Chapter 4

The Greedy Approach

Greedy Approach

- "Scrooge greedily grab as much as gold as he could without considering the past or future."

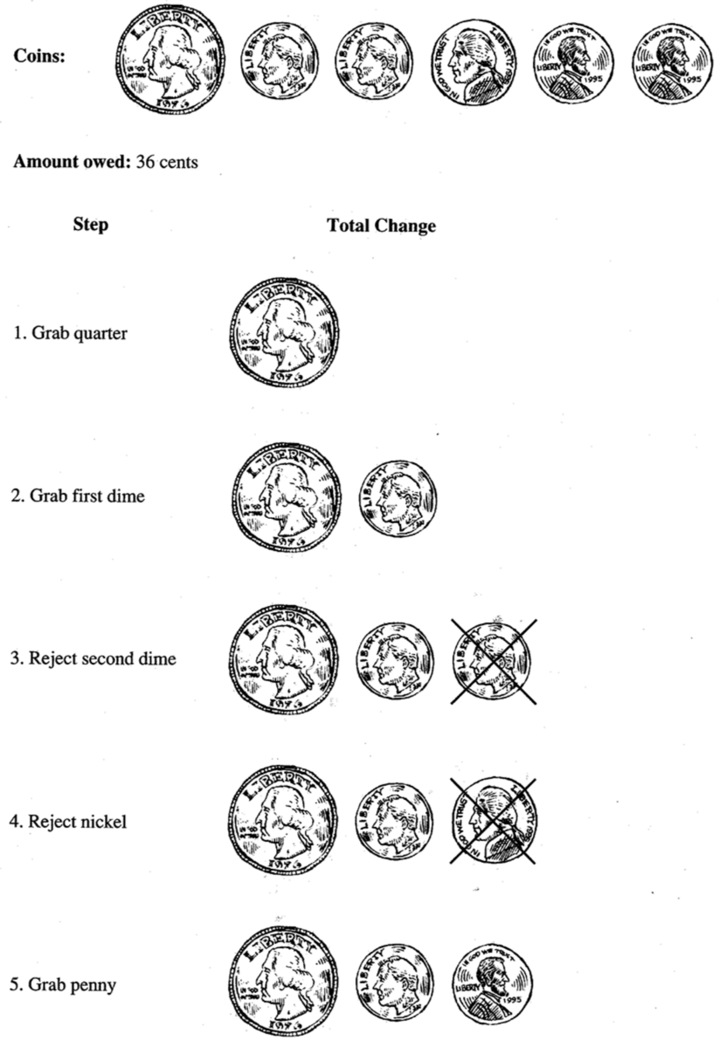
Greedy Approach

- The greedy algorithm grabs data items in sequence
  - that is the "best" at the moment according to some criterion
  - without regard for the choices it has made before or will make in the future.

Greedy Approach

- Example: Coin Exchange
  - There are 1 quarter, 2 dimes, 1 nickels, and 2 pennies. Exchange 36 cents with as smaller number of coins as possible.

Figure 4.1 A greedy algorithm for giving change.



Greedy Approach

- Steps in greedy algorithm:
  - Greedy algorithm starts with an empty set and adds items to the set ***in sequence*** until the set represents a solution to an instance of a problem:

(1) Selection procedure
- Choose the next "best" item according to some criterion and add it to the set.

(2) Feasibility check
- Determine 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.
- If it is feasible, then go to next step.
- Otherwise, discard the chosen item at step (1) from the set. Go to step (1)

(3) Solution check
- Determines whether the new set consitutes a solution to the instance.

Greedy Approach

- Exchange coins by a greedy algorithm:

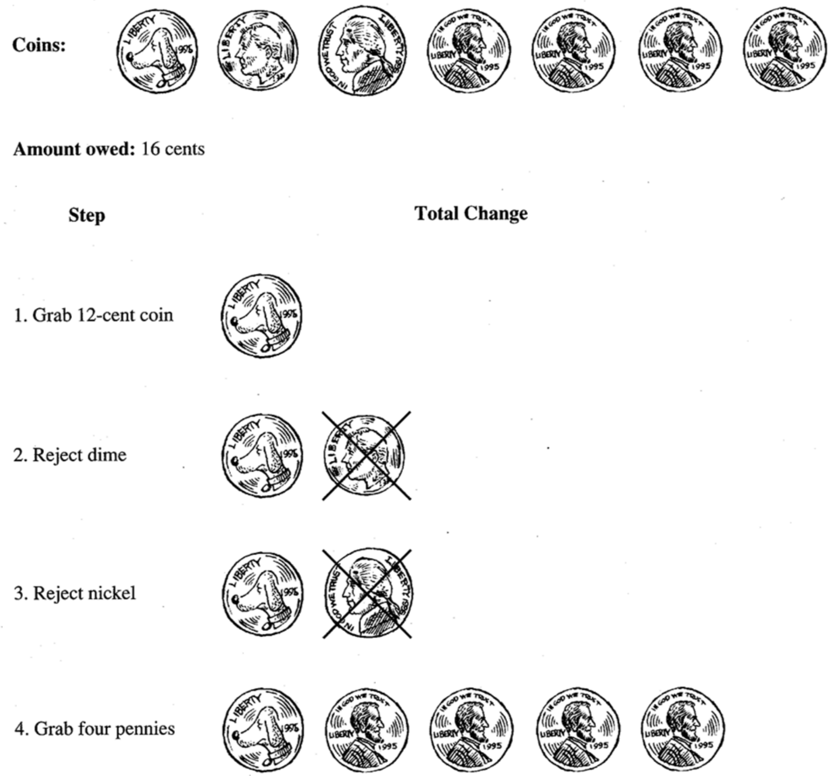
```
While (there are more coins and the instance is not solved) {
    Grab the largest remaining coin;                      // selection procedure

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

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

Greedy Approach

- Coin exchange problem revisited
  - There are one 12-cent coin, 1 dime, 1 nickels, and 4 pennies. Exchange 16 cents with as smaller number of coins as possible.
  - Greedy algorithm gives 5 coins

- We have a solution with 3 coins.
- For general cases, use **dynamic programming**.

Figure 4.2 The greedy algorithm is not optimal if a 12-cent coin is included.

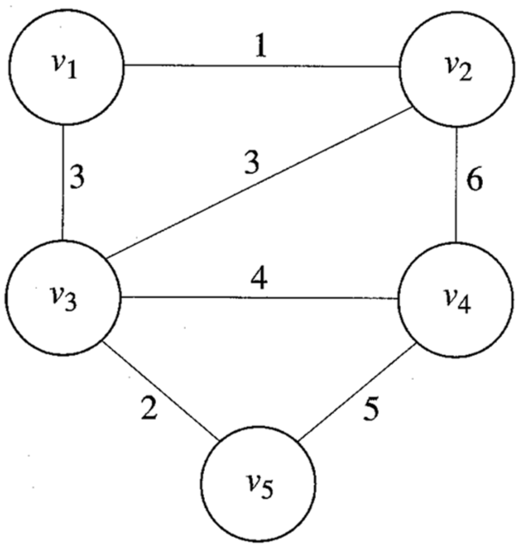


Minimum Spanning Trees

- Undirected graph $G = (V,E)$
  - V : set of vertices
  - E : set of edges

- Example

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



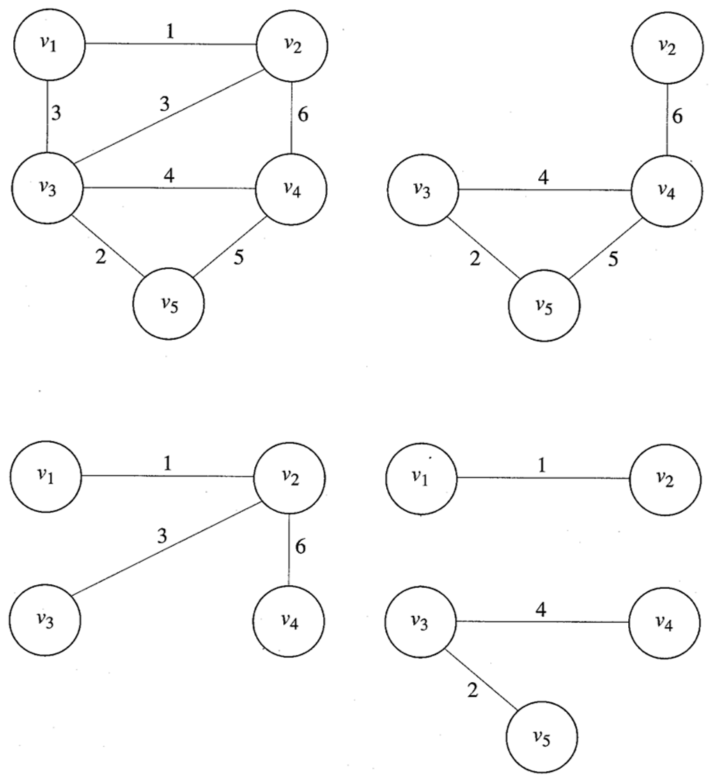
$$V = \{v_1, v_2, v_3, v_4, v_5\}$$

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

Minimum Spanning Trees

❑ Subgraph

- Subgraph *G'=(V',E')* of a graph *G=(V,E)*
  (1) $V' \subseteq V$
  (2) $E' = \{(u,v) \mid u,v \in V'\}$



Minimum Spanning Trees

- Terms
  - **Path**
  - An undirected graph is **connected** if there is a path between every pair of vertices
  - **Cycle**
  - **Cyclic** graph, **Acyclic** graph
  - **Tree**
    - Tree is an acyclic, connected, undirected graph.
  - **Rooted tree**
    - Tree with one vertex is singled out as a root.

Minimum Spanning Trees

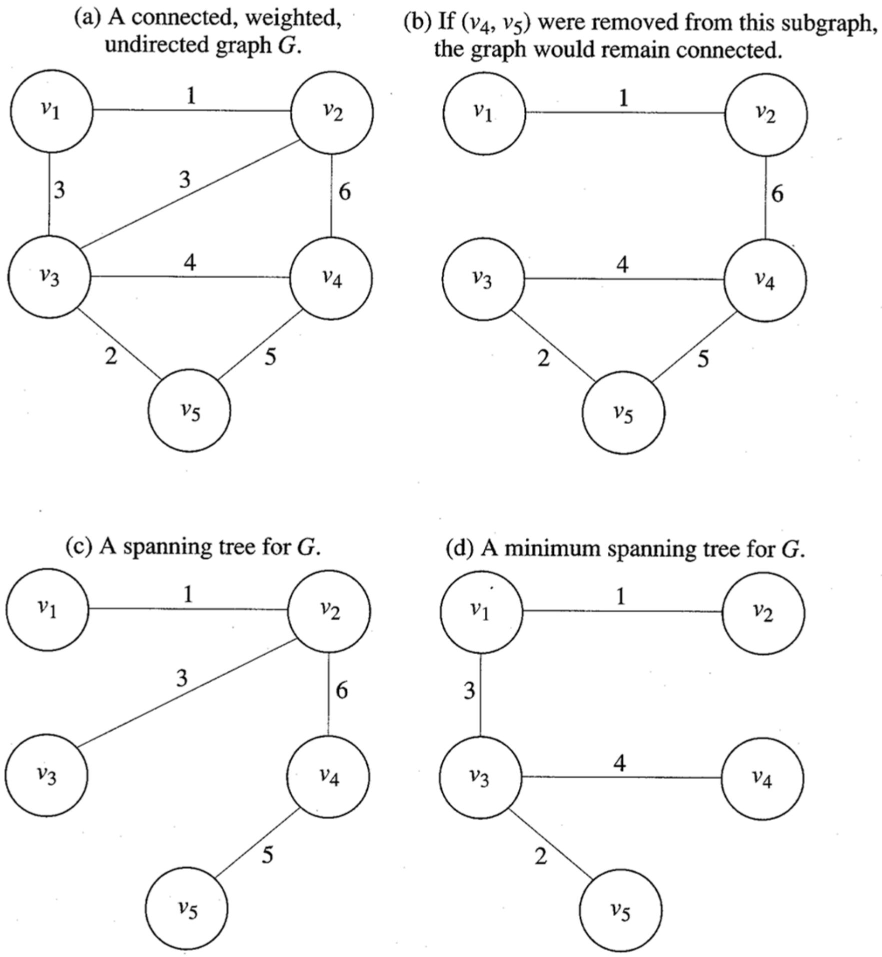
- A **spanning tree** for a graph *G*
  - A connected subgraph of *G* that
    - contains all the vertices of *G*
    - is a tree

- **Minimum spanning tree** of *G*
  - A spanning tree of G that has the minimum sum of weights.

Minimum Spanning Trees

- Example

Figure 4.3 A weighted graph and three subgraphs.



Minimum Spanning Trees

- Minimum spanning tree (MST) problem
  - Given undirected, weighted, connected graph $G=(V,E)$, compute the minimum spanning tree $G'=(V,F)$ of G.

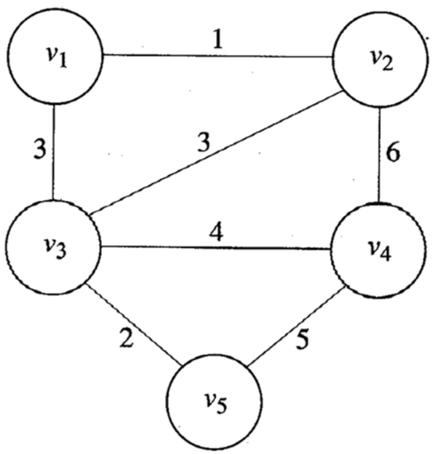
- Two greedy algorithms for MST
  - Prim's algorithm
  - Kruskal's algorithm

Prim's Algorithm

❑ MST

- Given undirected, weighted, connected graph $G=(V,E)$, compute the minimum spanning tree $G'=(V,F)$ of G.

- For a set $Y \subseteq V$ of vertices, a ***nearest*** vertex to Y is a vertex in V-Y that is connected to a vertex in Y by an edge of minimum weight.



A vertex nearest to $\{ v_1 \}$ is v_2
A vertex nearest to $\{ v_1 , v_2 \}$ is v_3
A vertex nearest to $\{ v_1 , v_2 , v_3 \}$ is v_5
A vertex nearest to $\{ v_1 , v_2 , v_3, v_5 \}$ is v_4

Prim's Algorithm

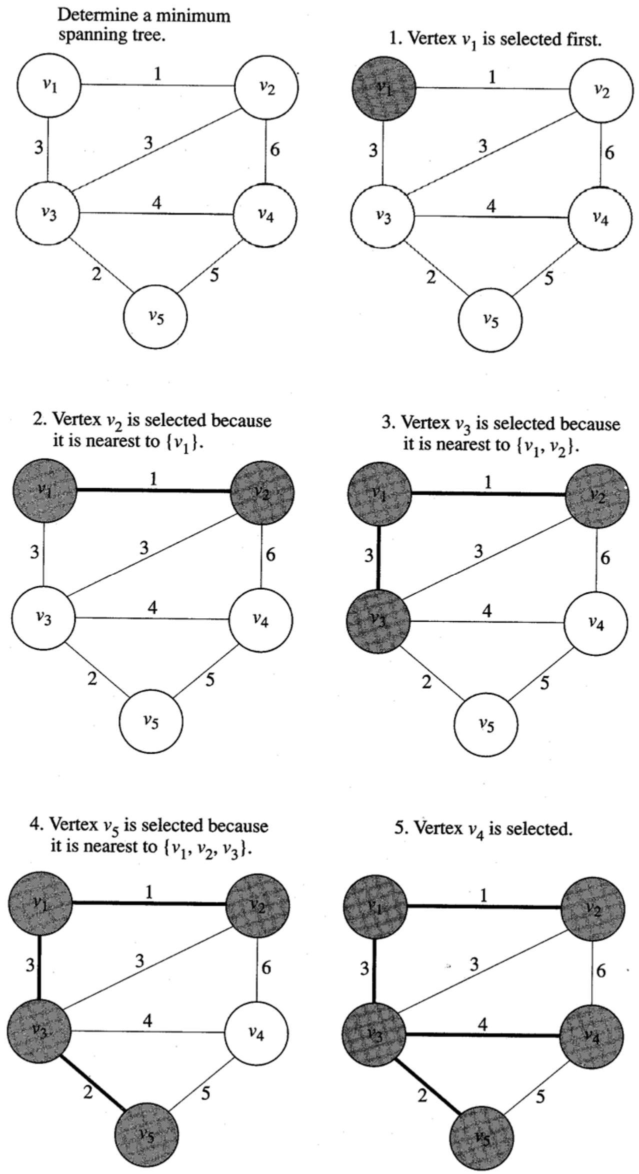
- High-level Prim's algorithm
  - Greedy approach

```
F = ∅;                                      // 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;
}
```

Prim's Algc

❑ Example

Figure 4.4 A weighted graph (in upper left corner) and the steps in Prim's Algorithm for that graph. The vertices in Y and the edges in F are shaded at each step.



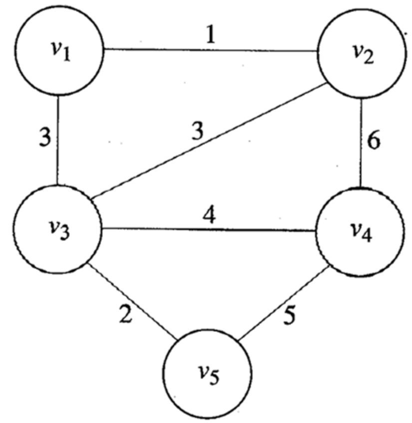
Prim's Algorithm

❑ Implementation of Prim's algorithm

- W : adjacency matrix of G

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

	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



Prim's Algorithm

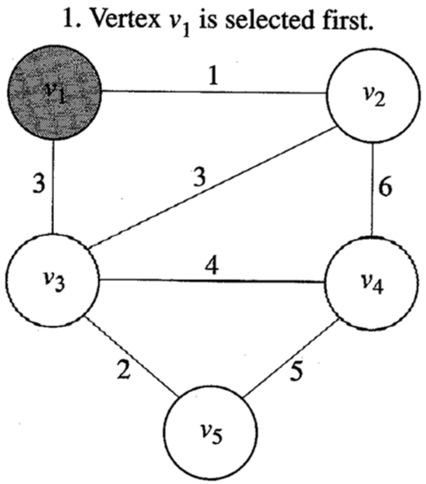
❑ Implementation of Prim's algorithm

- Arrays : *nearest[ ], distance[ ]*
  - *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].*

Prim's Algorithm

❑ Running Prim's algorithm

(1) $Y = \{v_1\}$

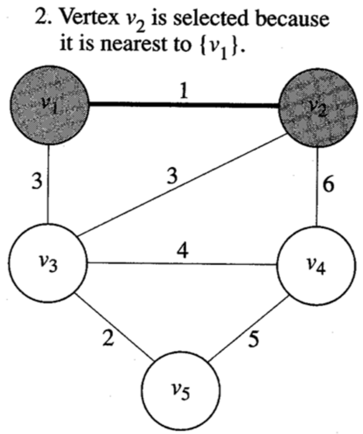


	1	2	3	4	5
nearest		1	1	1	1
distance		1	3	inf	inf

Here, we suppose that vertex 4 and 4 are connected to a vertex 1 by an edge of weight "infinity"

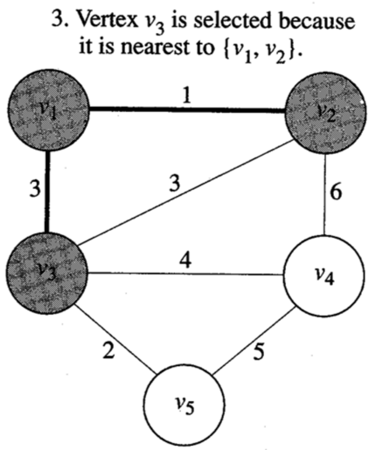
Prim's Algorithm

(2) $Y = \{v_1, v_2\}$



	1	2	3	4	5
nearest			1	2	1
distance			3	6	inf

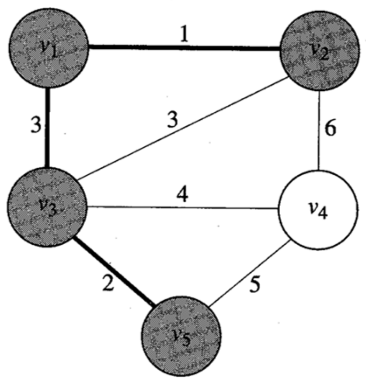
(3) $Y = \{v_1, v_2, v_3\}$



	1	2	3	4	5
nearest				3	3
distance				4	2

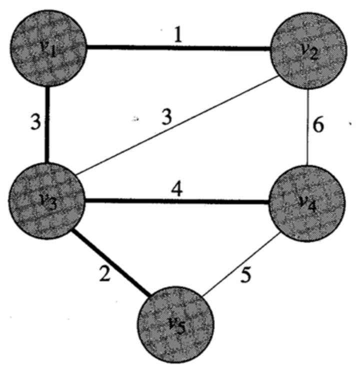
Prim's Algorithm

(4) $Y = \{v_1 , v_2 , v_3 , v_5\}$



	1	2	3	4	5
nearest				3	
distance				4	

(5) $Y = \{v_1 , v_2 , v_3 , v_5 , v_4\}$



	1	2	3	4	5
nearest					
distance					

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 = ∅;
  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.
```

Prim's Algorithm

```
repeat (n − 1 times) {                          // Add all n − 1 vertices to Y.
   min = ∞;
   for (i = 2; i <= n; i++)                     // Check each vertex for
      if (0 ≤ 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;
   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 Y,
         distance[i] = W[i][vnear];             // update its distance from Y.
         nearest[i] = vnear;
      }
  }
}
```

Prim’s Algorithm

- Analysis of Prim's algorithm
  - *Skip*

- Analysis of Prim's algorithm
  - Why Prim's algorithm generates a tree?
  - Why Prim's algorithm generates a minimum spanning tree?

  - *Skip*

Kruskal's Algorithm

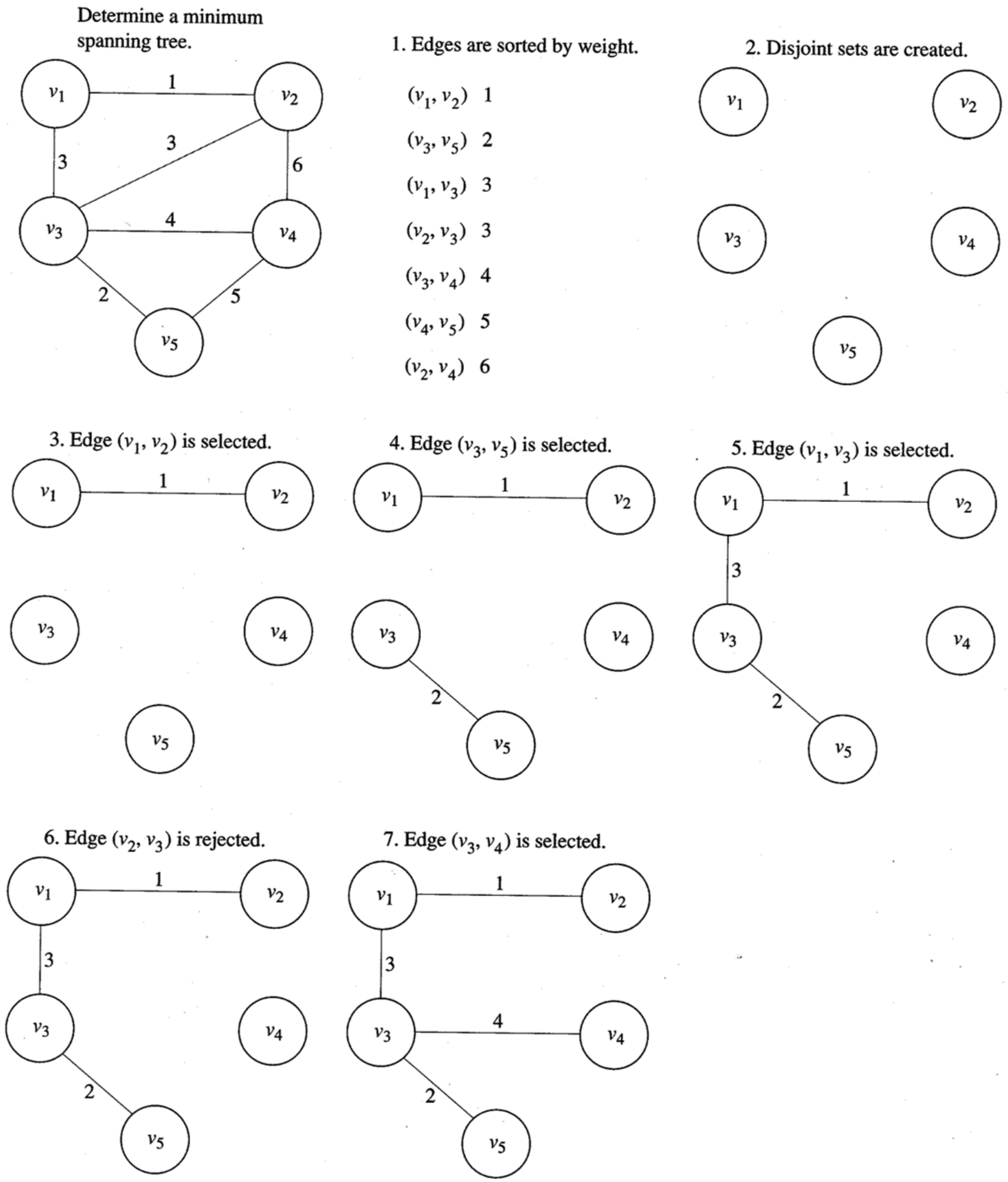
❑ High-level Kruskal's algorithm

```
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        // feasiblity check
                        disjoint subsets) {
      merge the subsets;
      add the edge to F;
   }

   if (all the subsets are merged)              // solution check
      the instance is solved;
}
```

Krus

- Example

Figure 4.7 A weighted graph (in upper left corner) and the steps in Kruskal's Algorithm for that graph.



Kruskal's Algorithm

❑ Implementation of 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 = ∅;
  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;
    }
  }
}
```

Kruskal’s Algorithm

Implementation of Kruskal’s Algorithm

- *initial(n) :*
  - *initialize 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

- Analysis of Kruskal's algorithm
  - *Skip*

- Analysis of Kruskal's algorithm
  - Why Kruskal's algorithm generates a tree?
  - Why Kruskal's algorithm generates a minimum spanning tree?
  - *Skip*

Single-Source Shortest Paths

- Dijkstra's algorithm
  - High level Dijkstra's algorithm

```
Y = {v1};
F = ∅;
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 feasibility check
    in Y as intermediates;
    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
}
```

Dijkstra's Algorithm

- Dijkstra's algorithm is very similar to Prim's algorithm

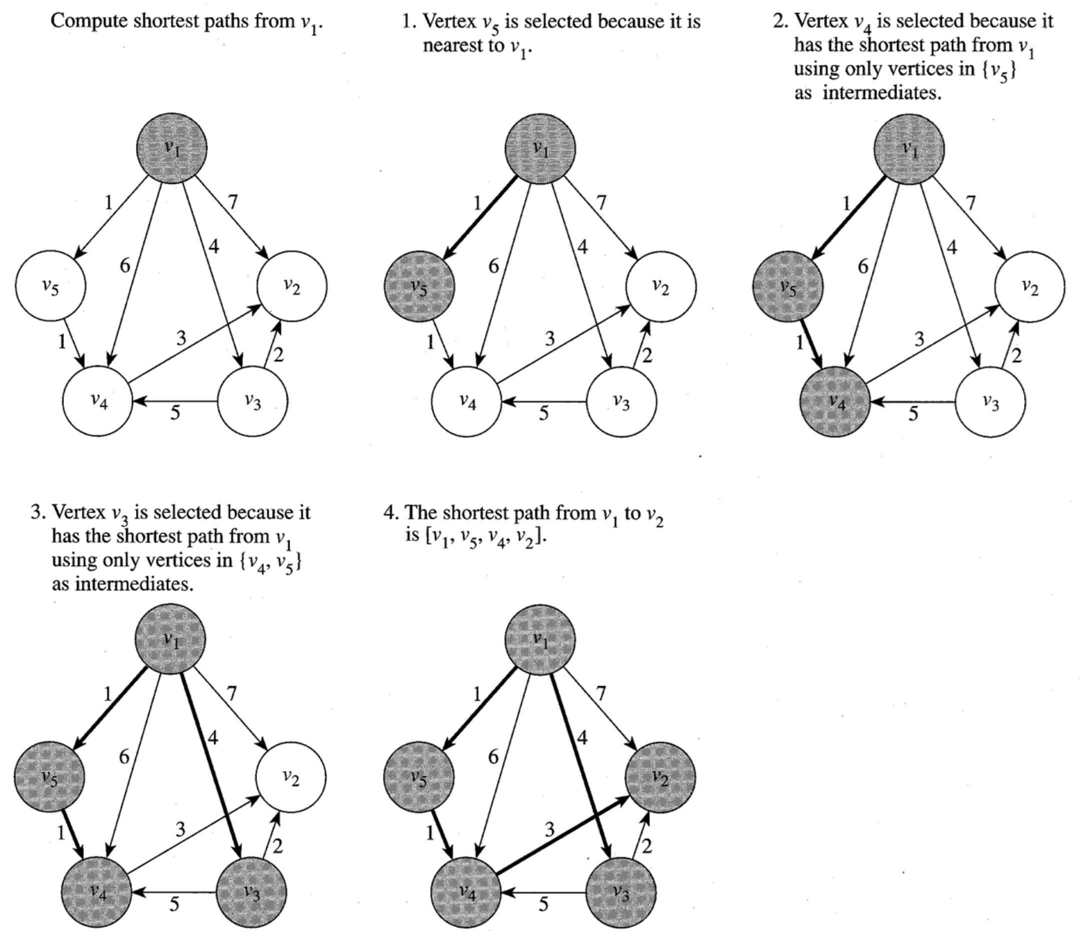
```
Y = {v1};
F = ∅;
while (the instance is not solv
   select a vertex v from V −
   shortest path from v1, usin
   in Y as intermediates;
   add the new vertex v to Y;
   add the edge (on the short
   if (Y == V)
      the instance is solved;
}
```

```
F = ∅;                                  // 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;
}
```

Dijkstra's Algorithm

❑ Example

Figure 4.8 A weighted, directed graph (in upper left corner) and the steps in Dijkstra's Algorithm for that graph. The vertices in *Y* and the edges in *F* are shaded at each step.



Dijkstra's Algorithm

Implementation of Dijkstra's algorithm

- *Arrays : touch[ ], length[ ]*

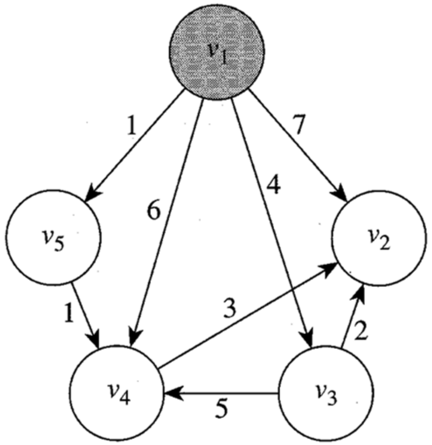
- *touch[i]* = index of vertex v in Y such that the edge < v, v_i> is the last edge on the current shortest path from v_1 to v_i using only vertices in Y as intermediates.

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

Dijkstra's Algorithm

- Running Dijkstra's algorithm

  (1) $Y = \{v_1\}$

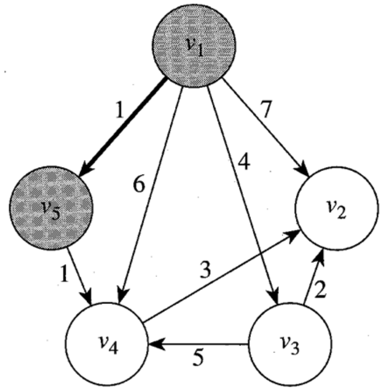


	1	2	3	4	5
touch	1	1	1	1	1
length	0	7	4	6	1

Initially, we set touch[i] = 1 and W[1][i] is copied to length[i],

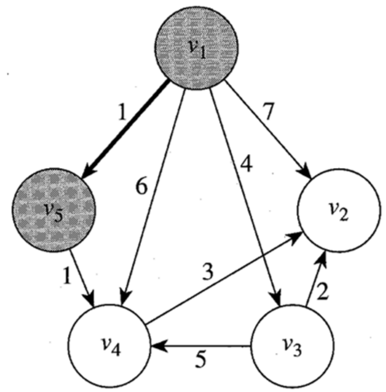
Dijkstra's Algorithm

(2) $Y = \{v_1, v_5\}$



	1	2	3	4	5
touch	1	1	1	5	1
length	0	7	4	2	-1

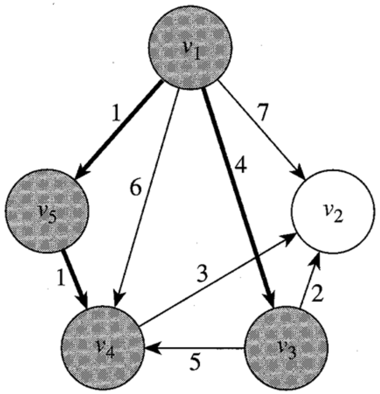
(3) $Y = \{v_1, v_5, v_4\}$



	1	2	3	4	5
touch	1	4	1	5	1
length	0	5	4	-2	-1

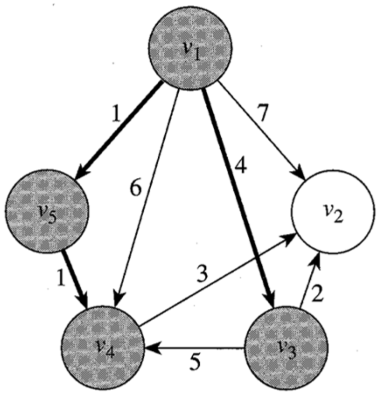
Dijkstra's Algorithm

(4) $Y = \{v_1 , v_5 , v_4 , v_3\}$



	1	2	3	4	5
touch	1	4	1	5	1
length	0	5	-4	-2	-1

(5) $Y = \{v_1 , v_5 , v_4 , v_3, v_2\}$



	1	2	3	4	5
touch	1	4	1	5	1
length	0	-5	-4	-2	-1

```
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 = ∅;
  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.
  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.
}
```

```
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 = ∅;
  for (i = 2; i <= n; i++) {                    // Fo
    touch[i] = 1;                               // to
    length[i] = W[1][i];                        // cu
  }                                             // v1,
                                                // tha
                                                // on
  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[vn
        to vertex indexed by vnear;
    add e to F;
    for (i = 2; i <= n; i++)
      if (length[vnear] + W[vnear][i] < length
        length[i] = length[vnear] + W[vnear]|
        touch[i] = vnear;                       // 
      }                                         // 
    length[vnear] = −1;                         // 
  }                                             // 
}
```

```
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 = ∅;
  for (i = 2; i <= n; i++) {
    nearest[i] = 1;                     // For all ver
    distance[i] = W[1][i];              // to be the
  }                                     // Y and initi
                                        // from Y to
                                        // on the ed

  repeat (n − 1 times) {                // Add all n − 
    min = ∞;
    for (i = 2; i <= n; i++)            // Check each v
      if (0 ≤ distance[i] < min) {      // being nearest
        min = distance[i];
        vnear = i;
      }
    e = edge connecting vertices indexed
        by vnear and nearest[vnear];
    add e to F;
    distance[vnear] = −1;               // Add vertex i
    for (i = 2; i <= n; i++)            // vnear to Y.
      if (W[i][vnear] < distance[i]) {  // For each vert
        distance[i] = W[i][vnear];      // update its dis
        nearest[i] = vnear;
      }
  }
}
```

Dijkstra's Algorithm

- Analysis of Dijkstra's algorithm
  - *Skip*