Algorithm and Data Structure Analysis (ADSA)

Minimum Spanning Trees

Properties of MSTs

An MST of a given graph G can be constructed by greedy algorithms.

Crucial properties:

- Cut property (Let e be an edge of minimum cost in a cut C. Then there is an MST that contains e)
- Cycle property (an edge of maximal cost in any cycle does not need to be considered for computing an MST)

Jarnik-Prim Algorithm

- Similar to Dijkstra's algorithm for the single-source shortest path problem.
- Start with an arbitrary node s of V.
- Let S be the set of already connected nodes.
- In the beginning S={s} holds.
- Insert in each iteration an edge of minimal cost that connects a node u of S to a node v not contained in S (it's an edge of minimal cost in this cut).
- Add v to S and continue until all nodes are contained in S.

Jarnik-Prim Algorithm Implementation

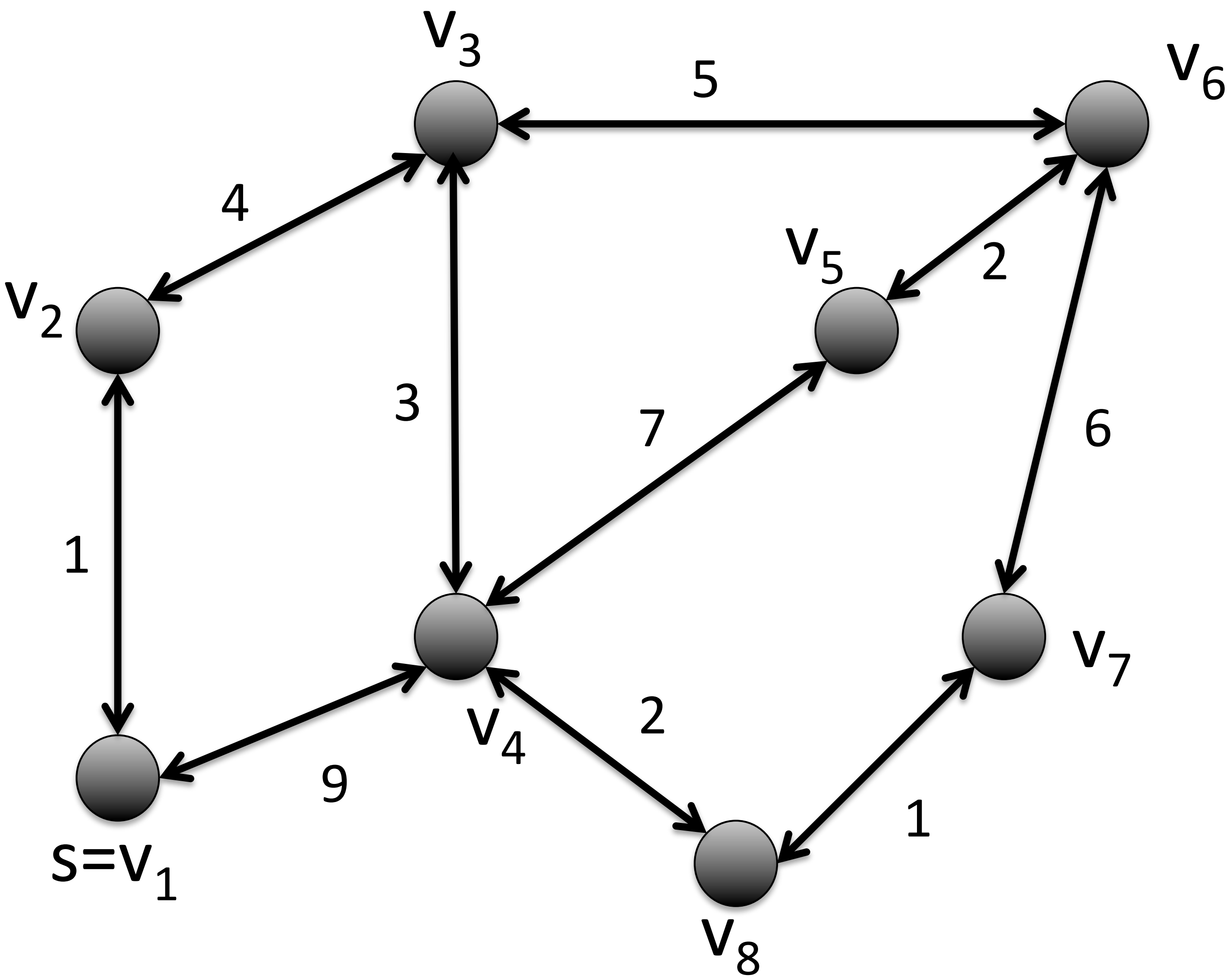
Function *jpMST* : *Set* **of** *Edge*
  $d = \langle\infty,\ldots,\infty\rangle$: $NodeArray[1..n]$ **of** $\mathbb{R}\cup\{\infty\}$ // $d[v]$ is the distance of v from the tree
  parent : *NodeArray* **of** *NodeId* // parent$[v]$ is shortest edge between S and v
  Q : *NodePQ* // uses $d[\cdot]$ as priority
  $Q.insert(s)$ for some arbitrary $s \in V$
  while $Q \neq \emptyset$ **do**
    $u := Q.deleteMin$
    $d[u] := 0$ // $d[u] = 0$ encodes $u \in S$
    foreach *edge* $e = (u,v) \in E$ **do**
      if $c(e) < d[v]$ **then** // $c(e) < d[v]$ implies $d[v] > 0$ and hence $v \notin S$
        $d[v] := c(e)$
        $parent[v] := u$
        if $v \in Q$ **then** $Q.decreaseKey(v)$ **else** $Q.insert(v)$
    invariant $\forall v \in Q : d[v] = \min\{c((u,v)) : (u,v) \in E \wedge u \in S\}$
  return $\{(v, parent[v]) : v \in V \setminus \{s\}\}$

Fig 11.3 Mehlhorn/Sanders

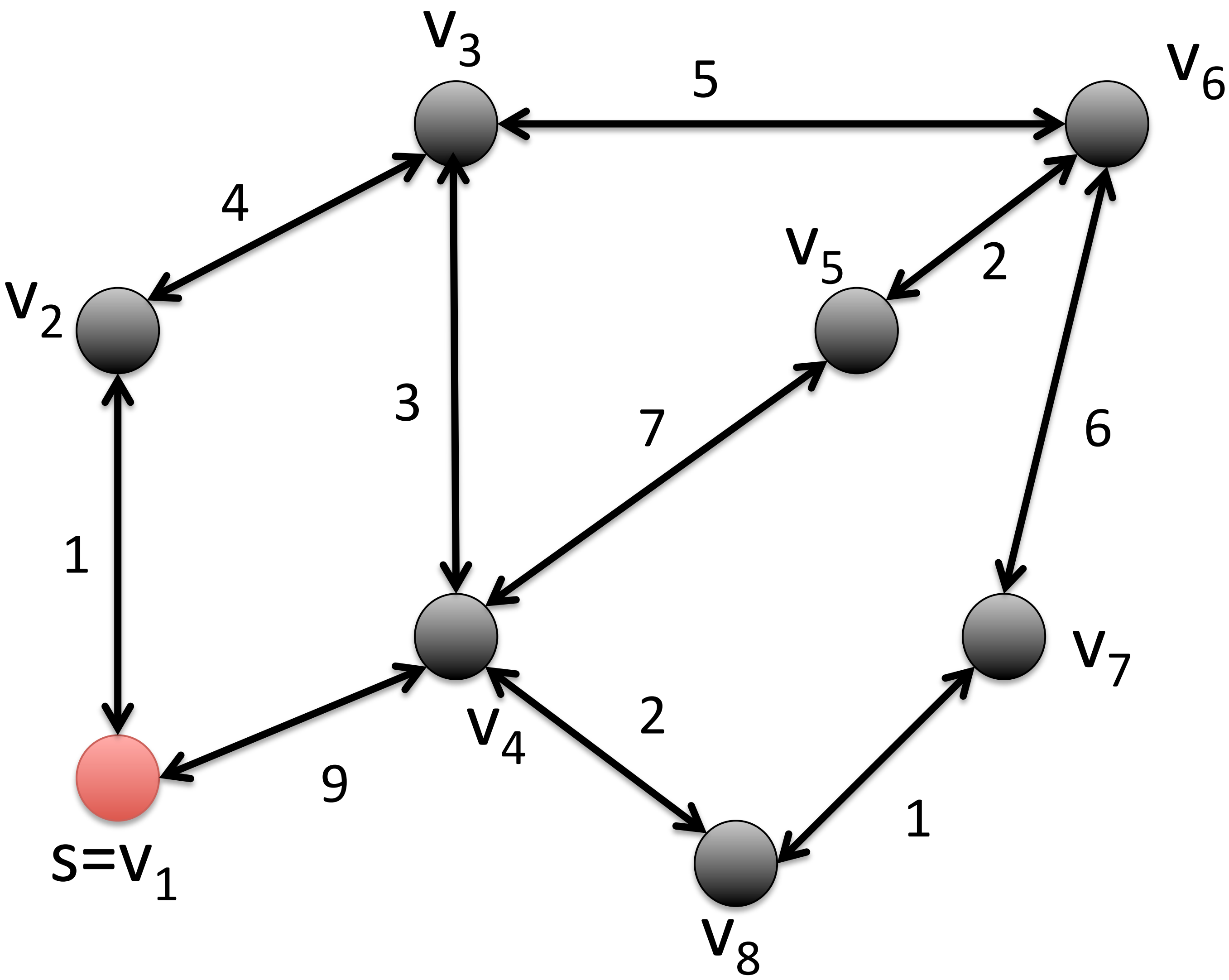
Runtime

- We can carry over the analysis for Dijkstra's algorithm.
- Crucial again is the implementation of the priority queue.
- Overall runtime is $O(m + n \log n)$ when using Fibonacci heaps for the implementation of the priority queue.

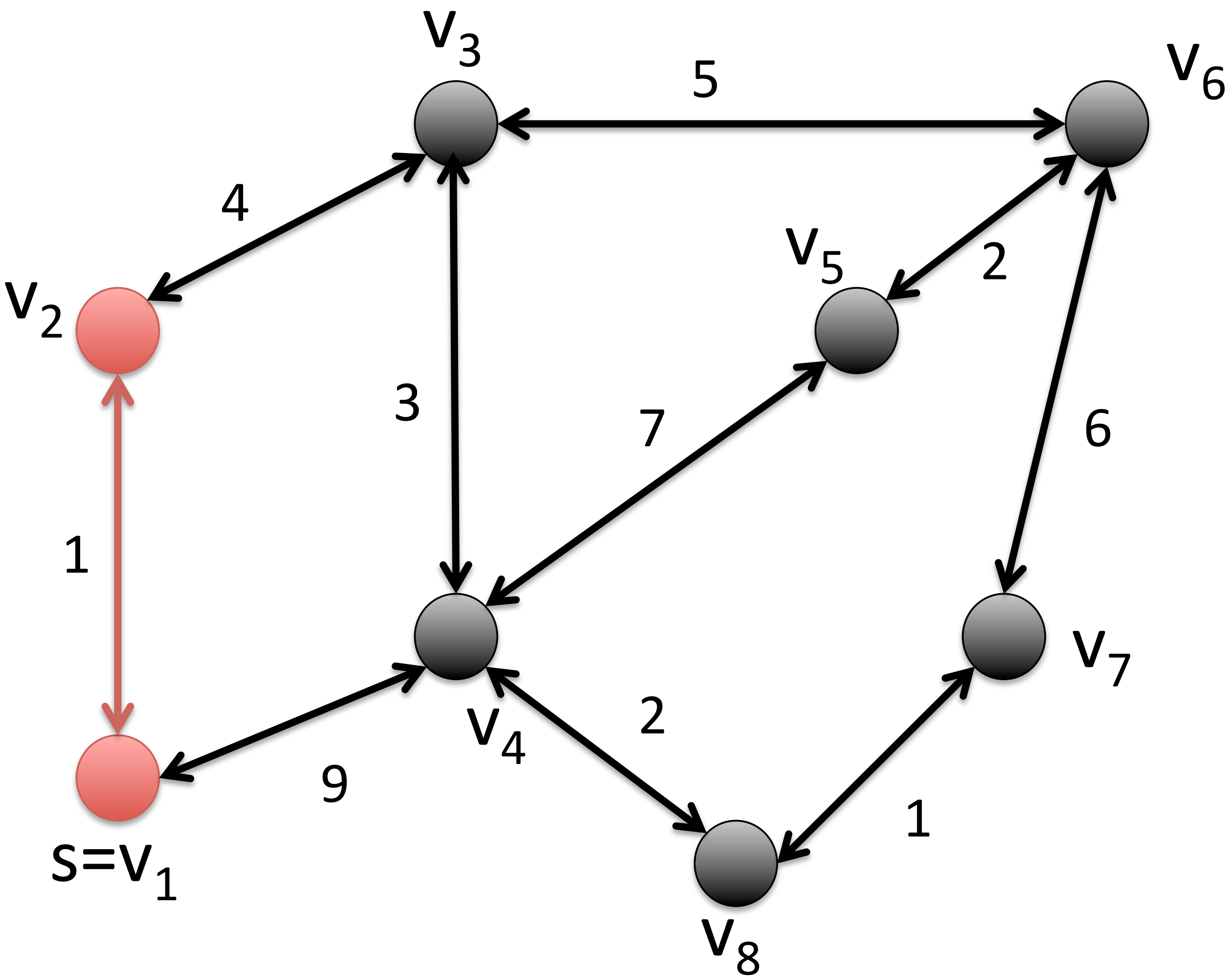
Example



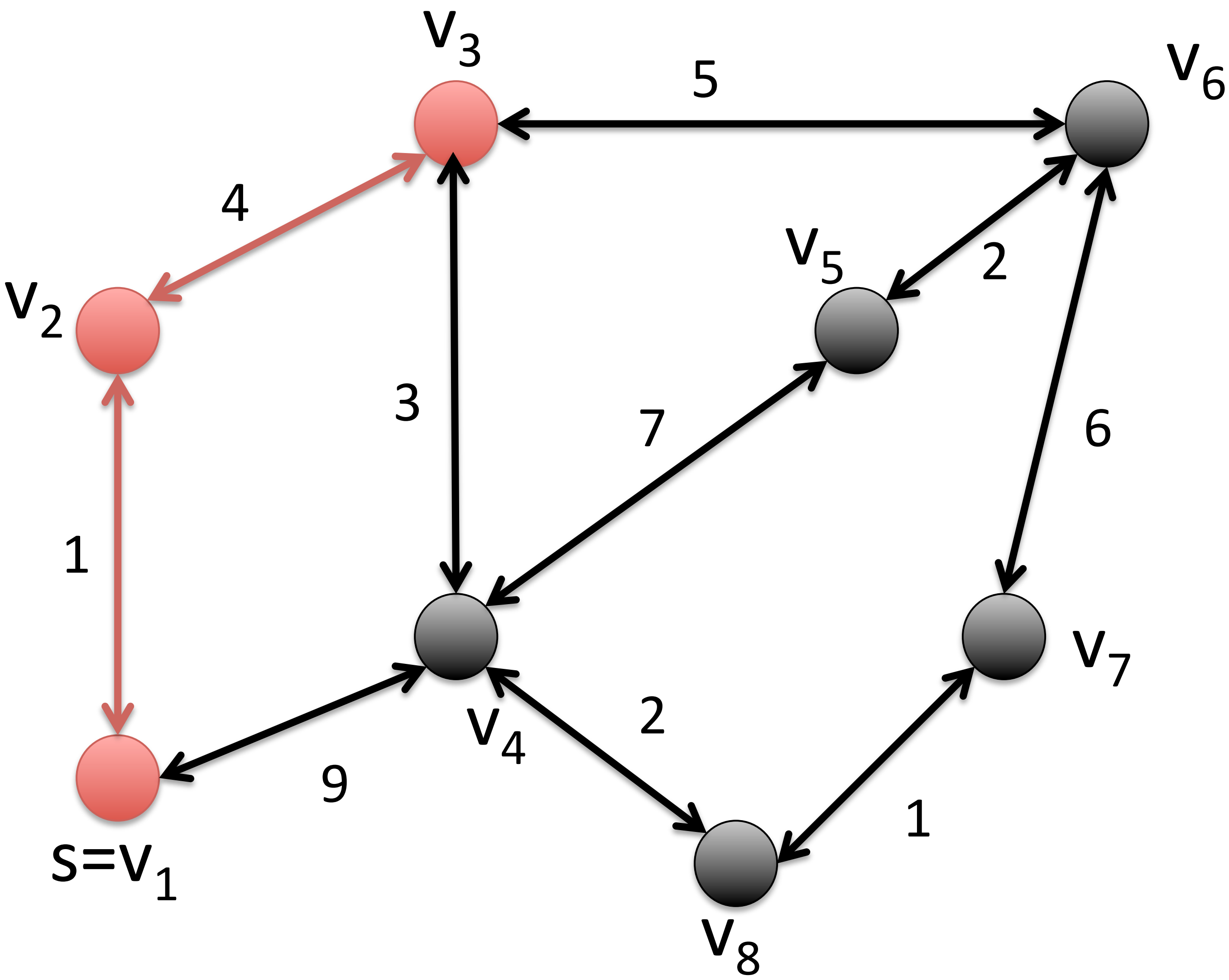
Example



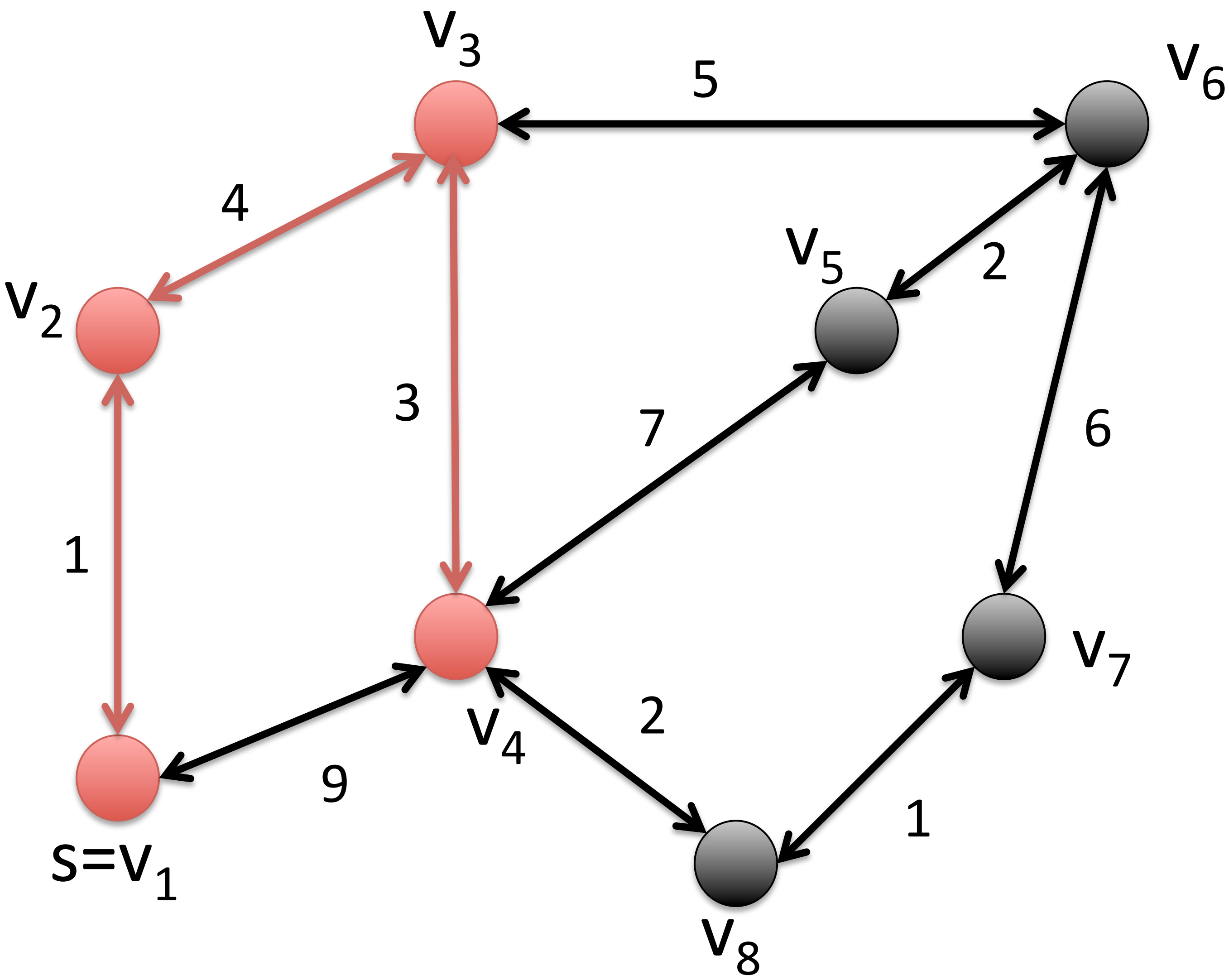
Example



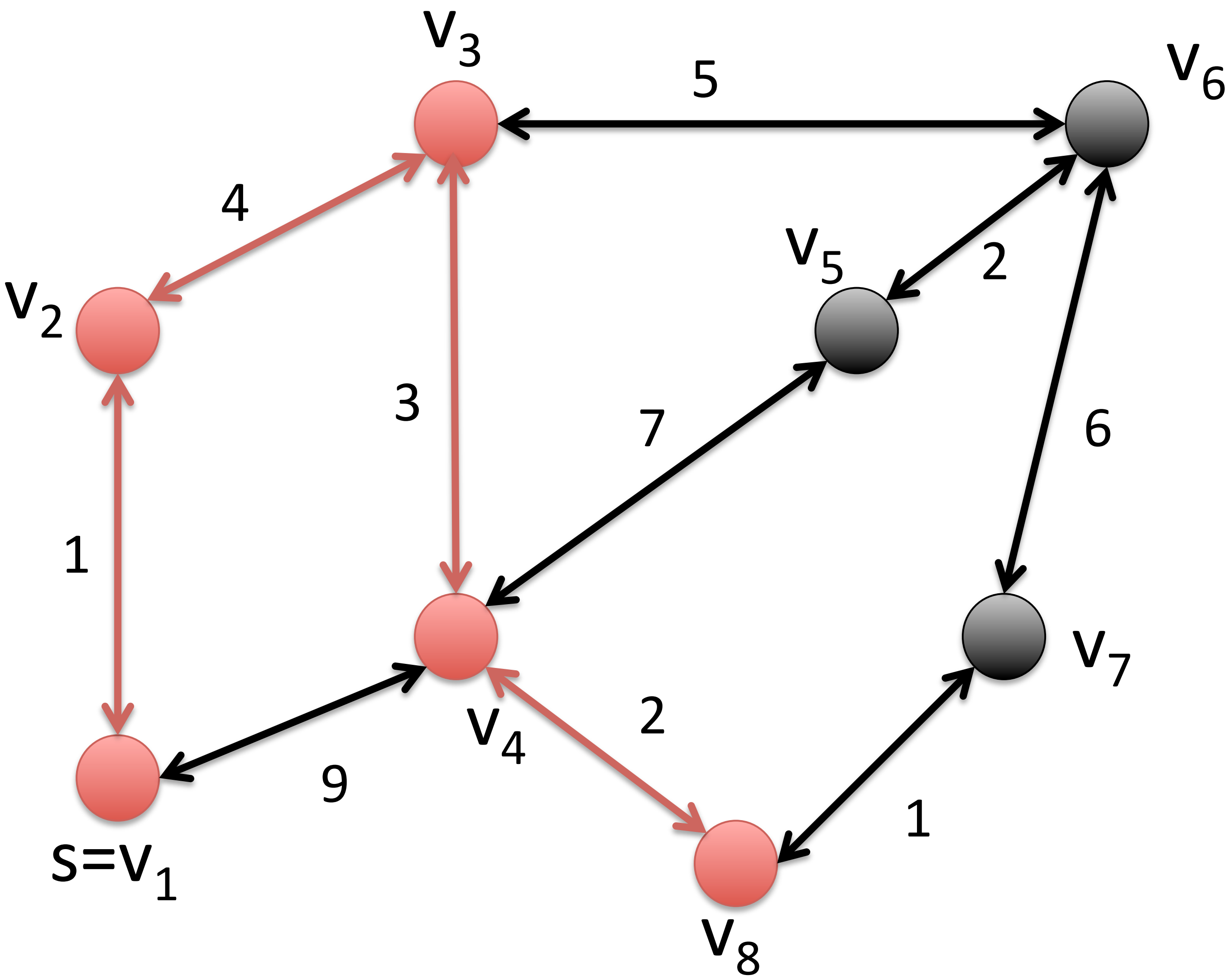
Example



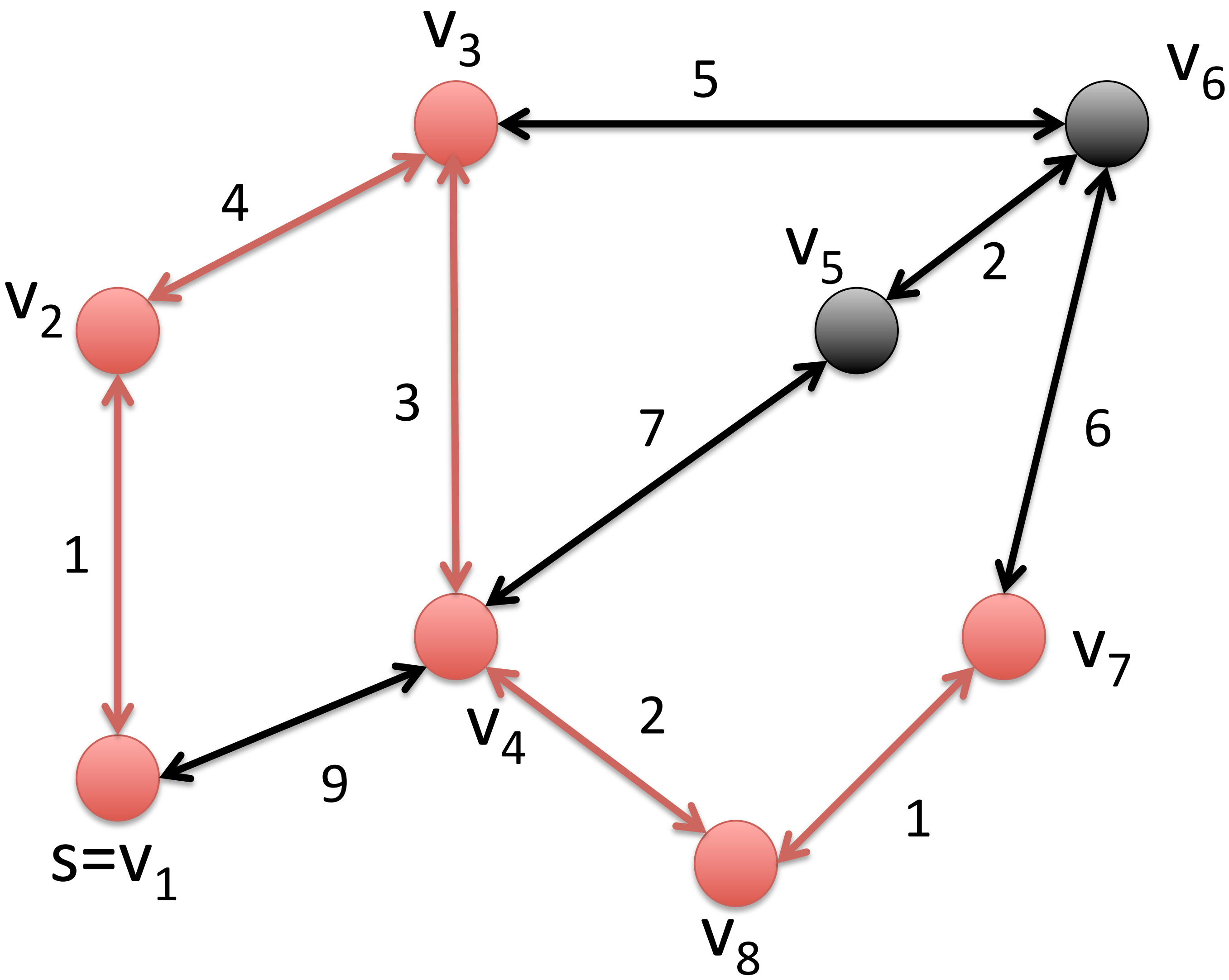
Example



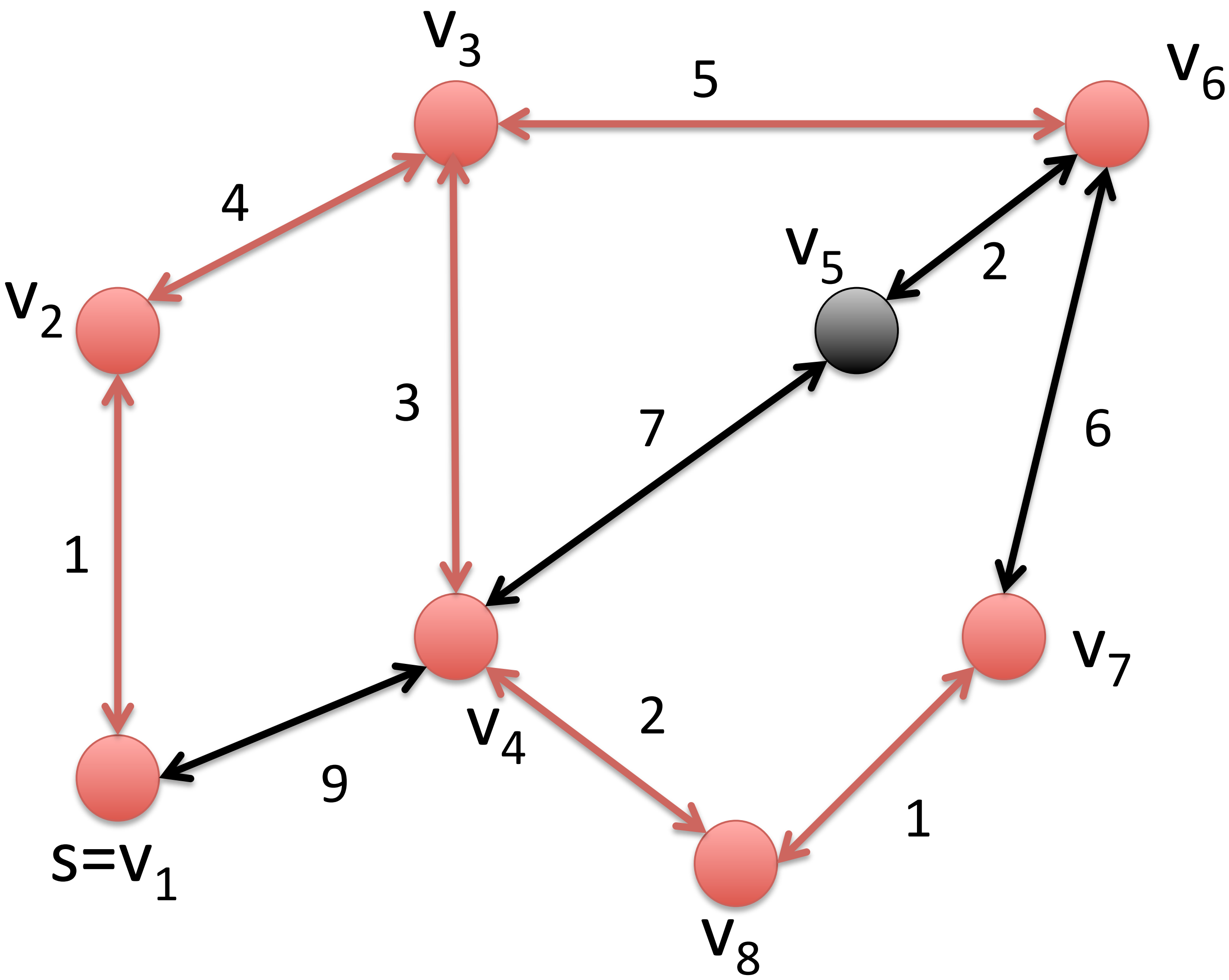
Example



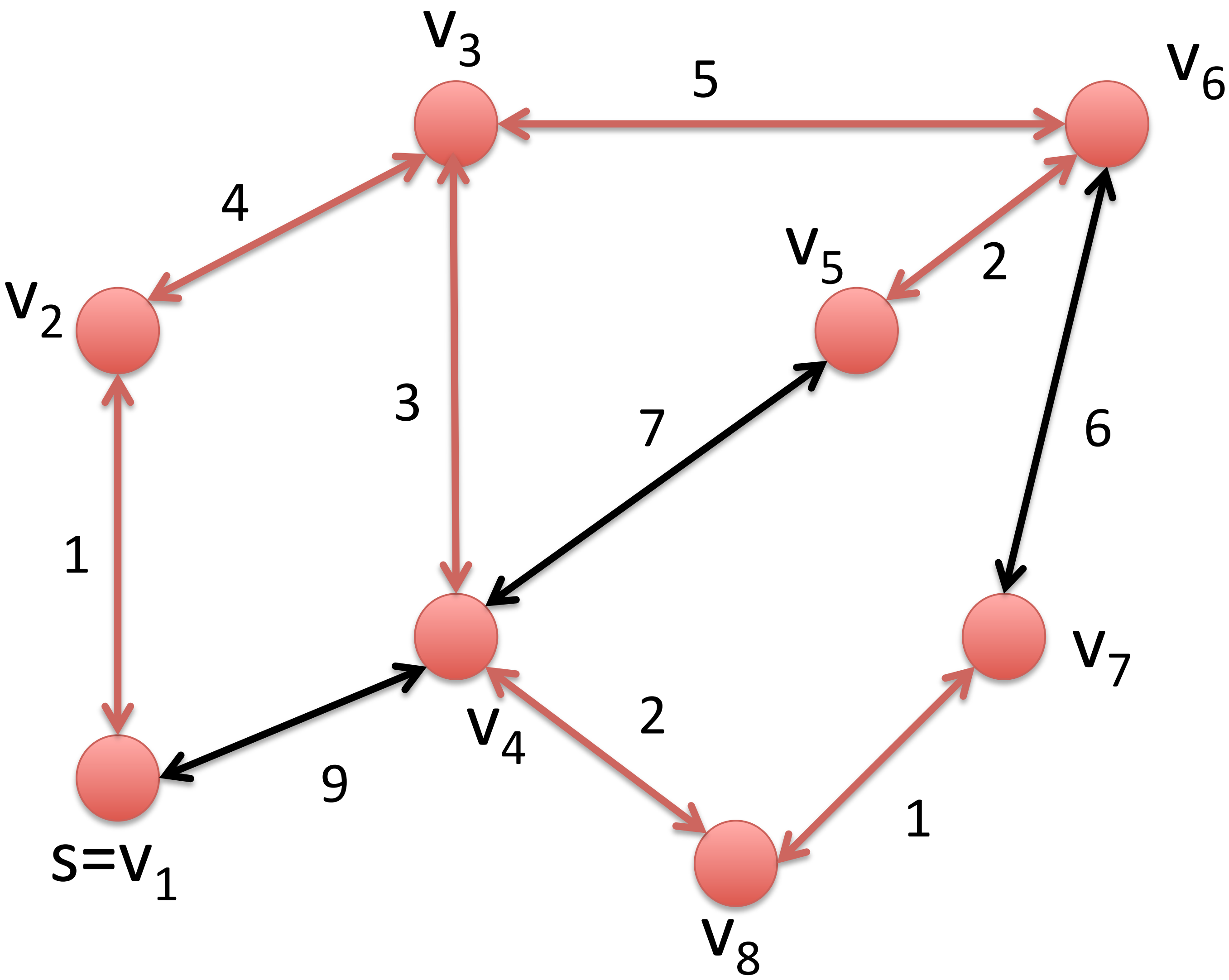
Example



Example



Example



Comparison

- Jarnik-Prim Algorithm can be implemented in time $O(n \log n + m)$
- Kruskal's Algorithm can be implemented in time $O(m \log m)$

Jarnik-Prim Algorithm is more efficient for dense graphs, i. e. where $m = \Theta(n^2)$ holds.