# Chapter 4:
# The Greedy Approach

# Contents

# Paradigm

- **Scrooge case – never consider past or future**
- **Greedy Algorithm arrives at a solution by making a sequence of choices, each of which simply looks the best at the moment (local optimal).**
- **For a given algorithm, we must determine whether the solution is always optimal.**

# Example – Coin change problem

- **Min. # of coins**
- **Selection procedure: looking for the largest coin in value (greedy)**
- **Feasibility check: exceeding the amount owed?**
- **Solution check: exact change?**
  - **Making 36 out of (25, 10, 10, 5, 5, 1)**
    - **25+10+1=36 3 coins: optimal**
  - **Making 30 out of (25, 10, 10, 10, 1, 1, 1, 1, 1)**
    - **25+1+1+1+1+1=30 6 coins: NOT optimal**
    - **10+10+10=30 3 coins: optimal**
  - **Making 30 out of (25, 10, 10, 10)**
    - **25+ ? can’t find solution**

# High-level coin change procedure

```
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;
}
```

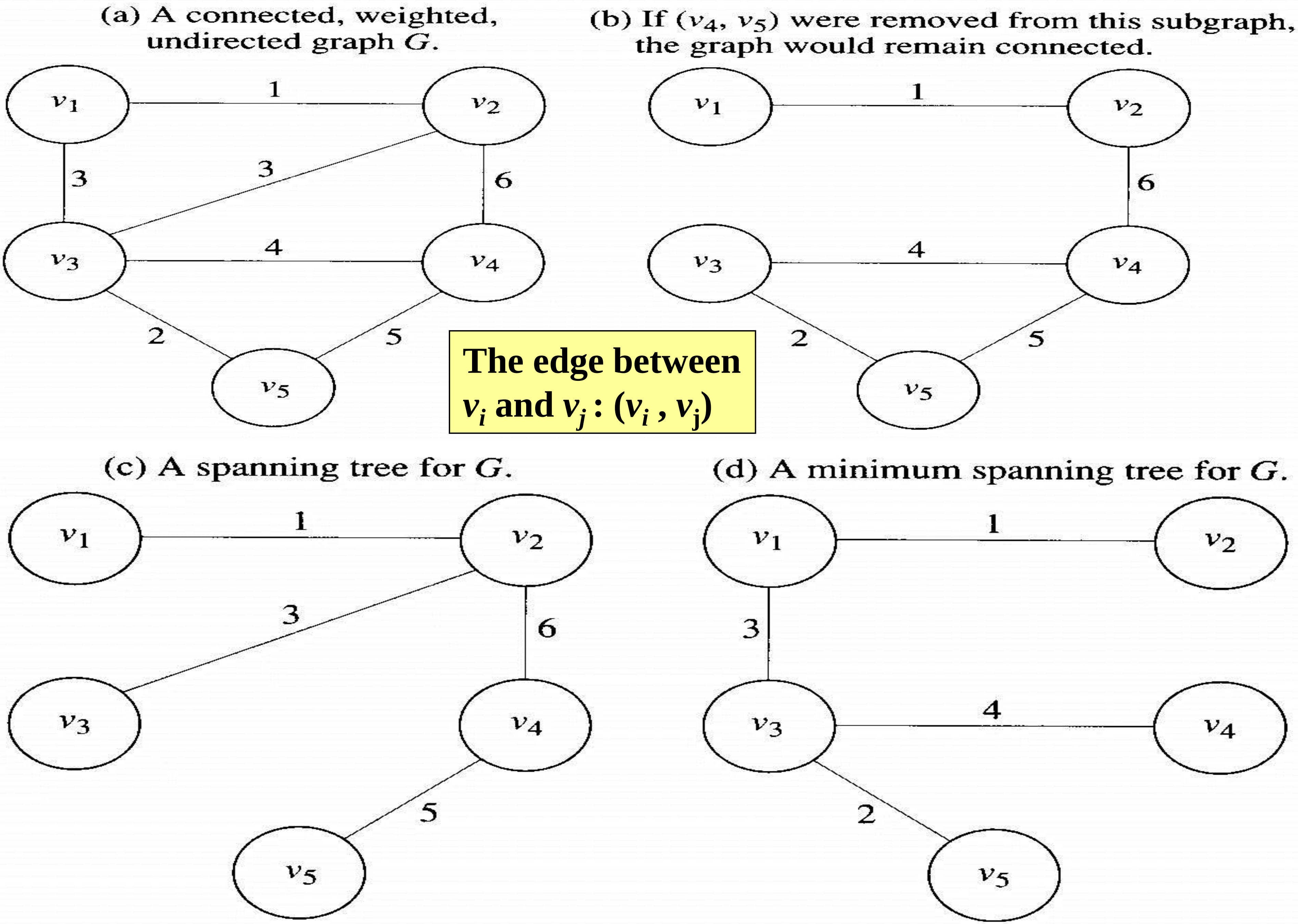
# 4.1 Minimum Spanning Trees

- **G = (V, E) undirected and connected**
- **A spanning tree for G is a connected subgraph that contains all the vertices in G and is a tree.**

  $$T = (V, F), \quad F \subseteq E$$

- **An MST: spanning tree of minimum weights**
  - **See Fig. 4.3**

**Figure 4.3** A weighted graph and three subgraphs.




**The edge between $v_i$ and $v_j$ : $(v_i, v_j)$**

# High-level greedy algorithm

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

# 4.1.1 Prim's Algorithm

- **High-level 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 $V - Y$** **// selection procedure and**

**that is nearest to $Y$ ;** **// feasibility check**

**add the vertex to $Y$ ;**

**add the edge to $F$ ;**

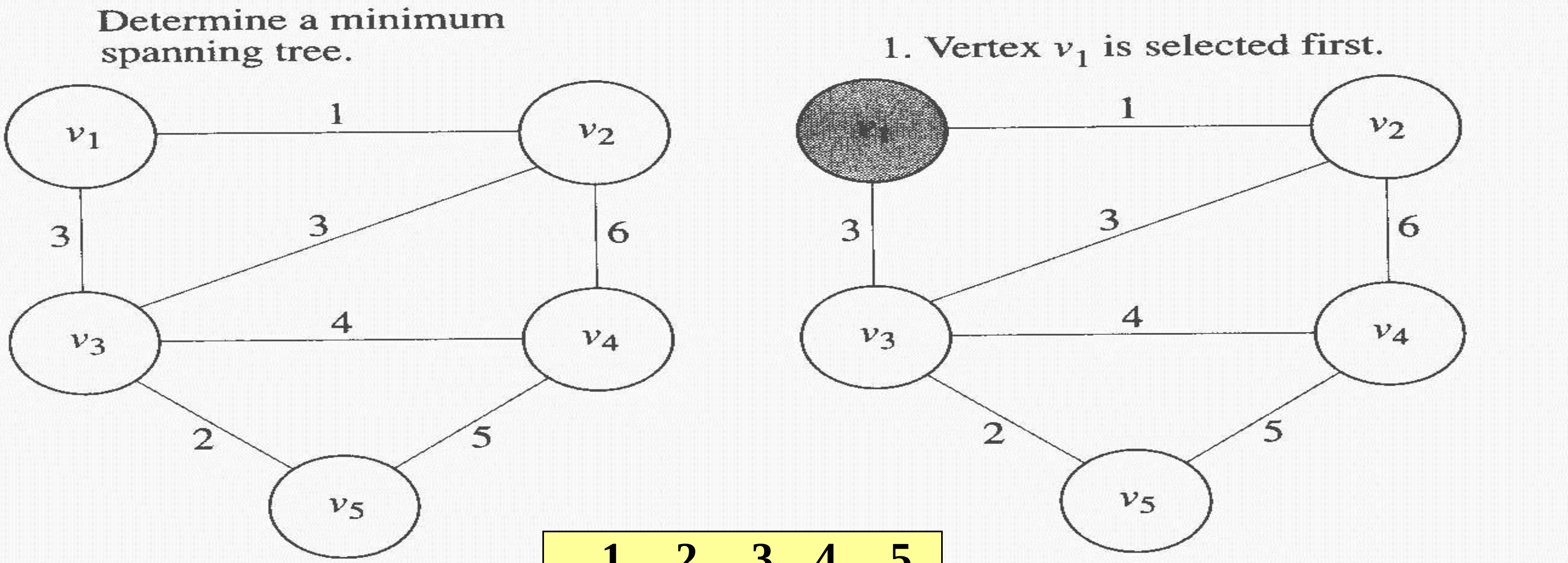
**if ($Y$ == $V$)** **// solution check**

**the instance is solved;**

**}**

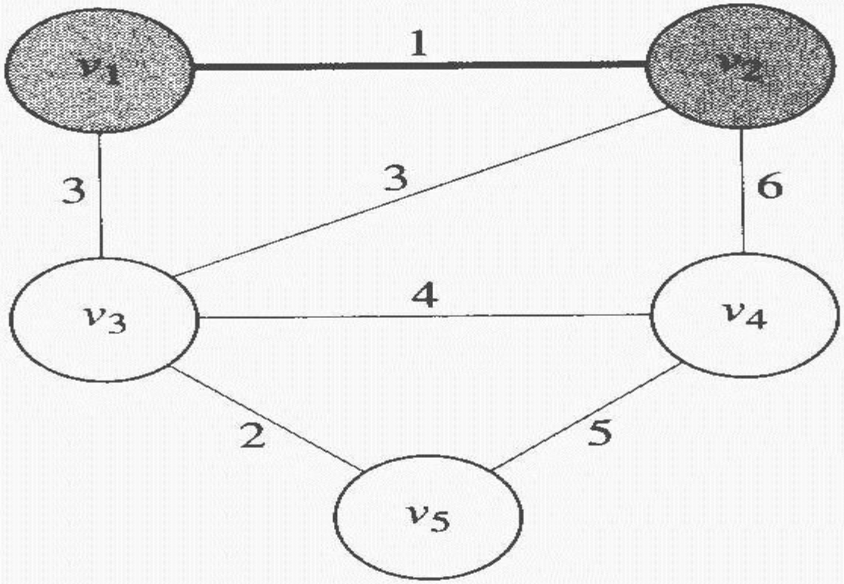
- **The new vertex from *V*-*Y* guarantees that a cycle is not created**
- **See Fig. 4.4**

**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.

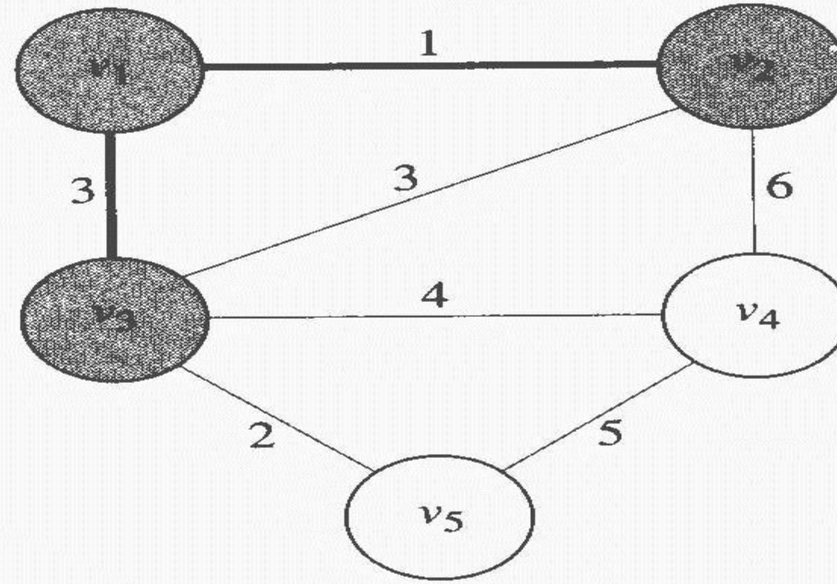



$$
\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 3 & \infty & \infty \\
2 & 1 & 0 & 3 & 6 & \infty \\
3 & 3 & 3 & 0 & 4 & 2 \\
4 & \infty & 6 & 4 & 0 & 5 \\
5 & \infty & \infty & 2 & 5 & 0
\end{array}
$$

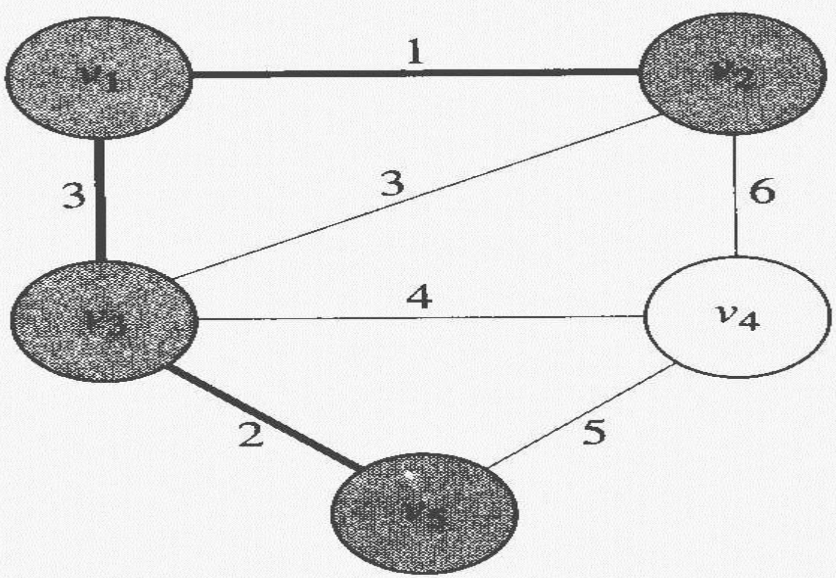
2. Vertex $v_2$ is selected because it is nearest to $\{v_1\}$.



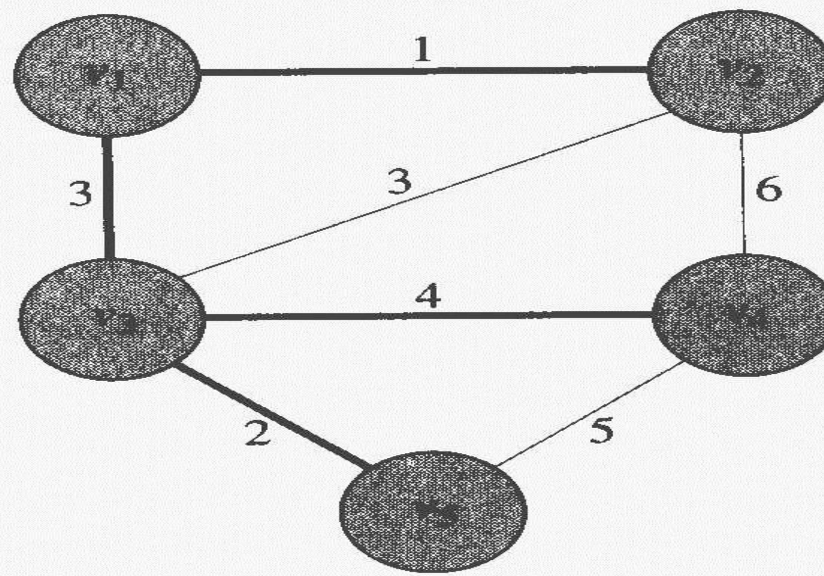

3. Vertex $v_3$ is selected because it is nearest to $\{v_1, v_2\}$.




4. Vertex $v_5$ is selected because it is nearest to $\{v_1, v_2, v_3\}$.




5. Vertex $v_4$ is selected.



# Variables for 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*]**
- **To determine which vertex to add to *Y*, in each iteration we compute the index for which *distance*[*i*] is the smallest. We call this index *vnear*.**
- ***vnear* is the node newly added.**
- **The vertex indexed by *vnear* is added to *Y* by setting *distance*[*vnear*] = -1**

# Steps of Prim's algorithm

**if (W[i][vnear] < distance[i])**

$$
\begin{array}{c}
\begin{array}{cccccc}
 & 1 & 2 & 3 & 4 & 5
\end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\begin{bmatrix}
0 & 1 & 3 & \infty & \infty \\
1 & 0 & 3 & 6 & \infty \\
3 & 3 & 0 & 4 & 2 \\
\infty & 6 & 4 & 0 & 5 \\
\infty & \infty & 2 & 5 & 0
\end{bmatrix}
\end{array}
$$

| *iter* | 1 | | 2 | | 3 | | 4 | |
|---|---|---|---|---|---|---|---|---|
| *i* | *n* | *d* | *n* | *d* | *n* | *d* | *n* | *d* |
| 2 | 1 | $\underline{\underline{1}}$ | 1 | $-1$ | 1 | $-1$ | 1 | $-1$ |
| 3 | 1 | 3 | 1 | $\underline{\underline{3}}$ | 1 | $-1$ | 1 | $-1$ |
| 4 | 1 | $\infty$ | 2 | 6 | 3 | 4 | 3 | $\underline{\underline{4}}$ |
| 5 | 1 | $\infty$ | 1 | $\infty$ | 3 | $\underline{\underline{2}}$ | 3 | $-1$ |
| added | $(v_2,v_1)$ | | $(v_3,v_1)$ | | $(v_5,v_3)$ | | $(v_4,v_3)$ | |

# Algorithm 4.1 Prim's Algorithm (1/3)

- **<u>Problem</u>: Determine an MST.**
- **<u>Inputs</u>: integer $n \geq 2$, and a connected, weighted, undirected graph containing $n$ vertices. The graph is represented by a 2D array $W$, which has both its rows and columns indexed from 1 to $n$, where $W[i][j]$ is the weight on the edge between the $i$-th vertex and the $j$-th vertex.**
- **<u>Outputs</u>: set of edges $F$ in an MST for the graph.**

# Algorithm 4.1 Prim's Algorithm (2/

$$
\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 3 & \infty & \infty \\
2 & 1 & 0 & 3 & 6 & \infty \\
3 & 3 & 3 & 0 & 4 & 2 \\
4 & \infty & 6 & 4 & 0 & 5 \\
5 & \infty & \infty & 2 & 5 & 0
\end{array}
$$

| *iter* | 1 | | 2 | | 3 | | 4 | |
|---|---|---|---|---|---|---|---|---|
| *i* | *n* | *d* | *n* | *d* | *n* | *d* | *n* | *d* |
| 2 | 1 | 1 | 1 | −1 | 1 | −1 | 1 | −1 |
| 3 | 1 | 3 | 1 | 3 | 1 | −1 | 1 | −1 |
| 4 | 1 | ∞ | 2 | 6 | 3 | 4 | 3 | 4 |
| 5 | 1 | ∞ | 1 | ∞ | 3 | 2 | 3 | −1 |
| added | $(v_2, v_1)$ | | $(v_3, v_1)$ | | $(v_5, v_3)$ | | $(v_4, v_3)$ | |

```
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 to be the
      distance[i] = W[1][i];     // nearest vertex in Y and initialize the
   }                             // distance from Y to be the weight on the
                                 // edge to v1.
```

$$
\begin{array}{c|c}
 & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} &
\begin{bmatrix}
0 & 1 & 3 & \infty & \infty \\
1 & 0 & 3 & 6 & \infty \\
3 & 3 & 0 & 4 & 2 \\
\infty & 6 & 4 & 0 & 5 \\
\infty & \infty & 2 & 5 & 0
\end{bmatrix}
\end{array}
$$

# Algorithm 4.1 Prim's Algorithm (3/

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

| *iter* | 1 | | 2 | | 3 | | 4 | |
|---|---|---|---|---|---|---|---|---|
| *i* | *n* | *d* | *n* | *d* | *n* | *d* | *n* | *d* |
| 2 | 1 | $\underline{\underline{1}}$ | 1 | −1 | 1 | −1 | 1 | −1 |
| 3 | 1 | 3 | 1 | $\underline{\underline{3}}$ | 1 | −1 | 1 | −1 |
| 4 | 1 | ∞ | 2 | 6 | 3 | 4 | 3 | $\underline{\underline{4}}$ |
| 5 | 1 | ∞ | 1 | ∞ | 3 | $\underline{\underline{2}}$ | 3 | −1 |
| added | $(v_2, v_1)$ | | $(v_3, v_1)$ | | $(v_5, v_3)$ | | $(v_4, v_3)$ | |

# Analysis of Algorithm 4.1

- **T(n)**
  - **Basic operation: instructions in two for-loops inside the repeat loop**
  - **Input size: n = |V|**

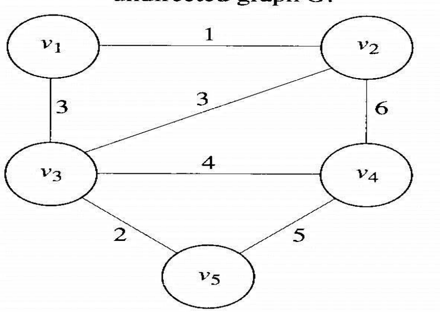
$$T(n) = 2(n-1)(n-1) \in \Theta(n^2)$$

- **Comparison**

| | **DP** | **GA** |
|---|---|---|
| **Algorithm design** | **difficult** | **easy** |
| **Proof** | **easy** | **difficult** |

**Figure 4.3** A weighted graph and thre

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




- **Def.)** $F \subseteq E$ **is called promising if edges can be added to it so as to form an MST.**
  - **Ex) In Fig 4.3(a)**
    - **{($v_1$ $v_2$), ($v_1$ $v_3$)}** **promising**
    - **{($v_2$ $v_4$)}** **not**

## Lemma 4.1

- **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 $F$ U $\{e\}$ is promising.**

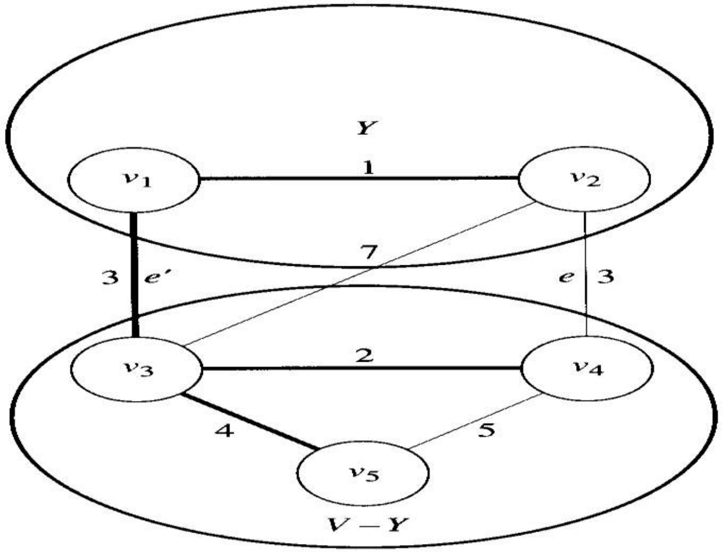



**Figure 4.6** A graph illustrating the proof in Lemma 4.1. The edges in $F'$ are shaded.

- **Proof**
  - **Since $F$ is promising, there must be $F'$ s.t. $F \subseteq F' \wedge (V, F')$ is a MST.**
  - **If $e \in F'$, then $F \cup \{e\} \subseteq F'$, which means $F \cup \{e\}$ is promising.**
  - **If $e \notin F'$, $F' \cup \{e\}$ creates a cycle. (see Fig. 4.6)**
  - **There must be another $e' \in F'$ in the cycle that also connects a vertex in $Y$ to one in $V - Y$.**

$$\text{weight}(e) \le \text{weight}(e') \quad \text{(by assumtion for } e\text{)}$$

  - **So $F' \cup \{e\} - \{e'\}$ is a MST.**
  - **Now $F \cup \{e\} \subseteq F' \cup \{e\} - \{e'\}$ since $e' \notin F$**

    **(by definition of $Y$, edges in $F$ connect only vertices in $Y$)**
  - **Therefore, $F \cup \{e\}$ is promising.**

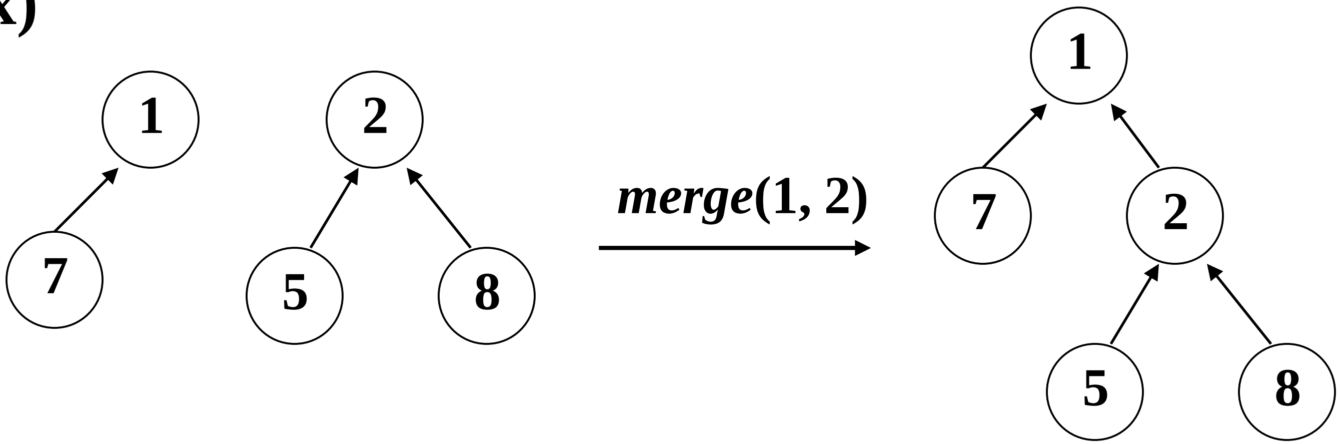
# Theorem 4.1

- **Prim's algorithm always produces an MST.**
- **Proof**
  - **Use induction to show that *F* is promising after each iteration of the repeat loop**
  - **Induction base: ∅ is promising**
  - **Induction hypothesis: *F* is promising after a given iterations.**
  - **Induction step: Because *e* selected in the next iteration is an edge of minimum weight that connects a vertex in *Y* to one in *V* – *Y*, *F* U {*e*} is promising by Lemma 4.1**
  - **Therefore, final set of edges is promising → MST.**

# Appendix C: Data Structures for Disjoint Sets

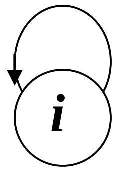
# Operations

- **Definition**
  - ***makeset*(*i*): makes a set out of a member *i* of *U*(universe).**
  - ***find*(*i*): find the set containing *i*.**
  - ***merge*(*p*, *q*): computes the union of set *p* and set *q*.**
- **Set representation: use trees with inverted pointers**
  - **Ex)**

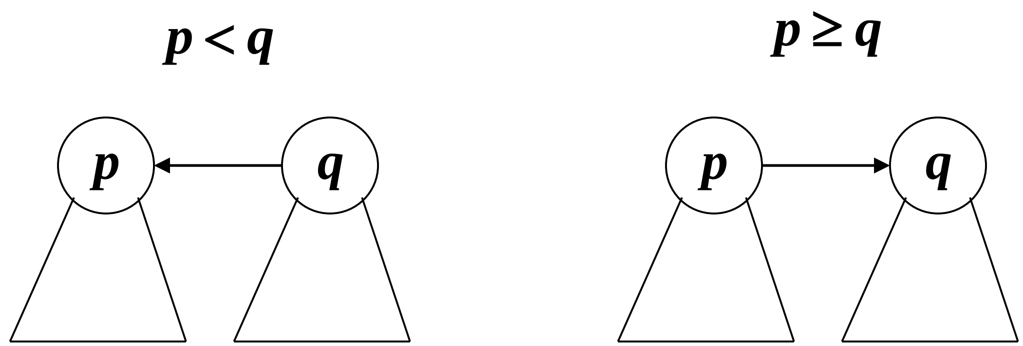


# Illustration

- ***makeset*(*i*)**



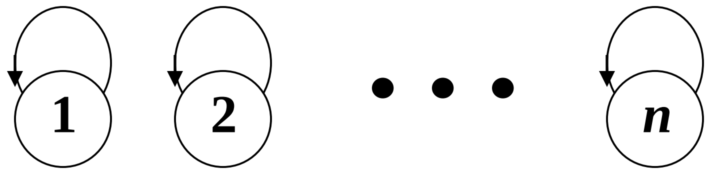

- ***merge*(*p*, *q*)**




- ***find*(*i*): find the root of the set containing *i***
  - **ex) *find*(5) = 1**

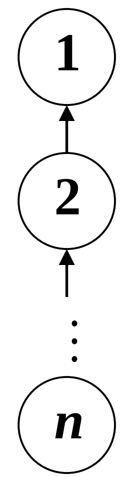
# Example

- $S_i = \{i\} \qquad 1 \le i \le n$




- **Do the following sequence of merge-find operations**

  ***merge*** **(*n* -1, *n*)**
  ***find*** **(*n*)**
  ***merge*** **(*n* -2, *n* -1)**
  ***find*** **(*n*)**
  **…**
  ***merge*** **(1, 2)**



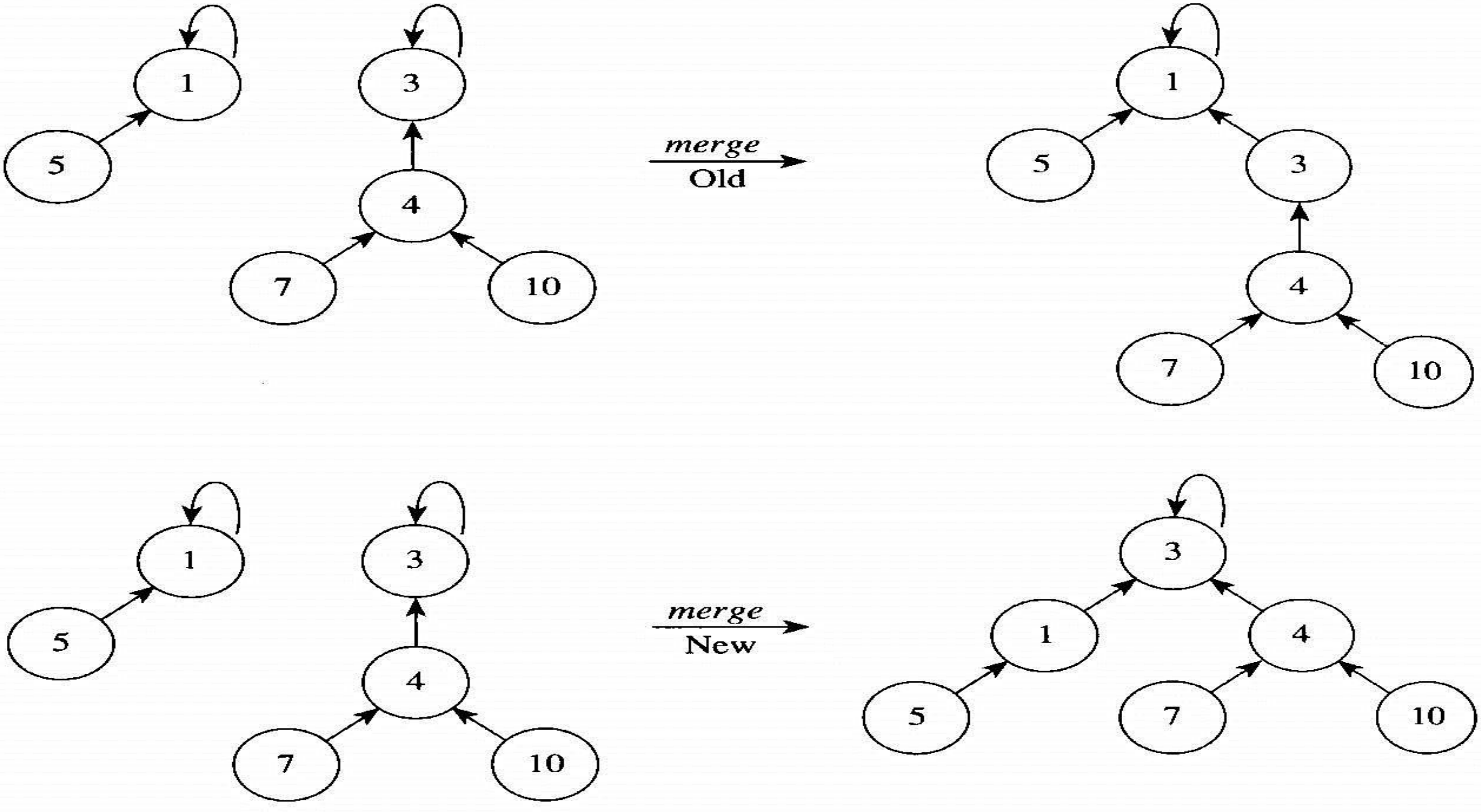

**$n$-1 merges** $\quad O(n)$

**$n$-2 find** $\quad O(\sum_{i=1}^{n-2} i) = O(n^2)$

# Weighting rule

- **Make the tree with the smaller depth (# of nodes) a child**

***Figure C.5*** In the new way of merging, we make the root of the tree with the smaller depth a child of the root of the other tree.



# Theorem C.1

- **Assume that we start with a forest of trees, each having one node. Let *T* be a tree with *m* nodes created as a result of a sequence of merging using weighting rule. Then,** $\underbrace{H(T)}_{\text{height (depth)}} \le \lfloor \log_2 m \rfloor + 1$

- **Proof by induction**

  ***m* = 1 clear**

  **Assume it is true for all trees with ≤ *m* - 1 nodes**

  **Let *T* be a tree with *m* nodes created by merge(*p*, *q*).**

  **tree *p* # of nodes = *a*** $H(p) \le \lfloor \log_2 a \rfloor + 1$

  **tree *q* # of nodes = *m* - *a*** $H(q) \le \lfloor \log_2 (m-a) \rfloor + 1$ **by I.H.**

# Proof continued

- **Without loss of generality we can assume** $a \le \frac{m}{2}$

  **i) If $H(p) = H(q)$**

$$H(T) = H(p) + 1 \le \lfloor \log_2 a \rfloor + 1 + 1 = \lfloor \log_2 2a \rfloor + 1 \le \lfloor \log_2 m \rfloor + 1$$

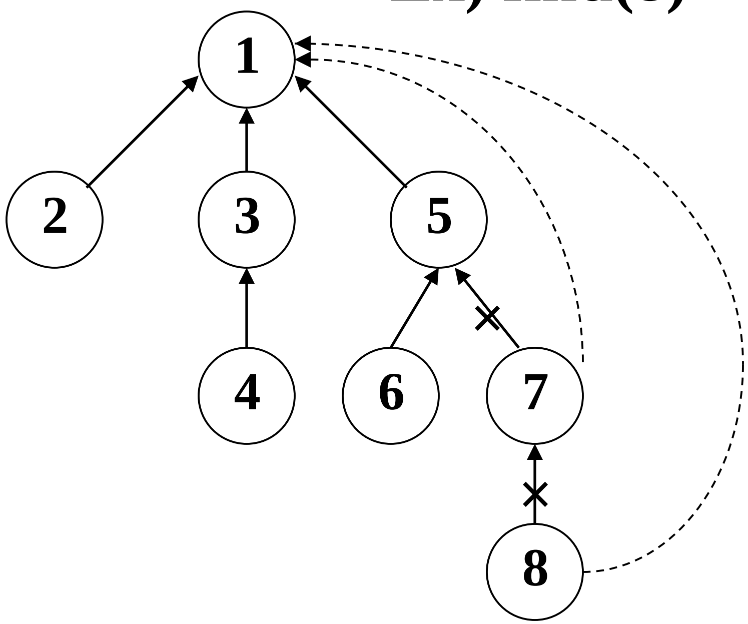
  **ii) If $H(p) < H(q)$**

$$H(T) = H(q) \le \lfloor \log_2 (m - a) \rfloor + 1 \le \lfloor \log_2 m \rfloor + 1$$

- **Complexity of a *find* in m-node tree : O(log *m*)**
- ***n* merges and *n* finds : O(*n* log *n*)**

# Collapsing rule (path compression)

- **We must use # of nodes**
- **Modify find algorithm**
  - **If *j* is a node on the path from *i* to its root, set *parent*[*j*] to root.**
  - **Ex) find(8)**




- ***n* find(8)**
- **Without collapsing: 3*n* moves**
- **With collapsing**
  - **first find 3 moves 2(+1) link change**
  - **remaining *n* -1 moves**
- **We can show that**
  - ***n* finds and *m* merges** $\approx O(n+m)$