Trees

Definition 2.1

- A graph having no cycles is acyclic or a forest.

Trees

Definition 2.1

- A graph having no cycles is acyclic or a forest.
- A connected forest is a tree.

Trees

Definition 2.1

- A graph having no cycles is acyclic or a forest.
- A connected forest is a tree.
- A leaf or a pendant vertex is a vertex of degree one.

Trees

Definition 2.1

- A graph having no cycles is acyclic or a forest.
- A connected forest is a tree.
- A leaf or a pendant vertex is a vertex of degree one.
- A subgraph of G is spanning if it has all the vertices of G.

Trees

Definition 2.1

- A graph having no cycles is acyclic or a forest.
- A connected forest is a tree.
- A leaf or a pendant vertex is a vertex of degree one.
- A subgraph of G is spanning if it has all the vertices of G.
- The distance between vertices u and v of G, written $d(u, v)$ or $d_G(u, v)$, is the length of the shortest path in G that contains both u and v. (Such a path is called a uv-path and u and v are its ends.)

Trees

Definition 2.1

- A graph having no cycles is acyclic or a forest.
- A connected forest is a tree.
- A leaf or a pendant vertex is a vertex of degree one.
- A subgraph of G is spanning if it has all the vertices of G.
- The distance between vertices u and v of G, written $d(u,v)$ or $d_G(u,v)$, is the length of the shortest path in G that contains both u and v. (Such a path is called a uv-path and u and v are its ends.) If a uv-path does not exist, then $d(u,v) = \infty$.

Definition 2.1

- A graph having no cycles is acyclic or a forest.
- A connected forest is a tree.
- A leaf or a pendant vertex is a vertex of degree one.
- A subgraph of G is spanning if it has all the vertices of G.
- The distance between vertices u and v of G, written $d(u, v)$ or $d_G(u, v)$, is the length of the shortest path in G that contains both u and v. (Such a path is called a uv-path and u and v are its ends.) If a uv-path does not exist, then $d(u, v) = \infty$.
- The distance between sets U and W of vertices of G, written $d(U, W)$, is the length of a shortest uw-path where $u \in U$ and $w \in W$, or infinity if no such path exists.

Induction for Trees

Theorem 2.2

Every tree with at least two vertices has at least two leaves.

Induction for Trees

Theorem 2.2

Every tree with at least two vertices has at least two leaves. Deleting a leaf from a tree of order n produces a tree of order $n-1$.

Induction for Trees

Theorem 2.2

Every tree with at least two vertices has at least two leaves. Deleting a leaf from a tree of order n produces a tree of order $n-1$.

Proof.

In an acyclic graph, the ends of a maximal non-trivial path have degree one. $\square$

Induction for Trees

Theorem 2.2

Every tree with at least two vertices has at least two leaves. Deleting a leaf from a tree of order n produces a tree of order $n-1$.

Proof.

In an acyclic graph, the ends of a maximal non-trivial path have degree one. Let v be a leaf of a tree T and let $T' = T - v$.

$\square$

Induction for Trees

Theorem 2.2

Every tree with at least two vertices has at least two leaves. Deleting a leaf from a tree of order n produces a tree of order $n - 1$.

Proof.

In an acyclic graph, the ends of a maximal non-trivial path have degree one.
Let v be a leaf of a tree T and let $T' = T - v$.
Then T' is acyclic.

$\square$

Induction for Trees

Theorem 2.2

Every tree with at least two vertices has at least two leaves. Deleting a leaf from a tree of order n produces a tree of order $n - 1$.

Proof.

In an acyclic graph, the ends of a maximal non-trivial path have degree one.
Let v be a leaf of a tree T and let $T' = T - v$.
Then T' is acyclic.
Suppose u and w are vertices of T'. Then, in T there is a uw-path P.

$\square$

Induction for Trees

Theorem 2.2

Every tree with at least two vertices has at least two leaves. Deleting a leaf from a tree of order n produces a tree of order $n-1$.

Proof.

In an acyclic graph, the ends of a maximal non-trivial path have degree one.
Let v be a leaf of a tree T and let $T' = T - v$.
Then T' is acyclic.
Suppose u and w are vertices of T'. Then, in T there is a uw-path P.
But P cannot contain v as $d_T(v) = 1$, and so it also lies in T'. $\square$

Characterization of Trees

Theorem 2.3

For a simple graph G of order n the following are equivalent:

(A) G *is connected and acyclic;*

Characterization of Trees

Theorem 2.3

For a simple graph G of order n the following are equivalent:

(A) G *is connected and acyclic;*

(B) G *is connected and has size* $n-1$*;*

Characterization of Trees

Theorem 2.3

For a simple graph G of order n the following are equivalent:

(A) *G is connected and acyclic;*

(B) *G is connected and has size $n-1$;*

(C) *G is acyclic and has size $n-1$; and*

Characterization of Trees

Theorem 2.3

For a simple graph G of order n the following are equivalent:

(A) *G is connected and acyclic;*

(B) *G is connected and has size $n-1$;*

(C) *G is acyclic and has size $n-1$; and*

(D) *For every two vertices u and v, the graph G contains exactly one uv-path.*

Edge Exchange

Theorem 2.4

If T and T' are two spanning trees of a connected graph G and $e \in E(T) \setminus E(T')$, then there is an edge $e' \in E(T') \setminus E(T)$ such that $T \setminus e \cup e'$ is a spanning tree of G.

Edge Exchange

Theorem 2.4

If T and T' are two spanning trees of a connected graph G and $e \in E(T) \setminus E(T')$, then there is an edge $e' \in E(T') \setminus E(T)$ such that $T \setminus e \cup e'$ is a spanning tree of G.

Proof.

Consider $T \setminus e$: it is disconnected with exactly two connected components (maximal connected subgraphs) S and S'. $\square$

Edge Exchange

Theorem 2.4

If T and T' are two spanning trees of a connected graph G and $e \in E(T) \setminus E(T')$, then there is an edge $e' \in E(T') \setminus E(T)$ such that $T \setminus e \cup e'$ is a spanning tree of G.

Proof.

Consider $T \setminus e$: it is disconnected with exactly two connected components (maximal connected subgraphs) S and S'. Since T' is connected, it must have an edge e' with one endpoint in each S and S'. $\square$

Edge Exchange

Theorem 2.4

If T and T' are two spanning trees of a connected graph G and $e \in E(T) \setminus E(T')$, then there is an edge $e' \in E(T') \setminus E(T)$ such that $T \setminus e \cup e'$ is a spanning tree of G.

Proof.

Consider $T \setminus e$: it is disconnected with exactly two connected components (maximal connected subgraphs) S and S'. Since T' is connected, it must have an edge e' with one endpoint in each S and S'. Clearly, $T \setminus e \cup e'$ is a spanning tree of G. $\square$

Minimum Cost Spanning Tree

Suppose G is a graph and $c : E(G) \to \mathbb{N}$ is a cost function.

Minimum Cost Spanning Tree

Suppose G is a graph and $c : E(G) \to \mathbb{N}$ is a cost function. The cost of a subgraph H of G is $\sum_{e \in E(H)} c(e)$.

Minimum Cost Spanning Tree

Suppose G is a graph and $c : E(G) \to \mathbb{N}$ is a cost function. The cost of a subgraph H of G is $\sum_{e \in E(H)} c(e)$. We want to find a minimum-cost spanning tree T of G.

Minimum Cost Spanning Tree

Suppose G is a graph and $c : E(G) \to \mathbb{N}$ is a cost function. The cost of a subgraph H of G is $\sum_{e \in E(H)} c(e)$. We want to find a minimum-cost spanning tree T of G.

Algorithm 2.5 (Kruskal)

- *Start with* $V(T) = V(G)$ *and* $E(T) = \emptyset$.

Minimum Cost Spanning Tree

Suppose G is a graph and $c : E(G) \to \mathbb{N}$ is a cost function. The cost of a subgraph H of G is $\sum_{e \in E(H)} c(e)$. We want to find a minimum-cost spanning tree T of G.

Algorithm 2.5 (Kruskal)

- *Start with $V(T) = V(G)$ and $E(T) = \emptyset$.*
- *Order the edges of G so that their costs are non-decreasing.*

Minimum Cost Spanning Tree

Suppose G is a graph and $c : E(G) \to \mathbb{N}$ is a cost function. The cost of a subgraph H of G is $\sum_{e \in E(H)} c(e)$. We want to find a minimum-cost spanning tree T of G.

Algorithm 2.5 (Kruskal)

- *Start with* $V(T) = V(G)$ *and* $E(T) = \emptyset$.
- *Order the edges of* G *so that their costs are non-decreasing.*
- *Proceed with each edge of* G, *one by one, in the above order: if its joins two components of* T, *add it to* T; *otherwise do nothing.*

Minimum Cost Spanning Tree

Suppose G is a graph and $c : E(G) \to \mathbb{N}$ is a cost function. The cost of a subgraph H of G is $\sum_{e \in E(H)} c(e)$. We want to find a minimum-cost spanning tree T of G.

Algorithm 2.5 (Kruskal)

- *Start with* $V(T) = V(G)$ *and* $E(T) = \emptyset$.
- *Order the edges of* G *so that their costs are non-decreasing.*
- *Proceed with each edge of* G*, one by one, in the above order: if its joins two components of* T*, add it to* T*; otherwise do nothing.*

Example 2.6

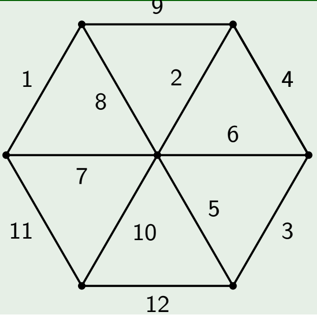
Minimum Cost Spanning Tree

Suppose G is a graph and $c : E(G) \to \mathbb{N}$ is a cost function. The cost of a subgraph H of G is $\sum_{e \in E(H)} c(e)$. We want to find a minimum-cost spanning tree T of G.

Algorithm 2.5 (Kruskal)

- *Start with $V(T) = V(G)$ and $E(T) = \emptyset$.*
- *Order the edges of G so that their costs are non-decreasing.*
- *Proceed with each edge of G, one by one, in the above order: if its joins two components of T, add it to T; otherwise do nothing.*

Example 2.6



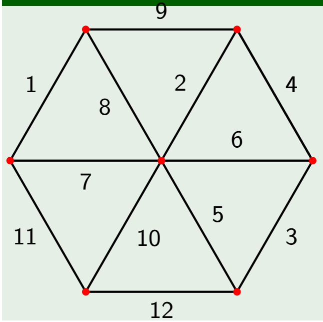
Minimum Cost Spanning Tree

Suppose G is a graph and $c : E(G) \to \mathbb{N}$ is a cost function. The cost of a subgraph H of G is $\sum_{e \in E(H)} c(e)$. We want to find a minimum-cost spanning tree T of G.

Algorithm 2.5 (Kruskal)

- *Start with* $V(T) = V(G)$ *and* $E(T) = \emptyset$.
- *Order the edges of* G *so that their costs are non-decreasing.*
- *Proceed with each edge of* G*, one by one, in the above order: if its joins two components of* T*, add it to* T*; otherwise do nothing.*

Example 2.6



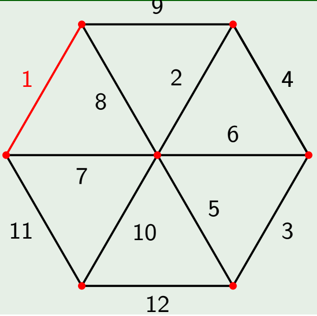
Minimum Cost Spanning Tree

Suppose G is a graph and $c : E(G) \to \mathbb{N}$ is a cost function. The cost of a subgraph H of G is $\sum_{e \in E(H)} c(e)$. We want to find a minimum-cost spanning tree T of G.

Algorithm 2.5 (Kruskal)

- *Start with $V(T) = V(G)$ and $E(T) = \emptyset$.*
- *Order the edges of G so that their costs are non-decreasing.*
- *Proceed with each edge of G, one by one, in the above order: if its joins two components of T, add it to T; otherwise do nothing.*

Example 2.6



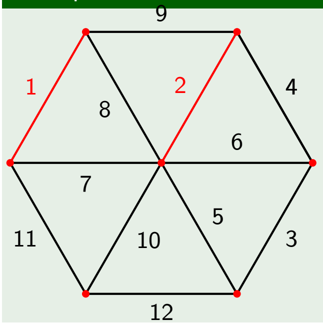
Minimum Cost Spanning Tree

Suppose G is a graph and $c : E(G) \to \mathbb{N}$ is a cost function. The cost of a subgraph H of G is $\sum_{e \in E(H)} c(e)$. We want to find a minimum-cost spanning tree T of G.

Algorithm 2.5 (Kruskal)

- *Start with $V(T) = V(G)$ and $E(T) = \emptyset$.*
- *Order the edges of G so that their costs are non-decreasing.*
- *Proceed with each edge of G, one by one, in the above order: if its joins two components of T, add it to T; otherwise do nothing.*

Example 2.6



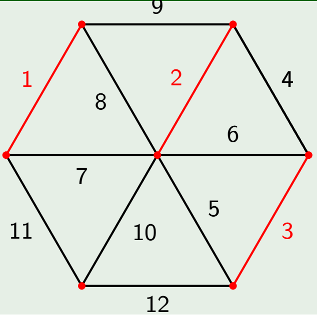
Minimum Cost Spanning Tree

Suppose G is a graph and $c : E(G) \to \mathbb{N}$ is a cost function. The cost of a subgraph H of G is $\sum_{e \in E(H)} c(e)$. We want to find a minimum-cost spanning tree T of G.

Algorithm 2.5 (Kruskal)

- *Start with* $V(T) = V(G)$ *and* $E(T) = \emptyset$.
- *Order the edges of* G *so that their costs are non-decreasing.*
- *Proceed with each edge of* G*, one by one, in the above order: if its joins two components of* T*, add it to* T*; otherwise do nothing.*

Example 2.6



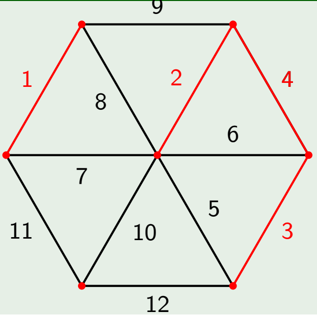
Minimum Cost Spanning Tree

Suppose G is a graph and $c : E(G) \to \mathbb{N}$ is a cost function. The cost of a subgraph H of G is $\sum_{e \in E(H)} c(e)$. We want to find a minimum-cost spanning tree T of G.

Algorithm 2.5 (Kruskal)

- *Start with* $V(T) = V(G)$ *and* $E(T) = \emptyset$.
- *Order the edges of* G *so that their costs are non-decreasing.*
- *Proceed with each edge of* G*, one by one, in the above order: if its joins two components of* T*, add it to* T*; otherwise do nothing.*

Example 2.6



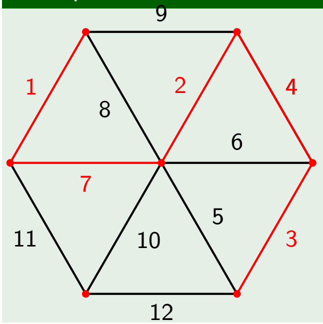
Minimum Cost Spanning Tree

Suppose G is a graph and $c : E(G) \to \mathbb{N}$ is a cost function. The cost of a subgraph H of G is $\sum_{e\in E(H)} c(e)$. We want to find a minimum-cost spanning tree T of G.

Algorithm 2.5 (Kruskal)

- *Start with $V(T) = V(G)$ and $E(T) = \emptyset$.*
- *Order the edges of G so that their costs are non-decreasing.*
- *Proceed with each edge of G, one by one, in the above order: if its joins two components of T, add it to T; otherwise do nothing.*

Example 2.6



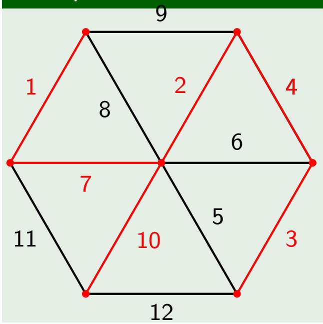
Minimum Cost Spanning Tree

Suppose G is a graph and $c : E(G) \to \mathbb{N}$ is a cost function. The cost of a subgraph H of G is $\sum_{e \in E(H)} c(e)$. We want to find a minimum-cost spanning tree T of G.

Algorithm 2.5 (Kruskal)

- *Start with $V(T) = V(G)$ and $E(T) = \emptyset$.*
- *Order the edges of G so that their costs are non-decreasing.*
- *Proceed with each edge of G, one by one, in the above order: if its joins two components of T, add it to T; otherwise do nothing.*

Example 2.6



Proof of Kruskal's Theorem

Theorem 2.7 (Kruskal)

In a connected graph, Kruskal's Algorithm produces a minimum-cost spanning tree.

Proof of Kruskal's Theorem

Theorem 2.7 (Kruskal)

In a connected graph, Kruskal's Algorithm produces a minimum-cost spanning tree.

Proof.

It is clear that the algorithm produces a spanning tree.

□

Proof of Kruskal's Theorem

Theorem 2.7 (Kruskal)

In a connected graph, Kruskal's Algorithm produces a minimum-cost spanning tree.

Proof.

It is clear that the algorithm produces a spanning tree.
Let T be the resulting graph, and suppose T' is a spanning tree of minimum cost.

□

Proof of Kruskal's Theorem

Theorem 2.7 (Kruskal)

In a connected graph, Kruskal's Algorithm produces a minimum-cost spanning tree.

Proof.

It is clear that the algorithm produces a spanning tree.
Let T be the resulting graph, and suppose T' is a spanning tree of minimum cost. If $T' = T$, then there is nothing to prove.

□

Proof of Kruskal's Theorem

Theorem 2.7 (Kruskal)

In a connected graph, Kruskal's Algorithm produces a minimum-cost spanning tree.

Proof.

It is clear that the algorithm produces a spanning tree.
Let T be the resulting graph, and suppose T' is a spanning tree of minimum cost. If $T' = T$, then there is nothing to prove. If $T \neq T'$, let e be the first edge chosen for T that is not in T'.

□

Proof of Kruskal's Theorem

Theorem 2.7 (Kruskal)

In a connected graph, Kruskal's Algorithm produces a minimum-cost spanning tree.

Proof.

It is clear that the algorithm produces a spanning tree.
Let T be the resulting graph, and suppose T' is a spanning tree of minimum cost. If $T' = T$, then there is nothing to prove. If $T \neq T'$, let e be the first edge chosen for T that is not in T'. Adding e to T' creates a cycle C, but since T does not have cycles, T' has an edge $e' \notin E(T)$.

$\square$

Proof of Kruskal's Theorem

Theorem 2.7 (Kruskal)

In a connected graph, Kruskal's Algorithm produces a minimum-cost spanning tree.

Proof.

It is clear that the algorithm produces a spanning tree.
Let T be the resulting graph, and suppose T' is a spanning tree of minimum cost. If $T' = T$, then there is nothing to prove. If $T \neq T'$, let e be the first edge chosen for T that is not in T'. Adding e to T' creates a cycle C, but since T does not have cycles, T' has an edge $e' \notin E(T)$. Consider the spanning tree $T' \setminus e' \cup e$.

$\square$

Proof of Kruskal's Theorem

Theorem 2.7 (Kruskal)

In a connected graph, Kruskal's Algorithm produces a minimum-cost spanning tree.

Proof.

It is clear that the algorithm produces a spanning tree.
Let T be the resulting graph, and suppose T' is a spanning tree of minimum cost. If $T' = T$, then there is nothing to prove. If $T \neq T'$, let e be the first edge chosen for T that is not in T'. Adding e to T' creates a cycle C, but since T does not have cycles, T' has an edge $e' \notin E(T)$. Consider the spanning tree $T' \setminus e' \cup e$.
Since T' contains e' and all edges of T chosen before e, both e and e' are available when the algorithm chooses e, and hence $c(e) \leq c(e')$. $\square$

Proof of Kruskal's Theorem

Theorem 2.7 (Kruskal)

In a connected graph, Kruskal's Algorithm produces a minimum-cost spanning tree.

Proof.

It is clear that the algorithm produces a spanning tree.
Let T be the resulting graph, and suppose T' is a spanning tree of minimum cost. If $T' = T$, then there is nothing to prove. If $T \neq T'$, let e be the first edge chosen for T that is not in T'. Adding e to T' creates a cycle C, but since T does not have cycles, T' has an edge $e' \notin E(T)$. Consider the spanning tree $T' \setminus e' \cup e$.
Since T' contains e' and all edges of T chosen before e, both e and e' are available when the algorithm chooses e, and hence $c(e) \leq c(e')$. Thus $T' \setminus e' \cup e$ is a spanning tree with cost at most T' that agrees with T for a longer initial list of edges than T' does. $\square$

Proof of Kruskal's Theorem

Theorem 2.7 (Kruskal)

In a connected graph, Kruskal's Algorithm produces a minimum-cost spanning tree.

Proof.

It is clear that the algorithm produces a spanning tree.
Let T be the resulting graph, and suppose T' is a spanning tree of minimum cost. If $T' = T$, then there is nothing to prove. If $T \neq T'$, let e be the first edge chosen for T that is not in T'. Adding e to T' creates a cycle C, but since T does not have cycles, T' has an edge $e' \notin E(T)$. Consider the spanning tree $T' \setminus e' \cup e$.
Since T' contains e' and all edges of T chosen before e, both e and e' are available when the algorithm chooses e, and hence $c(e) \leq c(e')$. Thus $T' \setminus e' \cup e$ is a spanning tree with cost at most T' that agrees with T for a longer initial list of edges than T' does. Repeating this argument yields a minimum-cost spanning tree that equals T, proving that the costs of T and T' are the same. $\square$

Enumerating Labeled Trees

We would like to know how many different (and here we really mean different rather than non-isomorphic) trees with the vertex set $\{1, 2, \ldots, n\}$ are there?

Enumerating Labeled Trees

We would like to know how many different (and here we really mean different rather than non-isomorphic) trees with the vertex set $\{1, 2, \ldots, n\}$ are there?

Theorem 2.8 (Cayley's Formula)

There are n^{n-2} *trees with vertex set* $\{1, 2, \ldots, n\}$.

Enumerating Labeled Trees

We would like to know how many different (and here we really mean different rather than non-isomorphic) trees with the vertex set $\{1, 2, \ldots, n\}$ are there?

Theorem 2.8 (Cayley's Formula due to Borchardt (1860))

There are n^{n-2} trees with vertex set $\{1, 2, \ldots, n\}$.

Enumerating Labeled Trees

We would like to know how many different (and here we really mean different rather than non-isomorphic) trees with the vertex set $\{1, 2, \ldots, n\}$ are there?

Theorem 2.8 (Cayley's Formula due to Borchardt (1860))

There are n^{n-2} *trees with vertex set* $\{1, 2, \ldots, n\}$.

Proof.

There are n^{n-2} sequences of length $n - 2$ with entries from $\{1, 2, \ldots, n\}$. $\square$

Enumerating Labeled Trees

We would like to know how many different (and here we really mean different rather than non-isomorphic) trees with the vertex set $\{1, 2, \ldots, n\}$ are there?

Theorem 2.8 (Cayley's Formula due to Borchardt (1860))

There are n^{n-2} trees with vertex set $\{1, 2, \ldots, n\}$.

Proof.

There are n^{n-2} sequences of length $n - 2$ with entries from $\{1, 2, \ldots, n\}$. We will establish a bijection between such sequences and trees on the vertex set $\{1, 2, \ldots, n\}$. $\square$

Prüfer Sequences

To find a Prüfer sequence $f(T)$ of a labeled tree T,

Prüfer Sequences

To find a Prüfer sequence $f(T)$ of a labeled tree T,

- delete the leaf with the smallest label, and

Prüfer Sequences

To find a Prüfer sequence $f(T)$ of a labeled tree T,

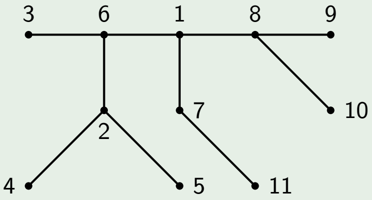
- delete the leaf with the smallest label, and
- append the label of its neighbor to the sequence until one edge remains.

Prüfer Sequences

To find a Prüfer sequence $f(T)$ of a labeled tree T,

- delete the leaf with the smallest label, and
- append the label of its neighbor to the sequence until one edge remains.

Example 2.9



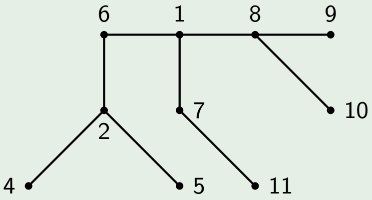
Prüfer sequence:

Prüfer Sequences

To find a Prüfer sequence $f(T)$ of a labeled tree T,

- delete the leaf with the smallest label, and
- append the label of its neighbor to the sequence until one edge remains.

Example 2.9



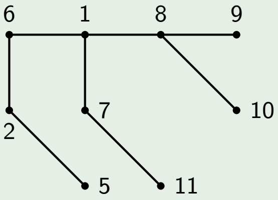
Prüfer sequence: 6

Prüfer Sequences

To find a Prüfer sequence $f(T)$ of a labeled tree T,

- delete the leaf with the smallest label, and
- append the label of its neighbor to the sequence until one edge remains.

Example 2.9



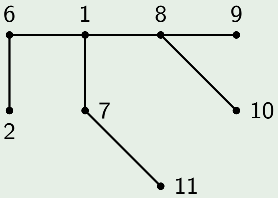
Prüfer sequence: 6 , 2

Prüfer Sequences

To find a Prüfer sequence $f(T)$ of a labeled tree T,

- delete the leaf with the smallest label, and
- append the label of its neighbor to the sequence until one edge remains.

Example 2.9



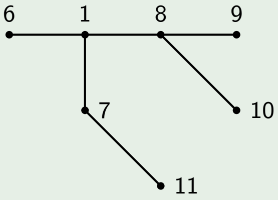
Prüfer sequence: 6 , 2 , 2

Prüfer Sequences

To find a Prüfer sequence $f(T)$ of a labeled tree T,

- delete the leaf with the smallest label, and
- append the label of its neighbor to the sequence until one edge remains.

Example 2.9



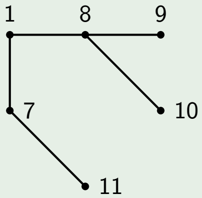
Prüfer sequence: 6 , 2 , 2 , 6

Prüfer Sequences

To find a Prüfer sequence $f(T)$ of a labeled tree T,

- delete the leaf with the smallest label, and
- append the label of its neighbor to the sequence until one edge remains.

Example 2.9



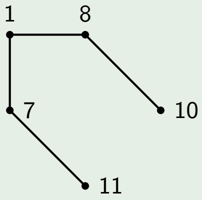
Prüfer sequence: 6 , 2 , 2 , 6 , 1

Prüfer Sequences

To find a Prüfer sequence $f(T)$ of a labeled tree T,

- delete the leaf with the smallest label, and
- append the label of its neighbor to the sequence until one edge remains.

Example 2.9



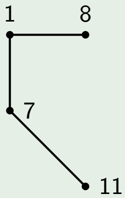
Prüfer sequence: 6 , 2 , 2 , 6 , 1 , 8

Prüfer Sequences

To find a Prüfer sequence $f(T)$ of a labeled tree T,

- delete the leaf with the smallest label, and
- append the label of its neighbor to the sequence until one edge remains.

Example 2.9



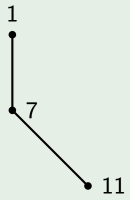
Prüfer sequence: 6 , 2 , 2 , 6 , 1 , 8 , 8

Prüfer Sequences

To find a Prüfer sequence $f(T)$ of a labeled tree T,

- delete the leaf with the smallest label, and
- append the label of its neighbor to the sequence until one edge remains.

Example 2.9



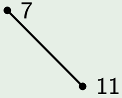
Prüfer sequence: 6 , 2 , 2 , 6 , 1 , 8 , 8 , 1

Prüfer Sequences

To find a Prüfer sequence $f(T)$ of a labeled tree T,

- delete the leaf with the smallest label, and
- append the label of its neighbor to the sequence until one edge remains.

Example 2.9



Prüfer sequence: 6 , 2 , 2 , 6 , 1 , 8 , 8 , 1 , 7

Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, $\ldots$, n.

Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, . . . , n.
- Proceed with all $n - 2$ elements of the sequence, and, at the ith step,

Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, . . . , n.
- Proceed with all $n - 2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.

Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, . . . , n.
- Proceed with all $n - 2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.
  - let y be the smallest label that does not appear at the ith or later position and has not yet been marked as "finished".

Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, ..., n.
- Proceed with all $n-2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.
  - let y be the smallest label that does not appear at the ith or later position and has not yet been marked as "finished".
  - add the edge xy, and

Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, . . . , n.
- Proceed with all $n - 2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.
  - let y be the smallest label that does not appear at the ith or later position and has not yet been marked as "finished".
  - add the edge xy, and
  - mark y as finished.

Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, . . . , n.
- Proceed with all $n - 2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.
  - let y be the smallest label that does not appear at the ith or later position and has not yet been marked as "finished".
  - add the edge xy, and
  - mark y as finished.
- Join the two remaining unfinished vertices with an edge.

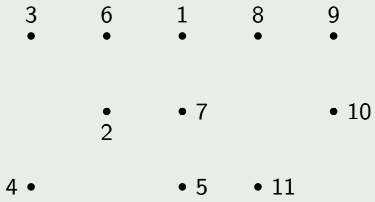
Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, $\ldots$, n.
- Proceed with all $n-2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.
  - let y be the smallest label that does not appear at the ith or later position and has not yet been marked as "finished".
  - add the edge xy, and
  - mark y as finished.
- Join the two remaining unfinished vertices with an edge.

Example 2.10

Sequence: 6, 2, 2, 6, 1, 8, 8, 1, 7 Finished:



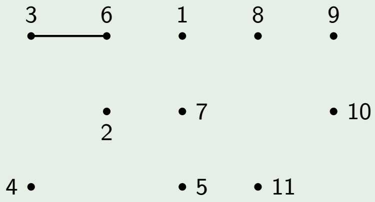
Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, . . . , n.
- Proceed with all $n-2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.
  - let y be the smallest label that does not appear at the ith or later position and has not yet been marked as “finished”.
  - add the edge xy, and
  - mark y as finished.
- Join the two remaining unfinished vertices with an edge.

Example 2.10

Sequence: 6, 2, 2, 6, 1, 8, 8, 1, 7 Finished: 3



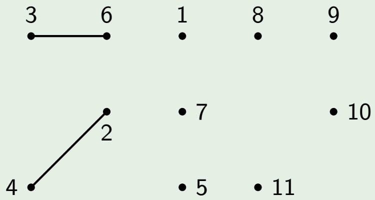
Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, ..., n.
- Proceed with all $n-2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.
  - let y be the smallest label that does not appear at the ith or later position and has not yet been marked as "finished".
  - add the edge xy, and
  - mark y as finished.
- Join the two remaining unfinished vertices with an edge.

Example 2.10

Sequence: 6, 2, 2, 6, 1, 8, 8, 1, 7 Finished: 3, 4



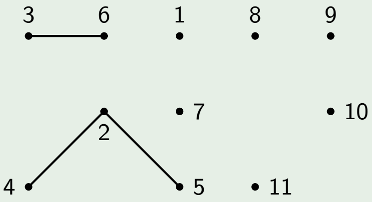
Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, $\ldots$, n.
- Proceed with all $n-2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.
  - let y be the smallest label that does not appear at the ith or later position and has not yet been marked as "finished".
  - add the edge xy, and
  - mark y as finished.
- Join the two remaining unfinished vertices with an edge.

Example 2.10

Sequence: 6, 2, 2, 6, 1, 8, 8, 1, 7 Finished: 3, 4, 5



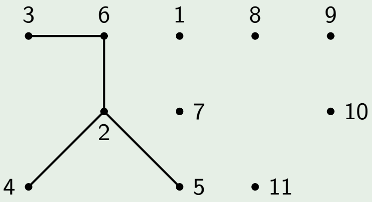
Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, $\ldots$, n.
- Proceed with all $n-2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.
  - let y be the smallest label that does not appear at the ith or later position and has not yet been marked as "finished".
  - add the edge xy, and
  - mark y as finished.
- Join the two remaining unfinished vertices with an edge.

Example 2.10

Sequence: 6, 2, 2, 6, 1, 8, 8, 1, 7   Finished: 3, 4, 5, 2



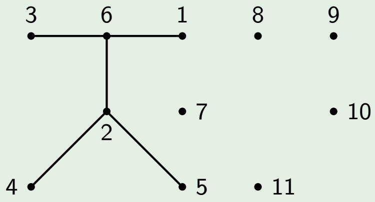
Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, . . . , n.
- Proceed with all $n - 2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.
  - let y be the smallest label that does not appear at the ith or later position and has not yet been marked as "finished".
  - add the edge xy, and
  - mark y as finished.
- Join the two remaining unfinished vertices with an edge.

Example 2.10

Sequence: 6, 2, 2, 6, 1, 8, 8, 1, 7 Finished: 3, 4, 5, 2, 6



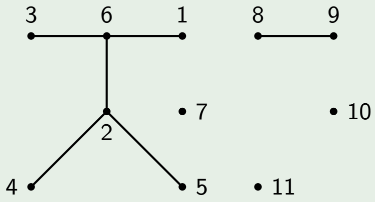
Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, . . . , n.
- Proceed with all $n-2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.
  - let y be the smallest label that does not appear at the ith or later position and has not yet been marked as "finished".
  - add the edge xy, and
  - mark y as finished.
- Join the two remaining unfinished vertices with an edge.

Example 2.10

Sequence: 6, 2, 2, 6, 1, 8, 8, 1, 7 Finished: 3, 4, 5, 2, 6, 9



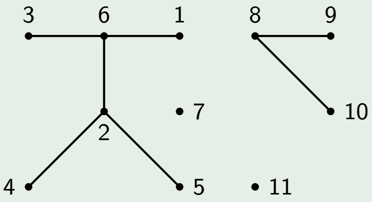
Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, . . . , n.
- Proceed with all $n - 2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.
  - let y be the smallest label that does not appear at the ith or later position and has not yet been marked as "finished".
  - add the edge xy, and
  - mark y as finished.
- Join the two remaining unfinished vertices with an edge.

Example 2.10

Sequence: 6, 2, 2, 6, 1, 8, 8, 1, 7 Finished: 3, 4, 5, 2, 6, 9, 10



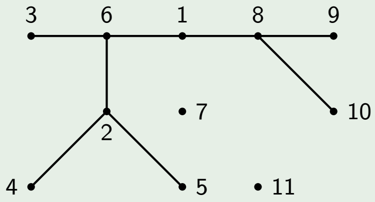
Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, ..., n.
- Proceed with all $n-2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.
  - let y be the smallest label that does not appear at the ith or later position and has not yet been marked as "finished".
  - add the edge xy, and
  - mark y as finished.
- Join the two remaining unfinished vertices with an edge.

Example 2.10

Sequence: 6, 2, 2, 6, 1, 8, 8, 1, 7   Finished: 3, 4, 5, 2, 6, 9, 10, 8



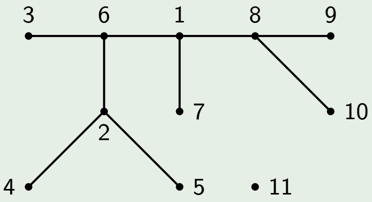
Trees from Sequences

Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, . . . , n.
- Proceed with all $n - 2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.
  - let y be the smallest label that does not appear at the ith or later position and has not yet been marked as "finished".
  - add the edge xy, and
  - mark y as finished.
- Join the two remaining unfinished vertices with an edge.

Example 2.10

Sequence: 6, 2, 2, 6, 1, 8, 8, 1, 7 Finished: 3, 4, 5, 2, 6, 9, 10, 8, 1



Trees from Sequences

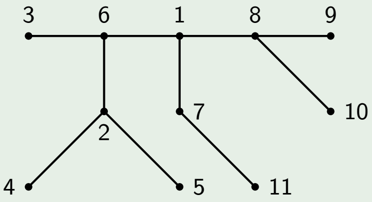
Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, ..., n.
- Proceed with all $n-2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.
  - let y be the smallest label that does not appear at the ith or later position and has not yet been marked as "finished".
  - add the edge xy, and
  - mark y as finished.
- Join the two remaining unfinished vertices with an edge.

Example 2.10

Sequence: 6, 2, 2, 6, 1, 8, 8, 1, 7

Finished: 3, 4, 5, 2, 6, 9, 10, 8, 1



Counting Trees with Prescribed Degrees

Corollary 2.11

The number of trees with vertex set $\{1, 2, \ldots, n\}$ *in which vertices* 1, 2, $\ldots$, n *have respective degrees* d_1, d_2, $\ldots$, d_n *is*

$$\frac{(n-2)!}{\prod (d_i - 1)!}.$$

Counting Trees with Prescribed Degrees

Corollary 2.11

The number of trees with vertex set $\{1, 2, \dots, n\}$ *in which vertices* 1, 2, $\dots$, n *have respective degrees* d_1, d_2, $\dots$, d_n *is*

$$\frac{(n-2)!}{\prod(d_i - 1)!}.$$

Proof.

When we delete vertex x from T when constructing the Prüfer sequence, all neighbors of x except for one have already been deleted.

$\square$

Counting Trees with Prescribed Degrees

Corollary 2.11

The number of trees with vertex set $\{1, 2, \ldots, n\}$ *in which vertices* 1, 2, $\ldots$, n *have respective degrees* d_1, d_2, $\ldots$, d_n *is*

$$\frac{(n-2)!}{\prod (d_i - 1)!}.$$

Proof.

When we delete vertex x from T when constructing the Prüfer sequence, all neighbors of x except for one have already been deleted. We record x in the sequence once for each deleted neighbor and x does not appear in the sequence again.

$\square$

Counting Trees with Prescribed Degrees

Corollary 2.11

The number of trees with vertex set $\{1, 2, \ldots, n\}$ *in which vertices* 1, 2, $\ldots$, n *have respective degrees* d_1, d_2, $\ldots$, d_n *is*

$$\frac{(n-2)!}{\prod (d_i - 1)!}.$$

Proof.

When we delete vertex x from T when constructing the Prüfer sequence, all neighbors of x except for one have already been deleted. We record x in the sequence once for each deleted neighbor and x does not appear in the sequence again. Hence x appears in the sequence $d(x) - 1$ times. $\square$

Counting Trees with Prescribed Degrees

Corollary 2.11

The number of trees with vertex set $\{1, 2, \ldots, n\}$ *in which vertices* 1, 2, $\ldots$, n *have respective degrees* d_1, d_2, $\ldots$, d_n *is*

$$\frac{(n-2)!}{\prod(d_i - 1)!}.$$

Proof.

When we delete vertex x from T when constructing the Prüfer sequence, all neighbors of x except for one have already been deleted. We record x in the sequence once for each deleted neighbor and x does not appear in the sequence again. Hence x appears in the sequence $d(x) - 1$ times.

Therefore we count the trees by counting sequences of length $n - 2$ having $d_i - 1$ copies of i, for each i. $\square$

Counting Trees with Prescribed Degrees

Corollary 2.11

The number of trees with vertex set $\{1, 2, \ldots, n\}$ in which vertices 1, 2, $\ldots$, n have respective degrees d_1, d_2, $\ldots$, d_n is

$$\frac{(n-2)!}{\prod(d_i - 1)!}.$$

Proof.

When we delete vertex x from T when constructing the Prüfer sequence, all neighbors of x except for one have already been deleted. We record x in the sequence once for each deleted neighbor and x does not appear in the sequence again. Hence x appears in the sequence $d(x) - 1$ times.
Therefore we count the trees by counting sequences of length $n-2$ having $d_i - 1$ copies of i, for each i. If we distinguish between various copies of i, then there are $(n-2)!$ such sequences. □

Counting Trees with Prescribed Degrees

Corollary 2.11

The number of trees with vertex set $\{1, 2, \ldots, n\}$ *in which vertices* 1, 2, $\ldots$, n *have respective degrees* d_1, d_2, $\ldots$, d_n *is*

$$\frac{(n-2)!}{\prod(d_i - 1)!}.$$

Proof.

When we delete vertex x from T when constructing the Prüfer sequence, all neighbors of x except for one have already been deleted. We record x in the sequence once for each deleted neighbor and x does not appear in the sequence again. Hence x appears in the sequence $d(x) - 1$ times.
Therefore we count the trees by counting sequences of length $n - 2$ having $d_i - 1$ copies of i, for each i. If we distinguish between various copies of i, then there are $(n-2)!$ such sequences. Since we really cannot distinguish between the copies, we have over-counted by a factor of $(d_i - 1)!$ for each i. □

Minors

Definition 2.12

- If e is an edge of G incident with two distinct vertices u and v, then the contraction of e is the operation of deleting e and identifying u and v.

Minors

Definition 2.12

- If e is an edge of G incident with two distinct vertices u and v, then the contraction of e is the operation of deleting e and identifying u and v.
- Contracting a loop is the same as deleting it.

Minors

Definition 2.12

- If e is an edge of G incident with two distinct vertices u and v, then the contraction of e is the operation of deleting e and identifying u and v.
- Contracting a loop is the same as deleting it.
- The graph obtained from G by contracting e is denoted G/e (extended to G/F if $F \subseteq E(G)$).

Minors

Definition 2.12

- If e is an edge of G incident with two distinct vertices u and v, then the contraction of e is the operation of deleting e and identifying u and v.
- Contracting a loop is the same as deleting it.
- The graph obtained from G by contracting e is denoted G/e (extended to G/F if $F \subseteq E(G)$).
- A graph H is a minor of G if it can be obtained from G by a sequence of operation each of which is one of the following:
  - deleting an edge;

Minors

Definition 2.12

- If e is an edge of G incident with two distinct vertices u and v, then the contraction of e is the operation of deleting e and identifying u and v.
- Contracting a loop is the same as deleting it.
- The graph obtained from G by contracting e is denoted G/e (extended to G/F if $F \subseteq E(G)$).
- A graph H is a minor of G if it can be obtained from G by a sequence of operation each of which is one of the following:
  - deleting an edge;
  - deleting an isolated vertex; and

Minors

Definition 2.12

- If e is an edge of G incident with two distinct vertices u and v, then the contraction of e is the operation of deleting e and identifying u and v.
- Contracting a loop is the same as deleting it.
- The graph obtained from G by contracting e is denoted G/e (extended to G/F if $F \subseteq E(G)$).
- A graph H is a minor of G if it can be obtained from G by a sequence of operation each of which is one of the following:
  - deleting an edge;
  - deleting an isolated vertex; and
  - contracting an edge.

Minors

Definition 2.12

- If e is an edge of G incident with two distinct vertices u and v, then the contraction of e is the operation of deleting e and identifying u and v.
- Contracting a loop is the same as deleting it.
- The graph obtained from G by contracting e is denoted G/e (extended to G/F if $F \subseteq E(G)$).
- A graph H is a minor of G if it can be obtained from G by a sequence of operation each of which is one of the following:
  - deleting an edge;
  - deleting an isolated vertex; and
  - contracting an edge.
- We write $H \leqslant_{\mathrm{m}} G$ to indicate that H is isomorphic to a minor of G.

Minors

Definition 2.12

- If e is an edge of G incident with two distinct vertices u and v, then the contraction of e is the operation of deleting e and identifying u and v.
- Contracting a loop is the same as deleting it.
- The graph obtained from G by contracting e is denoted G/e (extended to G/F if $F \subseteq E(G)$).
- A graph H is a minor of G if it can be obtained from G by a sequence of operation each of which is one of the following:
  - deleting an edge;
  - deleting an isolated vertex; and
  - contracting an edge.
- We write $H \leqslant_{\mathrm{m}} G$ to indicate that H is isomorphic to a minor of G.

Note 2.13

The order of operations of deleting and contracting to get a minor of a graph is irrelevant.

Counting Spanning Trees

Theorem 2.14

Let $\tau(G)$ denote the number of distinct spanning trees of a (labeled) graph G.

Counting Spanning Trees

Theorem 2.14

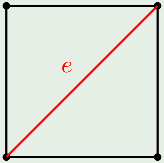
Let $\tau(G)$ denote the number of distinct spanning trees of a (labeled) graph G. If e is a non-loop edge of G, then $\tau(G) = \tau(G \setminus e) + \tau(G/e)$.

Counting Spanning Trees

Theorem 2.14

Let $\tau(G)$ denote the number of distinct spanning trees of a (labeled) graph G. If e is a non-loop edge of G, then $\tau(G) = \tau(G \setminus e) + \tau(G/e)$.

Example 2.15



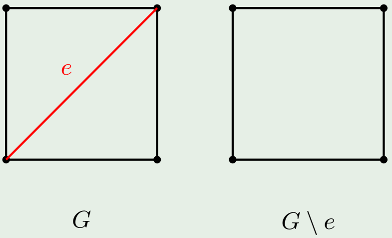
G

Counting Spanning Trees

Theorem 2.14

Let $\tau(G)$ denote the number of distinct spanning trees of a (labeled) graph G. If e is a non-loop edge of G, then $\tau(G) = \tau(G \setminus e) + \tau(G/e)$.

Example 2.15

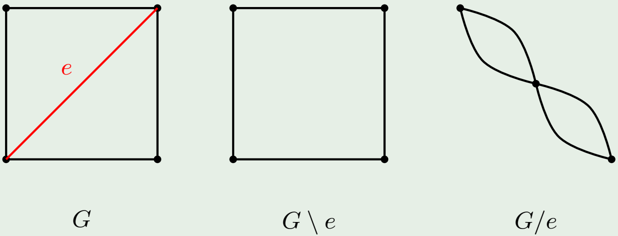


Counting Spanning Trees

Theorem 2.14

Let $\tau(G)$ denote the number of distinct spanning trees of a (labeled) graph G. If e is a non-loop edge of G, then $\tau(G) = \tau(G \setminus e) + \tau(G/e)$.

Example 2.15

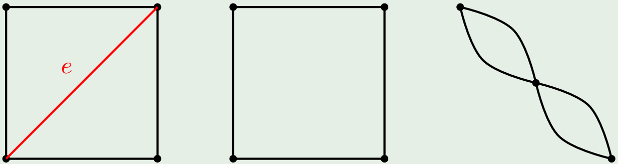


Counting Spanning Trees

Theorem 2.14

Let $\tau(G)$ denote the number of distinct spanning trees of a (labeled) graph G. If e is a non-loop edge of G, then $\tau(G) = \tau(G \setminus e) + \tau(G/e)$.

Example 2.15



G $G \setminus e$ G/e

$$\tau(G) = \tau(G \setminus e) + \tau(G/e) = 4 + 4 = 8$$

Proof of Spanning Tree Formula

- The spanning trees of $G \setminus e$ are precisely the spanning trees of G that avoid e.

Proof of Spanning Tree Formula

- The spanning trees of $G \setminus e$ are precisely the spanning trees of G that avoid e.
- The spanning trees of G/e correspond to the spanning trees of G using e.

Proof of Spanning Tree Formula

- The spanning trees of $G \setminus e$ are precisely the spanning trees of G that avoid e.
- The spanning trees of G/e correspond to the spanning trees of G using e. (If T is a spanning tree of G/e, then $E(T) \cup e$ form the edge-set of a spanning tree of G.)

Proof of Spanning Tree Formula

- The spanning trees of $G \setminus e$ are precisely the spanning trees of G that avoid e.
- The spanning trees of G/e correspond to the spanning trees of G using e. (If T is a spanning tree of G/e, then $E(T) \cup e$ form the edge-set of a spanning tree of G.)
- The formula follows.

Proof of Spanning Tree Formula

- The spanning trees of $G \setminus e$ are precisely the spanning trees of G that avoid e.
- The spanning trees of G/e correspond to the spanning trees of G using e. (If T is a spanning tree of G/e, then $E(T) \cup e$ form the edge-set of a spanning tree of G.)
- The formula follows.

Using the deletion-contraction formula for calculating the number of spanning trees is inefficient.

Proof of Spanning Tree Formula

- The spanning trees of $G \setminus e$ are precisely the spanning trees of G that avoid e.
- The spanning trees of G/e correspond to the spanning trees of G using e. (If T is a spanning tree of G/e, then $E(T) \cup e$ form the edge-set of a spanning tree of G.)
- The formula follows.

Using the deletion-contraction formula for calculating the number of spanning trees is inefficient. A much more efficient method is to construct a special matrix, called the Laplacian of the graph, and to compute its determinant.