DSC40B: Theoretical Foundations of Data Science II

Lecture 16: *Minimum Spanning Tree, properties, and general greedy algorithms*

Instructor: Yusu Wang

Previously

- Given directed or undirected graphs
  - Graph search / traversal strategies (DFS / BFS)
  - Single source shortest paths in weighted graphs
    - Bellman-Ford algorithm for general graphs
    - Dijkstra algorithm for graphs with positive edge weights

- Today:
  - Computing a minimum spanning tree (MST) of an undirected graph

First: Shortest Path Trees

- Recall: BFS-tree and DFS-tree
  - Formed by the collection of edges from a node's BFS or DFS predecessor to the node itself.
  - BFS-tree encodes shortest paths info (as well as shortest path distance) from the source node.

- Similarly:
  - Both Bellman-Ford algorithm and Dijkstra algorithm will produce a tree, which we call shortest path trees:
    - consisting of all edges of the form (u, v) where u is the predecessor of v recorded by the algorithm that gives rise to the current distance estimate at v.
      - Hence u must be a shortest-path predecessor for v.

Dijkstra using priority queue

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    priority_queue = PriorityQueue(est)
    while priority_queue:
        u = priority_queue.extract_min()
        for v in graph.neighbors(u):
            changed = update(u, v, weights, est, pred)
            if changed:
                priority_queue.change_priority(v, est[v])

    return est, pred
```

Modified Bellman-Ford with early stopping and negative cycle detection

```python
def bellman_ford(graph, weights, source):
    """Early stopping version, detects negative cycles."""
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    predecessor = {node: None for node in graph.nodes}

    for i in range(len(graph.nodes)):
        any_changes = False
        for (u, v) in graph.edges:
            changed = update(u, v, weights, est, predecessor)
            any_changes = changed or any_changes
        if not any_changes:
            break
    # this will be True if negative cycles exist
    contains_negative_cycles = any_changes
    return est, predecessor, contains_negative_cycles
```

Implementing update() in python

```python
def update(u, v, weights, est, predecessor):
    """Update edge (u,v)."""
    if est[v] > est[u] + weights(u,v):
        est[v] = est[u] + weights(u,v)
        predecessor[v] = u
        return True
    else:
        return False
```

Trees, spanning trees, and minimum spanning tree

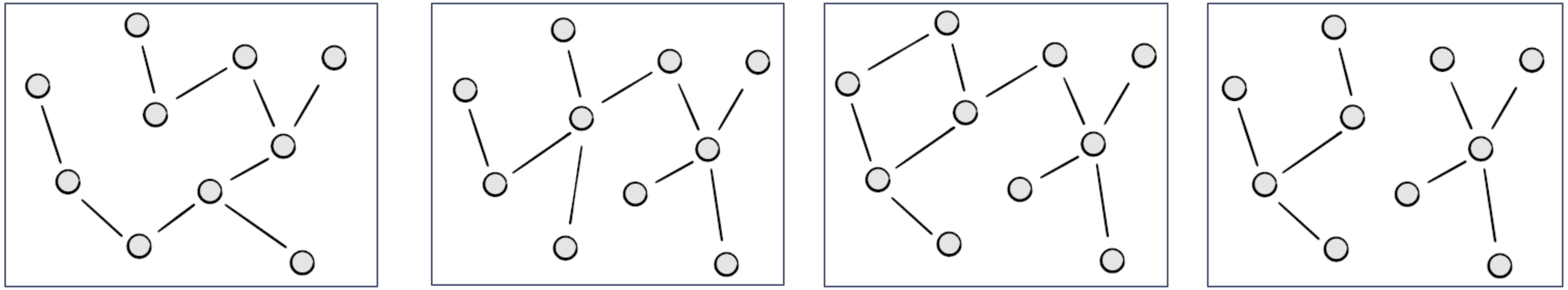
Trees

- An undirected graph $G = (V, E)$ is a tree if and only if
  - (i) it is connected; and
  - (ii) it is acyclic (i.e., does not contain any cycle)

- Claim [Tree Edges]:
  - If $T = (V, E)$ is a tree, then we have that $|E| = |V| - 1$

Alternative definition

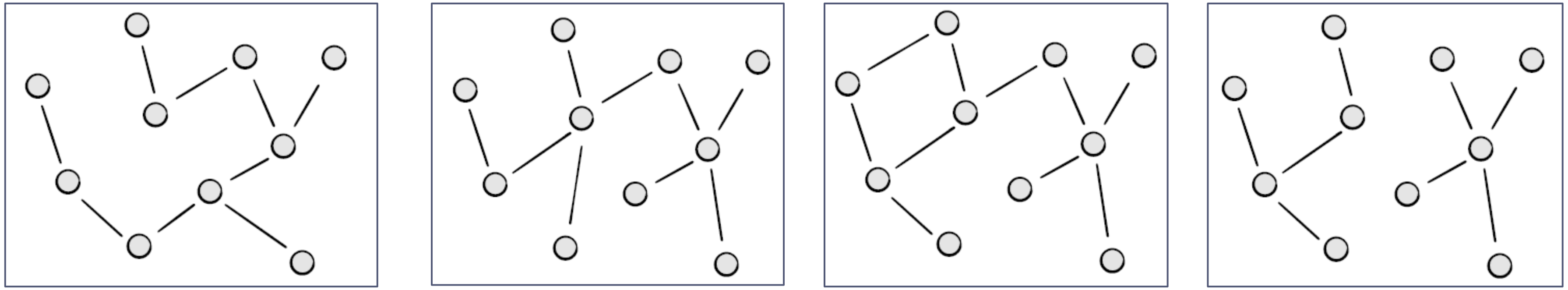
- Alternative definition:
  - An undirected graph $G = (V, E)$ is a tree if and only if that (i) it is connected; and (ii) $|E| = |V| - 1$



Alternative definition

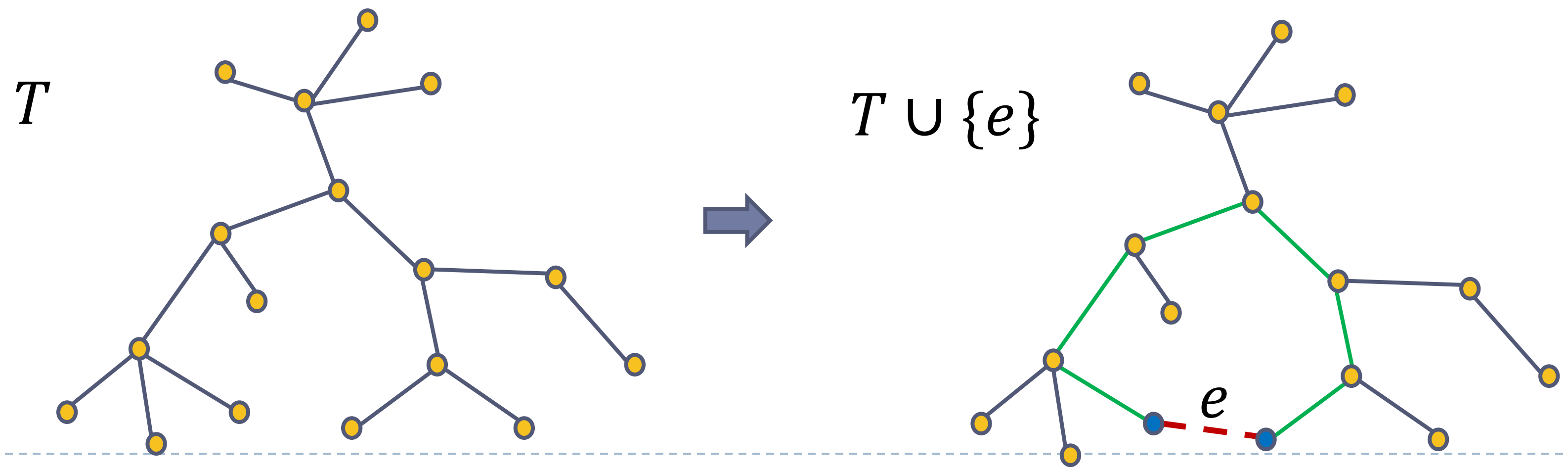
- Alternative definition:
  - An undirected graph $G = (V, E)$ is a **tree** if and only if that (i) it is connected; and (ii) $|E| = |V| - 1$

A tree | A tree | NOT a tree | NOT a tree



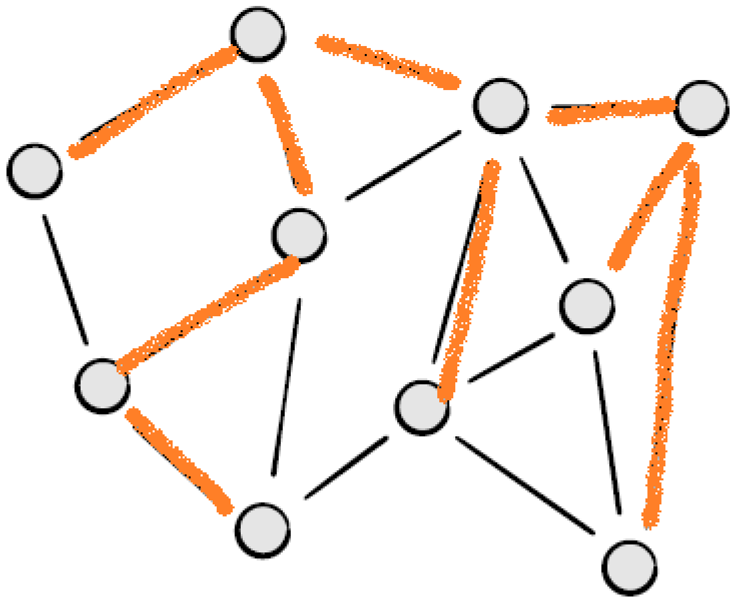
Remarks

- Key properties: If $T = (V, E)$ is a tree,
  - there is a unique path between any two nodes in V
  - adding any other edge e to T will create a unique cycle containing e
    - i.e., $T \cup \{e\}$ contains a cycle for any $e \notin T$
  - removing an edge from T will disconnect it
- Out of all connected graphs on n nodes, a tree has least number (i.e., $n-1$) of edges



Spanning Tree

- Given an undirected graph $G = (V, E)$, a spanning tree of G is any graph $T = (V, E' \subseteq E)$ that is a tree.



Example of spanning trees for the graph on the right.

Spanning Tree

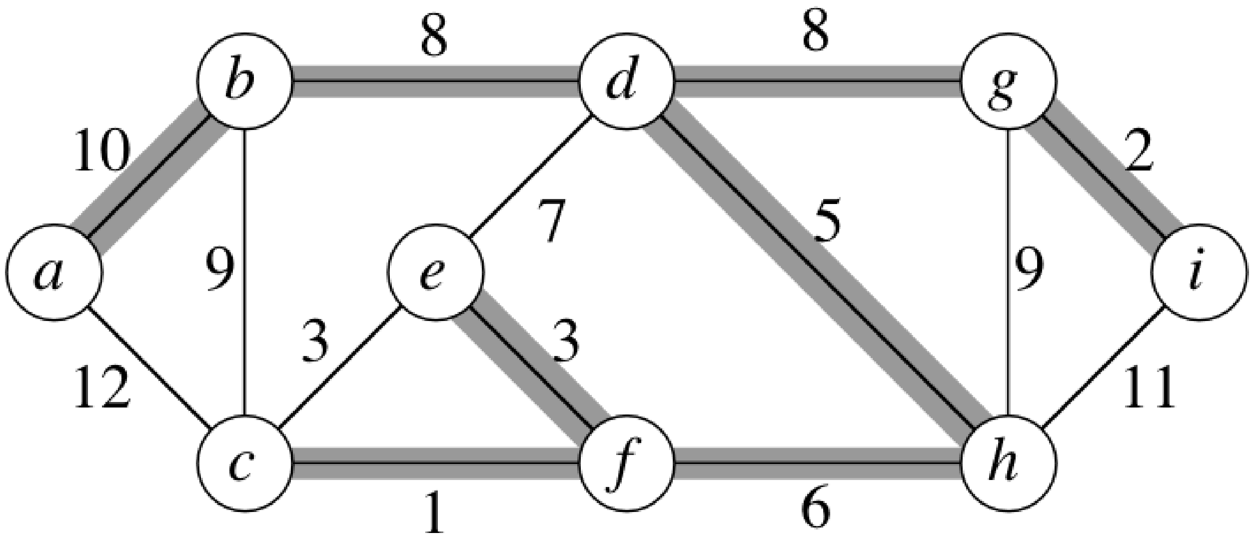
- Given an undirected graph $G = (V, E)$, a spanning tree of G is any graph $T = (V, E' \subseteq E)$ that is a tree.

- Intuitively, a spanning tree of G contains smallest number of edges in E to connect all nodes in G.

- Note that if the input graph G is not connected, then there exists no spanning tree.
  - We can talk about spanning forest, consisting a set of spanning trees, one for each connected component in G.

Minimum spanning tree (MST)

- Weight of spanning tree T of a weighted graph $G = (V, E)$ is
  - the total weights of all edges in T, i.e., $\omega(T) = \sum_{e \in T} \omega(e)$,
  - where $\omega: E \to R$ is the edge weights associated to G.

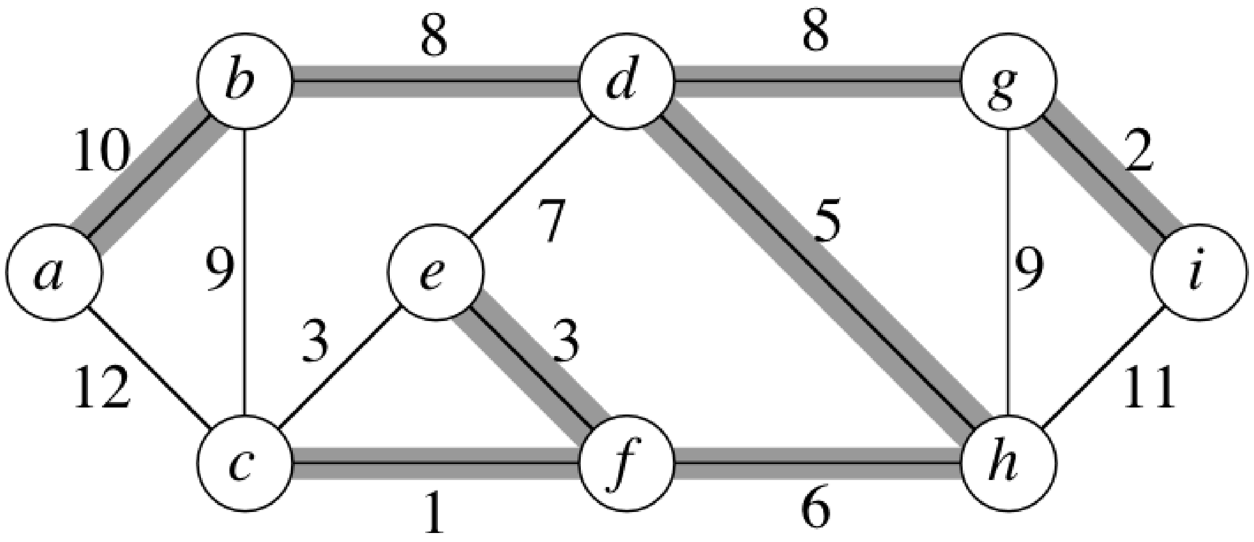


Weight of this spanning tree: 43

Minimum spanning tree (MST)

- Weight of spanning tree T of a weighted graph $G = (V, E)$ is
  - the total weights of all edges in T, i.e., $\omega(T) = \sum_{e \in T} \omega(e)$,
  - where $\omega: E \to R$ is the edge weights associated to G.

- A minimum spanning tree (MST) of a weighted graph $G = (V, E)$ is a spanning tree with smallest possible weight.



Weight of this spanning tree: 43

Turns out this is also a minimum spanning tree.

MSTs

- MST may not be unique
- All MSTs of a given graph $G = (V, E)$ have the same number of edges!
  - They all have $|V| - 1$ number of edges
- If all edges in input graph have the same weight, then how can we find a MST for it?
  - Any spanning tree of it is a minimum spanning tree!

Exercise:

Design an efficient algorithm to compute an MST for a graph where all edges have weight 1

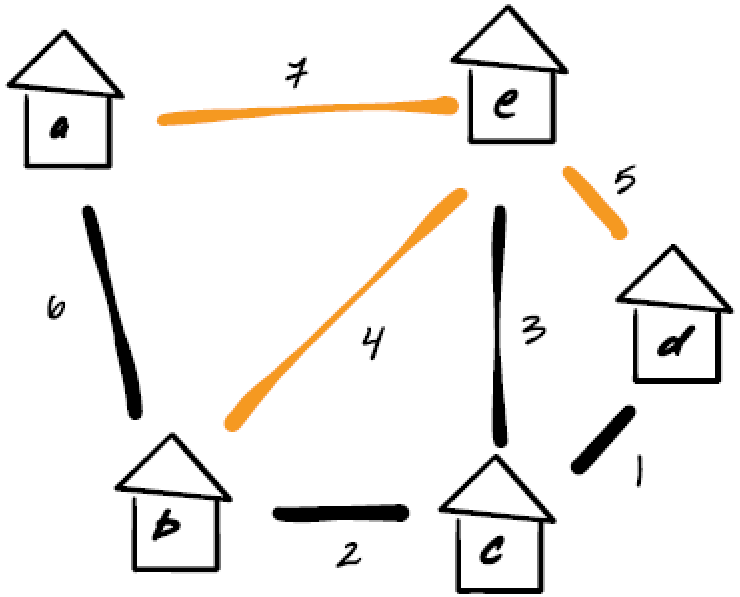
Exercise:

Design an efficient algorithm to compute an MST for a graph where all edges have weight 1

This can be done in time $\Theta(|V| + |E|)$!

Motivation and properties of MST

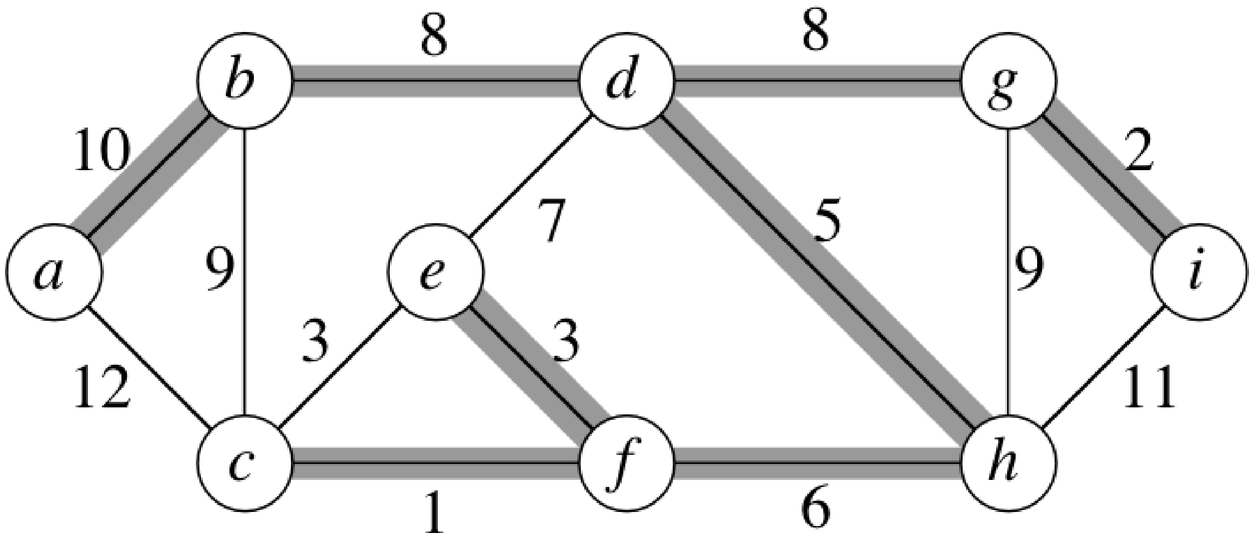
Motivation



- Among all possible road segment choices, build a set of road segments so that all houses are connected and the total cost is minimized
- Solution: Find the MST of the input weighted graph where edge weight represents the cost of build that road segment.

MST Problem

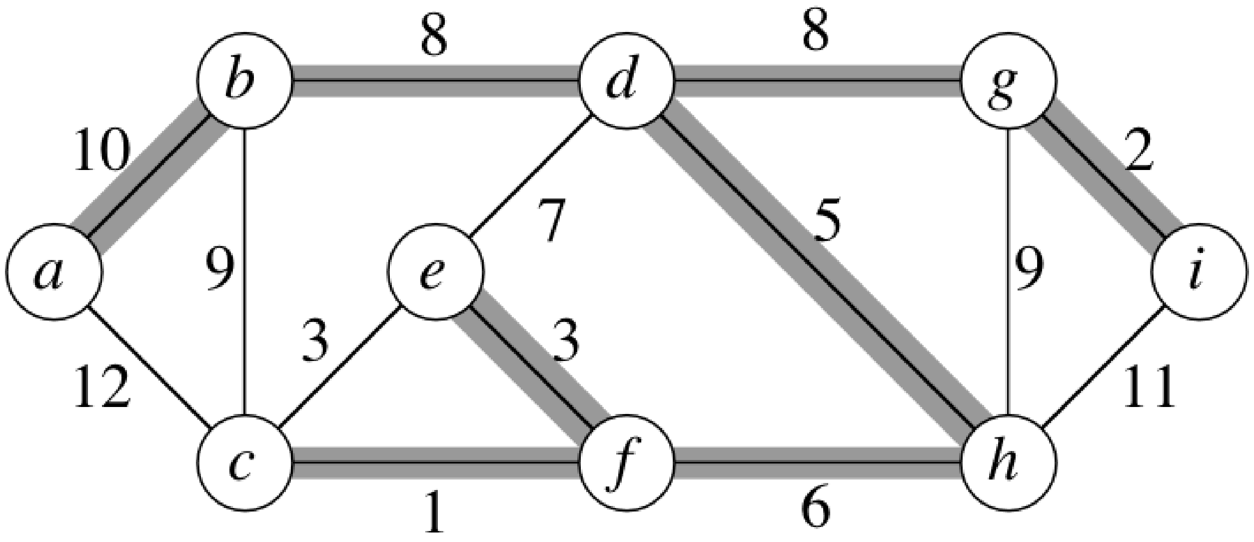
- Input:
  - a weighted undirected graph G
- Output:
  - the set of edges in a MST of G



Key property

Key property of MST:

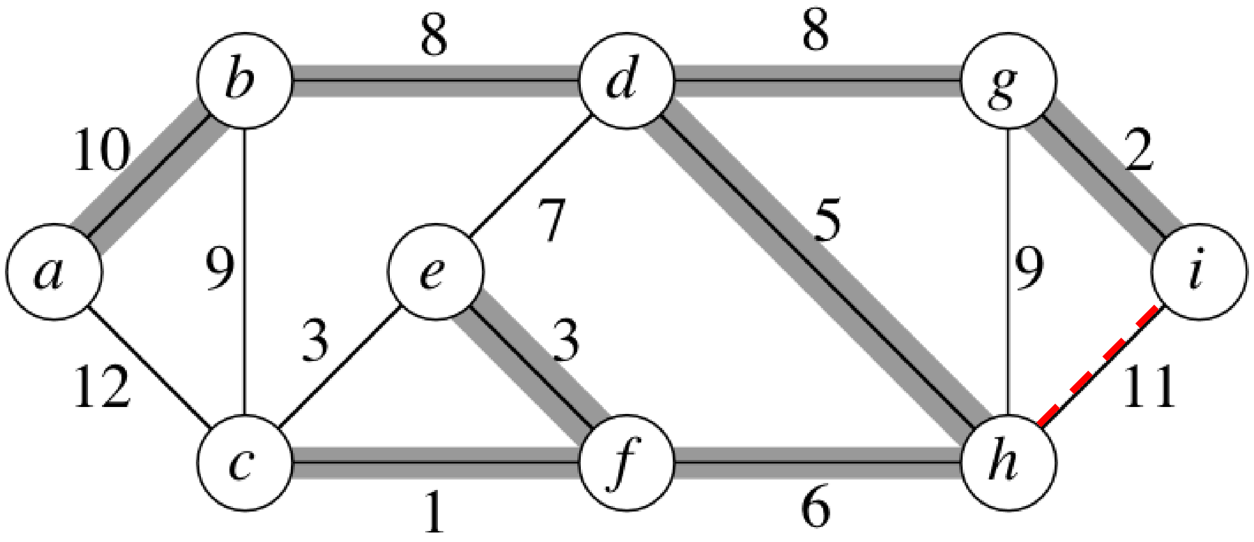
- Given a MST T of $G = (V, E)$, let $e \in E$ be any edge in E but not in T. The following then holds:
  - there is a unique cycle C containing e in $T \cup \{e\}$.
  - e has the largest weight among all edges in this cycle C.



Key property

Key property of MST:

- Given a MST T of $G = (V, E)$, let $e \in E$ be any edge in E but not in T. The following then holds:
  - there is a unique cycle C containing e in $T \cup \{e\}$.
  - e has the largest weight among all edges in this cycle C.



Key property

Key property of MST:

- Given a MST T of $G = (V, E)$, let $e \in E$ be any edge in E but not in T. The following then holds:
  - there is a unique cycle C containing e in $T \cup \{e\}$.
  - e has the largest weight among all edges in this cycle C.

- Proof sketch:
  - If e does not have largest weight, let $e' \in C$ be an edge with largest weight in C.
    - $T' = T \ - \{e'\} + \{e\}$ is also a spanning tree of G
    - $weight(T') \leq weight(T) \Rightarrow T$ cannot be MST.
    - Contradiction $\Rightarrow e$ must have largest weight in C.

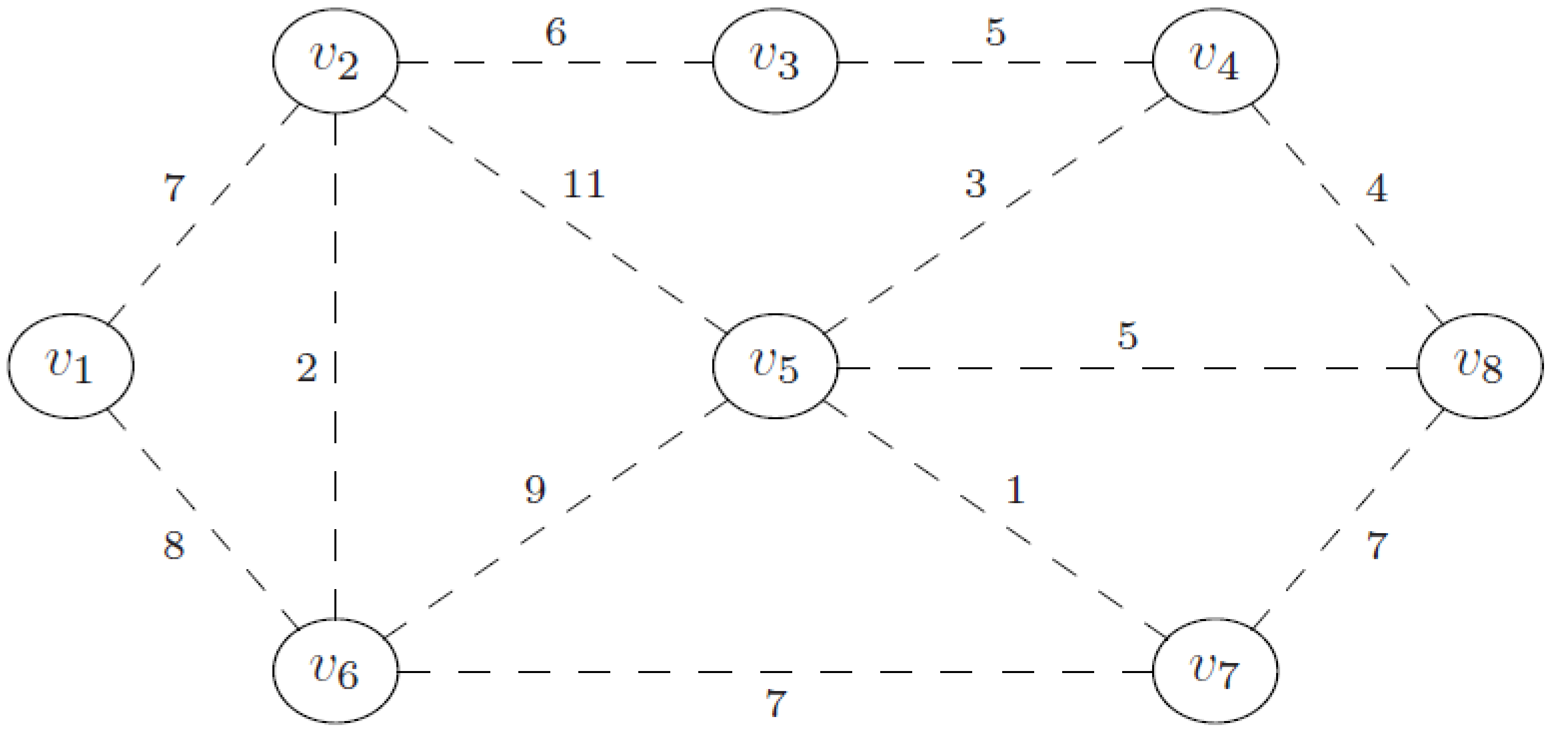
First greedy algorithm for MST: Prim’s algorithm

General greedy idea:

- Input:
  - a weighted undirected graph $G = (V, E)$, with $\omega: E \rightarrow R$
- Output:
  - the set of edges in a MST T of G

- A MST T consist of $V - 1$ number of edges that connect all nodes, with no cycle.
- Intuitively, we will grow the tree edge-by-edge, and choose "safe" edges greedily to incrementally build T
  - such that any time, the edges we choose will form a part of some MST

Example



- What is a "safe" edge to add first?

Two greedy algorithms

- Two greedy algorithms
  - Today: Prim's algorithm
  - Next class: Kruskal's algorithm
  - They differ in the order of edges they visit and thus ``safe" edges they add

Idea for Prim's algorithm

- Input:
  - a weighted undirected graph $G = (V, E)$, with $\omega: E \to R$
- Output:
  - the set of edges in a MST T of G

- Intuitively,
  - Incrementally grow a partial tree $T(S) \subseteq E$ connecting a subset of nodes $S \subset V$
  - At the beginning of each iteration, $T(S)$ is a sub-tree of some MST of G
  - At each iteration, grow $T\ (S')$ to include one more vertex $S' = S \cup \{u\}$
    - such that $T(S')$ is still a sub-tree of some MST of G
    - the new node is reached via a greedy choice of a crossing-edge
    - in particular, the greedy choice is the minimum weight edge connect some node in S to some node in $\mathrm{U} = V - S$ (i.e., outside S)

High level outline (not code)

```
procedure PrimMST(G)
U ← V(G) − {v1} ;   /* V(G) = set of vertices of graph G */
v1.predecessor ← NULL;
while (U ≠ ∅) and (∃ edge from (V(G) − U) to U) do
    (vi, vj) ← minimum weight edge from V(G) − U to U;
    vj.predecessor ← vi;
    U ← U − {vj};
end
```

- U : unconnected vertices
- $S = V - U$: vertices connected by current partial tree

High level outline (not code)

procedure PrimMST(G)

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

v_1.predecessor $\leftarrow$ **NULL**;

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

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

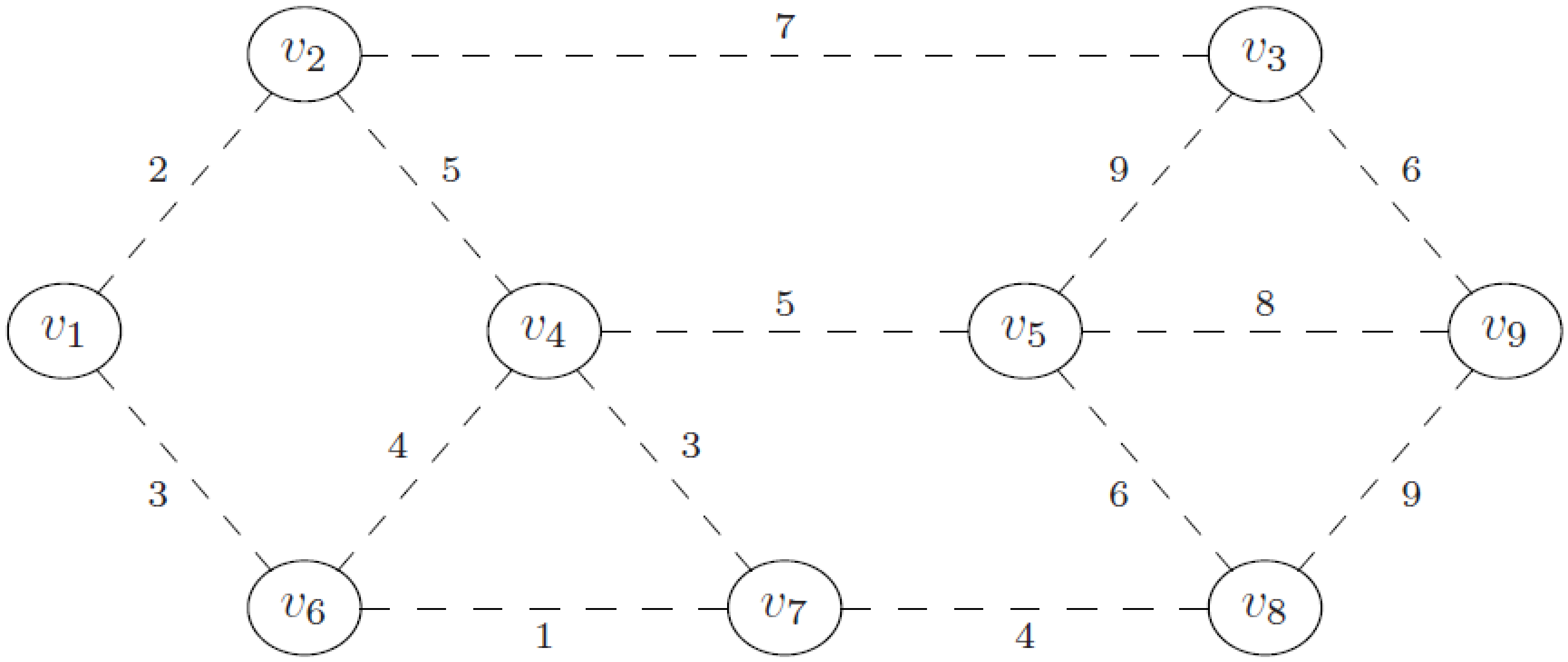
  v_j.predecessor $\leftarrow v_i$;

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

end

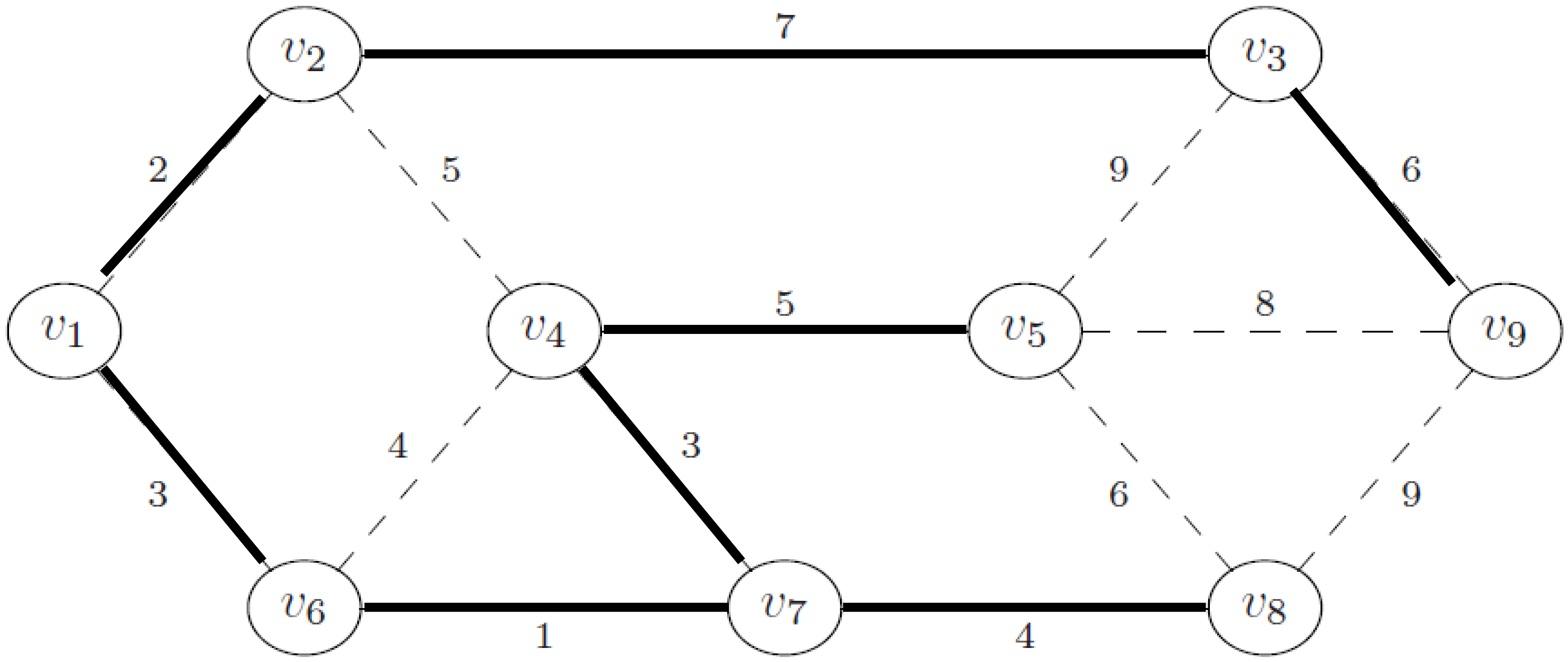
- U : unconnected vertices
- $S = V - U$: vertices connected by current partial tree

Example



- Suppose we grow the tree starting from v_1

Example



- Suppose we grow the tree starting from v_1

Correctness

MST Theorem:
Let T be a sub-tree 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.

- Key to the correctness of PrimMST algorithm.
  - *Loop invariant*:

    each time PrimMST() algorithm grows the partial tree (i.e, adds another edge to it), the invariant is that the new tree is still a subtree of *some* minimum spanning tree of input graph *G*.
  - *Termination*:

    when all nodes are connected, we obtain a MST of *G*.

    (or if we cannot reach all nodes, then the input graph is not connected)

Idea for proving loop invariant

Idea for proving loop invariant

- By the theorem's hypothesis, *T* is a subtree of some MST *A* of *G*.
- If e is not an edge of *A*, then $A \cup \{e\}$ contains a cycle.
- Let C be this cycle. There must exists some edge $e' \in C$ from *T(S)* to a vertex not in *S (those vertices already connected)*.
- Since e is a minimum weight edge from vertices in *T* to vertices not in *T*, $weight(e) \leq weight(e')$.
- Replacing $e' \in A$ by e gives a new tree $B = A - \{e'\} + \{e\}$ such that $weight(B) \leq weight(A)$.
- $T \cup \{e\} \subseteq B$. So $T \cup \{e\}$ is also a subtree of some MST.
- Done.

Implementation of Prim's algorithm

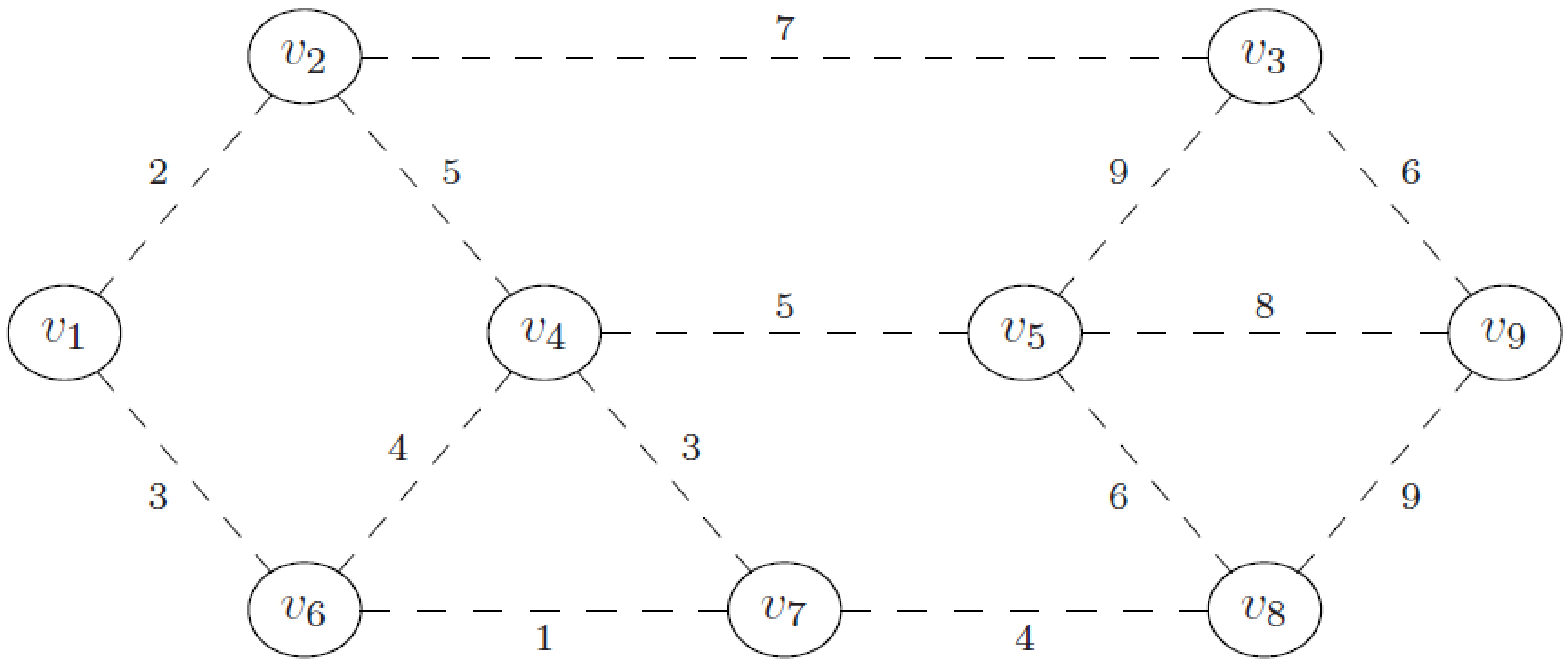
Naïve implementation of Prim's Alg

```
procedure PrimMST(G)
U ← V(G) − {v1} ;    /* V(G) = set of vertices of graph G */
v1.predecessor ← NULL;
while (U ≠ ∅) and (∃ edge from (V(G) − U) to U) do
    (vi, vj) ← minimum weight edge from V(G) − U to U;
    vj.predecessor ← vi;
    U ← U − {vj};
end
```

- Naïve implementation: linear scan all edges to identify min-weight edge (v_i, v_j) at each iteration
- Total time complexity: $O(VE)$

First improvement

- Storing costs at nodes
  - Each unvisted nodes v in U maintain $v.cost$, which is the smallest weight of any edge from v to visited nodes in S



Outline of first improvement

```
procedure PrimMST(G)
U ← V(G) ;                          /* V(G) = set of vertices of graph G */
foreach v_i ∈ V(G) − {v_1} do  v_i.cost ← ∞;
v_1.cost ← 0;
v_1.predecessor ← NULL;
while (U ≠ ∅) do
    v_j ← v_i ∈ U  with minimum v_i.cost;
    U ← U − {v_j} ;                               /* Remove v_j from U */
    /* (v_j, v_j.predecessor ) is an MST edge                          */
    foreach edge (v_j, v_k) incident on v_j do
        if (v_k is in U and weight(v_j, v_k) < v_k.cost) then
            v_k.predecessor ← v_j;
            v_k.cost ← weight(v_j, v_k);
        end
    end
end
```

- If we use linear scan to find the outside node with minimum cost for Line 6 in the algorithm in previous slide, then the entire algorithm taks $O(V^2)$ time.
  - Line 6 takes $O(V)$ time
  - Lines 8-13 takes $\Theta\left(\deg\left(v_j\right)\right) = O(V)$
  - Hence each iteration of the while-loop takes $O(V)$ time
  - The while-loop runs V iterations
  - Hence total time complexity is $O(V^2)$

Better implementation

- Similar to Dijkstra algorithm, we can use priority-queue to significantly speed up the time complexity!

- In particular, we need a data structure to maintain the costs of unvisited nodes, which supports:
  - deleting the node with minimum cost (.extract_min !)
  - update (decrease) the cost value stored at a node (change_priority !)

- In our case, a priority queue stores (key, value) pairs, where key refers to identity of some node, while value is the cost of this node.

Recall Heap implementation

- A priority queue can be implemented using a (min) heap
- min-heap implementation of priority queue:
  - PriorityQueue(priorities): takes $\Theta(n)$ time for $n = |\text{priorities}|$
  - .extract_min() : takes $\Theta(\log n)$ time where n is the size of priority queue
  - .change_priority(key, value) : takes $\Theta(\log n)$ time where n is the size of priority queue

Final implementation of Prim's Alg

```
def prim(graph, weight):
    tree = UndirectedGraph()

    estimated_predecessor = {node: None for node in graph.nodes}
    cost = {node: float('inf') for node in graph.nodes}
    priority_queue = PriorityQueue(cost)

    while priority_queue:
        u = priority_queue.extract_min()
        if estimated_predecessor[u] is not None:
            tree.add_edge(estimated_predecessor[u], u)
        for v in graph.neighbors(u):
            if weight(u, v) < cost[v] and v not in tree.nodes:
                priority_queue.decrease_priority(v, weight(u, v))
                cost[v] = weight(u, v)
                estimated_predecessor[v] = u
    return tree
```

Time complexity analysis

- We use min-heap to implement the priority queue
- The maximum size of the priority-queue is *V*
- # iterations of While-loop?
  - *V*
- # iterations of each call of the inner for-loop?
  - $\deg(v_j)$
- Total #times lines 7—10 are executed:
  - $\sum_{v_j \in V} \deg(v_j) = 2E$

Time complexity analysis

- We use min-heap to implement the priority queue
- The maximum size of the priority-queue is ____
- # iterations of While-loop?
  -
- # iterations of each call of the inner for-loop?
  -
- Total #times lines 7—10 are executed:
  -

Time complexity analysis

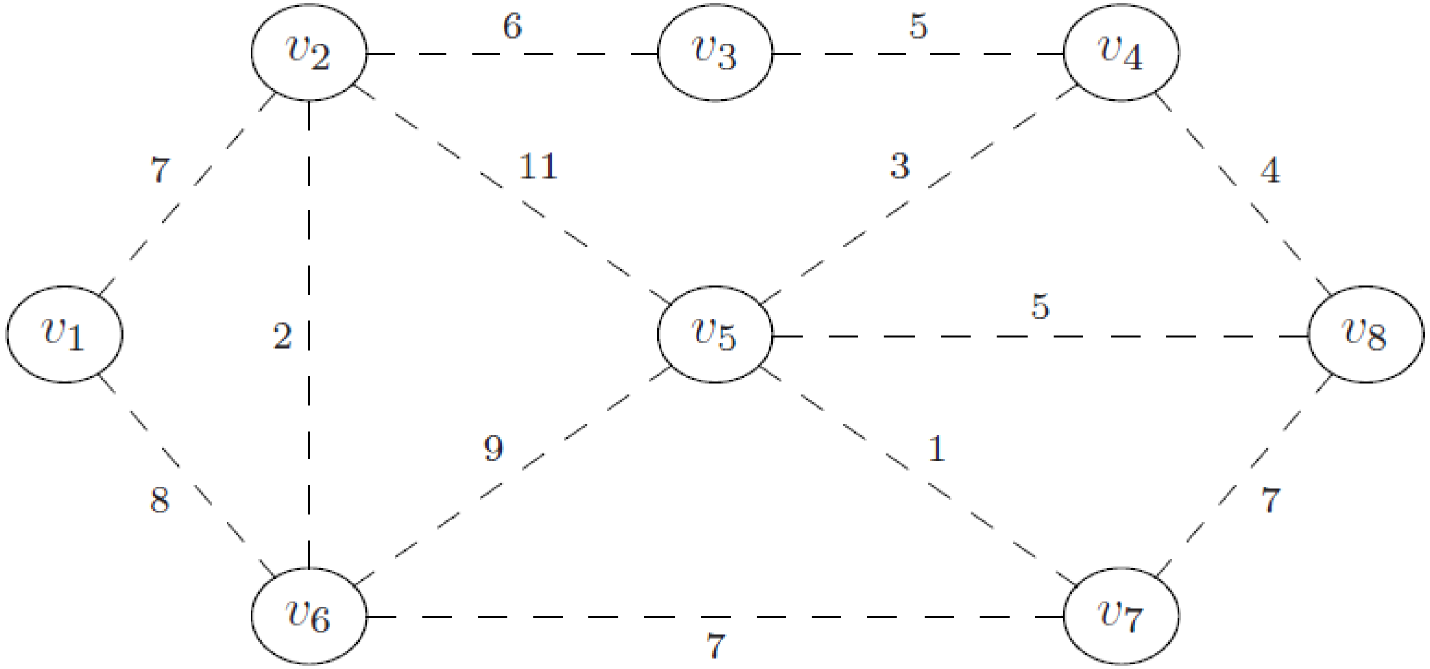
- We use min-heap to implement the priority queue
- The maximum size of the priority-queue is *V*
- # iterations of While-loop?
  - *V*
- # iterations of each call of the inner for-loop?
  - $\deg(v_j)$
- Total #times lines 7—10 are executed:
  - $\sum_{v_j \in V} \deg(v_j) \ = 2\mathrm{E}$

- Initialize priority_queue
  - Total cost:
- extract_min
  - Total #:
  - Total cost:
- decrease_priority
  - Total #:
  - Total cost:

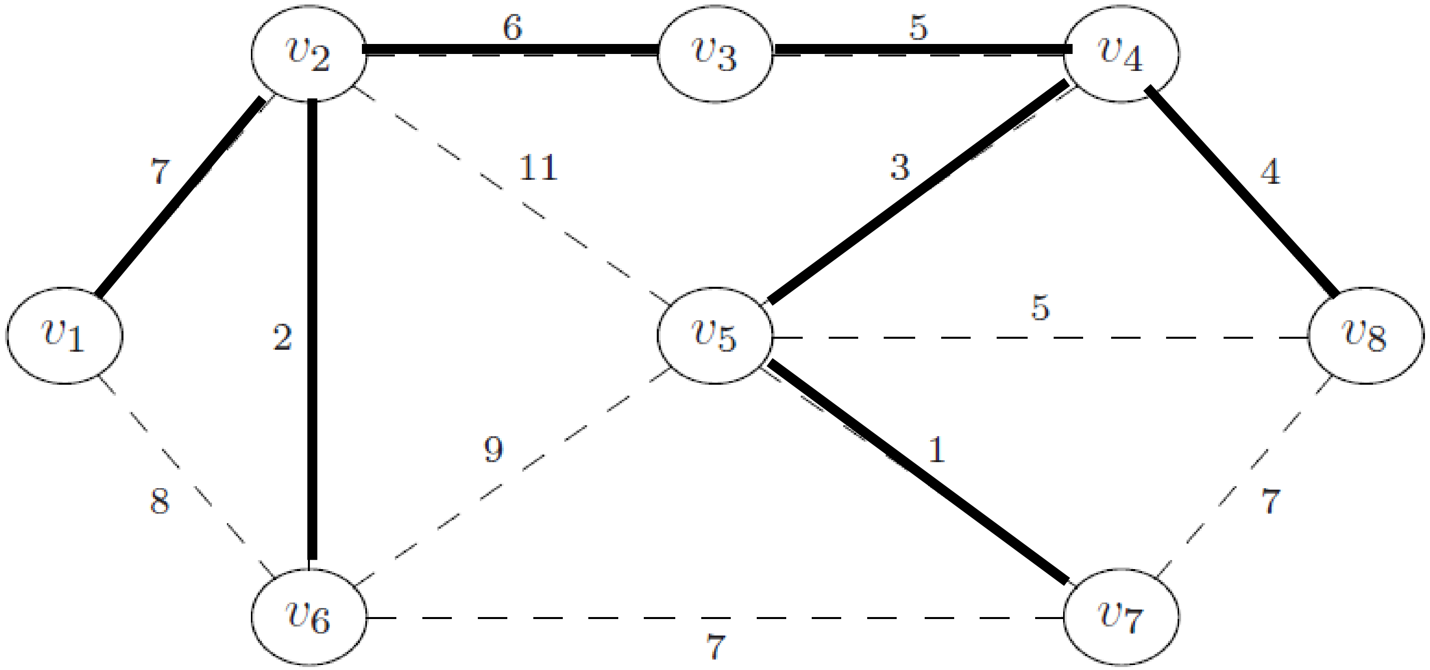
- Initialize priority_queue
  - Total cost: $\Theta(V)$
- extract_min
  - Total #:V
  - Total cost: $\Theta(V \lg V)$
- decrease_priority
  - Total #: at most 2E
  - Total cost: $O(E \lg V)$

Total time complexity:
$\Theta\big((V+E)\lg V\big)$

Example



Example

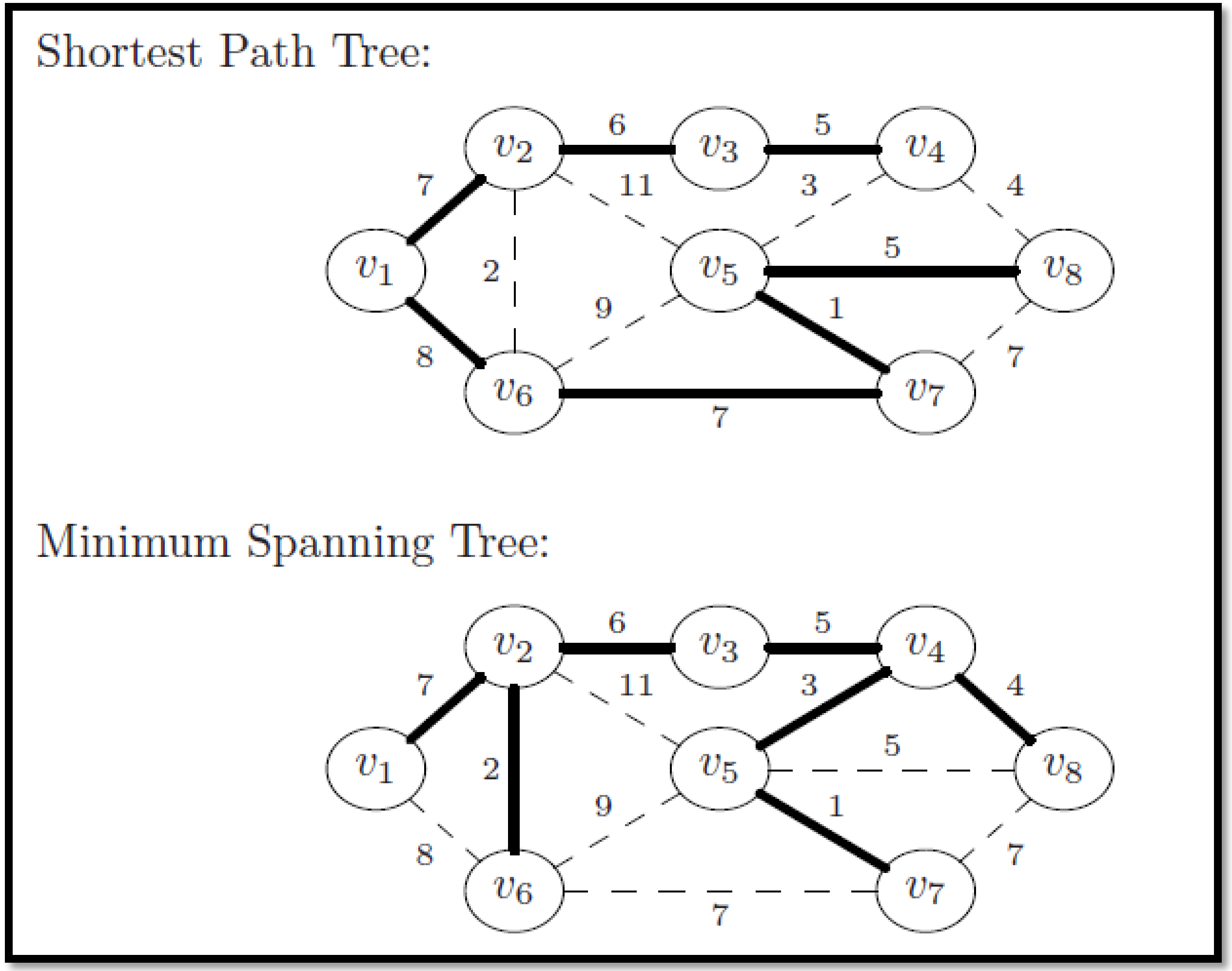


Comparison with Dijkstra algorithm

- Dijkstra:
  - Each node maintains the best distance estimate from source to the current node
    - when inspecting a new (crossing) edge (u, v),
      - $v.distance \; = \; \min(v.distnace, \quad u.distance + \; weight(u, v))$

- Prim's:
  - Each node (not yet visited) maintains the minimum weight of any edge to reach a visited-node.
    - when inspecting a new crossing edge (u, v),
      - $v.cost \; = \; \min(v.cost, weight(u, v))$

Comparison with Dijkstra



Summary and remarks

- Prim's algorithm:
  - A greedy algorithm which repeatedly choose the minimum-weight edge to reach an unvisited node
  - Share similarity to Dijkstra algorithm
  - Runs in $\Theta((V+E)\lg V)$ time using min-heap
- Similar to Dijkstra algorithm, we can further improve the time complexity to $\Theta(E + V \lg V)$ using Fibonacci heap, which is a more efficient implementation of priority queue.
- Next time,
  - Another greedy algorithm, called Kruskal algorithm, which has other properties too.

FIN