COMP3251
Lecture 11: Prim's Algorithm
(Chapter 5.1)

Recap of Graph Algorithms

DFS is much like walking in a maze:

- **Basic exploration step:** When reaching some vertex u, pick an adjacent vertex v and (recursively) explore v.
- If we hit a dead end, backtrack.
- Mark visited vertices and do not re-visit them.
- **Applications:** detecting cycles, topological ordering, finding strongly connected components

BFS is like expanding water front:

- We first visit all vertices that are directly adjacent to the root s, then all vertices that have distance 2 from s, etc.
- **Applications:** single-source shortest path problem when all edges have length 1.

Recap of Graph Algorithms

Dijkstra solves the single-source shortest path problem with **non-negative edge lengths**:

- Similar to BFS, it explore vertices in ascending order of their distance from the root vertex *s*.
- Dijkstra's greedy rule along with a non-trivial priority heap implementation allows us to do it in nearly linear time.

Bellman-Ford solves the single-source shortest path problem with arbitrary edge lengths and **without negative cycles**.

Greedy Algorithms

Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.

Example 1: Finish the homework assignment with the closest deadline first.

Example 2: Dijkstra's algorithm is a greedy algorithm that gives us the shortest path tree (SPT).

Example 1: Job Scheduling

Input: A set of n jobs (homework assignments),
where each job j is associated with a size s_j and a deadline d_j.

Output: An assignment of jobs to time slots such that:

1) each job gets a number of time slots that equals its size;
2) each job is completed before its corresponding deadline.

Example:

jobs	1	2	3	4	5
size	1	2	1	2	2
deadline	2	4	5	8	9

time	1	2	3	4	5	6	7	8	9	10
job assignment	1	-	2	2	3	4	4	5	5	-

Greed is good!

A greedy algorithm for job scheduling:
Finish the job (homework assignment) with the closest deadline, and then the job with the second closest deadline, and so on.

Greed is good!

A greedy algorithm for job scheduling:
Finish the job (homework assignment) with the closest deadline, and then the job with the second closest deadline, and so on.

Claim. If the greedy algorithm fails to find a feasible schedule, then the input jobs do not admit a feasible schedule.

Greed is good!

A greedy algorithm for job scheduling:
Finish the job (homework assignment) with the closest deadline, and then the job with the second closest deadline, and so on.

Claim. If the greedy algorithm fails to find a feasible schedule, then the input jobs do not admit a feasible schedule.

Proof. (Assume for simplicity that $d_1 < d_2 < \ldots < d_n$.)

Greed is good!

A greedy algorithm for job scheduling:
Finish the job (homework assignment) with the closest deadline, and then the job with the second closest deadline, and so on.

Claim. If the greedy algorithm fails to find a feasible schedule, then the input jobs do not admit a feasible schedule.

Proof. (Assume for simplicity that $d_1 < d_2 < \ldots < d_n$.)

- The first s_1 time slots are assigned to job *1*, the next s_2 time slots are assigned to job *2*, and so on.

Greed is good!

A greedy algorithm for job scheduling:
Finish the job (homework assignment) with the closest deadline, and then the job with the second closest deadline, and so on.

Claim. If the greedy algorithm fails to find a feasible schedule, then the input jobs do not admit a feasible schedule.

Proof. (Assume for simplicity that $d_1 < d_2 < \ldots < d_n$.)

- The first s_1 time slots are assigned to job 1, the next s_2 time slots are assigned to job 2, and so on.
- Let j be a job that is not completed by its deadline.

Greed is good!

A greedy algorithm for job scheduling:
Finish the job (homework assignment) with the closest deadline, and then the job with the second closest deadline, and so on.

Claim. If the greedy algorithm fails to find a feasible schedule, then the input jobs do not admit a feasible schedule.

Proof. (Assume for simplicity that $d_1 < d_2 < \ldots < d_n$.)

- The first s_1 time slots are assigned to job *1*, the next s_2 time slots are assigned to job *2*, and so on.
- Let j be a job that is not completed by its deadline.
- That means, $s_1 + s_2 + \ldots + s_j > d_j$, namely, the amount of work that has to be done from time 1 to time d_j is more than d_j!

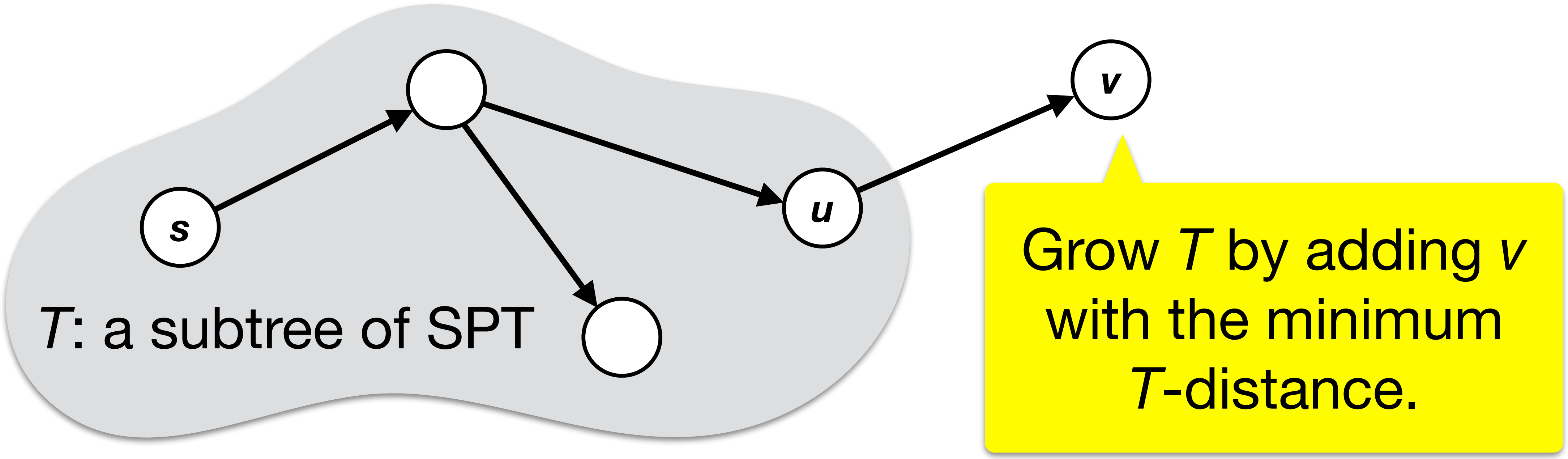
Dijkstra Algorithm as a Greedy Algorithm

Input: A directed graph $G = (E, V)$, where each edge (u, v) is associated with a length $L(u, v)$, and a starting vertex s.

Output: A Shortest Path Tree (SPT) rooted at s.

Dijkstra Algorithm:

- Starting from the smallest subtree of SPT that contains only s.
- Iteratively attached a "correct" edge to the subtree such that the larger tree is still a subtree of SPT.



This Lecture:

A Greedy Algorithm for the

Minimum Spanning Tree Problem

Spanning Tree

Definition. Given a connected undirected graph $G = (V, E)$, a spanning tree of G is a subset of edges that forms a tree that contains all the vertices.

Examples:

- Connecting all cities in a country by building the minimum number of highways.
- Building a network that connects a set of hubs using the minimum number of cables.

Solutions:

- BFS or DFS (the BFS and DFS trees are spanning trees if the graph is connected).

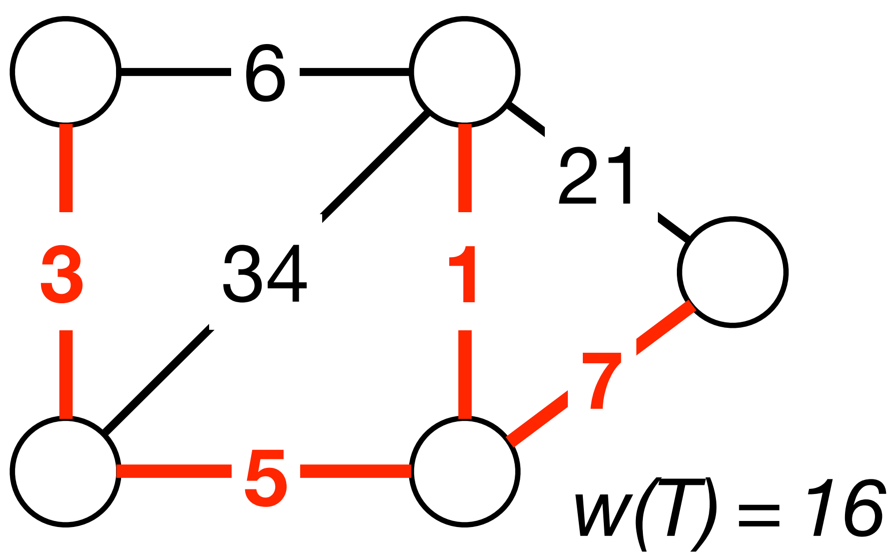
Minimum Spanning Tree

Definition: Given a connected undirected graph $G = (V, E)$ in which every edge $e \in E$ is associated with a positive weight $w(e)$, a **minimum spanning tree (MST)** is a subset of edges $T \subseteq E$ s.t.

(i) T forms a spanning tree; and

(ii) the sum of edge weights of T is minimized.

Example:



For the short path problem, each edge is associated with a length. For the MST problem, each edge is associated with a weight.

Two Useful Properties

1. Removing an edge in a cycle will not disconnect a graph.
2. *Let* $G = (V, E)$ be an undirected graph. The following three statements are equivalent:
   - G is a spanning tree.
   - G is connected and does not have any cycle.
   - G is connected and has $|V| - 1$ edges.

Two Greedy Algorithms for MST

Prim’s algorithm (this lecture)

- Start with some root node *s* and grow a tree *T* outward.
- At each step, add the minimum weight outgoing edge.
- This algorithm is almost the same as the Dijkstra's algorithm, except that we add the outgoing edge with the minimum weight, not the one with minimum *T*-distance.

Kruskal’s algorithm (sequel lectures)

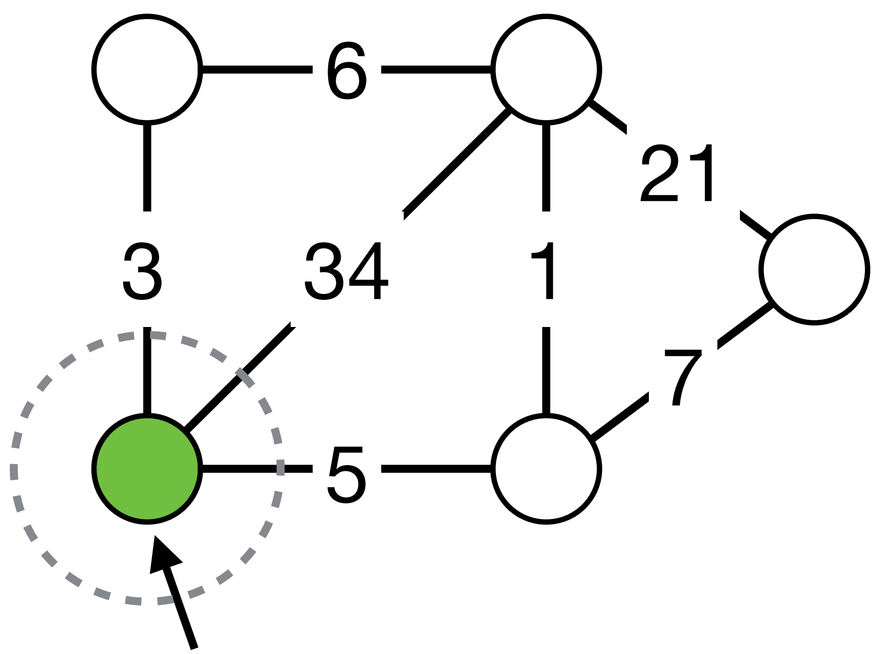
- Start with *T* being the empty tree.
- Consider edges in ascending order of cost; insert edge *e* in *T* unless doing so would create a cycle.

Prim's Algorithm (Chapter 5.1.5)

1) Start with some root node *s* and grow a tree *T* outward.
2) At each step, add the minimum weight outgoing edge.

Prim's Algorithm (Chapter 5.1.5)

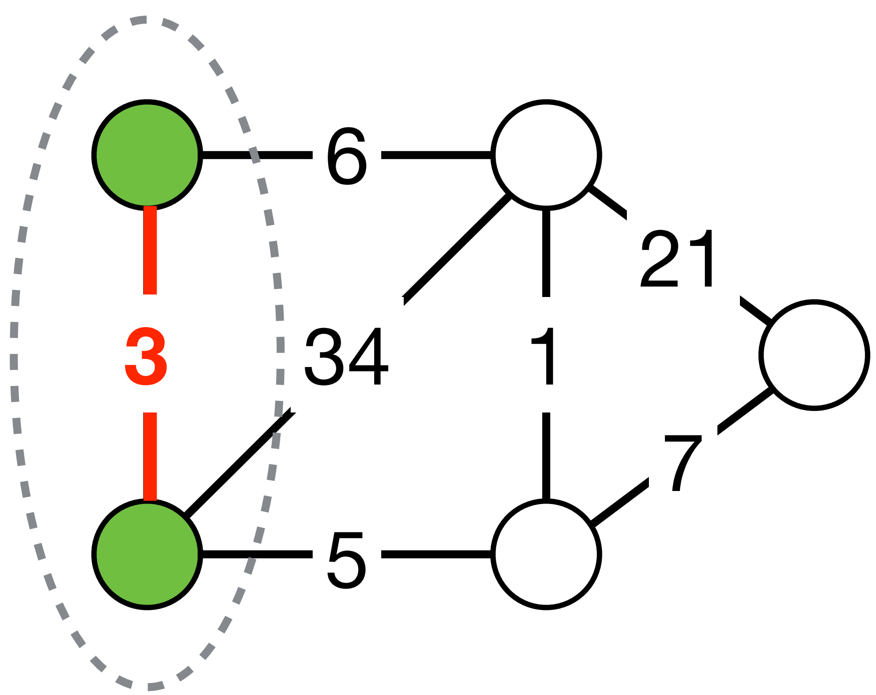
1) Start with some root node *s* and grow a tree *T* outward.
2) At each step, add the minimum weight outgoing edge.



Initially, we start from a root node *s*

Prim's Algorithm (Chapter 5.1.5)

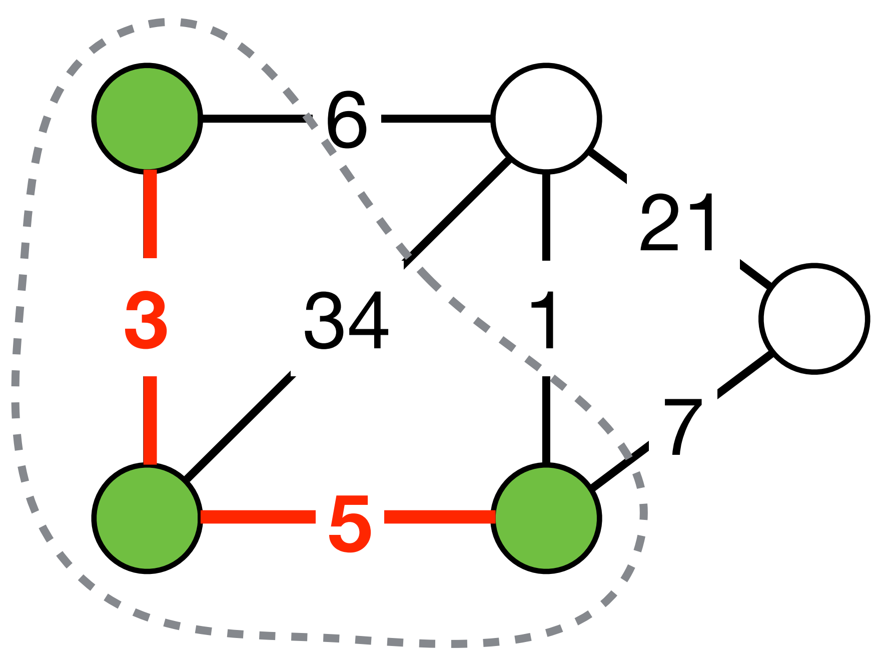
1) Start with some root node *s* and grow a tree *T* outward.
2) At each step, add the minimum weight outgoing edge.



At step one, there are two outgoing edges with weight 3 and 5; add the edge with weight 3.

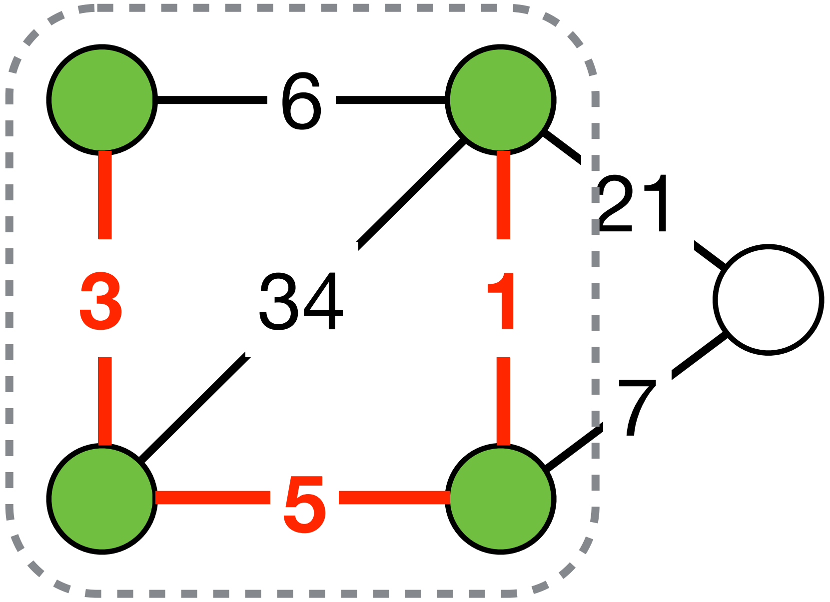
Prim's Algorithm (Chapter 5.1.5)

1) Start with some root node *s* and grow a tree *T* outward.
2) At each step, add the minimum weight outgoing edge.



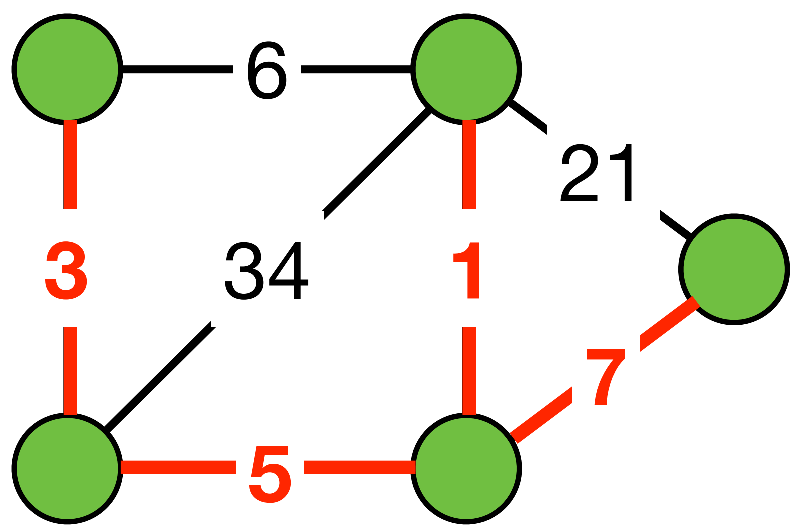
Prim's Algorithm (Chapter 5.1.5)

1) Start with some root node *s* and grow a tree *T* outward.
2) At each step, add the minimum weight outgoing edge.



Prim's Algorithm (Chapter 5.1.5)

1) Start with some root node *s* and grow a tree *T* outward.
2) At each step, add the minimum weight outgoing edge.

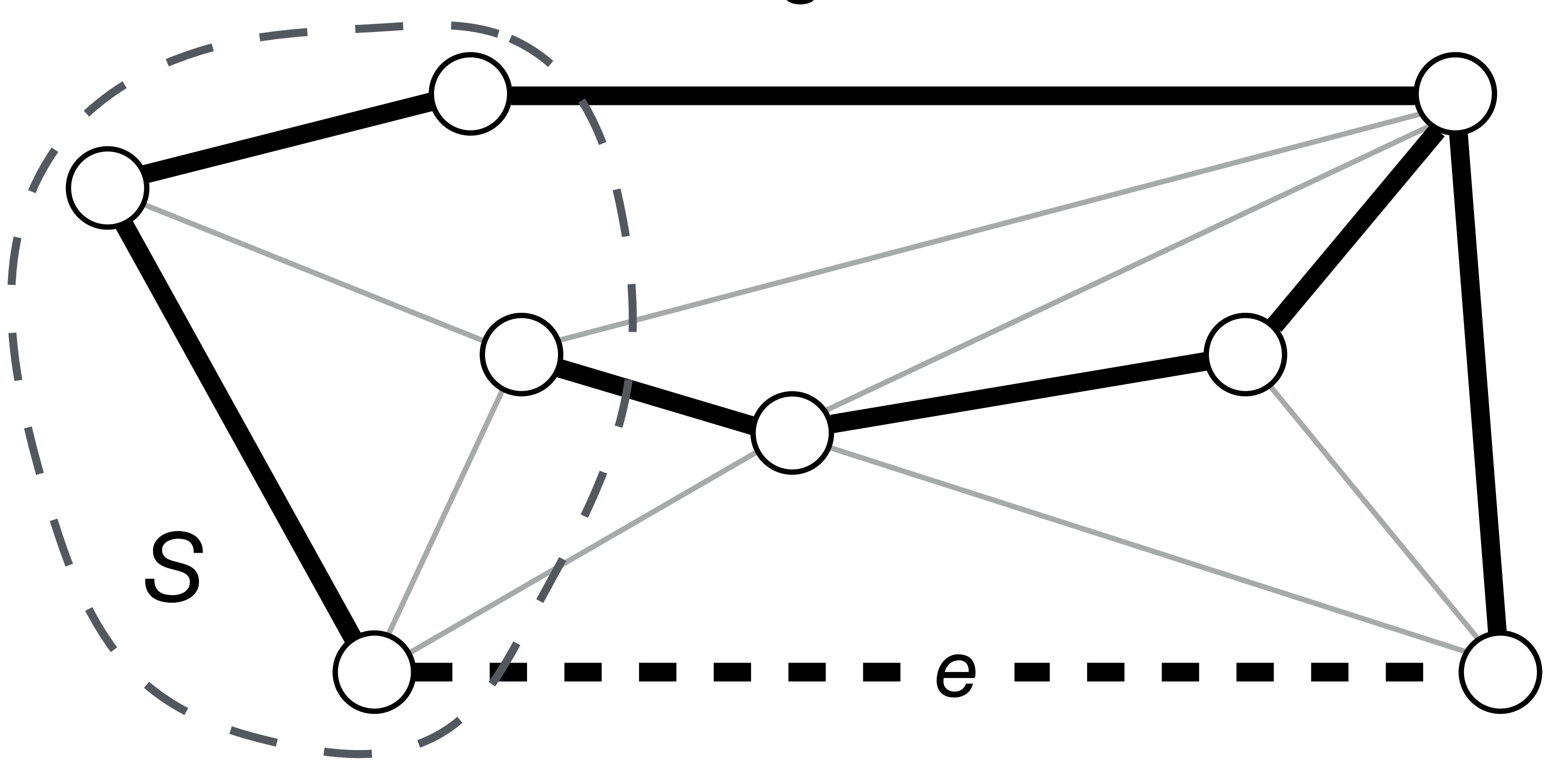


Correctness of Prim's Algorithm

Lemma. Let $S \subseteq V$ be any subset of vertices, and let $e \in E$ be the outgoing edge of S with the smallest weight (call this edge the minimum outgoing edge of S). Then the MST T^* contains e.

Proof. (exchange argument)

- To simplify the discussion, we assume that all edges have distinct weights. In this case, the MST is unique.
- Suppose e does not belong to T*.

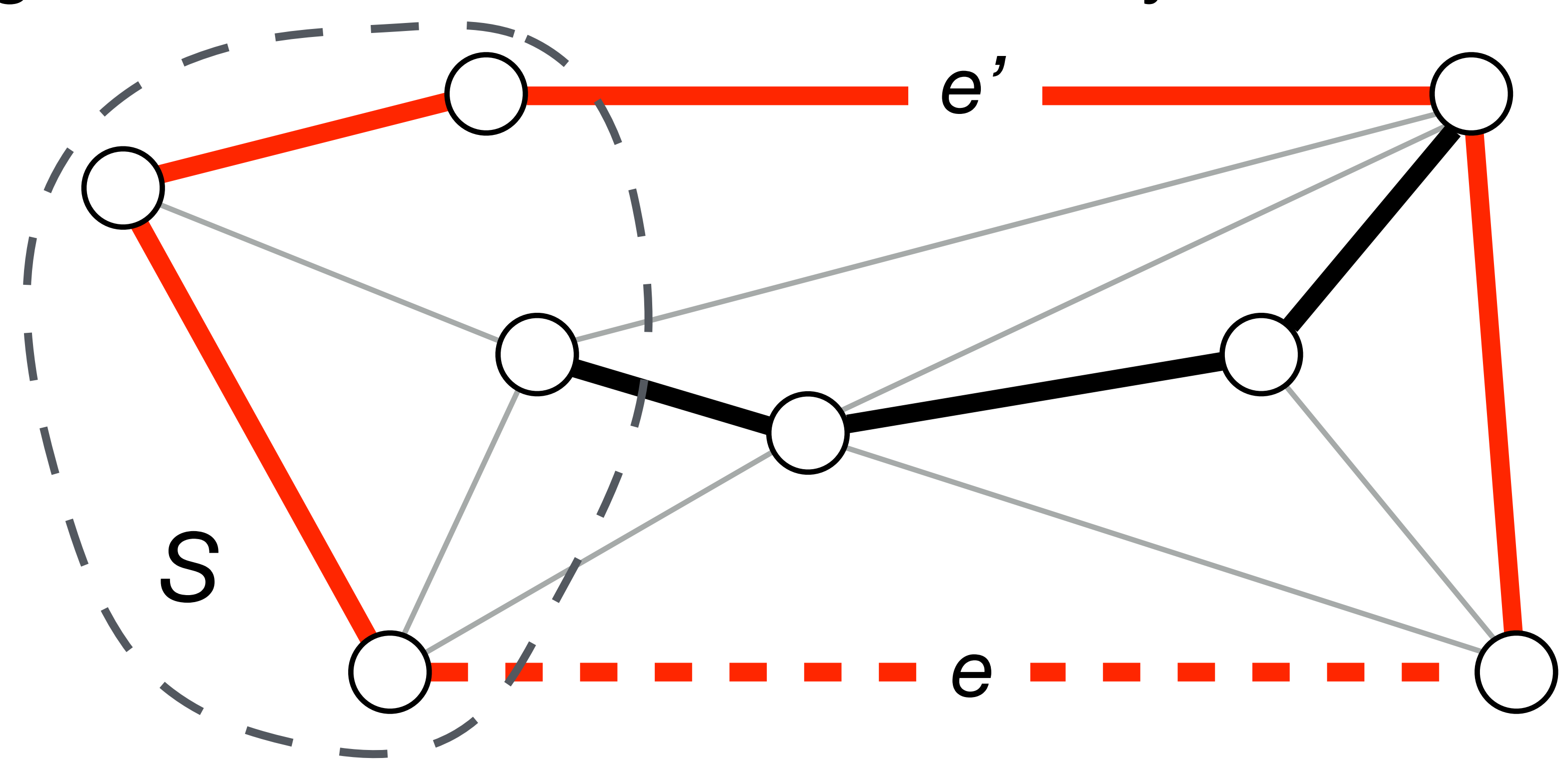


Correctness of Prim's Algorithm

Lemma. Let $S \subseteq V$ be any subset of vertices, and let $e \in E$ be the outgoing edge of S with the smallest weight (call this edge the minimum outgoing edge of S). Then the MST T^* contains e.

Proof. (exchange argument)

- Adding e to T^* creates a cycle C (the red edges) in T^*.
- There must be an edge in the cycle other than e which bring us from inside S to outside, say, e'.

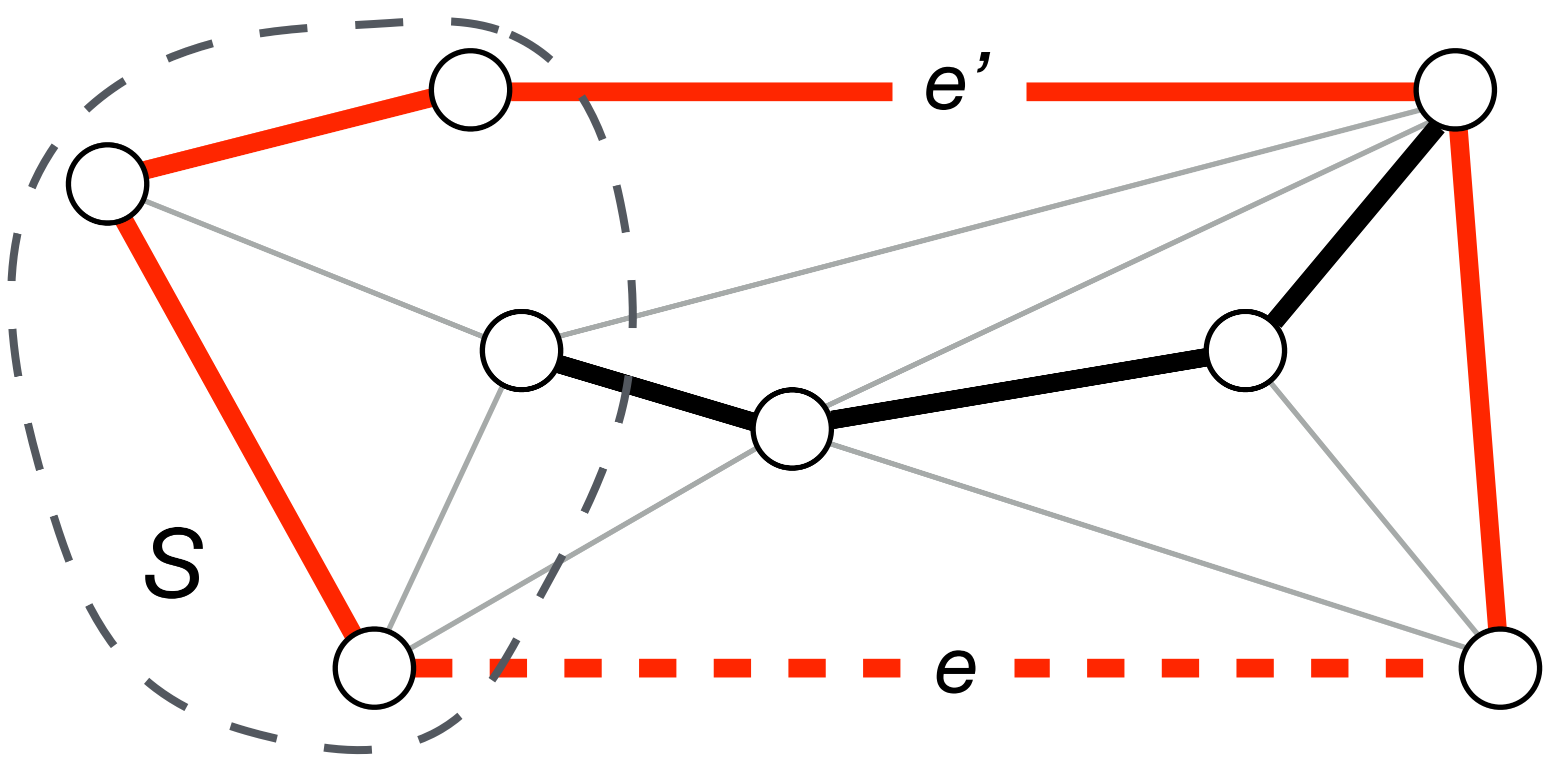


Correctness of Prim's Algorithm

Lemma. Let $S \subseteq V$ be any subset of vertices, and let $e \in E$ be the outgoing edge of S with the smallest weight (call this edge the minimum outgoing edge of S). Then the MST T^* contains e.

Proof. (exchange argument)

- By our assumption, w(e') > w(e).

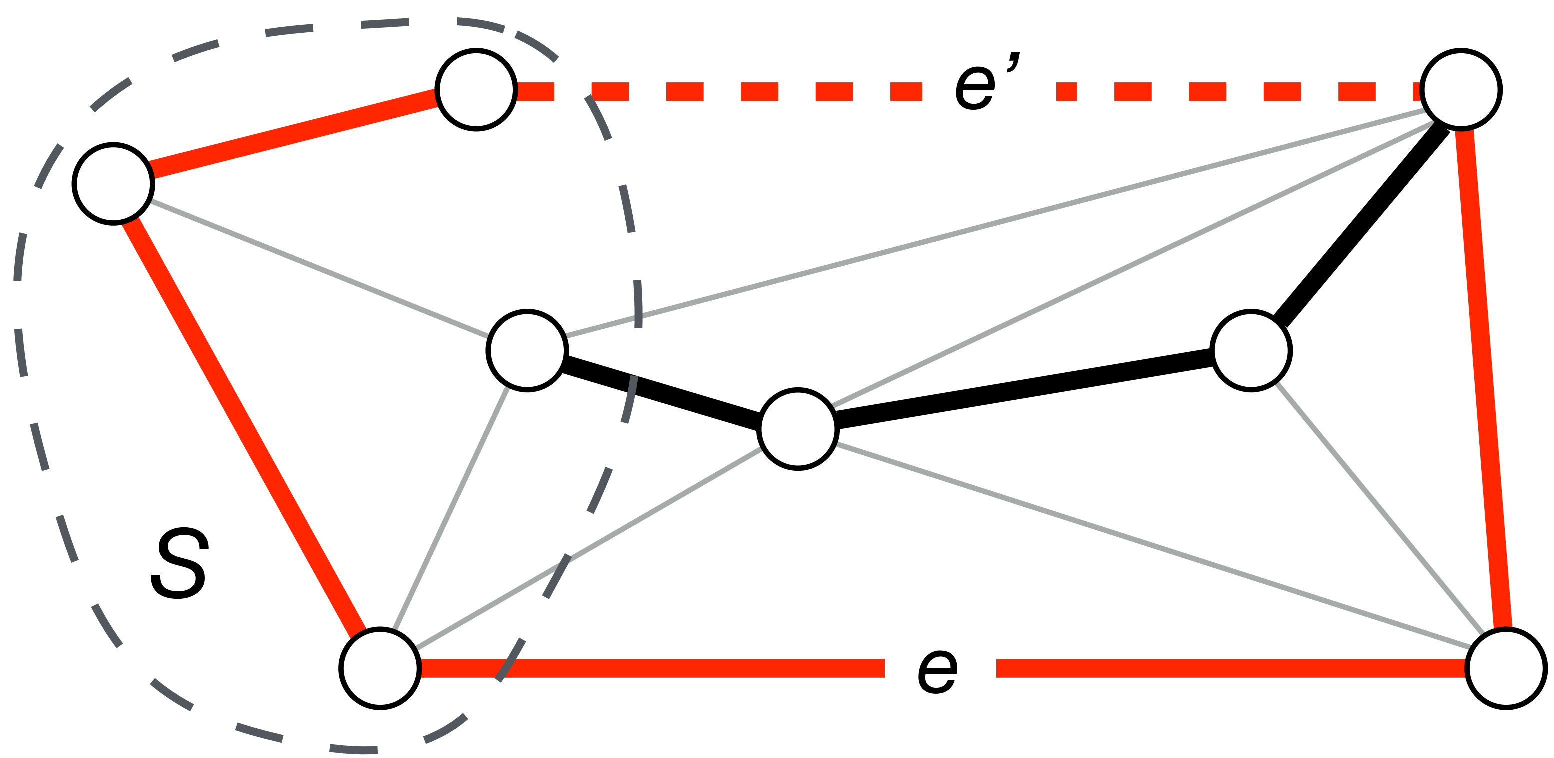


Correctness of Prim's Algorithm

Lemma. Let $S \subseteq V$ be any subset of vertices, and let $e \in E$ be the outgoing edge of S with the smallest weight (call this edge the minimum outgoing edge of S). Then the MST T^* contains e.

Proof. (exchange argument)

- Removing e' from T^* and adding e to it, we get another spanning tree whose total weight of edges is smaller.



Prim's Algorithm

[Jarník ’30, Prim ’57, Dijkstra ’59]

1) Choose the starting node s arbitrarily;
2) **initialize** $S = \{ s \}$ and $T = \{ \}$;
3) **for** i = 1 to $|V| - 1$ **:**
4) add the outgoing edge from S with minimum weight to T;
5) add the corresponding new vertex to S.

Correctness: Follow from the previous Lemma.

Running Time: (leave as exercise)

- **Key question:** How to efficiently find the edge in step 4?
- **Hint:** This is the similar to the Dijkstra's algorithm.
- $O((|E| + |V|) \log |V|)$ if we use a binary heap implementation.
- $O(|E| \log_{|E|/|V|} |V|)$ using d-heap for an appropriate value of d.