Minimum Cost Spanning Trees (MSTs)

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,
National Taiwan Ocean University

Fall 2025

Outline


Outline

Cost & Minimum-Cost Spanning Tree

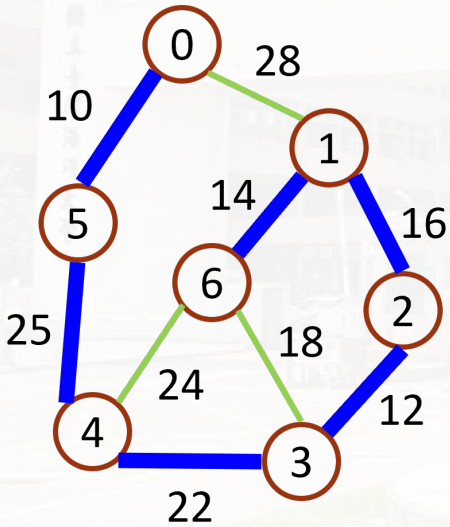
- The cost of a spanning tree of a weighted undirected graph is the sum of the weights of the edges in the spanning tree.

Cost & Minimum-Cost Spanning Tree

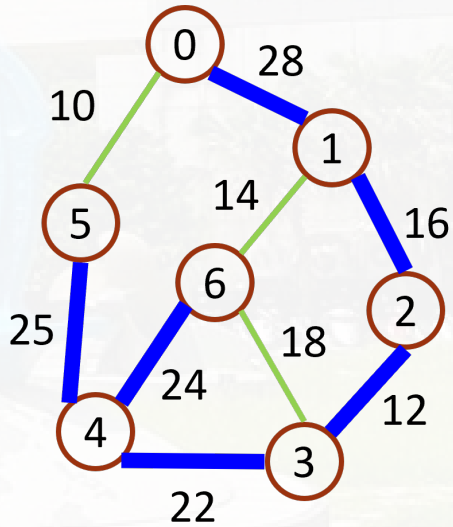
- The cost of a spanning tree of a weighted undirected graph is the sum of the weights of the edges in the spanning tree.
- A minimum-cost spanning tree is a spanning tree of least cost.

Cost & Minimum-Cost Spanning Tree

- The cost of a spanning tree of a weighted undirected graph is the sum of the weights of the edges in the spanning tree.
- A minimum-cost spanning tree is a spanning tree of least cost.



cost $= 10+25+22+12+16+14 = 99.$



cost $= 28+16+12+22+24+25 = 127.$

Greedy Methods

- We will introduce three different greedy algorithms can be used to obtain a minimum cost spanning tree of a connected undirected graph.

Greedy Methods

- We will introduce three different greedy algorithms can be used to obtain a minimum cost spanning tree of a connected undirected graph.
  - Kruskal's algorithm
  - Prim's algorithm
  - Sollin's algorithm

⋆ Further reading: Greedy Algorithms & Matroids [link]

Greedy Method (1/2)

- Construct an optimal solution in stages.
- A feasible solution is one which works within the constrains specified by the problem.
- At each stage, we make a decision that is the best decision at that time.
- Typically, the selection of an item at each stage is based on either a least cost or a highest profit criterion.

Greedy Method (2/2)

- For minimum spanning trees, we use a least cost criterion. Our solution must satisfy the following constrains:

Greedy Method (2/2)

- For minimum spanning trees, we use a least cost criterion. Our solution must satisfy the following constrains:
  - We must use only edges within the graph.

Greedy Method (2/2)

- For minimum spanning trees, we use a least cost criterion. Our solution must satisfy the following constrains:
  - We must use only edges within the graph.
  - Exactly $n-1$ edges are used.

Greedy Method (2/2)

- For minimum spanning trees, we use a least cost criterion. Our solution must satisfy the following constrains:
  - We must use only edges within the graph.
  - Exactly $n-1$ edges are used.
  - Never use edges that would produce a cycle.

Outline


Kruskal's Algorithm

- Build a minimum cost spanning tree T by adding edges to T one at a time.

Kruskal's Algorithm

- Build a minimum cost spanning tree T by adding edges to T one at a time.
- Select the edges for inclusion in T in nondecreasing order of their cost.

Kruskal's Algorithm

- Build a minimum cost spanning tree T by adding edges to T one at a time.
- Select the edges for inclusion in T in nondecreasing order of their cost.
- An edge is added to T if it does not form a cycle with the edges that are already in T.

Kruskal's Algorithm

- Build a minimum cost spanning tree T by adding edges to T one at a time.
- Select the edges for inclusion in T in nondecreasing order of their cost.
- An edge is added to T if it does not form a cycle with the edges that are already in T.
- Exactly $n-1$ edges will be selected for inclusion in T.

Kruskal's Algorithm

- Build a minimum cost spanning tree T by adding edges to T one at a time.
- Select the edges for inclusion in T in nondecreasing order of their cost.
- An edge is added to T if it does not form a cycle with the edges that are already in T.
  - **How to do this?**
- Exactly $n - 1$ edges will be selected for inclusion in T.

Kruskal's Algorithm

- Build a minimum cost spanning tree T by adding edges to T one at a time.
- Select the edges for inclusion in T in nondecreasing order of their cost.
- An edge is added to T if it does not form a cycle with the edges that are already in T.
  - **How to do this? $\Rightarrow$ Union-Find Operations!**
- Exactly $n-1$ edges will be selected for inclusion in T.

Kruskal's Algorithm

- Build a minimum cost spanning tree T by adding edges to T one at a time.
- Select the edges for inclusion in T in nondecreasing order of their cost.
- An edge is added to T if it does not form a cycle with the edges that are already in T.
  - **How to do this? $\Rightarrow$ Union-Find Operations!**
- Exactly $n-1$ edges will be selected for inclusion in T.

- Time complexity: $O(e \log n)$ (assume using tree-based Union-Find).
  - Sorting the edges: $\approx e \log e < e \log n^2 = O(e \log n)$.
  - At most $2e$ find operations $\approx \log n$ time for each.
  - At most $2n-1$ union operations $\approx n$ time.

Illustration of Kruskal's Algorithm

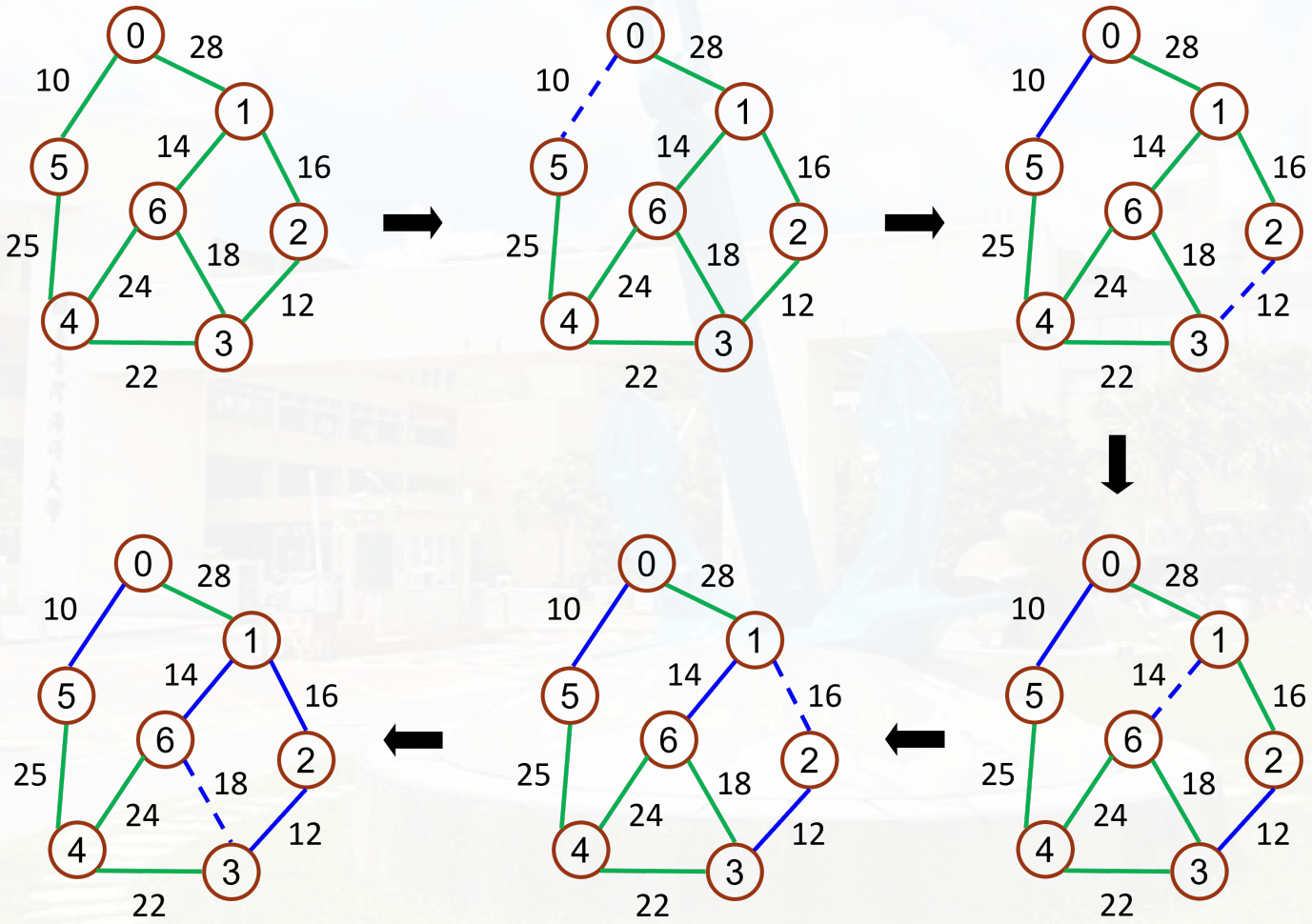
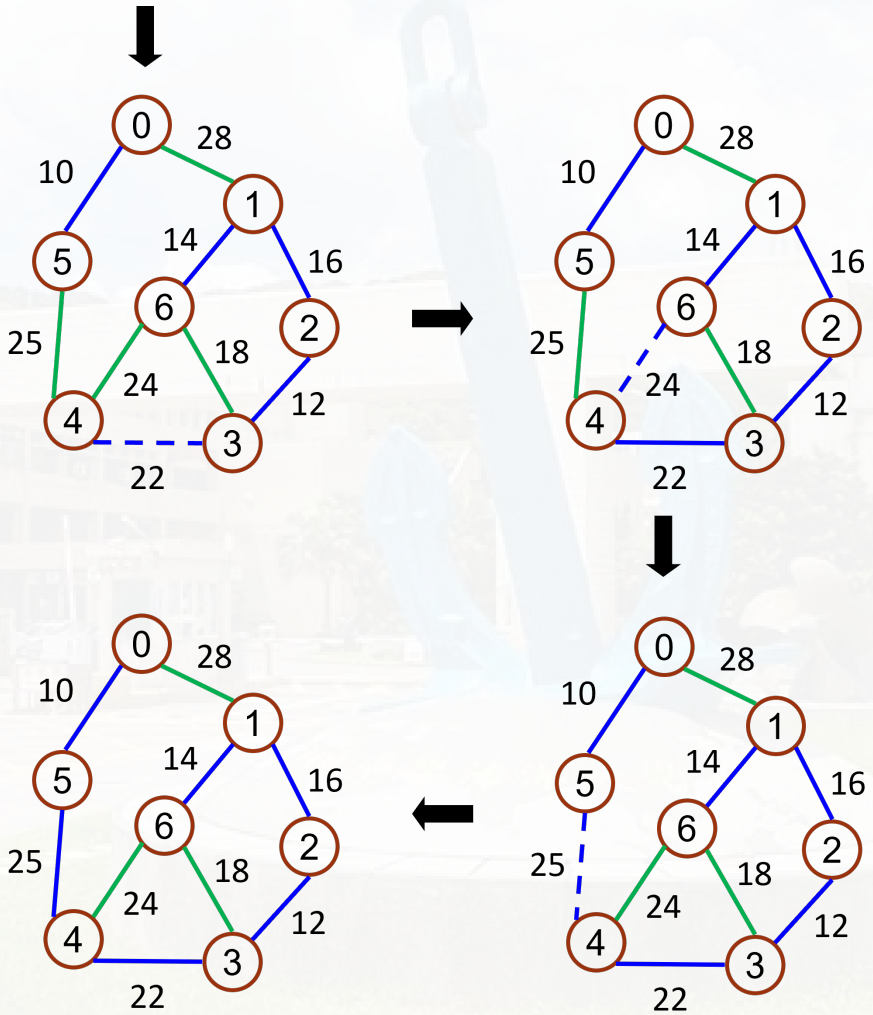


Illustration of Kruskal's Algorithm

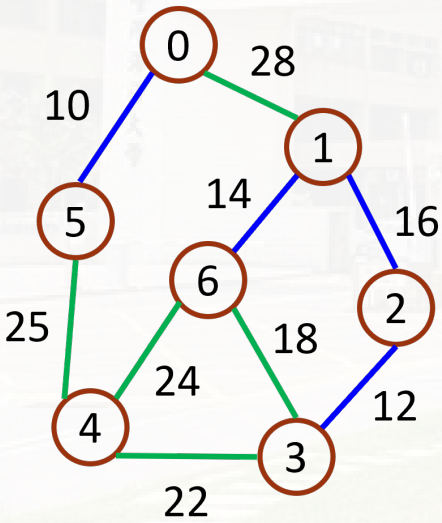


Union-Find Operations

- Union-Find Operations: to determine whether or not adding an edge would cause a cycle.

Union-Find Operations

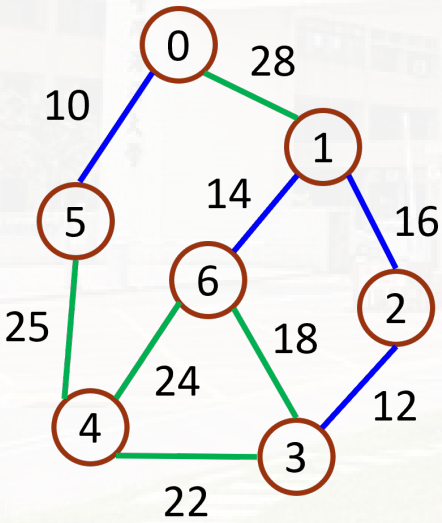
- Union-Find Operations: to determine whether or not adding an edge would cause a cycle.
- For example,



- $\{0, 5\}, \{1, 2, 3, 6\}, \{4\}$: the sets corresponding to existing subtrees.

Union-Find Operations

- Union-Find Operations: to determine whether or not adding an edge would cause a cycle.
- For example,



- $\{0, 5\}, \{1, 2, 3, 6\}, \{4\}$: the sets corresponding to existing subtrees.
- Vertex 3 and 6 are already in the same set $\Rightarrow$ edge $(3, 6)$ is rejected!

The Pseudo-code of Kruskal's Algorithm

```
T = { };
while (T contains fewer than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning tree\n");
```

- How could "No spanning tree" happen?

Outline

Prim's Algorithm (1/3)

- Another greedy MST algorithm.

Prim's Algorithm (1/3)

- Another greedy MST algorithm.
- The main difference:

Prim's Algorithm (1/3)

- Another greedy MST algorithm.
- The main difference:
  - The set of selected edges forms a tree at all times in Prim's algorithm.
  - The set of selected edges in Kruskal's algorithm forms a *forest at each stage*.

Prim's Algorithm (2/3)

- Prim's algorithm begins with a tree T that contains one arbitrary single vertex.

Prim's Algorithm (2/3)

- Prim's algorithm begins with a tree T that contains one arbitrary single vertex.
- Next, we add a **least cost edge** (u, v) to T such that $T \cup (u, v)$ is also a tree.

Prim's Algorithm (2/3)

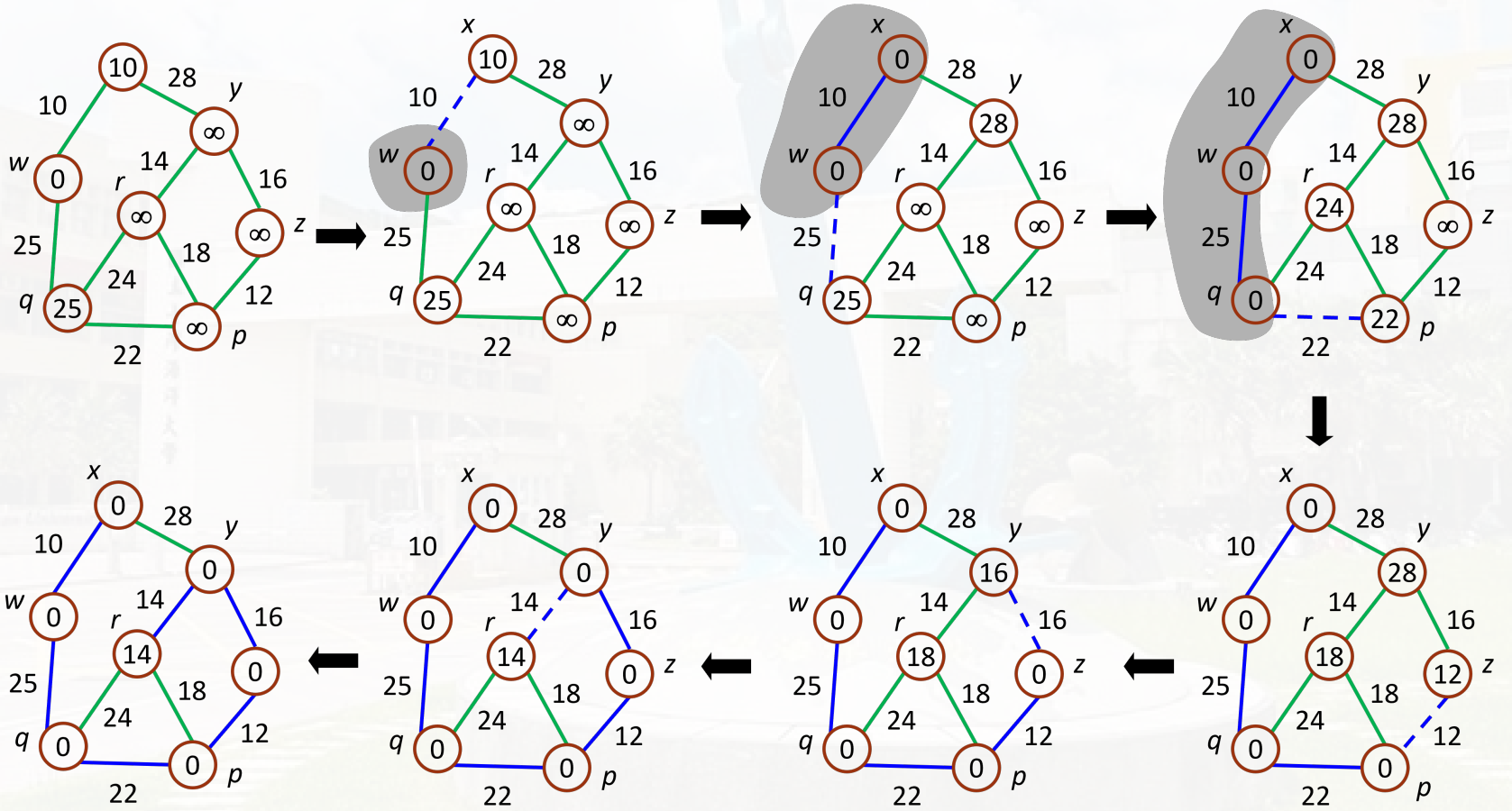
- Prim's algorithm begins with a tree T that contains one arbitrary single vertex.
- Next, we add a **least cost edge** (u, v) to T such that $T \cup (u, v)$ is also a tree.
- Repeat this edge addition until T contains $n - 1$ edges.

Prim's Algorithm (2/3)

- Prim's algorithm begins with a tree T that contains one arbitrary single vertex.
- Next, we add a **least cost edge** (u, v) to T such that $T \cup (u, v)$ is also a tree.
- Repeat this edge addition until T contains $n - 1$ edges.

Time complexity: $O(n^2)$.

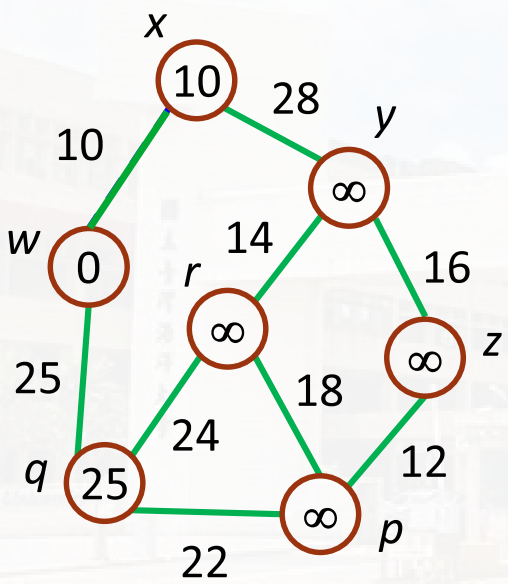
Illustration of Prim's Algorithm



Prim's Algorithm (3/3)

```
T = {}; TV = {0};
while (T contains fewer than n-1 edges) {
    let (u,v) be a least cost edge such that u is in TV and
    v is not in TV;
        if (there is no such edge )
            break;
        add v to TV;
        add (u,v) to T;
}
if (T contains fewer than n-1 edges)
    printf("No spanning tree\n");
```

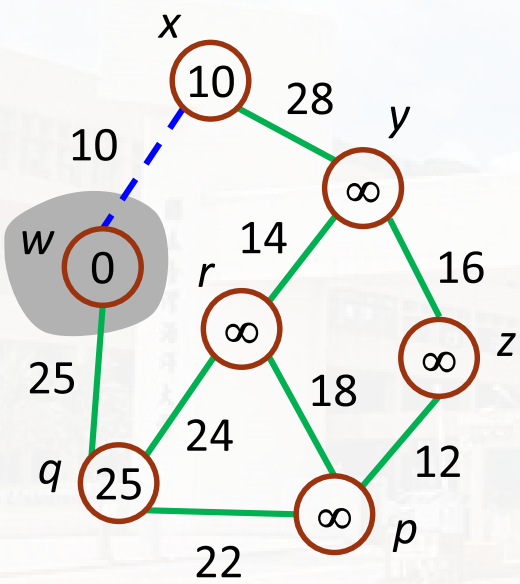
- Each vertex $v \notin TV$ has a companion vertex "near(v)" such that near(v) $\in TV$ and cost(near(v), v) is minimum over all such choices for near(v).
- Therefore, it takes $O(n)$ time to choose an edge.
- We can implement Prim's algorithm in $O(n^2)$ time.



	inMST
w	F
x	F
y	F
z	F
p	F
q	F
r	F

	cost
w	∞
x	∞
y	∞
z	∞
p	∞
q	∞
r	∞

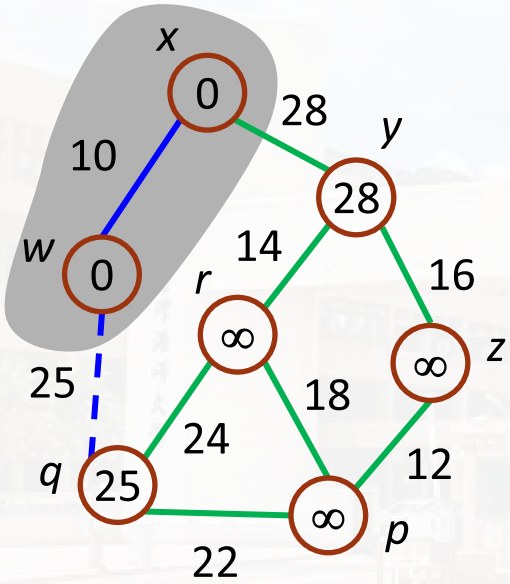
TV = { }



	inMST
w	T
x	F
y	F
z	F
p	F
q	F
r	F

	cost
w	∞
x	10
y	∞
z	∞
p	∞
q	25
r	∞

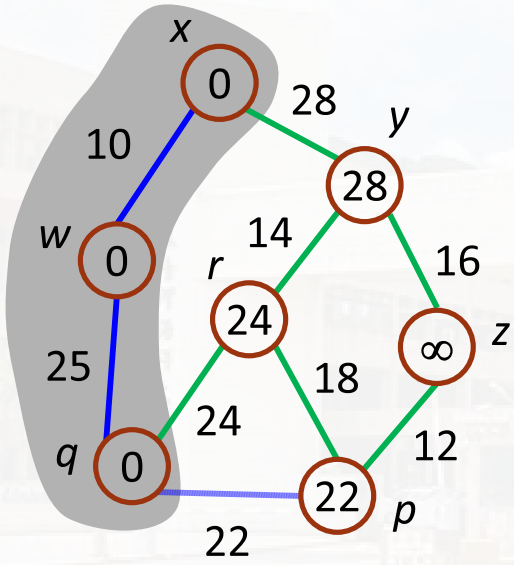
TV = { *w* }



	inMST
w	T
x	T
y	F
z	F
p	F
q	F
r	F

	cost
w	∞
x	10
y	28
z	∞
p	∞
q	25
r	∞

TV = { *w*, *x* }



	inMST
w	T
x	T
y	F
z	F
p	F
q	T
r	F

	cost
w	∞
x	10
y	28
z	∞
p	22
q	25
r	24

TV = { *w*, *x*, *q* }

Outline

Sollin's/ Borůvka's Algorithm (1/2)

- Sollin's algorithm selects several edges for inclusion in T at each stage.
- At the start of a stage, the selected edges, together with all the n vertices, form a spanning forest.
- During each stage, we select one edge for each tree in the forest.
  - This edge is a minimum cost edge that has exactly one vertex in the tree.

Sollin's/ Borůvka's Algorithm (2/2)

- **Note:** two trees in the forest could select the same edge.

Sollin's/ Borůvka's Algorithm (2/2)

- **Note:** two trees in the forest could select the same edge.
  - We need to eliminate duplicate edges.

Sollin's/ Borůvka's Algorithm (2/2)

- **Note:** two trees in the forest could select the same edge.
  - We need to eliminate duplicate edges.
- At the start of the first stage the set of selected edges is empty.

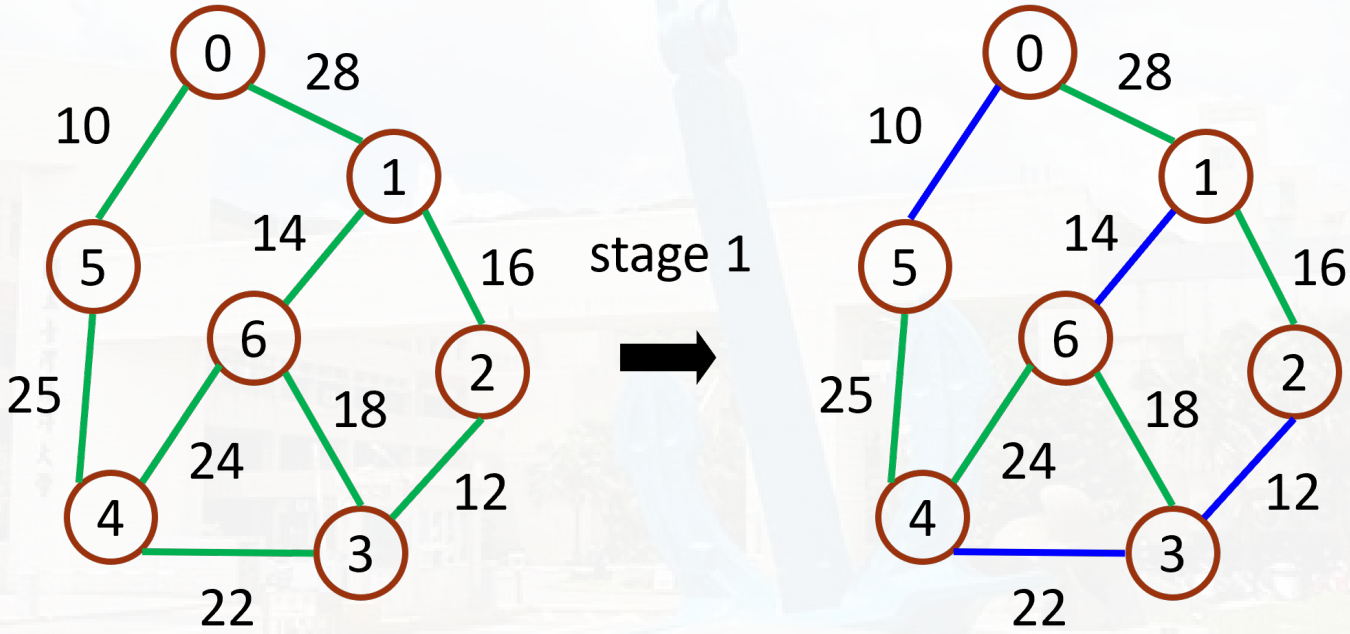
Sollin's/ Borůvka's Algorithm (2/2)

- **Note:** two trees in the forest could select the same edge.
  - We need to eliminate duplicate edges.
- At the start of the first stage the set of selected edges is empty.
- The algorithm terminates when there is only one tree at the end of a stage or no edges remain for selection.

We can implement Sollin's algorithm in $O(e \cdot \alpha(n) \log v)$ time (WHY?).

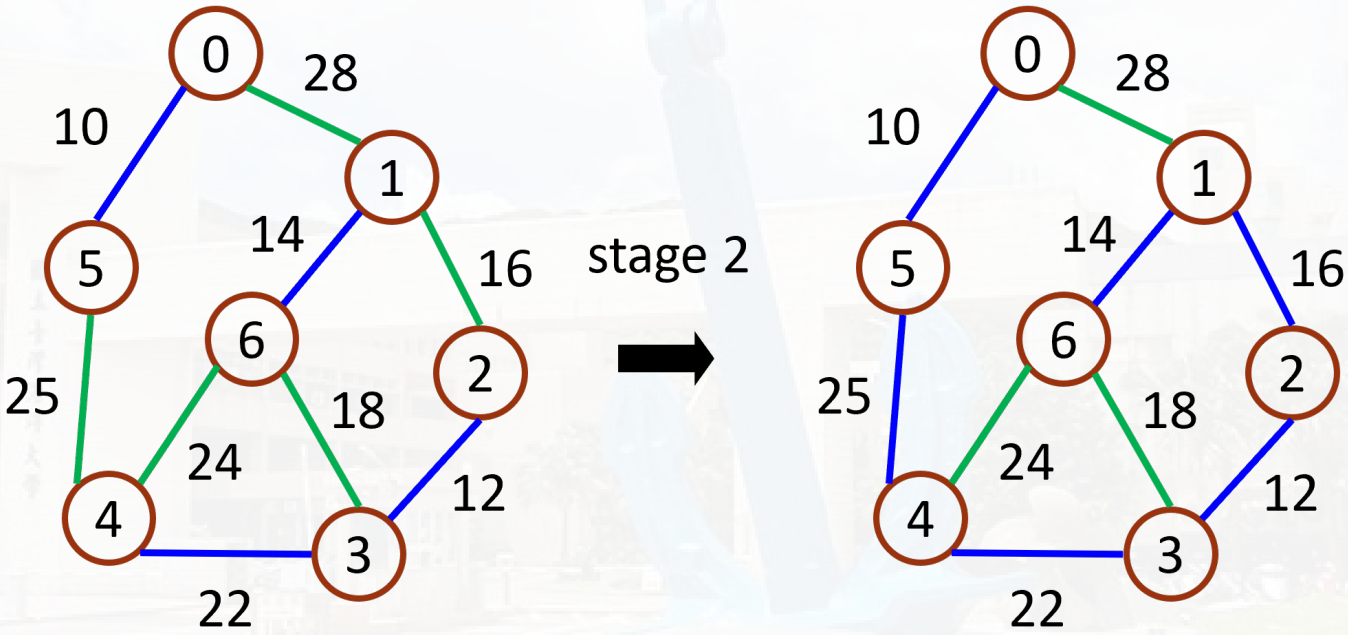
  - $\alpha(n)$: the inverse Ackermann function.

Illustration of Sollin's/ Borůvka's Algorithm



- Stage 1: $(0,5), (1,6), (2,3), (3,2), (4,3), (5,0)$, and $(6,1)$ are selected, and then duplicates are removed.

Illustration of Sollin's/ Borůvka's Algorithm



- Stage 2: $(5, 4), (1, 2)$, and $(2, 1)$ are selected, and then duplicates are removed.

Borůvka-Sollin-MST (Use **Union-Find** structure to maintain components)

```
for each v in V: make_set(v) # initialization
MST = []
repeat until there is only one component: # stages
    for each component C:
        cheapest[C] = NULL  # initialization
    for each edge (u,v) in E with weight w(u,v): # update the cheapest edges
        Cu = find_set(u); Cv = find_set(v); # find the trees u and v locate
        if Cu == Cv: continue # u and v are in the same tree, skipped
        if cheapest[Cu] = NULL or w(u,v) < w(cheapest[Cu]):
            cheapest[Cu] = (u,v)
        if cheapest[Cv] = NULL or w(u,v) < w(cheapest[Cv]):
            cheapest[Cv] = (u,v)
    for each component C:
        if cheapest[C] != NULL:
            (u,v) = cheapest[C]
            Cu = find_set(u); Cv = find_set(v);
            if Cu != Cv: # u and v belong two different components
                MST = MST + {(u,v)} # add (u,v) into MST
                union(Cu, Cv) # combine the two trees
return MST
```

Complexity: Number of Stages

Key observation

In each phase, every component selects at least one outgoing edge.

- Suppose at the beginning of a stage, we have k components.
- Each component chooses at least one edge to another component.
- Components merge in the best case by "pairing-up", so the number of components drops to at most $\lceil \frac{k}{2} \rceil$.
- Starting from $k_0 = n$ components, after t stages we have at most $n/2^t$ components.
- We stop when the number of components reaches 1, which happens after $t = O(\lg n)$ stages.

Total running time

- There are $O(\log n)$ stages.
- Each stage has cost $O(e\,\alpha(n))$.
- Total time complexity of $O\big(e\,\alpha(n)\,\log n\big)$.
- Since $\alpha(n) \leq 4$ for all practical n, this is often informally written as $O(e \log n)$.

Notes on $\alpha(n)$

$\alpha(n) = O(\lg^* n)$, where $\lg^* n := \min\{k \geq 0 : \underbrace{\lg \lg \ldots \lg}_{k \text{ times}}(n) \leq 1\}$.

Key to the Correctness

Cut Property (Key Tool for MST Algorithms)

A *cut* of G is a partition of the vertex set into two nonempty parts:

$$(S, V \setminus S), \quad \emptyset \subset S \subset V.$$

An edge $e = (u, v)$ *crosses* the cut if $u \in S$ and $v \in V \setminus S$ (or vice versa).

Cut Property

Let $(S, V \setminus S)$ be any cut, and let

$$e^\star = \arg\min\{\, w(e) : e \text{ crosses the cut} \,\}$$

be the (unique) minimum-weight edge crossing the cut. Then $e^\star$ belongs to *every* MST of G.

Outline

Cut Property (Stronger Version with Distinct Weights)

Distinct Edge Weights

Suppose all edge weights are distinct. Then each cut $(S, V \setminus S)$ has a *unique* minimum-weight edge crossing it.

Cut Property (Unique-Minimum Version)

Assume all edge weights are distinct. Let $(S, V \setminus S)$ be any cut, and let e be the unique minimum-weight edge crossing that cut. Then:

Every MST $T^\star$ of G contains e.

Exchange Argument: Setup

Setup

- Let $G = (V, E)$ be a connected, weighted graph.
- Fix a cut $(S, V \setminus S)$.
- Let $e = (u, v)$ be a minimum-weight edge crossing this cut:

$$u \in S, \quad v \in V \setminus S,$$

  and $w(e) \leq w(f)$ for every edge f crossing the cut.
- Let T be an MST of G (an arbitrary one).

Goal

Show: there exists an MST T' such that $e \in T'$.

Exchange Argument: Adding e and Creating a Cycle

Case 1: $e \in T$

If e already belongs to T, then T itself is an MST containing e. We are done.

Case 2: $e \notin T$

- Consider the graph $T \cup \{e\}$.
- Since T is a tree, adding one edge e creates exactly one simple cycle C.
- This cycle C contains e, plus a unique simple path in T connecting u and v.

Key Observation

The cycle C must contain at least one other edge $f \in T$ that also crosses the cut $(S, V \setminus S)$.

Why the Cycle Contains Another Crossing Edge

- Edge $e = (u, v)$ crosses the cut:

$$u \in S, \quad v \in V \setminus S.$$

- Walk around the cycle C, starting from u.
  - Each time we traverse an edge that crosses the cut, we move from S to $V \setminus S$ or vice versa.
  - To return to u (which is in S), the number of crossings of the cut along the cycle must be even.
- Since e is one crossing edge in C, there must be at least one *other* edge f in C that crosses the same cut.
- This edge f belongs to T (because $C \subseteq T \cup \{e\}$ and $f \neq e$).

Conclusion

There exists $f \in T$ on the cycle C such that f crosses $(S, V \setminus S)$ and $f \neq e$.

Exchange Argument: Replacing f by e

Weight Comparison

- Both e and f cross the same cut $(S, V \setminus S)$.
- e is chosen as an edge of minimum-weight crossing this cut.
- Therefore, $w(e) \leq w(f)$.

Construct a New Spanning Tree

Define $T' = T - \{f\} \cup \{e\}$.

- Removing f breaks the cycle C, so T' is acyclic.
- Adding e keeps the graph connected (it reconnects the two components formed by removing f).
- Thus T' is a spanning tree.

Its total weight: $w(T') = w(T) - w(f) + w(e) \leq w(T)$.

Complexity: Cost per Stage and Overall

Cost per stage

- Use adjacency lists and a Union-Find structure.
- Initializing `cheapest`$[\cdot]$: $O(n)$.
- Scanning all edges $(u, v) \in E$:
  - Each scan does a constant number of `find` operations.
  - With weighted union (union-by-rank) and path compression, each find/union is $O(\alpha(n))$ amortized, where $\alpha(\cdot)$ is the inverse Ackermann function.
- Therefore, scanning all edges takes $O(e\,\alpha(n))$ time.
- Merging components with `union` also costs $O(n\,\alpha(n))$ in total per stage.
- Since $e \geq n - 1$ in a connected graph, the stage cost is $O(e\,\alpha(n))$.

Discussions