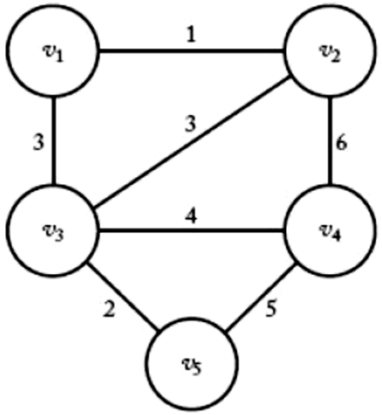
- انتخاب یال هایی از یک گراف بدون جهت وزن دار متصل، به نحوی که زیرگراف حاصل متصل باشد و تشکیل یک درخت بدهد که جمع وزن های آن حداقل باشد.

- کاربرد:
  - جاده سازی و متصل کردن شهرها به هم
  - ارتباطات راه دور (استفاده از حداقل کابل)
  - لوله کشی (استفاده از حداقل لوله)

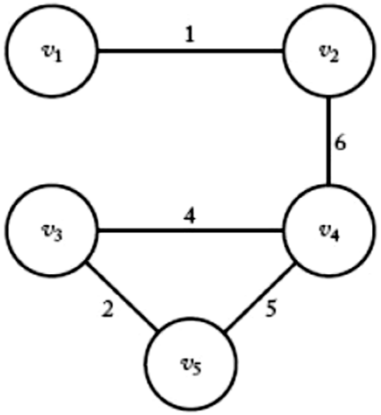
- درخت پوشا (spanning tree) برای گراف G، یک زیرگراف متصل است که حاوی همه رئوس موجود در G باشد و یک درخت باشد.

- درخت پوشایی که جمع وزن های آن حداقل باشد را درخت پوشای کمینه می گویند.

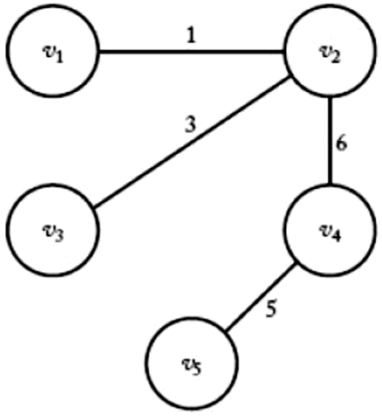
(a) A connected, weighted, undirected graph G.



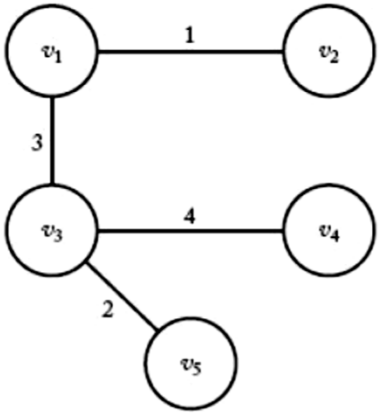
(b) If (v_4,v_5) were removed from this subgraph, the graph would remain connected.



(c) A spanning tree for G.



(d) A minimum spanning tree for G.



- مثال

$$V = \{v_1, v_2, v_3, v_4, v_5\}$$
$$E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_4), (v_3, v_4)$$
$$(v_3, v_5), (v_4, v_5)\}.$$

یک الگوریتم حریصانه سطح بالا

```
F = Ø                                   // Initialize set of edges to empty.
while (the instance is not solved){

      select an edge according to some locally optimal consideration; // selection procedure

      if (adding the edge to F does not create a cycle)
            add it;                                   // feasibility check

      if (T = (V, F) is a spanning tree)          // solution check
            the instance is solved;
}
```

- بسته به چگونگی قانون انتخاب، دو الگوریتم حریصانه برای یافتن درخت پوشای کمینه خواهیم داشت:
  - الگوریتم پریم
  - الگوریتم کروسکال

الف) الگوریتم پریم (Prim's Algorithm)

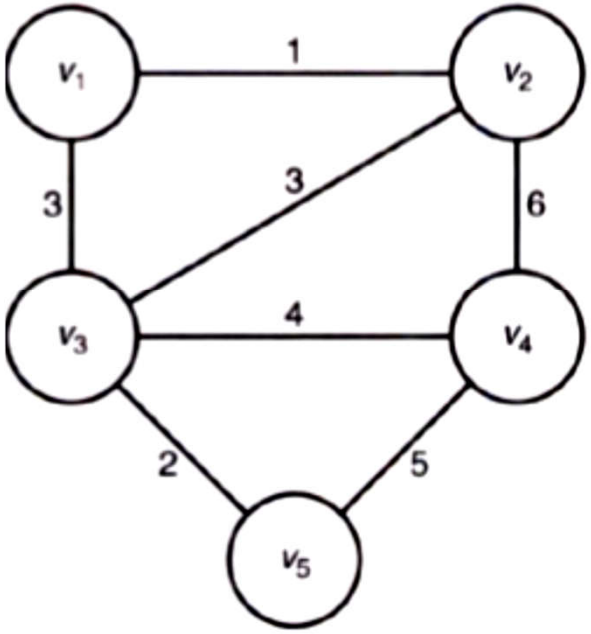
برای یافتن درخت پوشای کمینه

- با یک مجموعه خالی شروع می کنیم.

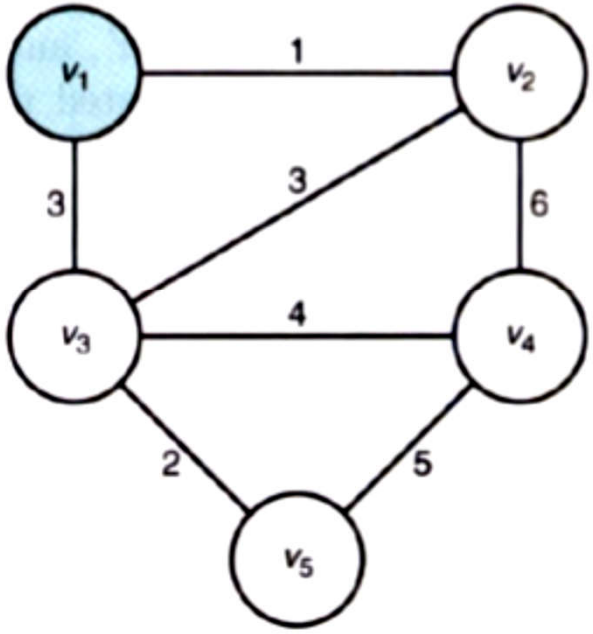
- هر بار کم وزن ترین یالی که به مجموعه انتخاب شده، متصل است، انتخاب می شود و به مجموعه اضافه می شود به شرطی که شرایط مورد نظر برآورده شود.

مثال

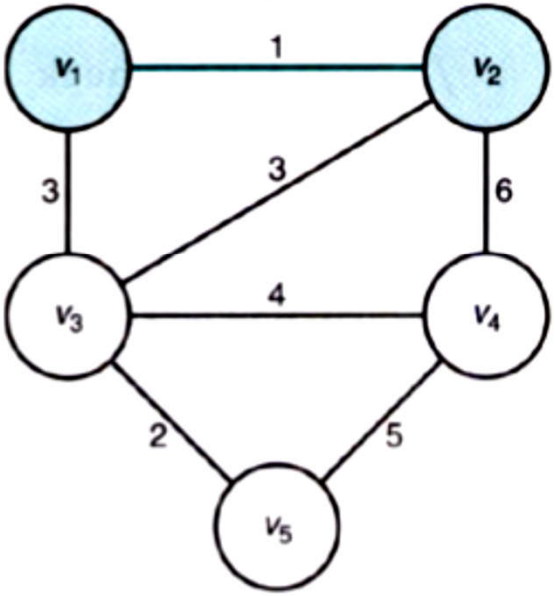
Determine a minimum spanning tree.



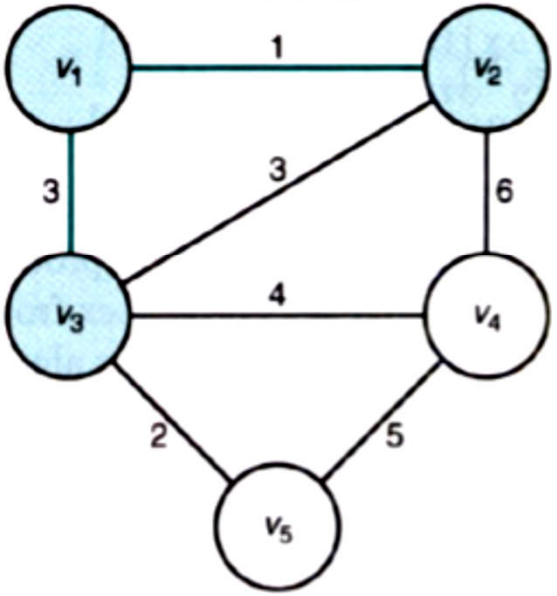
1. Vertex v_1 is selected first.



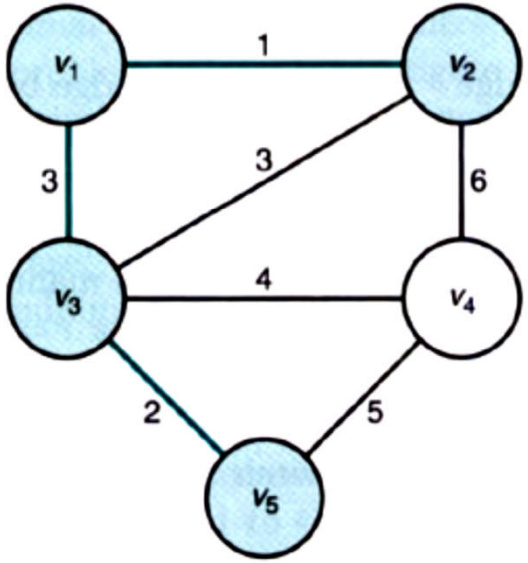
2. Vertex v_2 is selected because it is nearest to $\{v_1\}$.



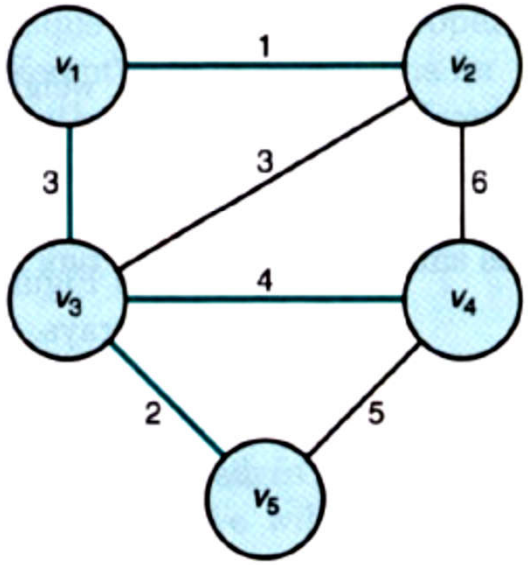
3. Vertex v_3 is selected because it is nearest to $\{v_1, v_2\}$.



4. Vertex v_2 is selected because it is nearest to $\{v_1, v_2, v_3\}$.



5. Vertex v_4 is selected.



الگوریتم پریم به صورت سطح بالا

```
F = Ø                    // Initialize set of edges to empty.
Y = {v1}                 // Initialize set of vertices to contain only the first one.

while (the instance is not solved){

    select a vertex in V - Y that is   // selection procedure and
    nearest to Y                       // feasibility check

    add the vertex to Y;
    add the edge to F;

     if (Y == V)                  // solution check
        the instance is solved;
}
```

الگوریتم پریم با ذکر جزئیات

$$W[i][j] = \begin{cases} \text{weight on edge} & \text{if there is an edge between } v_i \text{ and } v_j \\ \infty & \text{if there is no edge between } v_i \text{ and } v_j \\ 0 & \text{if } i = j. \end{cases}$$

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

nearest $[i]$ = index of the vertex in Y nearest to v_i

distance $[i]$ = weight on edge between v_i and the vertex indexed by ***nearest*** $[i]$

الگوریتم پریم با ذکر جزئیات

```
void prim (int n, const number W[] [], set_of_edges& F)
{
 index i, vnear;
 number min;
 edge e;
 index nearest [2 . . n];
 number distance [2 . . n];

 F = Ø;
 for (i = 2; i <= n; i++){
    nearest [i] = 1;              // For all vertices, initialize v1
    distance [i] = W[1] [i] ;     // to be the nearest vertex in
    }                             //Y and initialize the distance
                                  // from Y to be the weight
                                  // on the edge to v1.
repeat (n - 1 times) {  // Add all n - 1 vertices to Y.
   min = ∞;
   for (i = 2; i <= n; i++)              // Check each vertex for
     if (o ≤ distance [i] < min) {       // being nearest to Y.
        min = distance [i];
        vnear = i;
       }
   e = edge connecting vertices indexed by vnear and nearest [vnear];
    add e to F;            // Add vertex indexed by vnear to Y.
    distance [vnear] = - 1;
    for (i = 2; i <= n; i++)
         if (W[i] [vnear] < distance [i]){  // For each  vertex not in Y,
             distance[i] = W[i] [vnear];    // update its distance from Y.
              nearest [i] = vnear;
              }
   }
}
```

تحلیل پیچیدگی الگوریتم پریم در حالت معمول

- در حلقه repeat دو حلقه وجود دارد که هر یک n-1 بار تکرار می شود.
- اجرای دستورات داخل هر یک از آنها را می توان به عنوان عمل اصلی در نظر گرفت.

- چون حلقه repeat به تعداد n-1 بار تکرار می شود:

$$T(n) = 2\,(n-1)\,(n-1) \in \Theta\,(n^2).$$

- پس از نوشتن هر الگوریتم حریصانه، نیاز است اثبات شود که الگوریتم حریصانه بهترین راه حل را ارائه می کند.

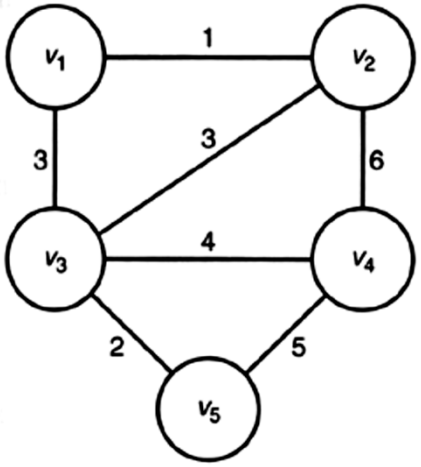
- اثبات برای الگوریتم پریم با استفاده از استقرا ...

ب) الگوریتم کروسکال (Kruskal's Algorithm)
برای یافتن درخت پوشای کمینه

- در این الگوریتم، ساخت درخت پوشا با مجموعه های مستقل از هم از راس ها شروع می شود.

- اگر یالی دو مجموعه مستقل را به هم متصل کند، آن یال به درخت اضافه می شود و دو زیرمجموعه با هم ادغام می شوند.

Determine a minimum spanning tree.



1. Edges are sorted by weight.

(v_1, v_2) 1

(v_3, v_5) 2

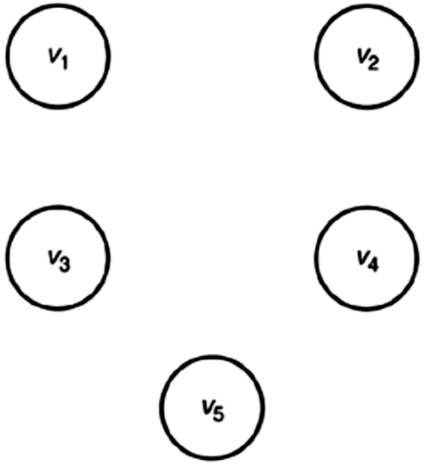
(v_1, v_3) 3

(v_2, v_3) 4

(v_4, v_5) 5

(v_2, v_4) 6

2. Disjoint set are created.



1. Edges are sorted by weight.

(v_1, v_2) 1

(v_3, v_5) 2

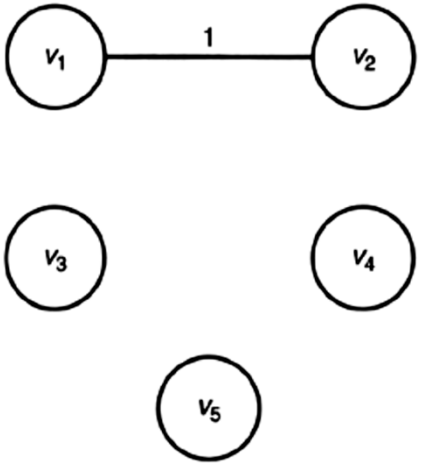
(v_1, v_3) 3

(v_2, v_3) 4

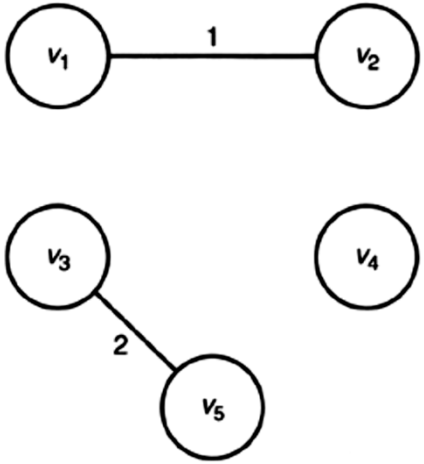
(v_4, v_5) 5

(v_2, v_4) 6

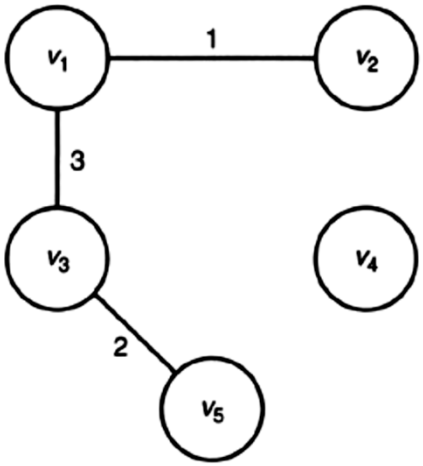
3. Edge (v_1, v_2) is selected.



4. Edge (v_3, v_5) is selected.



5. Edge (v_1, v_3) is selected.



1. Edges are sorted by weight.

(v_1, v_2) 1

(v_3, v_5) 2

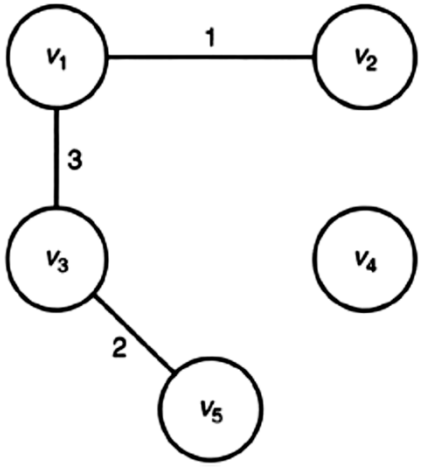
(v_1, v_3) 3

(v_2, v_3) 4

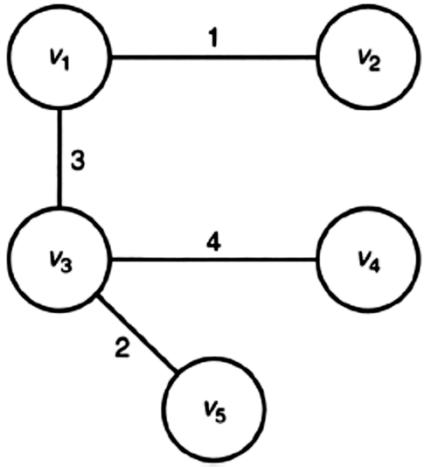
(v_4, v_5) 5

(v_2, v_4) 6

6. Edge (v_2, v_3) is selected.



7. Edge (v_3, v_4) is selected.



الگوریتم کروسکال به صورت سطح بالا

```
F = Ø                                   // Initialize set of edges to empty.

create disjoint subsets of V, one for each vertex and containing only that vertex;

sort the edges in E in nondecreasing order;
while (the instance is not solved){

     select next edge;                                  // selection procedure
     if (the edge connects two vertices in disjoint subsets){   // feasibility check
          merge the subsets;
          add the edge to F;
          }
     if (all the subsets are merged)                    // solution check
        the instance is solved;
}
```

الگوریتم کروسکال با ذکر جزئیات

```
void kruskal (int n, int m, set_of_edges E, set_of_edges& F)
{
 index i, j;
 set_pointer p, q;
 edge e;

Sort the m edges in E by weight in nondecreasing order;

F = Ø;
initial (n);            // Initialize n disjoint subsets.
while (number of edges in F is less than n - 1){

    e = edge with least weight not yet considered;
    i, j = indices of vertices connected by e;
    p = find(i);
    q = find(j);

    if (! equal(p, q)){
        merge(p, q);
        add e to F;
       }
 }
}
```

initial(n) initializes n disjoint subsets, each of which contains exactly one of the indices between 1 and n.

p = find(i) makes p point to the set containing index i.

merge(p, q) merges the two sets, to which p and q point

equal(p, q) returns true if p and q both point to the same set.

- اثبات اینکه الگوریتم کروسکال همواره کمینه ترین درخت پوشا را پیدا می کند، با استفاده از استقرا ...

تحلیل پیچیدگی زمانی الگوریتم کروسکال در بدترین حالت

- زمان مرتب سازی یال ها $W(m) \in \Theta(m \lg m)$.
- زمان برای مقداردهی به n مجموعه متمایز $T(n) \in \Theta(n)$.
- زمان در حلقه while، در بدترین حالت به تعداد یال ها یعنی m
- در مجموع، پیچیدگی در بدترین حالت: $W(m,n) \in \Theta(m \lg m)$.

تحلیل پیچیدگی زمانی الگوریتم کروسکال در بدترین حالت

- ممکن است به نظر برسد که بدترین حالت به n بستگی ندارد ولی در بدترین حالت، هر راس را می توان به هر یک از رئوس دیگر متصل کرد. یعنی:

$$m = \frac{n(n-1)}{2} \in \Theta\left(n^2\right).$$

- بنابراین می توان بدترین حالت را به صورت زیر نوشت:

$$w(m,n) \in \Theta\left(n^2 \lg n^2\right) = \Theta\left(n^2 2 \lg n\right) = \Theta\left(n^2 \lg n\right).$$

مقایسه الگوریتم پریم و کروسکال:

- Prim's Algorithm: $T(n) \in \theta(n^2)$
- Kruskal's Algorithm: $W(m, n) \in \theta(m \lg m)$

- در یک گراف متصل داریم: $n - 1 \leq m \leq \frac{n(n-1)}{2}$.

- برای گرافی که تعداد یال های آن یعنی m نزدیک به کران پایین باشد (یعنی تراکم کم یال ها)، پیچیدگی الگوریتم کروسکال $\theta(n \lg n)$ است یعنی سریع تر از از پریم است.
- برای گرافی که تعداد یال های آن یعنی m نزدیک به کران بالا باشد (یعنی گراف بسیار متصل باشد)، پیچیدگی الگوریتم کروسکال $\theta(n^2 \lg n)$ است یعنی الگوریتم پریم سریع تر است.

مثال۲ : الگوریتم دیکسترا (Dijkstra's Algorithm)

برای تعیین کوتاه ترین مسیر از مبداء واحد

شرح مساله

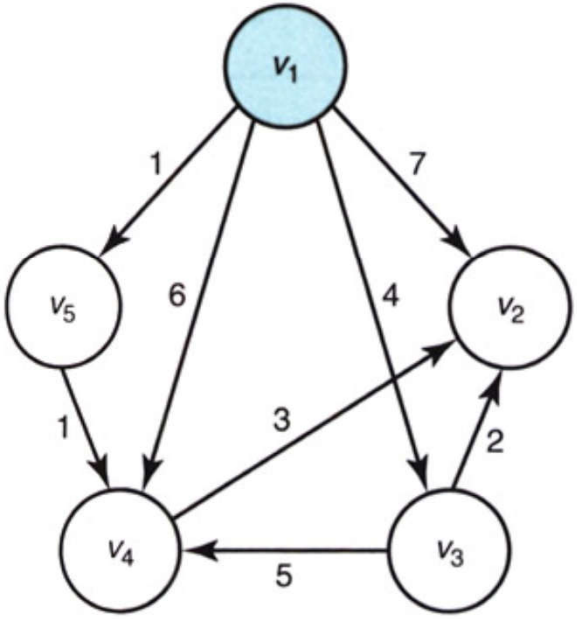
- الگوریتم فلوید یک الگوریتم به روش برنامه ریزی پویا برای یافتن کوتاه ترین مسیر بین هر دو راس یک گراف بود (پیچیدگی $\theta(n^3)$)

- ولی در اینجا کوتاه ترین مسیر از یک مبدا واحد به تک تک راس ها را نیاز داریم.
- الگوریتمی با مرتبه $\theta(n^2)$ برای این کار ارائه می دهیم.

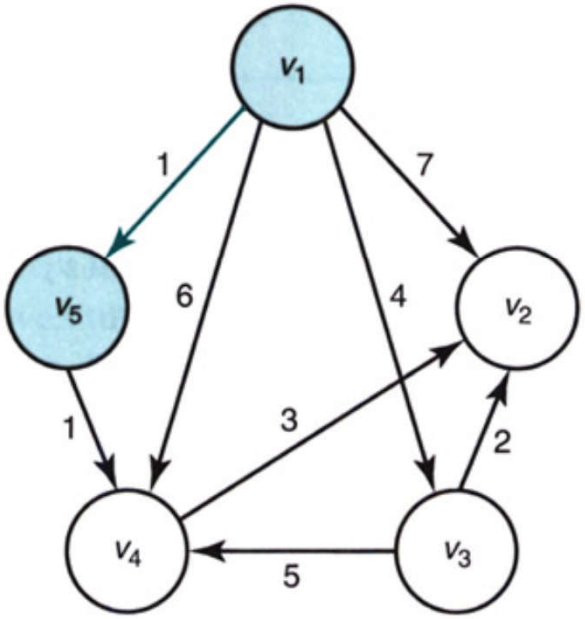
- این الگوریتم مقداری شبیه به الگوریتم پریم برای درخت پوشای کمینه است.
- فرض می کنیم از هر راس به راس های دیگر مسیری وجود دارد.

مثال

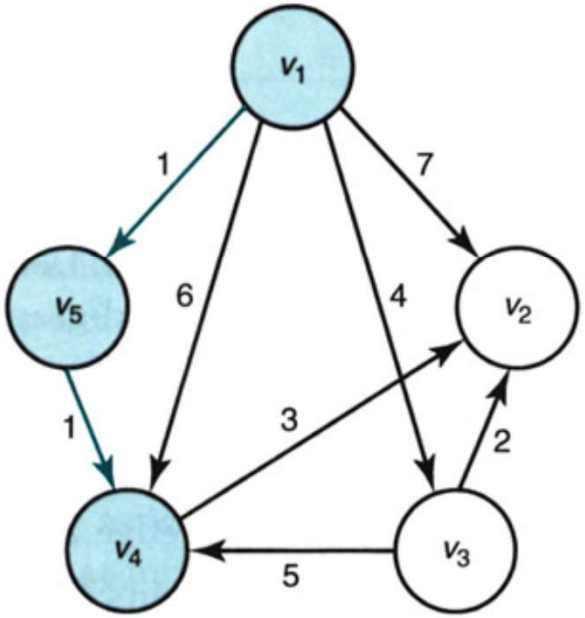
Compute shortest paths from v_1.



1. Vertex v_5 is selected because it is nearest to v_1.

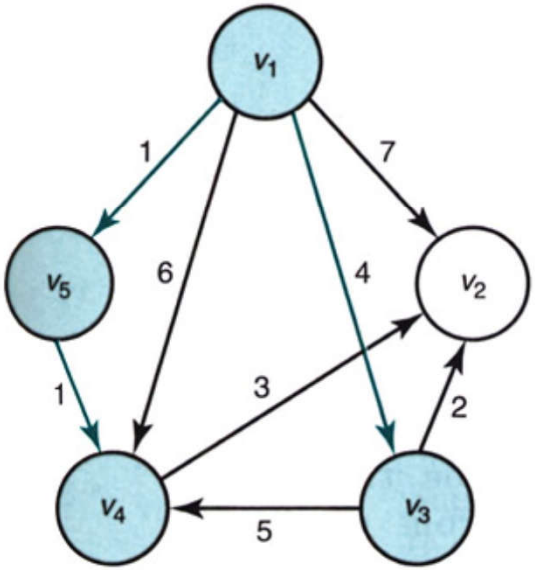


2. Vertex v_4 is selected because it has the shortest path from v_1 using only vertices in $\{v_5\}$ as intermediates.

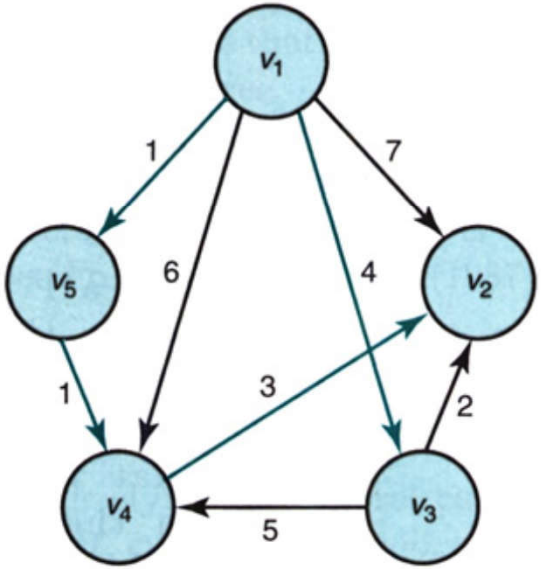


مثال (ادامه)

3. Vertex v_3 is selected because it has the shortest path from v_1 using only vertices in $\{v_4, v_5\}$ as intermediates.



4. The shortest path from v_1 to v_2 is $[v_1, v_5, v_4, v_2]$.



الگوریتم دیکسترا به صورت سطح بالا

```
Y = {v1};
F = Ø

while (the instance is not solved) {

    select a vertex v from V - Y, that has a          // selection procedure
    shortest path from v1, using only vertices        // and
    in Y as intermediates;                            // feasibility check

    add the new vertex v to Y;
    add the edge (on the shortest path) that touches v to F;

   if (Y == V)
      the instance is solved;          // solution check
}
```

الگوریتم دیکسترا با ذکر جزئیات

- *touch[i]* = index of vertex *v* in *Y* such that the edge <*v*, v_i> is the last edge on the current shortest path from v_1 to v_i using only vertices in *Y* as intermediates.

- *length[i]* = length of the current shortest path from v_1 to v_i using only vertices in *Y* as intermediates.

الگوریتم دیکسترا با ذکر جزئیات

```
void dijkstra (int n, const number W[][],  set_of_edges&F)
{
 index i, unear;
 edge e;
 index touch [2 .. n];
 number length [2 .. n];

 F = Ø;
 for (i = 2; i<= n; i++){                  // For all vertices, initialize v1
      touch [i] = 1;                       // to be the last vertex on the
      length [i] = W[1] [i];               // current shortest path from
}                                          // v1, and initialize length of
                                           // that path to be the weight
                                           // on the edge from v1.
```

الگوریتم دیکسترا با ذکر جزئیات (ادامه)

```
repeat (n - 1 times){                    // Add all n - 1 vertices to Y.
   min = ∞;
   for (i = 2; i < = n; i++)             // Check each vertex for
     if ( 0 ≤ length [i] < min) {        // having shortest path.
          min = length [i];
          vnear = i;
      }
   e = edge from vertex indexed by touch [vnear]
       to vertex indexed by vnear;
   add e to F;
   for (i = 2; i < = n; i++)
        if (length [vnear] + W[vnear] [i] < length [i]){
              length[i] = length[vnear] + W[vnear][i];
              touch[i] = vnear;          // For each vertex not in Y,
      }                                  // update its shortest path.
      length[vnear] = -1;                // Add vertex indexed by vnear to Y.
   }
}
```

- این الگوریتم یال های موجود در کوتاه ترین مسیرها را تعیین می کند.
- با تغییر کوچکی در الگوریتم می توان طول کوتاه ترین مسیرها را هم محاسبه کرد و در یک آرایه نگهداری کرد.

- اثبات اینکه الگوریتم دیکسترا همواره کوتاه ترین مسیرها را پیدا می کند، با استفاده از استقرا ...

تحلیل پیچیدگی

- مشابه الگوریتم پریم:

$$T(n) = 2(n-1)^2 \in \Theta(n^2).$$

مثال ۳: مساله های زمان بندی (Scheduling)

- **مساله اول: کمینه سازی زمان کل در سیستم**
  - انجام کار هر کدام از مشتریان به مدت مشخصی زمان نیاز دارد.
  - هدف: زمان بندی مشتریان به نحوی که کمترین زمان سیستم را داشته باشند.
  - زمان صرف شده برای انتظار کشیدن و نیز ارائه خدمات را زمان سیستم می گویند.

- **مساله دوم: زمانبندی با مهلت معین**
  - همه مشتریان به یک میزان زمان نیاز دارند.
  - ولی برای انجام هر یک از کارها، یک مهلت معین وجود دارد تا سود موردنظر به دست آید.
  - هدف زمان بندی کارها برای به حداکثر رساندن سود است.

مساله اول: کمینه سازی زمان کل در سیستم

- یک راه حل ساده: همه زمانبندی های ممکن را در نظر بگیریم و آن را که کمینه است انتخاب کنیم.

$$t_1 = 5, \qquad t_2 = 10, \qquad \text{and} \qquad t_3 = 4.$$

Job	Time in the System
1	5 (service time)
2	5 (wait for job 1) + 10 (service time)
3	5 (wait for job 1) + 10 (wait for job 2) + 4 (service time)

$$\underbrace{5}_{\text{Time for job 1}} + \underbrace{(5+10)}_{\text{Time for job 2}} + \underbrace{(5+10+4)}_{\text{Time for job 3}} = 39.$$

Schedule	*Total Time in the System*
[1, 2, 3]	$5 + (5 + 10) + (5 + 10 + 4) = 39$
[1, 3, 2]	$5 + (5 + 4) + (5 + 4 + 10) = 33$
[2, 1, 3]	$10 + (10 + 5) + (10 + 5 + 4) = 44$
[2, 3, 1]	$10 + (10 + 4) + (10 + 4 + 5) = 43$
[3, 1, 2]	$4 + (4 + 5) + (4 + 5 + 10) = 32$
[3, 2, 1]	$4 + (4 + 10) + (4 + 10 + 5) = 37$

- الگوریتمی که همه زمانبندی های ممکن را در نظر می گیرد از مرتبه فاکتوریل است.

- در حل مسائل، این روش که همه حالت های ممکن را ایجاد شوند و بعد بهترین حالت از بین آنها انتخاب شود، روش brute force نامیده می شود.

- راه حل حریصانه: قرار دادن کوتاه ترین کارها در ابتدا

یک الگوریتم حریصانه سطح بالا برای کمینه کردن زمان کل در سیستم

```
sort the jobs by service time in nondecreasing order;

while ( the instance is not solved){
  schedule the next job;            // selection procedure and
                                    // feasibility check
  if ( there are no more jobs)      // solution check
     the instance is solved;
}
```

$$W(n) \in \Theta(n \lg n).$$

تعمیم الگوریتم به مساله زمانبندی چند سرویس دهنده

- فرض می کنیم m سرویس دهنده داریم
- کارها به ترتیب غیرنزولی انتخاب می شوند
- سرویس دهنده اول، برای کار اول، سرویس دهنده دوم برای کار دوم، و..... و سرویس دهنده m ام برای کار m ام
- سرویس دهنده اول، کار خود را زودتر از همه به پایان می رساند (چون کار کوتاه تری داشته است)
- بنابراین کار m+1 ام را به سرویس دهنده اول اختصاص می دهیم و .....

- تخصیص کارها:

Server 1 serves jobs $1, (1+m), (1+2m), (1+3m), \ldots$
Server 2 serves jobs $2, (2+m), (2+2m), (2+3m), \ldots$
$\vdots$
Server i serves jobs $i, (i+m), (i+2m), (i+3m), \ldots$
$\vdots$
Server m serves jobs $m, (m+m), (m+2m), (m+3m), \ldots$

- ترتیب انجام کارها:

$$1, 2, \ldots, m, 1+m, 2+m, \ldots, m+m, 1+2m, \ldots$$

- اثبات بهینه بودن زمانبندی، به روش برهان خلف .....

مساله دوم: زمان بندی با مهلت معین

- هر کدام از کارها دارای مهلتی برای انجام شدن است
- هر کاری که در مهلت معین آن انجام شود، سود مورد نظر آن به دست می آید.

- هدف زمانبندی کارها است با این هدف که سود بیشینه به دست آید.
- لازم نیست همه کارها زمانبندی و انجام شوند.

مثال

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

- فرض می شود که هر یک از کارها در یک واحد زمانی انجام می شود.

Schedule	*Total Profit*
[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

- یک راه حل ساده: در نظر گرفتن همه زمانبندی ها
- ولی زمان انجام این کار از مرتبه فاکتوریل است.
- راه حل حریصانه: مرتب سازی غیرصعودی همه کارها بر اساس سود آنها و سپس وارسی هر کدام از کارها به ترتیب و در صورت امکان افزودن آن به زمانبندی
- یک ترتیب امکان پذیر: اگر همه کارهای موجود در آن ترتیب، در مهلت مقرر خود انجام شوند.
- هدف: یافتن یک ترتیب امکان پذیر، با سود کل بیشینه

یک الگوریتم حریصانه برای زمانبندی در مهلت معین

```
sort the jobs in nonincreasing order by profit;

S = Ø

while (the instance is not solved){

select next job;                    // selection procedure

    if (S is feasible with this job added)   // feasibility check
        add this job to S;
    if (there are no more jobs)          // solution check
        the instance is solved;
}
```

مثال

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

فرض می شود که هر یک از کارها در یک واحد زمانی انجام می شود.

1. S is set to $\varnothing$.
2. S is set to $\{1\}$ because the sequence [1] is feasible.
3. S is set to $\{1, 2\}$ because the sequence [2, 1] is feasible.
4. $\{1, 2, 3\}$ is rejected because there is no feasible sequence for this set. 5. S is set to $\{1, 2, 4\}$ because the sequence [2, 1, 4] is feasible.
6. $\{1, 2, 4, 5\}$ is rejected because there is no feasible sequence for this set.
7. $\{1, 2, 4, 6\}$ is rejected because there is no feasible sequence for this set.
8. $\{1, 2, 4, 7\}$ is rejected because there is no feasible sequence for this set.

یک روش برای تعیین امکان پذیر بودن یک مجموعه

- مجموعه کارها (مثلا مجموعه S) امکان پذیر است اگر و فقط اگر، ترتیب حاصل از مرتب شدن کارهای موجود در آن، بر اساس مهلت های غیرنزولی امکان پذیر باشد.

- مثلا: آیا {1,2,4,7} امکان پذیر است؟
- باید امکان پذیر بودن ترتیب زیر را بررسی کنیم:

$$[\ 2,\ 7,\ 1,\ 4\ ].$$
$$\uparrow\ \ \uparrow\ \ \uparrow\ \ \uparrow$$
$$1\ \ 2\ \ 3\ \ 3$$

مهات هر کار در زیر آن آمده است.

این ترتیب امکان پذیر نیست چون کار ۴ در مهلتش انجام نمی شود.

پس این مجموعه امکان پذیر نیست.

الگوریتم با ذکر جزئیات

- کارها از ابتدا بر اساس سودشان مرتب شده اند.

```
void schedule (int n, const int deadline [], sequence_of_integer& J)
{
 index i;
 sequence_of_integer K;

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

مثال

Job	*Deadline*
1	3
2	1
3	1
4	3
5	1
6	3
7	2

1. J is set to [1].
2. K is set to [2, 1] and is determined to be feasible.
   J is set to [2, 1] because K is feasible.
3. K is set to [2, 3, 1] and is rejected because it is not feasible.
4. K is set to [2, 1, 4] and is determined to be feasible.
   J is set to [2, 1, 4] because K is feasible.
5. K is set to [2, 5, 1, 4] and is rejected because it is not feasible.
6. K is set to [2, 1, 6, 4] and is rejected because it is not feasible.
7. K is set to [2, 7, 1,4] and is rejected because it is not feasible.

The final value of J is [2, 1, 4].

تحلیل پیچیدگی زمانی زمانبندی در مهلت معین

$$\sum_{i=2}^{n} [(i-1)+i] = n^2 - 1 \in \Theta\left(n^2\right).$$

$$W(n) \in \Theta\left(n^2\right).$$

- اثبات بهینه بودن زمانبندی، به روش استقرا .....

مثال ۴: کد هافمن

(Huffman code)

- فایل داده باید به صورت کارآمد بر روی حافظه جانبی ذخیره شود.

- در کد دودویی طول ثابت، طول کد همه کاراکترها یکسان است.
- مثال:

a: 00 b: 01 c: 11.

ababcbbbc

000100011101010111.

- با استفاده از کد دودویی با طول متغیر، می توانیم کدگذاری موثرتری داشته باشیم.
- چون در این رشته، کاراکتر b بیشتر از همه تکرار شده است، کد 0 را آن اختصاص می دهیم.
- سپس، کد کاراکتر a نمی تواند 00 باشد چون در اینصورت نمی توان یک a را از دو bb تشخیص داد.
- همچنین نمی توان کد را به صورت 01 درنظر گرفت. چون وقتی به یک 0 رسیدیم نمی توان تشخیص داد که b است یا بخشی از a .

ababcbbbc

- با این کدگذاری، رشته کد شده در مجموع به ۱۳بیت نیاز دارد.

a: 10 b: 0 c: 11

- در کدگذاری با طول ثابت ۱۸ بیت نیاز بود.

1001001100011.

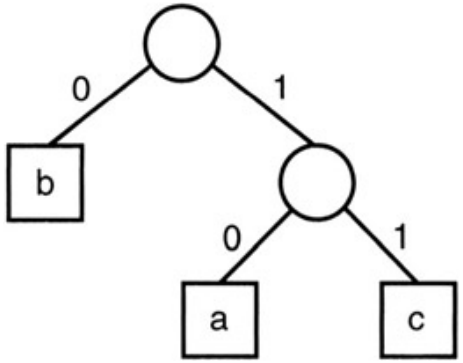
مساله کد دودویی بهینه

- هدف این است که برای یک فایل معین، کد دودویی برای هر یک از کاراکترهای فایل پیدا کنیم تا فایل با حداقل تعداد بیت نمایش داده شود.

کدهای پیشوندی

- کدهای پیشوندی نوع خاصی از کدهای باطول متغیر هستند که در آنها، هیچ کد مربوط به یک کاراکتر، آغاز کد کاراکتر دیگر را نمی سازد.

- کد طول ثابت نیز، یک کد پیشوندی است.



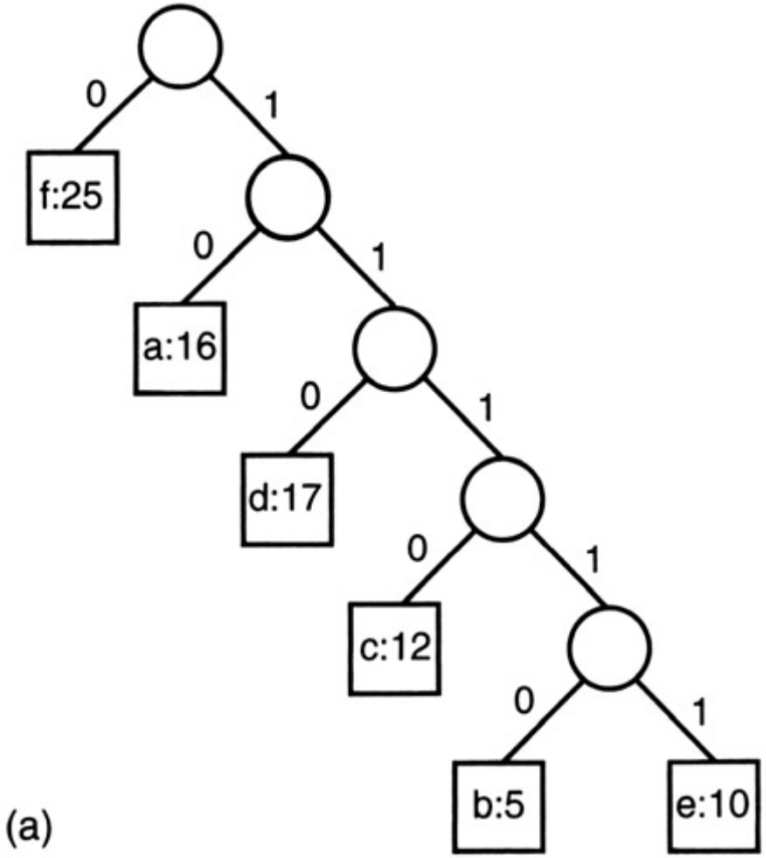
- مزیت کد پیشوندی، در زمان تجزیه فایل (رمزگشایی فایل کد شده) است.
- برای رمزگشایی، از اولین بیت سمت چپ فایل و ریشه درخت شروع می کنیم.
- در بیت ها حرکت می کنیم و در درخت به سمت چپ یا راست می رویم.
- به برگ که رسیدیم، کاراکتر را در برگ می یابیم.

مثال

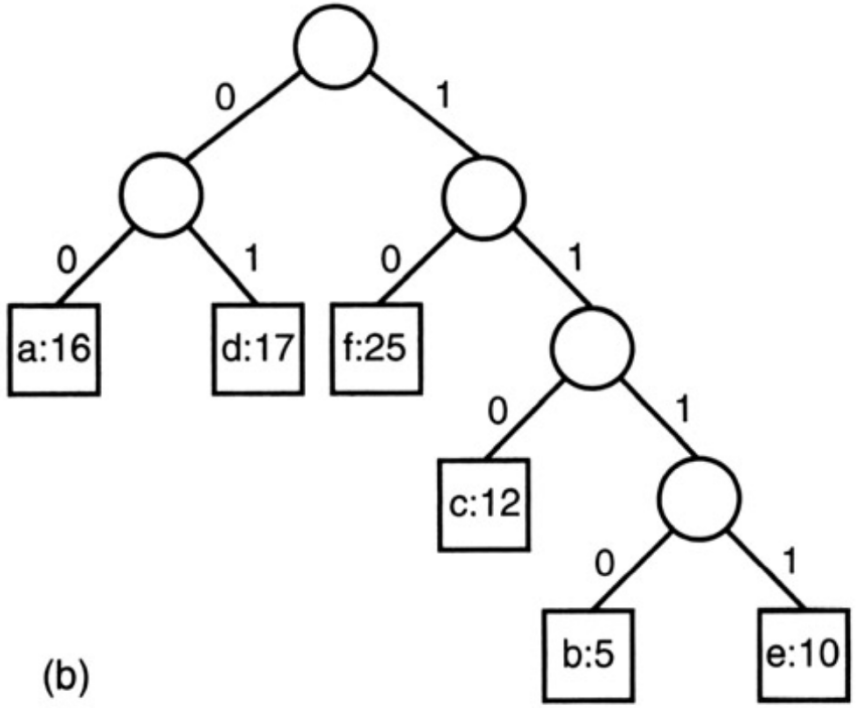
سه کدگذاری مختلف برای فایلی که فراوانی کاراکترها در آن در زیر آمده است:

Character	Frequency	*C1* (Fixed-Length)	*C2*	*C3* (Huffman)
a	16	000	10	00
b	5	001	11110	1110
c	12	010	1110	110
d	17	011	110	01
e	10	100	11111	1111
f	25	101	0	10

$$Bits(C1) = 16(3) + 5(3) + 12(3) + 17(3) + 10(3) + 25(3) = 255$$
$$Bits(C2) = 16(2) + 5(5) + 12(4) + 17(3) + 10(5) + 25(1) = 231$$
$$Bits(C3) = 16(2) + 5(4) + 12(3) + 17(2) + 10(4) + 25(2) = 212.$$



(a)



(b)

$$bits(T) = \sum_{i=1}^{n} frequency(v_i) depth(v_i)$$

الگوریتم هافمن

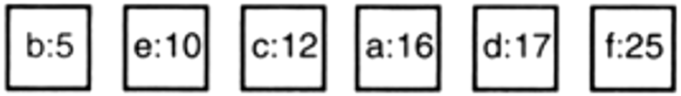
- هافمن، الگوریتمی را طراحی کرد که با ساختن یک درخت دودویی، یک کد بهینه دودویی را تولید می کند.

- کدی که به این روش تولید می شود، کد هافمن نامیده می شود.

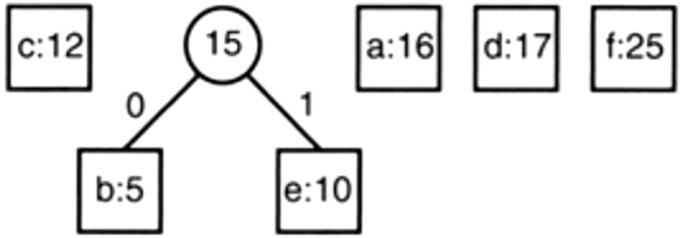
ابتدا مثالی از الگوریتم کدگذاری هافمن

- کاراکترهای موجود در فایل: {a,b,c,d,e,f}

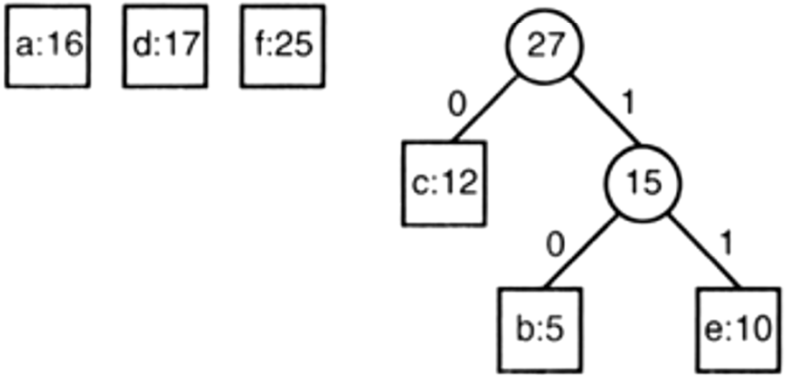
Character	Frequency
a	16
b	5
c	12
d	17
e	10
f	25


b:5
e:10
c:12
a:16
d:17
f:25

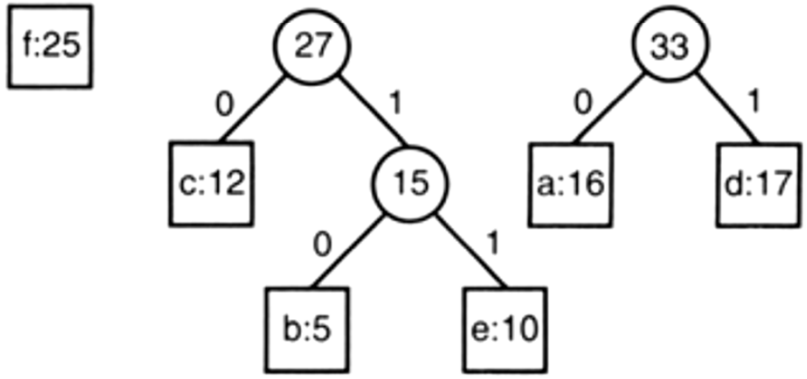
(0)


c:12
15
0
1
a:16
d:17
f:25
b:5
e:10

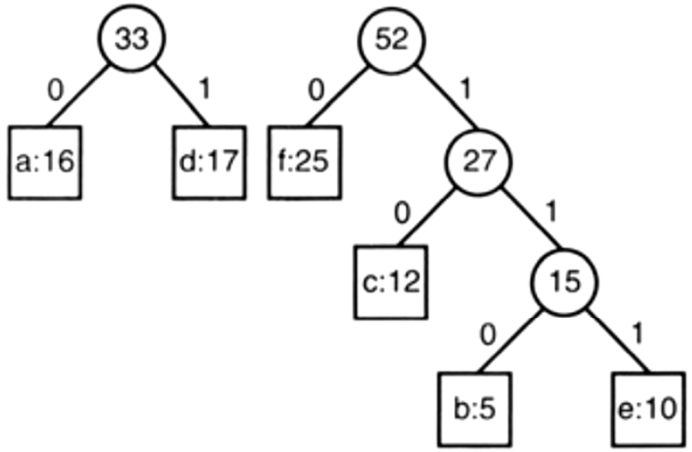
(1)


a:16
d:17
f:25
27
0
1
c:12
15
0
1
b:5
e:10

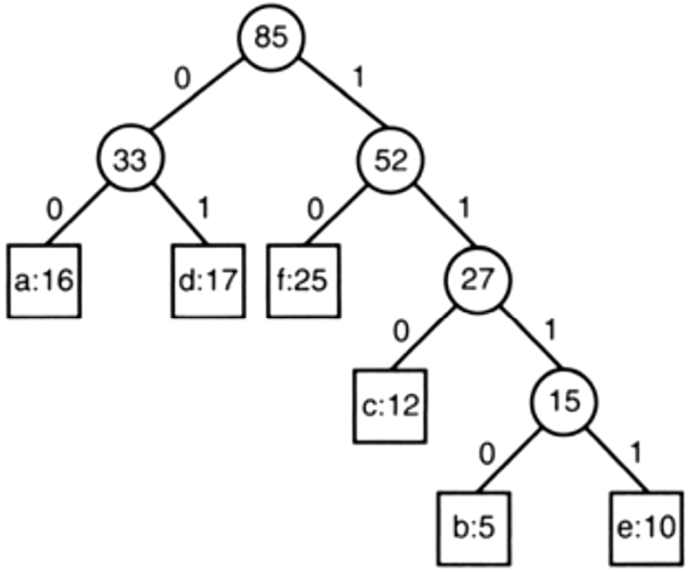
(2)


f:25
27
0
1
c:12
15
0
1
b:5
e:10
33
0
1
a:16
d:17

(3)



(4)



(5)

Character	Frequency	Huffman
a	16	00
b	5	1110
c	12	110
d	17	01
e	10	1111
f	25	10

الگوریتم هافمن

- ساختمان داده

```
struct nodetype
{
  char symbol;      // The value of a character.
  int frequency;    // The number of times the character  is in the file.
  nodetype* left;
  nodetype* right;
};
```

- n : تعداد کاراکترها در فایل است.
- n تا نود به صورت زیر تعریف می کنیم.

$p\text{->}\ symbol$ = a distinct character in the file;
$p\text{->}\ frequency$ = the frequency of that character in the file;
$p\text{->}\ left = p\text{->}\ right$ = NULL;

- از صف اولویت استفاده می کنیم.
- در صف اولویت، عنصری با بالاترین اولویت، زودتر از همه خارج می شود.
- در این مساله، عنصر با اولویت بیشتر، کاراکتری است که کمترین فراوانی را در فایل دارد.
- صف اولویت را می توان به صورت لیست پیوندی پیاده سازی کرد و کارایی آن به صورت heap بیشتر است.

الگوریتم هافمن

```
for (i = 1; i < = n-1; i++) {  // There is no solution check; rather, solution is obtained when i = n - 1.

   remove (PQ, p);          // Selection procedure.
   remove (PQ, q);          // Selection procedure.
   r = new nodetype;        // There is no feasibility check.
   r->left = p;
   r->right = q;
   r->frequency = p->frequency + q->frequency;
   insert (PQ, r);
}
remove (PQ, r);
return r;
```

تحلیل پیچیدگی زمانی

- اگر صف اولویت به صورت heap پیاده شود، هر عمل heap مستلزم $\theta(\lg n)$ عمل است و n-1 گذار از حلقه for وجود دارد. بنابراین مرتبه زمانی الگوریتم به صورت $\theta(n\lg n)$ است.

- اثبات بهینه بودن کدها، با استفاده از استقرا ....

مثال ۵:

روش حریصانه در مقابل برنامه ریزی پویا: مساله کوله پشتی

- روش حریصانه و روش برنامه ریزی پویا دو روش برای حل مسائل بهینه سازی هستند.

- غالبا مساله را می توان با استفاده از هر روشی حل کرد.

- غالبا هنگامیکه مساله ای با روش حریصانه حل می شود، نتیجه الگوریتمی ساده تر با کارایی بیشتر است.
- از طرف دیگر، معمولا تعیین اینکه آیا یک الگوریتم حریصانه، حل بهینه را تولید می کند یا نه، دشوار است.
- همانطور که مساله پرداخت بقیه پول نشان داد، همه الگوریتم های حریصانه چنین حلی را تولید نمی کنند.

روش حریصانه در حل مساله کوله پشتی صفر و یک

- یک کوله پشتی با قابلیت تحمل وزن W
- n تا قطعه با ارزش ها و وزن های معین
- هدف: انتخاب قطعات و قرار دادن آنها در کوله پشتی، به نحوی که وزن کل قطعات از W بیشتر نشود و بیشترین ارزش حاصل شود.

بیان مساله

$$S = \{item_1, item_2, ..., item_n\}$$
$$w_i = \text{weight of } item_i$$
$$p_i = \text{profit of } item_i$$
$$W = \text{maximum weight the knapsack can hold,}$$

$$\sum_{item_i \in A} p_i \quad \text{is maximized subject to} \quad \sum_{item_i \in A} w_i \leq W.$$

یک راه حل ساده: جستجوی جامع

- یعنی در نظر گرفتن همه زیرمجموعه های ممکن این n قطعه
- کنار گذاشتن زیرمجموعه هایی که جمع وزن آنها بالاتر از W است.
- و پیدا کردن ماکزیمم ارزش ها

- تعداد کل زیرمجموعه ها نمایی است.

- راه بهتر: استفاده یک رهیافت حریصانه

روش حریصانه برای مساله کوله پشتی صفر و یک

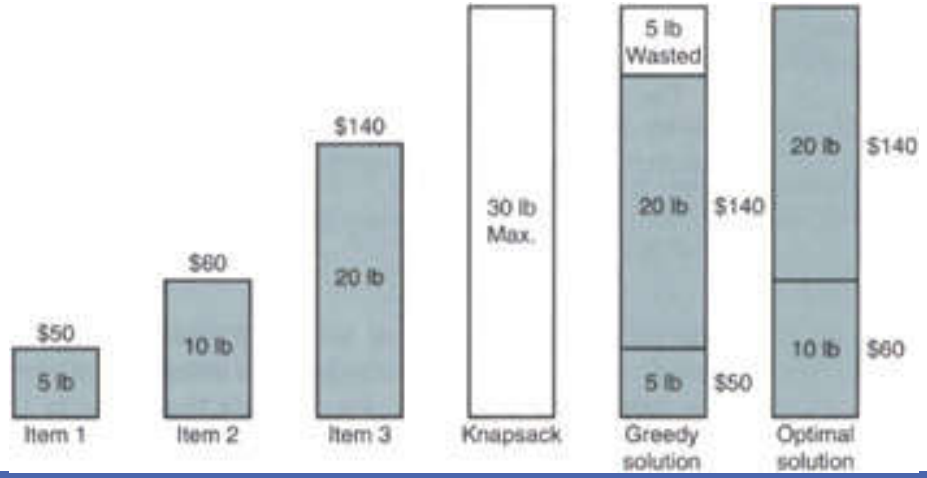
- رهیافت حریصانه اول: قطعاتی با بیشترین ارزش زودتر برداشته شوند....
- رهیافت حریصانه دوم: قطعاتی با کمترین وزن (قطعات سبک تر) زودتر برداشته شوند....
- رهیافت حریصانه سوم: قطعاتی با بزرگ ترین ارزش به ازای واحد وزن، زودتر برداشته شوند....

و هر قطعه در صورتی در کوله پشتی گذاشته شود که مجموع وزن قطعات از W فراتر نرود.

مثال

$$item_1 : \frac{\$50}{5} = \$10 \qquad item_2 : \frac{\$60}{10} = \$6 \qquad item_3 : \frac{\$140}{20} = \$7.$$

- به ترتیب افزایش داریم (از چپ به راست): $item_1, item_3, item_2$
- این روش حریصانه $item_1, item_3$ را در کوله پشتی قرار می دهد و سود کل ۱۹۰ را می دهد.
- ولی جواب بهینه، $item_3, item_2$ را در کوله پشتی قرار می دهد و سود ۲۰۰ را دارد.



یک روش حریصانه برای مساله کوله پشتی کسری

- در مساله کوله پشتی کسری، مجبور نیستیم حتما یک قطعه را به طور کامل برداریم. بلکه می توانیم کسری از یک قطعه را برداریم.

- رهیافت حریصانه: قطعاتی با بزرگ ترین ارزش به ازای واحد وزن، زودتر برداشته شوند....

$$\$50 + \$140 + \frac{5}{10}(\$60) = \$220.$$

- الگوریتم حریصانه برای کوله پشتی کسری، ظرفیت را هدر نمی دهد و همواره حل بهینه را به دست می آورد.

اکنون: یک روش برنامه ریزی پویا برای مساله کوله پشتی صفر و یک

- ابتدا بررسی می کنیم که اصل بهینگی برقرار باشد.....

- سپس طراحی رابطه بازگشتی:

$$P[i][w] = \begin{cases} maximum\left(P[i-1][w], p_i + P[i-1][w-w_i]\right) & \text{if } w_i \le w \\ P[i-1][w] & \text{if } w_i > w. \end{cases}$$

- P[i][w] : سود بهینه برای مساله کوله پشتی با ظرفیت w و در اختیار داشتن i شیء اول
- مقدارهای P[i][o] و P[o][w] را مساوی صفر قرار می دهیم.
- پیچیدگی الگوریتم برنامه ریزی پویا: تعداد عناصری که محاسبه می شود:

$$nW \in \Theta(nW).$$

پایان فصل چهارم