VE281

Data Structures and Algorithms

Shortest Path Problem and Minimum Spanning Trees

Announcement

- Pre-test for programming project two will be available to you by this Friday.
  - Please see TA announcement on Sakai.
- Programming project three will be put online by this Friday.
  - About graph algorithms.
  - Due in two weeks.
- Participate in the online course evaluation "IDEA".
  - It will close on Dec. 16th.
  - Follow the link in an email sent to your SJTU email account.

Review

- Breadth-First Search
- Topological Sorting
- Shortest Path Problem
  - Unweighted graph

Outline

- Shortest Path Problem for Weighted Graph
- Minimum Spanning Tree

Shortest Path for Weighted Graphs

- The problem becomes more difficult when edges have different weights.
  - Weights represent different costs on using those edges.

- We require all weights to be **non-negative**.

- The standard algorithm is the **Dijkstra’s Algorithm**.

Dijkstra's Algorithm

- A greedy algorithm for solving single source all destinations shortest path problem.

- Basic idea: if the shortest path from s to d passes through an intermediate node v_i, i.e., $P = (s, \ldots, v_i, \ldots, d)$, then $P' = (s, \ldots, v_i)$ must be the shortest path from s to v_i.

Dijkstra's Algorithm

- Keep **distance estimates** $D(v)$ and **predecessor** $P(v)$ for each node v.
  - Predecessor: the previous node on the shortest path.

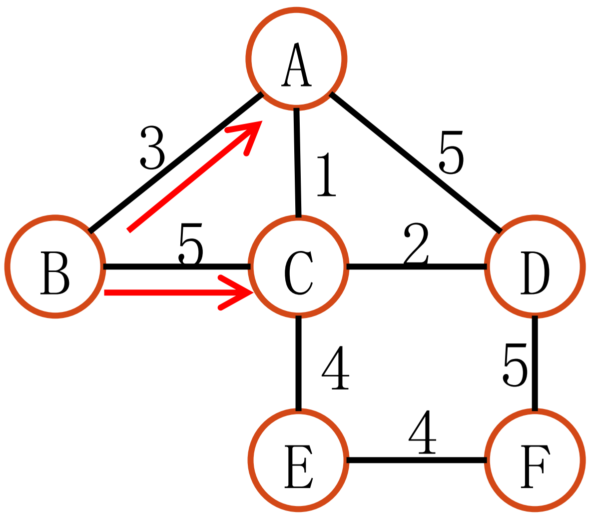
1. Initially, $D(s) = 0$; $D(v)$ for each of the other nodes is infinite; $P(v)$ is unknown.
2. Store all the nodes in a set R.
3. While R is not empty
   1. Choose node v in R such that $D(v)$ is the smallest. Remove v from the set R.
   2. Declare that v's shortest distance is known, which is $D(v)$.
   3. For each of v's neighbors u that is **still in** R, update distance estimate $D(u)$ and predecessor $P(u)$.

Updating

- For each of v's neighbors u that is still in R, if $D(v) + w(v,u) < D(u)$, then update $D(u) = D(v) + w(v,u)$ and the predecessor $P(u) = v$.
  - I.e., update $D(u)$ if the path going through v is cheaper than the best path so far to u.

Dijkstra’s Algorithm
Example

- Suppose B is the source.



Initial set up

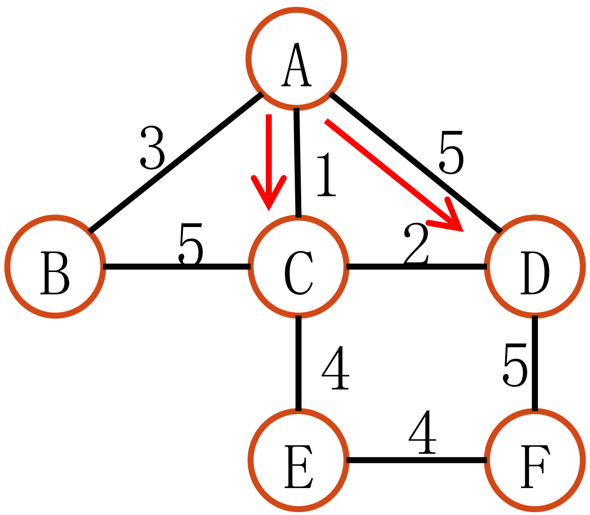
R = {A, B, C, D, E, F}

	A	B	C	D	E	F
$D(\cdot)$	∞	0	∞	∞	∞	∞
$P(\cdot)$	–	–	–	–	–	–

Update B’s neighbors (still in R)

Dijkstra’s Algorithm
Example

- Suppose B is the source.



R = {A, C, D, E, F}

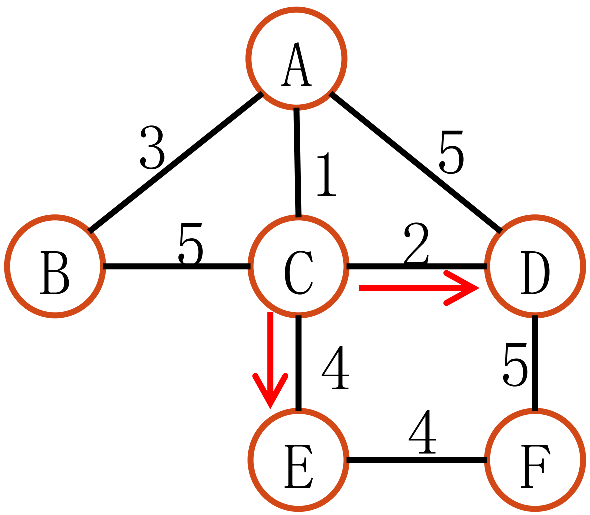
	A	B	C	D	E	F
$D(\cdot)$	$\cancel{\infty}$ 3	0	$\cancel{\infty}$ 5	∞	∞	∞
$P(\cdot)$	B	–	B	–	–	–

$$D(B) + w(B, A) = 3 < D(A)$$
$$D(B) + w(B, C) = 5 < D(C)$$

Update A’s neighbors (still in R)

Dijkstra's Algorithm
Example

- Suppose B is the source.



R = {C, D, E, F}

	A	B	C	D	E	F
$D(\cdot)$	3	0	$\cancel{5}$ 4	$\cancel{\infty}$ 8	∞	∞
$P(\cdot)$	B	–	$\cancel{B}$ A	A	–	–

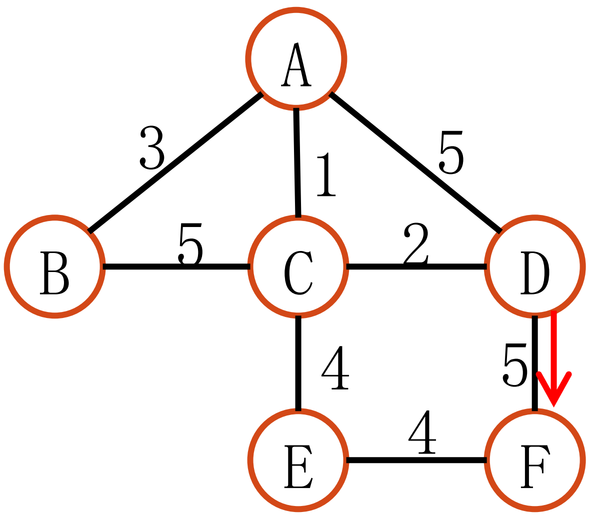
$$D(A) + w(A, C) = 4 < D(C)$$
$$D(A) + w(A, D) = 8 < D(D)$$

Update C's neighbors (still in R)

Dijkstra’s Algorithm

Example

- Suppose B is the source.



R = {D, E, F}

	A	B	C	D	E	F
$D(\cdot)$	3	0	4	$\cancel{8}$ 6	$\cancel{\infty}$ 8	∞
$P(\cdot)$	B	–	A	$\cancel{A}$ C	C	–

$$D(C) + w(C, D) = 6 < D(D)$$

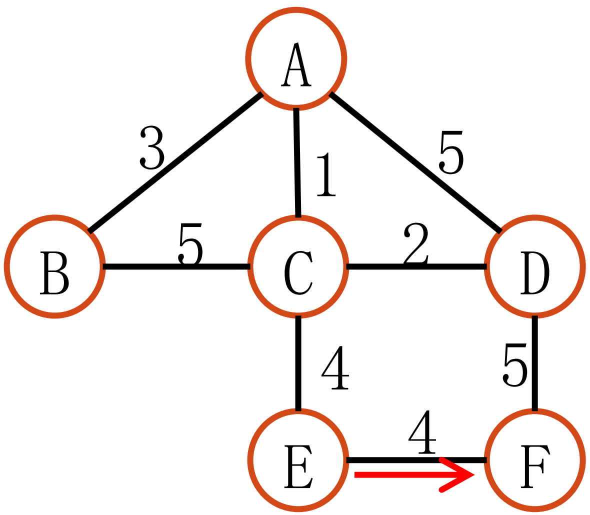
$$D(C) + w(C, E) = 8 < D(E)$$

Update D’s neighbors (still in R)

Dijkstra’s Algorithm

Example

- Suppose B is the source.



R = {E, F}

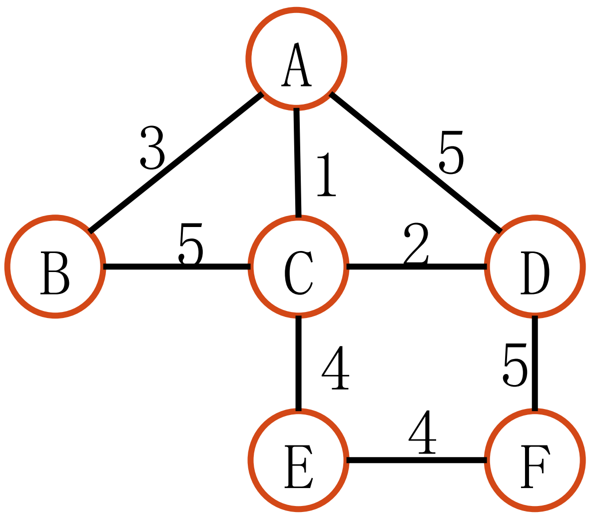
	A	B	C	D	E	F
$D(\cdot)$	3	0	4	6	8	$\cancel{\infty}$ 11
$P(\cdot)$	B	–	A	C	C	D

$$D(D) + w(D, F) = 11 < D(F)$$

Update E’s neighbors (still in R)

Dijkstra's Algorithm
Example

- Suppose B is the source.



R = {F}

	A	B	C	D	E	F
$D(\cdot)$	3	0	4	6	8	11
$P(\cdot)$	B	–	A	C	C	D

$D(E) + w(E, F) = 12 > D(F)$ No update

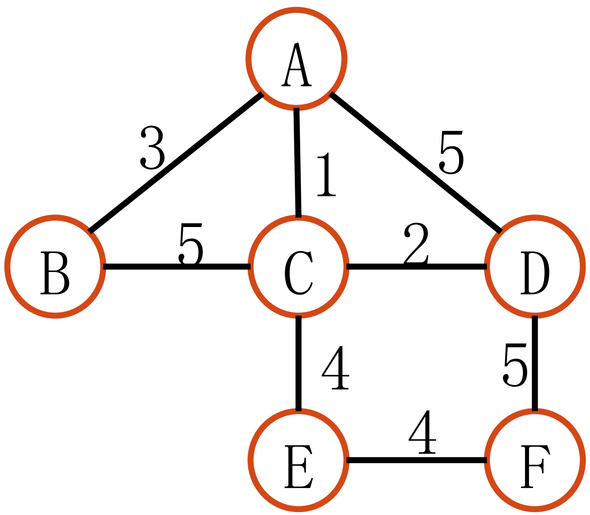
Update F's neighbors (still in R)

F has no neighbor in R

Dijkstra's Algorithm

Example

- Suppose B is the source.



$R = \emptyset$

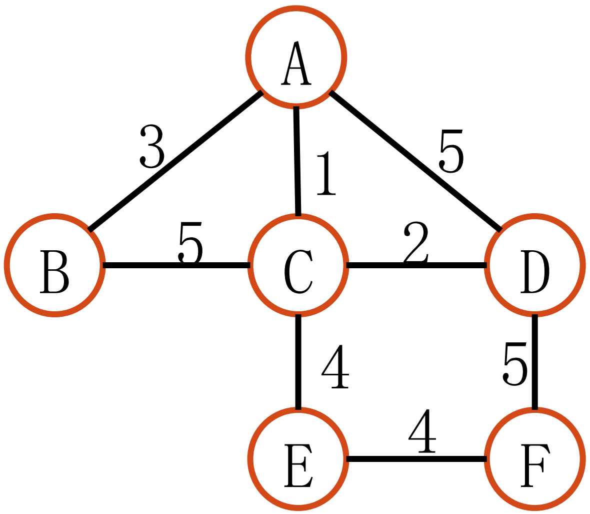
We are done.

	A	B	C	D	E	F
$D(\cdot)$	3	0	4	6	8	11
$P(\cdot)$	B	–	A	C	C	D

Obtaining the Shortest Path

- We can obtain the shortest path by backtracking.
  - E.g., shortest from B to F

$B \rightarrow A \rightarrow C \rightarrow D \rightarrow F$



	A	B	C	D	E	F
$D(\cdot)$	3	0	4	6	8	11
$P(\cdot)$	B	–	A	C	C	D

Dijkstra's Algorithm
Proof

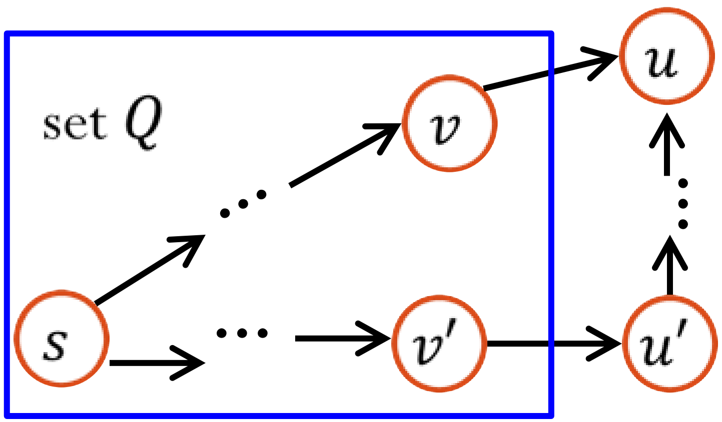
- We want to prove that each time when we choose $D(v)$ that is the smallest, then $D(v)$ is the shortest distance for v.

- We prove this by mathematical induction.
- Base case: the source node is chosen. Its shortest distance is 0.
- Inductive step: Assume that the set of nodes chosen so far all have their $D(v)$ as the shortest distance. We want to prove that adding the closest neighbor is also correct.
  - Prove by contradiction.

Dijkstra's Algorithm

Proof

- Suppose in this step, $D(u)$ is the smallest. So u is chosen. Suppose its predecessor is v.
- Contradiction: the path from node s to u through v is not the shortest; there exists an even shorter path from s to u.
- Assume the set of nodes chosen so far is Q.
- Assume the shorter path is $P = (s, \dots, v', u', \dots, u)$, with $s, \dots, v' \in Q$ and $u' \notin Q$.
- The path to u' should be shorter than the path to u.
- Then we should have chosen u' instead of choosing u.



Dijkstra's Algorithm

Time Complexity

- Method 1: linear scan the set R to find the smallest $D(v)$.
- Number of times to find the smallest $D(v)$: $|V|$.
  - Each cost: $O(|V|)$.
- Total number of times to update the neighbors: $|E|$.
  - Since each neighbor of each node could be potentially updated.
  - Each cost: $O(1)$.
- Total running time is $O(|E| + |V|^2) = O(|V|^2)$.

Dijkstra's Algorithm

Time Complexity

- Method 2: use a priority queue to store $D(v)$'s.
- Number of times to find the smallest $D(v)$: $|V|$.
  - Each cost: $O(\log |V|)$.
- Total number of times to update the neighbors: $|E|$.
  - Each cost is $O(\log |V|)$, since after updating $D(v)$, we should restore the priority queue property.
- Total running time is $O(|V| \log |V| + |E| \log |V|)$ $= O(|E| \log |V|)$.

Dijkstra's Algorithm

Time Complexity

- Method 1: linear scan the set R to find the smallest $D(v)$.
  - Total running time: $O(|V|^2)$.

- Method 2: use a priority queue to store $D(v)$'s.
  - Total running time: $O(|E| \log |V|)$.

- Which one is better?
  - For sparse graphs, i.e., $|E| \approx |V|$, using priority queue is better.
  - For dense graphs, i.e., $|E| \approx |V|^2$, using linear scan is better.

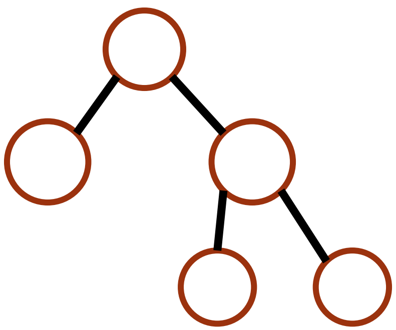
Outline

- Shortest Path Problem for Weighted Graph
- Minimum Spanning Tree

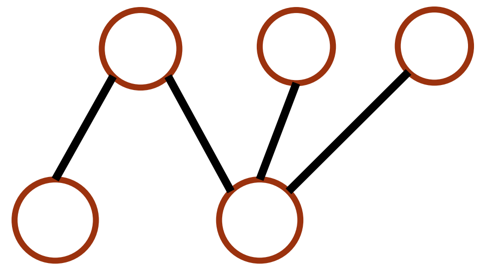
Tree and Graph

- A **tree** is an **acyclic, connected undirected** graph.

The tree we see before



However, this is also a tree

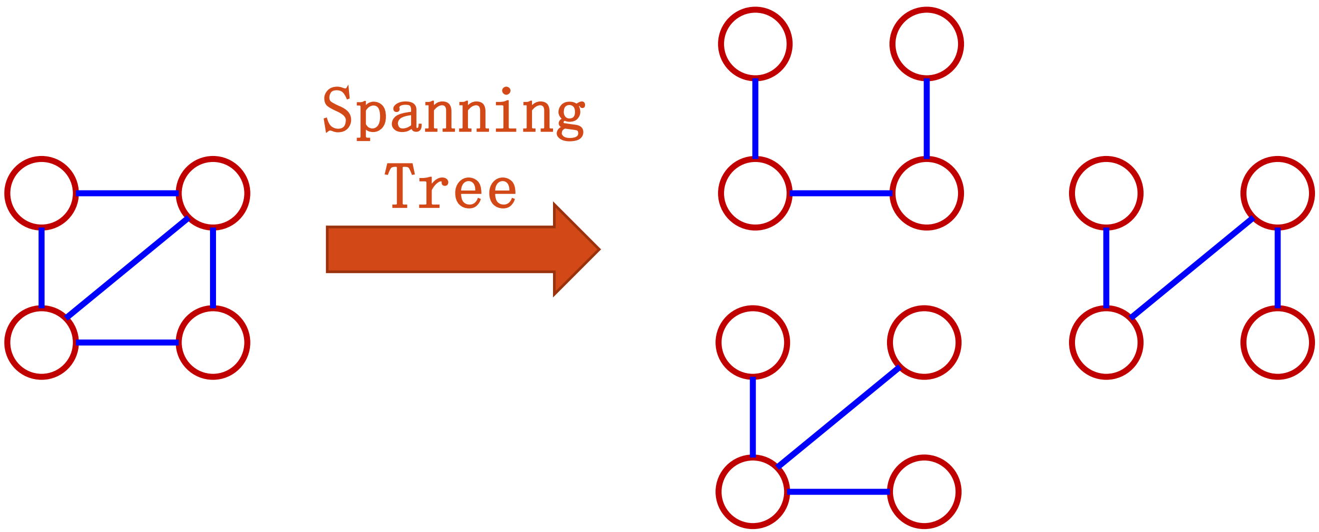


Any node can be the root of the tree.

- For a tree, $|E| = |V| - 1$.
- Any connected graph with N nodes and $N - 1$ edges is a tree.

Subgraph and Spanning Tree

- $G' = (V', E')$ is a **subgraph** of $G = (V, E)$ if and only if $V' \subseteq V$ and $E' \subseteq E$.
- A **spanning tree** of a connected undirected graph G is a subgraph of G that
  - contains all the nodes of G;
  - is a tree, i.e., connected and acyclic.

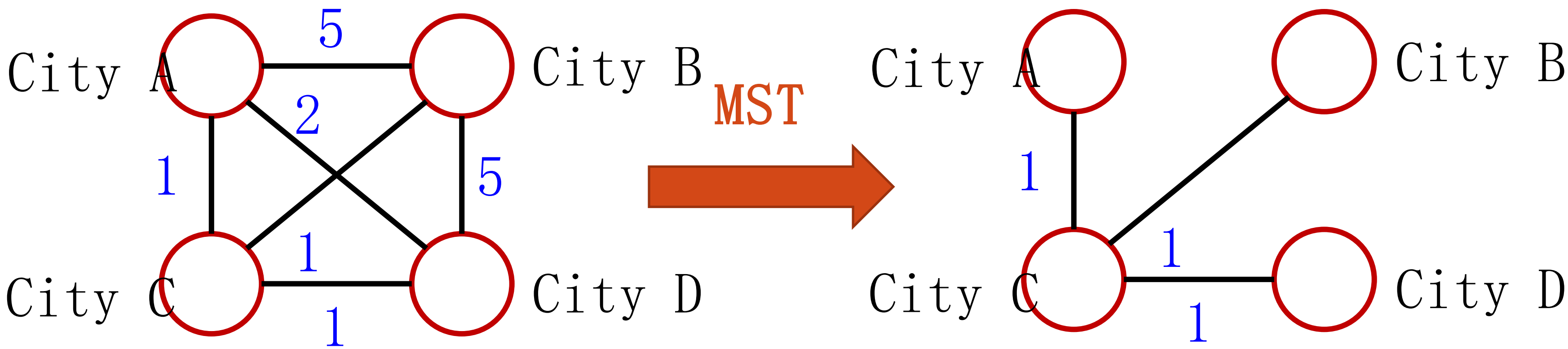


Minimum Spanning Tree (MST)

- Given a weighted, connected, undirected graph $G = (V, E)$, a **minimum spanning tree** T of G is a spanning tree of G whose sum of all edge weights is the minimal.

Application of MST

- A government planning a freeway system to connect all the cities.



- A railroad company planning where to lay down tracks.
- A power company planning where to lay down high-voltage power lines.

Minimum Spanning Tree

Algorithms

- Main idea: greedily select edges one by one and add to a growing sub-graph.

- Two standard algorithms:
  - Prim's algorithm
  - Kruskal's algorithm

Prim's Algorithm

- Separate V into two sets:
  - T: the set of nodes that we have added to the MST.
  - T': those nodes that have not been added to the MST, i.e., $T' = V - T$.

- Prim's algorithm initially sets T as empty and T' as V. The algorithm moves one node from T' to T in each iteration. After the last iteration, $T = V$ and we have constructed the MST.

Prim's Algorithm

Basic Version

- Keep **distance estimates** $D(v)$ and **predecessor** $P(v)$ for each node v.
  - Predecessor: the previous node on the shortest path.

1. Initially, $D(s) = 0$; $D(v)$ for each of the other nodes is infinite; $P(v)$ is unknown.
2. Store all the nodes in a set R.
3. While R is not empty
   1. Choose node v in R such that $D(v)$ is the smallest. Remove v from the set R.
   2. Declare that v's shortest distance is known, which is $D(v)$.
   3. For each of v's neighbors u that is **still in R**, update distance estimate $D(u)$ and predecessor $P(u)$.

Selecting the Smallest Edge and Node

- For each node $v \in T'$, keep a measure $D(v)$, storing the **smallest weight** of any edge that connects any node in T to v.
- To choose the edge with the smallest weight that connects between a node in T and a node in T', we pick the node $v \in T'$ with the smallest $D(v)$.
- If we move a node v from T' to T, then for each of v's neighbor u that is in T', we update its $D(u)$ as follows:
  - If $D(u) > w(v, u)$, then let $D(u) = w(v, u)$.
  - I.e., update $D(u)$ if the weight of edge (v, u) is smaller than the weight of any other edge that connects a node in T to u.

Prim’s Algorithm
Full Version

- We keep previous node $P(v)$ for each node v to record the edges chosen in the MST.

1. Arbitrarily pick one node s. Set $D(s) = 0$. For any other node v, set $D(v)$ as infinite and $P(v)$ as unknown.
2. While $T' \neq \emptyset$
   1. Choose node v in T' such that $D(v)$ is the smallest. Remove v from the set T'.
   2. For each of v’s neighbors u that is still in T', if $D(u) > w(v,u)$, then update $D(u)$ as $w(v,u)$ and $P(u)$ as v.

Prim’s algorithm is similar to Dijkstra’s algo

Prim’s Algorithm v.s. Dijkstra’s Algorithm

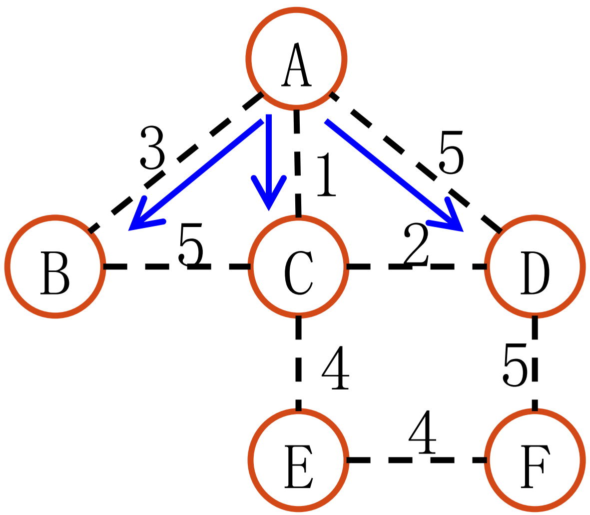
- Dijkstra’s algorithm: grow the set of nodes to which we know the shortest path.

- Prim’s algorithm: grow the set of nodes we have added to the MST.

Prim's Algorithm

Example

Randomly choose a node, say node A



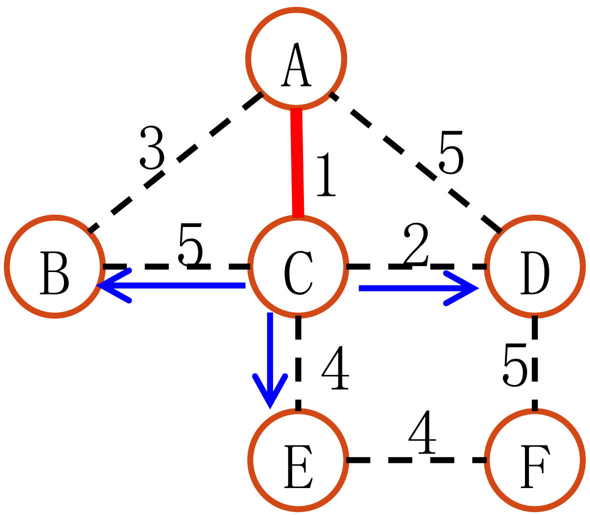
$D(A) + w(A, C) = 4 < D(C)$

	A	B	C	D	E	F
$D(\cdot)$	0	∞	∞	∞	∞	∞
$P(\cdot)$	–	–	–	–	–	–

$D(A) + w(A, D) = 8 < D(D)$

Prim’s Algorithm

Example



$$D(C) + w(C, D) = 6 < D(D)$$

	A	B	C	D	E	F
$D(\cdot)$	0	$\cancel{\infty}$ 3	$\cancel{\infty}$ 1	$\cancel{\infty}$ 5	∞	∞
$P(\cdot)$	–	A	A	A	–	–

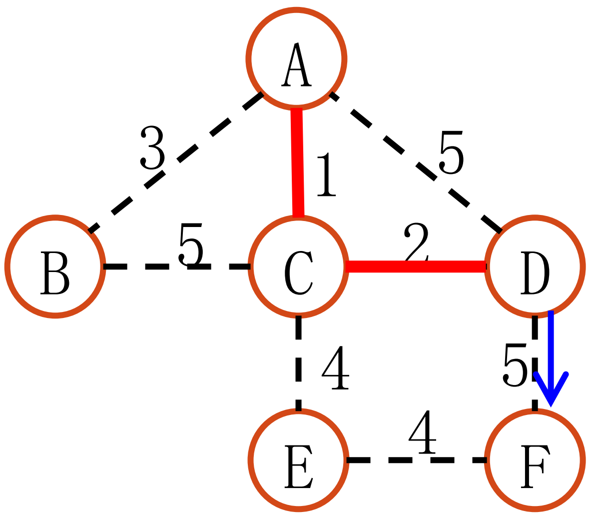
$$w(A, C) = 1 < D(C)$$

$$w(A, D) = 5 < D(D)$$

$$D(C) + w(C, E) = 8 < D(E)$$

Prim’s Algorithm

Example



$T' = \{B, D, E, F\}$

	A	B	C	D	E	F
$D(\cdot)$	0	3	1	$\cancel{5}$ 2	$\cancel{\infty}$ 4	∞
$P(\cdot)$	–	A	A	A C	C	–

$w(C, B) = 5 > D(B)$ No update

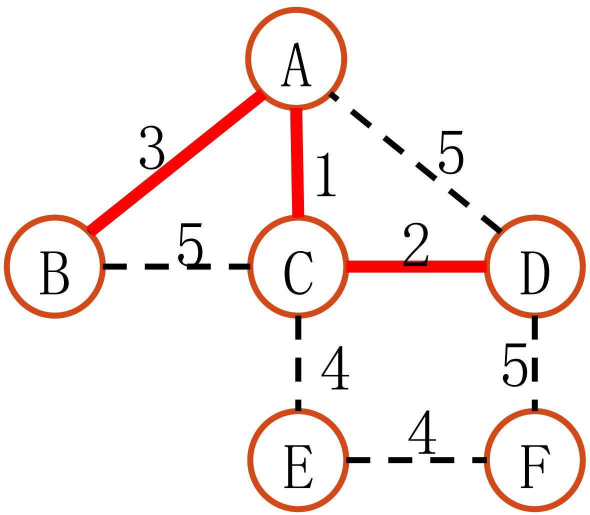
$w(C, D) = 2 < D(D)$

	A	B	C	D	E	F
$D(\cdot)$	3	0	4	6	8	$\cancel{\infty}$ 11
$P(\cdot)$	B	–	A	C	C	D

Update D’s neighbors (still in T')

Prim’s Algorithm

Example



	A	B	C	D	E	F
$D(\cdot)$	3	0	4	6	8	11
$P(\cdot)$	B	–	A	C	C	D

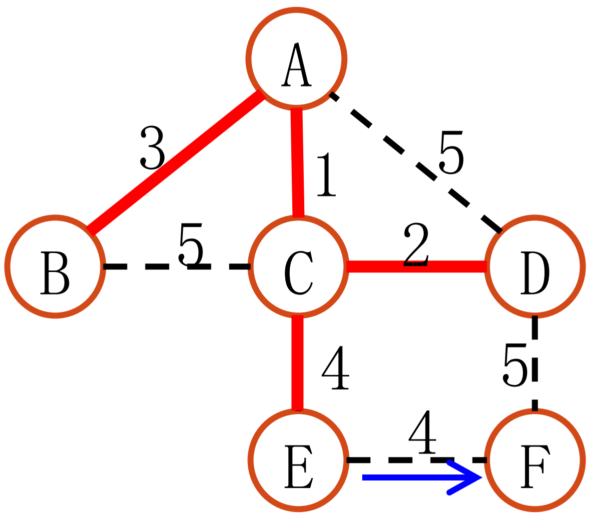
	A	B	C	D	E	F
	0	3	1	2	4	~~∞~~ 5
	–	A	A	C	C	D

$R = \emptyset$

$$D(E) + w(E, F) = 12 > D(F)$$

- We want to prove that each time when we choose $D(v)$ that is the smallest, then $D(v)$ is the shortest distance for v.
- We prove this by mathematical induction.
- Base case: the source node is chosen. Its shortest distance is 0.
- Inductive step: Assume that the set of nodes chosen so far all have their $D(v)$ as the shortest distance. We want to prove that adding the closest neighbor is also correct.
  - Prove by contradiction.

Prim’s Algorithm
Example



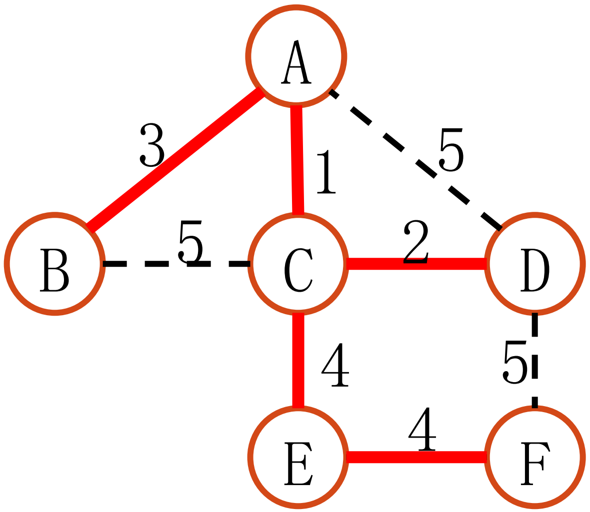
v

	A	B	C	D	E	F
$D(\cdot)$	0	3	1	2	4	5
$P(\cdot)$	–	A	A	C	C	D

S

Prim’s Algorithm

Example



u'

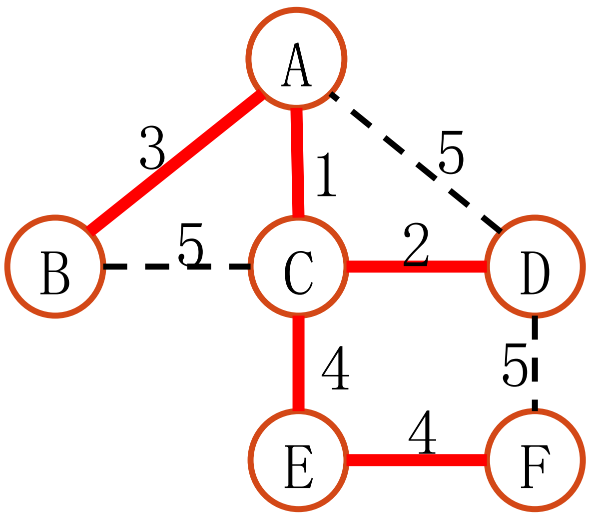
	A	B	C	D	E	F
	0	3	1	2	4	~~5~~ 4
	–	A	A	C	C	~~D~~ E

v'

set Q

Prim’s Algorithm

Example



- Method 2: use a priority queue to store $D(v)$'s.
- Number of times to find the smallest $D(v)$: $|V|$.
  - Each cost: $O(\log|V|)$.
- Total number of times to update the neighbors: $|E|$.
  - Each cost is $O(\log|V|)$, since after updating $D(v)$, we should restore the priority queue property.
- Total running time is $O(|V|\log|V| + |E|\log|V|)$ $= O(|E|\log|V|)$.

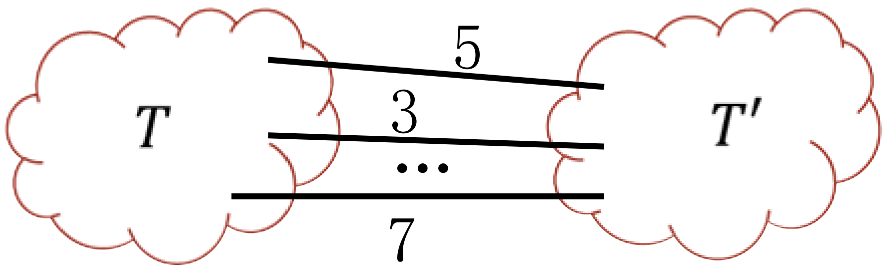
We are done.

	A	B	C	D	E	F
$D(\cdot)$	0	3	1	2	4	4
$P(\cdot)$	–	A	A	C	C	E

Prim's Algorithm

Justification

- Let T and T' be a partition of V. In a spanning tree, there must exist at least one edge that connects one node in T to another node in T'.
  - Otherwise, it is not a spanning tree.



- Prim's algorithm grows set T and each time greedily picks the edge with the smallest weight that connects a node in T to a node in T'. It ensures:
  1. All nodes are connected and there are no cycles, i.e., a tree.
  2. The sum of all edge weights is minimal.

Prim’s Algorithm

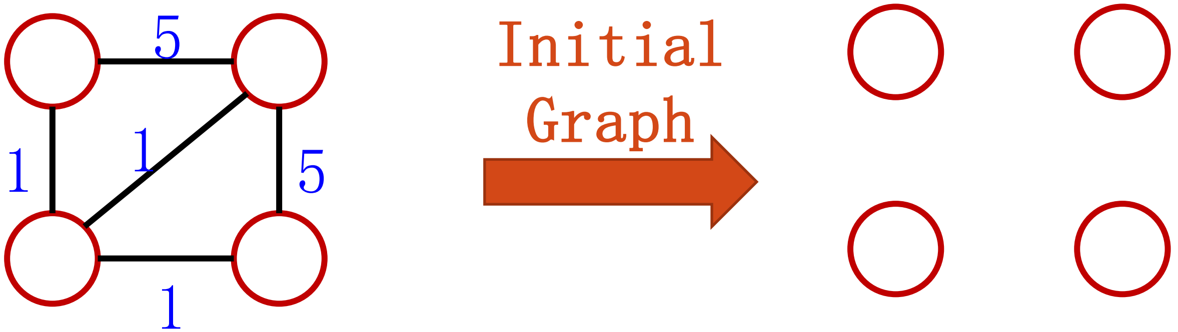
Time Complexity

- Number of times to find the smallest $D(v)$: $|V|$.
  - Cost? Linear scan: $O(|V|)$; Priority queue: $O(\log|V|)$
- Total number of times to update the neighbors: $|E|$.
  - Since each neighbor of each node could be potentially updated.
  - Cost? Linear scan: $O(1)$; Priority queue: $O(\log|V|)$

- Total time complexity
  - Linear scan: $O(|E|+|V|^2)=O(|V|^2)$.
  - Priority queue:
    $O(|V|\log|V|+|E|\log|V|)=O(|E|\log|V|)$.

Kruskal's Algorithm

- Start with a graph containing $|V|$ nodes and no edges

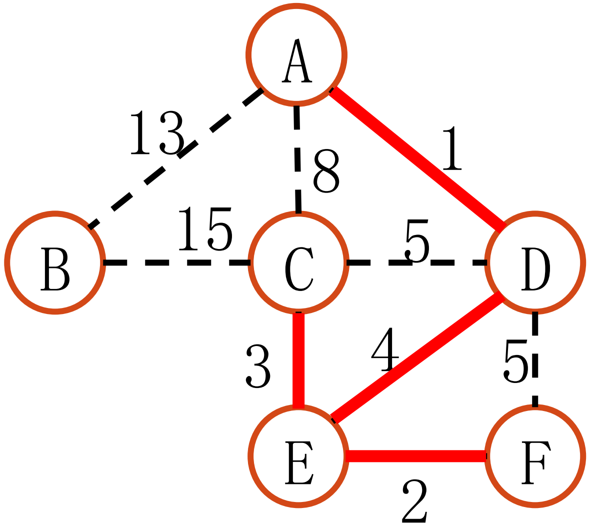


- This initial graph can be viewed as a **forest** of trees.
  - Each tree only has a single node.
- Main idea: repeatedly add the edge with the **smallest weight** that **does not cause a cycle** until no such edges exist.
  - Each added edge performs a union on two trees in the forest.
  - After adding $|V| - 1$ edges, there is only one tree. This tree is the MST.

Kruskal's Algorithm

Example

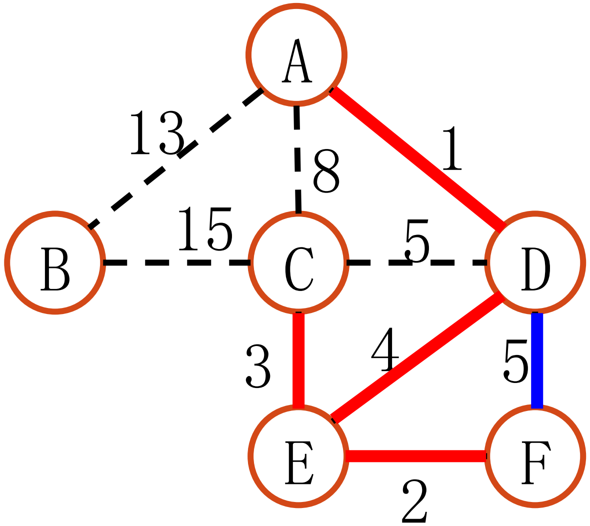
Repeatedly add the edge with the **smallest weight** that does not cause a cycle until no such edges exist.



Kruskal’s Algorithm

Example

Repeatedly add the edge with the **smallest weight** that does not cause a cycle until no such edges exist.



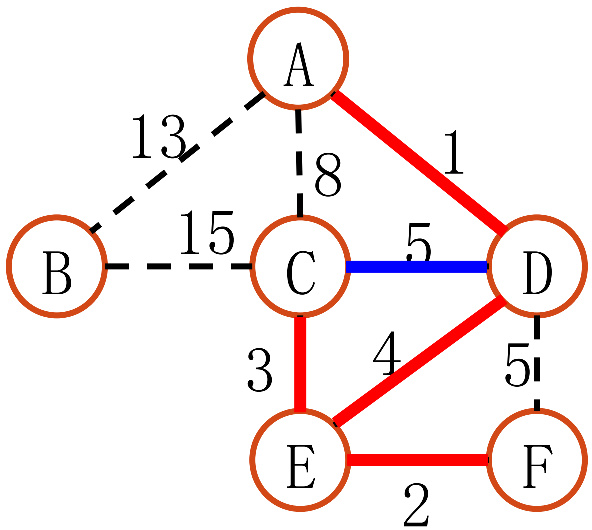
The next edge with the smallest weight is (D,

However, adding it causes a cycle. So it is discarded.

Kruskal’s Algorithm

Example

Repeatedly add the edge with the **smallest weight** that does not cause a cycle until no such edges exist.



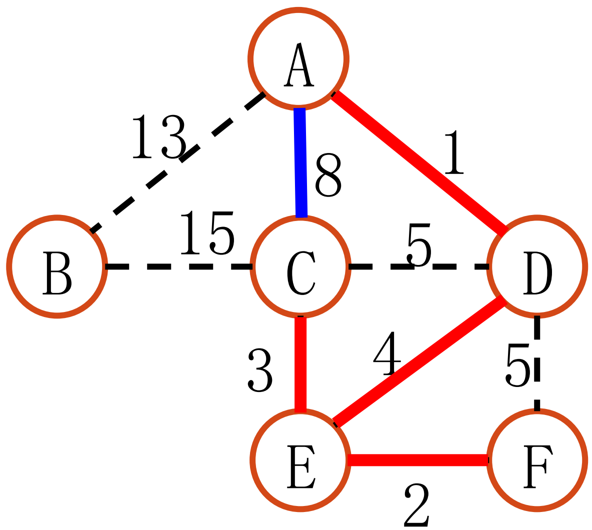
The next edge with the smallest weight is (C,

However, adding it causes a cycle. So it is discarded.

Kruskal’s Algorithm

Example

Repeatedly add the edge with the **smallest weight** that does not cause a cycle until no such edges exist.



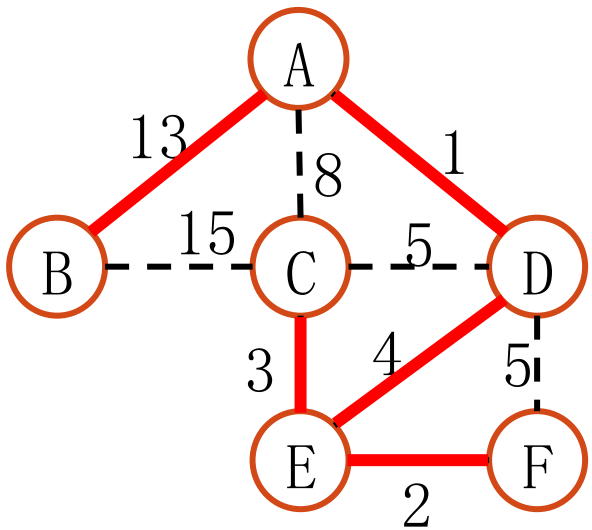
The next edge with the smallest weight is (A,

However, adding it causes a cycle. So it is discarded.

Kruskal' s Algorithm

Example

Repeatedly add the edge with the **smallest weight** that **does not cause a cycle** until no such edges exist.



The next edge with the smallest weight is (A,

MST construction

done.

Detecting Cycles

- Not simple.
- Connected nodes form a **component**.
- Detecting cycle: an edge (u, v) causes a cycle if nodes u and v are in the same component.
- If the edge does not cause a cycle, we add the edge and make union on the two different components connected by the edge.
  - Update the set of components for later detecting cycle purpose.

Kruskal's Algorithm

Implementation and Time Complexity

- Sorting the edges by weights
  - Time complexity: $O(|E| \log |E|)$.
- Detecting cycle. If no cycle, add edge and merge two trees.
  - Time complexity: $O(\log |V|)$. (Not covered)
  - In the worst case, we detect cycles for all edges. The time complexity is $O(|E| \log |V|)$.

- Since $|E| = O(|V|^2)$, the total running time is $O(|E| \log |V|)$.