MST & Shortest-Path Algs.

สมชาย ประสิทธิ์จูตระกูล

ภาควิชาวิศวกรรมคอมพิวเตอร์

จุฬาลงกรณ์มหาวิทยาลัย

(28/10/46, 5/10/47)

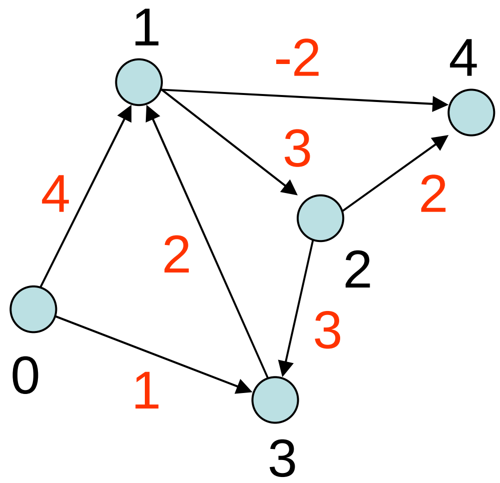
Outline

- Minimum spanning trees
  - Problem definition
  - Kruskal's algorithm
  - Prim's algorithm
- Single-source shortest paths
  - Problem definition
  - Bellman-Ford's algorithm
  - Dijkstra's algorithm

Graph Representations

- adjacency matrix

	0	1	2	3	4
0	-	4	-	1	-
1	-	-	3	-	-2
2	-	-	-	3	2
3	-	2	-	-	-
4	-	-	-	-	-

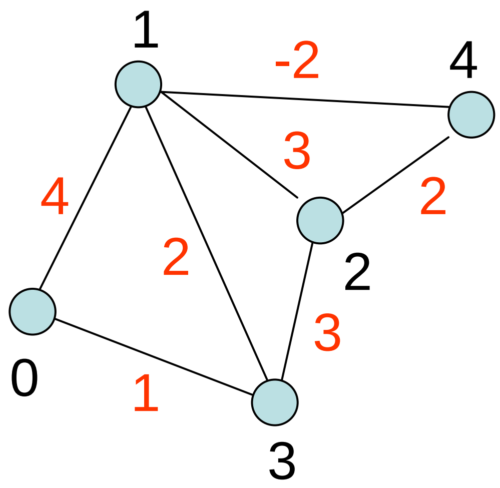


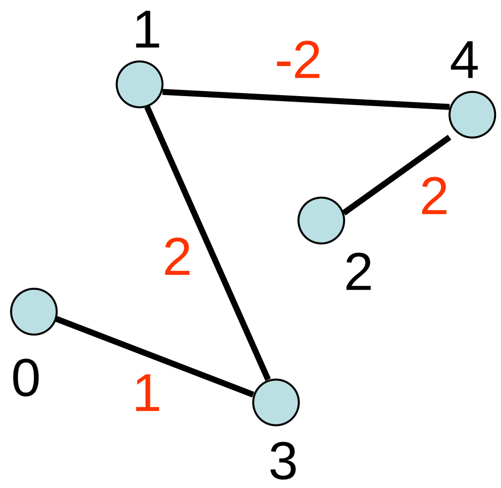
- adjacency list

```
0 → < (1,4), (3,1) >
1 → < (2,3), (4,-2) >
2 → < (3,3), (4,2) >
3 → < (1,2) >
4 → < >
```

Minimum Spanning Trees

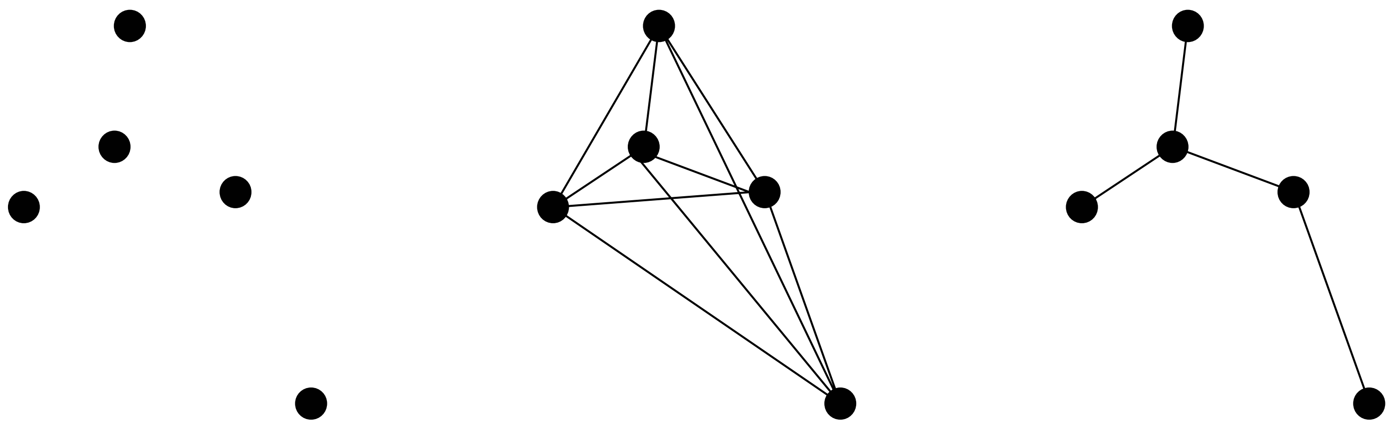
- Given a weighted undirected graph G = (V,E)
- Find a subset of E of minimum weight forming a tree on V





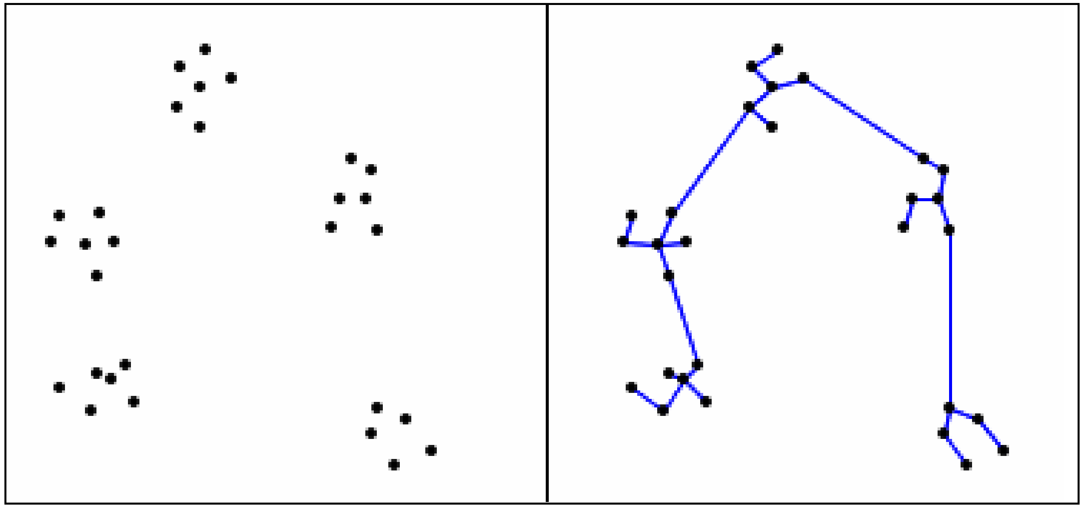
Euclidean MST

- Given a set of points on a Euclidean space
- Find a set of lines with minimum length connecting the points in the space



Applications of MSTs

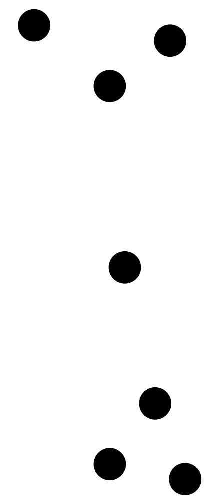
- Minimum cost power wiring
- Computer networks
- Clustering
- Approximated solutions to harder problems
- . . .



Kruskal's Algorithm

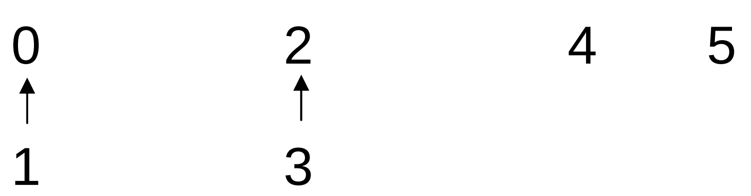
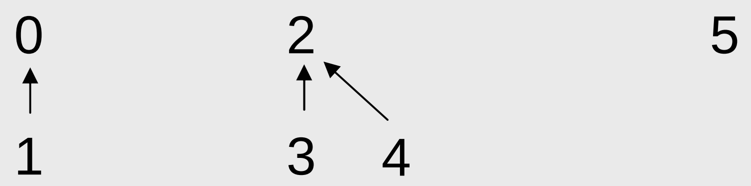
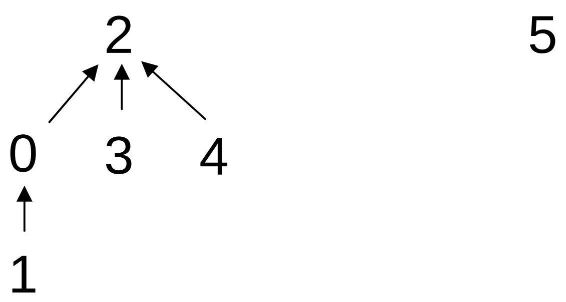
```
Krusal( G=(V,E) ) {
  for each v in V  create a set {v}

  put all edges in E in a priority
    queue Q ordered by weight
  T = {}
  while ( |T| < |V|-1 ) {
    (u,v) = Q.removeMin()
    if ( findSet(u) != findSet(v) ) {
      T = T U {(u,v)}
      union(findSet(u), findSet(v))
    }
  }
  return T
}
```

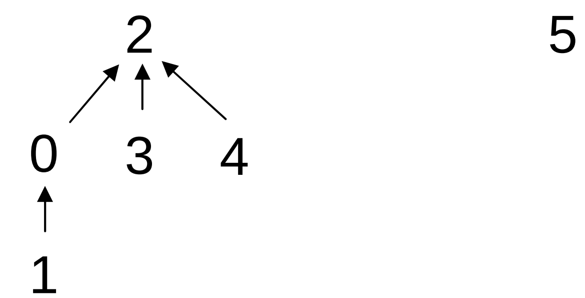


Disjoint Sets

- Disjoint sets whose elements are numbers from 0 to n-1

0 1 2 3 4 5	{0}, {1}, {2}, {3}, {4}, {5}
	{0,1}, {2,3}, {4}, {5}
	{0,1}, {2,3,4}, {5}
	{0,1,2,3,4}, {5}

Disjoint Sets : Representation



{0,1,2,3,4}, {5}

	0	1	2	3	4	5
P	2	0	2	2	2	5

	0	1	2	3	4	5
R	-	-	2	-	-	0

```
findSet( e ) {
  if ( P[e] == e )
    return e;
  else
    return findSet(P[e]);
```

O(log n)

```
unionSet( s, t ) {
  if ( R[s] > R[t] )
    P[t] = s;
  else {
    P[s] = t;
    if (R[s] == R[t]) R[t]++;
  }
}
```

O(1)

Kruskal's Algorithm

```
Krusal( G=(V,E) ) {
  for each v in V   create a set {v}

  put all edges in E in a priority
    queue Q ordered by weight // binary heap
  T = {}
  while ( |T| < |V|-1 ) {
    (u,v) = Q.removeMin()
    if ( findSet(u) != findSet(v) ) {
      T = T U {(u,v)}
      union(findSet(u), findSet(v))
    }
  }
  return T
}
```

$\Theta(v)$

$O(e)$

$O(e \log e)$

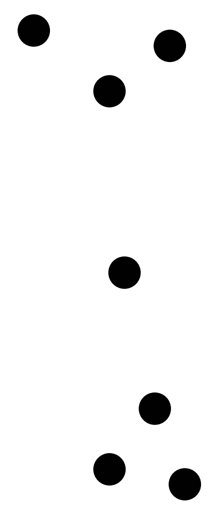
$O(e \log v)$

$O(e \log v)$

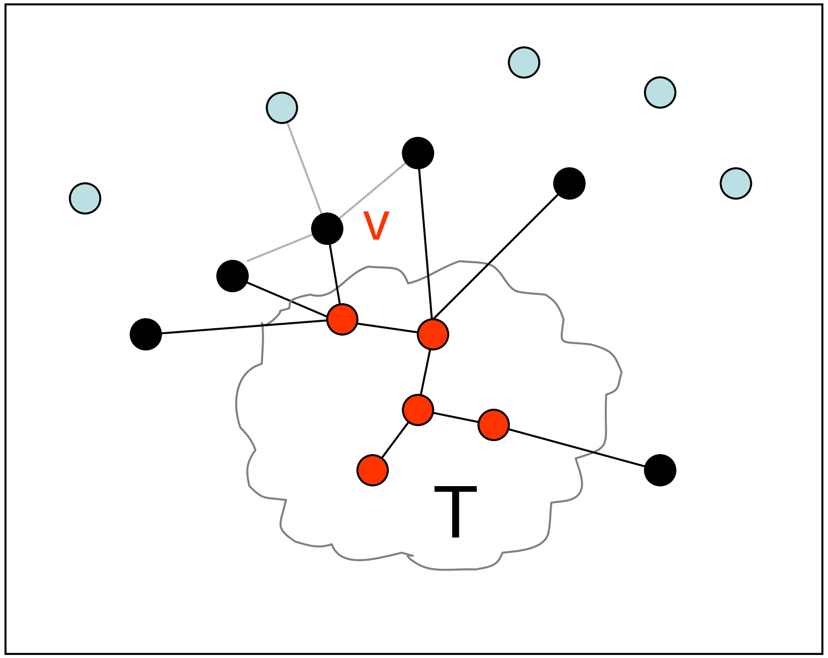
note : $e = O(v^2)$ for simple graphs

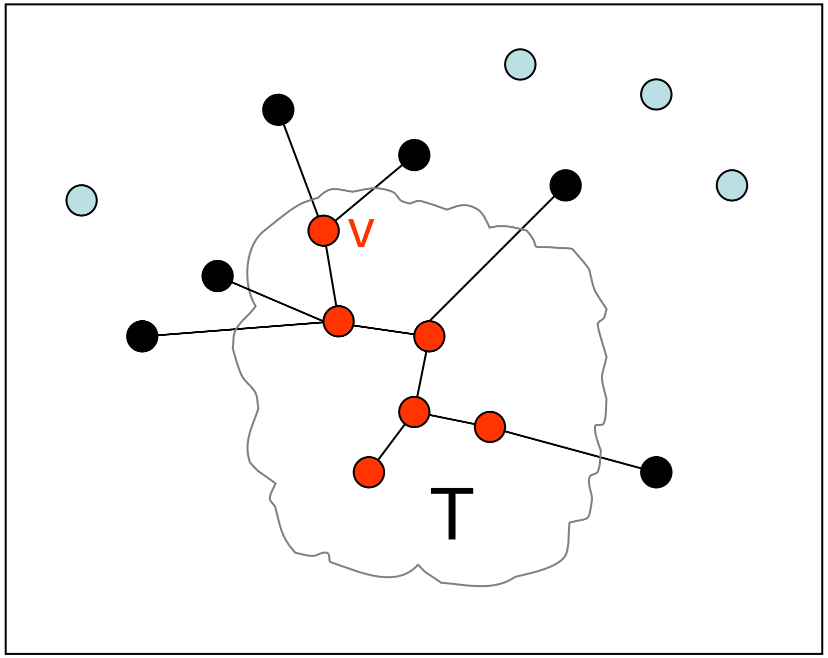
Prim's Algorithm

```
Prim( G=(V,E) ) {
  select an arbitrary vertex v
  S = {v}, T = {}
  while( |S| < n-1 ) {
    select the edge (u,v) of minimum
      weight where u is in T and v is not
    add v into S
    add the edge into T
  }
}
```



Prim's Algorithm






```
for each u adjacent to v {
  if( w(v,u) < d[u] ) d[u] = w(v,u);
}
```

Prim's Algorithm

```
Prim( G=(V,E) ) {
  for each vertex u in V {
    d[u] = INFINITY;  p[u] = null;
  }
  select an arbitrary vertex v and let's d[v] = 0
  put all vertices in V in a priority
    queue Q ordered by d[]  // use binary heap
  T = {}
  while( !Q.isEmpty() ) {
    v = Q.removeMin()
    T = T U ((p[v], v)}
    for each u in Q which is adjacent to v {
      if( w(v,u) < d[u] ) {
        Q.decreaseKey(u, w(v,u));
        p[u] = v;
      }
    }
  }
  return T;
}
```

$\Theta(v)$

$O(v)$

$O(v \log v)$

$O(e \log v)$

$O(e \log v)$

Prim's Algorithm

Θ(v)

O(v^2)

O(v^2)

```
Prim( g[][] ) {
  for each vertex u in V {
    d[u] = INFINITY;  p[u] = null;
  }
  select an arbitrary vertex v and let's d[v] = 0
  T = {}
  for ( i = 1 to |V| ) {
    v = minIndex(d);
    d[v] = INFINITY;
    T = T U ((p[v], v)}
    for ( u = 1 to |V| ) {
      if( d[u] < INFINITY AND w(v,u) < d[u] ) {
        d[u] = w(v,u);
        p[u] = v;
      }
    }
  }
  return T;
}
```

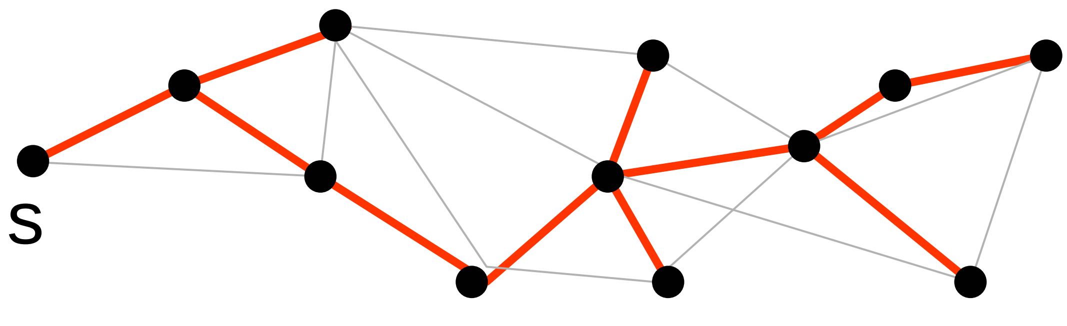
O(v^2)

Kruskal vs. Prim

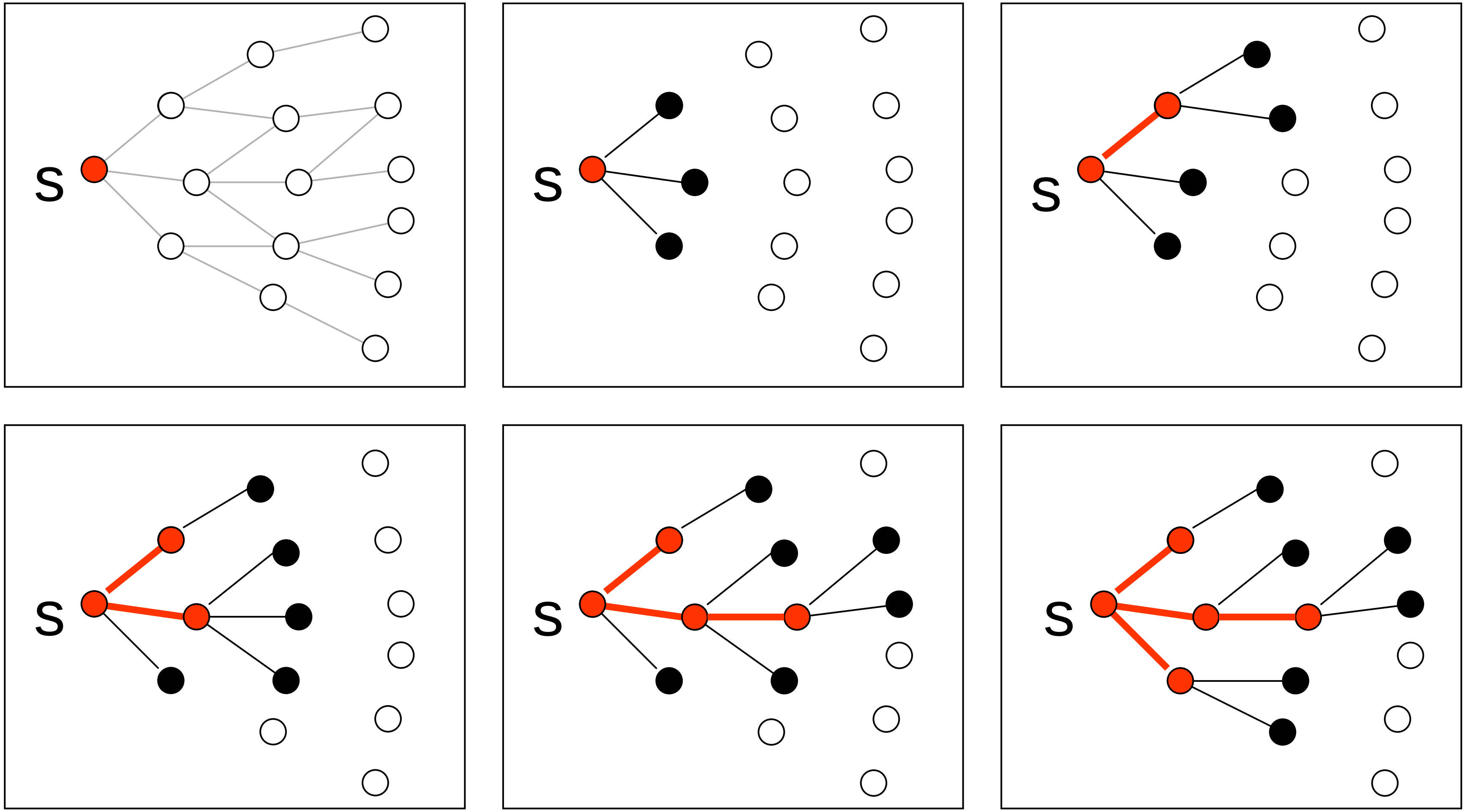
- Kruskal
  - good for sparse graph O(e log v)
  - still O(e log v), if we use path compression
- Prim
  - using simple list gives $O(v^2)$ which is optimal for dense graphs
  - using binary heap takes O(e log v)

Single-Source Shortest Paths

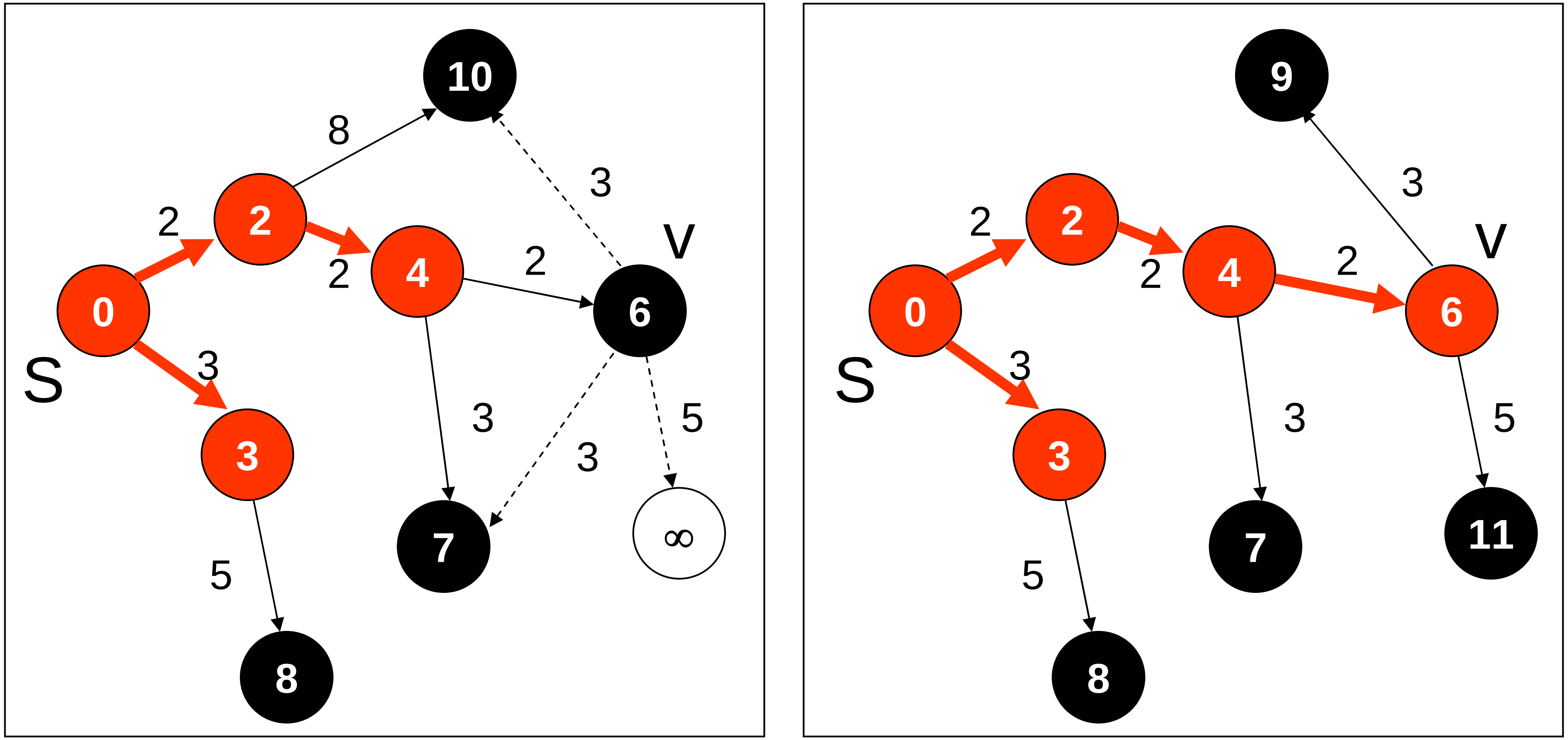
- Given a weighted directed graph G and a source vertex s
- Find a shortest path from s to every vertex in G
- No negative-weight cycle
- the problem has optimal substructures



Dijkstra's Algorithm



Relaxation




```
for each u adjacent to v {
  if( d[v]+w(v,u) < d[u] )
    d[u] = d[v]+w(v,u);
}
```

Prim's Algorithm

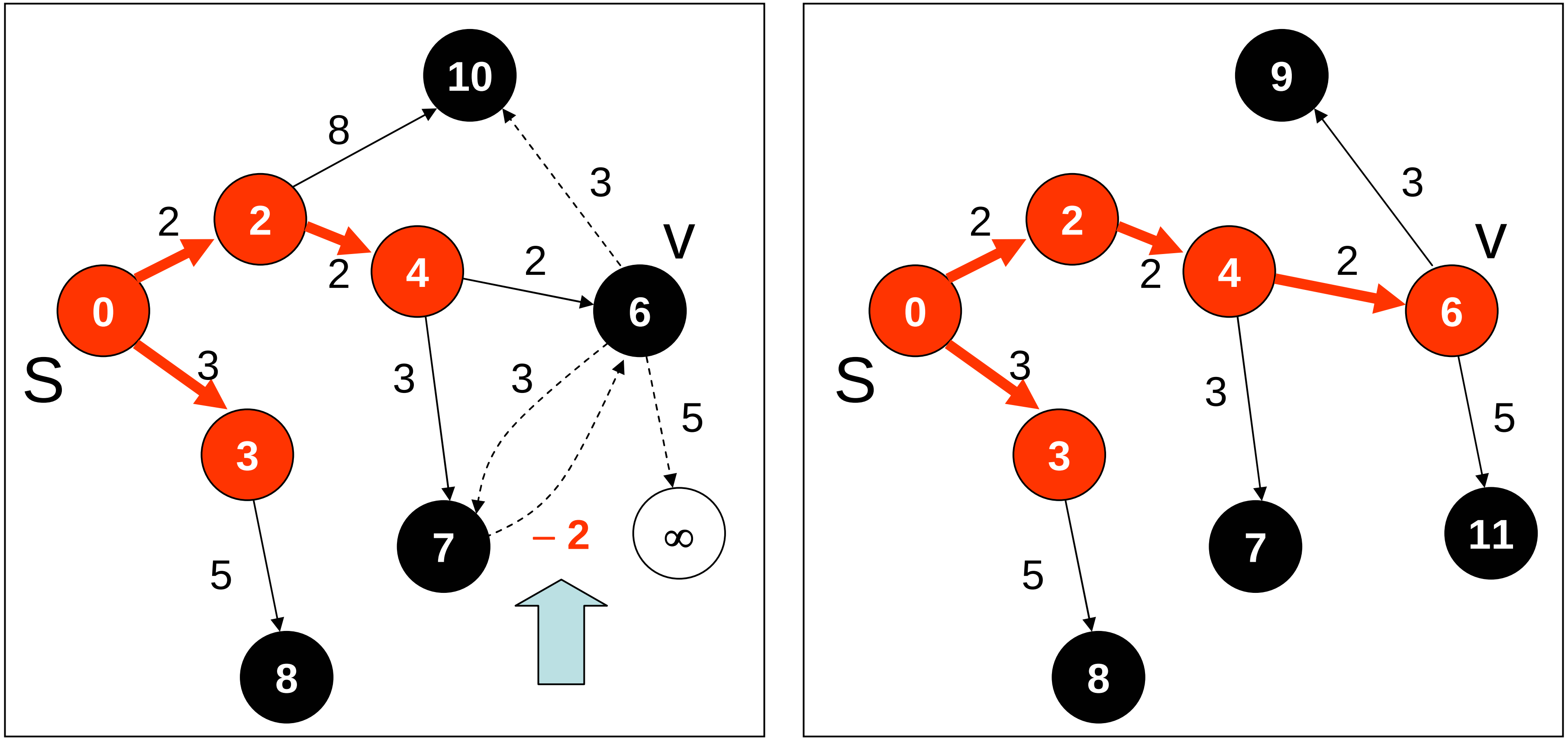
```
Prim( G=(V,E) ) {
  for each vertex u in V {
    d[u] = INFINITY;  p[u] = null;
  }
  select an arbitrary vertex v and let's d[v] = 0
  put all vertices in V in a priority
    queue Q ordered by d[]  // binary heap
  T = {}
  while( !Q.isEmpty() ) {
    v = Q.removeMin()
    T = T U {(p[v], v)}
    for each u in Q which is adjacent to v {
      if( w(v,u) < d[u] ) {
        Q.decreaseKey(u, w(v,u));
        p[u] = v;
      }
    }
  }
  return T;
}
```

Dijkstra's Algorithm

```
Dijkstra( G=(V,E), s ) {
  for each vertex u in V {
    d[u] = INFINITY;  p[u] = null;
  }
  d[s] = 0
  put all vertices in V in a priority
    queue Q ordered by d[]  // binary heap
  while( !Q.isEmpty() ) {
    v = Q.removeMin()
    for each u in Q which is adjacent to v {
      if( d[v]+w(v,u) < d[u] ) {
        Q.decreaseKey(u, d[v]+w(v,u));
        p[u] = v;
      }
    }
  }
  return (p[], d[]);
}
```

O(e log v)

Negative Weight



Dijkstra's algorithm gives an optimal solution if there is no negative-weight edges in the graph.

Bellman-Ford Algorithm

- work with negative-weight edge
- can detect negative-weight cycle

Bellman-Ford's Algorithm

```
Bellman_Ford( G=(V,E), s ) {
  for each vertex u in V {
    d[u] = INFINITY;
  }
  d[s] = 0; p[s] = null;
  for ( i = 1 to |V| ) {
    for each edge (u,v) in E {
      if( d[v]+w(v,u) < d[u] ) {
        p[u] = v;  d[u] = d[v]+w(v,u);
      }
    }
  }
  for each edge (u,v) in E {
    // if true -> negative-weight cycle
    if( d[v]+w(v,u) < d[u] ) return (null, null);
  }
  return (p[], d[]);
}
```

O(ve)

Etc.

- Shortest path on DAG
- Using MST to approx. TSP