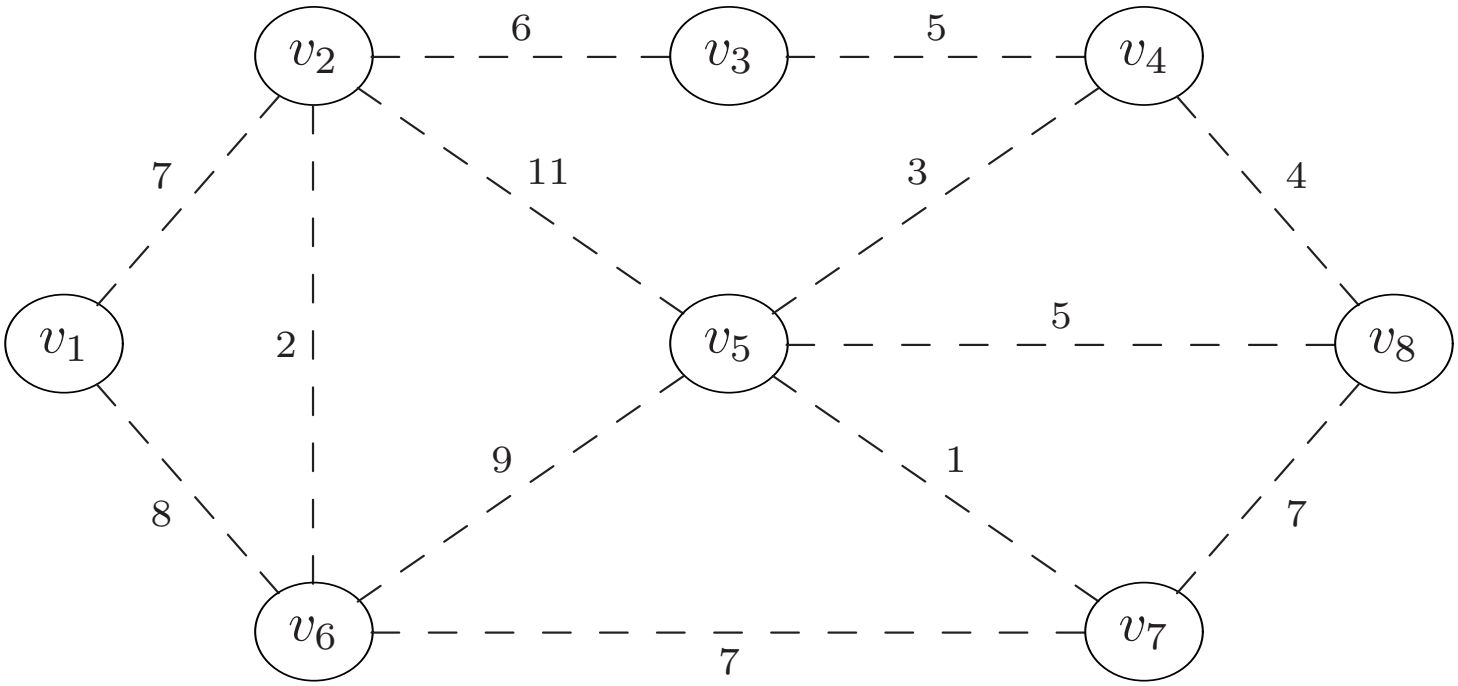
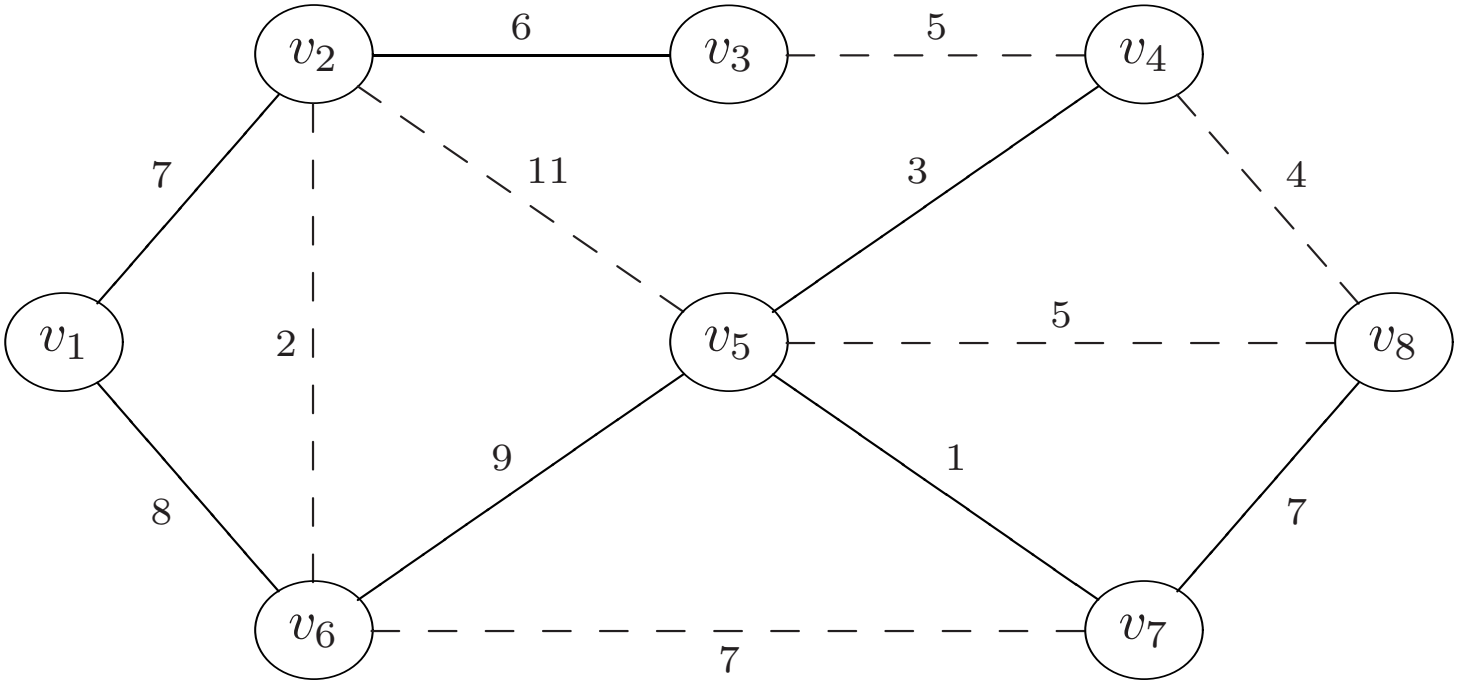
Minimum Spanning Tree

Edge Weights: Example



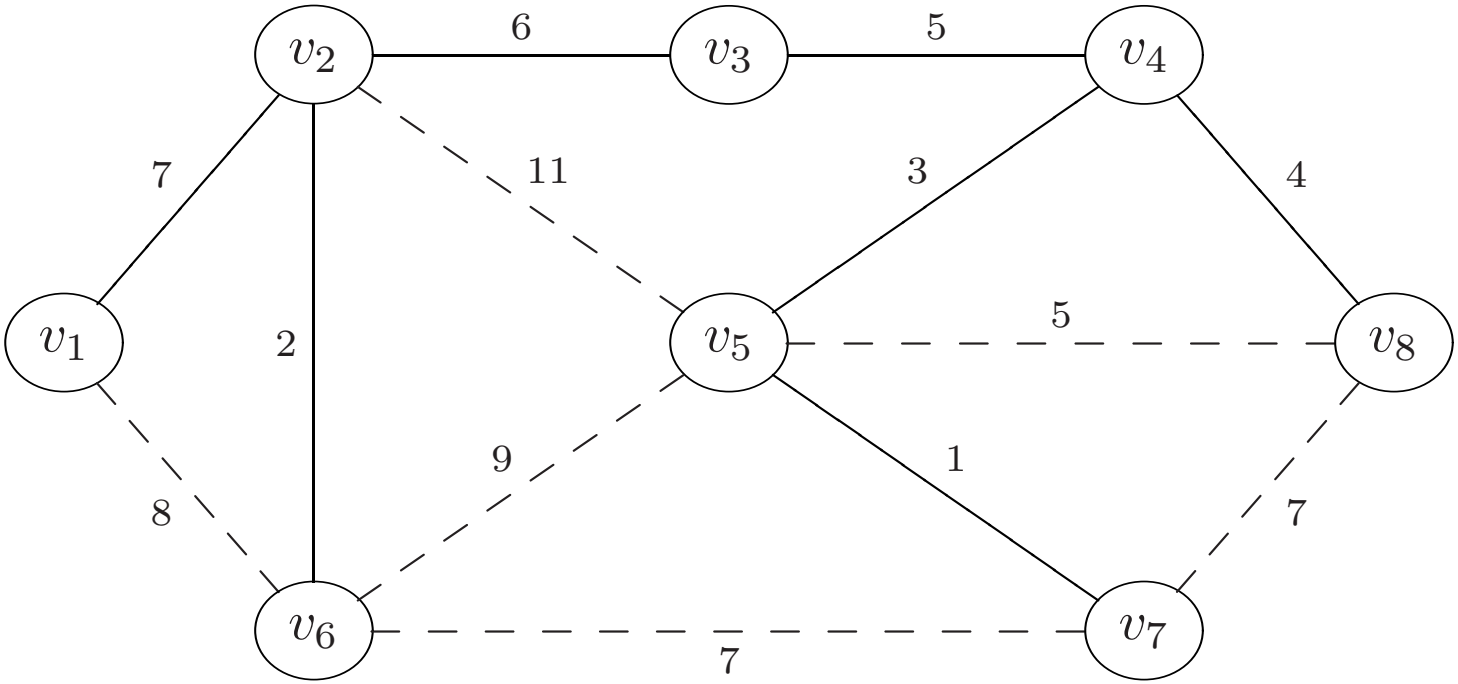
Spanning Tree



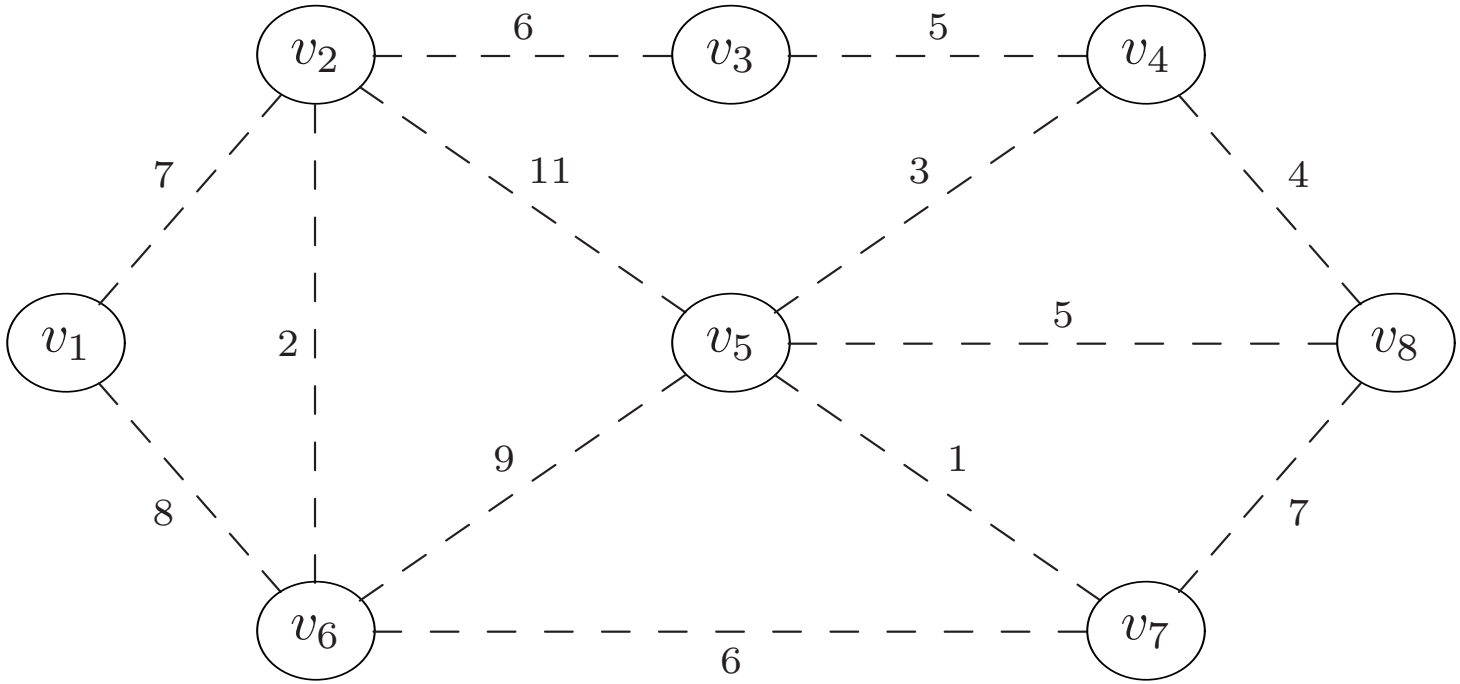
A **spanning tree** of graph G is a tree composed of all the vertices and some of the edges of G.

The **weight** of a spanning tree is the sum of the edge weights.

Minimum (Weight) Spanning Tree: Example



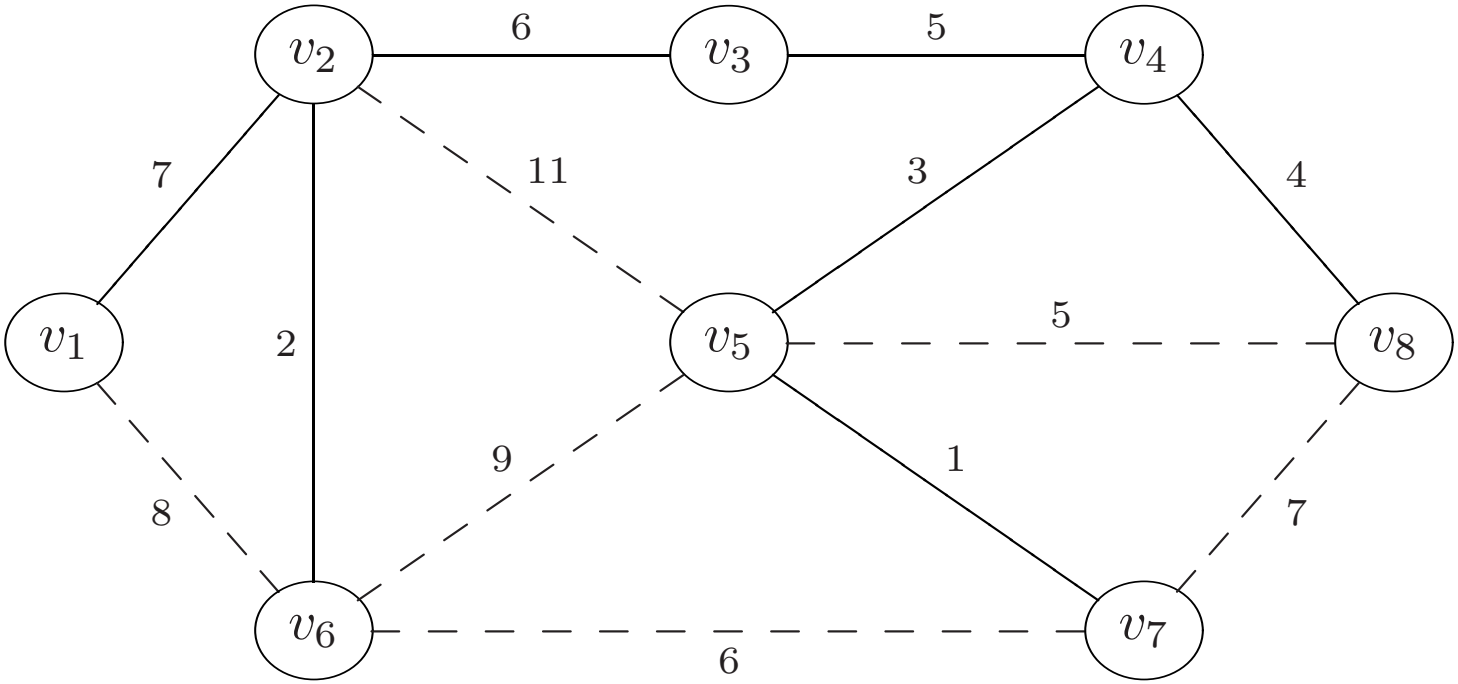
Minimum (Weight) Spanning Tree: Example 2



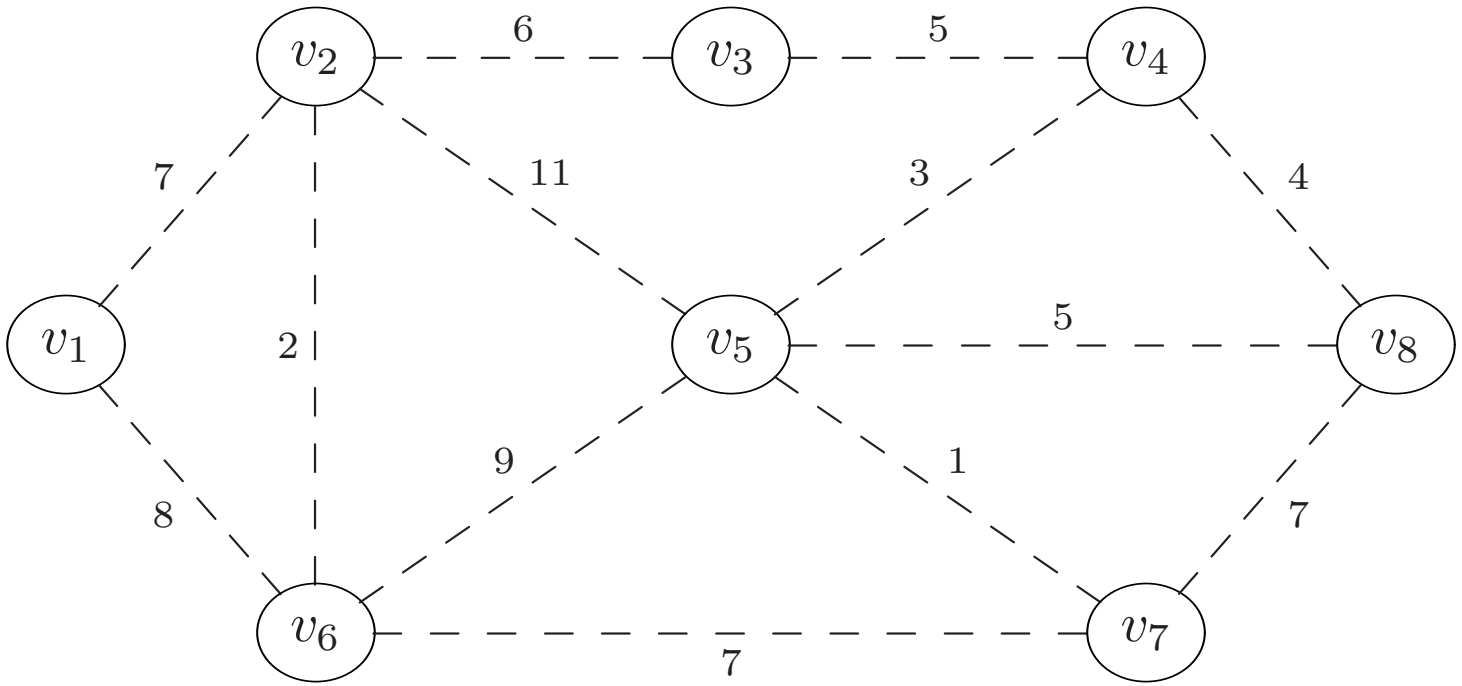
Note: Weight of edge (v_6, v_7) is 6 (previously was 7).

Two Minimum (Weight) Spanning Trees

What is the other minimum (weight) spanning tree of this graph?



Minimum (Weight) Spanning Tree: Example



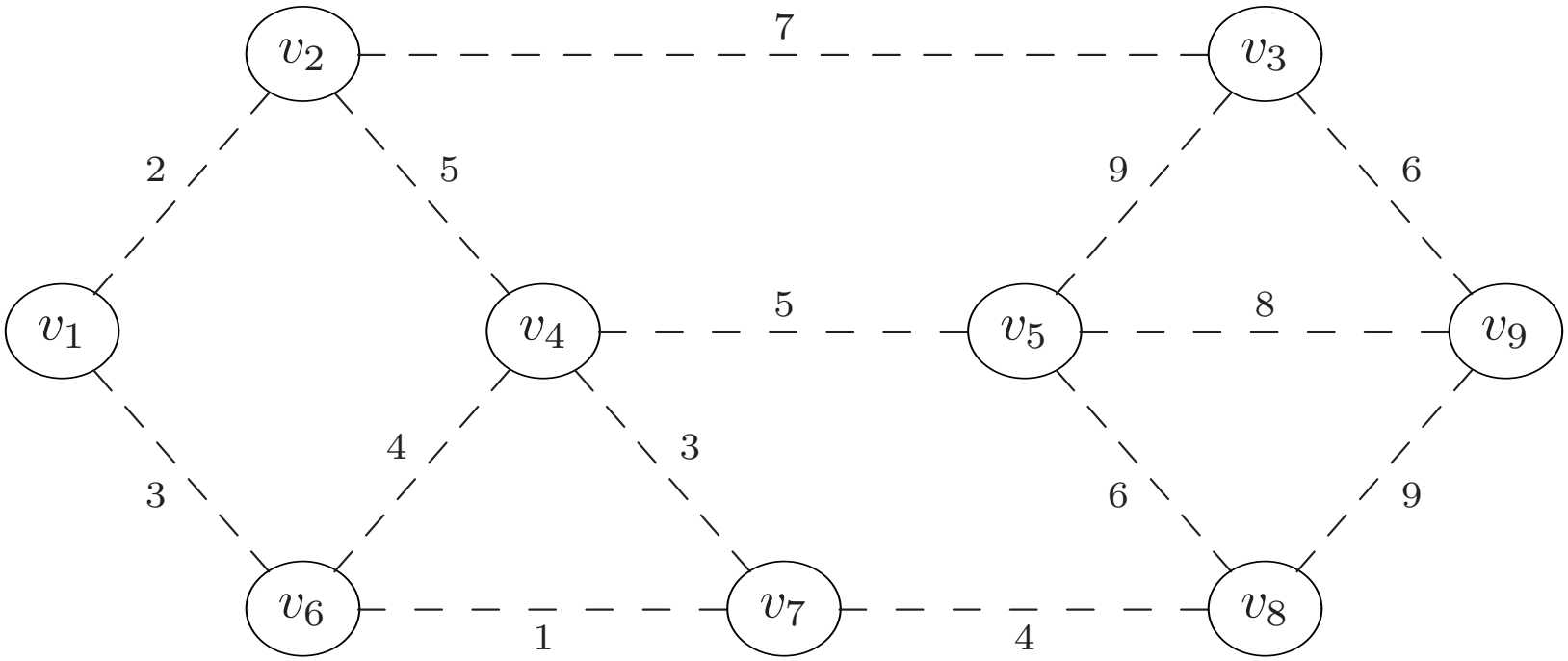
Prim's Minimum Spanning Tree Algorithm

Input : Weighted graph G

Output : Minimum spanning tree of G

```
   procedure PrimMST(G)
1  U ← G.V − {v1} ;        /* G.V = set of vertices of graph G */
2  v1.MSTparent ← NULL;
3  while (U ≠ ∅) and (∃ edge from (G.V − U) to U) do
4  |  (vi, vj) ← minimum weight edge from G.V − U to U;
5  |  vj.MSTparent ← vi;
6  |  U ← U − {vj};
7  end
```

Minimum Spanning Tree: Example 2



MST Lemma

MST Lemma: Let A be a spanning tree of graph G.
Let e_B be an edge of G which is not in A.

1. $A \cup \{e_B\}$ contains a cycle C.
2. If $e_A \neq e_B$ is an edge of cycle C, then
   $B = A \cup \{e_B\} - \{e_A\}$ is a spanning tree of G.
3. If $\mathsf{weight}(e_B) \leq \mathsf{weight}(e_A)$, then $\mathsf{weight}(B) \leq \mathsf{weight}(A)$.

Prim's Minimum Spanning Tree Algorithm

procedure `PrimMST(G)`

1 $U \leftarrow \mathsf{G}.V - \{v_1\}$; /* $\mathsf{G}.V$ = *set of vertices of graph* G */

2 $v_1.\mathsf{MSTparent} \leftarrow$ **NULL**;

3 **while** $(U \neq \emptyset)$ **and** ($\exists$ edge from $(\mathsf{G}.V - U)$ to U) **do**

4 | $(v_i, v_j) \leftarrow$ minimum weight edge from $\mathsf{G}.V - U$ to U;

5 | $v_j.\mathsf{MSTparent} \leftarrow v_i$;

6 | $U \leftarrow U - \{v_j\}$;

7 **end**

MST Theorem: Let T be a subtree of a minimum spanning tree. If e is a minimum weight edge connecting T to some vertex not in T, then $T \cup \{e\}$ is a subtree of a minimum spanning tree.

Proof of MST Theorem

MST Theorem: Let T be a subtree of a minimum spanning tree. If e is a minimum weight edge connecting T to some vertex not in T, then $T \cup \{e\}$ is a subtree of a minimum spanning tree.

Proof. By the theorem's hypothesis, T is a subtree of some minimum spanning tree A of G.

If e is not in A, then $A \cup \{e\}$ contains a cycle C. (MST Lemma)

Some edge e_A of $C - \{e\}$ must be an edge from T to a vertex not in T.

Let $B = A \cup \{e\} - \{e_A\}$.

B is a spanning tree of G. (MST Lemma)

Since e is a minimum weight edge from T to vertices not in T, $\mathsf{weight}(e) \leq \mathsf{weight}(e_A)$.

Thus, $\mathsf{weight}(B) \leq \mathsf{weight}(A)$. (MST Lemma)

Therefore, B is a minimum spanning tree of G containing $T \cup \{e\}$.

□

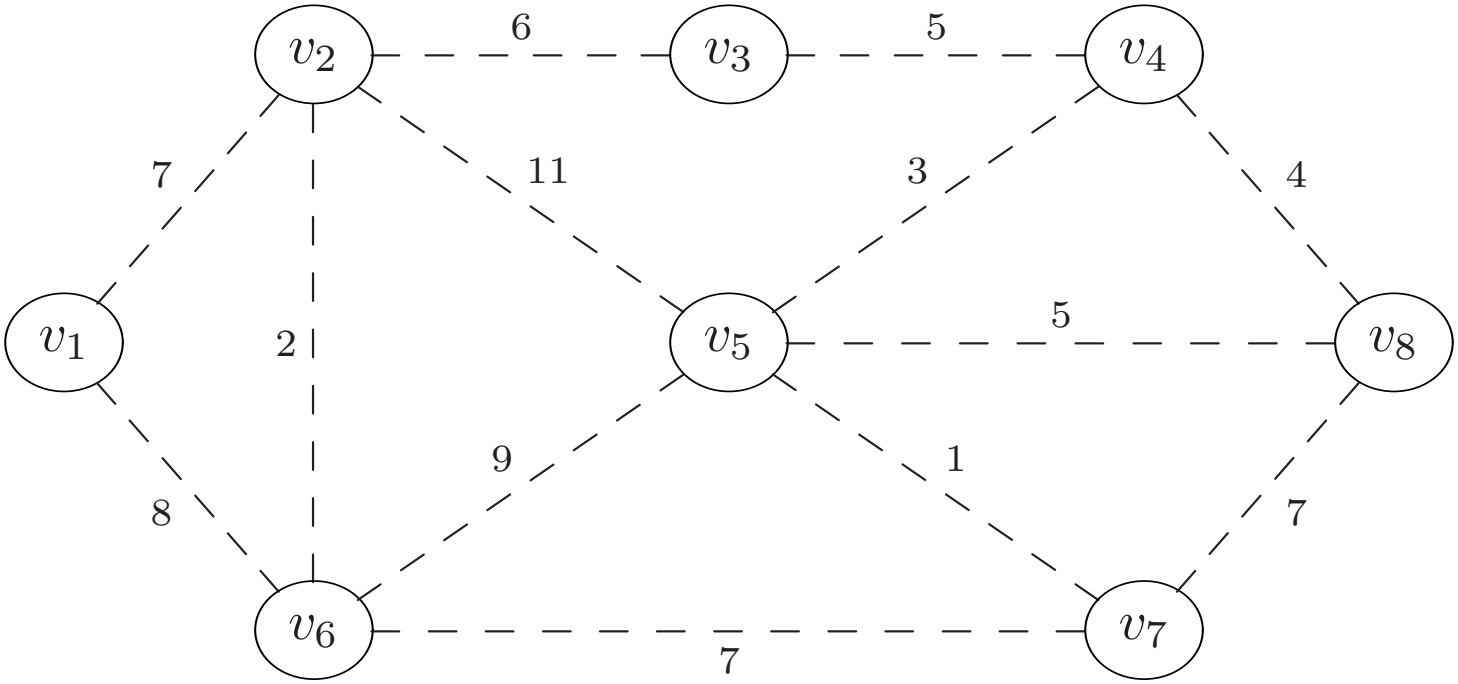
Running Time of PrimMST: Naive Implementation

```
   procedure PrimMST(G)
1  U ← G.V − {v_1} ;        /* G.V = set of vertices of graph G */
2  v_1.MSTparent ← NULL;
3  while (U ≠ ∅) and (∃ edge from (G.V − U) to U) do
4  |   (v_i, v_j) ← minimum weight edge from G.V − U to U;
5  |   v_j.MSTparent ← v_i;
6  |   U ← U − {v_j};
7  end
```

n = number of vertices of G.
m = number of edges of G.

What is the running time of PrimMST in terms of n and m?

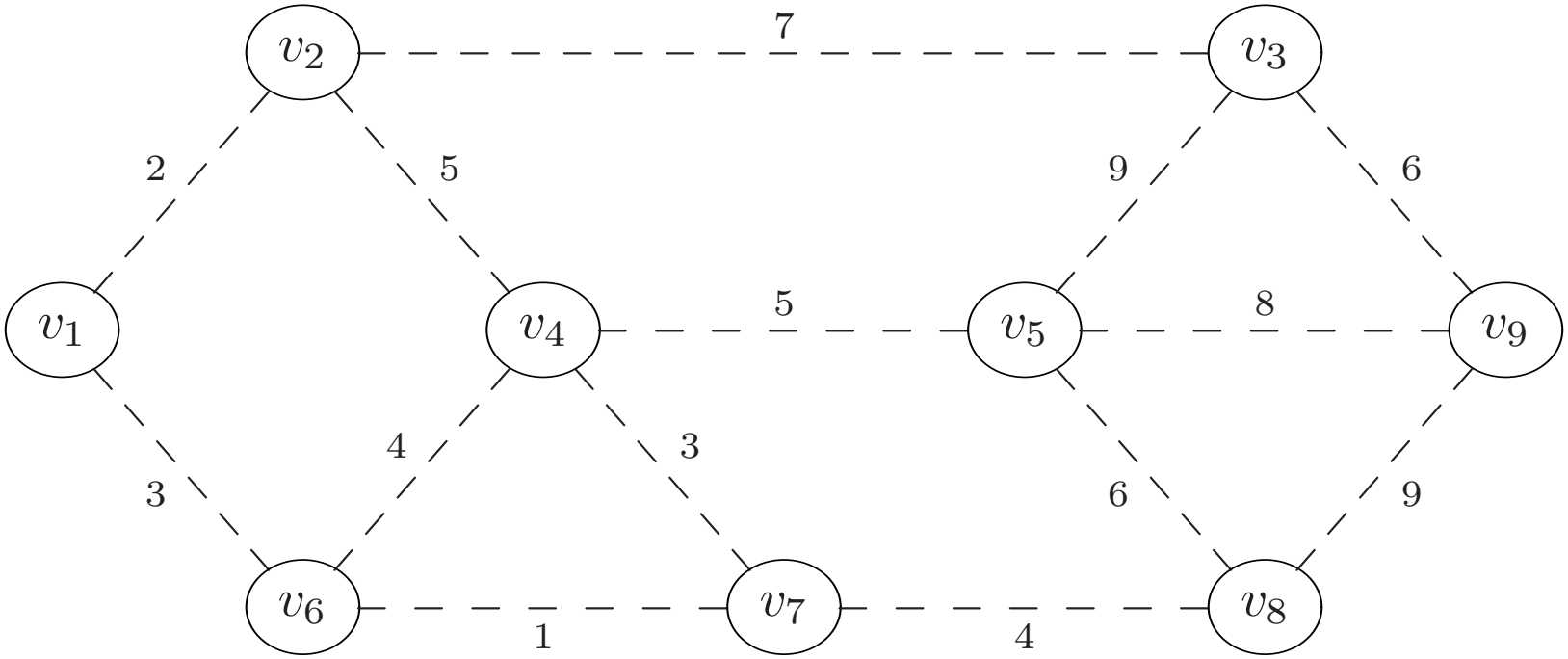
Minimum Spanning Tree: Storing Costs at Vertices



Prim's MST: Storing Costs at Vertices

```
   procedure PrimMST(G)
 1 U ← G.V ;                          /* G.V = set of vertices of graph G */
 2 foreach v_i ∈ G.V − {v_1} do  v_i.cost ← ∞;
 3 v_1.cost ← 0;
 4 v_1.parent ← NULL;
 5 while (U ≠ ∅) and (v_i.cost < ∞ for some v_i ∈ U) do
 6 |   v_j ← v_i ∈ U with minimum v_i.cost;
 7 |   U ← U − {v_j} ;                  /* Remove v_j from U */
 8 |   v_j.MSTparent ← v_j.parent;
 9 |   foreach edge (v_j, v_k) incident on v_j do
10 |   |   if (v_k is in U and weight(v_j, v_k) < v_k.cost) then
11 |   |   |   v_k.parent ← v_j ;       /* Store edge (v_j, v_k) */
12 |   |   |   v_k.cost ← weight(v_j, v_k);
13 |   |   end
14 |   end
15 end
```

Minimum Spanning Tree: Storing Costs at Vertices



Running Time Analysis

```
   procedure PrimMST(G)
 1 U ← G.V ;                          /* G.V = set of vertices of graph G */
 2 foreach v_i ∈ G.V − {v_1} do  v_i.cost ← ∞;
 3 v_1.cost ← 0;
 4 v_1.parent ← NULL;
 5 while (U ≠ ∅) and (v_i.cost < ∞ for some v_i ∈ U) do
 6 |   v_j ← v_i ∈ U with minimum v_i.cost;
 7 |   U ← U − {v_j} ;                        /* Remove v_j from U */
 8 |   v_j.MSTparent ← v_j.parent;
 9 |   foreach edge (v_j, v_k) incident on v_j do
10 |   |   if (v_k is in U and weight(v_j, v_k) < v_k.cost) then
11 |   |   |   v_k.parent ← v_j ;              /* Store edge (v_j, v_k) */
12 |   |   |   v_k.cost ← weight(v_j, v_k);
13 |   |   end
14 |   end
15 end
```

Priority Queue

Operations	Running Time (Heap Implementation)
Q.Insert(Object x, Key k);	
$x \leftarrow$ Q.DeleteMin();	
Q.IsEmpty();	
Q.DecreaseKey(Object x, Key k);	
Q.IsNotEmpty();	
Q.Contains(Object x);	
$k \leftarrow$ Q.Key(Object x);	
$k \leftarrow$ Q.MinKey();	

s = number of objects in the queue (queue size).

Prim's MST: Storing Costs at Vertices

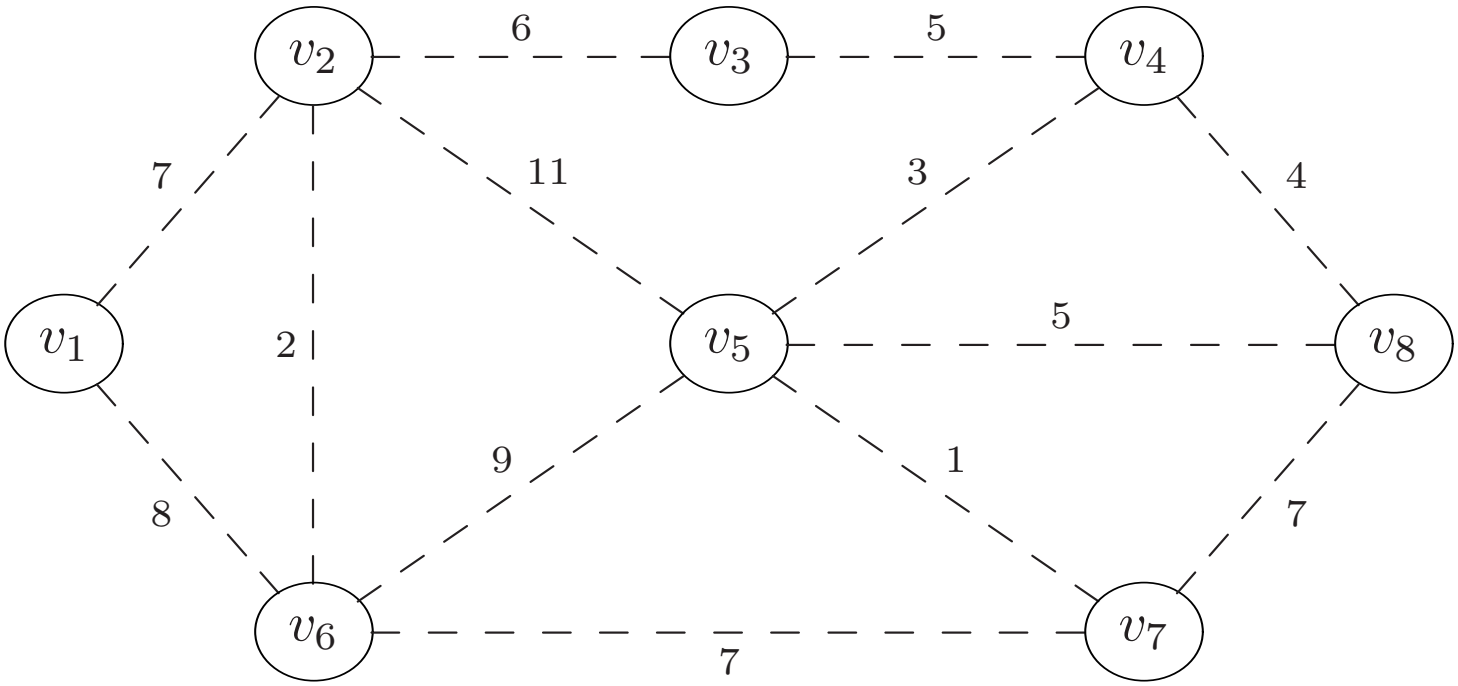
```
   procedure PrimMST(G)
 1 U ← G.V ;                          /* G.V = set of vertices of graph G */
 2 foreach v_i ∈ G.V − {v_1} do  v_i.cost ← ∞;
 3 v_1.cost ← 0;
 4 v_1.parent ← NULL;
 5 while (U ≠ ∅) and (v_i.cost < ∞ for some v_i ∈ U) do
 6 |   v_j ← v_i ∈ U with minimum v_i.cost;
 7 |   U ← U − {v_j} ;                  /* Remove v_j from U */
 8 |   v_j.MSTparent ← v_j.parent;
 9 |   foreach edge (v_j, v_k) incident on v_j do
10 |   |   if (v_k is in U and weight(v_j, v_k) < v_k.cost) then
11 |   |   |   v_k.parent ← v_j ;      /* Store edge (v_j, v_k) */
12 |   |   |   v_k.cost ← weight(v_j, v_k);
13 |   |   end
14 |   end
15 end
```

Prim's MST: Priority Queue of Vertices

```
   procedure PrimMST(G)
 1 foreach v_i ∈ G.V − {v_1} do  Q.Insert(v_i, ∞);
 2 Q.Insert(v_1, 0) ;            /* Q is a priority queue of vertices */
 3 v_1.parent ← NULL;
 4 while Q.IsNotEmpty() and (Q.MinKey() ≠ ∞) do
 5 |   v_j ← Q.DeleteMin();
 6 |   v_j.MSTparent ← v_j.parent;
 7 |   foreach edge (v_j, v_k) incident on v_j do
 8 |   |   if (Q.Contains(v_k) and Q.Key(v_k) > weight(v_j, v_k))
   |   |   then
 9 |   |   |   v_k.parent ← v_j ;          /* Store edge (v_j, v_k) */
10 |   |   |   Q.DecreaseKey(v_k, weight(v_j, v_k));
11 |   |   end
12 |   end
13 end
```

Shortest Paths

Single Source Shortest Path: Example



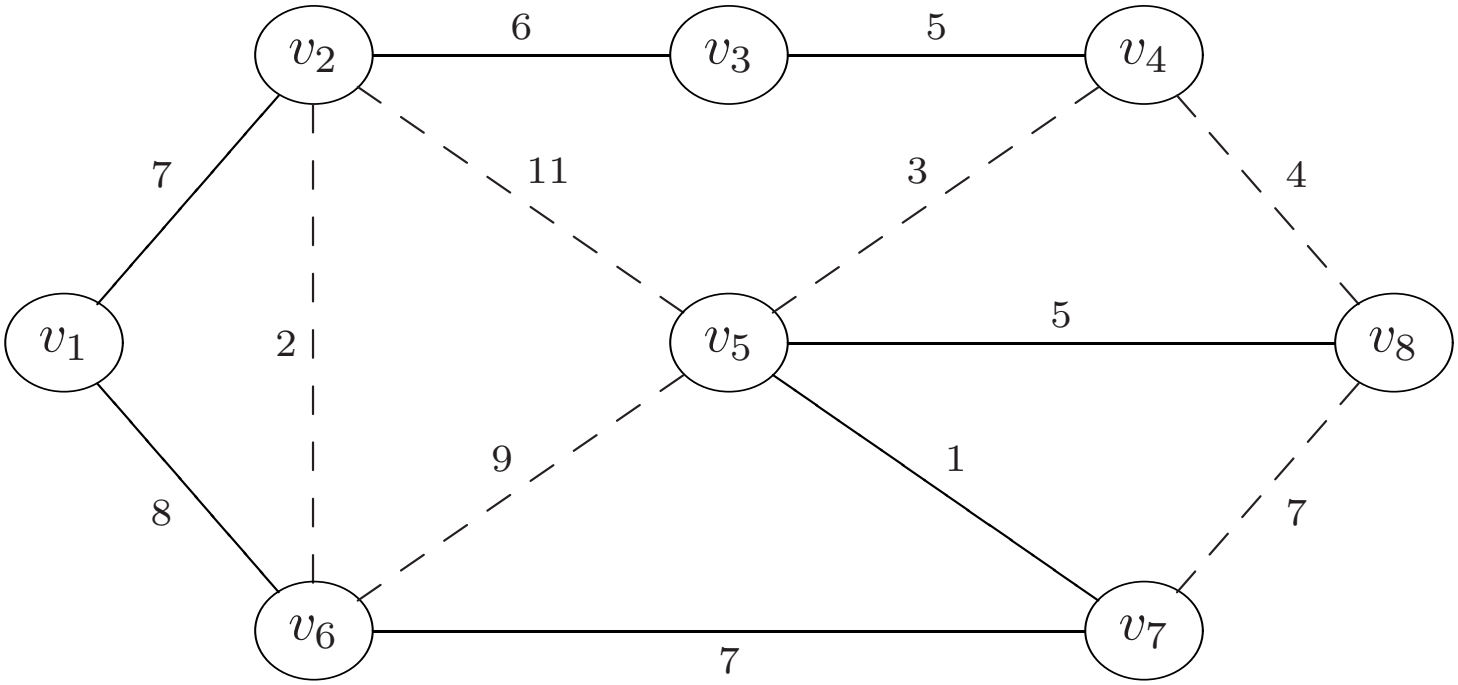
Shortest Path Theorem

Shortest Path Theorem: If $(v_1, v_2, \ldots, v_{i-1}, v_i)$ is a shortest path from v_1 to v_i, then $(v_1, v_2, \ldots, v_{i-1})$ is a shortest path from v_1 to v_{i-1}.

Proof. If there is a shorter path P from v_1 to v_{i-1}, then $P \cup (v_{i-1}, v_i)$ would be shorter than $(v_1, v_2, \ldots, v_{i-1}, v_i)$.

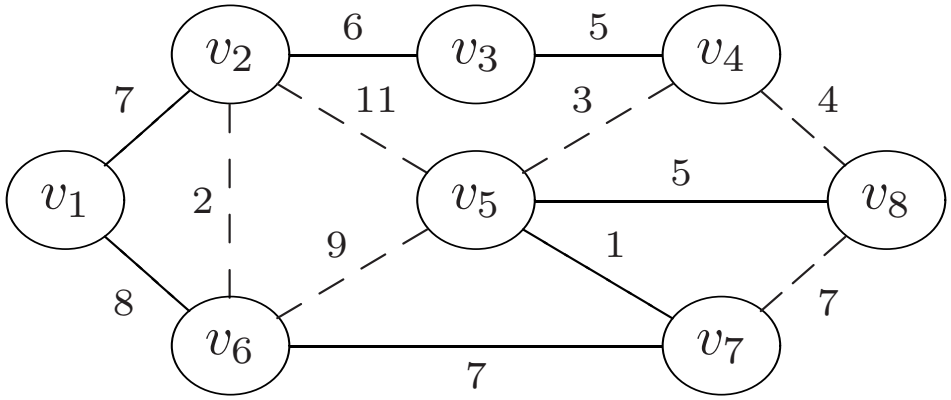
□

Single Source Shortest Path Tree from v_1

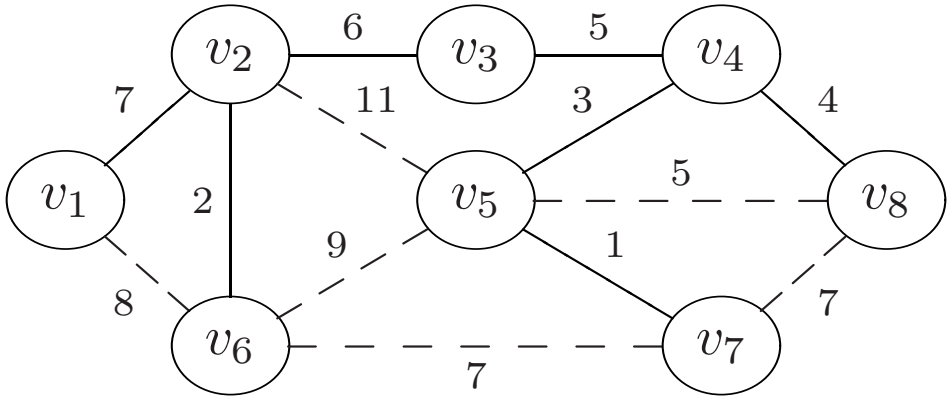


Shortest Path Tree Versus Min Spanning Tree

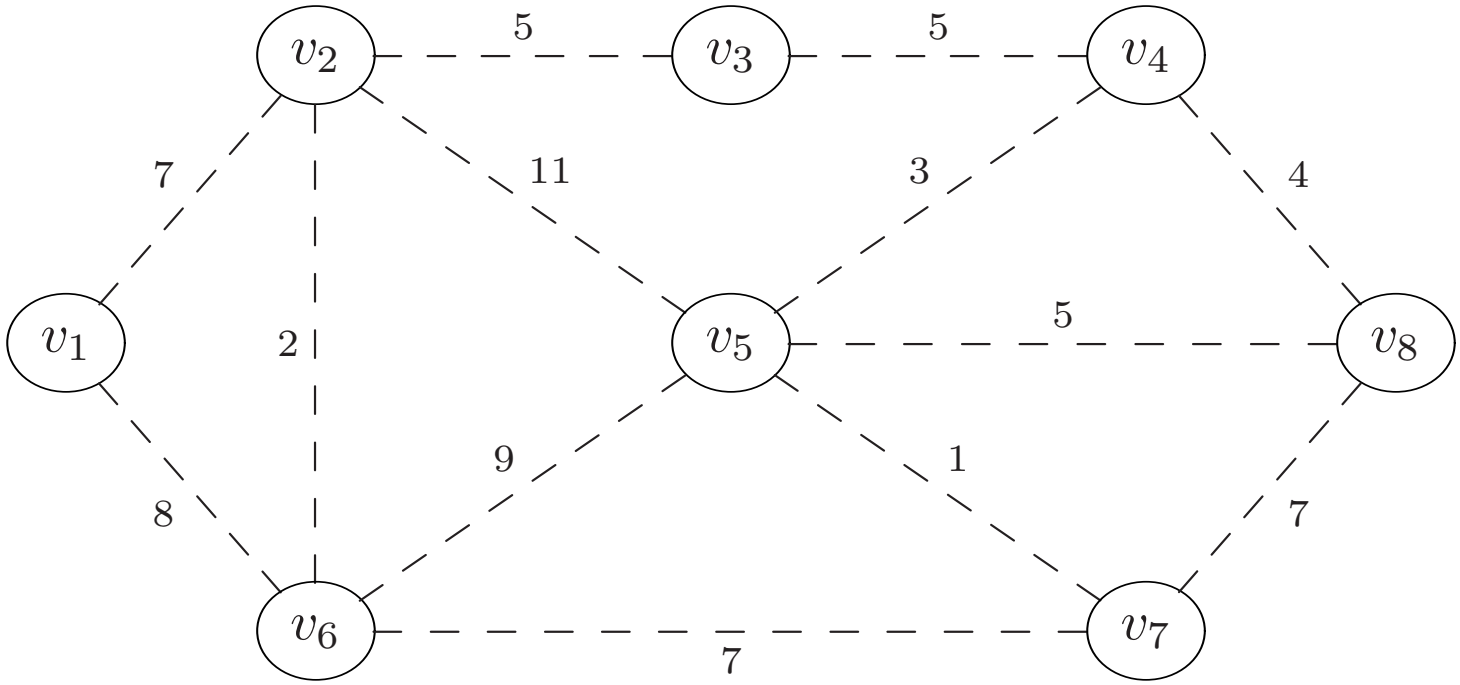
Shortest Path Tree from v_1:



Minimum Spanning Tree:



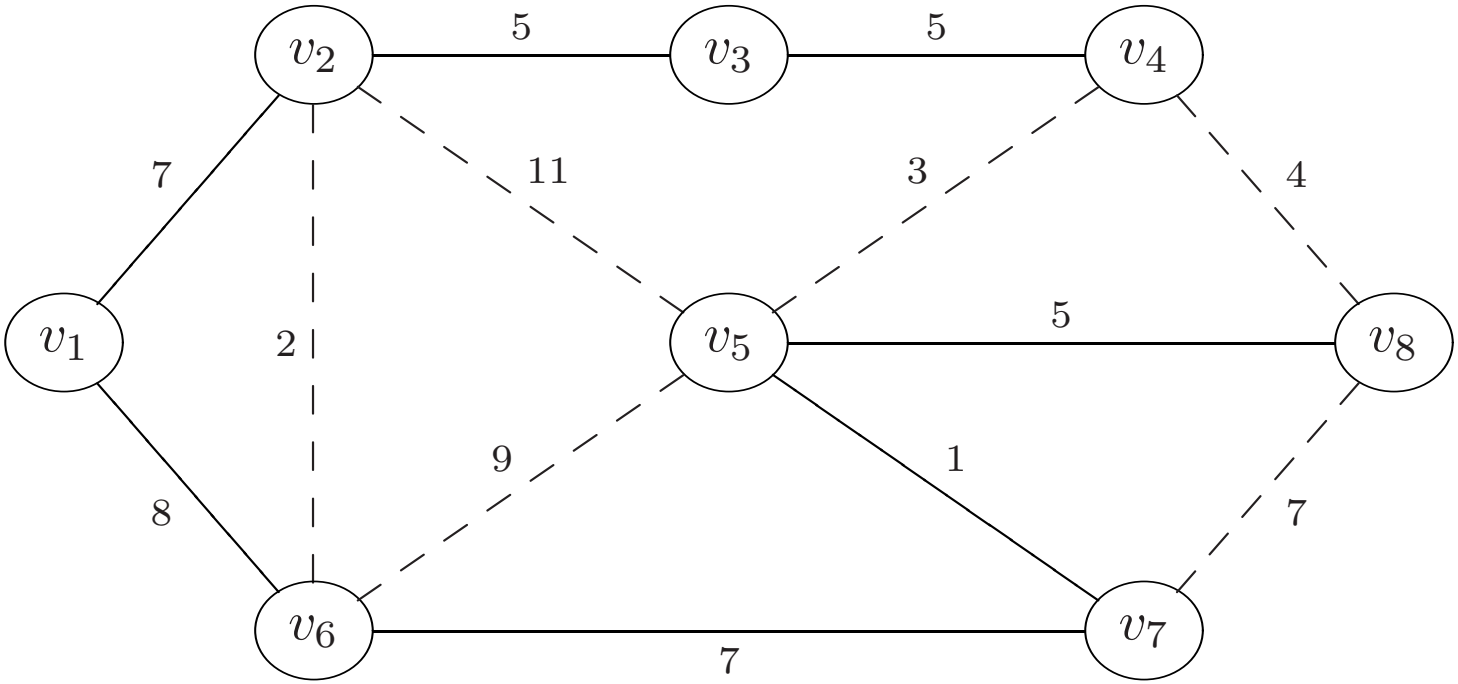
Single Source Shortest Path: Example 2



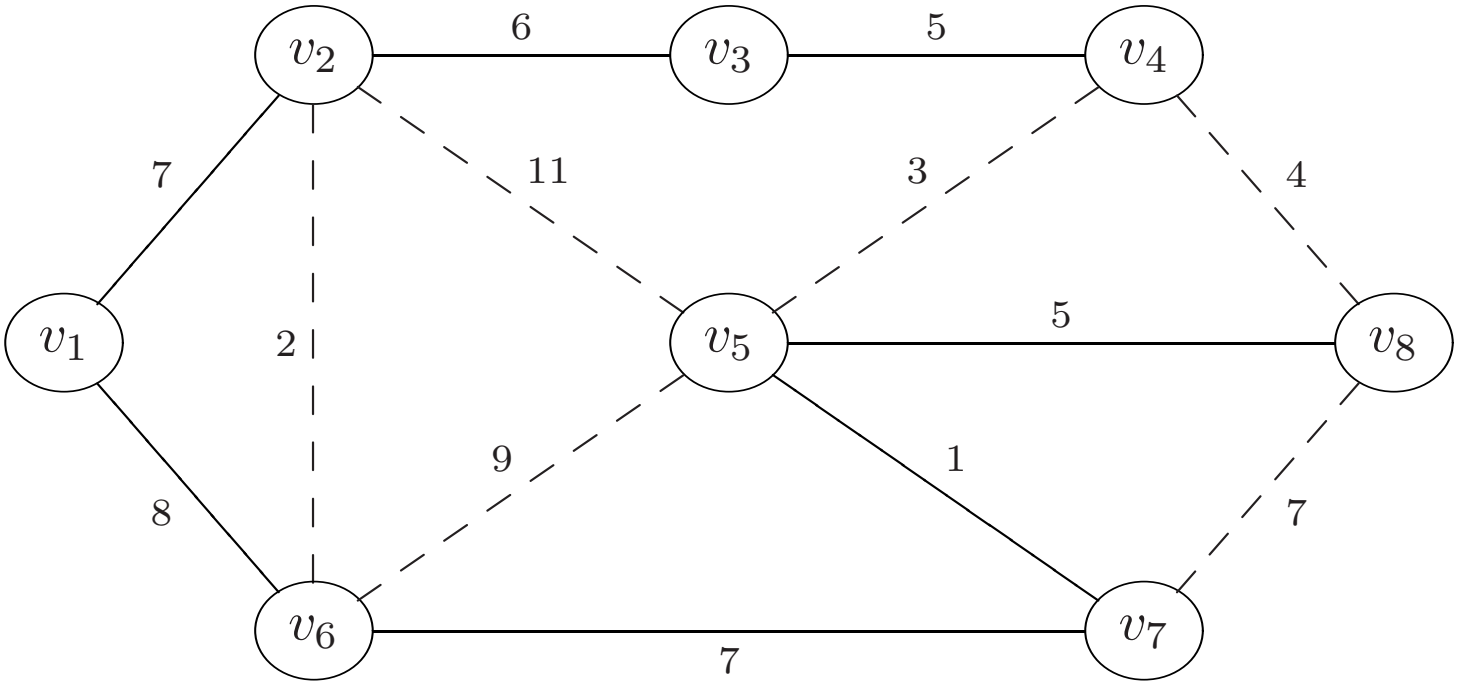
Note: Weight of edge (v_2, v_3) is 5 (previously was 6).

Two Shortest Path Trees

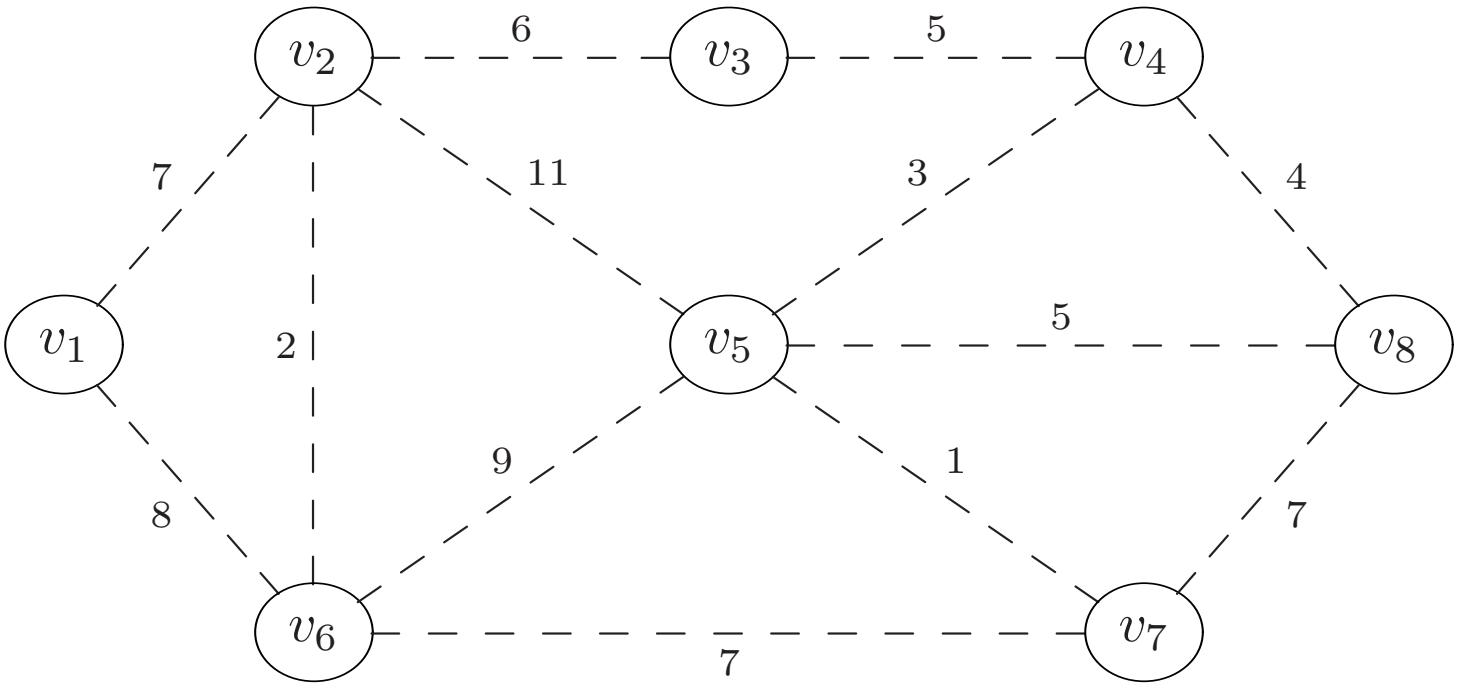
What is the other shortest path tree of this graph?



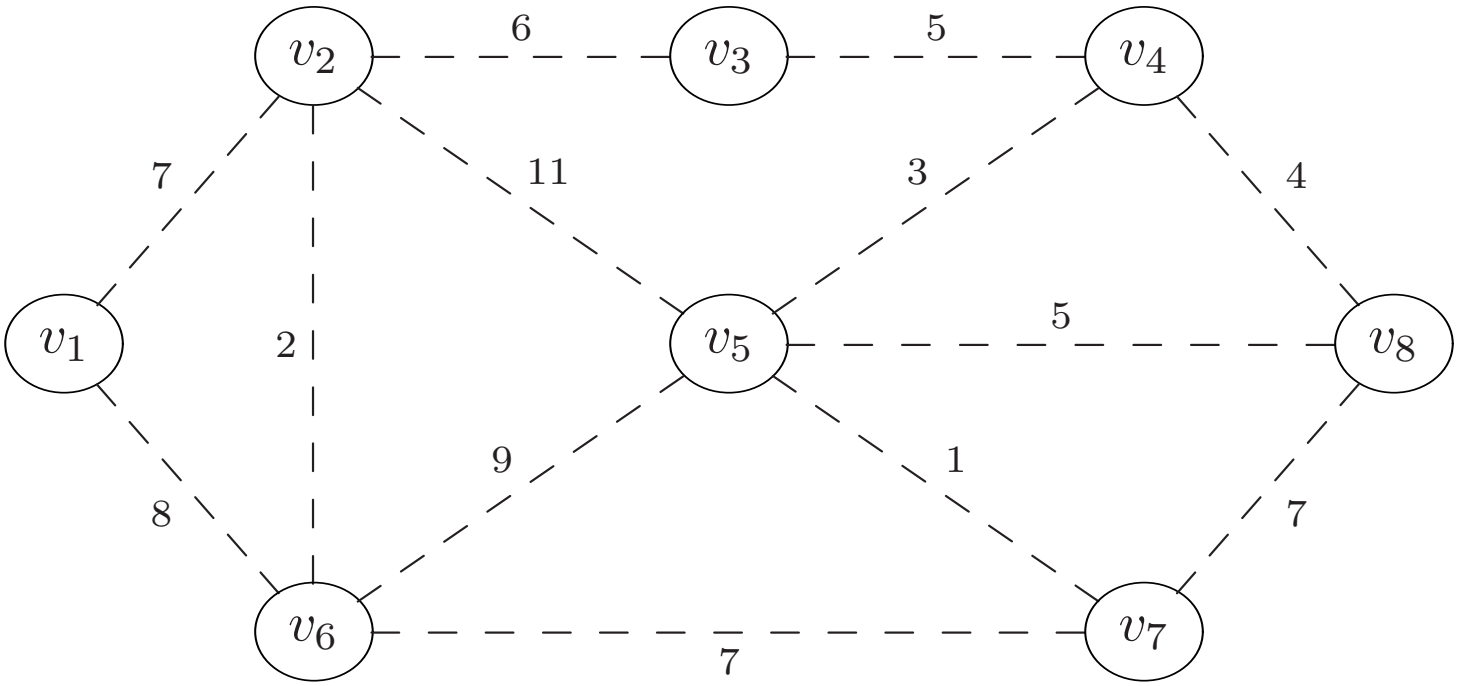
Shortest Path Tree from v_1



Shortest Path from v_6



Single Source Shortest Path: Example



Dijkstra's Shortest Path Algorithm

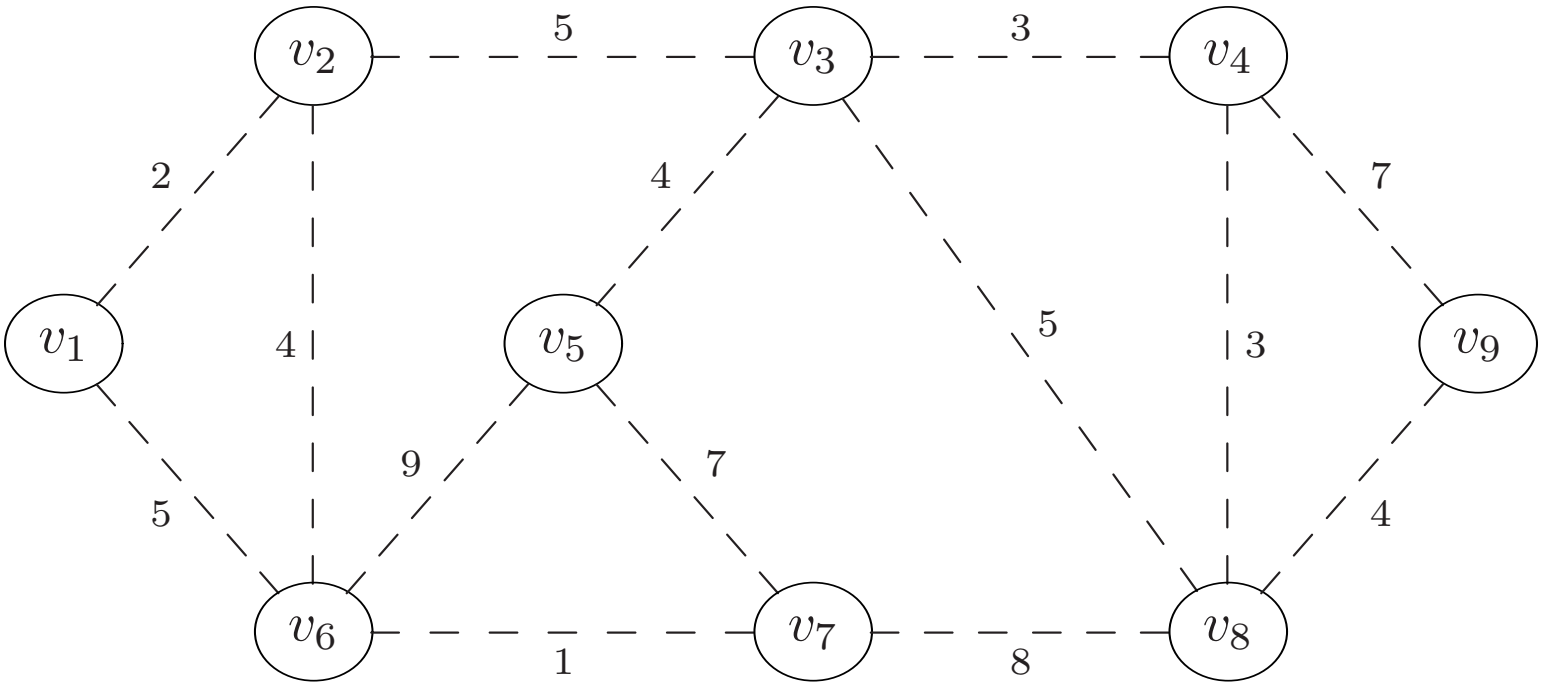
Input : Weighted graph G.
Start vertex v_s.

Output : Shortest path tree from v_s to all vertices of G.

procedure `DijkstraShortestPath(`G, v_s`)`

```
1 U ← G.V − {v_s} ;        /* G.V = set of vertices of graph G */
2 v_s.SPparent ← NULL;
3 v_s.distance ← 0;
4 while (U ≠ ∅) and (∃ edge from (G.V − U) to U) do
5 |   (v_i, v_j) ← edge from G.V − U to U which minimizes
  |                 v_i.distance + weight(v_i, v_j);
6 |   v_j.SPparent ← v_i;
7 |   v_j.distance ← v_i.distance + weight(v_i, v_j);
8 |   U ← U − {v_j};
9 end
```

Single Source Shortest Path: Example 2



Shortest Path Theorem

$d(v_s, v_i)$ = length of shortest path from v_s to v_i.

Shortest Path Theorem 2:

Let U be a subset of $G.V$ where $v_s \notin U$.

If $(v_i, v_j) \in G.E$ minimizes $d(v_s, v_i) + \mathsf{weight}(v_i, v_j)$ among all edges connecting $v_i \in G.V - U$ to $v_j \in U$, then

$$d(v_s, v_j) = d(v_s, v_i) + \mathsf{weight}(v_i, v_j).$$

Shortest Path Theorem 2: Proof Part 1

$d(v_s, v_i)$ = length of shortest path from v_s to v_i.

Shortest Path Theorem 2:
Let U be a subset of $G.V$ where $v_s \notin U$.
If $(v_i, v_j) \in G.E$ minimizes $d(v_s, v_i) + \mathsf{weight}(v_i, v_j)$ among all edges connecting $v_i \in G.V - U$ to $v_j \in U$, then $d(v_s, v_j) = d(v_s, v_i) + \mathsf{weight}(v_i, v_j)$.

Proof Part 1: $d(v_s, v_j) \leq d(v_s, v_i) + \mathsf{weight}(v_i, v_j)$*:*

Let P be a shortest path from v_s to v_i.

Length$(P) = d(v_s, v_i)$.

Let $P' = P \cup (v_i, v_j)$.

Length$(P') = d(v_s, v_i) + \mathsf{weight}(v_i, v_j)$.

Therefore, $d(v_s, v_j) \leq$ Length$(P') = d(v_s, v_i) + \mathsf{weight}(v_i, v_j)$.

□

Shortest Path Theorem 2: Proof Part 2

$d(v_s, v_i)$ = length of shortest path from v_s to v_i.

Shortest Path Theorem 2:
Let U be a subset of $G.V$ where $v_s \notin U$.
If $(v_i, v_j) \in G.E$ minimizes $d(v_s, v_i) + \mathsf{weight}(v_i, v_j)$ among all edges connecting $v_i \in G.V - U$ to $v_j \in U$, then $d(v_s, v_j) = d(v_s, v_i) + \mathsf{weight}(v_i, v_j)$.

Proof Part 2: $d(v_s, v_j) \geq d(v_s, v_i) + \mathsf{weight}(v_i, v_j)$*:*

Let P be a shortest path from v_s to v_j.

Since $v_s \in G.V - U$ and $v_j \in U$, path P contains some edge (v_a, v_b) where $v_a \in G.V - U$ and $v_b \in U$.

Let P_a be the subpath of P from v_s to v_a.

$\text{Length}(P) \geq \text{Length}(P_a) + \mathsf{weight}(v_a, v_b) \geq d(v_s, v_a) + \mathsf{weight}(v_a, v_b)$.

Since (v_i, v_j) minimizes $d(v_s, v_i) + \mathsf{weight}(v_i, v_j)$,
$d(v_s, v_a) + \mathsf{weight}(v_a, v_b) \geq d(v_s, v_i) + \mathsf{weight}(v_i, v_j)$.

Therefore, $\text{Length}(P) \geq d(v_s, v_a) + \mathsf{weight}(v_a, v_b) \geq d(v_s, v_i) + \mathsf{weight}(v_i, v_j)$.

Since P is a shortest path,
$d(v_s, v_j) = \text{Length}(P) \geq d(v_s, v_i) + \mathsf{weight}(v_i, v_j)$. □

Dijkstra's Shortest Path Algorithm

```
   procedure DijkstraShortestPath(G, v_s)
1  U ← G.V − {v_s} ;        /* G.V = set of vertices of graph G */
2  v_s.SPparent ← NULL;
3  v_s.distance ← 0;
4  while (U ≠ ∅) and (∃ edge from (G.V − U) to U) do
5  |   (v_i, v_j) ← edge from G.V − U to U which minimizes
   |                 v_i.distance + weight(v_i, v_j);
6  |   v_j.SPparent ← v_i;
7  |   v_j.distance ← v_i.distance + weight(v_i, v_j);
8  |   U ← U − {v_j};
9  end
```

$d(v_s, v_i)$ = shortest distance from v_s to v_i.

Claim: At each iteration, v_k.distance $= d(v_s, v_k)$ for every $v_k \in G.V - U$.

Proof of Correctness

$d(v_s, v_i)$ = shortest distance from v_s to v_i.

Claim: At each iteration, $v_k.\mathsf{distance} = d(v_s, v_k)$ for every $v_k \in G.V - U$.

Proof. Let U_p equal U at the beginning of the p'th iteration.

$G.V - U_1 = \{v_s\}$ and $\mathsf{distance}(v_s, v_s) = 0 = d(v_s, v_s)$.

Assume $v_k.\mathsf{distance} = d(v_s, v_k)$ for every $v_k \in G.V - U_p$.

$G.V - U_{p+1}$ has one more vertex v_j than $G.V - U_p$.

Edge (v_i, v_j) minimizes $d(v_s, v_i) + \mathsf{weight}(v_i, v_j)$ among all edges connecting $v_i \in G.V - U_p$ to $v_j \in U_p$.

By Shortest Path Theorem 2, $d(v_s, v_j) = d(v_s, v_i) + \mathsf{weight}(v_i, v_j)$.

Since $v_i \in G.V - U_p$, distance $v_i.\mathsf{distance} = d(v_s, v_i)$. Thus,

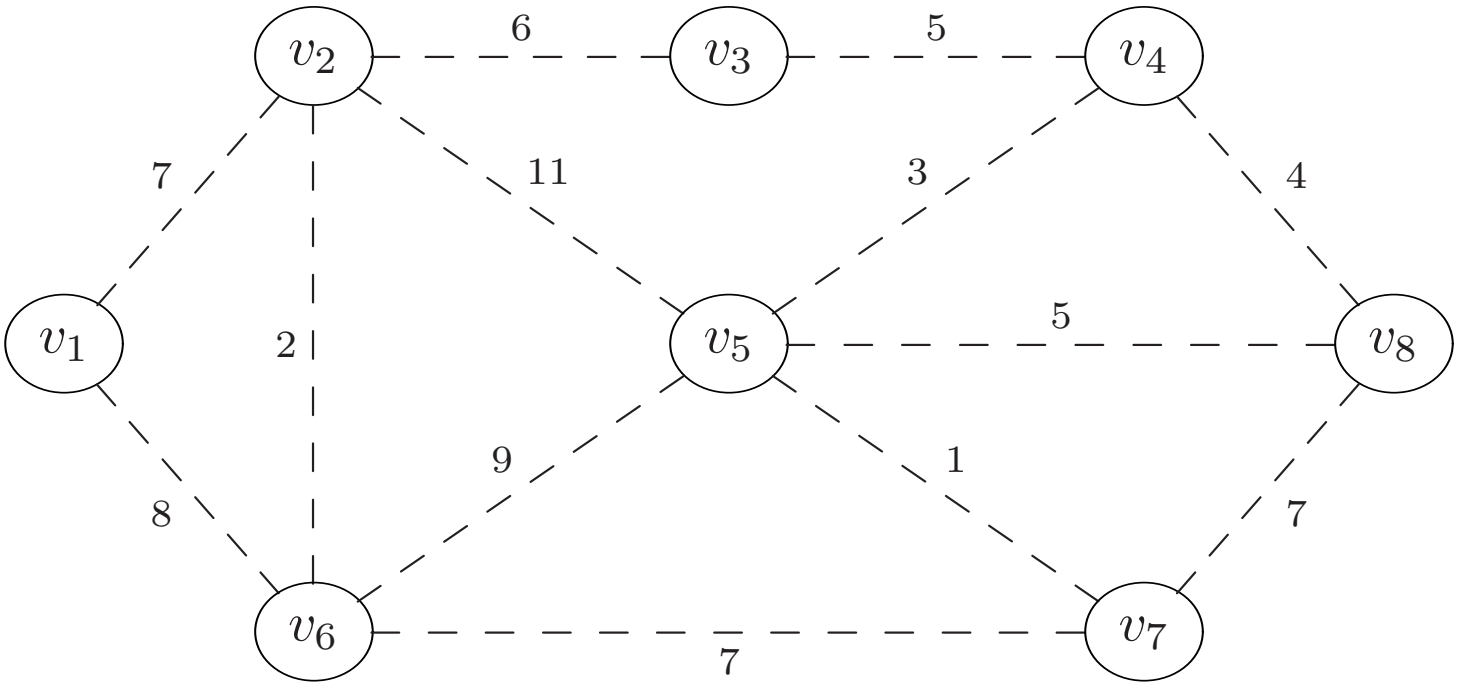
$$\begin{aligned} v_j.\mathsf{distance} &= v_i.\mathsf{distance} + \mathsf{weight}(v_i, v_j) \\ &= d(v_s, v_i) + \mathsf{weight}(v_i, v_j) = d(v_s, v_j). \end{aligned}$$

□

Running Time of Dijkstra's Algorithm

```
   procedure DijkstraShortestPath(G, v_s)
1  U ← G.V − {v_s} ;        /* G.V = set of vertices of graph G */
2  v_s.SPparent ← NULL;
3  v_s.distance ← 0;
4  while (U ≠ ∅) and (∃ edge from (G.V − U) to U) do
5  |   (v_i, v_j) ← edge from G.V − U to U which minimizes
   |                 v_i.distance + weight(v_i, v_j);
6  |   v_j.SPparent ← v_i;
7  |   v_j.distance ← v_i.distance + weight(v_i, v_j);
8  |   U ← U − {v_j};
9  end
```

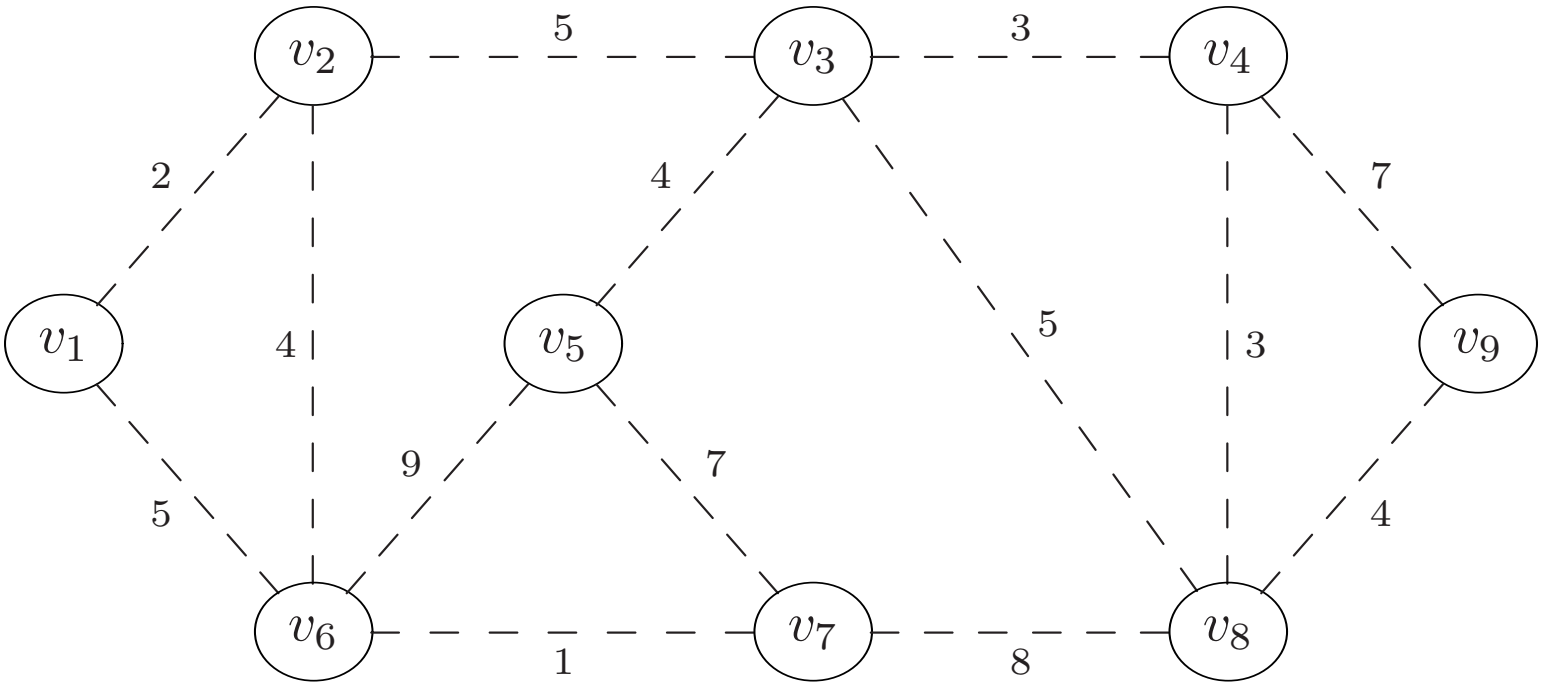
Shortest Path Tree: Storing Cost at Vertices



Dijkstra's Algorithm: Storing Costs at Vertices

```
   procedure DijkstraShortestPath(G, v_s)
 1 U ← G.V ;                          /* G.V = set of vertices of graph G */
 2 foreach v_i ∈ G.V − {v_s} do  v_i.distance ← ∞;
 3 v_s.distance ← 0;
 4 v_s.parent ← NULL;
 5 while (U ≠ ∅) and (v_i.distance < ∞ for some v_i ∈ U) do
 6 |   v_j ← v_i ∈ U with minimum v_i.distance;
 7 |   U ← U − {v_j} ;                          /* Remove v_j from U */
 8 |   v_j.SPparent ← v_j.parent;
 9 |   foreach edge (v_j, v_k) incident on v_j do
10 |   |   newDist ← v_j.distance + weight(v_j, v_k);
11 |   |   if (v_k ∈ U and newDist < v_k.distance) then
12 |   |   |   v_k.parent ← v_j ;                  /* Store edge (v_j, v_k) */
13 |   |   |   v_k.distance ← newDist;
14 |   |   end
15 |   end
16 end
```

Single Source Shortest Path: Example 2



Running Time Analysis

```
   procedure DijkstraShortestPath(G, v_s)
 1 U ← G.V ;                          /* G.V = set of vertices of graph G */
 2 foreach v_i ∈ G.V − {v_s} do  v_i.distance ← ∞;
 3 v_s.distance ← 0;
 4 v_s.parent ← NULL;
 5 while (U ≠ ∅) and (v_i.distance < ∞ for some v_i ∈ U) do
 6 |   v_j ← v_i ∈ U with minimum v_i.distance;
 7 |   U ← U − {v_j} ;                          /* Remove v_j from U */
 8 |   v_j.SPparent ← v_j.parent;
 9 |   foreach edge (v_j, v_k) incident on v_j do
10 |   |   newDist ← v_j.distance + weight(v_j, v_k);
11 |   |   if (v_k is in U and newDist < v_k.distance) then
12 |   |   |   v_k.parent ← v_j ;                  /* Store edge (v_j, v_k) */
13 |   |   |   v_k.distance ← newDist;
14 |   |   end
15 |   end
16 end
```

Dijkstra's Algorithm: Priority Queue of Vertices

```
   procedure DijkstraShortestPath(G, v_s)
 1 foreach v_i ∈ G.V − {v_s} do  Q.Insert(v_i, ∞);
 2 Q.Insert(v_s, 0) ;                    /* Q is a priority queue of vertices */
 3 v_s.parent ← NULL;
 4 while Q.IsNotEmpty() and (Q.MinKey() ≠ ∞) do
 5 |   minDist ← Q.MinKey();
 6 |   v_j ← Q.DeleteMin();
 7 |   v_j.SPparent ← v_j.parent;
 8 |   v_j.distance ← minDist;
 9 |   foreach edge (v_j, v_k) incident on v_j do
10 |   |   newDist ← v_j.distance + weight(v_j, v_k);
11 |   |   if (Q.Contains(v_k) and newDist < Q.Key(v_k)) then
12 |   |   |   v_k.parent ← v_j ;                    /* Store edge (v_j, v_k) */
13 |   |   |   Q.DecreaseKey(v_k, newDist);
14 |   |   end
15 |   end
16 end
```