Greedy Approach

روش حریصانه

The idea

- داده‌ها یکی یکی انتخاب می‌شوند.
- هر انتخاب باید بهترین باشد، بدون توجه به انتخاب‌های قبلی یا در آینده.
- اغلب برای حل مسائل بهینه‌سازی استفاده می‌شود.
- بهینگی راه حل ارایه شده توسط الگوریتم حریصانه باید اثبات شود.

مثال:
پول خرد کردن

Coins

Amount owed: 36 cents

Step

1. Grab **25**

2. Grab first **10**

3. Reject second **10**

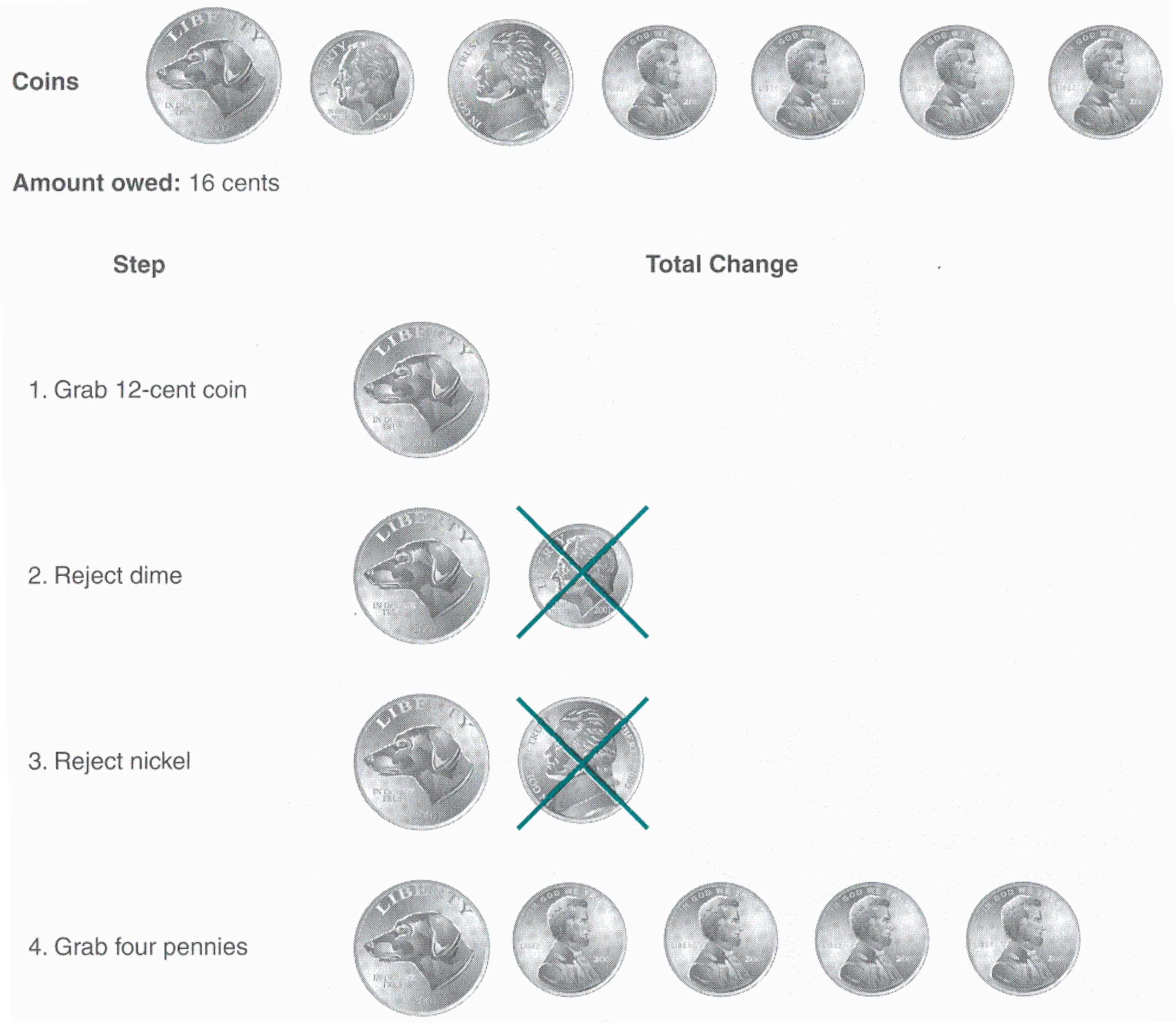
4. Reject **5**

5. Grab **1**

Figure 4.1 • A greedy algorithm for giving change.

U.S. coins (penny(1), nickel(5), dime(10), quarter(25), half dollar(50))

حالتی که روش حریصانه جواب بهینه را نمی‌دهد



if we include a 12-cent coin with the U.S. coins, the greedy algorithm does not always give an optimal solution.

Greedy solution: 16=12+1+1+1+1
Optimal solution: 16=10+5+1

The algorithm

```
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)
          reject the coin;             // feasibility check
    else
          add the coin to the change;

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

Basic components in Greedy approach

- A ***selection procedure*** chooses the next item to add to the set. The selection is performed according to a **greedy criterion** that satisfies some locally optimal consideration at the time.

- A ***feasibility check*** determines if the new set is feasible by checking whether it is possible to complete this set in such a way as to give a solution to the instance.

- A ***solution check*** determines whether the new set constitutes a solution to the instance.

Minimum spanning trees
درخت پوشای کمینه

- تعریف درخت

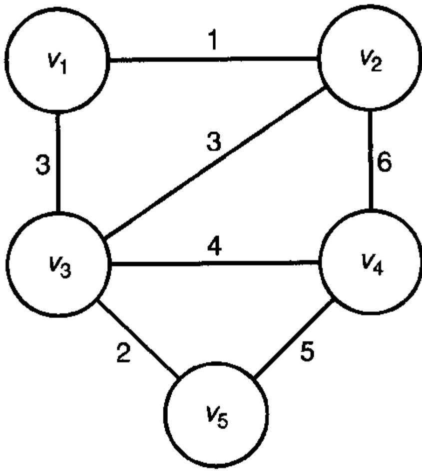
Definition
An ***undirected graph*** G consists of a finite set V whose members are called the vertices of G, together with a set E of pairs of vertices in V. These pairs are called the edges of G. We denote G by

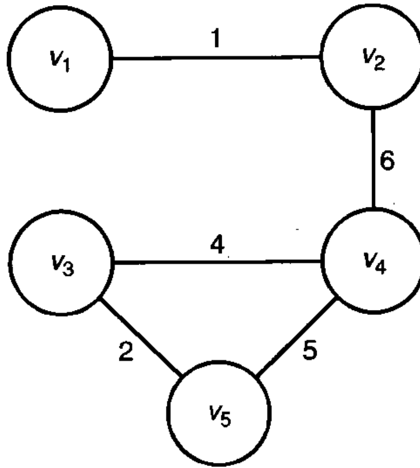
$$G = (V, E) .$$

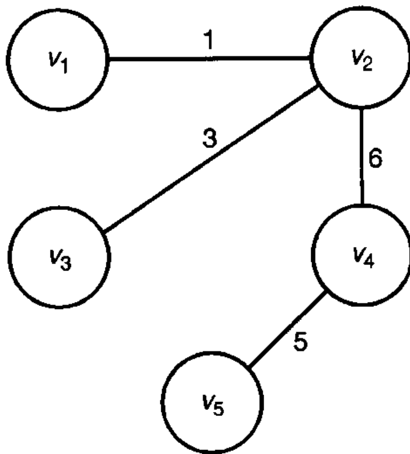
- دو روش حل برای یافتن درخت پوشای کمینه:
  - روش کراسکال (Kruskal)
  - روش پریم (Prim)

مثال:

گراف و چند درخت متناظر با آن







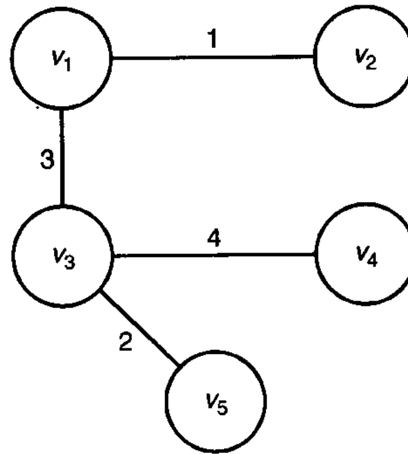
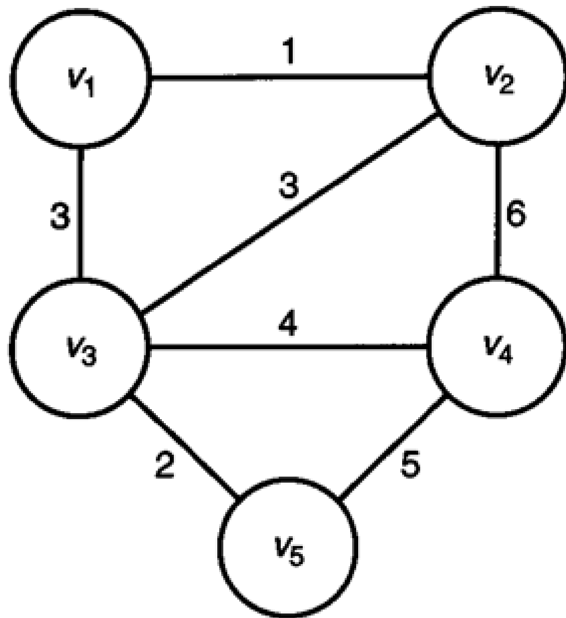


Figure 4.3 ● A weighted graph and three subgraphs.

An illustration (1)

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

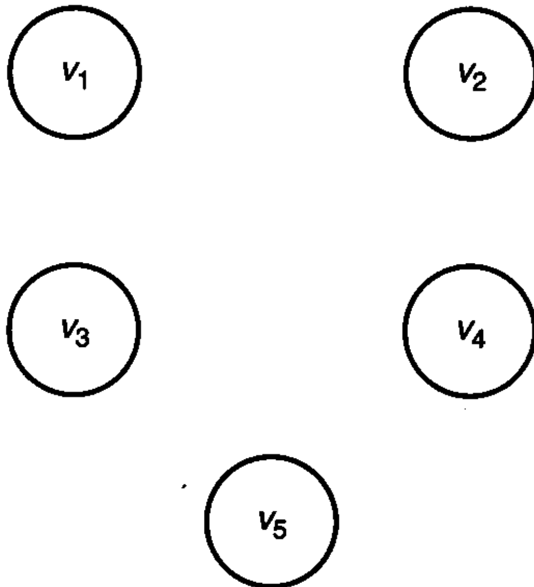
(v_2, v_3) 3

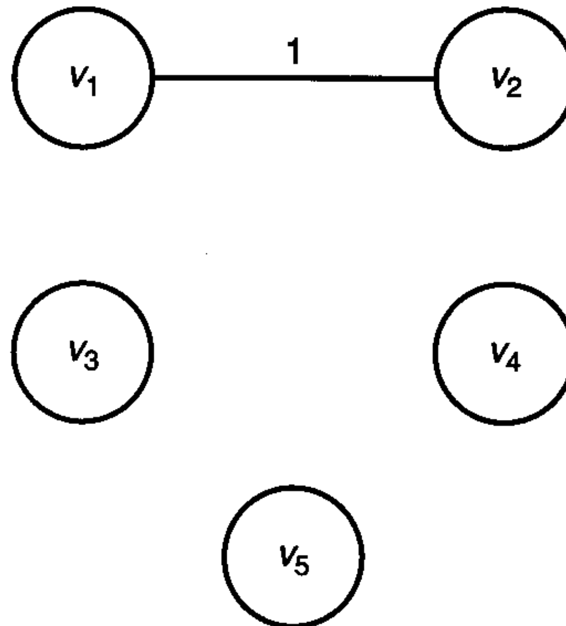
(v_3, v_4) 4

(v_4, v_5) 5

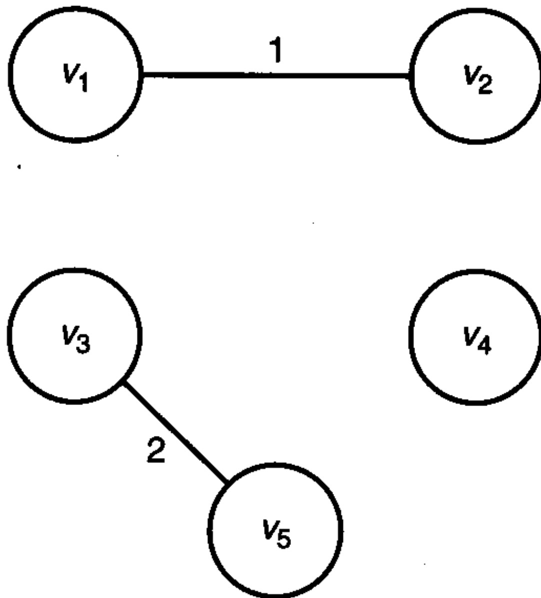
(v_2, v_4) 6

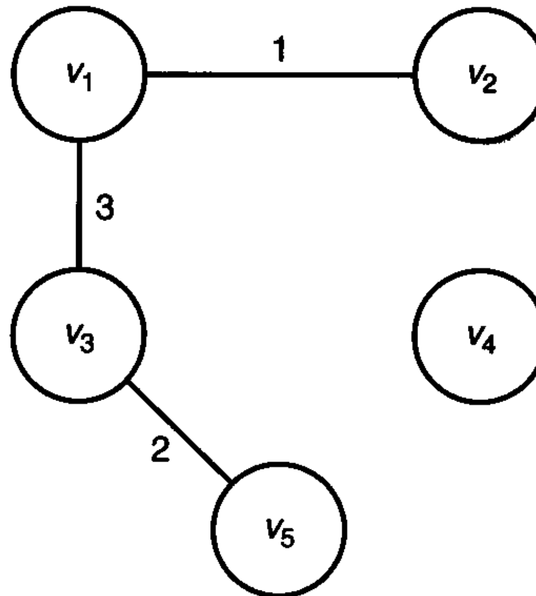
An illustration (2)



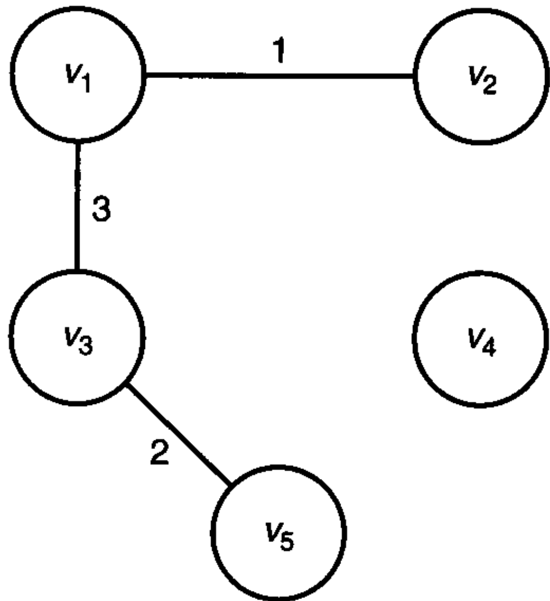


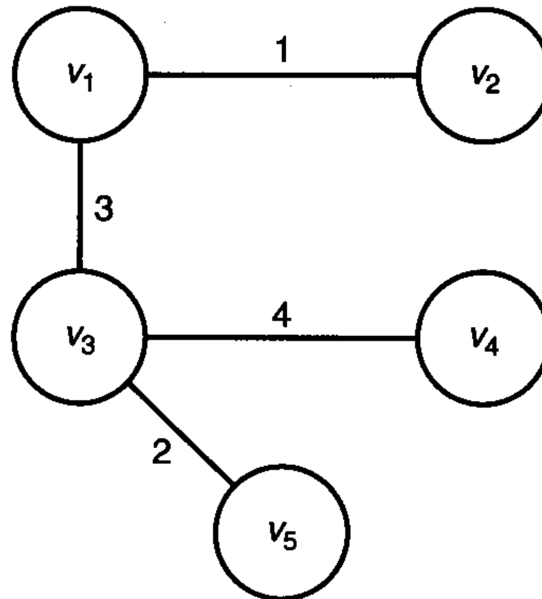
An illustration (3)





An illustration (4)





Data types and operations

- Data types
  - index *i*;
  - set_pointer *p*, *q*;
- Operations

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

Kruskal's algorithm

```
F = ∅;                                              // 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             // feasibility check
                         disjoint subsets){
      merge the subsets;
      add the edge to F;
  }
  if (all the subsets are merged)                   // solution check
      the instance is solved;
}
```

Kruskal's algorithm

```
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;
        }
    }
}
```

Worst-case time complexity

- Basic operation: a comparison instruction
- Input size: n, the number of vertices, and m, the number of edges
- Analysis:
  - The time to sort the edges: $W(m) \in \Theta(m \lg m)$
  - The time in the while loop: $W(m) \in \Theta(m \lg m)$
  - The time to initialize n disjoint sets: $T(n) \in \Theta(n)$
  - Overall: $W(m, n) \in \Theta(m \lg m)$
  - In the worst case every vertex can be connected to every other vertex $m=n(n-1)/2$, Therefore $W(m,n) = \Theta(n^2 lgn)$

Proof

▲ **Lemma 4.2** Let $G = (V, E)$ be a connected, weighted, undirected graph; let F be a promising subset of E; and let e be an edge of minimum weight in $E - F$ such that $F \cup \{e\}$ has no simple cycles. Then $F \cup \{e\}$ is promising.

- Theorem 4.2: Kruskal's algorithm always produces a minimum spanning tree

Prim's algorithm

$F = \emptyset$ // Initialize set of edges to empty.

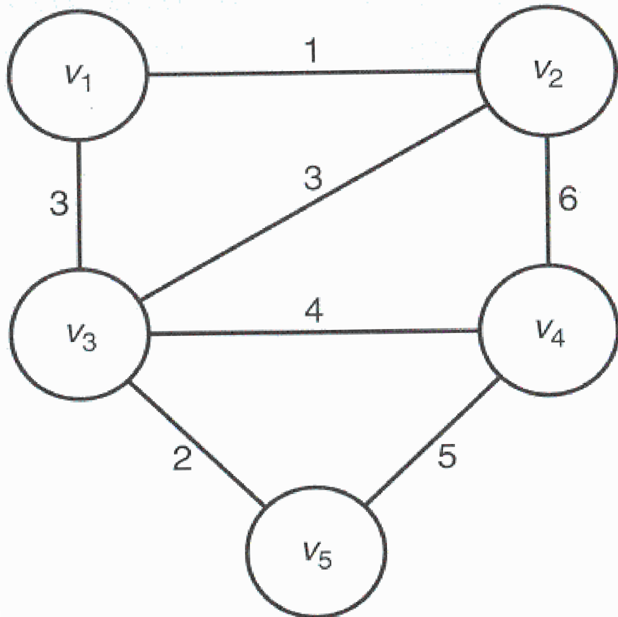
Y = {*v*1}   // Initialize set of vertices to
                  // contain only the first one.

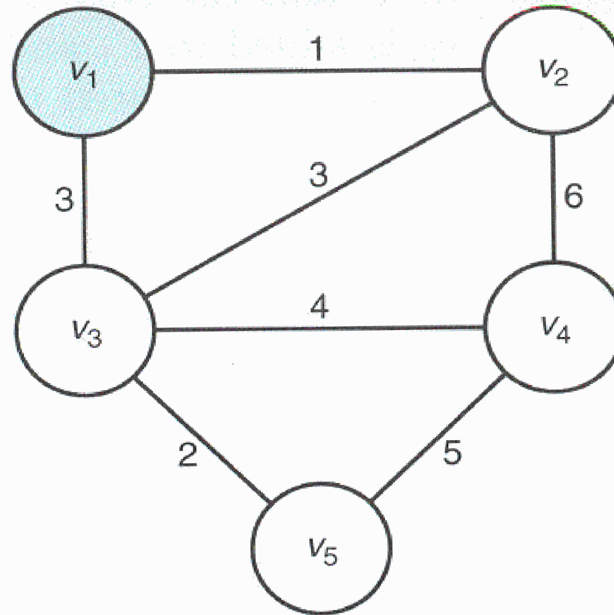
while (the instance is not solved){
   select a vertex in *V* - *Y* that is nearest to *Y;*
                  // selection procedure and feasibility check

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

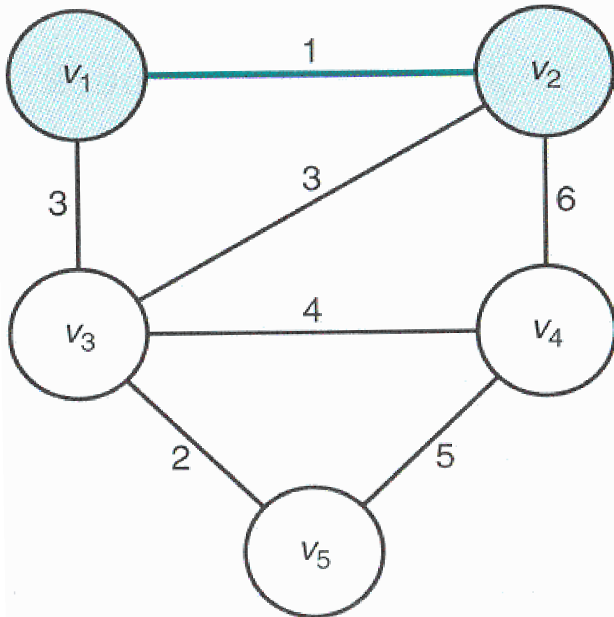
   if (*Y* == *V*)// solution check the instance is solved;
}

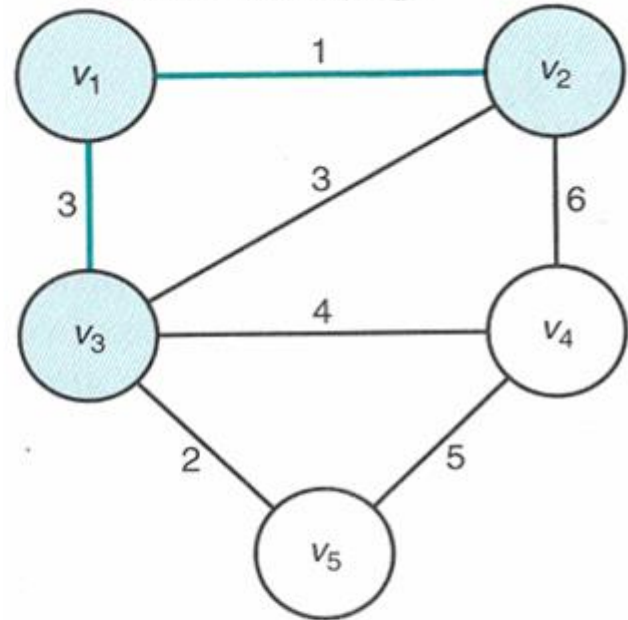
An example



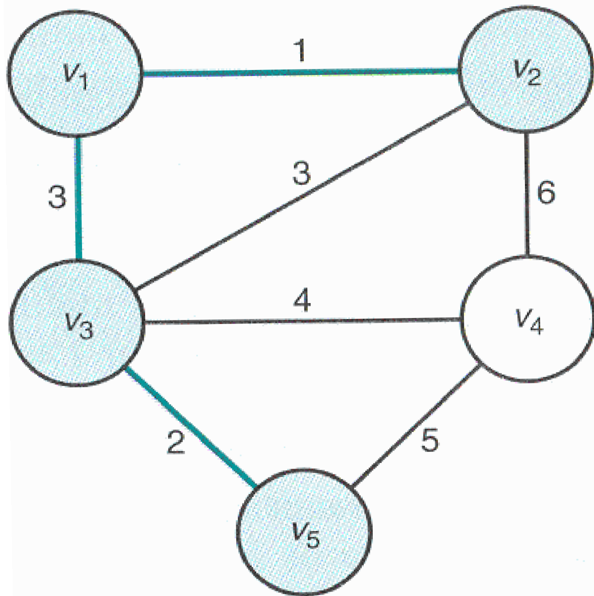


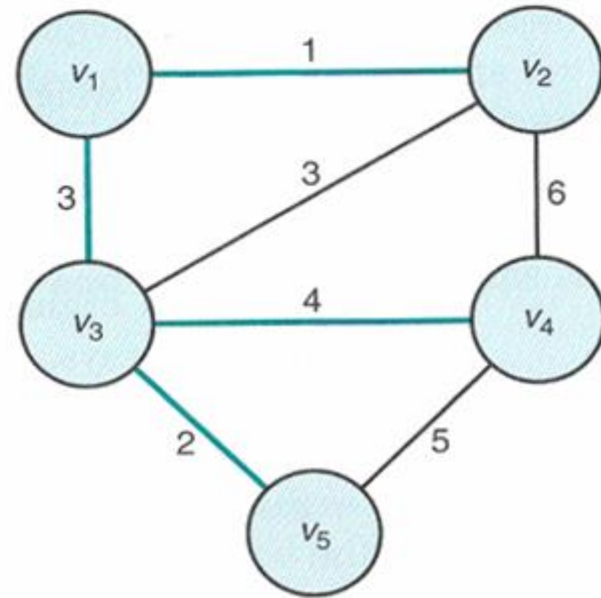
An example





An example





Adjacency matrix

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

	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

Figure 4.5 ● The array representation *W* of the graph in Figure 4.3(a).

Prim's algorithm

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

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

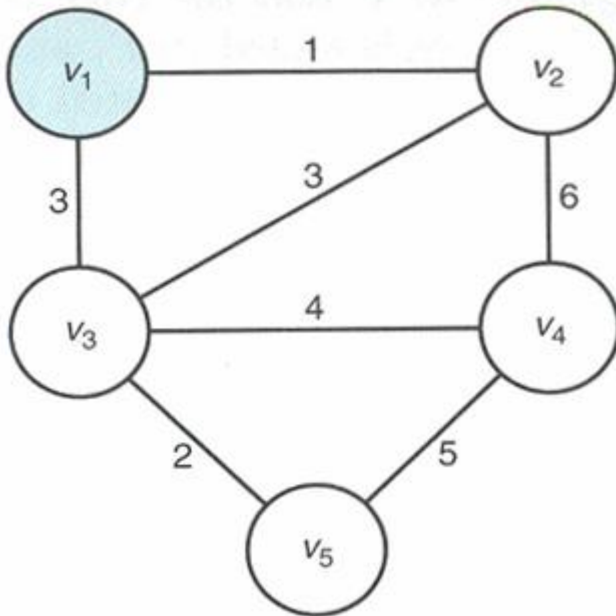
Algorithm 4.1: Prim's Algorithm

```
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
    }
…
```

Prim's algorithm (cont'd)

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

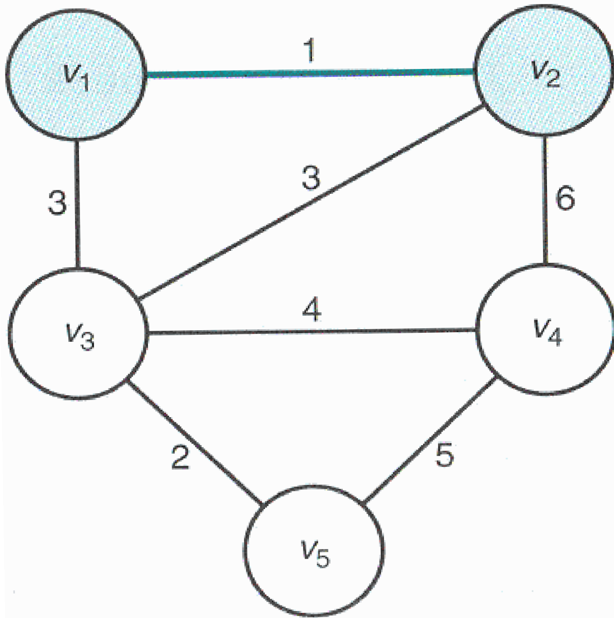
An example



nearest			
2	3	4	5
1	1	1	1

distance			
2	3	4	5
1	3		

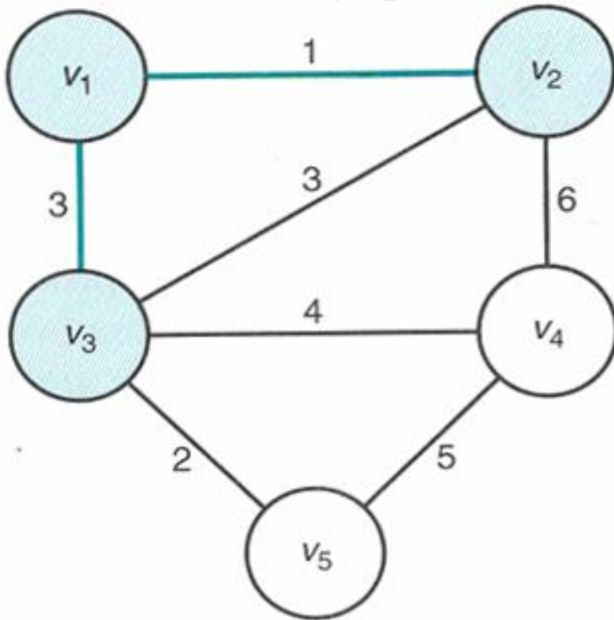
```
F = Ø;
   for (i = 2; i <= n; i++){
         nearest [i] = 1;            // For all vertices, initialize v1
         distance [i] = W[1][i] ; // to be the nearest vertex in
   }
```



nearest			
2	3	4	5
1	1	1	1
		2	

distance			
2	3	4	5
1	3		
-1		6	

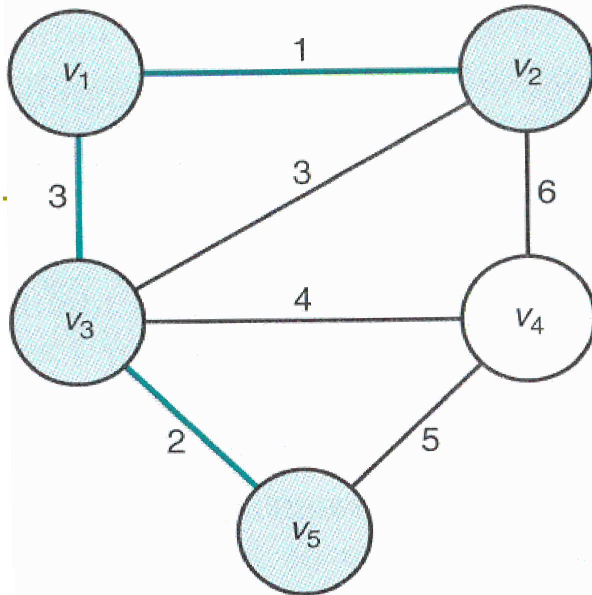
```
repeat (n - 1 times){      min = ∞;
          for (i = 2; i <= n; i++)
                    if (0 ≤ distance[i] < min){
                              min = distance[i];
                              vnear = i;}
    e = edge connecting vertices indexed by vnear and nearest[vnear];
    add e to F;
    distance[vnear] = -1;
    for (i = 2; i <= n; i++)
                    if (W[i][vnear] < distance[i]){
                              distance[i]= W[i] [vnear];
                              nearest [i] = vnear; }
    }
```



nearest			
2	3	4	5
1	1	1	1
		2	
		3	3

distance			
2	3	4	5
1	3		
-1		6	
	-1	4	2

```
repeat (n - 1 times){      min = ∞;
        for (i = 2; i <= n; i++)
                if (0 ≤ distance[i] < min){
                        min = distance[i];
                        vnear = i;}
    e = edge connecting vertices indexed by vnear and nearest[vnear];
    add e to F;
    distance[vnear] = -1;
    for (i = 2; i <= n; i++)
                if (W[i][vnear] < distance[i]){
                        distance[i]= W[i] [vnear];
                        nearest [i] = vnear; }
    }
```

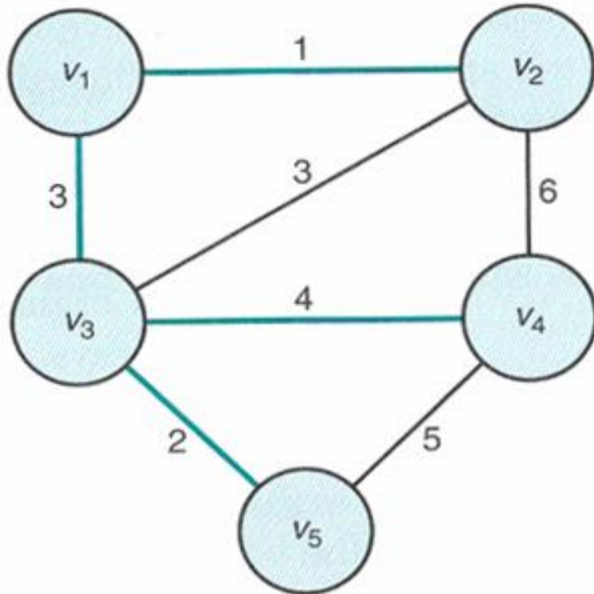


nearest			
2	3	4	5
1	1	1	1
		2	
		3	3

distance			
2	3	4	5
1	3		
-1		6	
	-1	4	2
			-1

```
repeat (n - 1 times){      min = ∞;
        for (i = 2; i <= n; i++)
                if (0 ≤ distance[i] < min){
                        min = distance[i];
                        vnear = i;}
    e = edge connecting vertices indexed by vnear and nearest[vnear];
    add e to F;
    distance[vnear] = -1;
    for (i = 2; i <= n; i++)
                if (W[i][vnear] < distance[i]){
                        distance[i]= W[i] [vnear];
                        nearest [i] = vnear; }
    }
```

5. Vertex v_4 is selected.



nearest			
2	3	4	5
1	1	1	1
		2	
		3	3

distance			
2	3	4	5
1	3		
-1		6	
	-1	4	2
			-1
		-1	

```
repeat (n - 1 times){      min = ∞;
        for (i = 2; i <= n; i++)
                if (0 ≤ distance[i] < min){
                        min = distance[i];
                        vnear = i;}
   e = edge connecting vertices indexed by vnear and nearest[vnear];
   add e to F;
   distance[vnear] = -1;
   for (i = 2; i <= n; i++)
                if (W[i][vnear] < distance[i]){
                        distance[i]= W[i] [vnear];
                        nearest [i] = vnear; }
   }
```

Every-case time complexity

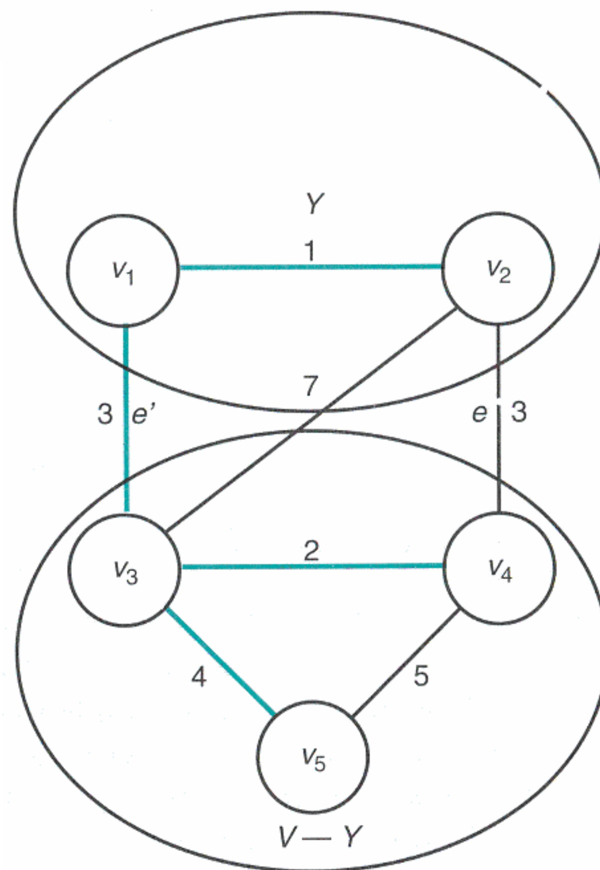
- Basic Operation: There are two loops, each with $n - 1$ iterations, inside the **repeat** loop. Executing the instructions inside each of them can be considered to be doing the basic operation once.
- Input size: n, the number of vertices
- Then:

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

Proof

▲ Lemma 4.1 Let $G = (V, E)$ be a connected, weighted, undirected graph; let F be a promising subset of E; and let Y be the set of vertices connected by the edges in F. If e is an edge of minimum weight that connects a vertex in Y to a vertex in $V - Y$, then $F \cup \{e\}$ is promising.

- Theorem 4.1: Prim's algorithm always produces a minimum spanning tree



Comparing Prim's algorithm with Kruskal's algorithm

- Prim's algorithm: $T(n) \in \Theta(n^2)$
- Kruskal's algorithm:
  - $W(m, n) \in \Theta(m \lg m)$
- $n - 1 \leq m \leq n(n-1)/2$
- Sparse graph
  - *Kruskal's algorithm should be faster.*
- Highly connected
  - *Prim's algorithm should be faster.*

Dijkstra's algorithm

shortest paths from one source to all the others

الگوریتم دایکسترا

کوتاهترین مسیر از مبدا به همه راس‌ها

Dijkstra's algorithm

shortest paths from one source to all the others

Y = {*v*1};

F = Ø;

while (the instance is not solved) {

select a vertex *v* from *V* - *Y*, that has a shortest path from *v*1, using only vertices in *Y* as intermediates;

//selection procedure and feasibility check

add the new vertex *v* to *Y*;

add the edge (on the shortest path) that touches *v* to *F*;

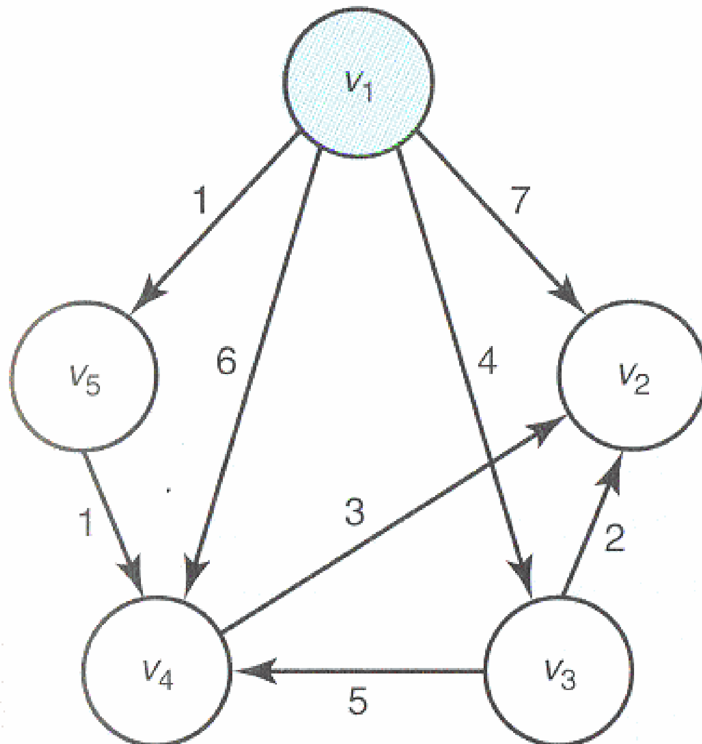
if (*Y* == *V*) the instance is solved;

// solution check

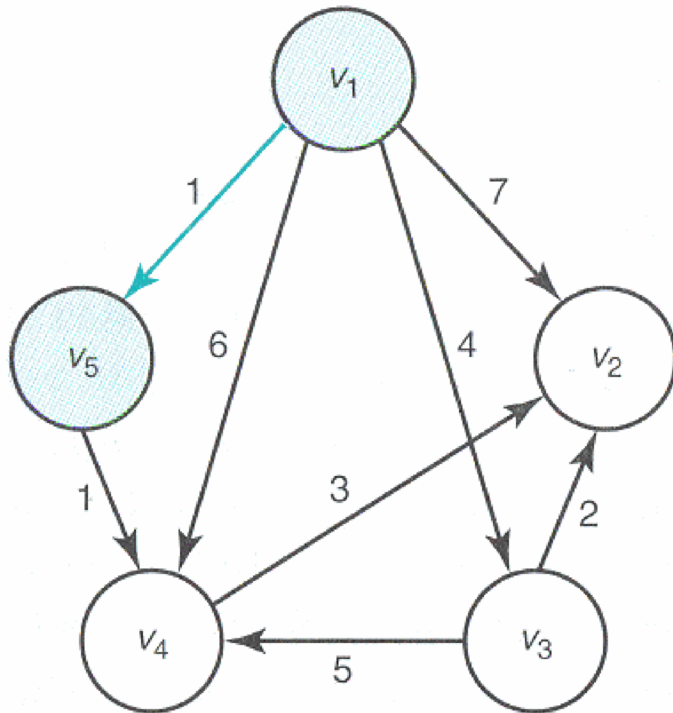
}

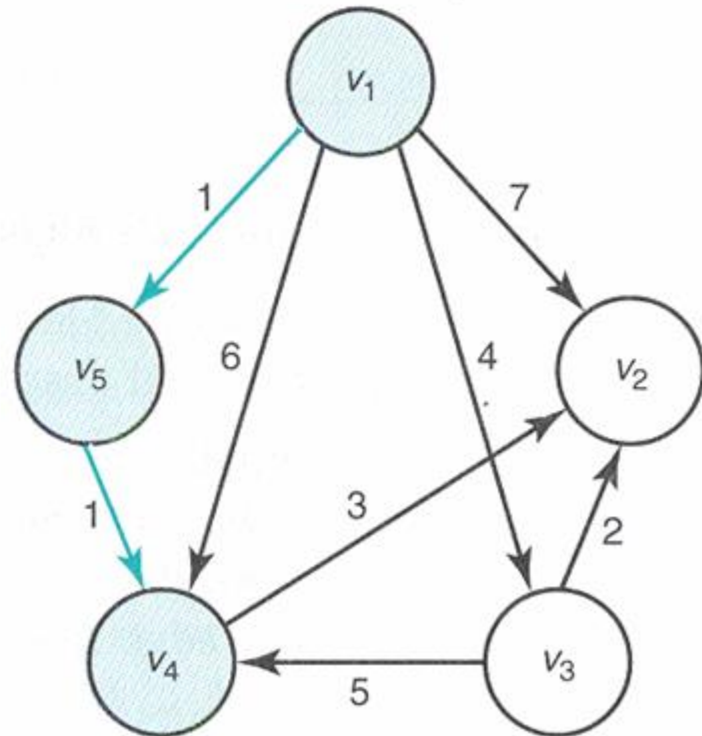
An example (1)

Compute shortest paths from v_1.

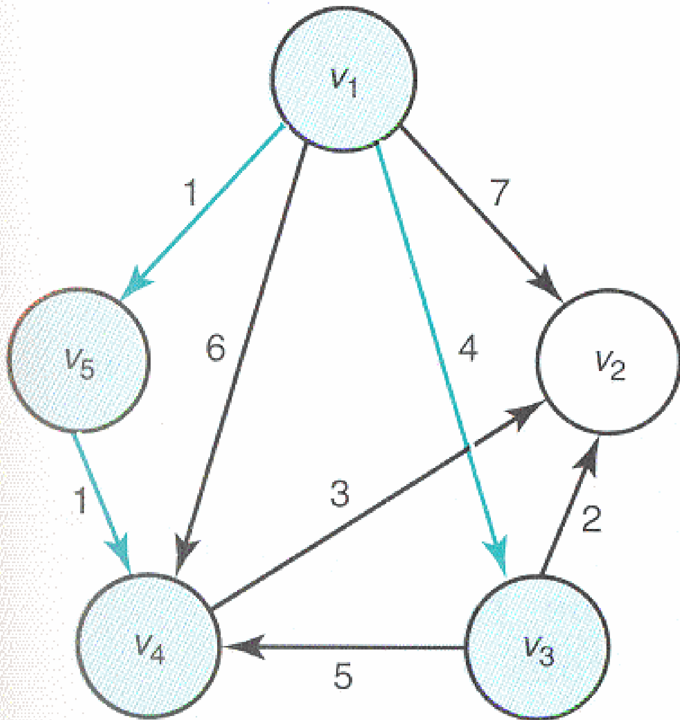


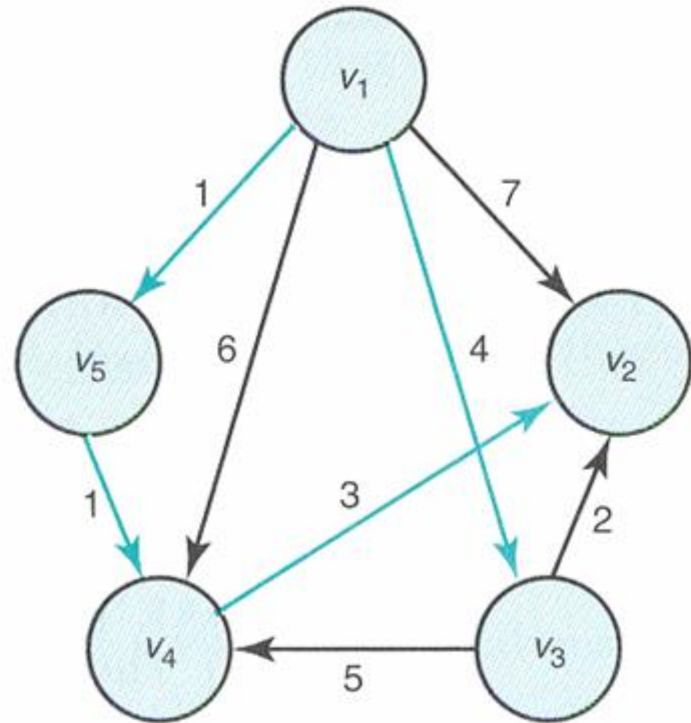
An example (2)





An example (3)





Auxiliary arrays

- *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

آخرین گره برای دسترسی به گرهی که انتخاب نشده است

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

The algorithm (1)

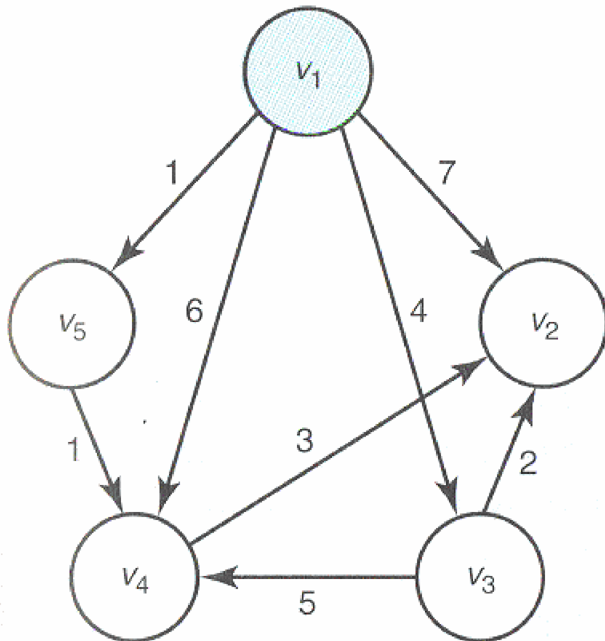
- Algorithm 4.3: Dijkstra's Algorithm

void *dijkstra* (**int** *n*, **const number** *W[][]*, **set_of_edges**&*F*) {

    index *i, vnear*;
    edge *e*;
    index *touch* [2 .. *n*]; **number** *length* [2 .. *n*];
    F = Ø;
    for (*i* = 2; *i*<= *n; i*++){
        touch [i] = 1;
        length [i] = *W*[1] [*i*];
    }

The algorithm (2)

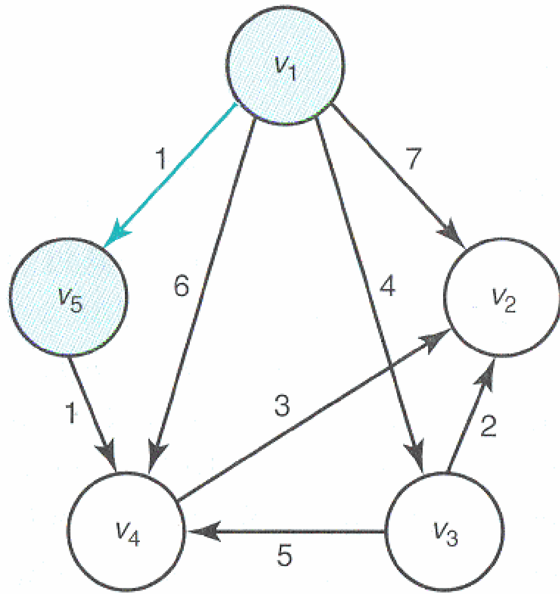
```
repeat (n - 1 times){
    min = ∞;
    for (i = 2; i < = n; i++)
        if ( 0 ≤ length [i] < min) {
            min = length [i];
            vnear = i;}
        e = edge from vertex indexed by touch [vnear] to
    vertex indexed by vnear;
        add e to F;
        for (i = 2; i < = n; i++)
            if (length [vnear] + W[vnear] [i] < length [i]){
                length[i] = length[vnear] + W[vnear][i];
                touch[i] = vnear;}
        length[vnear] = -1;
}
```



touch			
2	3	4	5
1	1	1	1

length			
2	3	4	5
7	4	6	1

```
for (i = 2; i<= n; i++){
        touch [i] = 1;
        length [i] = W[1] [i];}
```

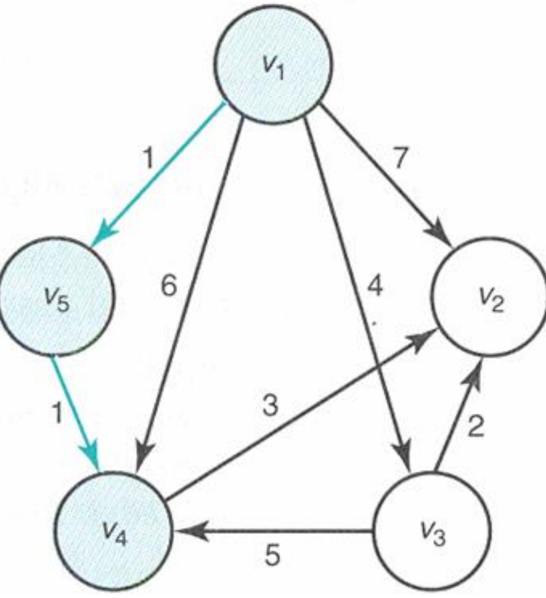


le

touch			
2	3	4	5
1	1	1	1
		5	

length			
2	3	4	5
7	4	6	1
		2	-1

```
repeat (n - 1 times){       min = ∞;
    for (i = 2; i < = n; i++)
        if ( 0 ≤ length [i] < min) {
                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;}
        length[vnear] = -1;}
```

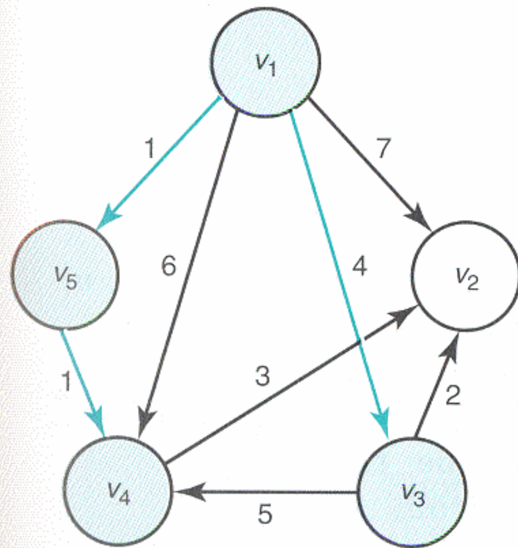


touch			
2	3	4	5
1	1	1	1
		5	
4			

length			
2	3	4	5
7	4	6	1
		2	-1
5		-1	

repeat (*n* - 1 times){ *min* = ∞;
    for (*i* = 2; *i* < = *n; i*++)
        if (0 ≤ *length [i]* < *min*) {
            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*;}
        length[vnear] = -1;}

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.

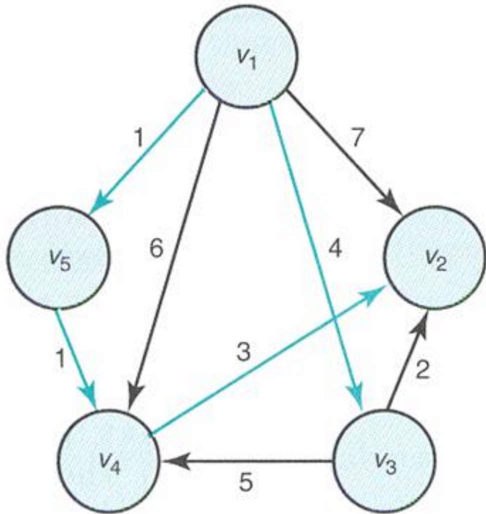


touch			
2	3	4	5
1	1	1	1
		5	
4			

length			
2	3	4	5
7	4	6	1
		2	-1
5		-1	
	-1		

```
repeat (n - 1 times){        min = ∞;
    for (i = 2; i < = n; i++)
        if ( 0 ≤ length [i] < min) {
                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;}
        length[vnear] = -1;}
```

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



ple

touch			
2	3	4	5
1	1	1	1
		5	
4			

length			
2	3	4	5
7	4	6	1
		2	-1
5		-1	
	-1		
-1			

```
repeat (n - 1 times){        min = ∞;
    for (i = 2; i < = n; i++)
        if ( 0 ≤ length [i] < min) {
            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;}
        length[vnear] = -1;}
```

Every-case time complexity

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

Dijkstra vs Prim

```
repeat (n - 1 times){              min = ∞;
    for (i = 2; i < = n; i++)
          if ( 0 ≤ length [i] < min) {
                    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;}
          length[vnear] = -1;}
```

```
repeat (n - 1 times){              min = ∞;
          for (i = 2; i <= n; i++)
                    if (0 ≤ distance[i] < min){
                              min = distance[i];
                              vnear = i;}
    e = edge connecting vertices indexed by vnear and nearest[vnear];
    add e to F;
    distance[vnear] = -1;
    for (i = 2; i <= n; i++)
                    if (W[i][vnear] < distance[i]){
                              distance[i]= W[i] [vnear];
                              nearest [i] = vnear; }
    }
```

Scheduling
زمان‌بندی

- دو نوع زمان‌بندی

۱- جمع زمانی که کارها منتظرند یا در حال سرویس گرفتن هستند کمینه شود.

- Minimize the total time in **waiting** and **being served** (time in the system)

۲- زمان‌بندی بامهلت

- scheduling with deadlines

زمان‌بندی نوع اول (بدون مهلت)

- Compute the optimal scheduling:

  $t_1 = 5$, $t_2 = 10$, and $t_3 = 4$

  three jobs and their service times

- برای حل کارها را به ترتیب از کوچک به بزرگ مرتب می‌کنیم.

The algorithm

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

while (the instance is not solved){

      schedule the next job; // selection procedure
                             // feasibility check

      if (there are no more jobs) // solution check
            the instance is solved;
}
```

Proof

- Time complexity: $W(n) \in \Theta(n\lg n)$
- Theorem 4.3

  The only schedule that minimizes the total time in the system is one that schedules jobs in nondecreasing order by service time

Multiple-server scheduling problem

- Server 1 serves jobs 1, $(1+m)$, $(1+2m)$, $(1+3m)$, …
- Server 2 serves jobs 2, $(2+m)$, $(2+2m)$, $(3+3m)$, …
- …
- Server i serves jobs i, $(i+m)$, $(i+2m)$, $(i+3m)$, …
- …
- Server m serves jobs m, $(m+m)$, $(m+2m)$, $(m+3m)$, …

Scheduling with deadlines
زمان‌بندی با مهلت

- در این نوع مساله زمان‌بندی، انجام هر کار، یک واحد زمان می‌برد تا به پایان برسد و یک مهلت (ضرب‌الاجل) و یک سود دارد.
- اگر کار قبل یا در مهلت شروع شود، سود حاصل می‌شود.

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

- هدف: به دست آوردن بیشترین منفعت

Some terms

- Feasible sequence
  - all the jobs in the sequence start by their deadlines
- Feasible set
  - there exists at least one feasible sequence for the jobs in this set.
- Optimal sequence
  - a feasible sequence with maximum total profit.

The algorithm

```
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)
              add this job to S;

     if (there are no more jobs)
              the instance is solved;
}
```

The formal algorithm

- Algorithm 4.4: Scheduling with Deadlines

  void *schedule* (**int** *n*, **const int** deadline [], **sequence_of_integer**& *j*)

```
{
        //sorted jobs in nonincreasing order by profit
        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;
        }
}
```

Example

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

Example

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].

Worst-case time complexity

- Basic operation: comparison instructions
- Input size: *n*, the number of jobs
- Time complexity:
  - time for sorting: $\Theta(n\lg n)$
  - comparisons in for-i loop:
    $$\sum_{i=2}^{n}[(i-1)+i] = n^2 - 1 \in \Theta(n^2)$$
  - Overall: W(n) $\in \Theta(n^2)$

Theorem 4.4

- Algorithm 4.4 always produces an optimal set of jobs
- Proof: induction on the number of jobs n

Huffman code
کدهافمن

- در فشرده‌سازی کاربرد دارد.

- File: ababcbbbc
- Encoding scheme:
  - a: 00, b: 01, c: 11
  - a: 10, b: 0, c: 11

Prefix codes

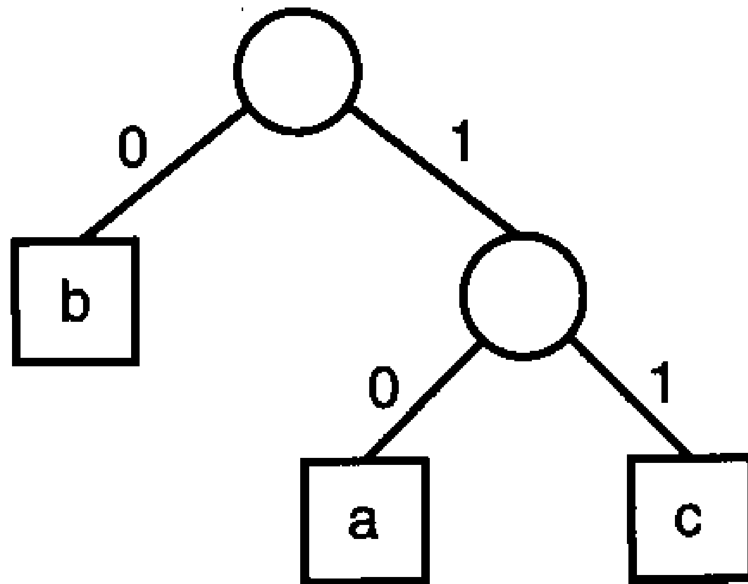


Figure 4.9 ● Binary tree corresponding to Code 4.2.

An example

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

Table 4.1 ● Three codes for the same file. C3 is optimal.

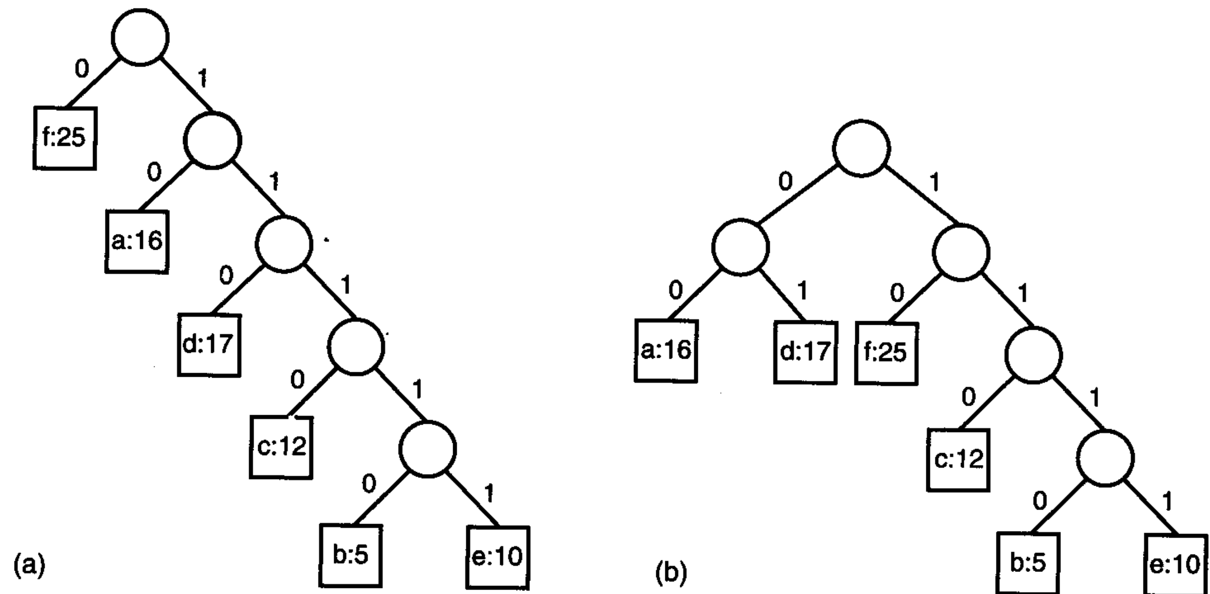


Figure 4.10 ● The binary character code for Code C2 in Example 4.7 appears in (a), while the one for Code C3 (Huffman) appears in (b).

The number of bits taken to encode a file

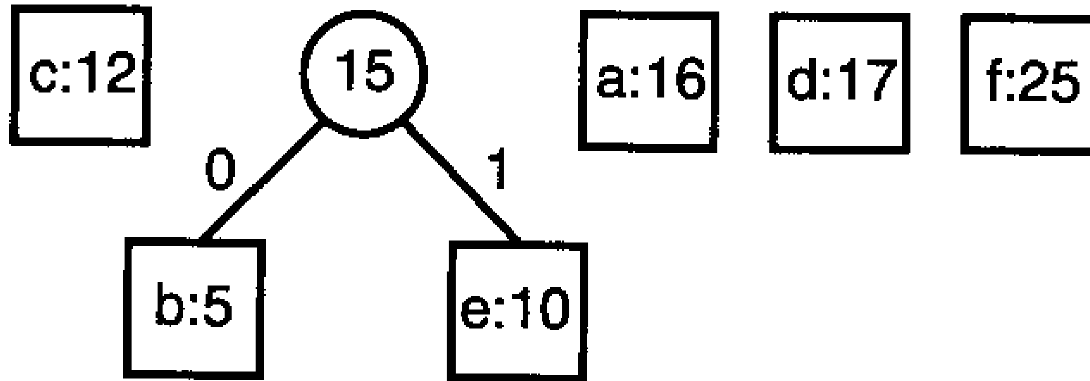
$$bits(T) = \sum_{i=1}^{n} frequency(v_i)depth(v_i)$$

- Compute Bits(C1), Bits(C2), and Bits(C3)

An example of application (1)

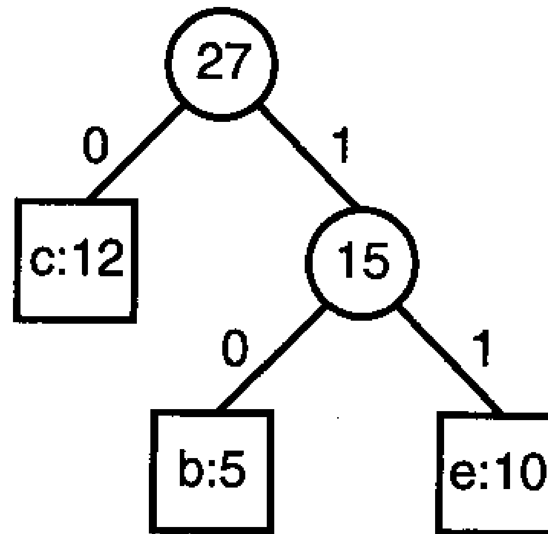
b:5 e:10 c:12 a:16 d:17 f:25

An example of application (2)

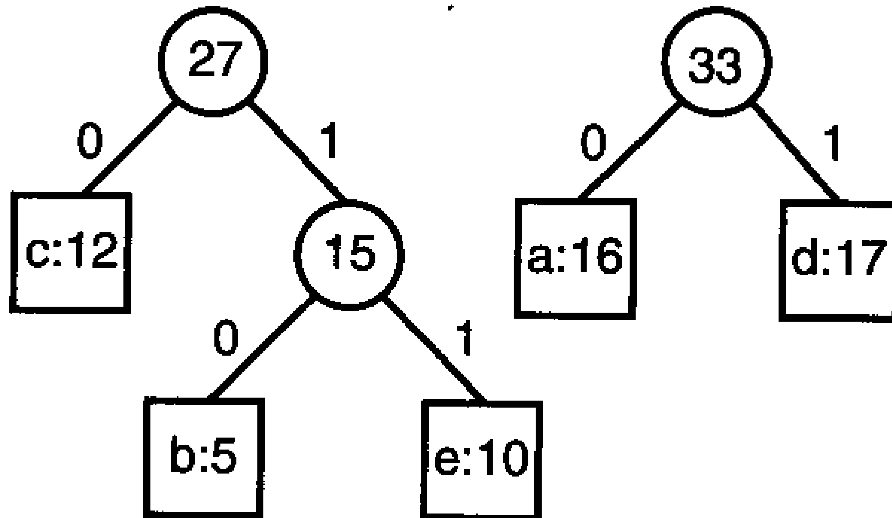


An example of application (3)

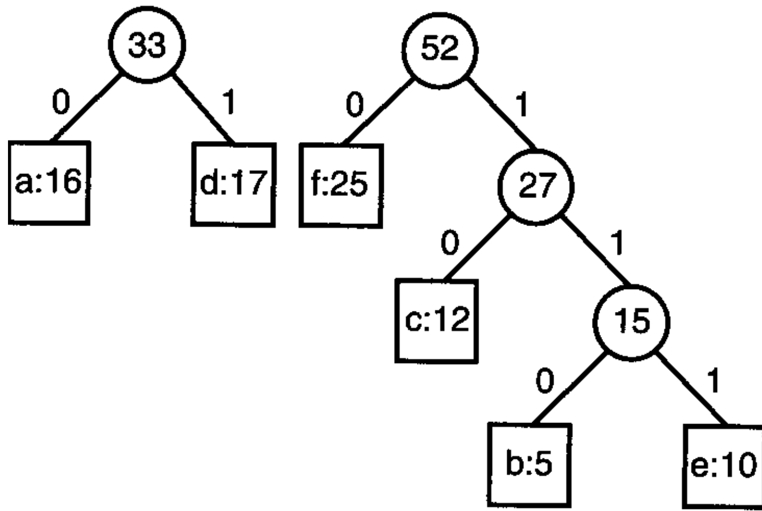
a:16 d:17 f:25

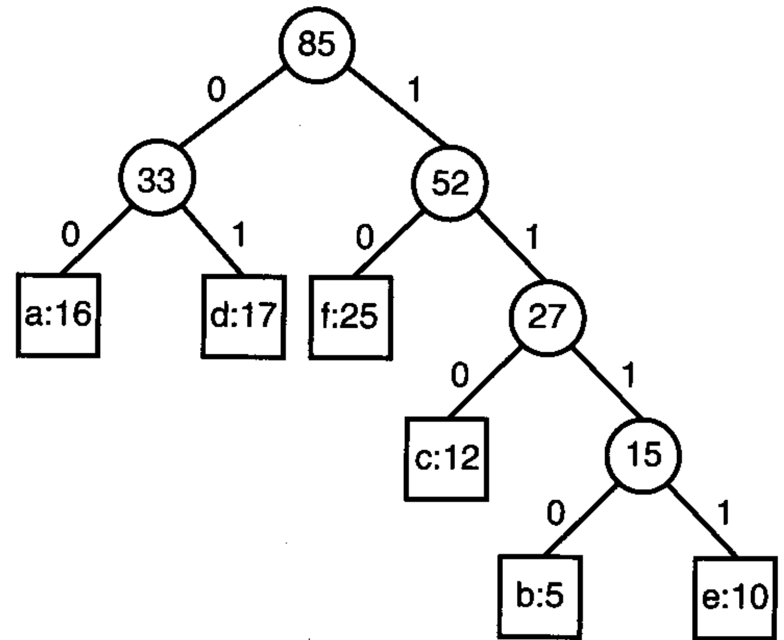


An example of application (4)



An example of application (5)





Priority queue *PQ*

- Arrange *n* pointers to nodetype records in *PQ* so that for each pointer *p* in *PQ*

```
p -> symbol = a distinct character in the file
p -> frequency = the frequency of that character in the file
p -> left = p -> right = NULL
for (i = 1; i < = n-1; i++) { // There is no solution check; rather,
    remove (PQ, p); // solution is obtained when i = n - 1.
    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;
```

Proof

- Lemma 4.4

  The binary tree corresponding to an optimal binary prefix code is full. That is, every nonleaf has two children

- Theorem 4.5

  Huffman's algorithm produces an optimal binary code

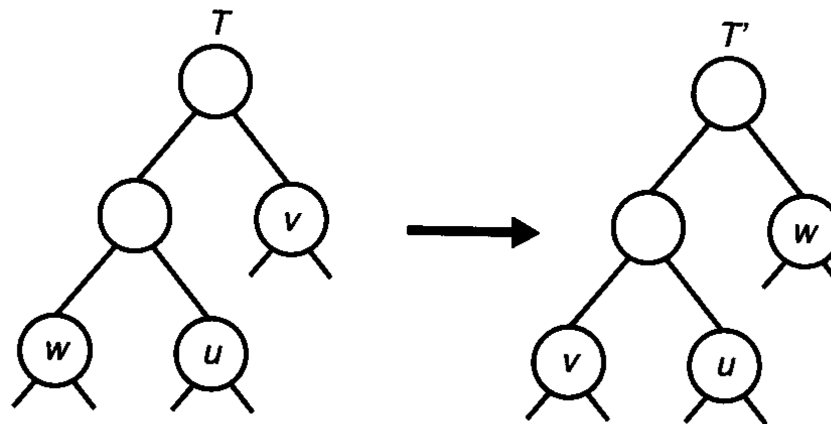


Figure 4.12 ● The branches rooted at v and w are swapped.

The knapsack problem
مساله کوله‌پشتی

- The knapsack problem
  - Let
    - $S = \{item_1, item_2, \ldots, item_n\}$
    - w_i = weight of $item_i$
    - p_i = profit of $item_i$
    - W = maximum weight the knapsack can hold
- **کوله‌پشتی کسری**
- The fractional knapsack problem
- **کوله‌پشتی صفر و یک**
- The 0-1knapsack problem

روش حریصانه برای حل مساله کوله‌پشتی کسری

Item	1	2	3
Profit	50$	60$	140$
Weight	5	10	20

- Knapsack capacity = 30
- مرتب‌سازی آیتم‌ها بر اساس منفعت وزنی
- Total profit in the previous example
  $50 + $140 + (5/10)($60) = $220

روش حریصانه برای حل مساله کوله‌پشتی صفر و یک

- The 0-1knapsack problem
  - Let
    - S = {$item_1$, $item_2$, …, $item_n$}
    - w_i = weight of $item_i$
    - p_i = profit of $item_i$
    - W = maximum weight the knapsack can hold
  - Determine a subset *A* such that

$$\sum_{item_i \in A} p_i \quad is \quad \max imized \quad subject \quad to \quad \sum_{item_i \in A} w_i \leq W$$

در مساله صفر و یک، حریصانه راه حل بهینه نمی‌دهد

- W = 30

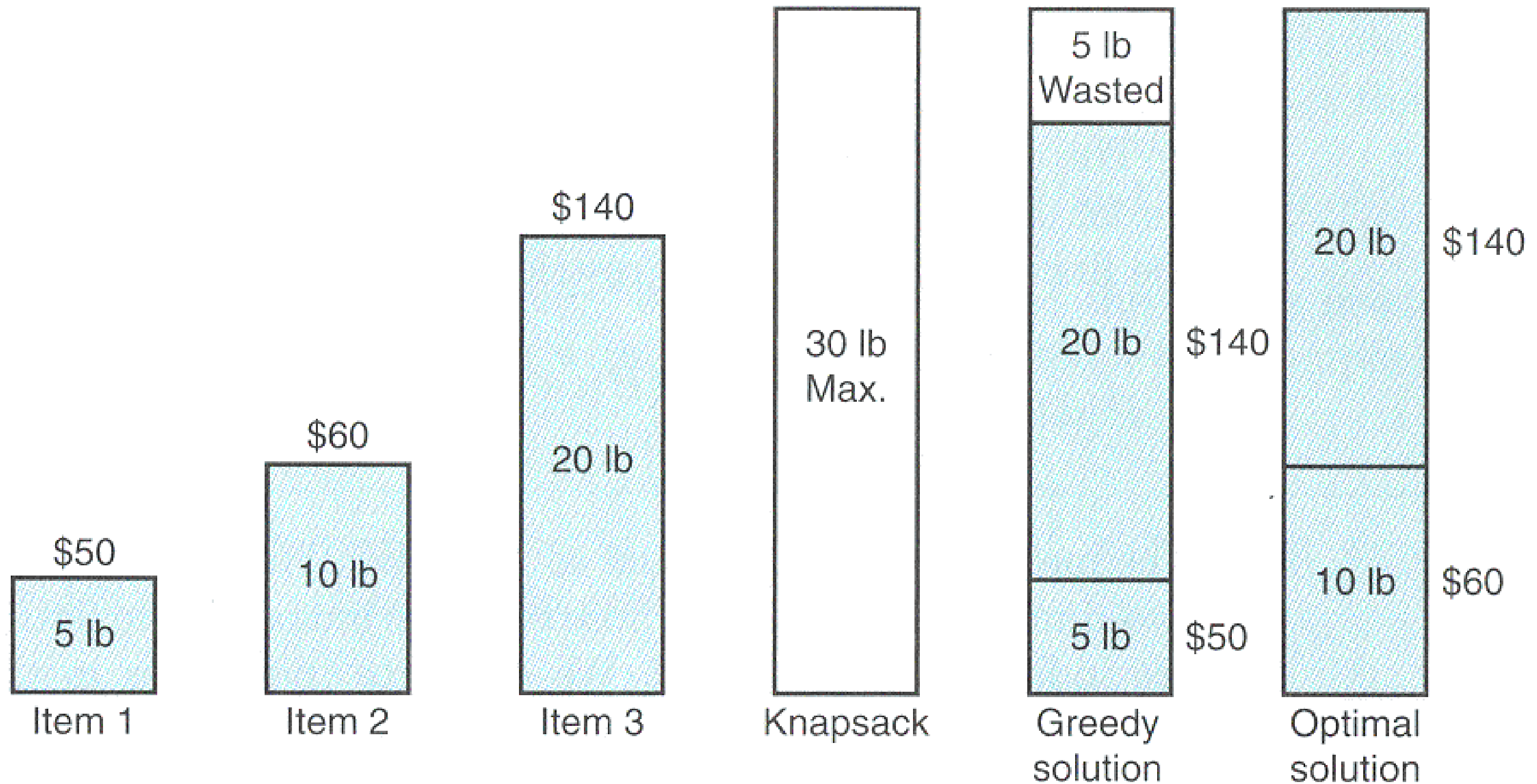


Figure 4.13 • A greedy solution and an optimal solution to the 0-1 Knapsack problem.