# **DS221**: Introduction to Scalable Systems

## Topic: Algorithms and Data Structures

# L5: Graphs (continued)

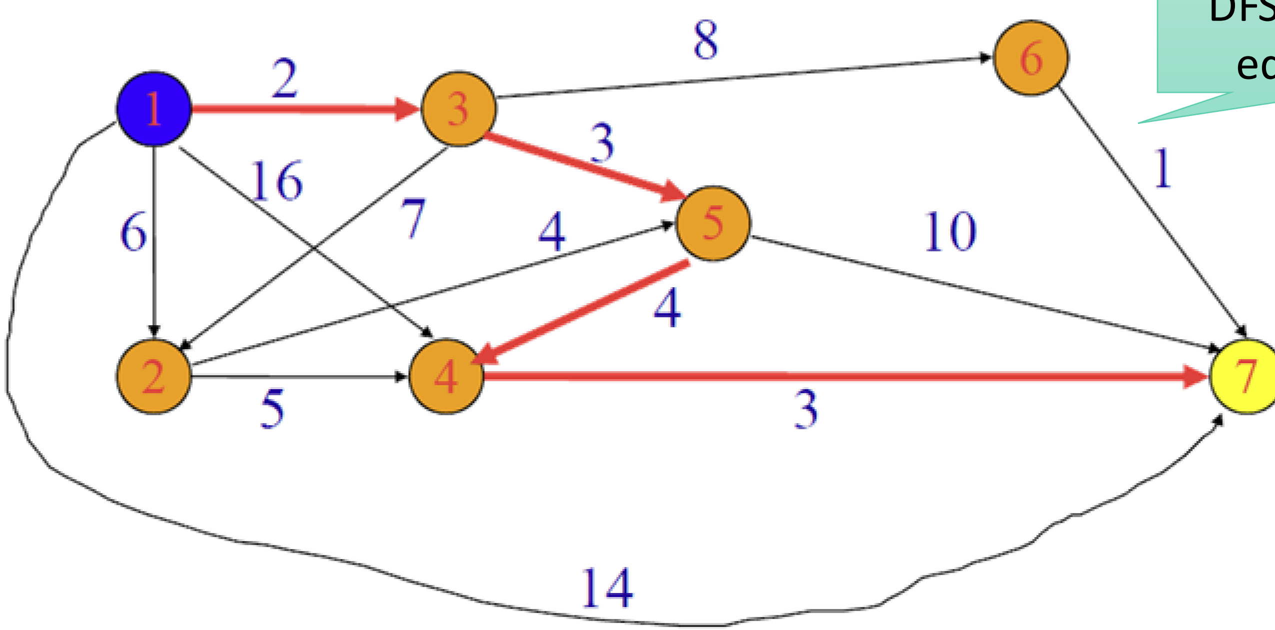
Graph ADT, Algorithms

# **Shortest Path**: Single source, single destination

- Possible greedy algorithm
  - Leave source vertex using *shortest outgoing edge*
  - Leave new vertex again using shortest outgoing edge to an *unvisited vertex*
  - Continue until destination is reached

This is same as the DFS with shortest edge selected!

*Greedy Path from 1 To 7*



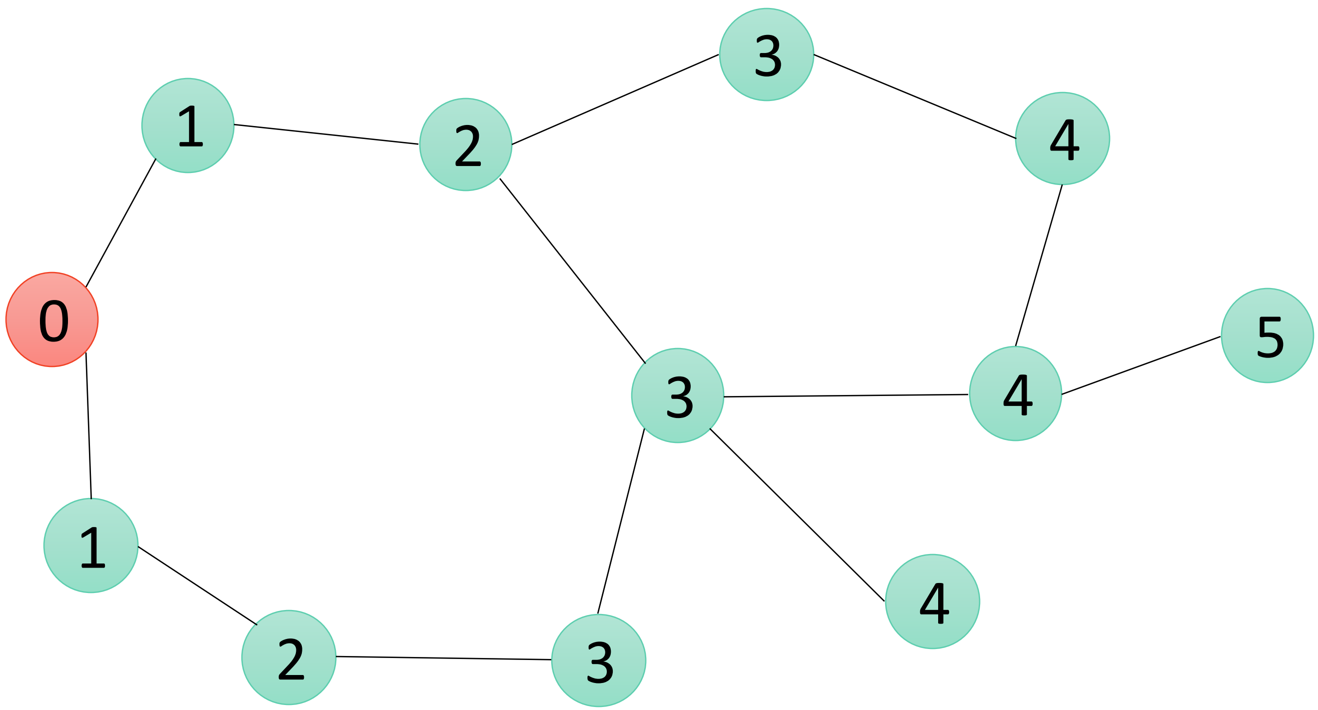

Length of 12 is not shortest path!

# Single Source Shortest Path

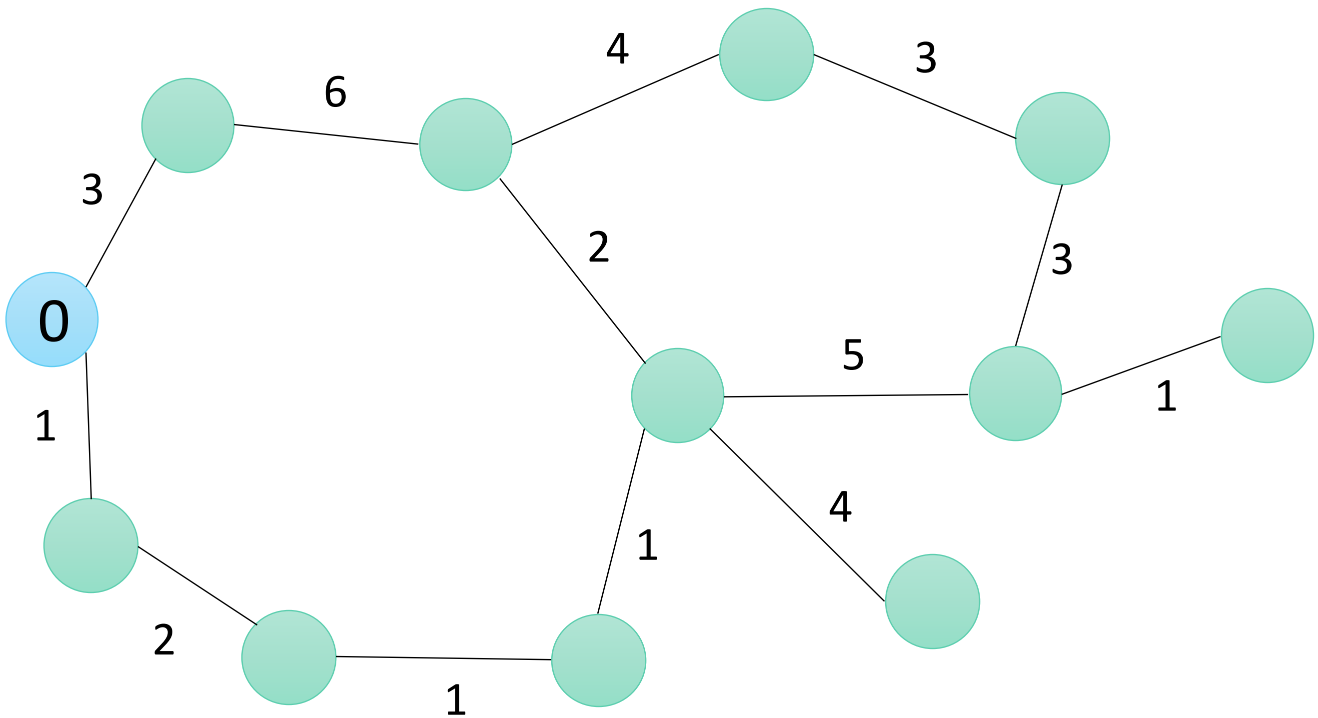
- Shortest distance from one source vertex to all destination vertices
- Is there a simple way to solve this?
- …Say if you had an unit-weighted graph?

- Just do Breadth First Search (BFS)! ☺

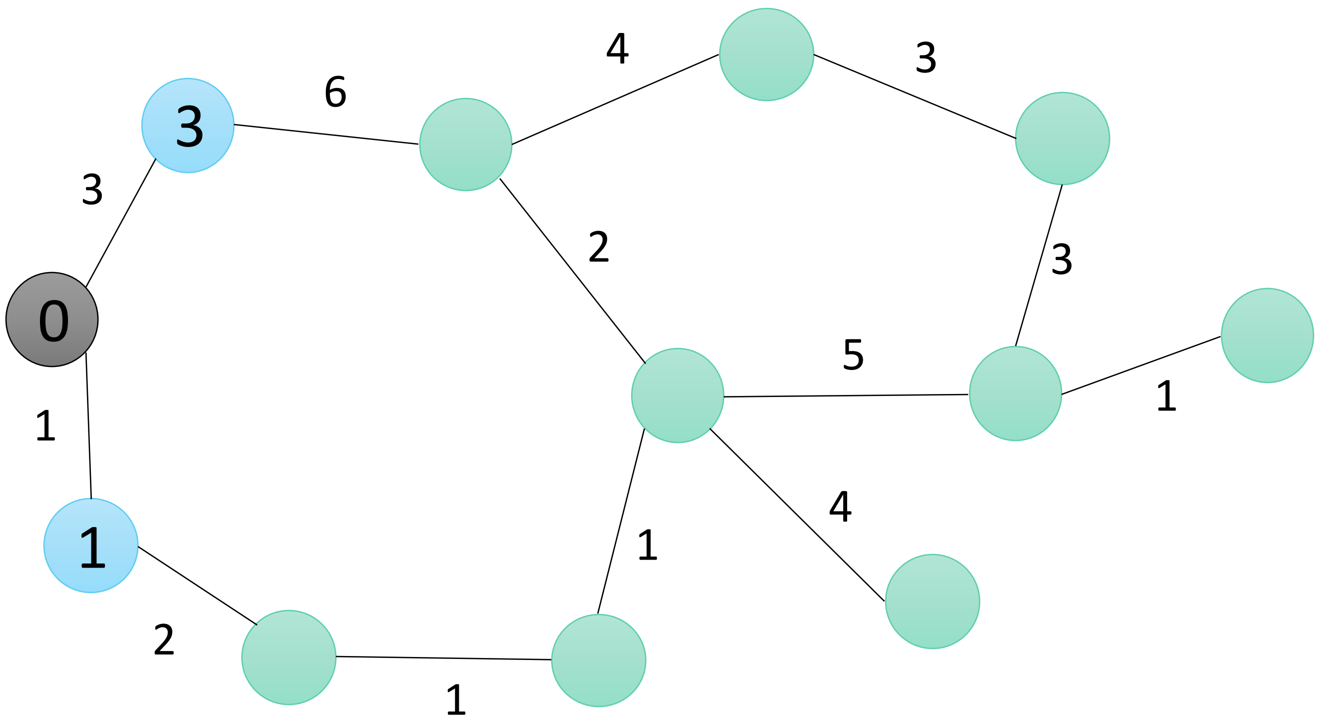
# SSSP: BFS on Unweighted Graphs



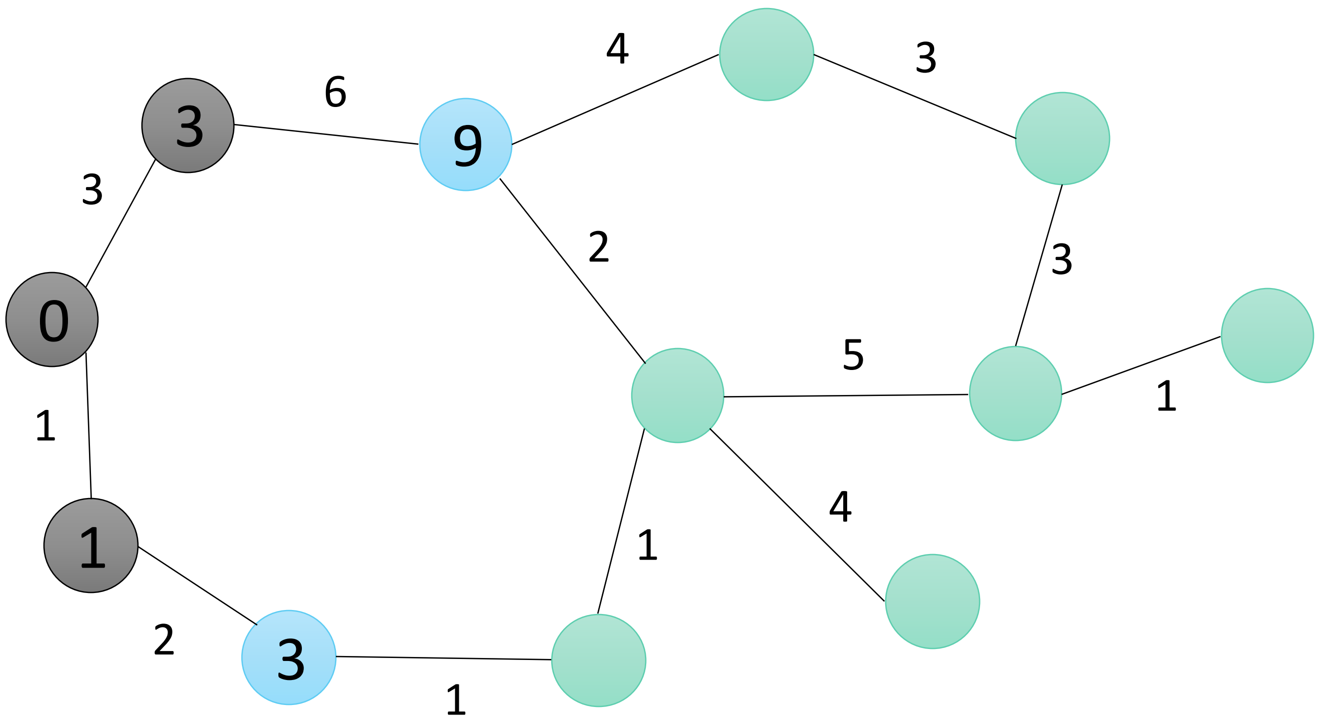
# SSSP: *BFS on Weighted Graphs?*



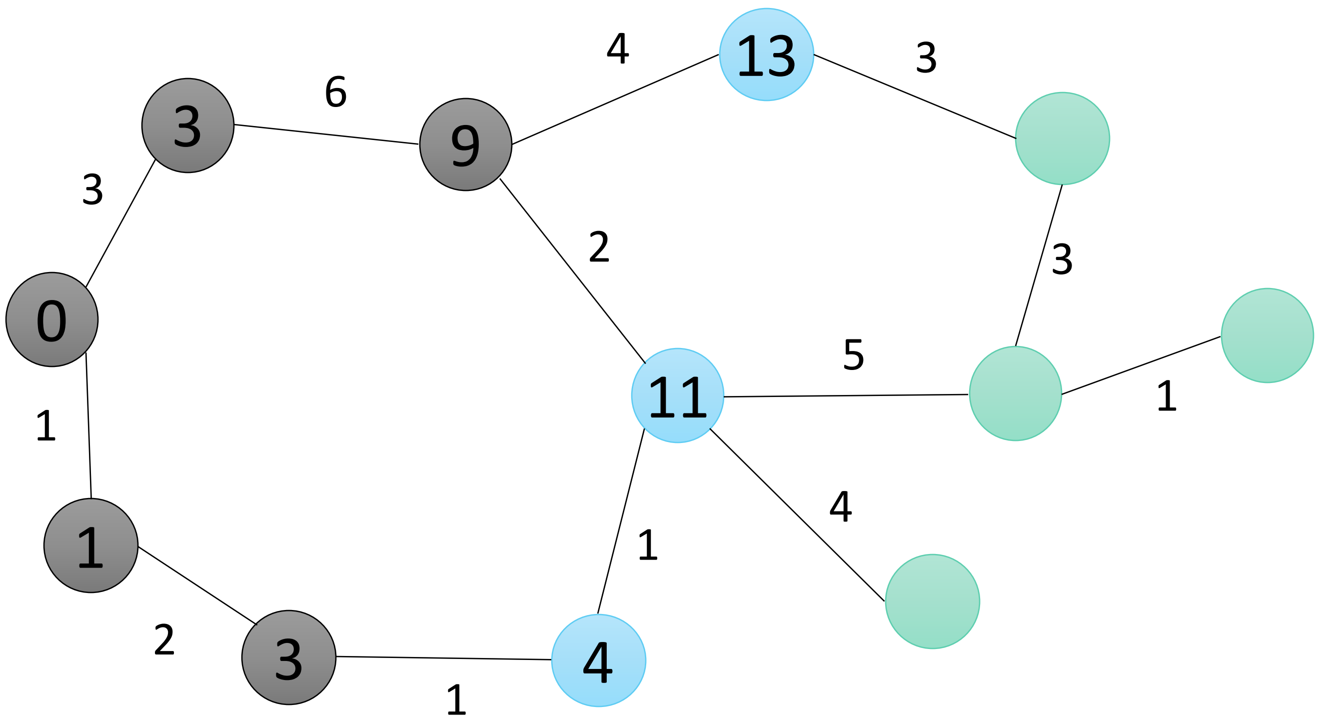
# SSSP: BFS on Weighted Graphs?



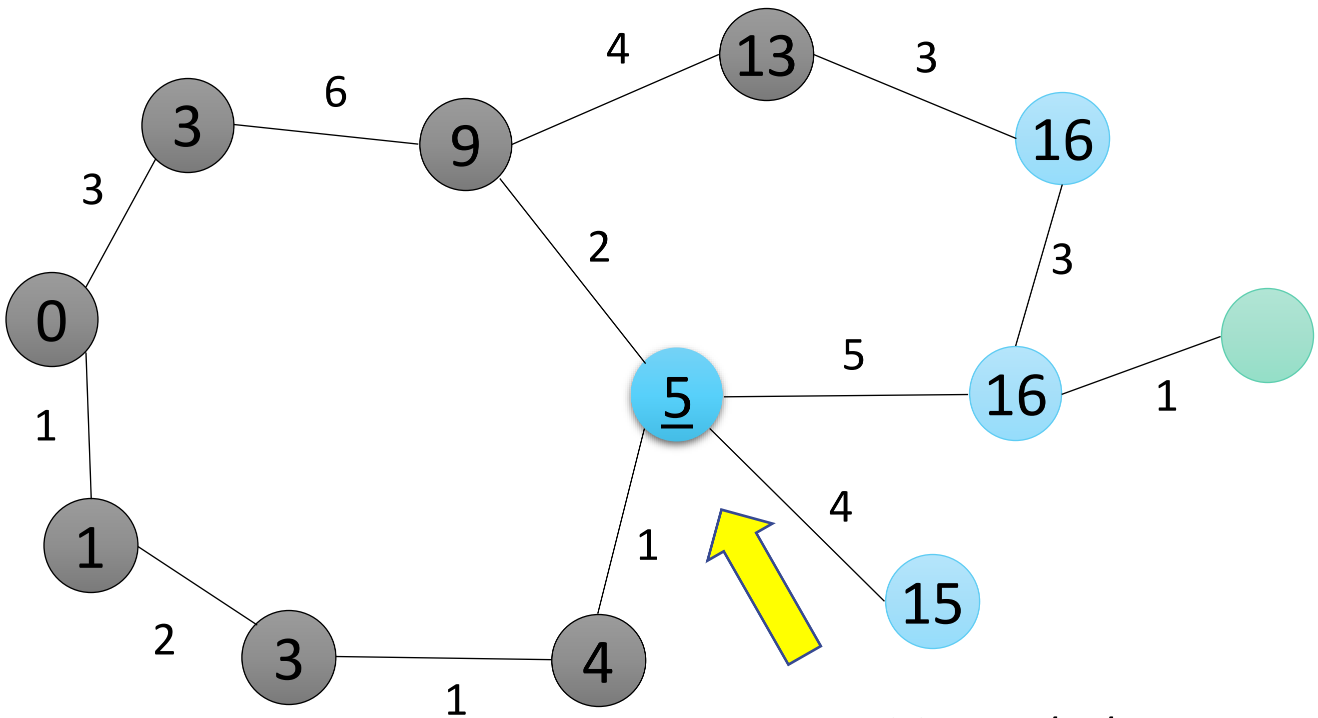
# SSSP: BFS on Weighted Graphs?



# SSSP: BFS on Weighted Graphs?

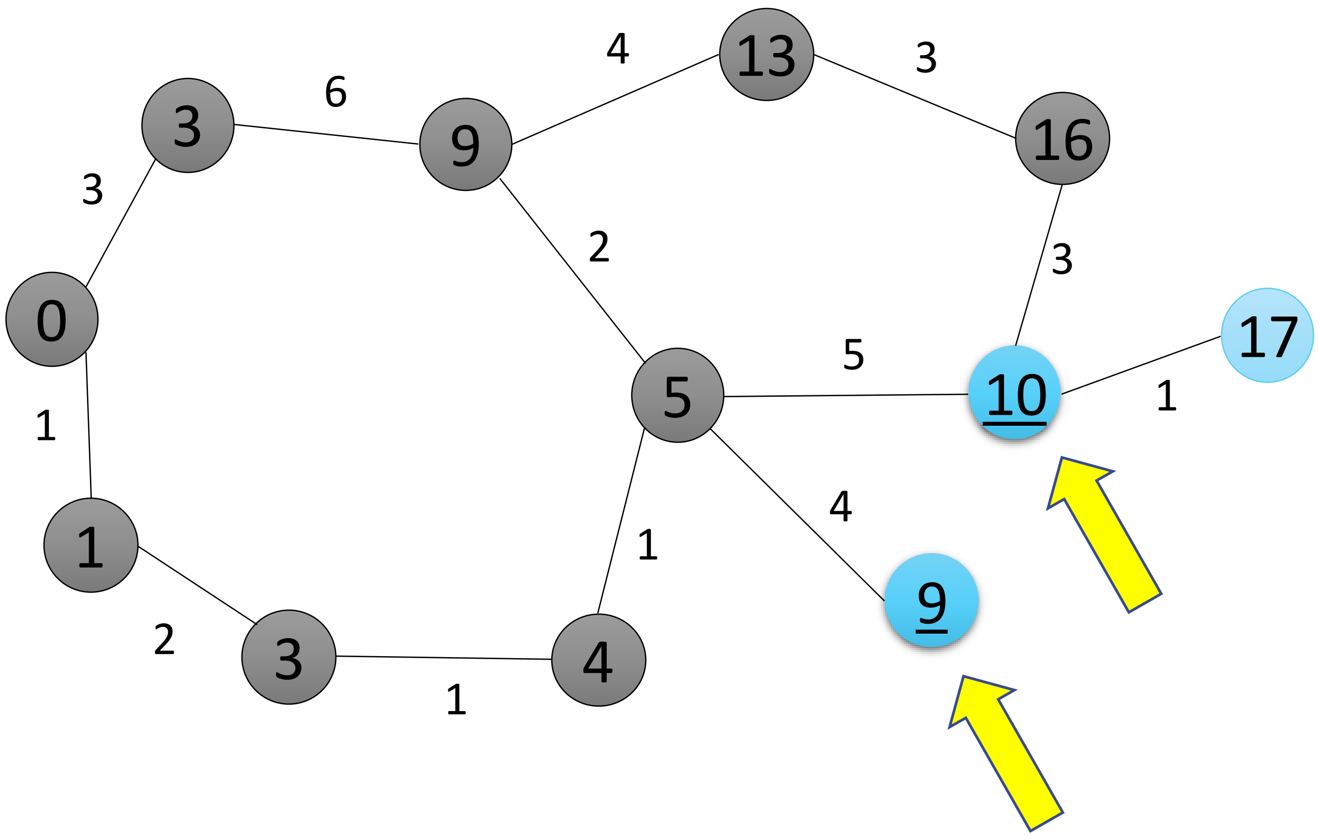


# SSSP: BFS on Weighted Graphs?

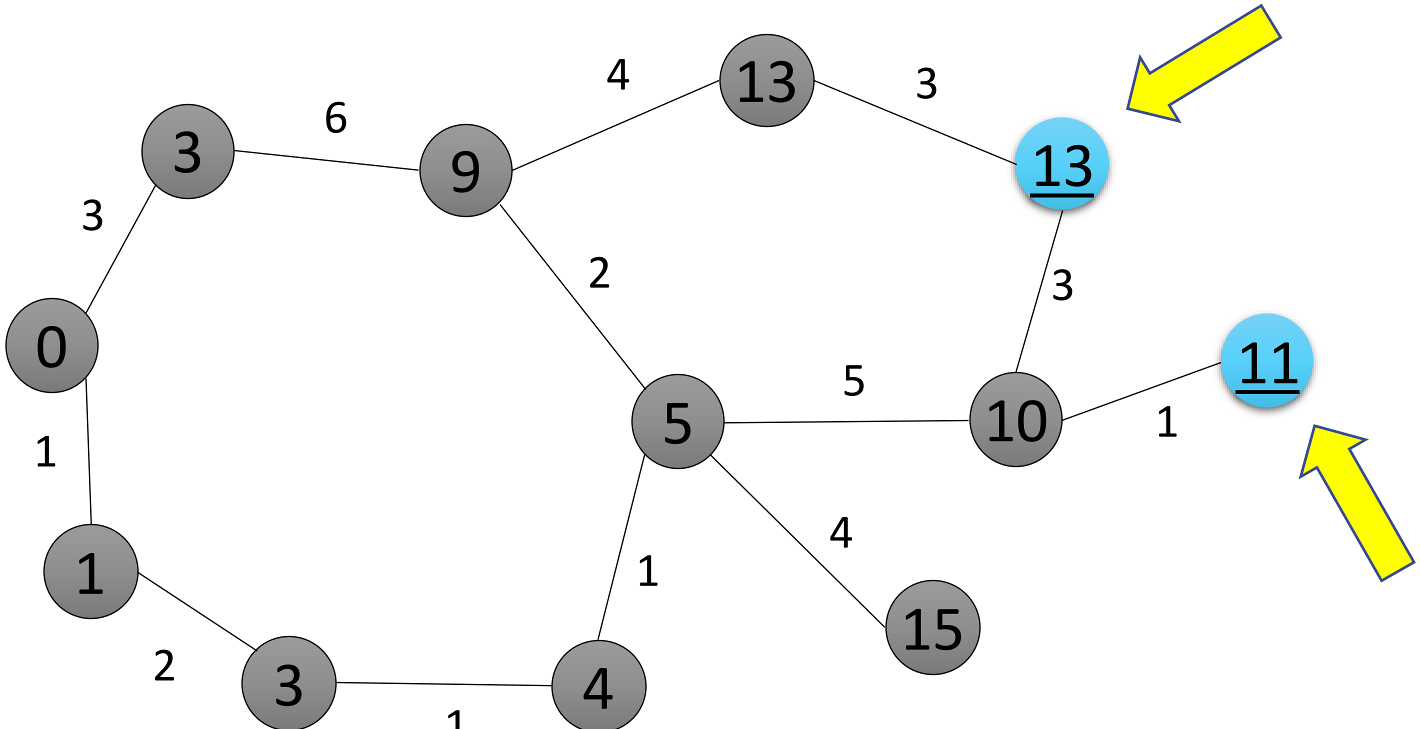



*Revisit, recalculate, re-propagate…*
cascading effect

# SSSP: BFS on Weighted Graphs?



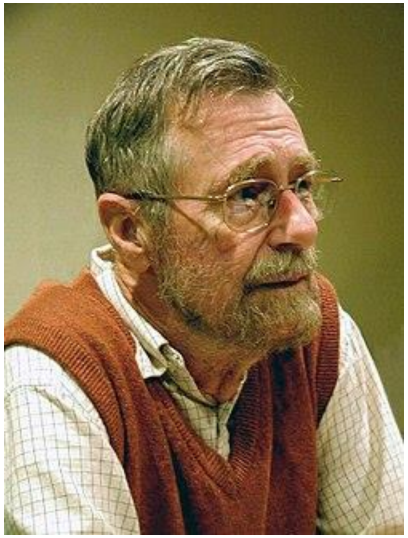
# SSSP: BFS on Weighted Graphs?




BFS with revisits is not efficient. Can we be smart about order of visits?

# Dijkstra's Single Source Shortest Path Algorithm (SSSP)

- Named after its inventor Edsger Dijkstra (1930-2002)
- One of the "founders" of computer science; this algorithm is one of his many contributions



# Invariant of an algorithm

- An **invariant of an algorithm** is a property (a condition or statement) that:
  - Holds true before the algorithm starts,
  - Remains true throughout every step
  - Is still true when the algorithm finishes
- Invariants are often used to **prove correctness** of algorithms.
- Example: In the binary search algorithm over a sorted array, an invariant is:

  *"The target value, if it exists, is always within the current search interval."*

# Dijkstra's Single Source Shortest Path Algorithm (SSSP)

- Solves SSSP problem on a weighted, directed graph G = (V, E)
- Assumption: All edge weights are nonnegative
- s is the source vertex
- Let $\delta$(s,x) be the shortest distance from s to x
- Algorithm will maintain the following invariants:
  - $\forall$x $\in$ V, the algorithm maintains an **estimate** d[x] of the distance of vertex x from s such that:
    - At any point in time, d[x] $\geq \delta$(s,x)
    - When x is "finished", d[x] $= \delta$(s,x)

# Dijkstra’s Single Source Shortest Path Algorithm (SSSP)

- Let **w(x,y)** denote the weight of an edge from x to y
- Initialize distance estimates of all vertices to $\infty$
- Update d[s] to 0
- **F** is the set of vertices that are yet to achieve final distance estimates. Initialize F to V
- **D** is the set of vertices that have achieved final distance estimates

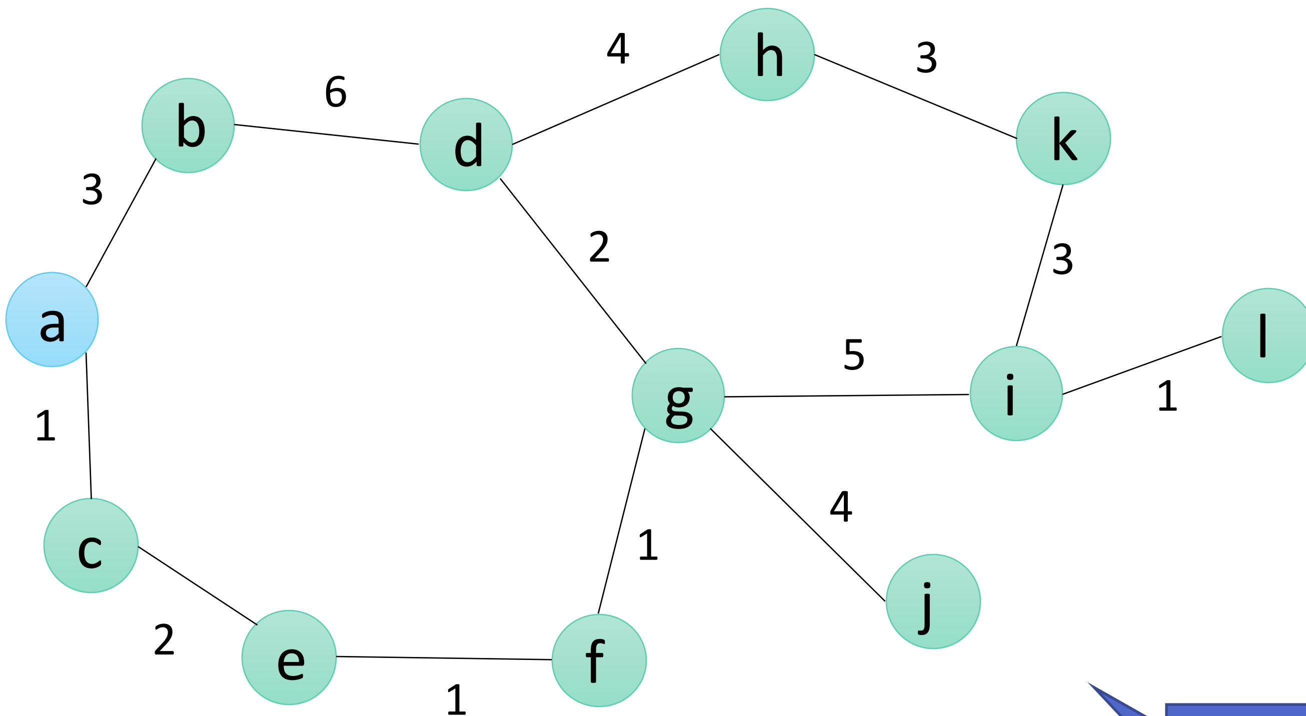
# Dijkstra's Single Source Shortest Path Algorithm (SSSP)

- Let **w(x,y)** denote the weight of an edge from x to y
- Initialize distance estimates of all vertices to $\infty$
- Update d[s] to 0
- **F** is the set of vertices that are yet to achieve final distance estimates. Initialize F to V
- **D** is the set of vertices that have achieved final distance estimates
- Algorithm:

while (**F** is not empty)

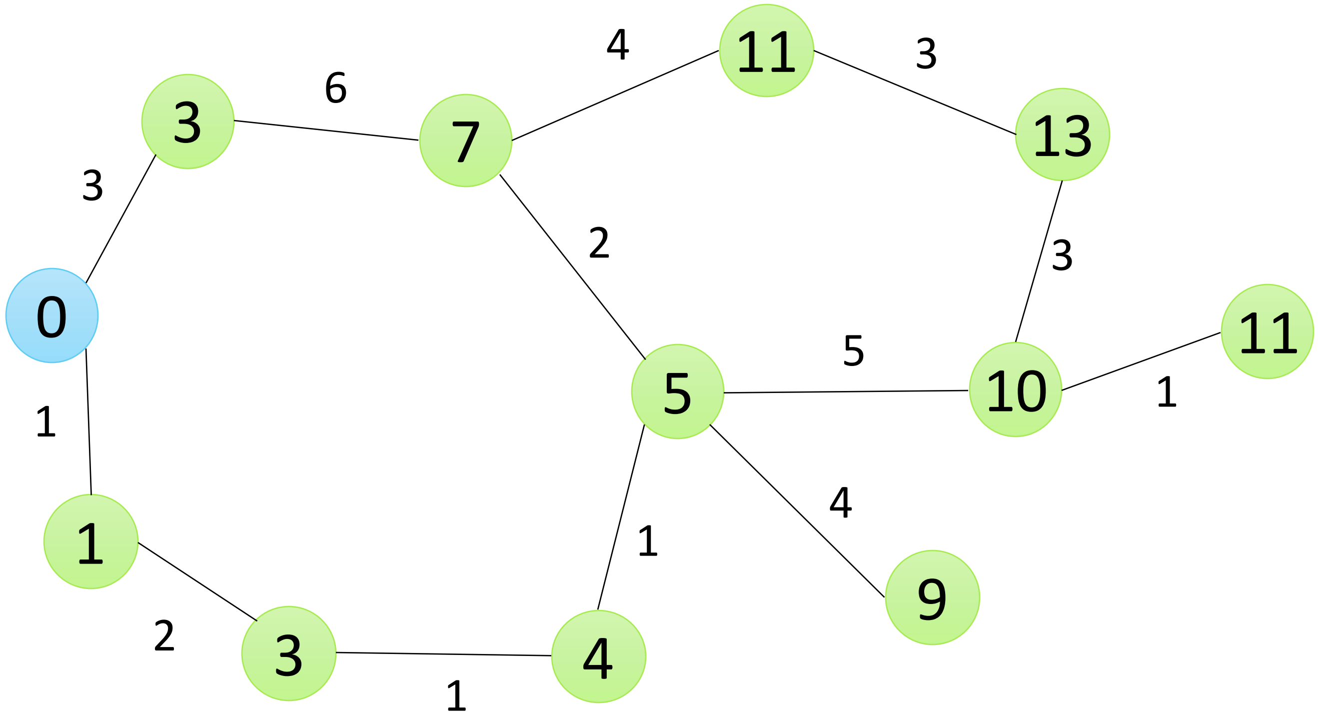
- Let **x** be the vertex in **F** with minimum distance estimate
- Remove x from **F** and add it to **D**
- For all edges (x,y) from x, update distance estimate of y:
  d[y] = min (d[y], d[x] + w(x,y))

# SSSP on Weighted Graphs




Work out yourself !

# SSSP on Weighted Graphs



# Dijkstra's Single Source Shortest Path Algorithm (SSSP)

- Let **w(x,y)** denote the weight of an edge from x to y
- Initialize distance estimates of all vertices to $\infty$
- Update d[s] to 0
- **F** is the set of vertices that are yet to achieve final distance estimates. Initialize F to V
- **D** is the set of vertices that have achieved final distance estimates
- Algorithm:

while (**F** is not empty)

- Let **x** be the vertex in **F** with minimum distance estimate
- Remove x from **F** and add it to **D**
- For all edges (x,y) from x, update distance estimate of y:
  d[y] = min (d[y], d[x] + w(x,y))

Does invariant d[u] ≥ $\delta$(s,u) hold for all vertices u∈ $V$ ?

# Dijkstra's Single Source Shortest Path Algorithm (SSSP)

- **Claim 1:** For every vertex u, at any point of time d[u] ≥ $\delta$(s,u)
- We will prove that at any point in time, if d[u] < ∞, then **d[u] is the weight of some path from s to u**

Proof by induction

- Base case (at initialization): d[s] = 0 = $\delta$(s,s) and all other estimates are $+\infty$
- Induction hypothesis: Claim holds till vertex x was picked from set F
- Now if y is a out-neighbour of x, we may update d[y] to d[x]+w(x,y)
- By the induction hypothesis, there exists a path from s to x with weight d[x]. We also have edge (x,y) of weight w(x,y). Clearly, there exists a path from s to y of weight d[x]+w(x,y)

# Dijkstra's Single Source Shortest Path Algorithm (SSSP)

- Let **w(x,y)** denote the weight of an edge from x to y
- Initialize distance vector **d[ ]** for all vertices to infinity, except for source which is set to 0
- **F** is the set of vertices that are yet to … distance estimates. Initialize F to V
- **D** is the set of vertices that have achiev… al distance estimates
- Algorithm:

while (**F** is not empty)

- Let **x** be the vertex in **F** with minimum distance estimate
- Remove x from **F** and add it to **D**
- For all edges (x,y) from x, update distance estimate of y:

  d[y] = min (d[y], d[x] + w(x,y))

Now, how do we show that d[x]= $\delta$(s,x) ?

# Dijkstra's Single Source Shortest Path Algorithm (SSSP)

- **Claim 2:** When vertex x is placed in D, d[x] = $\delta$(s,x)

(Proof by contradiction)

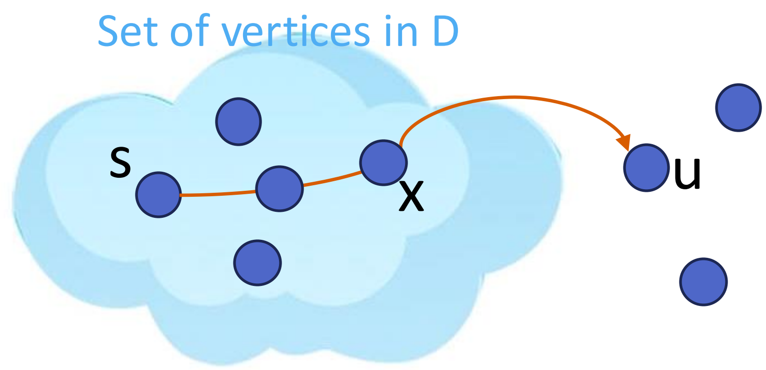
- Suppose there exist at least one vertex in V for which this property is violated.
- Assume **u** is the **first vertex** added to D with d[u] $\neq$ $\delta$(s,u)
- Note that u cannot be s (Why? d[s] = $\delta$(s,s) = 0)
- There must exist a shortest path from s to u in the graph. Call this path **P**

# Dijkstra's Single Source Shortest Path Algorithm (SSSP)

**Case 1**: All vertices on path P except u are in D

Let x be the last vertex on path P which is in D

When x was placed in D, d[x] = $\delta$(s,x) and d[u] must have been updated to $\delta$(s,u)



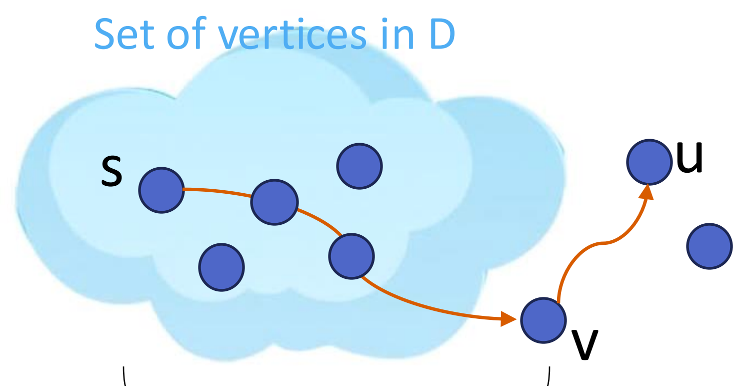

**Case 2**: Besides u, there is at least one vertex on path P that is not in D

Let v be the first vertex on path P which is not in D

d[v] = $\delta$(s,v) $\leq$ $\delta$(s,u)

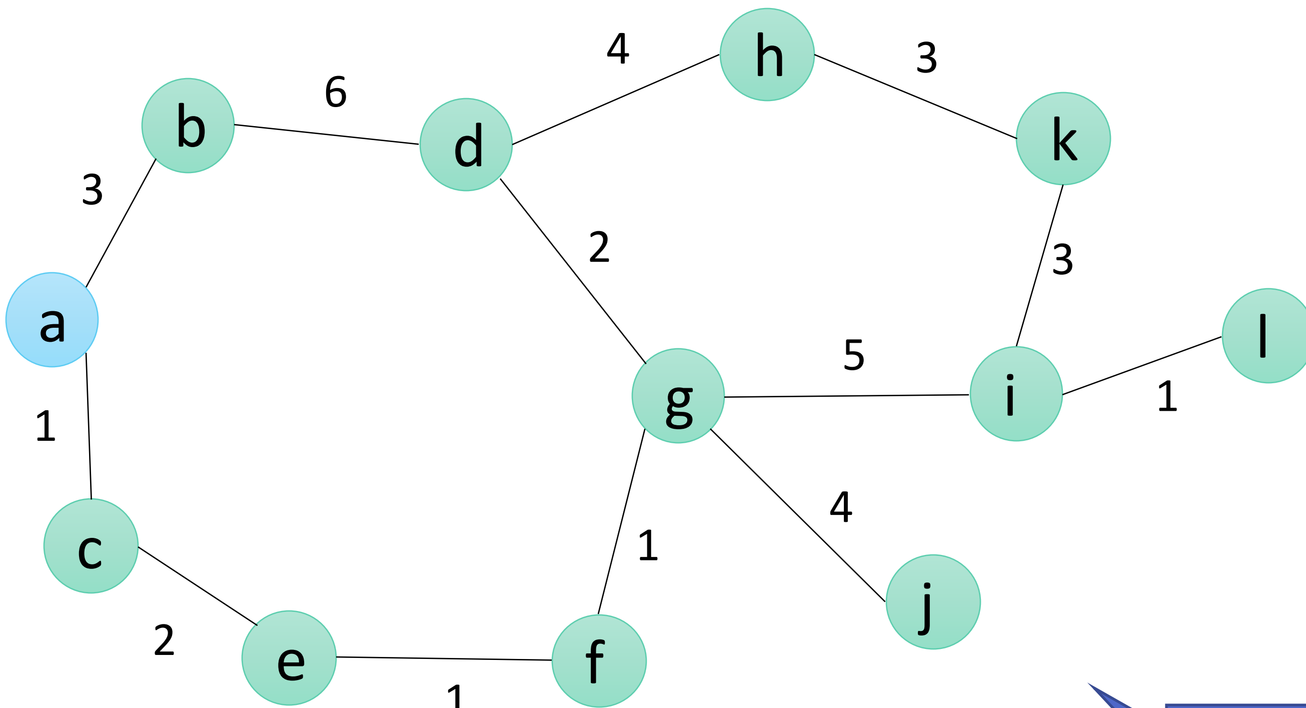
d[u] $\neq$ $\delta$(s,u) implies d[u] $>$ $\delta$(s,u)

If u was picked before v, then d[u] $\leq$ d[v] (contradiction!)



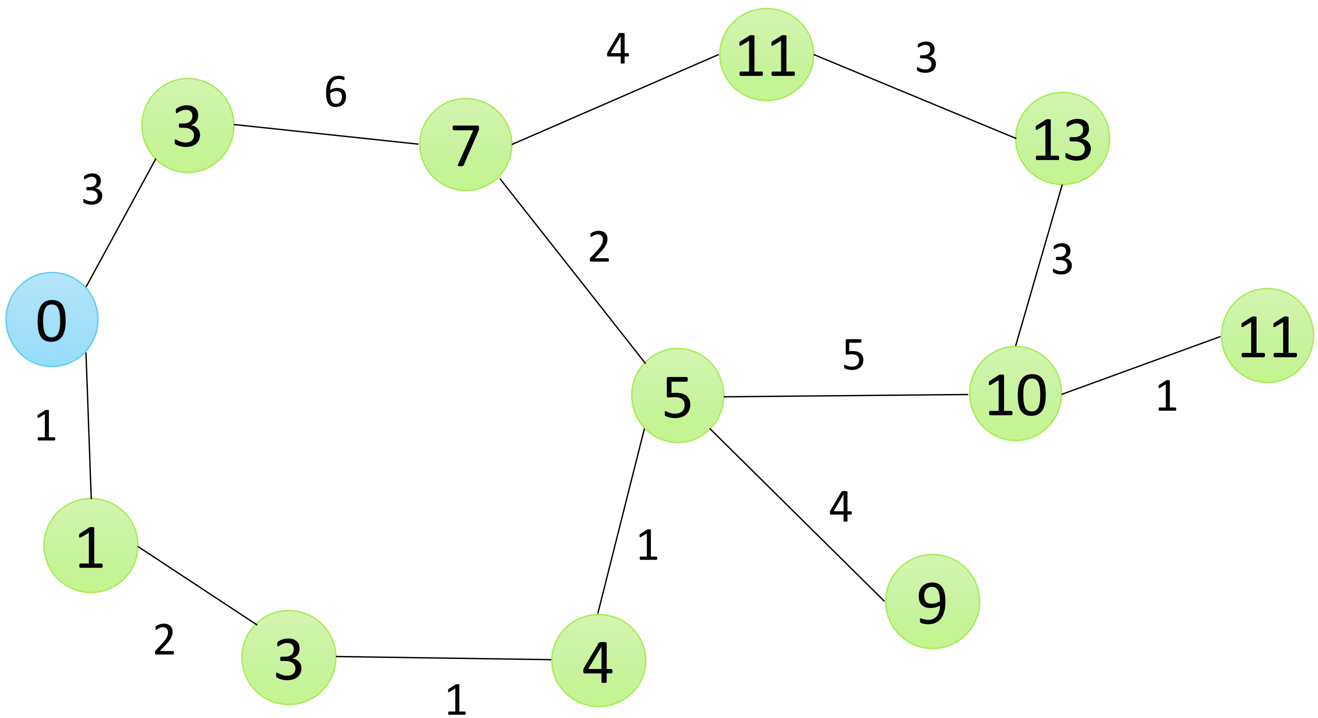

This subpath of P from s to v must be shortest path from s to v (WHY?)

# SSSP on Weighted Graphs




Work out yourself !

# SSSP on Weighted Graphs



# Complexity

- Depends on how we store set F
- Need two operations:
  - **EXTRACT-MIN** (to remove the vertex with minimum distance estimate from F)
  - **DECREASE-KEY** (to revise a distance estimate)

# Complexity

- Depends on how we store set F
- Need two operations:
  - **EXTRACT-MIN** (to remove the vertex with minimum distance estimate from F)
  - **DECREASE-KEY** (to revise a distance estimate)

We execute EXTRACT-MIN operation |V| times

We execute DECREASE-KEY operation |E| times

# Complexity

- Depends on how we store set F
- Need two operations:
  - **EXTRACT-MIN** (to remove the vertex with minimum distance estimate from F)
  - **DECREASE-KEY** (to revise a distance estimate)
- Option 1: Assume vertices are numbered from 1 to $|V|$. Use an **array** of size $|V|$, store d[u] at the $u^{th}$ entry, have a separate boolean array of size $|V|$ to indicate if u is in set F or not

# Complexity

- Depends on how we store set F
- Need two operations:
  - **EXTRACT-MIN** (to remove the vertex with minimum distance estimate from F)
  - **DECREASE-KEY** (to revise a distance estimate)
- Option 1: Assume vertices are numbered from 1 to |V|. Use an **array** of size |V|, store d[u] at the $u^{th}$ entry, have a separate boolean array of size |V| to indicate if u is in set F or not
  - EXTRACT-MIN: we linearly search the array for the smallest: O(|V|)
  - DECREASE-KEY: Each operation takes O(1) time
  - Total time: $\mathbf{O(|V|^2 + |E|)}$

# Complexity

- Depends on how we store set F
- Need two operations:
  - **EXTRACT-MIN** (to remove the vertex with minimum distance estimate from F)
  - **DECREASE-KEY** (to revise a distance estimate)
- Option 1: Assume vertices are numbered from 1 to |V|. Use an **array** of size |V|, store d[u] at the $u^{th}$ entry, have a separate boolean array of size |V| to indicate if u is in set F or not
  - EXTRACT-MIN: we linearly search the array for the smallest: O(|V|)
  - DECREASE-KEY: Each operation takes O(1) time
  - Total time: **$O(|V|^2 + |E|)$**
- Option 2: When a **min heap** (priority queue) with distance as priority key, total time is **O((|E| + |V|) log |V|)**
  - O(log |V|) to DECREASE-KEY
  - O(log |V|) to EXTRACT-MIN

# Complexity

- Depends on how we store set F
- Need two operations:
  - **EXTRACT-MIN** (to remove the vertex with minimum distance estimate from F)
  - **DECREASE-KEY** (to revise a distance estimate)
- Option 1: Assume vertices are numbered from 1 to |V|. Use an **array** of size |V|, store d[u] at the u$^{th}$ entry, have a separate boolean array of size |V| to indicate if u is in set F or not
  - EXTRACT-MIN: we linearly search the array for the smallest: O(|V|)
  - DECREASE-KEY: Each operation takes O(1) time
  - Total time: **O(|V|$^2$ + |E|)**
- Option 2: When a **min heap** (priority queue) with distance as priority key, total time is **O((|E| + |V|) log |V|)**
  - O(log |V|) to DECREASE-KEY
  - O(log |V|) to EXTRACT-MIN
- When |E| is O(**|V|$^2$** ) *[highly connected, dense graph]*, using a min heap is worse than using a linear array

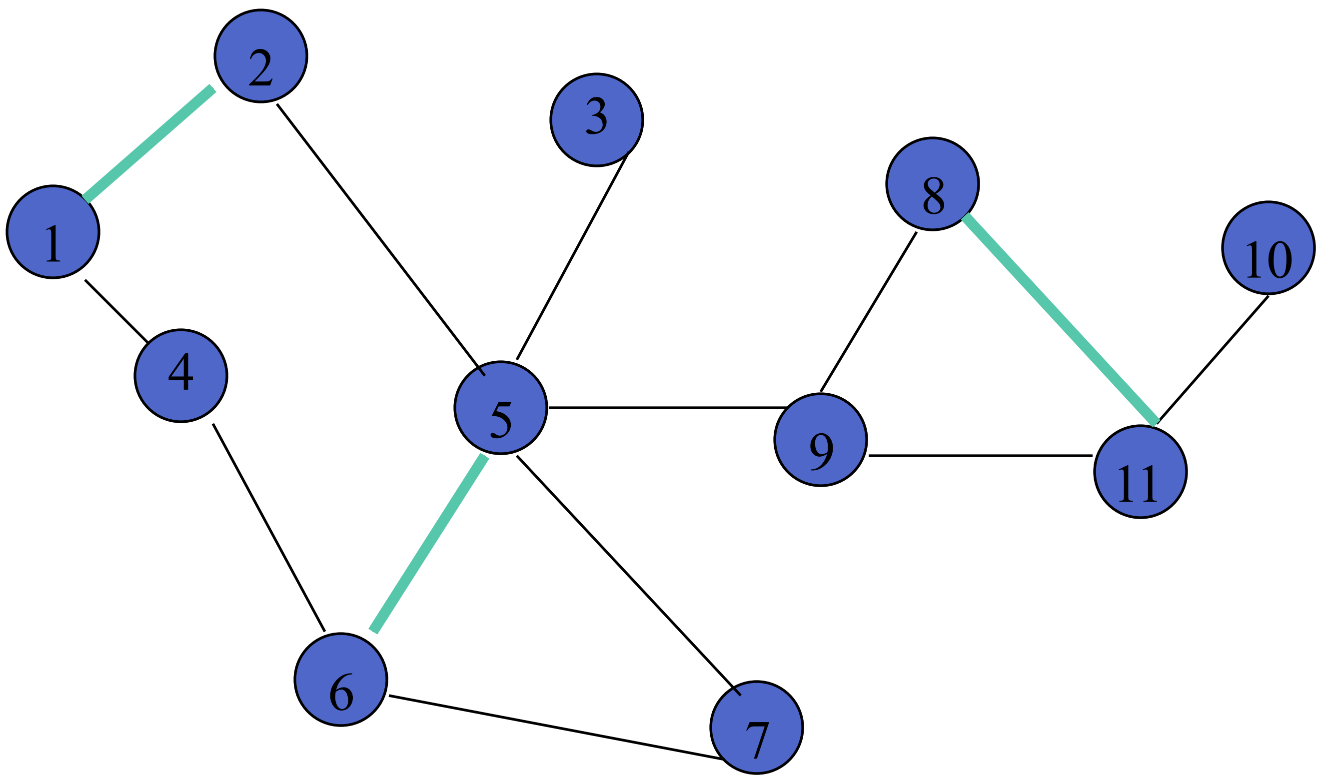
# Complexity

- Depends on how we store set F
- Need two operations:
  - **EXTRACT-MIN** (to remove the vertex with minimum distance estimate from F)
  - **DECREASE-KEY** (to revise a distance estimate)
- Option 1: Assume vertices are numbered from 1 to |V|. Use an **array** of size |V|, store d[u] at the $u^{th}$ entry, have a separate boolean array of size |V| to indicate if u is in set F or not
  - EXTRACT-MIN: we linearly search the array for the smallest: O(|V|)
  - DECREASE-KEY: Each operation takes O(1) time
  - Total time: **$O(|V|^2 + |E|)$**
- Option 2: When a **min heap** (priority queue) with distance as priority key, total time is **O((|E| + |V|) log |V|)**
  - O(log |V|) to DECREASE-KEY
  - O(log |V|) to EXTRACT-MIN
- When |E| is $O(|V|^2)$ *[highly connected, dense graph]*, using a min heap is worse than using a linear array
- When a **Fibonacci heap** is used, the total time is O(|E| + |V| log |V|)

# Minimum Spanning Tree

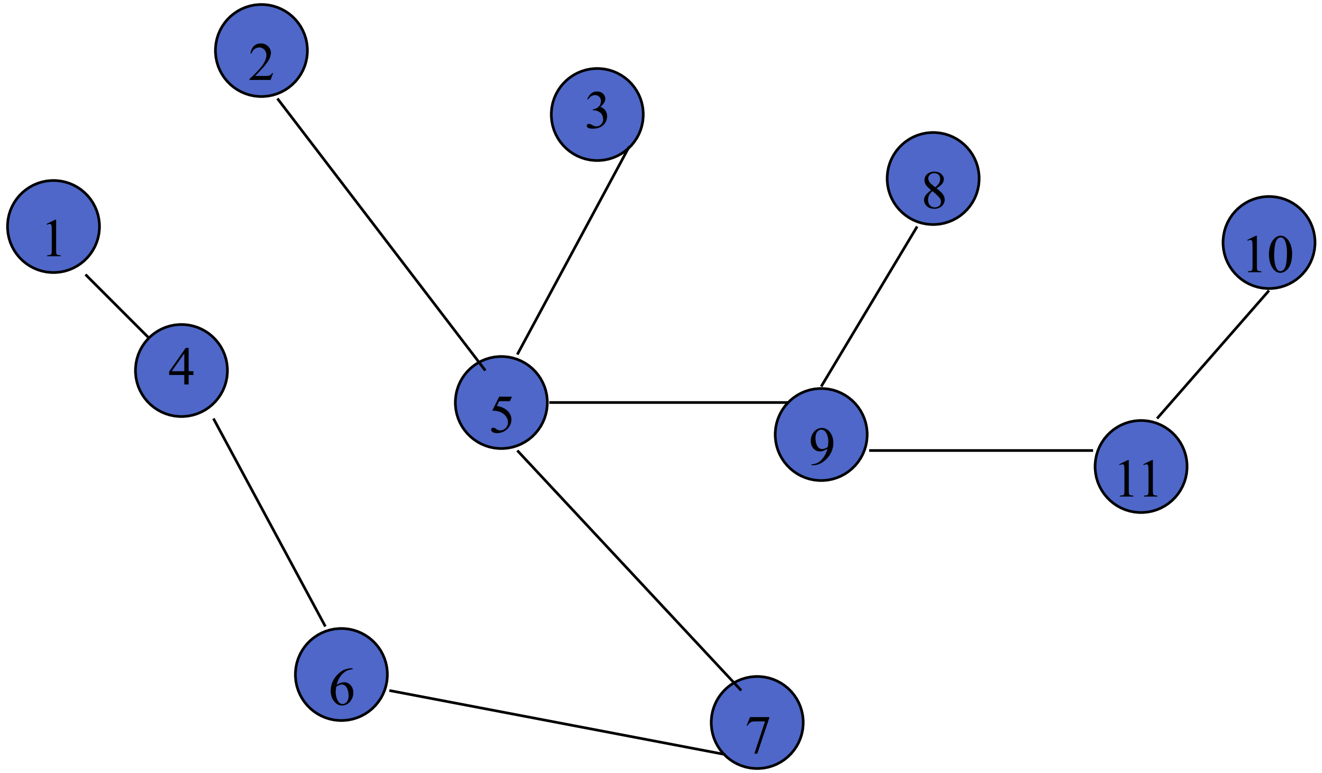
# Cycles And Connectedness

Removal of an edge that is on a cycle does not affect connectedness.



# Cycles And Connectedness

Connected subgraph with all vertices and minimum number of edges has no cycles.



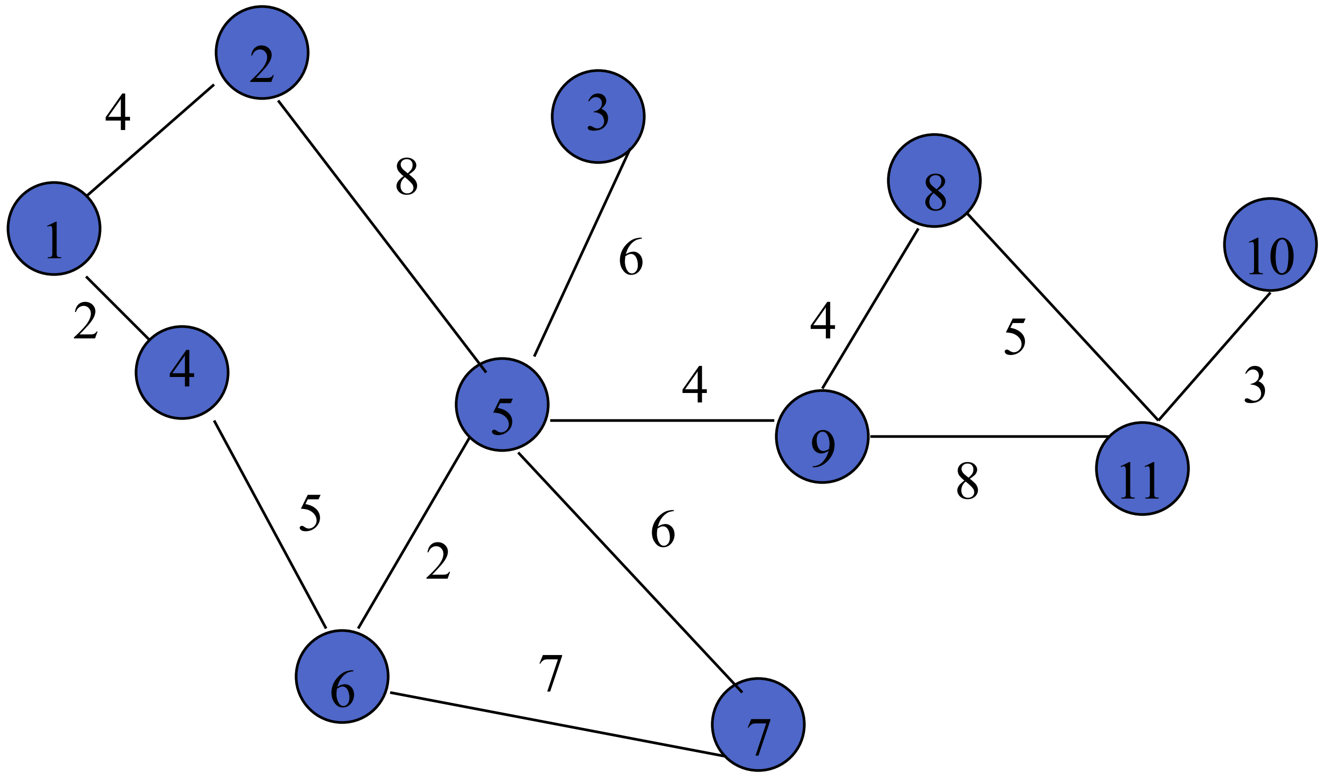
# Spanning Tree

- Communication Network Problems
  - Is the network connected?
  - Can we communicate between every pair of cities?
  - Find the components.
  - Want to construct smallest number of feasible links so that resulting network is connected.

- Subgraph that includes all vertices of the original graph.
- Subgraph is a tree.
  - If original graph has n vertices, the spanning tree has n vertices and n-1 edges.

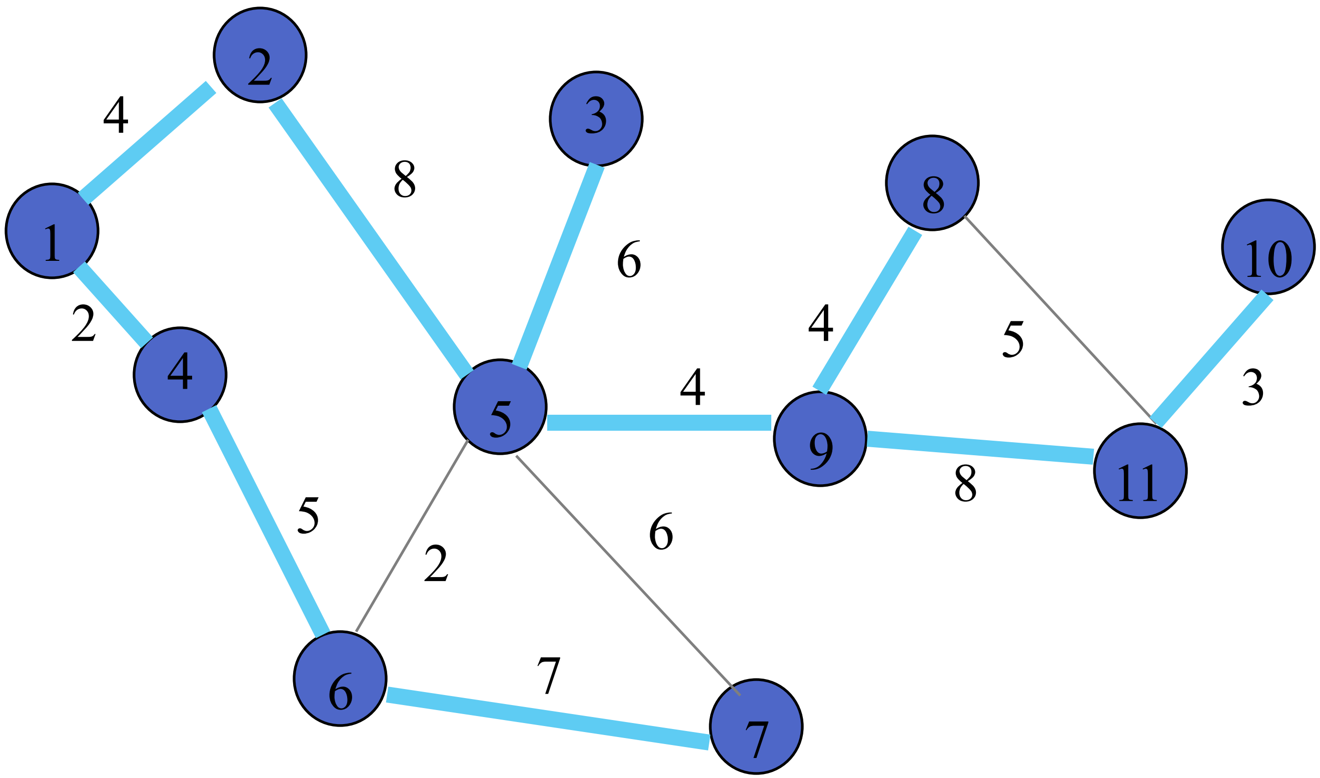
# Minimum Cost Spanning Tree

- Tree cost is sum of edge weights/costs.



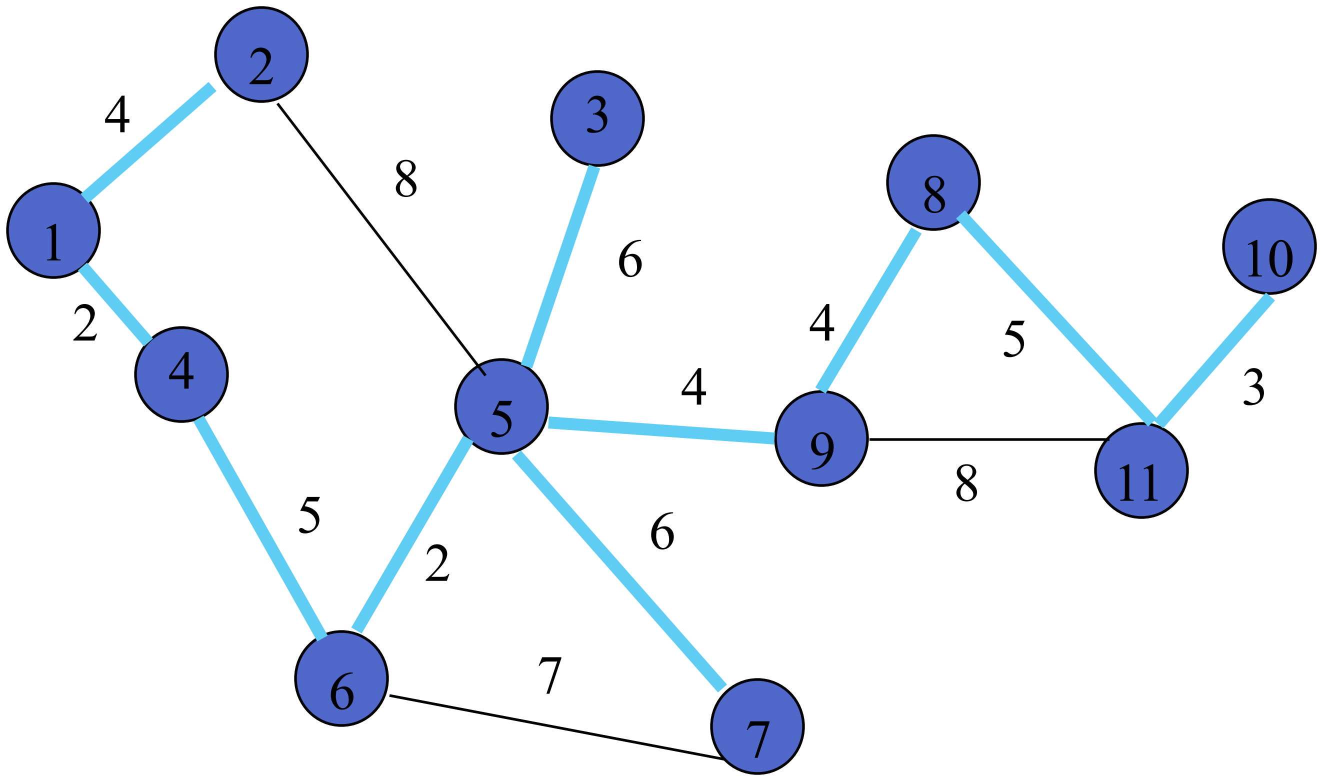
# A Spanning Tree

A Spanning tree, cost = 51



# Minimum Cost Spanning Tree

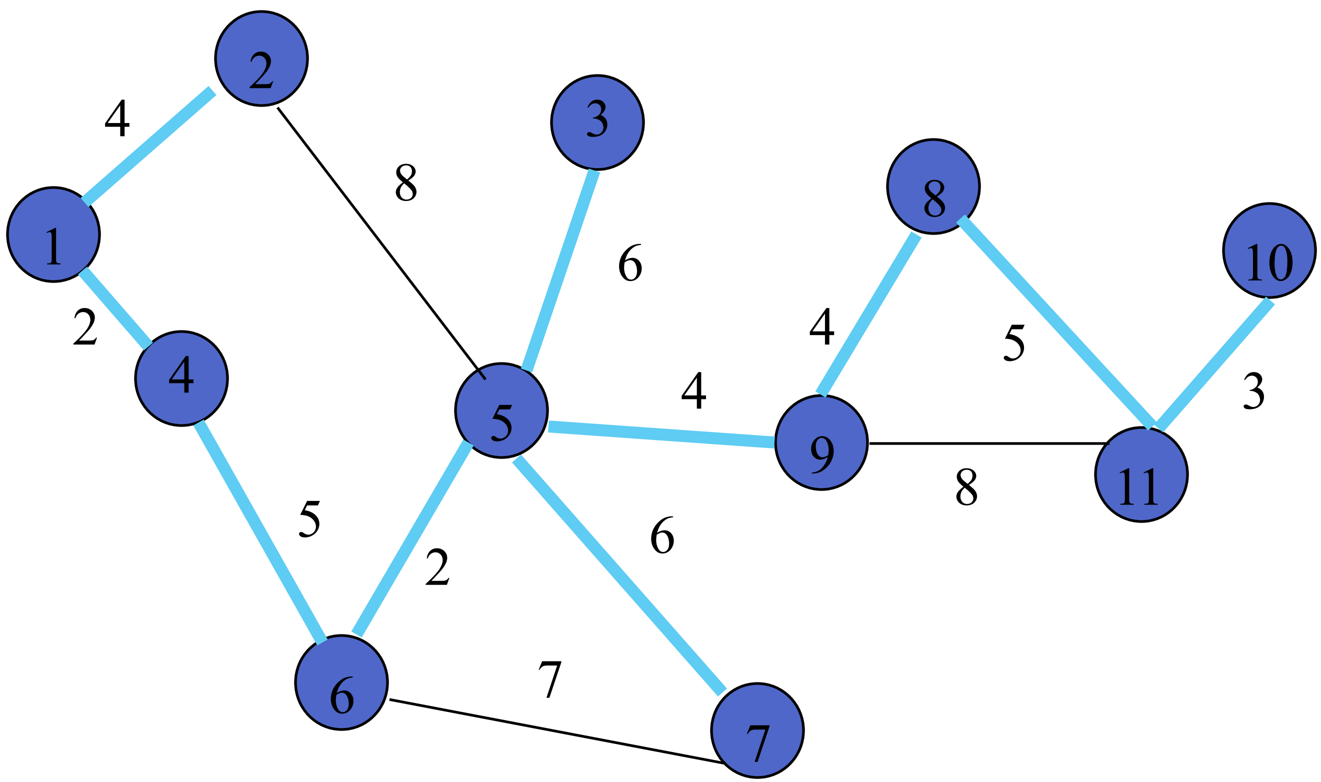
Minimum Spanning tree, cost = 41.



# A Wireless Broadcast Tree

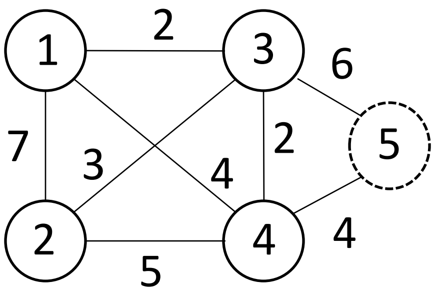
Source = 1, say edge weights = power requirement

Cost = 4 + 2 + 5 + 2 + 6 + 6 + 4 + 4 + 5 + 3 = 41



# Prim’s Minimum Spanning Tree (MST)

**Given input graph G**



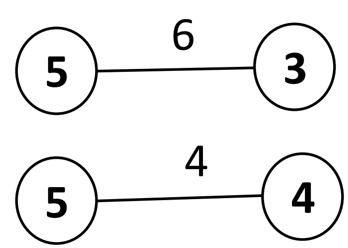

**Select a vertex randomly e.g., vertex 5**


**Initialize an empty graph T with vertex 5**

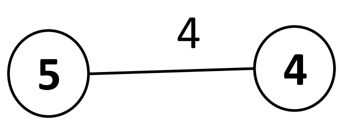
T


**Repeat until all vertices are added to T**

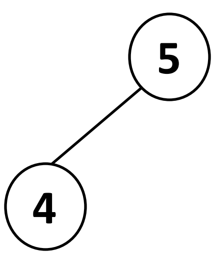
**Consider all those edges in G that connect a vertex in T to a vertex not in T**




**From such edges, select the edge with minimum weight**




**Add the edge to T**



# Tasks

- Self study
  - Read the relevant sections of CLRS “Introduction to Algorithms” textbook

# Questions?