Greedy Algorithms

Textbook Reading

Chapters 16, 17, 21, 23 & 24

Overview

Design principle:

Make progress towards a globally optimal solution by making locally optimal choices, hence the name.

Problems:

- Interval scheduling
- Minimum spanning tree
- Shortest paths
- Minimum-length codes

Proof techniques:

- Induction
- The greedy algorithm "stays ahead"
- Exchange argument

Data structures:

- Priority queue
- Union-find data structure

Interval Scheduling

Given:

A set of activities competing for time intervals on a certain resource
(E.g., classes to be scheduled competing for a classroom)

Goal:

Schedule as many non-conflicting activities as possible

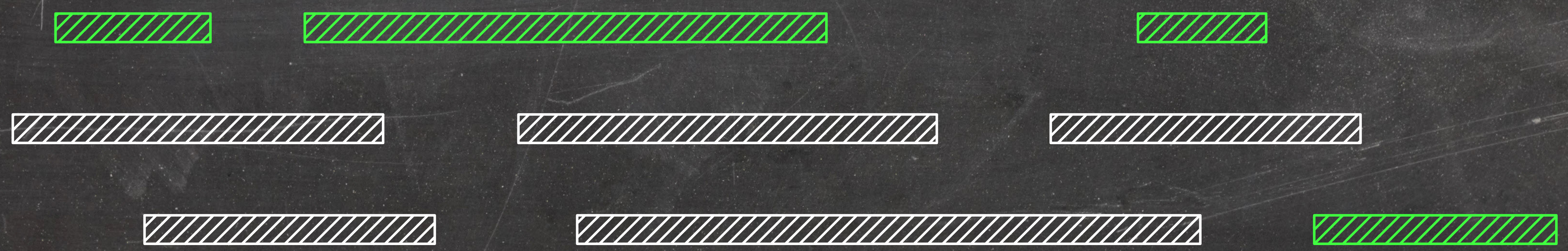
Interval Scheduling

Given:

A set of activities competing for time intervals on a certain resource
(E.g., classes to be scheduled competing for a classroom)

Goal:

Schedule as many non-conflicting activities as possible



Interval Scheduling

Given:

A set of activities competing for time intervals on a certain resource
(E.g., classes to be scheduled competing for a classroom)

Goal:

Schedule as many non-conflicting activities as possible

A Greedy Framework for Interval Scheduling

```
FindSchedule(S)
S′ = ∅
while S is not empty
    do pick an interval I in S
       add I to S′
       remove all intervals from S that conflict with I
return S′
```

A Greedy Framework for Interval Scheduling

FindSchedule(S)

```
S′ = ∅
while S is not empty
    do pick an interval I in S
        add I to S′
        remove all intervals from S that conflict with I
return S′
```

Main questions:

- Can we choose an arbitrary interval I in each iteration?
- How do we choose interval I in each iteration?

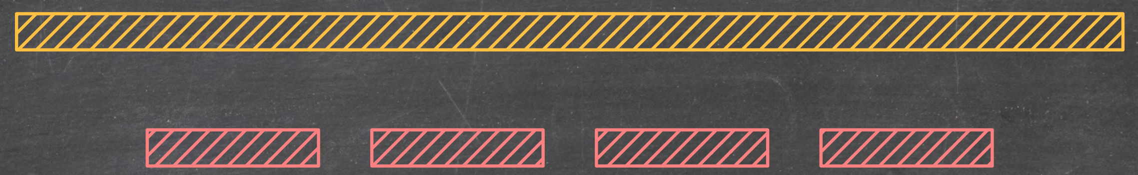
Greedy Strategies for Interval Scheduling

Greedy Strategies for Interval Scheduling

Choose the interval that starts first.

Greedy Strategies for Interval Scheduling

Choose the interval that starts first.



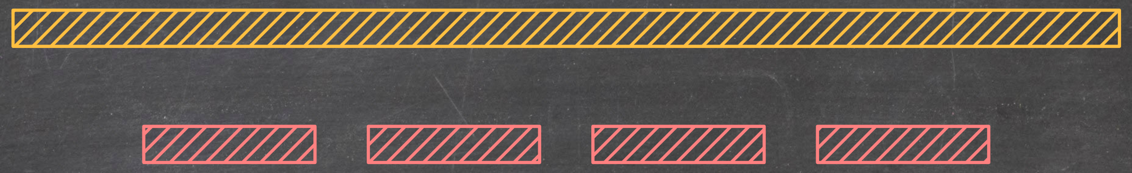
Greedy Strategies for Interval Scheduling

Choose the interval that starts first.

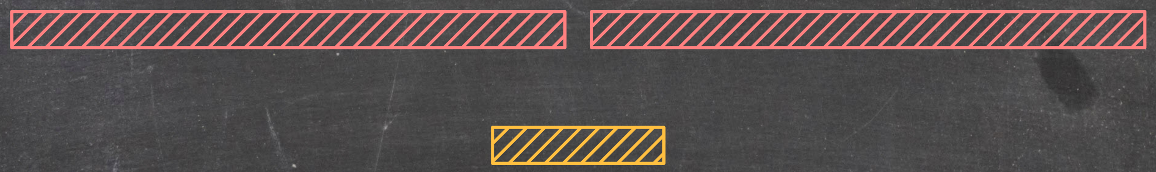
Choose the shortest interval.

Greedy Strategies for Interval Scheduling

Choose the interval that starts first.



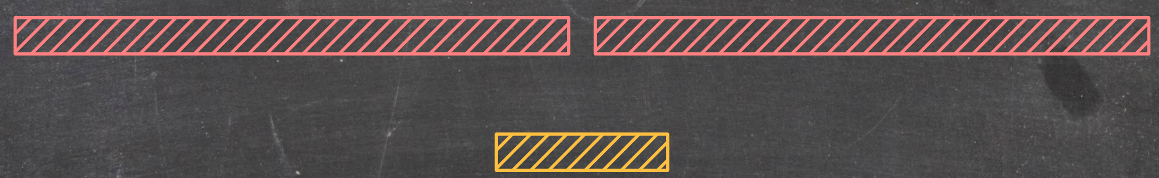
Choose the shortest interval.



Greedy Strategies for Interval Scheduling

Choose the interval that starts first.

Choose the shortest interval.

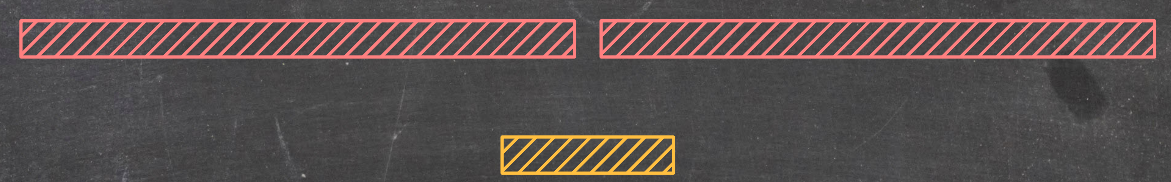


Choose the interval with the fewest conflicts.

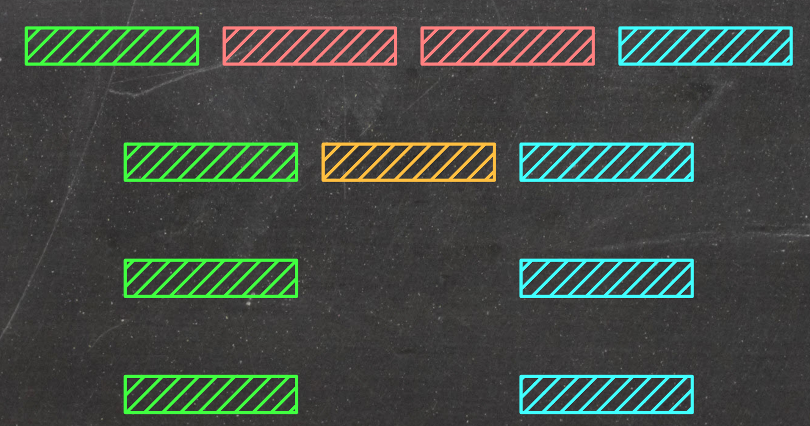
Greedy Strategies for Interval Scheduling

Choose the interval that starts first.

Choose the shortest interval.



Choose the interval with the fewest conflicts.



The Strategy That Works

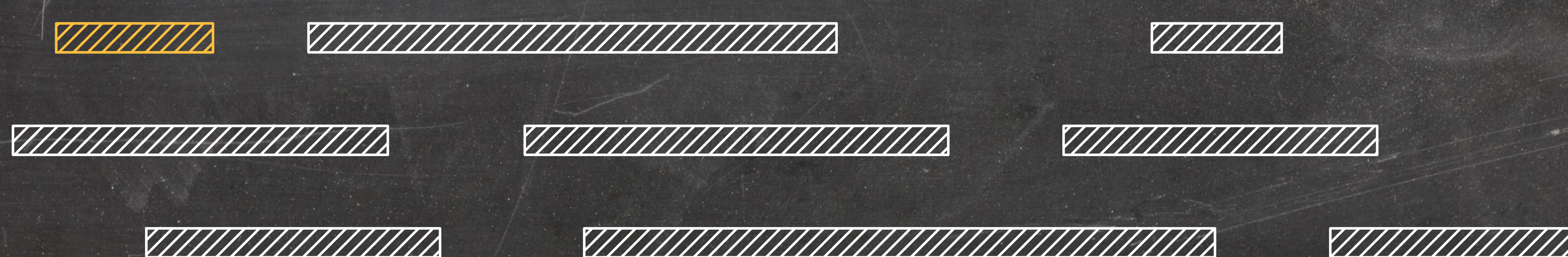
FindSchedule(S)

```
S′ = ∅
while S is not empty
   do let I be the interval in S that ends first
         add I to S′
         remove all intervals from S that conflict with I
return S′
```

The Strategy That Works

FindSchedule(S)

```
S' = ∅
while S is not empty
    do let I be the interval in S that ends first
        add I to S'
        remove all intervals from S that conflict with I
return S'
```



The Strategy That Works

FindSchedule(S)

```
S′ = ∅
while S is not empty
    do let I be the interval in S that ends first
        add I to S′
        remove all intervals from S that conflict with I
return S′
```

The Strategy That Works

FindSchedule(S)

```
S′ = ∅
while S is not empty
    do let I be the interval in S that ends first
        add I to S′
        remove all intervals from S that conflict with I
return S′
```

The Strategy That Works

FindSchedule(S)

```
S′ = ∅
while S is not empty
    do let I be the interval in S that ends first
        add I to S′
        remove all intervals from S that conflict with I
return S′
```

The Strategy That Works

FindSchedule(S)

```
S′ = ∅
while S is not empty
    do let I be the interval in S that ends first
        add I to S′
        remove all intervals from S that conflict with I
return S′
```

The Strategy That Works

FindSchedule(S)

```
S′ = ∅
while S is not empty
    do let I be the interval in S that ends first
        add I to S′
        remove all intervals from S that conflict with I
return S′
```

The Strategy That Works

FindSchedule(S)

```
S′ = ∅
while S is not empty
    do let I be the interval in S that ends first
        add I to S′
        remove all intervals from S that conflict with I
return S′
```

The Greedy Algorithm Stays Ahead

Lemma: FindSchedule finds a maximum-cardinality set of conflict-free intervals.

The Greedy Algorithm Stays Ahead

Lemma: FindSchedule finds a maximum-cardinality set of conflict-free intervals.

Let $I_1 \prec I_2 \prec \cdots \prec I_k$ be the schedule we compute.

Let $O_1 \prec O_2 \prec \cdots \prec O_m$ be an optimal schedule.

Prove by induction on j that I_j ends no later than O_j.

The Greedy Algorithm Stays Ahead

Lemma: FindSchedule finds a maximum-cardinality set of conflict-free intervals.

Let $I_1 \prec I_2 \prec \cdots \prec I_k$ be the schedule we compute.

Let $O_1 \prec O_2 \prec \cdots \prec O_m$ be an optimal schedule.

Prove by induction on j that I_j ends no later than O_j.

$\Rightarrow$ Since O_{j+1} starts after O_j ends, it also starts after I_j ends.

The Greedy Algorithm Stays Ahead

Lemma: FindSchedule finds a maximum-cardinality set of conflict-free intervals.

Let $I_1 \prec I_2 \prec \cdots \prec I_k$ be the schedule we compute.

Let $O_1 \prec O_2 \prec \cdots \prec O_m$ be an optimal schedule.

Prove by induction on j that I_j ends no later than O_j.

$\Rightarrow$ Since O_{j+1} starts after O_j ends, it also starts after I_j ends.

$\Rightarrow$ If $k < m$, FindSchedule inspects O_{k+1} after I_k and thus would have added it to its output, a contradiction.

The Greedy Algorithm Stays Ahead

Lemma: FindSchedule finds a maximum-cardinality set of conflict-free intervals.

Proof by induction:

Base case(s): Verify that the claim holds for a set of initial instances.

Inductive step: Prove that, if the claim holds for the first k instances, it holds for the $(k + 1)$st instance.

The Greedy Algorithm Stays Ahead

Lemma: FindSchedule finds a maximum-cardinality set of conflict-free intervals.

Base case: I_1 ends no later than O_1 because both I_1 and O_1 are chosen from S and I_1 is the interval in S that ends first.

The Greedy Algorithm Stays Ahead

Lemma: FindSchedule finds a maximum-cardinality set of conflict-free intervals.

Base case: I_1 ends no later than O_1 because both I_1 and O_1 are chosen from S and I_1 is the interval in S that ends first.

Inductive step:

Since I_k ends before O_{k+1}, so do $I_1, I_2, \ldots, I_{k-1}$.

The Greedy Algorithm Stays Ahead

Lemma: FindSchedule finds a maximum-cardinality set of conflict-free intervals.

Base case: I_1 ends no later than O_1 because both I_1 and O_1 are chosen from S and I_1 is the interval in S that ends first.

Inductive step:

Since I_k ends before O_{k+1}, so do $I_1, I_2, \ldots, I_{k-1}$.

$\Rightarrow$ O_{k+1} does not conflict with $I_1, I_2, \ldots, I_k$.

The Greedy Algorithm Stays Ahead

Lemma: FindSchedule finds a maximum-cardinality set of conflict-free intervals.

Base case: I_1 ends no later than O_1 because both I_1 and O_1 are chosen from S and I_1 is the interval in S that ends first.

Inductive step:

Since I_k ends before O_{k+1}, so do $I_1, I_2, \ldots, I_{k-1}$.

$\Rightarrow$ O_{k+1} does not conflict with $I_1, I_2, \ldots, I_k$.

$\Rightarrow$ I_{k+1} ends no later than O_{k+1} because it is the interval that ends first among all intervals that do not conflict with $I_1, I_2, \ldots, I_k$.

Implementing The Algorithm

FindSchedule(S)

```
S′ = [ ]
sort the intervals in S by increasing finish times
S′.append(S[1])
f = S[1].f
for i = 2 to |S|
    do if S[i].s > f
            then S′.append(S[i])
                 f = S[i].f
return S′
```

Implementing The Algorithm

FindSchedule(S)

```
S′ = [ ]
sort the intervals in S by increasing finish times
S′.append(S[1])
f = S[1].f
for i = 2 to |S|
    do if S[i].s > f
            then S′.append(S[i])
                 f = S[i].f
return S′
```

Lemma: A maximum-cardinality set of non-conflicting intervals can be found in $O(n \lg n)$ time.

Minimum Spanning Tree

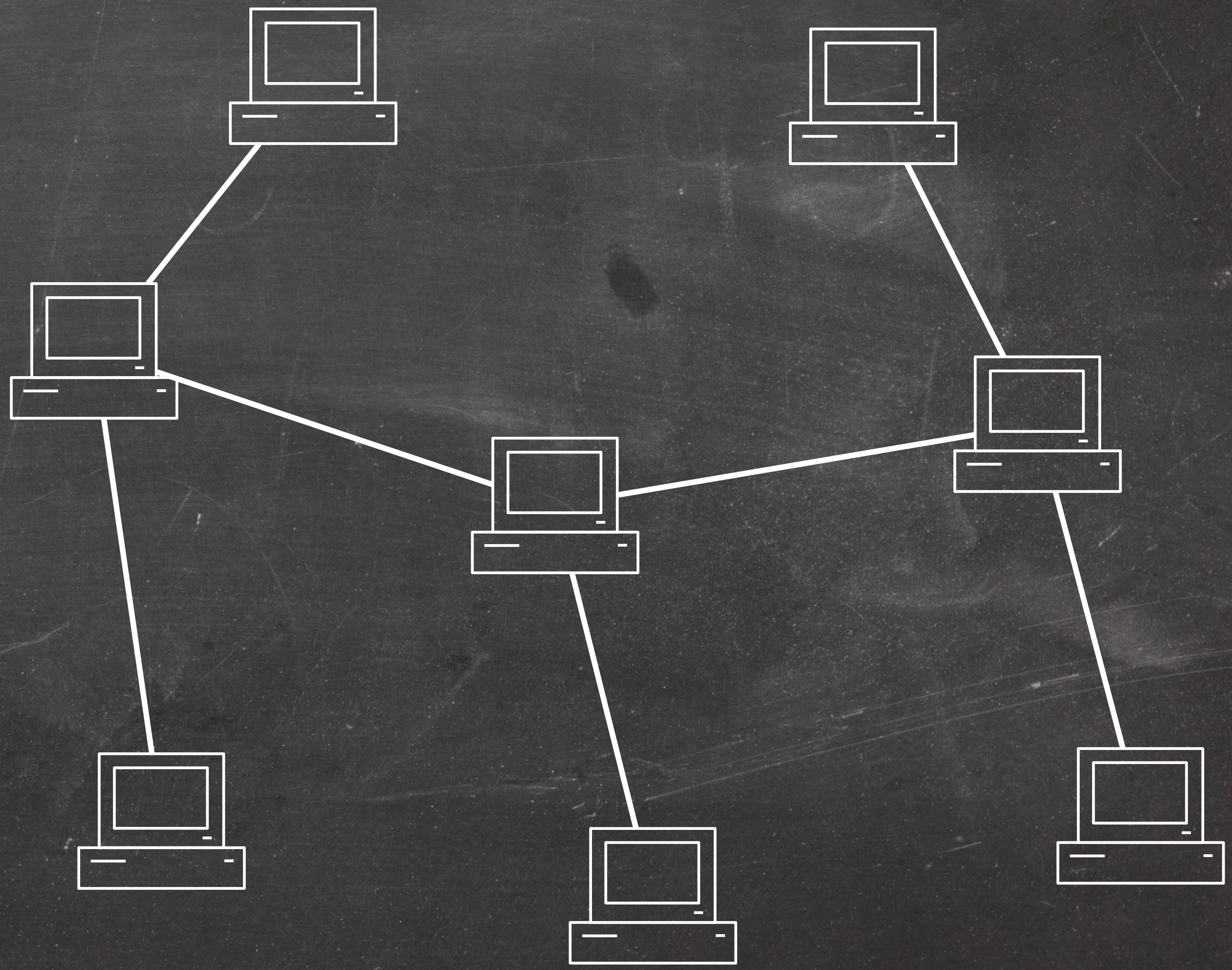
Given: n computers

Goal: Connect them so that every computer can communicate with every other computer.

We don't care whether the connection between any pair of computers is short.

We don't care about fault tolerance.

Every foot of cable costs us \$1.

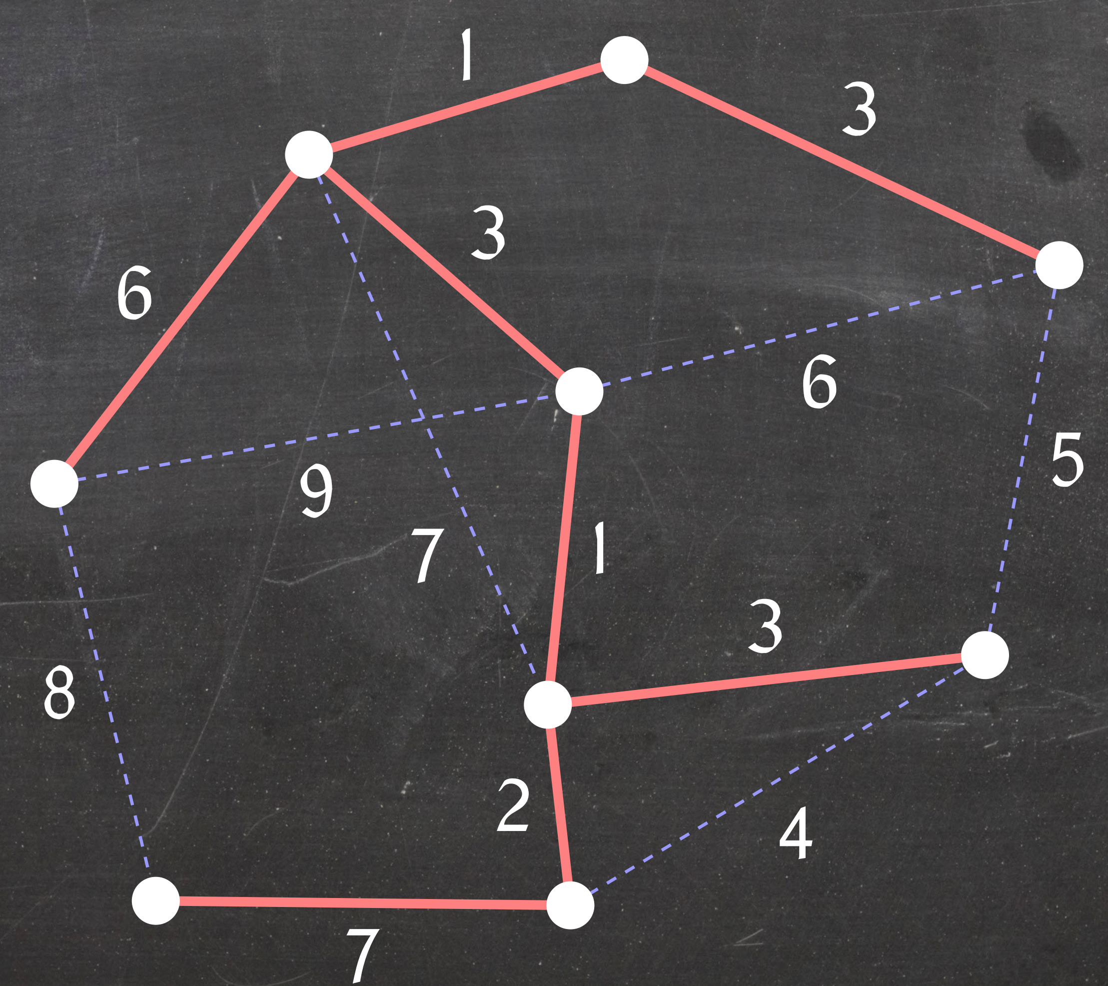


$\Rightarrow$ We want the cheapest possible network.

Minimum Spanning Tree

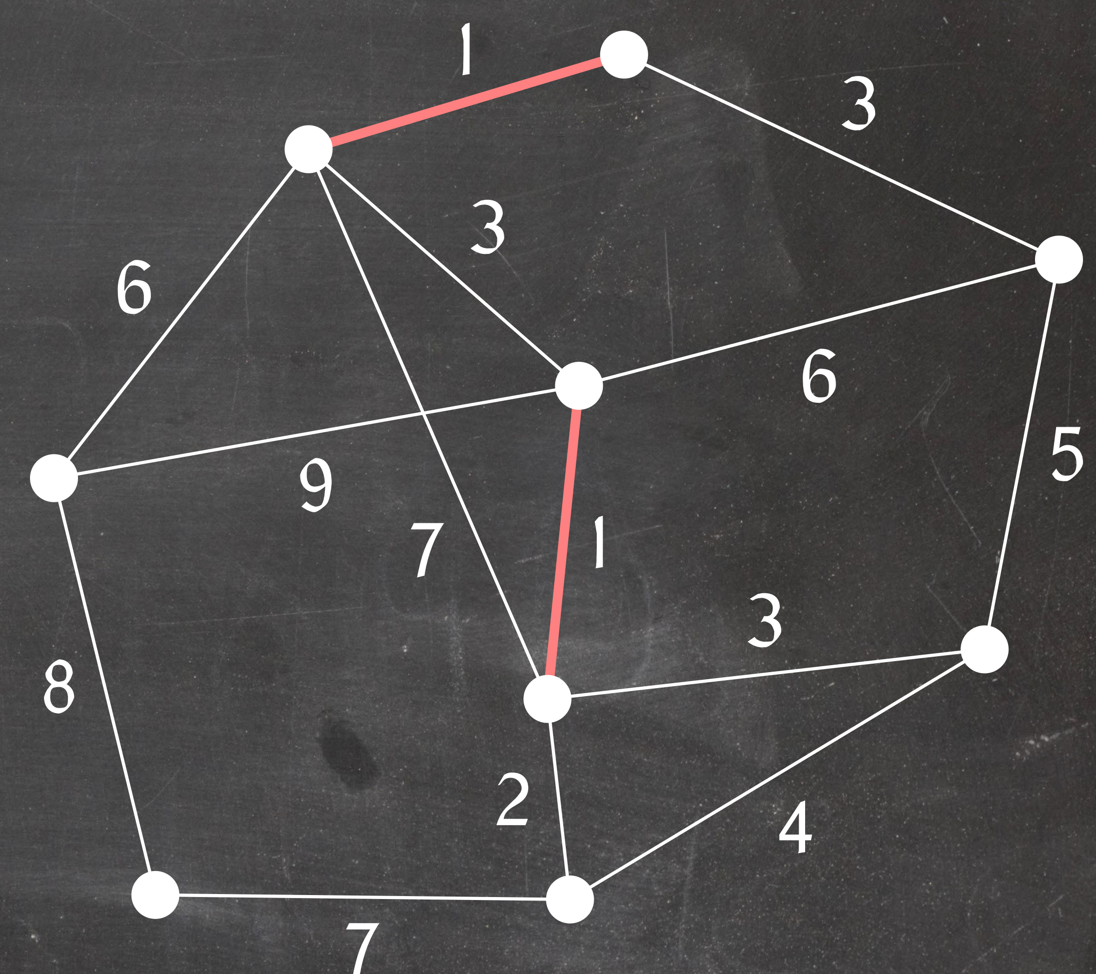
Given a graph $G = (V, E)$ and an assignment of weights (costs) to the edges of G, a minimum spanning tree (MST) T of G is a spanning tree with minimum total weight

$$w(T) = \sum_{e \in T} w(e).$$



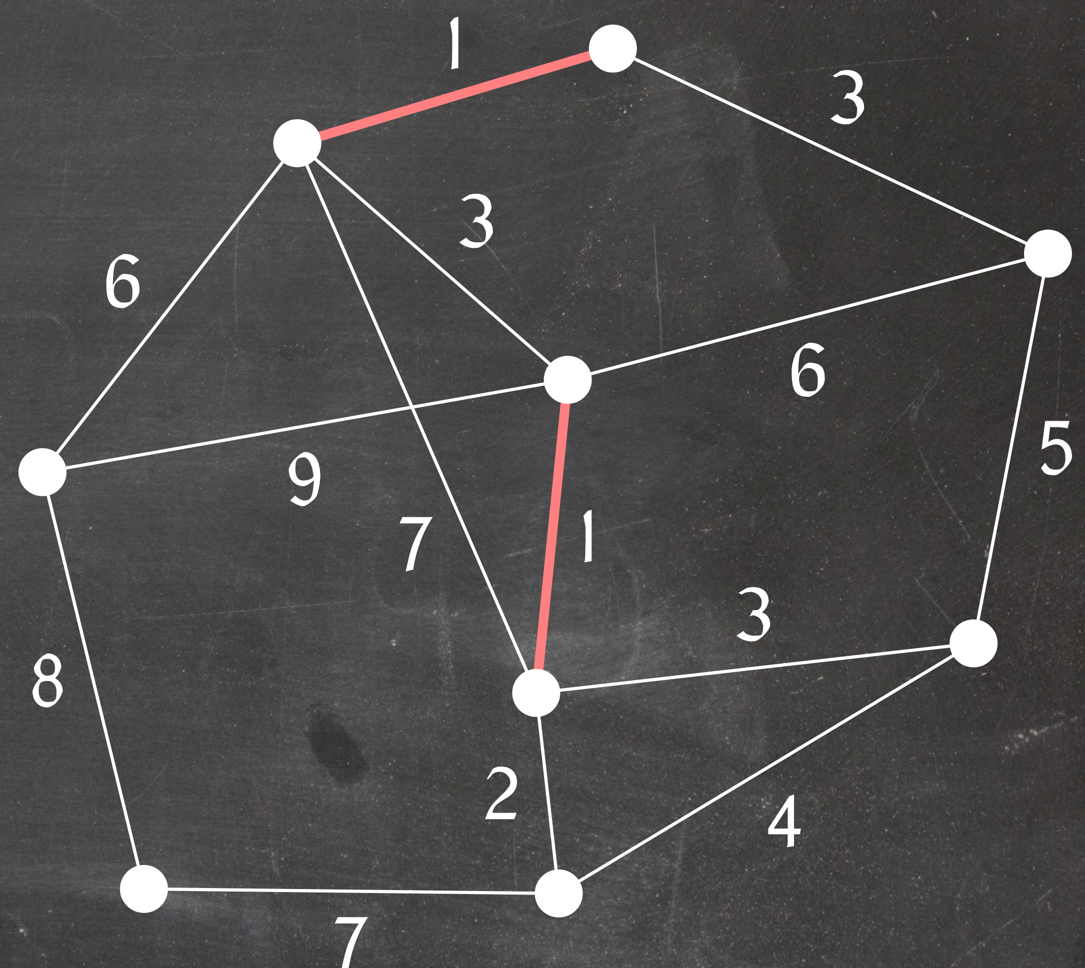
Kruskal's Algorithm

Greedy choice: Pick the shortest edge



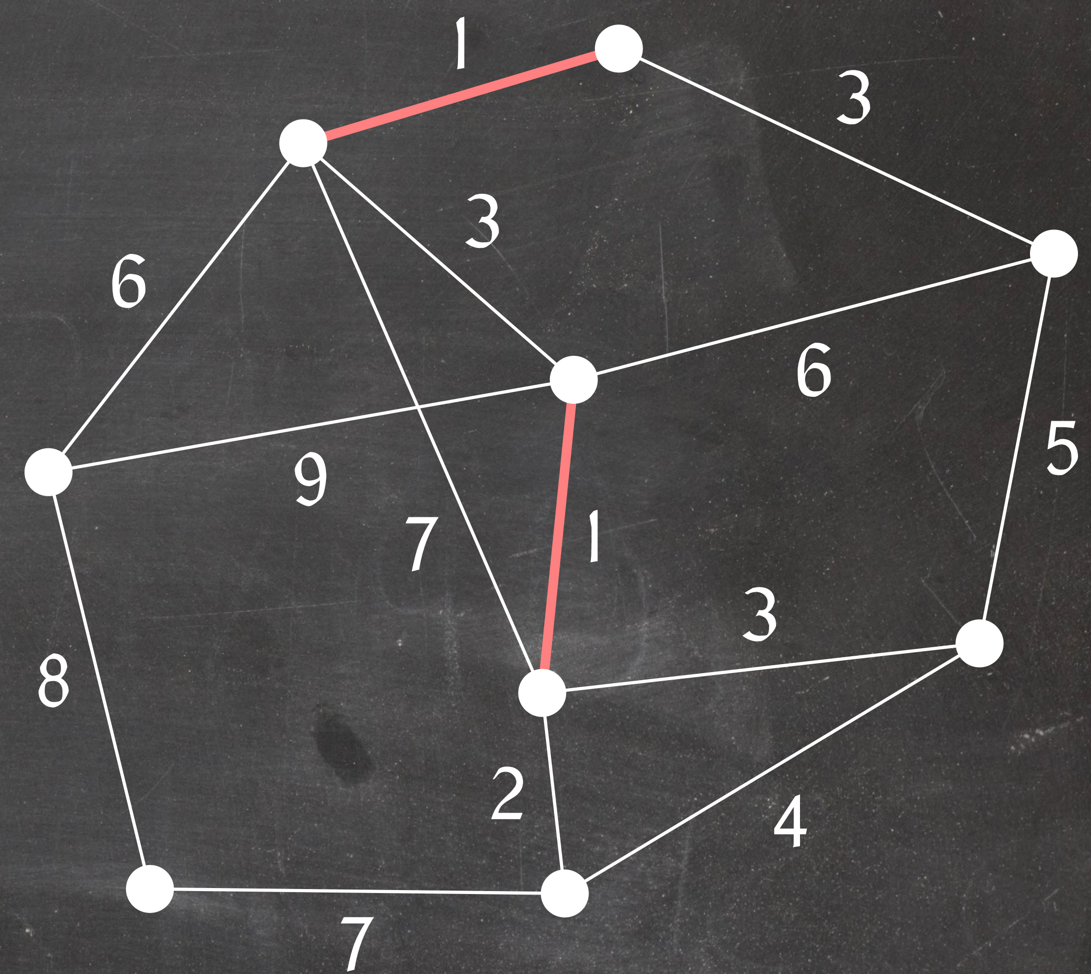
Kruskal's Algorithm

Greedy choice: Pick the shortest edge that connects two previously disconnected vertices.



Kruskal's Algorithm

Greedy choice: Pick the shortest edge that connects two previously disconnected vertices.

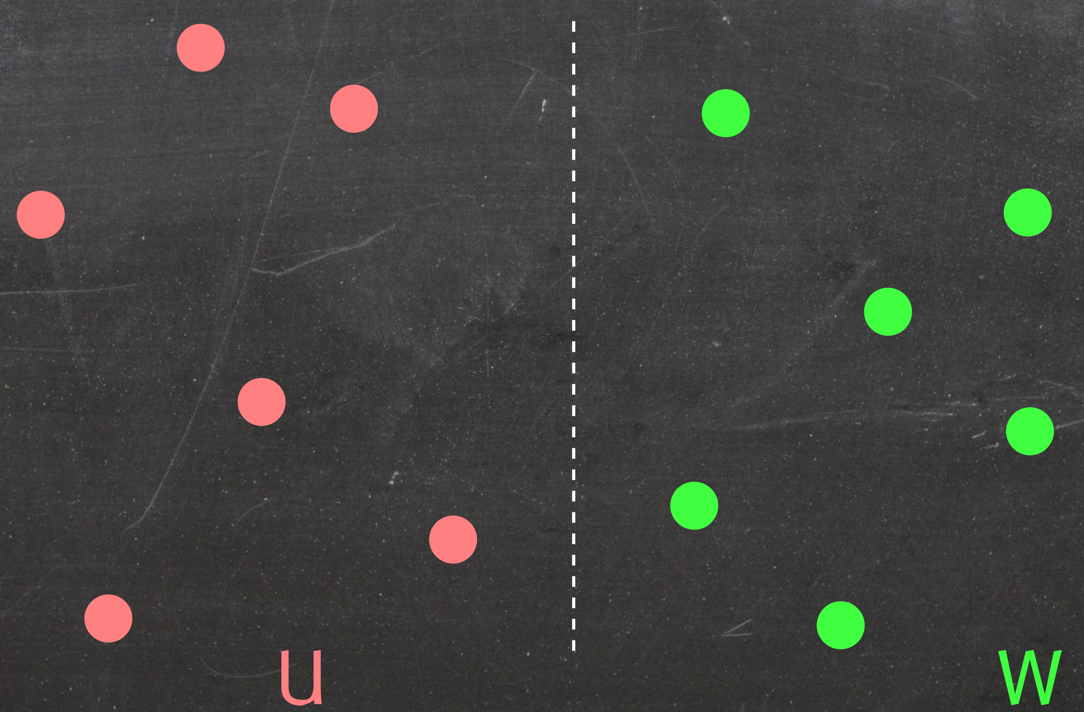


Kruskal(G)

```
T = (V, ∅)
while T has more than one connected component
   do let e be the cheapest edge of G whose endpoints belong to different
         connected components of T
         add e to T
return T
```

A Cut Theorem

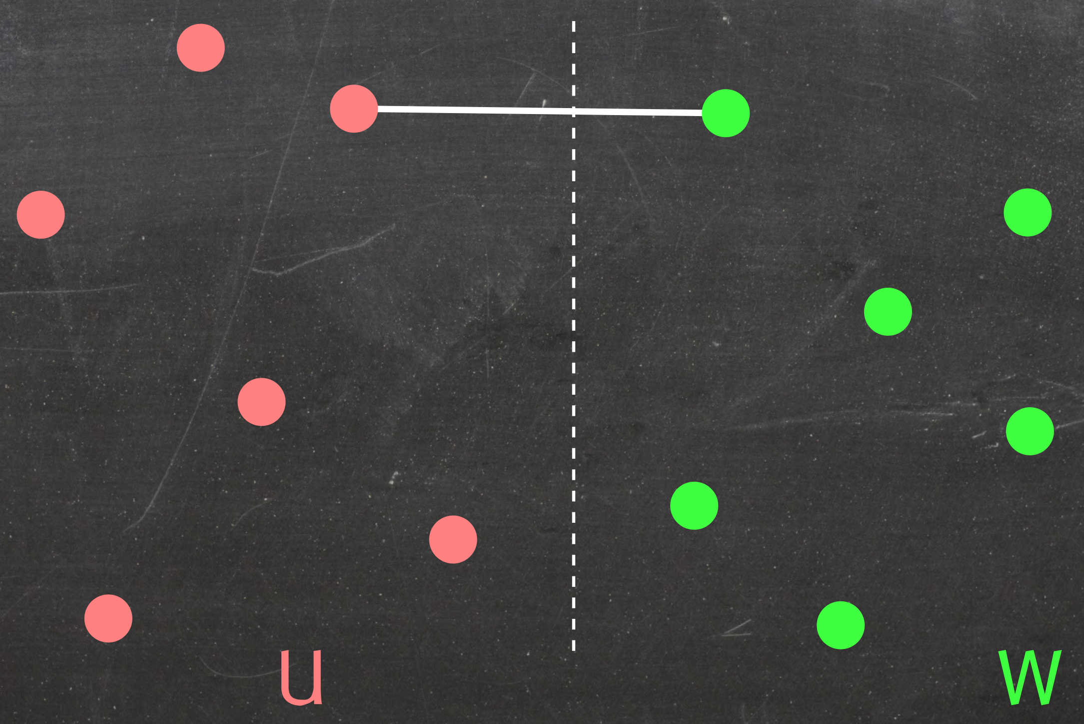
A cut is a partition (U, W) of V into two non-empty subsets: $\emptyset \subset U \subset V$ and $W = V \setminus U$.



A Cut Theorem

A cut is a partition (U, W) of V into two non-empty subsets: $\emptyset \subset U \subset V$ and $W = V \setminus U$.

An edge crosses the cut (U, W) if it has one endpoint in U and one in W.

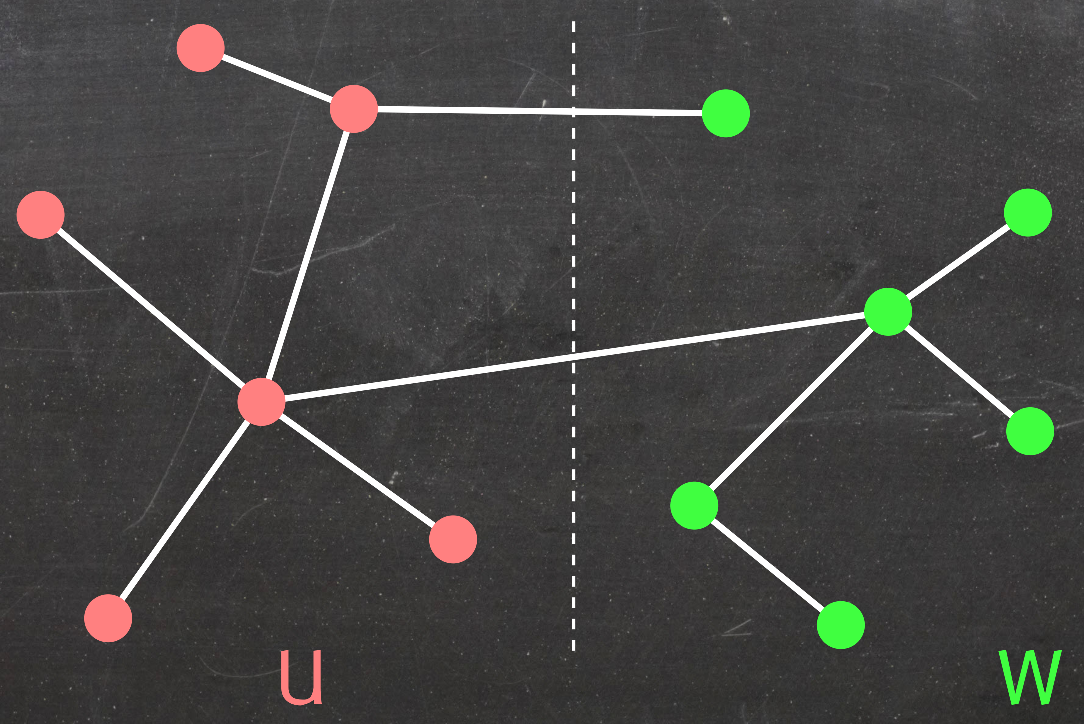


A Cut Theorem

A cut is a partition (U, W) of V into two non-empty subsets: $\emptyset \subset U \subset V$ and $W = V \setminus U$.

An edge crosses the cut (U, W) if it has one endpoint in U and one in W.

Theorem: Let T be a minimum spanning tree, let (U, W) be an arbitrary cut, and let e be the cheapest edge crossing the cut. Then there exists a minimum spanning tree that contains e and all edges of T that do not cross the cut.

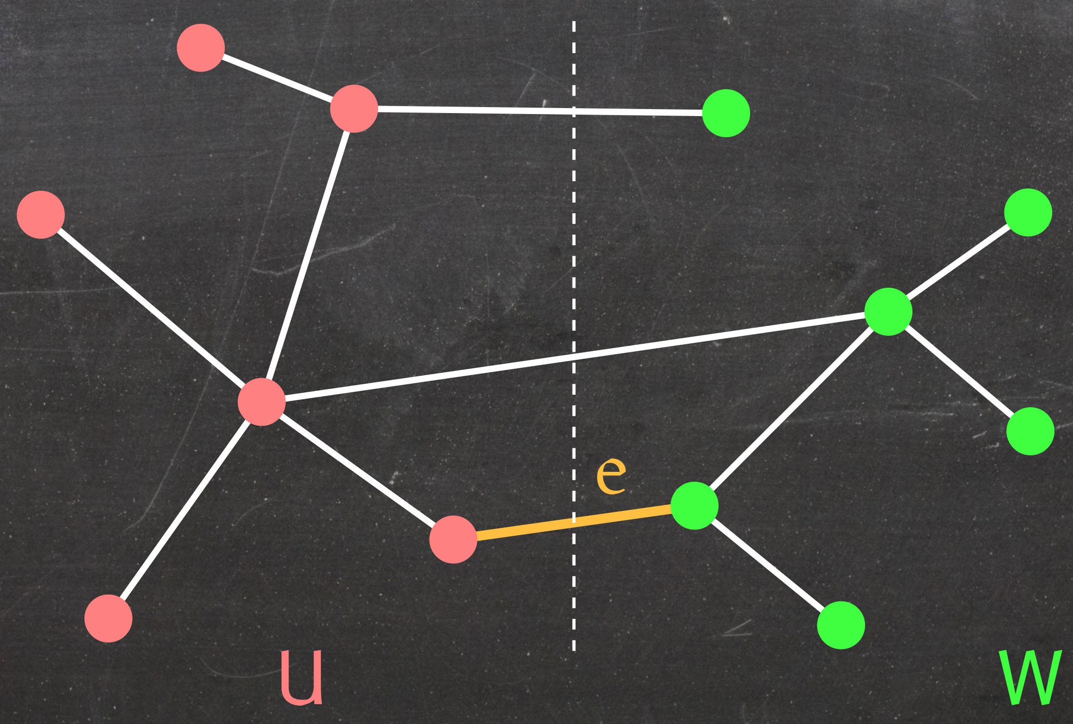


A Cut Theorem

A cut is a partition (U, W) of V into two non-empty subsets: $\emptyset \subset U \subset V$ and $W = V \setminus U$.

An edge crosses the cut (U, W) if it has one endpoint in U and one in W.

Theorem: Let T be a minimum spanning tree, let (U, W) be an arbitrary cut, and let e be the cheapest edge crossing the cut. Then there exists a minimum spanning tree that contains e and all edges of T that do not cross the cut.

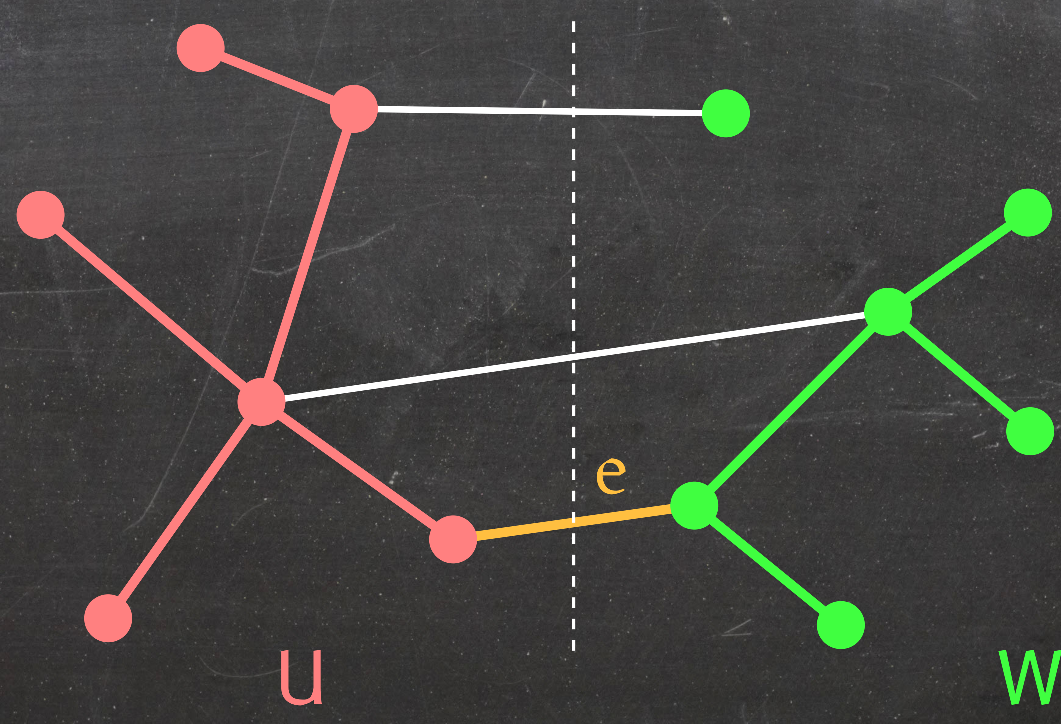


A Cut Theorem

A cut is a partition (U, W) of V into two non-empty subsets: $\emptyset \subset U \subset V$ and $W = V \setminus U$.

An edge crosses the cut (U, W) if it has one endpoint in U and one in W.

Theorem: Let T be a minimum spanning tree, let (U, W) be an arbitrary cut, and let e be the cheapest edge crossing the cut. Then there exists a minimum spanning tree that contains e and all edges of T that do not cross the cut.



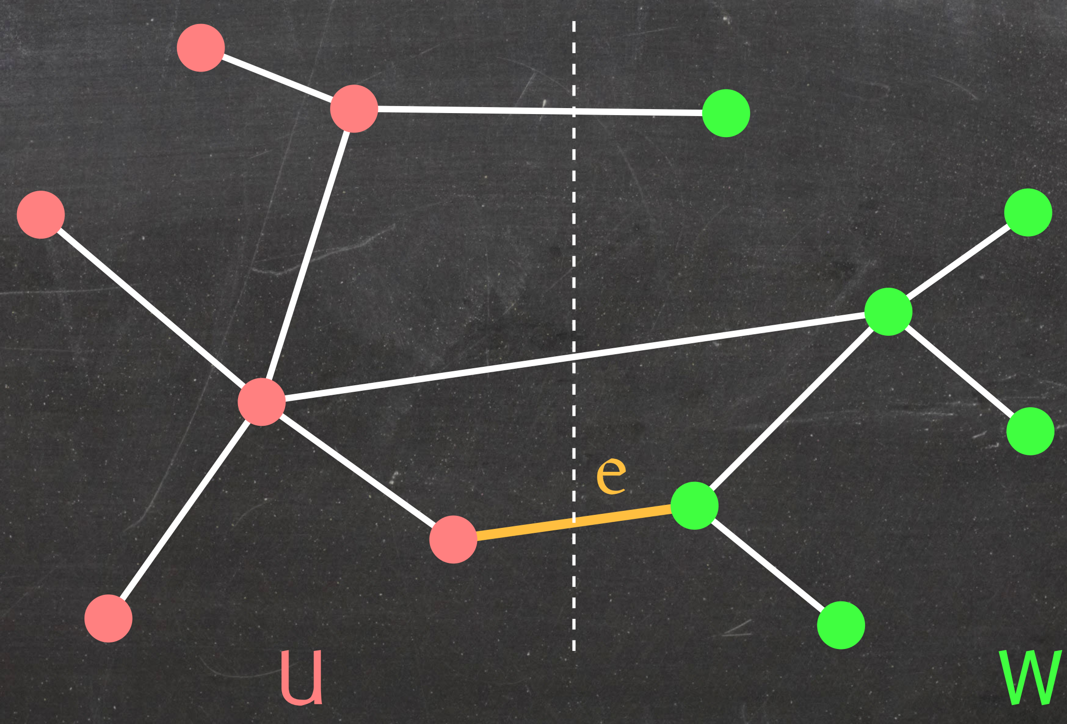
A Cut Theorem

A cut is a partition (U, W) of V into two non-empty subsets: $\emptyset \subset U \subset V$ and $W = V \setminus U$.

An edge crosses the cut (U, W) if it has one endpoint in U and one in W.

Theorem: Let T be a minimum spanning tree, let (U, W) be an arbitrary cut, and let e be the cheapest edge crossing the cut. Then there exists a minimum spanning tree that contains e and all edges of T that do not cross the cut.

An exchange argument:



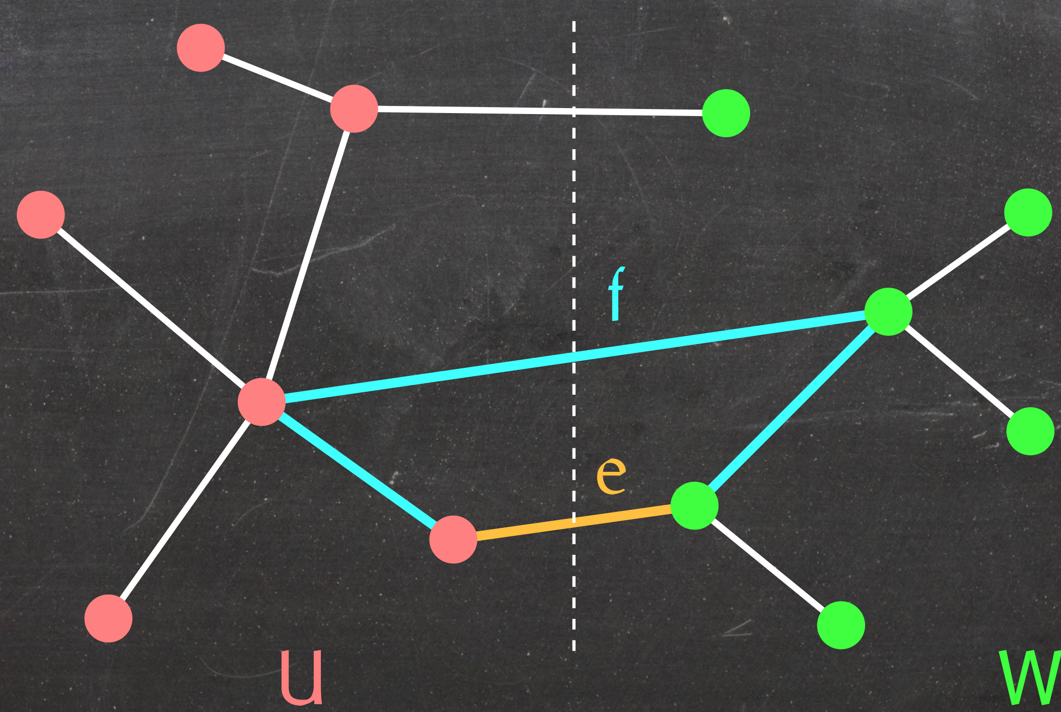
A Cut Theorem

A cut is a partition (U, W) of V into two non-empty subsets: $\emptyset \subset U \subset V$ and $W = V \setminus U$.

An edge crosses the cut (U, W) if it has one endpoint in U and one in W.

Theorem: Let T be a minimum spanning tree, let (U, W) be an arbitrary cut, and let e be the cheapest edge crossing the cut. Then there exists a minimum spanning tree that contains e and all edges of T that do not cross the cut.

An exchange argument:



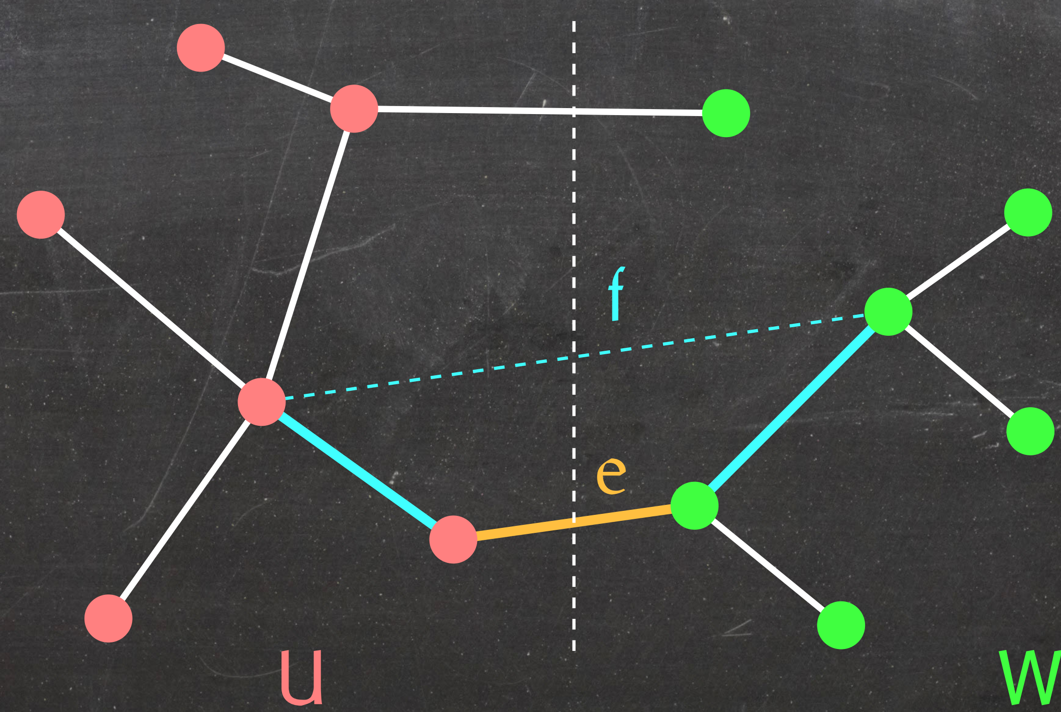
A Cut Theorem

A cut is a partition (U, W) of V into two non-empty subsets: $\emptyset \subset U \subset V$ and $W = V \setminus U$.

An edge crosses the cut (U, W) if it has one endpoint in U and one in W.

Theorem: Let T be a minimum spanning tree, let (U, W) be an arbitrary cut, and let e be the cheapest edge crossing the cut. Then there exists a minimum spanning tree that contains e and all edges of T that do not cross the cut.

An exchange argument:



Correctness Of Kruskal's Algorithm

Lemma: Kruskal's algorithm computes a minimum spanning tree.

Correctness Of Kruskal's Algorithm

Lemma: Kruskal's algorithm computes a minimum spanning tree.

Let $(V, \emptyset) = F_0 \subset F_1 \subset \cdots \subset F_{n-1} = T$ be the sequence of forests computed by Kruskal's algorithm.

Correctness Of Kruskal's Algorithm

Lemma: Kruskal's algorithm computes a minimum spanning tree.

Let $(V, \emptyset) = F_0 \subset F_1 \subset \cdots \subset F_{n-1} = T$ be the sequence of forests computed by Kruskal's algorithm.

Need to prove that, for all i, there exists an MST $T_i \supseteq F_i$.

Correctness Of Kruskal's Algorithm

Lemma: Kruskal's algorithm computes a minimum spanning tree.

Let $(V, \emptyset) = F_0 \subset F_1 \subset \cdots \subset F_{n-1} = T$ be the sequence of forests computed by Kruskal's algorithm.

Need to prove that, for all i, there exists an MST $T_i \supseteq F_i$.

Correctness Of Kruskal's Algorithm

Lemma: Kruskal's algorithm computes a minimum spanning tree.

Let $(V, \emptyset) = F_0 \subset F_1 \subset \cdots \subset F_{n-1} = T$ be the sequence of forests computed by Kruskal's algorithm.

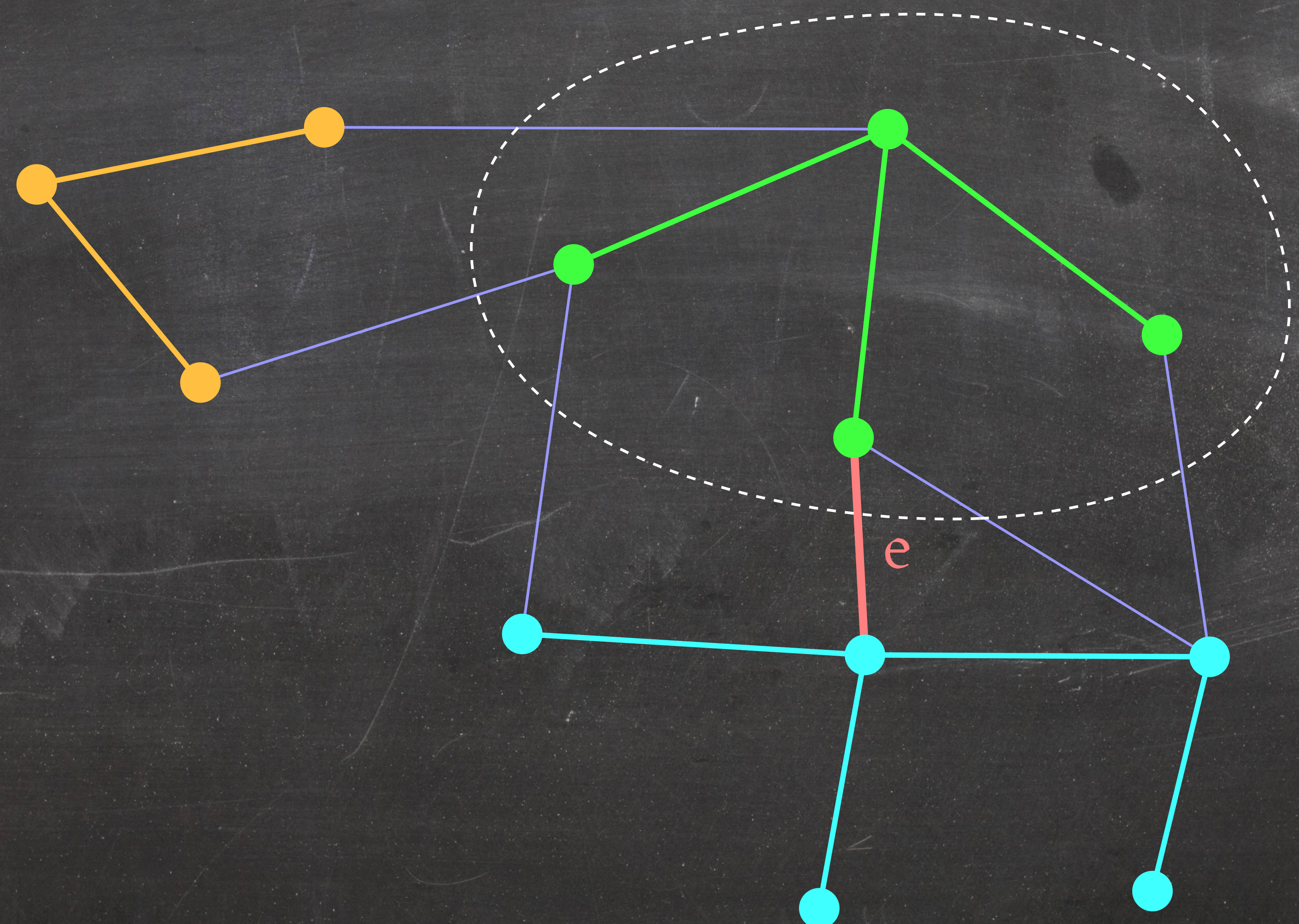
Need to prove that, for all i, there exists an MST $T_i \supseteq F_i$.

Correctness Of Kruskal's Algorithm

Lemma: Kruskal's algorithm computes a minimum spanning tree.

Let $(V, \emptyset) = F_0 \subset F_1 \subset \cdots \subset F_{n-1} = T$ be the sequence of forests computed by Kruskal's algorithm.

Need to prove that, for all i, there exists an MST $T_i \supseteq F_i$.



Implementing Kruskal's Algorithm

Kruskal(G)

```
T = (V, ∅)
while T has more than one connected component
    do let e be the cheapest edge of G whose endpoints belong to different
           connected components of T
           add e to T
return T
```

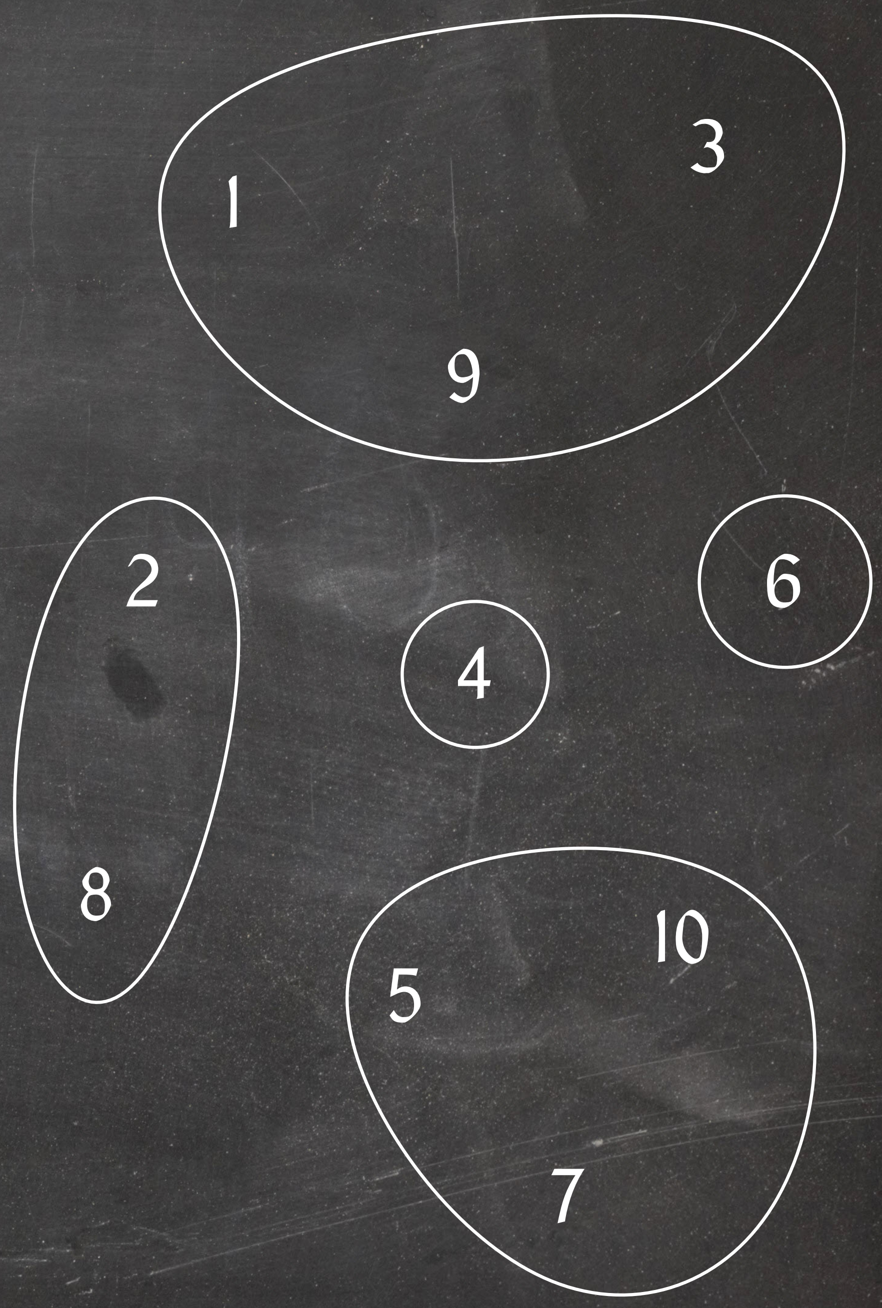
$$\Downarrow$$

Kruskal(G)

```
T = (V, ∅)
sort the edges in G by increasing weight
for every edge (v, w) of G, in sorted order
    do if v and w belong to different connected components of T
          then add (v, w) to T
return T
```

A Union-Find Data Structure

Given a set S of elements, maintain a partition of S into subsets $S_1, S_2, \ldots, S_k$.

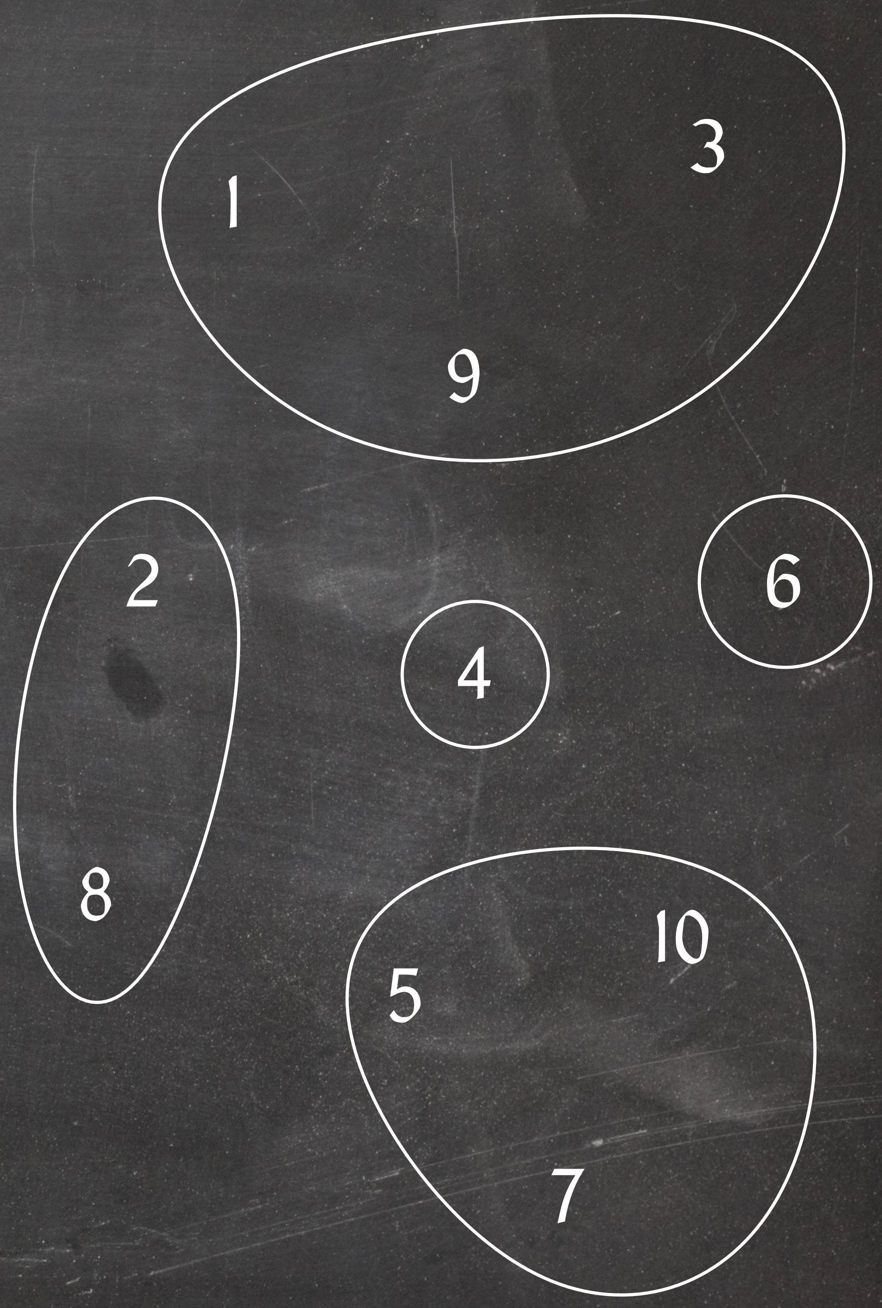


A Union-Find Data Structure

Given a set S of elements, maintain a partition of S into subsets $S_1, S_2, \ldots, S_k$.

Support the following operations:

Union(x, y): Replace sets S_i and S_j in the partition with $S_i \cup S_j$, where $x \in S_i$ and $y \in S_j$.

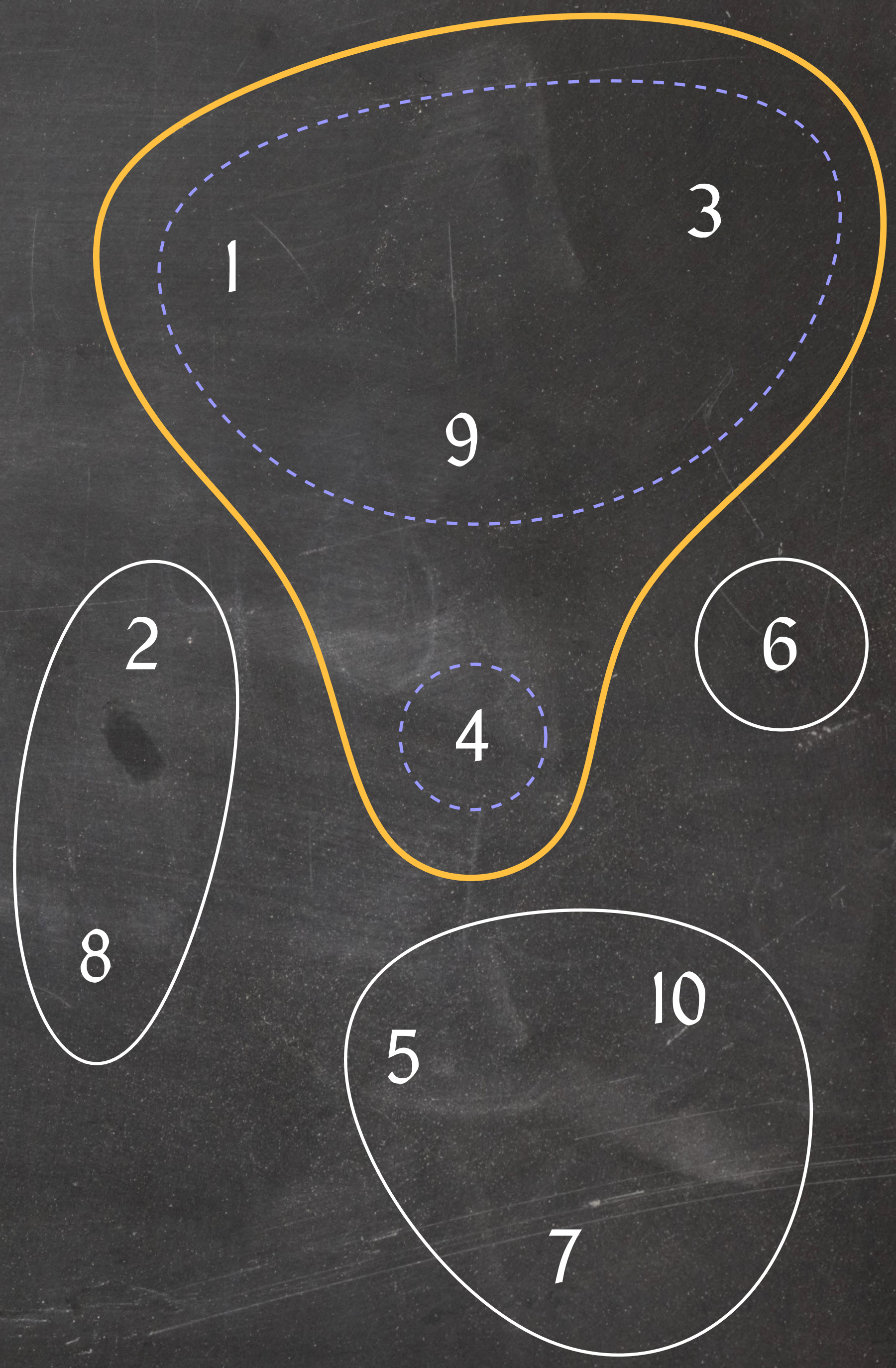


A Union-Find Data Structure

Given a set S of elements, maintain a partition of S into subsets $S_1, S_2, \ldots, S_k$.

Support the following operations:

Union(x, y): Replace sets S_i and S_j in the partition with $S_i \cup S_j$, where $x \in S_i$ and $y \in S_j$.



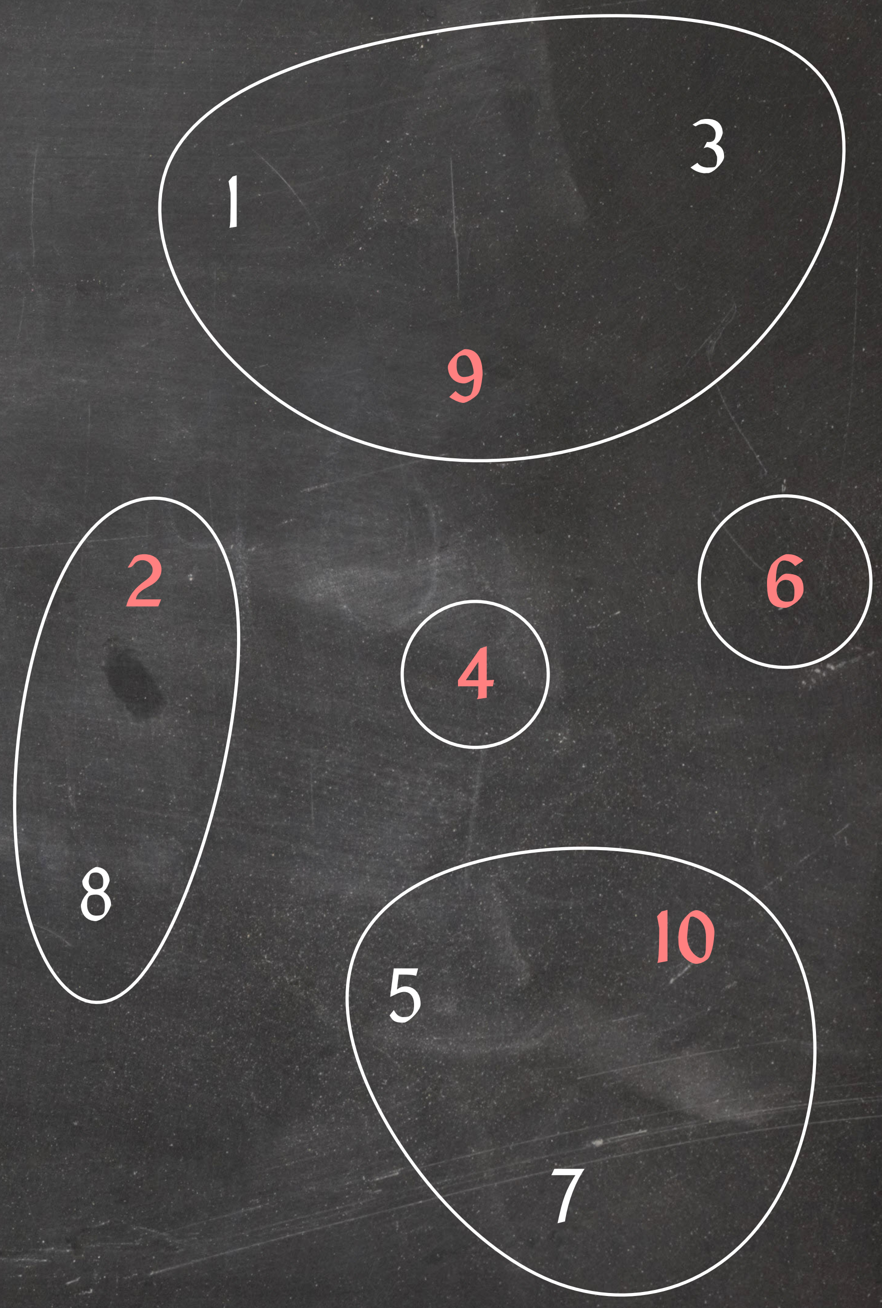
A Union-Find Data Structure

Given a set S of elements, maintain a partition of S into subsets $S_1, S_2, \ldots, S_k$.

Support the following operations:

Union(x, y): Replace sets S_i and S_j in the partition with $S_i \cup S_j$, where $x \in S_i$ and $y \in S_j$.

Find(x): Return a representative $r(S_i) \in S_i$ of the set S_i that contains x.



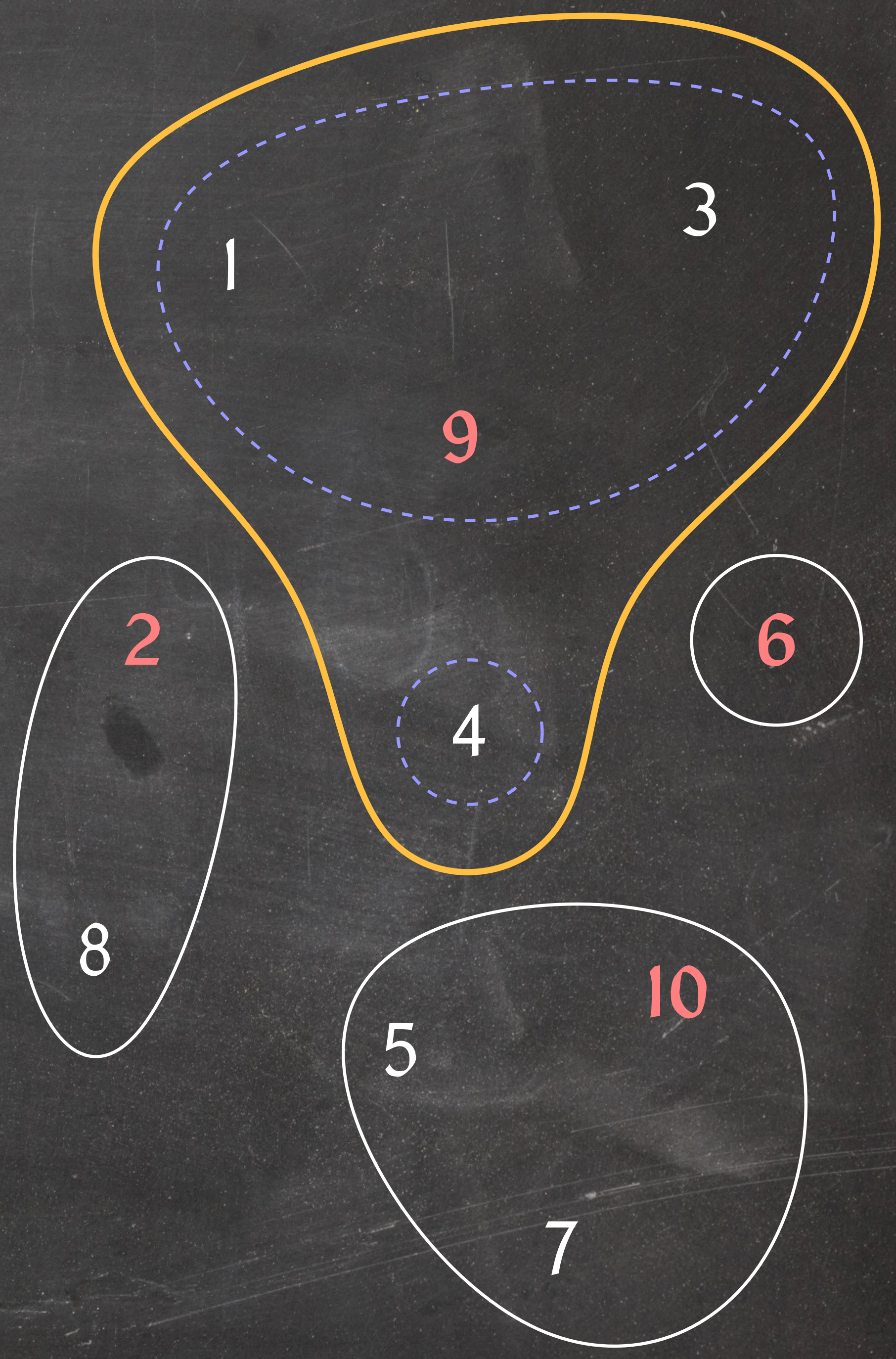
A Union-Find Data Structure

Given a set S of elements, maintain a partition of S into subsets $S_1, S_2, \ldots, S_k$.

Support the following operations:

Union(x, y): Replace sets S_i and S_j in the partition with $S_i \cup S_j$, where $x \in S_i$ and $y \in S_j$.

Find(x): Return a representative $r(S_i) \in S_i$ of the set S_i that contains x.



A Union-Find Data Structure

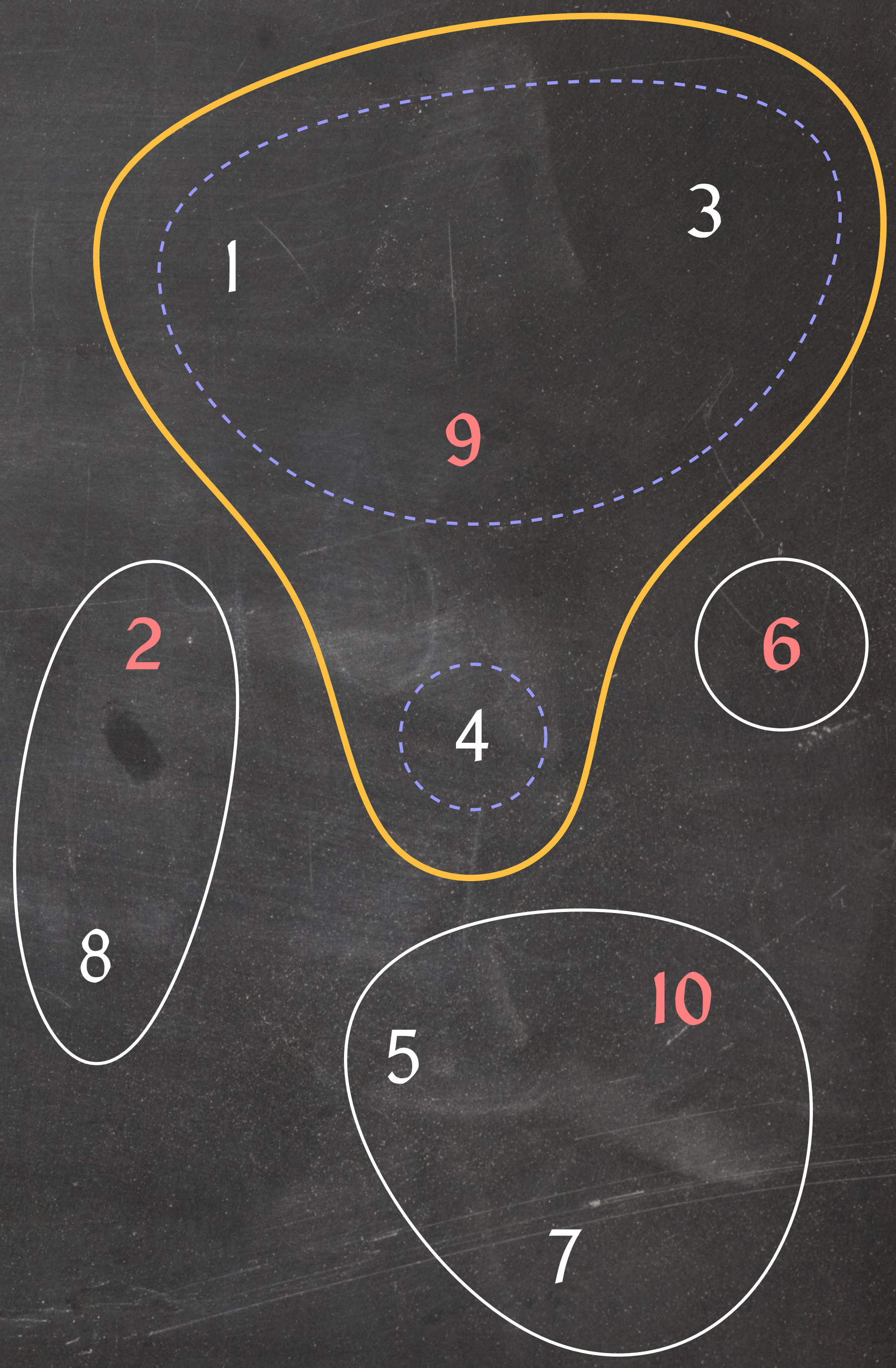
Given a set S of elements, maintain a partition of S into subsets $S_1, S_2, \ldots, S_k$.

Support the following operations:

Union(x, y): Replace sets S_i and S_j in the partition with $S_i \cup S_j$, where $x \in S_i$ and $y \in S_j$.

Find(x): Return a representative $r(S_i) \in S_i$ of the set S_i that contains x.

In particular, $\text{Find}(x) = \text{Find}(y)$ if and only if x and y belong to the same set.



Kruskal's Algorithm Using Union-Find

Idea: Maintain a partition of V into the vertex sets of the connected components of T.

Kruskal(G)

```
T = (V, ∅)
initialize a union-find structure D for V with every vertex v ∈ V in its own set
sort the edges in G by increasing weight
for every edge (v, w) of G, in sorted order
    do if D.find(v) ≠ D.find(w)
          then add (v, w) to T
               D.union(v, w)
return T
```

Kruskal's Algorithm Using Union-Find

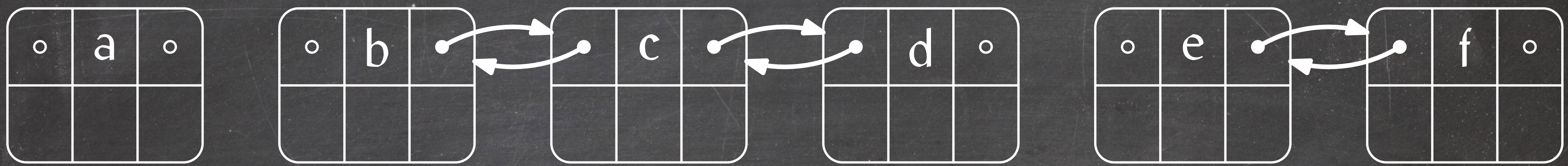
Idea: Maintain a partition of V into the vertex sets of the connected components of T.

Kruskal(G)

```
T = (V, ∅)
initialize a union-find structure D for V with every vertex v ∈ V in its own set
sort the edges in G by increasing weight
for every edge (v, w) of G, in sorted order
   do if D.find(v) ≠ D.find(w)
         then add (v, w) to T
              D.union(v, w)
return T
```

Lemma: Kruskal's algorithm takes $O(m \lg m)$ time plus the cost of $2m$ Find and $n - 1$ Union operations.

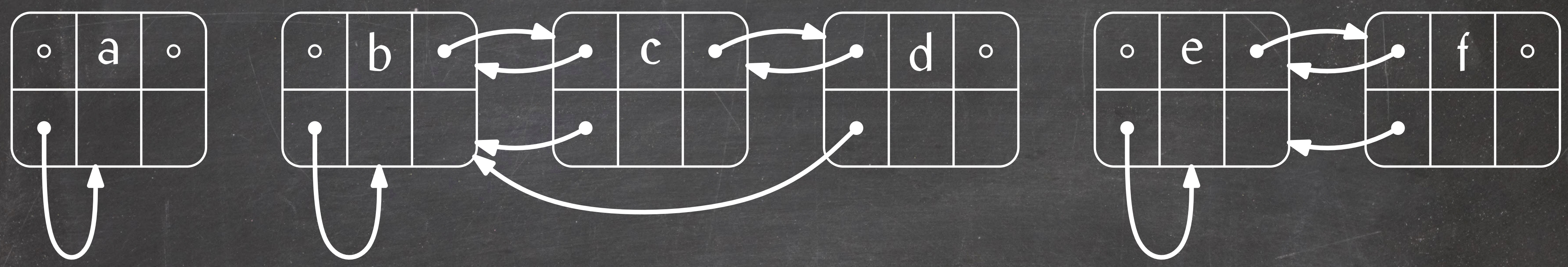
A Simple Union-Find Structure



List node:

- A set element
- Pointers to predecessor and successor
- Pointer to head of the list
- Pointer to tail of the list (only valid for head node)
- Size of the list (only valid for head node)

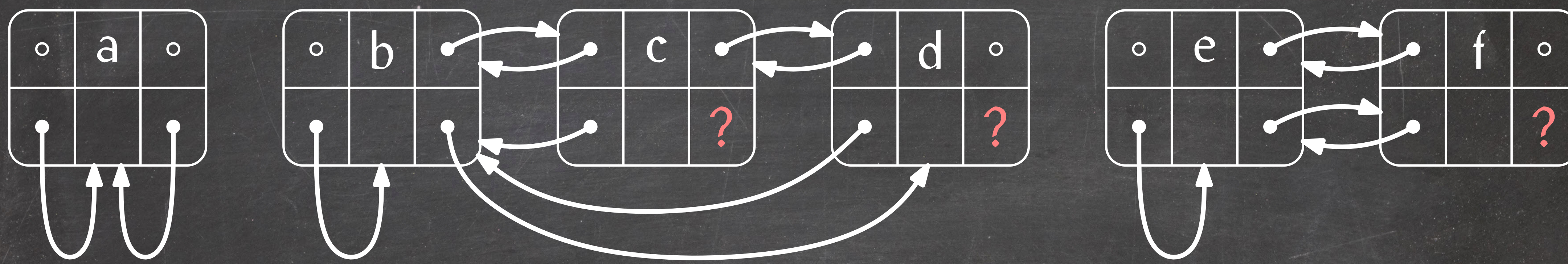
A Simple Union-Find Structure



List node:

- A set element
- Pointers to predecessor and successor
- Pointer to head of the list
- Pointer to tail of the list (only valid for head node)
- Size of the list (only valid for head node)

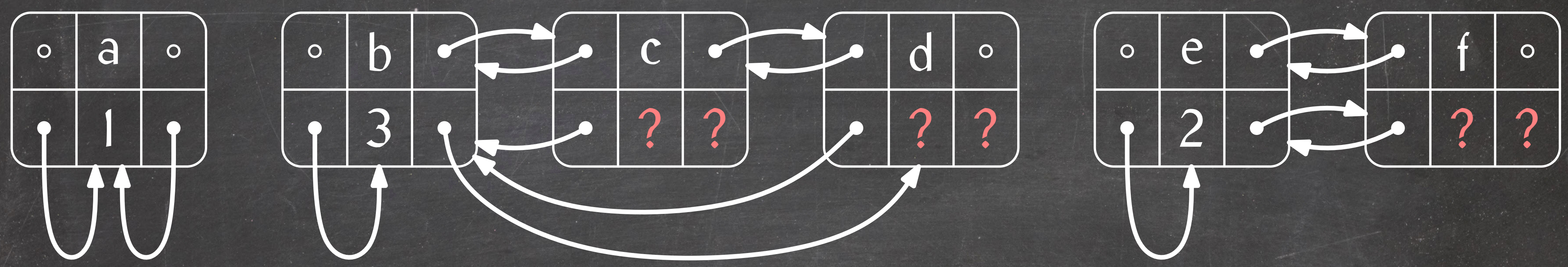
A Simple Union-Find Structure



List node:

- A set element
- Pointers to predecessor and successor
- Pointer to head of the list
- Pointer to tail of the list (only valid for head node)
- Size of the list (only valid for head node)

A Simple Union-Find Structure



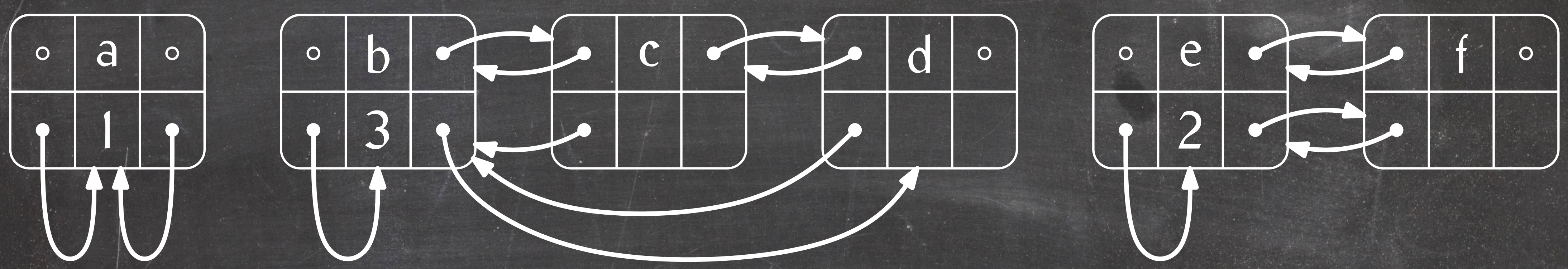
List node:

- A set element
- Pointers to predecessor and successor
- Pointer to head of the list
- Pointer to tail of the list (only valid for head node)
- Size of the list (only valid for head node)

Find

D.find(x)

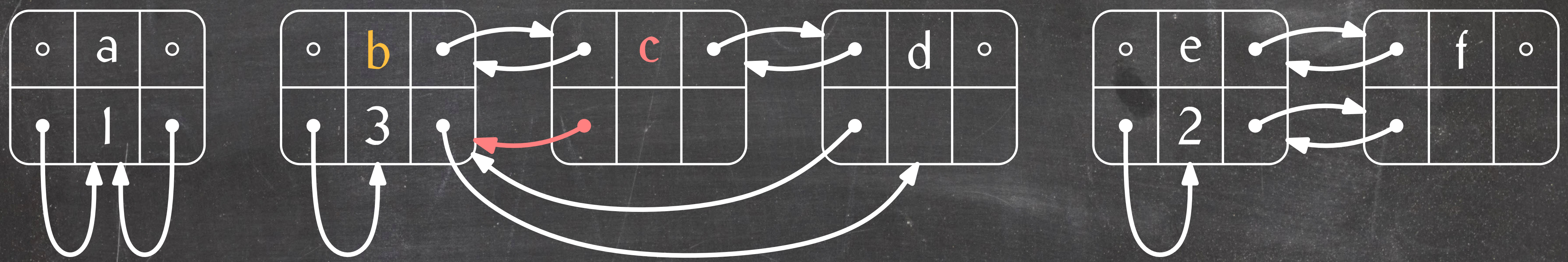
```
return x.head.key
```



Find

D.find(x)

```
return x.head.key
```

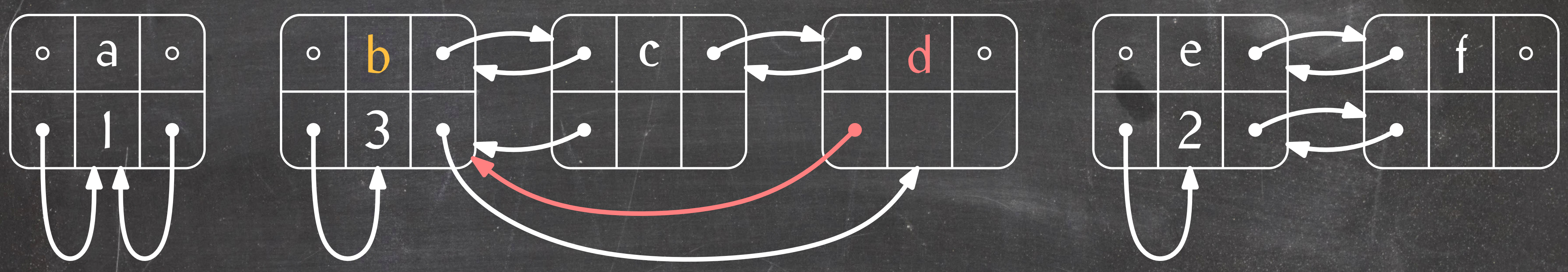


D.find(c) = b

Find

D.find(x)

```
return x.head.key
```



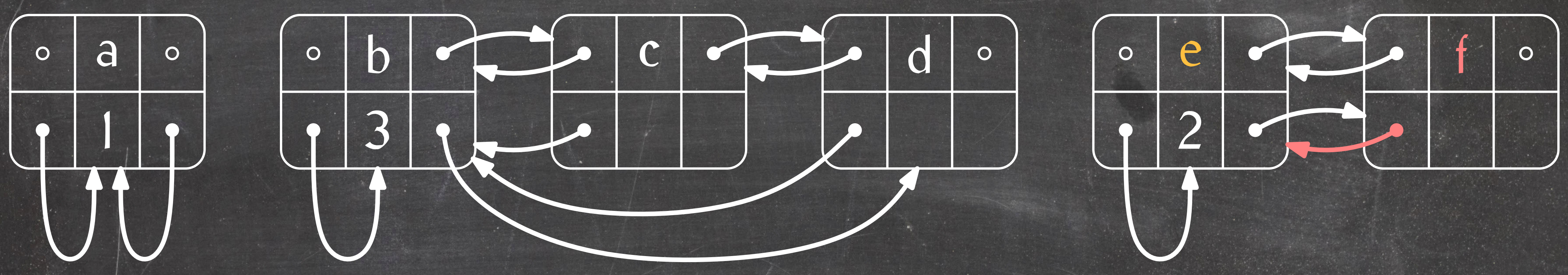
D.find(c) = b

D.find(d) = b

Find

D.find(x)

```
return x.head.key
```



D.find(c) = b

D.find(d) = b

D.find(e) = e

Union

D.union(x, y)

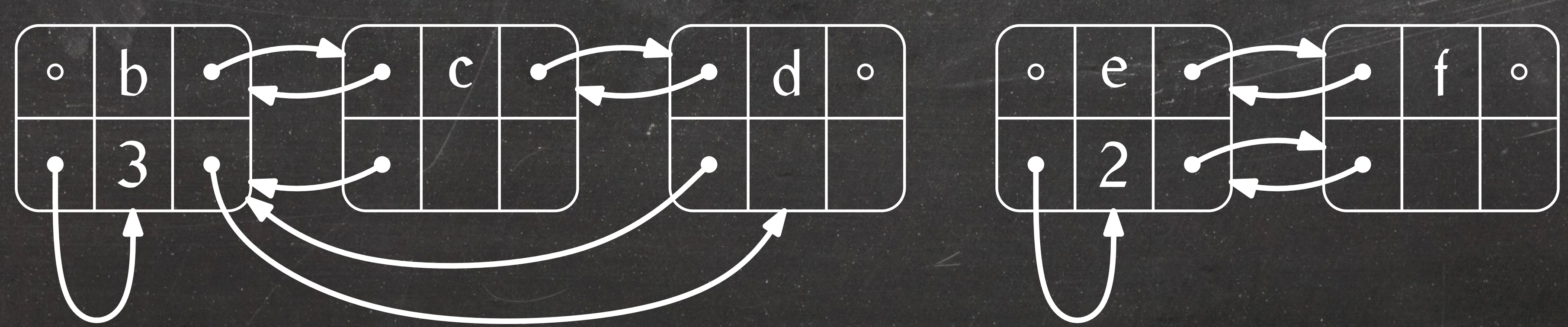
```
if x.head.listSize < y.head.listSize
    then swap x and y
y.head.pred = x.head.tail
x.head.tail.succ = y.head
x.head.listSize = x.head.listSize + y.head.listSize
x.head.tail = y.head.tail
z = y.head
while z ≠ null
    do z.head = x.head
       z = z.succ
```

Union

D.union(x, y)

```
if x.head.listSize < y.head.listSize
    then swap x and y
y.head.pred = x.head.tail
x.head.tail.succ = y.head
x.head.listSize = x.head.listSize + y.head.listSize
x.head.tail = y.head.tail
z = y.head
while z ≠ null
    do z.head = x.head
       z = z.succ
```

D.union(c, e):

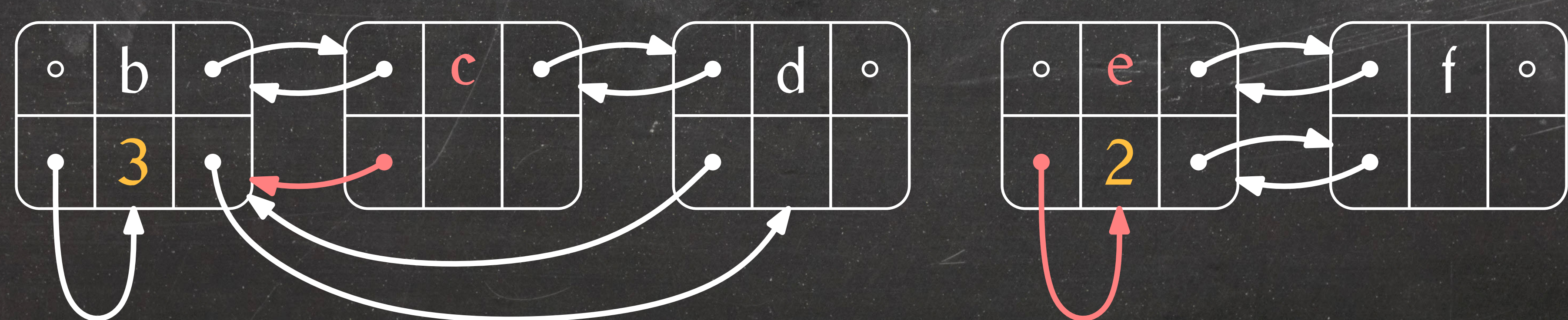


Union

D.union(x, y)

```
if x.head.listSize < y.head.listSize
    then swap x and y
y.head.pred = x.head.tail
x.head.tail.succ = y.head
x.head.listSize = x.head.listSize + y.head.listSize
x.head.tail = y.head.tail
z = y.head
while z ≠ null
    do z.head = x.head
       z = z.succ
```

D.union(c, e):

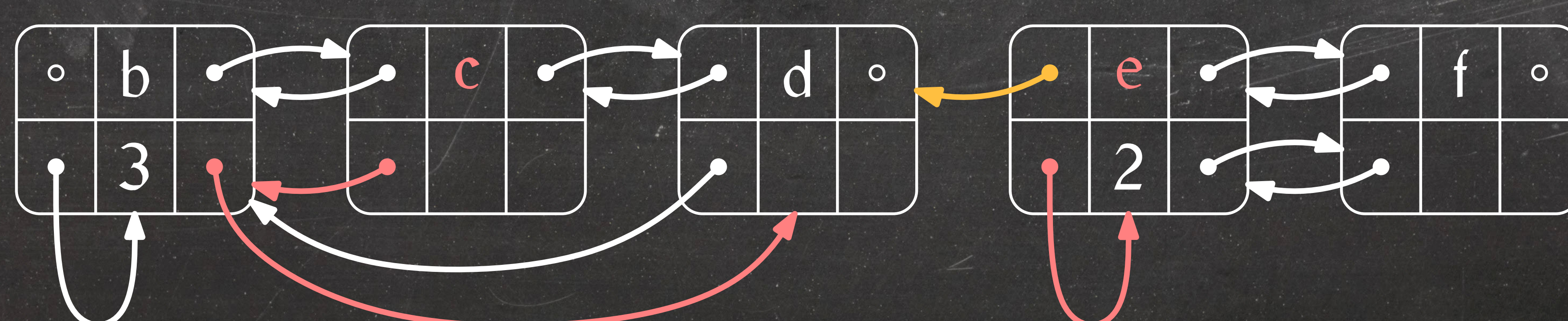


Union

D.union(x, y)

```
if x.head.listSize < y.head.listSize
    then swap x and y
y.head.pred = x.head.tail
x.head.tail.succ = y.head
x.head.listSize = x.head.listSize + y.head.listSize
x.head.tail = y.head.tail
z = y.head
while z ≠ null
    do z.head = x.head
       z = z.succ
```

D.union(c, e):

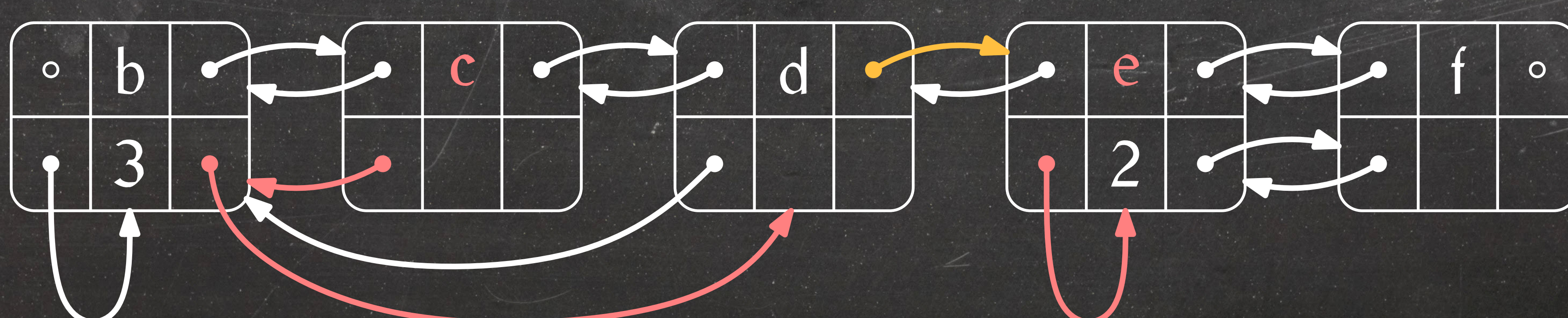


Union

D.union(x, y)

```
if x.head.listSize < y.head.listSize
    then swap x and y
y.head.pred = x.head.tail
x.head.tail.succ = y.head
x.head.listSize = x.head.listSize + y.head.listSize
x.head.tail = y.head.tail
z = y.head
while z ≠ null
    do z.head = x.head
       z = z.succ
```

D.union(c, e):

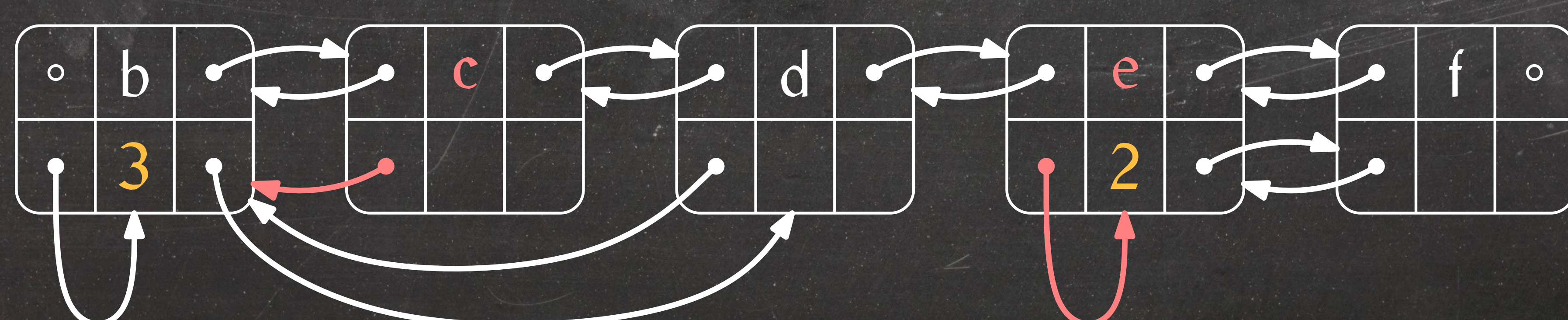


Union

D.union(x, y)

```
if x.head.listSize < y.head.listSize
    then swap x and y
y.head.pred = x.head.tail
x.head.tail.succ = y.head
x.head.listSize = x.head.listSize + y.head.listSize
x.head.tail = y.head.tail
z = y.head
while z ≠ null
    do z.head = x.head
       z = z.succ
```

D.union(c, e):

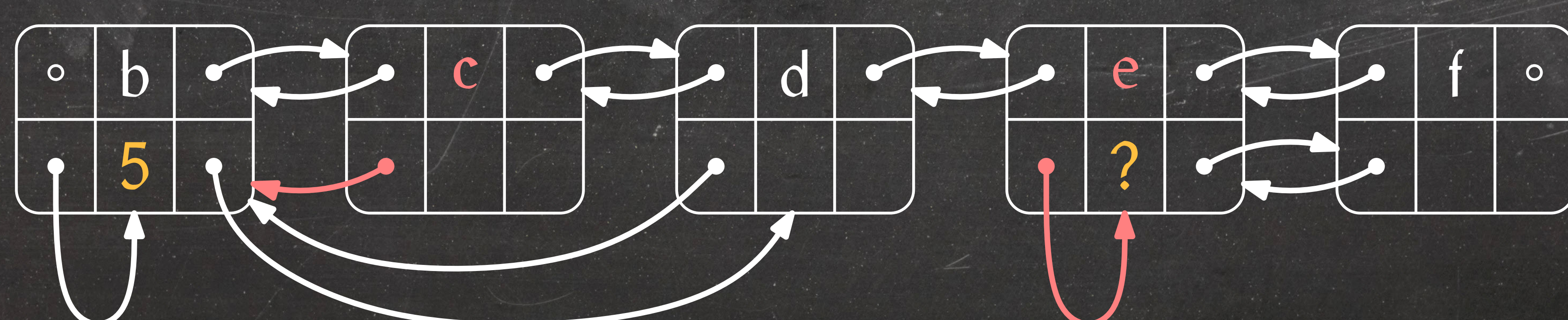


Union

D.union(x, y)

```
if x.head.listSize < y.head.listSize
    then swap x and y
y.head.pred = x.head.tail
x.head.tail.succ = y.head
x.head.listSize = x.head.listSize + y.head.listSize
x.head.tail = y.head.tail
z = y.head
while z ≠ null
    do z.head = x.head
       z = z.succ
```

D.union(c, e):

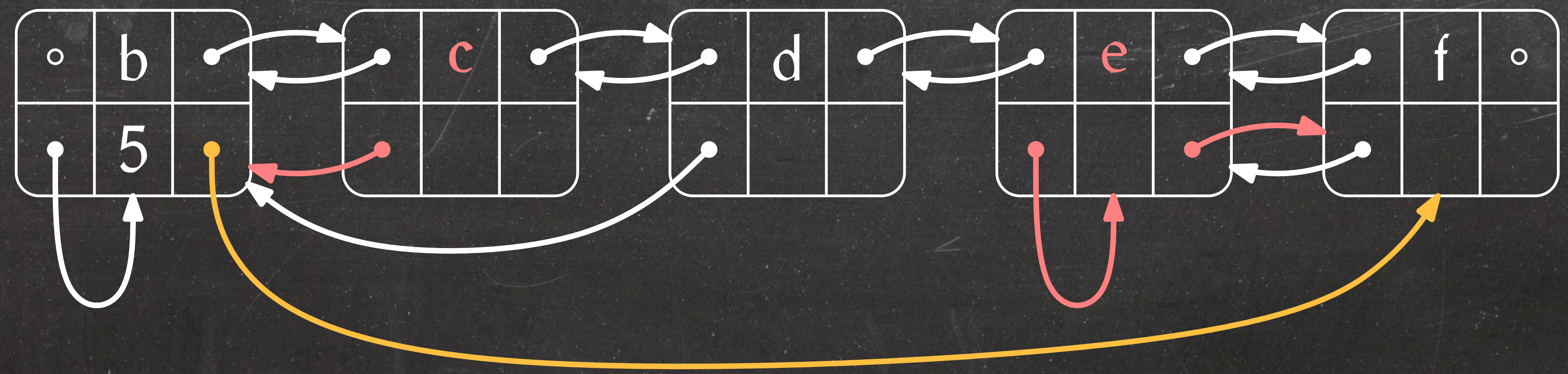


Union

D.union(x, y)

```
if x.head.listSize < y.head.listSize
    then swap x and y
y.head.pred = x.head.tail
x.head.tail.succ = y.head
x.head.listSize = x.head.listSize + y.head.listSize
x.head.tail = y.head.tail
z = y.head
while z ≠ null
    do z.head = x.head
       z = z.succ
```

D.union(c, e):

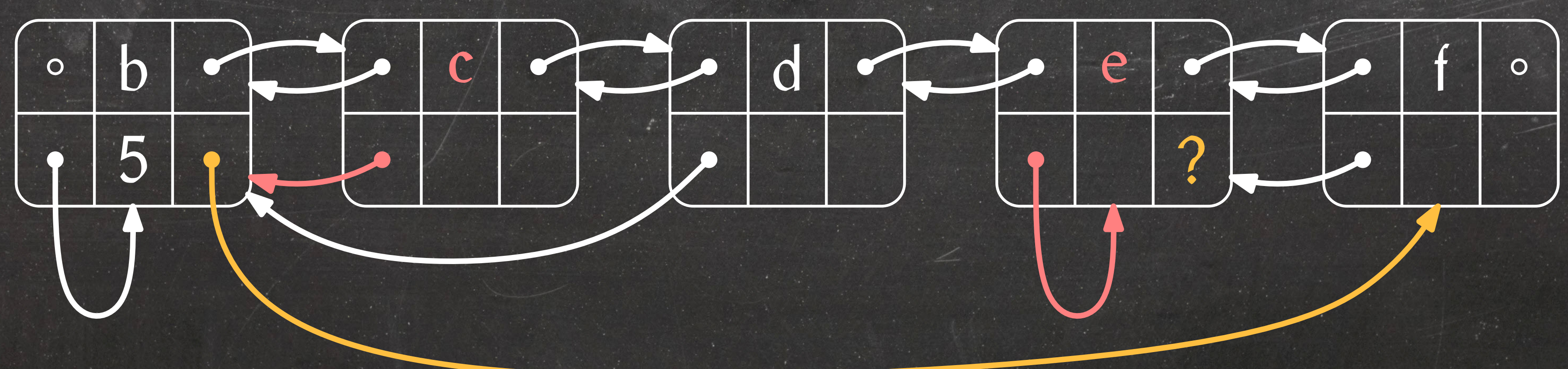


Union

D.union(x, y)

```
if x.head.listSize < y.head.listSize
    then swap x and y
y.head.pred = x.head.tail
x.head.tail.succ = y.head
x.head.listSize = x.head.listSize + y.head.listSize
x.head.tail = y.head.tail
z = y.head
while z ≠ null
    do z.head = x.head
       z = z.succ
```

D.union(c, e):

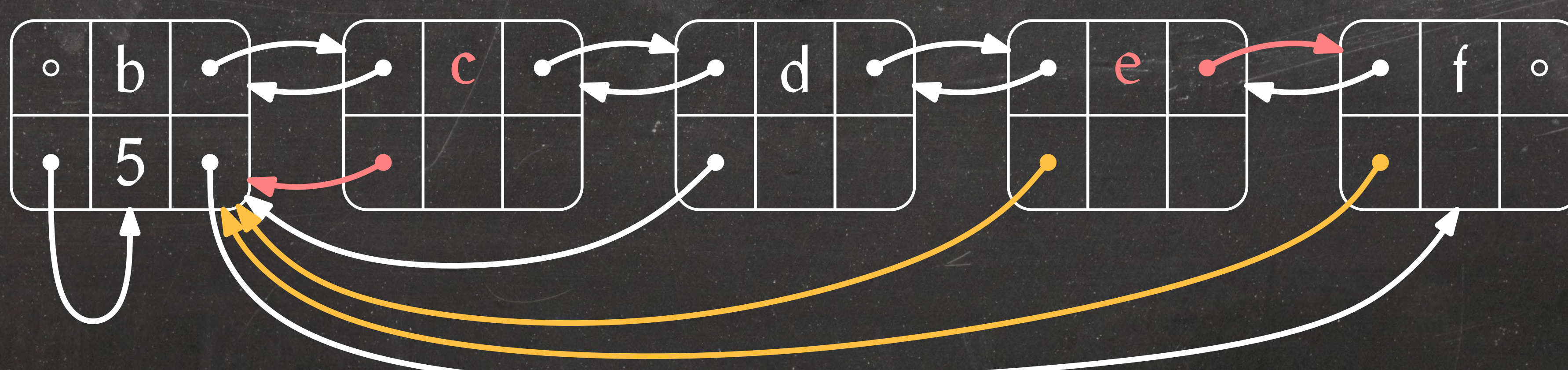


Union

D.union(x, y)

```
if x.head.listSize < y.head.listSize
   then swap x and y
y.head.pred = x.head.tail
x.head.tail.succ = y.head
x.head.listSize = x.head.listSize + y.head.listSize
x.head.tail = y.head.tail
z = y.head
while z ≠ null
   do z.head = x.head
      z = z.succ
```

D.union(c, e):

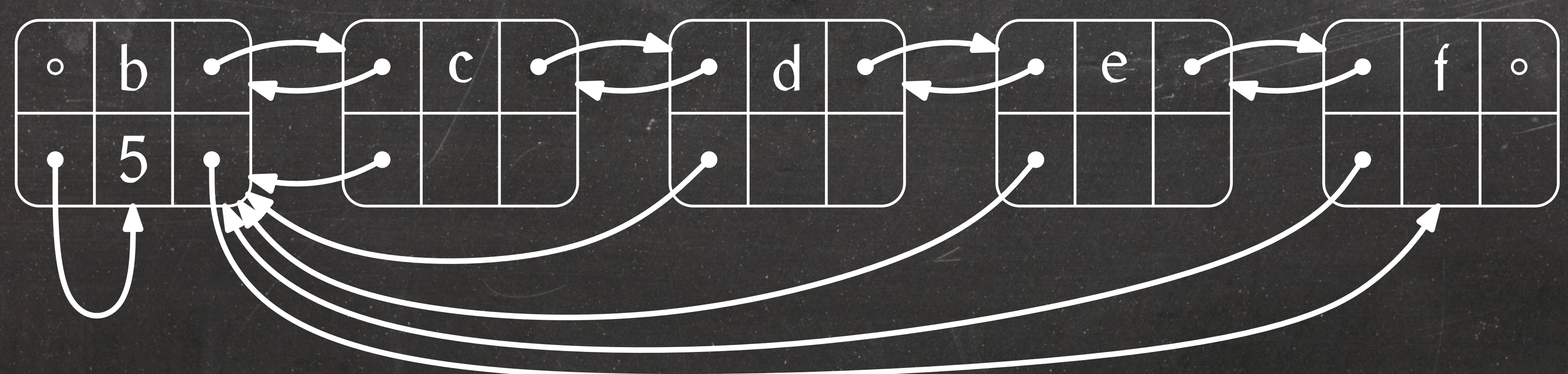


Union

D.union(x, y)

```
if x.head.listSize < y.head.listSize
    then swap x and y
y.head.pred = x.head.tail
x.head.tail.succ = y.head
x.head.listSize = x.head.listSize + y.head.listSize
x.head.tail = y.head.tail
z = y.head
while z ≠ null
    do z.head = x.head
       z = z.succ
```

D.union(c, e):



Analysis

Observation: A Find operation takes constant time.

Analysis

Observation: A Find operation takes constant time.

Observation: A Union operation takes $O(1 + s)$ time, where s is the size of the smaller list.

Analysis

Observation: A Find operation takes constant time.

Observation: A Union operation takes $O(1 + s)$ time, where s is the size of the smaller list.

Corollary: The total cost of m operations over a base set S is $O\left(m + \sum_{x \in S} c(x)\right)$, where $c(x)$ is the number of times x is in the smaller list of a Union operation.

Analysis

Observation: A Find operation takes constant time.

Observation: A Union operation takes $O(1 + s)$ time, where s is the size of the smaller list.

Corollary: The total cost of m operations over a base set S is $O\left(m + \sum_{x \in S} c(x)\right)$, where $c(x)$ is the number of times x is in the smaller list of a Union operation.

Lemma: Let $s(x, i)$ be the size of the list containing x after x was in the smaller list of i Union operations. Then $s(x, i) \geq 2^i$.

Analysis

Observation: A Find operation takes constant time.

Observation: A Union operation takes $O(1+s)$ time, where s is the size of the smaller list.

Corollary: The total cost of m operations over a base set S is $O\left(m + \sum_{x \in S} c(x)\right)$, where $c(x)$ is the number of times x is in the smaller list of a Union operation.

Lemma: Let $s(x,i)$ be the size of the list containing x after x was in the smaller list of i Union operations. Then $s(x,i) \geq 2^i$.

Base case: $i = 0$. The list containing x has size at least $1 = 2^0$.

Analysis

Observation: A Find operation takes constant time.

Observation: A Union operation takes $O(1+s)$ time, where s is the size of the smaller list.

Corollary: The total cost of m operations over a base set S is $O\left(m + \sum_{x \in S} c(x)\right)$, where $c(x)$ is the number of times x is in the smaller list of a Union operation.

Lemma: Let $s(x,i)$ be the size of the list containing x after x was in the smaller list of i Union operations. Then $s(x,i) \geq 2^i$.

Base case: $i = 0$. The list containing x has size at least $1 = 2^0$.

Inductive step: $i > 0$.

- Consider the ith Union operation where x is in the smaller list.
- Let S_1 and S_2 be the two unioned lists and assume $x \in S_2$.
- Then $|S_1| \geq |S_2| \geq 2^{i-1}$.
- Thus, $|S_1 \cup S_2| \geq 2^i$.

Analysis

Observation: A Find operation takes constant time.

Observation: A Union operation takes $O(1 + s)$ time, where s is the size of the smaller list.

Corollary: The total cost of m operations over a base set S is $O\left(m + \sum_{x \in S} c(x)\right)$, where $c(x)$ is the number of times x is in the smaller list of a Union operation.

Lemma: Let $s(x, i)$ be the size of the list containing x after x was in the smaller list of i Union operations. Then $s(x, i) \geq 2^i$.

Base case: $i = 0$. The list containing x has size at least $1 = 2^0$.

Inductive step: $i > 0$.

- Consider the ith Union operation where x is in the smaller list.
- Let S_1 and S_2 be the two unioned lists and assume $x \in S_2$.
- Then $|S_1| \geq |S_2| \geq 2^{i-1}$.
- Thus, $|S_1 \cup S_2| \geq 2^i$.

Corollary: $c(x) \leq \lg n$ for all $x \in S$.

Analysis

Corollary: A sequence of m Union and Find operations over a base set of size n takes $O(n \lg n + m)$ time.

Analysis

Corollary: A sequence of m Union and Find operations over a base set of size n takes $O(n \lg n + m)$ time.

Corollary: Kruskal's algorithm takes $O(n \lg n + m \lg m)$ time.

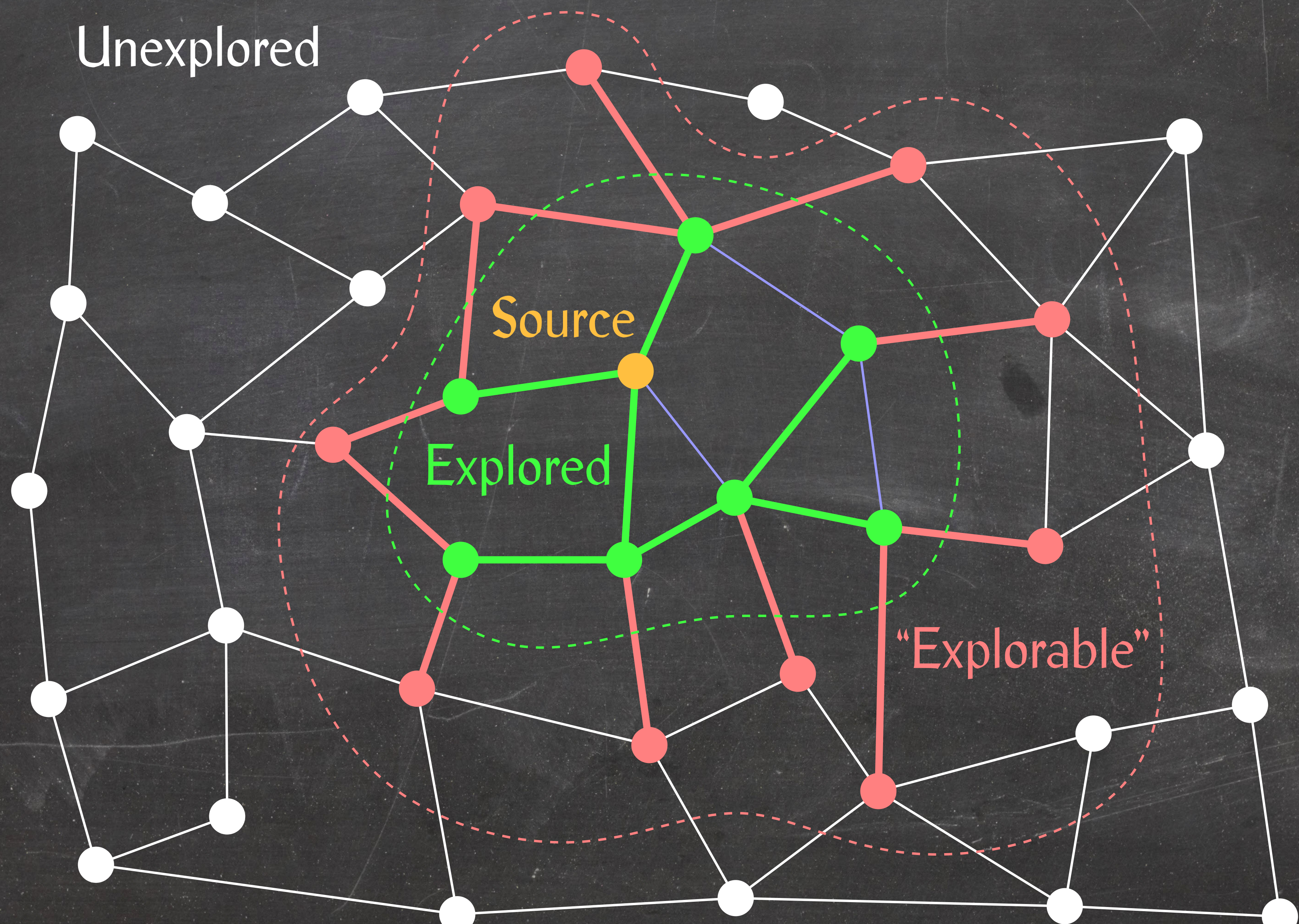
Analysis

Corollary: A sequence of m Union and Find operations over a base set of size n takes $O(n \lg n + m)$ time.

Corollary: Kruskal's algorithm takes $O(n \lg n + m \lg m)$ time.

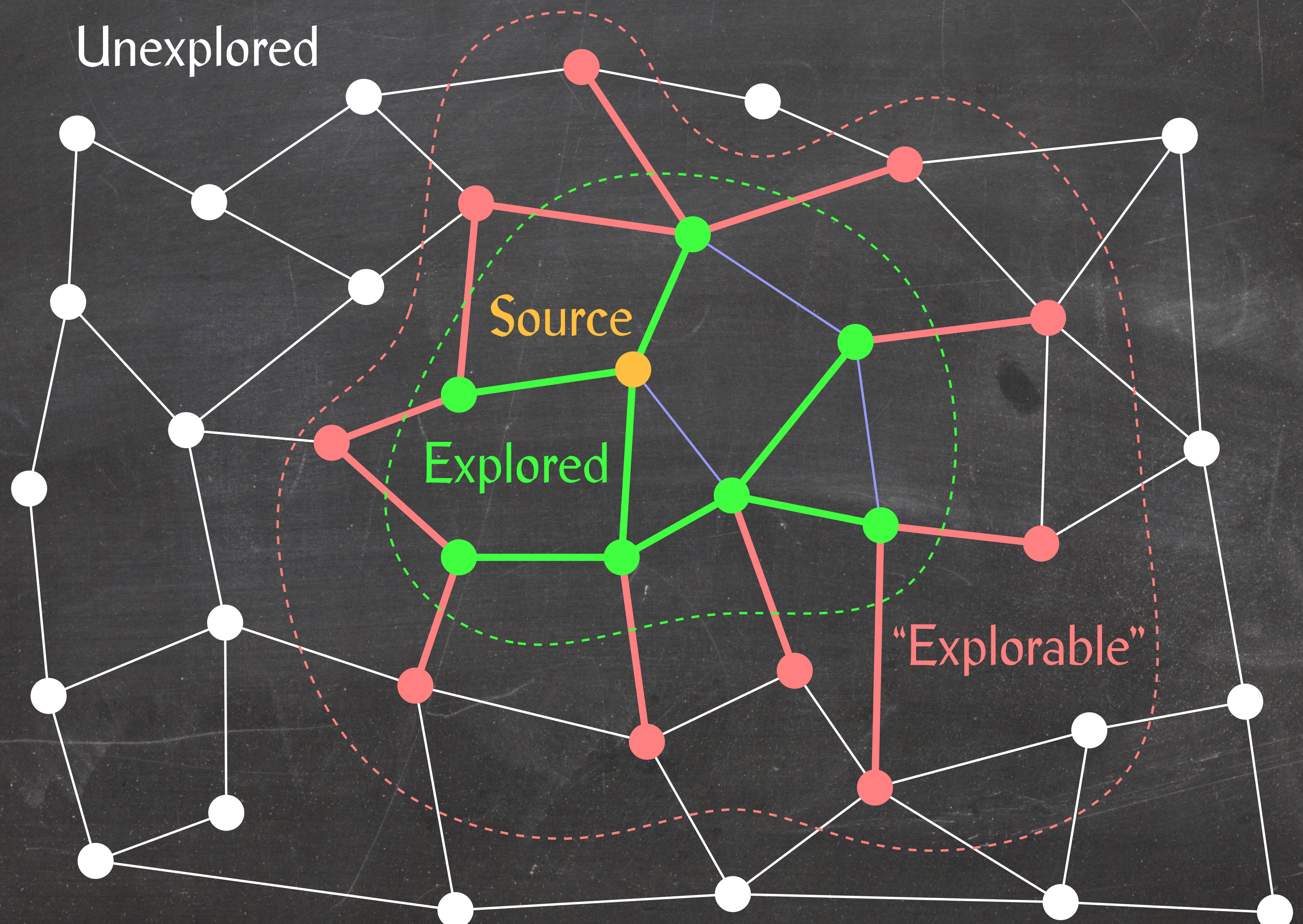
If the graph is connected, then $m \geq n - 1$, so the running time simplifies to $O(m \lg m)$.

The Cut Theorem And Graph Traversal



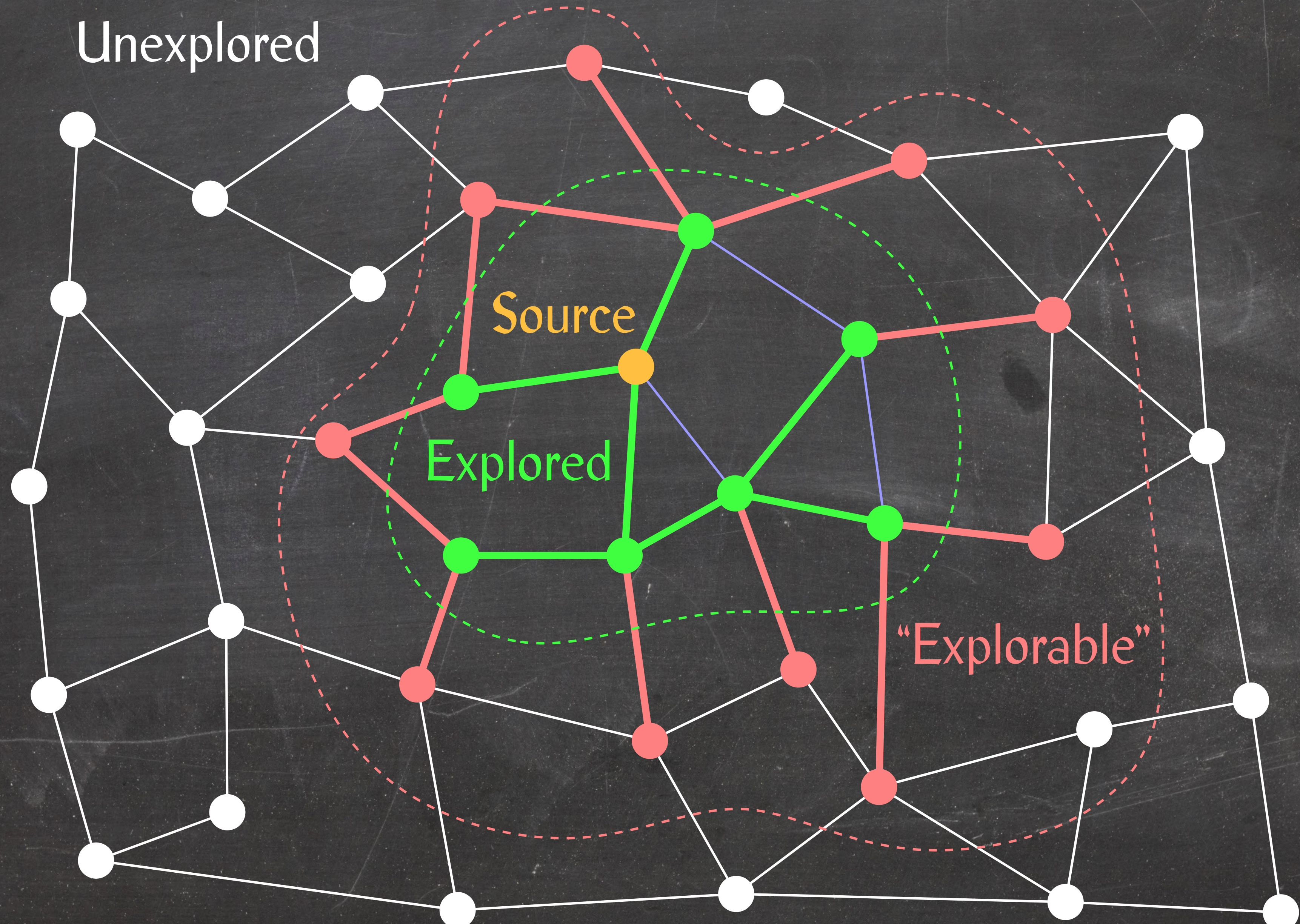
The Cut Theorem And Graph Traversal

If there exists an MST containing all green edges, then there exists an MST containing all green edges and the cheapest red edge.



The Cut Theorem And Graph Traversal

If there exists an MST containing all green edges, then there exists an MST containing all green edges and the cheapest red edge.



Cut: U = explored vertices, W = V \ U

Prim's Algorithm

Prim(G)

```
T = (V, ∅)
mark all vertices of G as unexplored
mark an arbitrary vertex s as explored
while not all vertices are explored
    do pick the cheapest edge e with exactly one unexplored endpoint v
       mark v as explored
       add e to T
return T
```

Prim's Algorithm

Prim(G)

```
T = (V, ∅)
mark all vertices of G as unexplored
mark an arbitrary vertex s as explored
while not all vertices are explored
    do pick the cheapest edge e with exactly one unexplored endpoint v
       mark v as explored
       add e to T
return T
```

Lemma: Prim's algorithm computes a minimum spanning tree.

Prim's Algorithm

Prim(G)

```
T = (V, ∅)
mark all vertices of G as unexplored
mark an arbitrary vertex s as explored
while not all vertices are explored
   do pick the cheapest edge e with exactly one unexplored endpoint v
      mark v as explored
      add e to T
return T
```

Lemma: Prim's algorithm computes a minimum spanning tree.

By induction on the number of edges in T, there exists an MST $T^* \supseteq T$.

Prim's Algorithm

Prim(G)

```
T = (V, ∅)
mark all vertices of G as unexplored
mark an arbitrary vertex s as explored
while not all vertices are explored
   do pick the cheapest edge e with exactly one unexplored endpoint v
      mark v as explored
      add e to T
return T
```

Lemma: Prim's algorithm computes a minimum spanning tree.

By induction on the number of edges in T, there exists an MST $T^* \supseteq T$.

Once T is connected, we have $T^* = T$.

The Abstract Data Type Priority Queue

Operations:

Q.insert(x, p):	Insert element x with priority p
Q.delete(x):	Delete element x
Q.findMin():	Find and return the element with minimum priority
Q.deleteMin():	Delete the element with minimum priority and return it
Q.decreaseKey(x, p):	Change the priority p_x of x to min(p, p_x)

Delete and DecreaseKey assume they're given a pointer to the place in Q where x is stored.

The Abstract Data Type Priority Queue

Operations:

Q.insert(x, p):	Insert element x with priority p
Q.delete(x):	Delete element x
Q.findMin():	Find and return the element with minimum priority
Q.deleteMin():	Delete the element with minimum priority and return it
Q.decreaseKey(x, p):	Change the priority p_x of x to $\min(p, p_x)$

Delete and DecreaseKey assume they're given a pointer to the place in Q where x is stored.

Example: A binary heap is a priority queue supporting all operations in $O(\lg |Q|)$ time.

Prim's Algorithm Using A Priority Queue

Prim(G)

```
T = (V, ∅)
mark every vertex of G as unexplored
mark an arbitrary vertex s as explored
Q = an empty priority queue
for every edge (s, v) incident to s
    do Q.insert((s, v), w(s, v))
while not Q.isEmpty()
    do (u, v) = Q.deleteMin()
        if v is unexplored
            then mark v as explored
                add edge (u, v) to T
                for every edge (v, w) incident to v
                    do Q.insert((v, w), w(v, w))
return T
```

Prim's Algorithm Using A Priority Queue

Invariant: Q contains all edges with exactly one unexplored endpoint.

Prim(G)

```
T = (V, ∅)
mark every vertex of G as unexplored
mark an arbitrary vertex s as explored
Q = an empty priority queue
for every edge (s, v) incident to s
    do Q.insert((s, v), w(s, v))
while not Q.isEmpty()
    do (u, v) = Q.deleteMin()
        if v is unexplored
            then mark v as explored
                add edge (u, v) to T
                for every edge (v, w) incident to v
                    do Q.insert((v, w), w(v, w))
return T
```

Prim's Algorithm Using A Priority Queue

Prim(*G*)

```
T = (V, ∅)
mark every vertex of G as unexplored
mark an arbitrary vertex s as explored
Q = an empty priority queue
for every edge (s, v) incident to s
    do Q.insert((s, v), w(s, v))
while not Q.isEmpty()
    do (u, v) = Q.deleteMin()
        if v is unexplored
            then mark v as explored
                 add edge (u, v) to T
                 for every edge (v, w) incident to v
                     do Q.insert((v, w), w(v, w))
return T
```

Invariant: Q contains all edges with exactly one unexplored endpoint.

$\Rightarrow$ This version of Prim's algorithm computes an MST.

Prim's Algorithm Using A Priority Queue

Prim(G)

```
T = (V, ∅)
mark every vertex of G as unexplored
mark an arbitrary vertex s as explored
Q = an empty priority queue
for every edge (s, v) incident to s
    do Q.insert((s, v), w(s, v))
while not Q.isEmpty()
    do (u, v) = Q.deleteMin()
        if v is unexplored
            then mark v as explored
                add edge (u, v) to T
                for every edge (v, w) incident to v
                    do Q.insert((v, w), w(v, w))
return T
```

Invariant: Q contains all edges with exactly one unexplored endpoint.

$\Rightarrow$ This version of Prim's algorithm computes an MST.

This version of Prim's algorithm takes O(m lg m) time:

Prim's Algorithm Using A Priority Queue

Prim(G)

```
T = (V, ∅)
mark every vertex of G as unexplored
mark an arbitrary vertex s as explored
Q = an empty priority queue
for every edge (s, v) incident to s
    do Q.insert((s, v), w(s, v))
while not Q.isEmpty()
    do (u, v) = Q.deleteMin()
        if v is unexplored
            then mark v as explored
                add edge (u, v) to T
                for every edge (v, w) incident to v
                    do Q.insert((v, w), w(v, w))
return T
```

Invariant: Q contains all edges with exactly one unexplored endpoint.

$\Rightarrow$ This version of Prim's algorithm computes an MST.

This version of Prim's algorithm takes $O(m \lg m)$ time:

Every edge is inserted into Q once.

Prim's Algorithm Using A Priority Queue

Prim(G)

```
T = (V, ∅)
mark every vertex of G as unexplored
mark an arbitrary vertex s as explored
Q = an empty priority queue
for every edge (s, v) incident to s
    do Q.insert((s, v), w(s, v))
while not Q.isEmpty()
    do (u, v) = Q.deleteMin()
        if v is unexplored
            then mark v as explored
                 add edge (u, v) to T
                 for every edge (v, w) incident to v
                     do Q.insert((v, w), w(v, w))
return T
```

Invariant: Q contains all edges with exactly one unexplored endpoint.

$\Rightarrow$ This version of Prim's algorithm computes an MST.

This version of Prim's algorithm takes $O(m \lg m)$ time:

Every edge is inserted into Q once.

$\Rightarrow$ Every edge is removed from Q once.

Prim's Algorithm Using A Priority Queue

Prim(G)

```
T = (V, ∅)
mark every vertex of G as unexplored
mark an arbitrary vertex s as explored
Q = an empty priority queue
for every edge (s, v) incident to s
    do Q.insert((s, v), w(s, v))
while not Q.isEmpty()
    do (u, v) = Q.deleteMin()
        if v is unexplored
            then mark v as explored
                 add edge (u, v) to T
                 for every edge (v, w) incident to v
                     do Q.insert((v, w), w(v, w))
return T
```

Invariant: Q contains all edges with exactly one unexplored endpoint.

$\Rightarrow$ This version of Prim's algorithm computes an MST.

This version of Prim's algorithm takes $O(m \lg m)$ time:

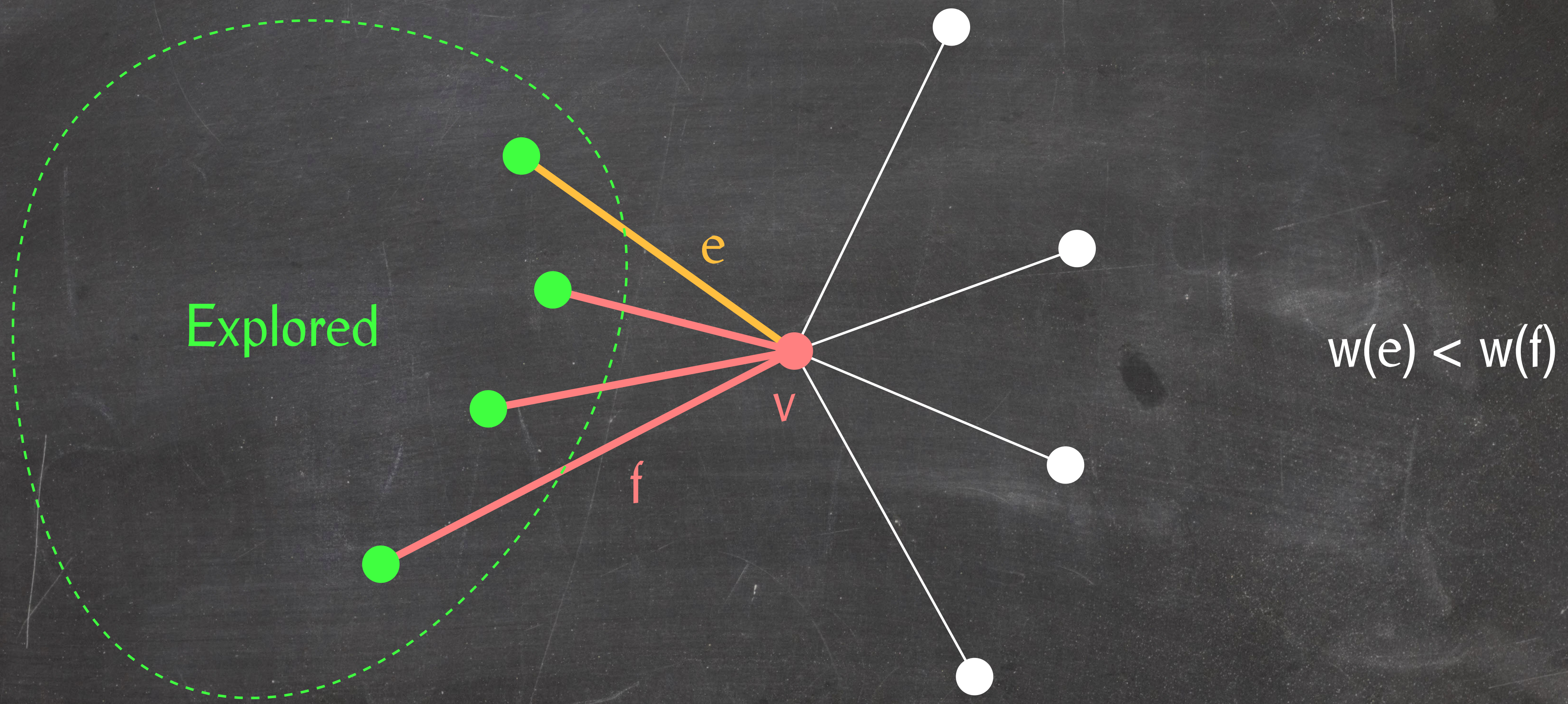
Every edge is inserted into Q once.

$\Rightarrow$ Every edge is removed from Q once.

$\Rightarrow$ 2m priority queue operations.

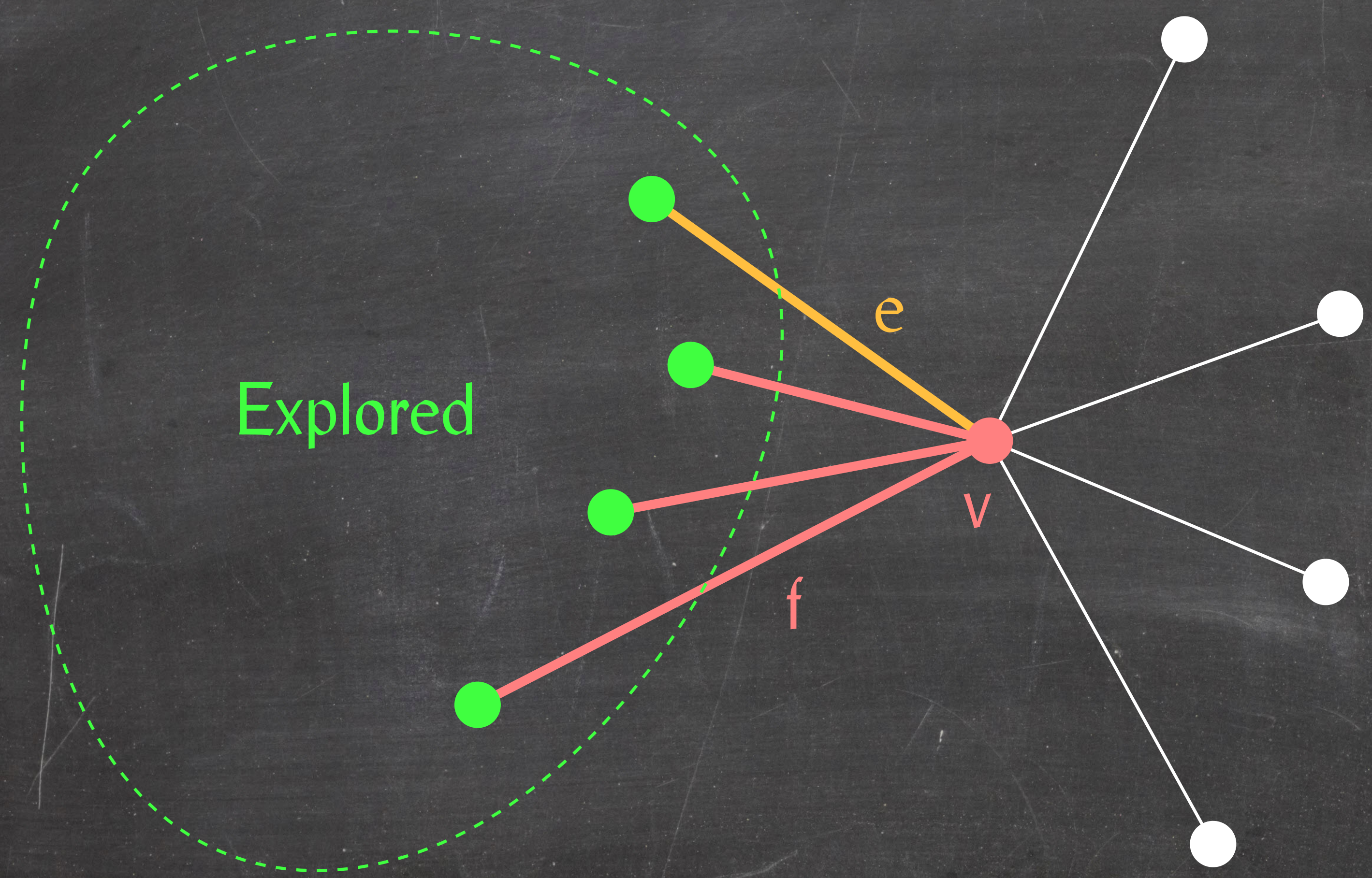
Most Edges In Q Are Useless

Observation: Of all the edges connecting an unexplored vertex to explored vertices only the cheapest has a chance of being added to the MST.



Most Edges In Q Are Useless

Observation: Of all the edges connecting an unexplored vertex to explored vertices only the cheapest has a chance of being added to the MST.

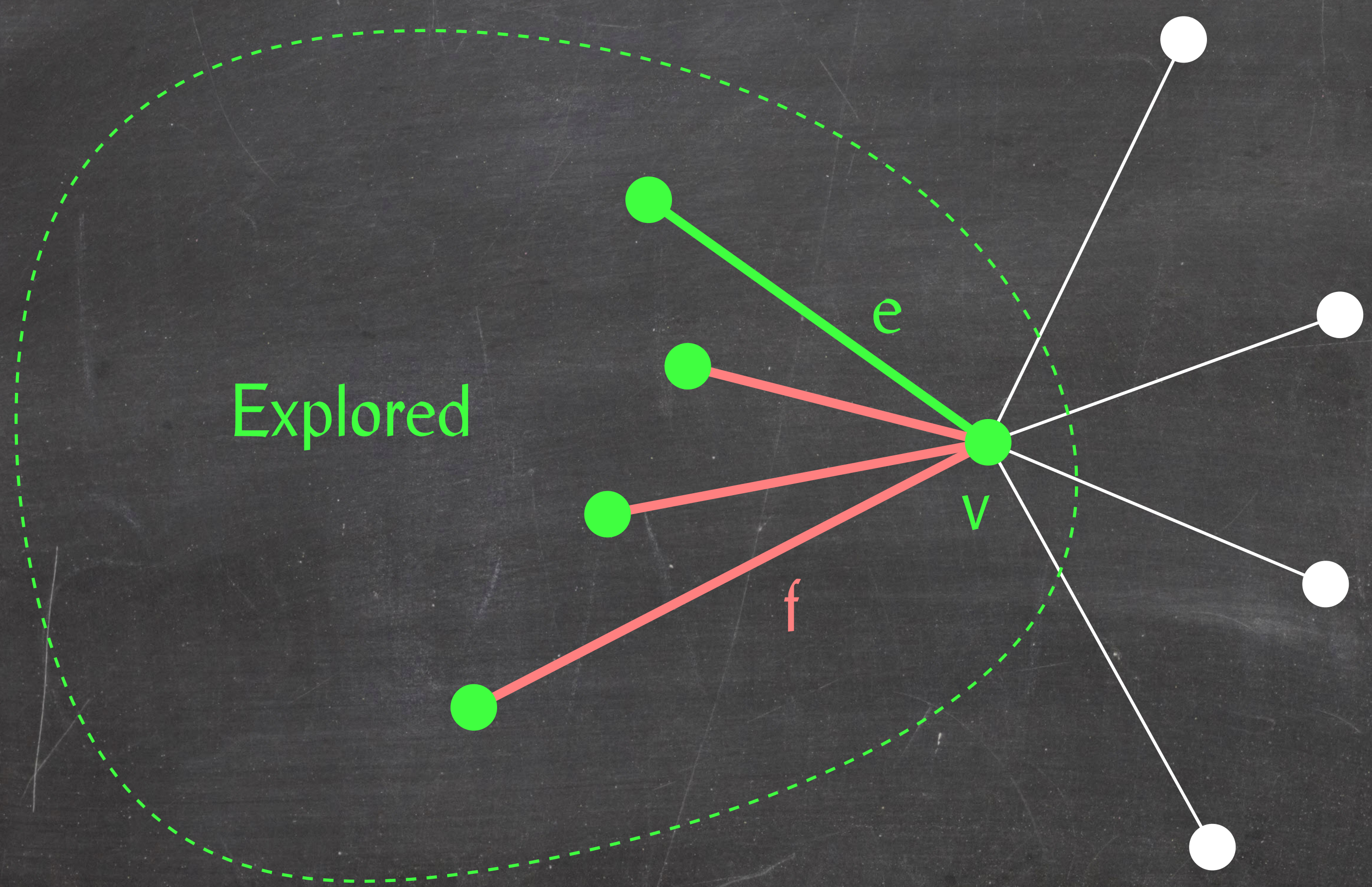


$w(e) < w(f)$

While v is unexplored, all red and orange edges are in Q, so none of the red edges can be the first edge to be removed from Q.

Most Edges In Q Are Useless

Observation: Of all the edges connecting an unexplored vertex to explored vertices only the cheapest has a chance of being added to the MST.



$w(e) < w(f)$

While v is unexplored, all red and orange edges are in Q, so none of the red edges can be the first edge to be removed from Q.

After marking v as explored, both endpoints of red edges are explored, so they cannot be added to T either.

A Faster Version Of Prim's Algorithm

Prim(G)

```
T = (V, ∅)
mark every vertex of G as unexplored
set e(v) = nil for every vertex v ∈ G
mark an arbitrary vertex s as explored
Q = an empty priority queue
for every edge (s, v) incident to s
    do Q.insert(v, w(s, v))
       e(v) = (s, v)
while not Q.isEmpty()
    do u = Q.deleteMin()
       mark u as explored
       add e(u) to T
       for every edge (u, v) incident to u
           do if v is unexplored and (v ∉ Q or w(u, v) < w(e(v)))
                  then if v ∉ Q
                           then Q.insert(v, w(u, v))
                           else Q.decreaseKey(v, w(u, v))
                       e(v) = (u, v)
return T
```

A Faster Version Of Prim's Algorithm

Prim(G)

```
T = (V, ∅)
mark every vertex of G as unexplored
set e(v) = nil for every vertex v ∈ G
mark an arbitrary vertex s as explored
Q = an empty priority queue
for every edge (s, v) incident to s
    do Q.insert(v, w(s, v))
        e(v) = (s, v)
while not Q.isEmpty()
    do u = Q.deleteMin()
        mark u as explored
        add e(u) to T
        for every edge (u, v) incident to u
            do if v is unexplored and (v ∉ Q or w(u, v) < w(e(v)))
                    then if v ∉ Q
                            then Q.insert(v, w(u, v))
                            else Q.decreaseKey(v, w(u, v))
                        e(v) = (u, v)
return T
```

This version of Prim's algorithm also takes $O(m \lg m)$ time:

A Faster Version Of Prim's Algorithm

Prim(G)

```
T = (V, ∅)
mark every vertex of G as unexplored
set e(v) = nil for every vertex v ∈ G
mark an arbitrary vertex s as explored
Q = an empty priority queue
for every edge (s, v) incident to s
   do Q.insert(v, w(s, v))
      e(v) = (s, v)
while not Q.isEmpty()
   do u = Q.deleteMin()
      mark u as explored
      add e(u) to T
      for every edge (u, v) incident to u
         do if v is unexplored and (v ∉ Q or w(u, v) < w(e(v)))
               then if v ∉ Q
                       then Q.insert(v, w(u, v))
                       else Q.decreaseKey(v, w(u, v))
                    e(v) = (u, v)
return T
```

This version of Prim's algorithm also takes $O(m \lg m)$ time:

- n Insert operations

A Faster Version Of Prim's Algorithm

Prim(G)

```
T = (V, ∅)
mark every vertex of G as unexplored
set e(v) = nil for every vertex v ∈ G
mark an arbitrary vertex s as explored
Q = an empty priority queue
for every edge (s, v) incident to s
    do Q.insert(v, w(s, v))
       e(v) = (s, v)
while not Q.isEmpty()
    do u = Q.deleteMin()
       mark u as explored
       add e(u) to T
       for every edge (u, v) incident to u
           do if v is unexplored and (v ∉ Q or w(u, v) < w(e(v)))
                 then if v ∉ Q
                         then Q.insert(v, w(u, v))
                         else Q.decreaseKey(v, w(u, v))
                      e(v) = (u, v)
return T
```

This version of Prim's algorithm also takes $O(m \lg m)$ time:

- n Insert operations
- m – n DecreaseKey operations

A Faster Version Of Prim's Algorithm

Prim(G)

```
T = (V, ∅)
mark every vertex of G as unexplored
set e(v) = nil for every vertex v ∈ G
mark an arbitrary vertex s as explored
Q = an empty priority queue
for every edge (s, v) incident to s
   do Q.insert(v, w(s, v))
      e(v) = (s, v)
while not Q.isEmpty()
   do u = Q.deleteMin()
      mark u as explored
      add e(u) to T
      for every edge (u, v) incident to u
         do if v is unexplored and (v ∉ Q or w(u, v) < w(e(v)))
               then if v ∉ Q
                       then Q.insert(v, w(u, v))
                       else Q.decreaseKey(v, w(u, v))
                    e(v) = (u, v)
return T
```

This version of Prim's algorithm also takes $O(m \lg m)$ time:

- n Insert operations
- m – n DecreaseKey operations
- n DeleteMin operations

A Faster Version Of Prim's Algorithm

Prim(G)

```
T = (V, ∅)
mark every vertex of G as unexplored
set e(v) = nil for every vertex v ∈ G
mark an arbitrary vertex s as explored
Q = an empty priority queue
for every edge (s, v) incident to s
    do Q.insert(v, w(s, v))
       e(v) = (s, v)
while not Q.isEmpty()
    do u = Q.deleteMin()
       mark u as explored
       add e(u) to T
       for every edge (u, v) incident to u
           do if v is unexplored and (v ∉ Q or w(u, v) < w(e(v)))
                 then if v ∉ Q
                         then Q.insert(v, w(u, v))
                         else Q.decreaseKey(v, w(u, v))
                      e(v) = (u, v)
return T
```

This version of Prim's algorithm also takes $O(m \lg m)$ time:

- n Insert operations
- $m - n$ DecreaseKey operations
- n DeleteMin operations

$\Rightarrow$ $n + m$ priority queue operations.

A Faster Version Of Prim's Algorithm

```
Prim(G)
T = (V, ∅)
mark every vertex of G as unexplored
set e(v) = nil for every vertex v ∈ G
mark an arbitrary vertex s as explored
Q = an empty priority queue
for every edge (s, v) incident to s
    do Q.insert(v, w(s, v))
       e(v) = (s, v)
while not Q.isEmpty()
    do u = Q.deleteMin()
       mark u as explored
       add e(u) to T
       for every edge (u, v) incident to u
           do if v is unexplored and (v ∉ Q or w(u, v) < w(e(v)))
                  then if v ∉ Q
                           then Q.insert(v, w(u, v))
                           else Q.decreaseKey(v, w(u, v))
                       e(v) = (u, v)
return T
```

This version of Prim's algorithm also takes $O(m \lg m)$ time:

- n Insert operations
- $m - n$ DecreaseKey operations
- n DeleteMin operations

$\Rightarrow$ $n + m$ priority queue operations.

Did we gain anything?

A Faster Version Of Prim's Algorithm

Prim(G)

```
T = (V, ∅)
mark every vertex of G as unexplored
set e(v) = nil for every vertex v ∈ G
mark an arbitrary vertex s as explored
Q = an empty priority queue
for every edge (s, v) incident to s
    do Q.insert(v, w(s, v))
       e(v) = (s, v)
while not Q.isEmpty()
    do u = Q.deleteMin()
       mark u as explored
       add e(u) to T
       for every edge (u, v) incident to u
           do if v is unexplored and (v ∉ Q or w(u, v) < w(e(v)))
                 then if v ∉ Q
                         then Q.insert(v, w(u, v))
                         else Q.decreaseKey(v, w(u, v))
                      e(v) = (u, v)
return T
```

This version of Prim's algorithm also takes $O(m \lg m)$ time:

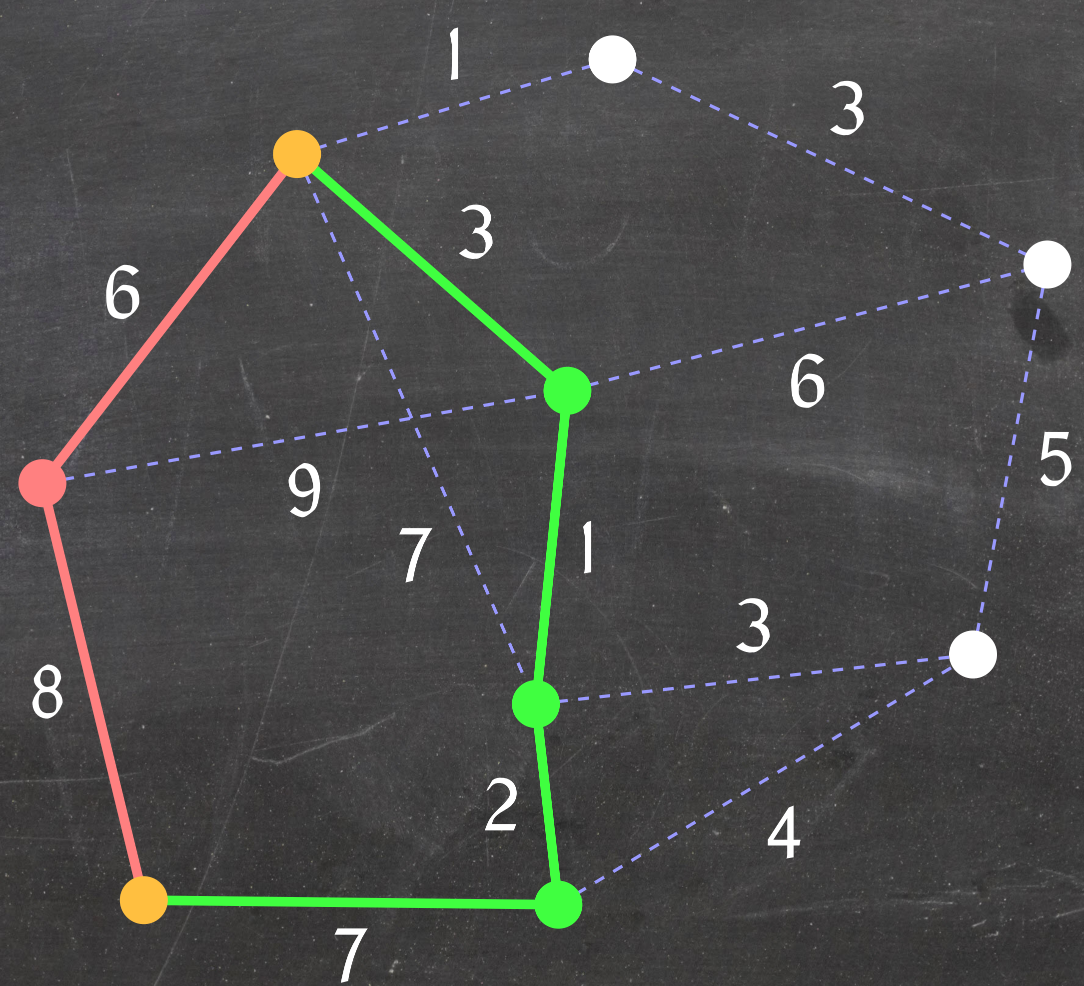
- n Insert operations
- $m - n$ DecreaseKey operations
- $\boxed{\bullet\ n \text{ DeleteMin operations}}$

$\Rightarrow$ $n + m$ priority queue operations.

Did we gain anything?

Shortest Path

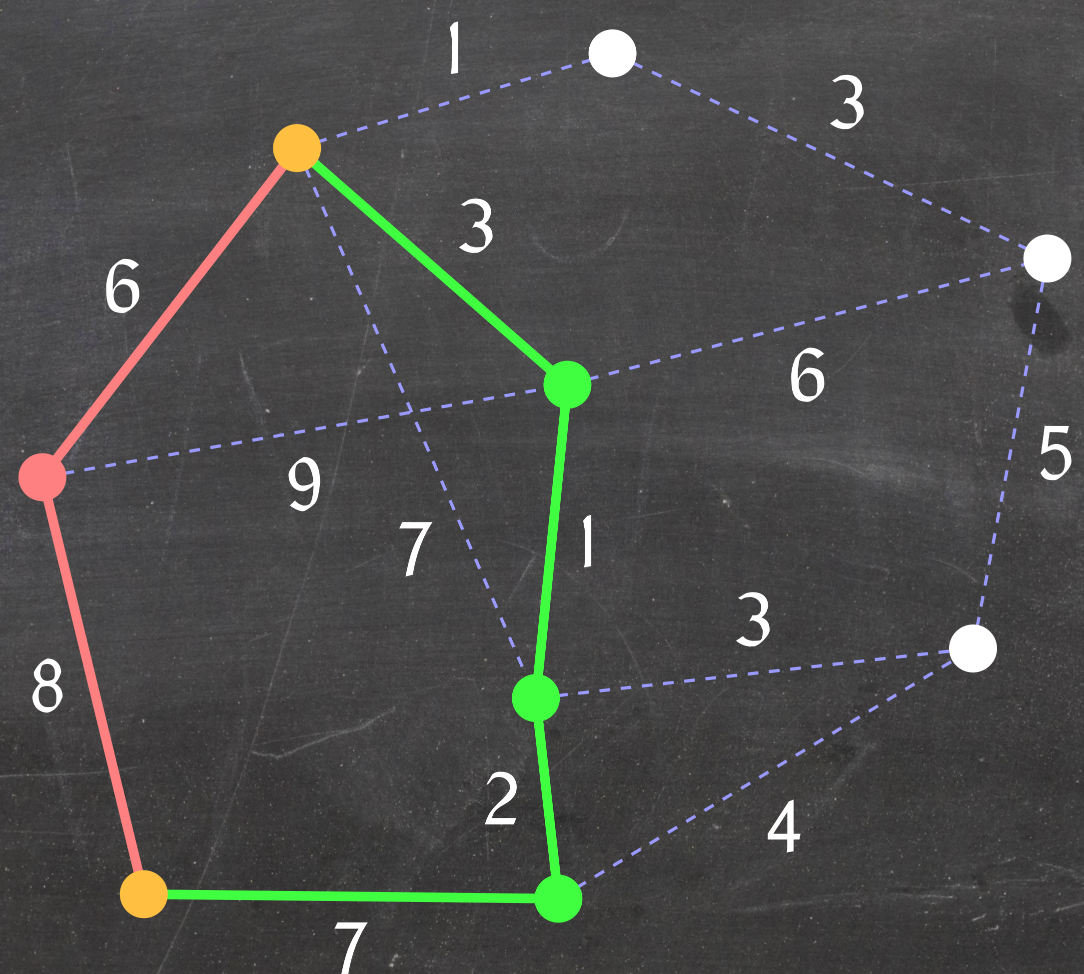
Given a graph $G = (V, E)$ and an assignment of weights (costs) to the edges of G, a shortest path from u to v is a path from u to v with minimum total edge weight among all paths from u to v.



Shortest Path

Given a graph $G = (V, E)$ and an assignment of weights (costs) to the edges of G, a shortest path from u to v is a path from u to v with minimum total edge weight among all paths from u to v.

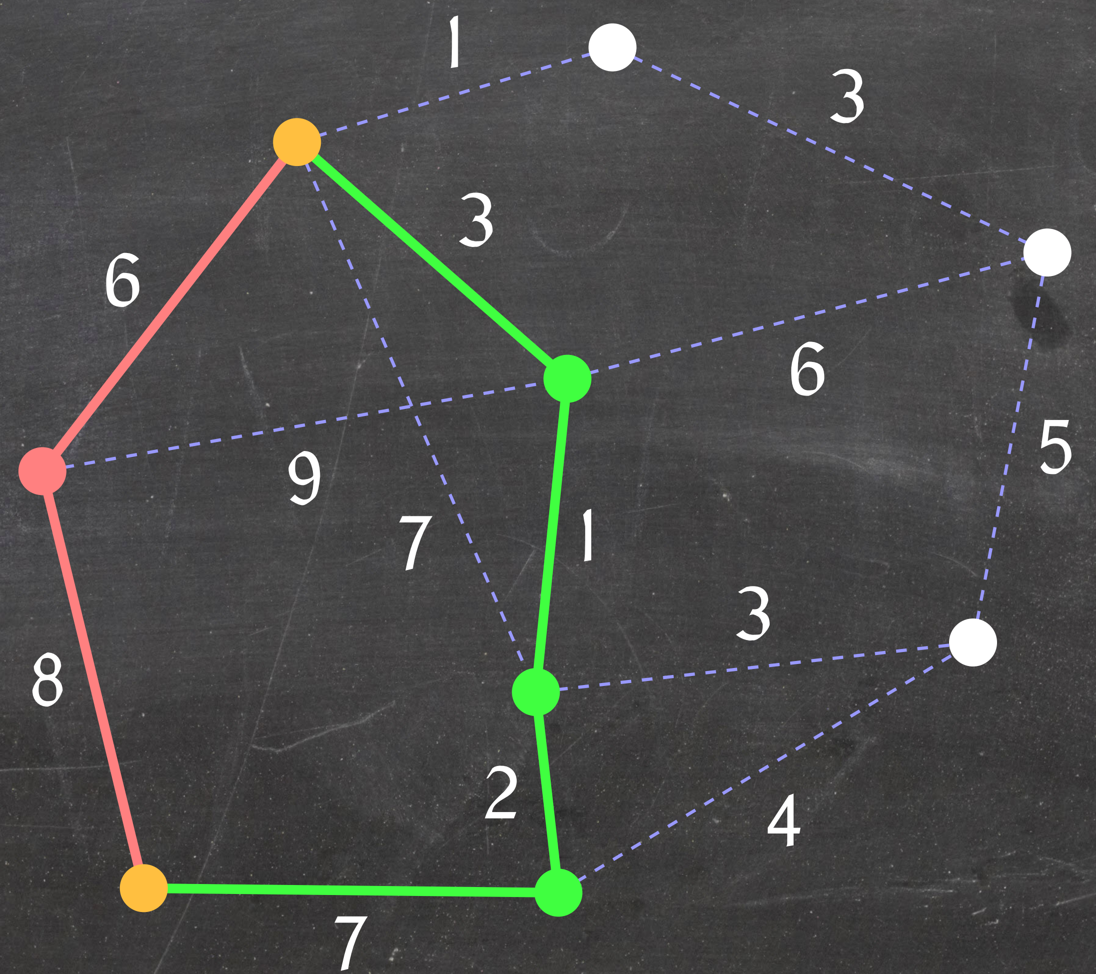
Let the distance dist(s, w) from s to v be the length of a shortest path from s to v.



Shortest Path

Given a graph $G = (V, E)$ and an assignment of weights (costs) to the edges of G, a shortest path from u to v is a path from u to v with minimum total edge weight among all paths from u to v.

Let the distance $dist(s, w)$ from s to v be the length of a shortest path from s to v.



This is well-defined only if there is no negative cycle (cycle with negative total edge weight) that has a vertex on a path from u to v.

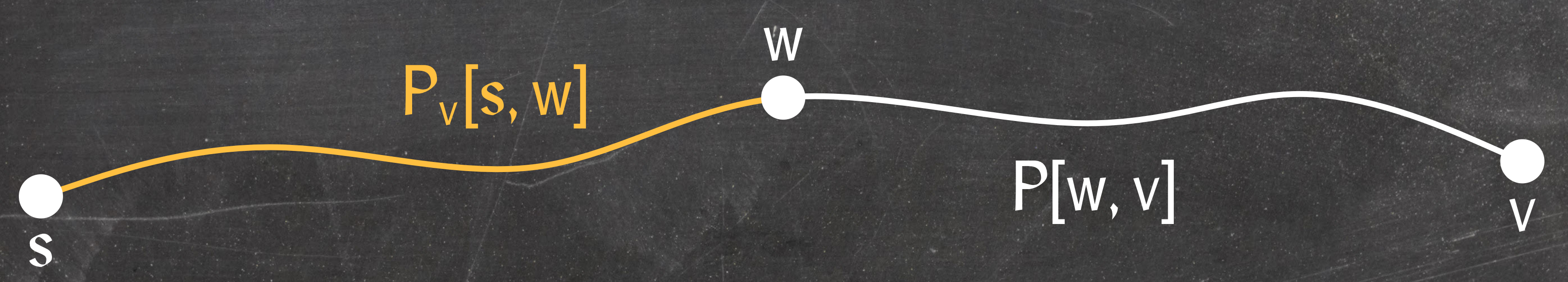
Optimal Substructure of Shortest Paths

For a path P and two vertices u and w in P, let P[u, w] be the subpath of P from u to w.

Optimal Substructure of Shortest Paths

For a path P and two vertices u and w in P, let $P[u, w]$ be the subpath of P from u to w.

Lemma: If P_v is a shortest path from s to v and w is a vertex in P_v, then $P_v[s, w]$ is a shortest path from s to w.

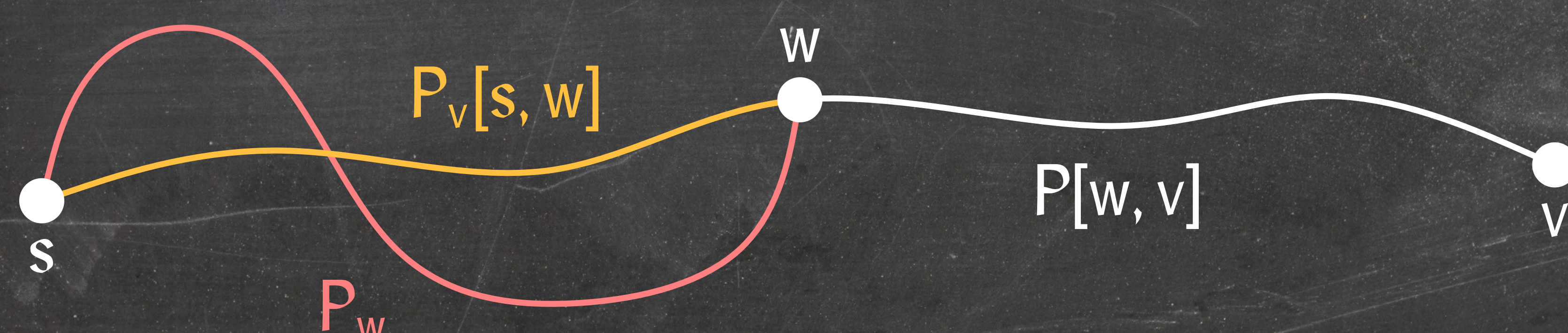


Optimal Substructure of Shortest Paths

For a path P and two vertices u and w in P, let $P[u, w]$ be the subpath of P from u to w.

Lemma: If P_v is a shortest path from s to v and w is a vertex in P_v, then $P_v[s, w]$ is a shortest path from s to w.

Assume there exists a path P_w from s to w with $w(P_w) < w(P_v[s, w])$.

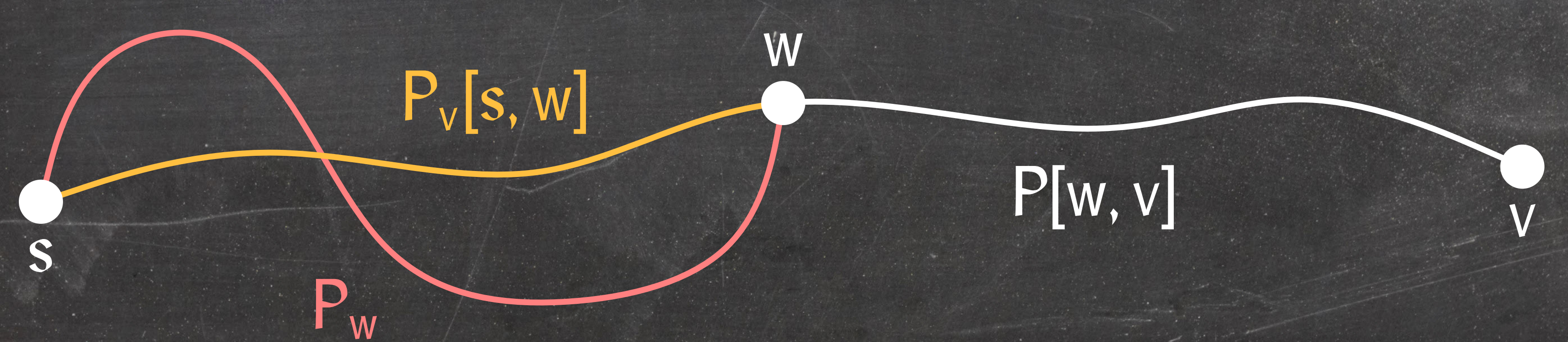


Optimal Substructure of Shortest Paths

For a path P and two vertices u and w in P, let $P[u, w]$ be the subpath of P from u to w.

Lemma: If P_v is a shortest path from s to v and w is a vertex in P_v, then $P_v[s, w]$ is a shortest path from s to w.

Assume there exists a path P_w from s to w with $w(P_w) < w(P_v[s, w])$.

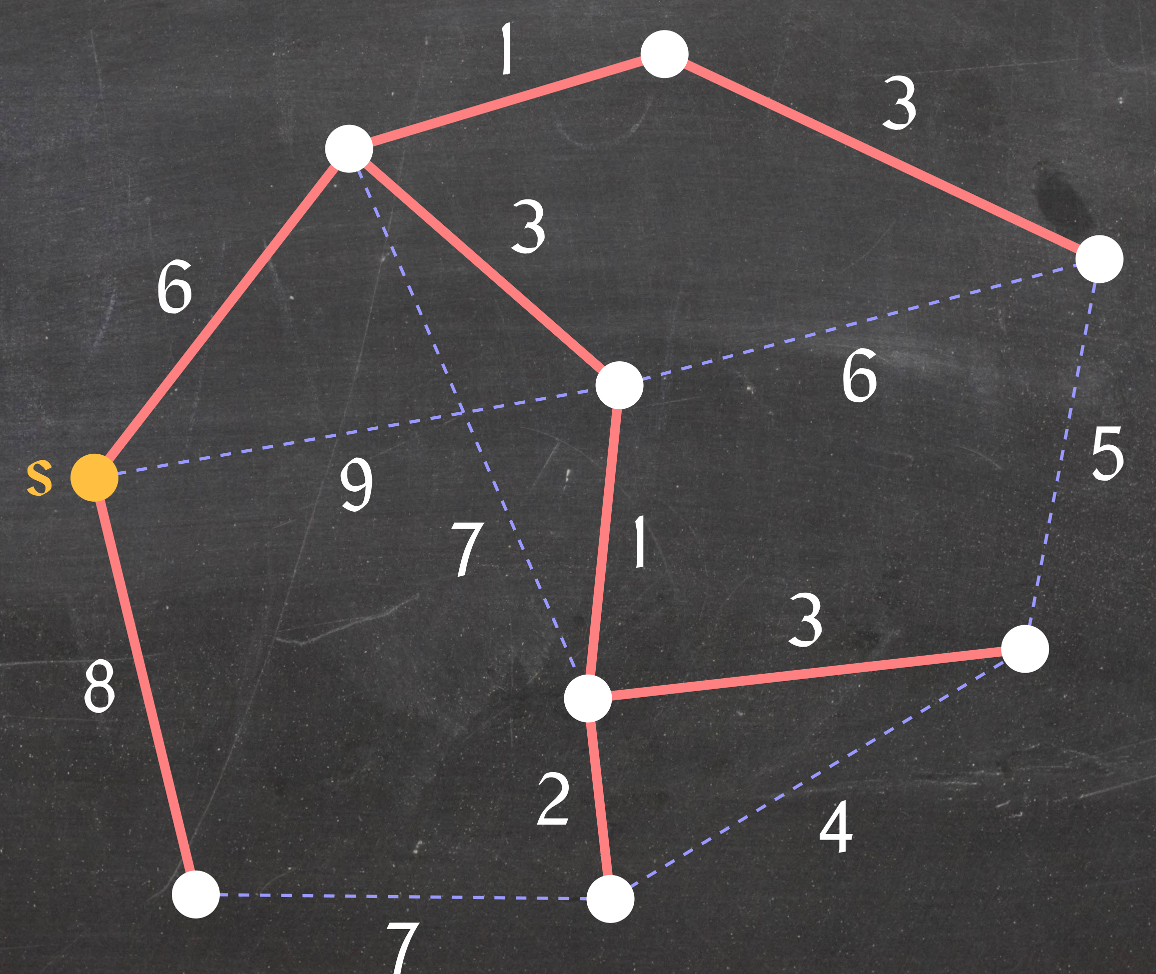


Then $w(P_w \circ P_v[w, v]) < w(P_v[s, w] \circ P_v[w, v]) = w(P_v)$, a contradiction because P_v is a shortest path from s to v.

Shortest Path Tree

For a vertex $s \in G$, let $R(s)$ be the set of vertices reachable from s: for every vertex $v \in R(s)$, there exists a path from s to v.

Lemma: For every node $s \in G$, there exists a collection of paths $\mathcal{S} = \{P_v \mid v \in R(s)\}$ such that P_v is a shortest path from s to v and $\bigcup_{v \in R(s)} P_v$ is a tree.



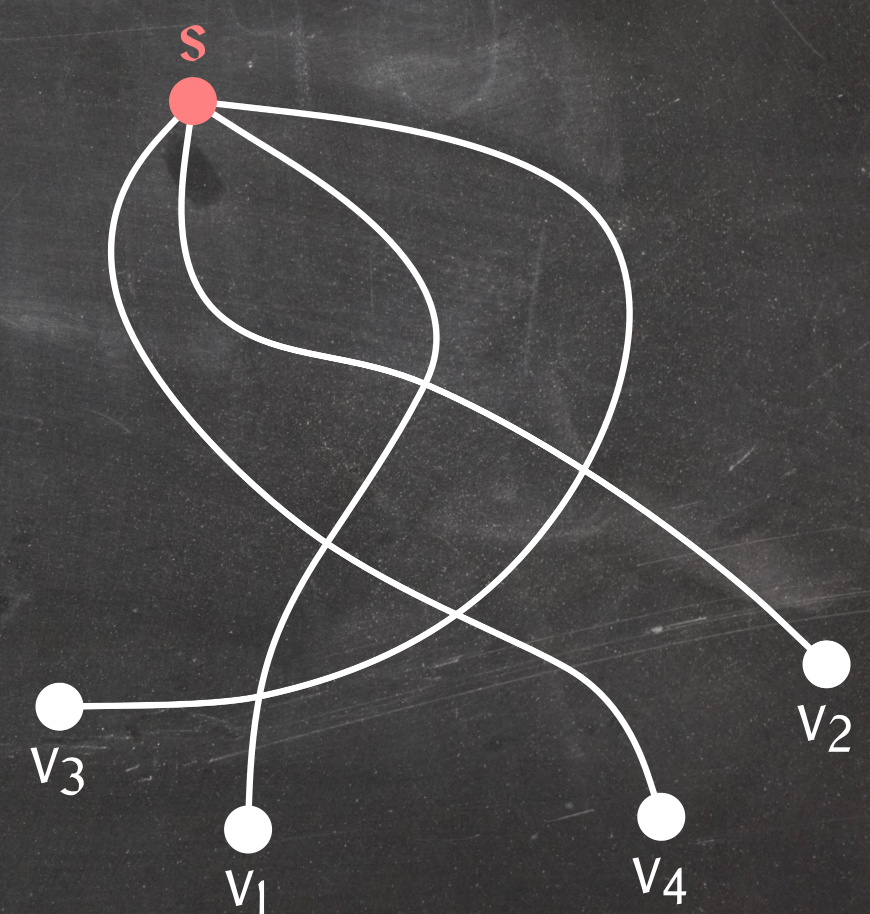
Shortest Path Tree

For a vertex $s \in G$, let $R(s)$ be the set of vertices reachable from s: for every vertex $v \in R(s)$, there exists a path from s to v.

Lemma: For every node $s \in G$, there exists a collection of paths $\mathcal{S} = \{P_v \mid v \in R(s)\}$ such that P_v is a shortest path from s to v and $\bigcup_{v \in R(s)} P_v$ is a tree.

Let $R(s) = \{v_1, v_2, \ldots, v_t\}$ and let $\{P'_{v_1}, P'_{v_2}, \ldots, P'_{v_t}\}$ be a collection of shortest paths from s to these vertices.

We define a sequence of trees $\langle T_1, T_2, \ldots, T_t \rangle$ and shortest paths $\langle P_{v_1}, P_{v_2}, \ldots, P_{v_t} \rangle$ as follows:



Shortest Path Tree

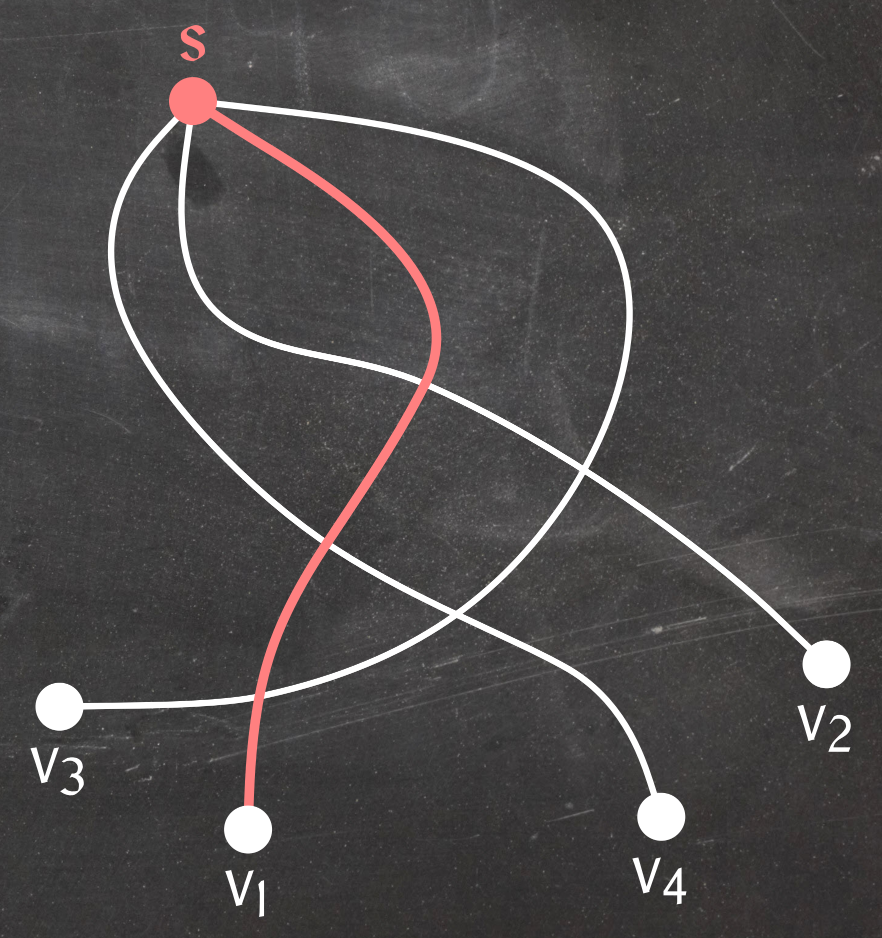
For a vertex $s \in G$, let $R(s)$ be the set of vertices reachable from s: for every vertex $v \in R(s)$, there exists a path from s to v.

Lemma: For every node $s \in G$, there exists a collection of paths $\mathcal{S} = \{P_v \mid v \in R(s)\}$ such that P_v is a shortest path from s to v and $\bigcup_{v \in R(s)} P_v$ is a tree.

Let $R(s) = \{v_1, v_2, \ldots, v_t\}$ and let $\{P'_{v_1}, P'_{v_2}, \ldots, P'_{v_t}\}$ be a collection of shortest paths from s to these vertices.

We define a sequence of trees $\langle T_1, T_2, \ldots, T_t \rangle$ and shortest paths $\langle P_{v_1}, P_{v_2}, \ldots, P_{v_t} \rangle$ as follows:

- $T_1 = P_{v_1} = P'_{v_1}$.



Shortest Path Tree

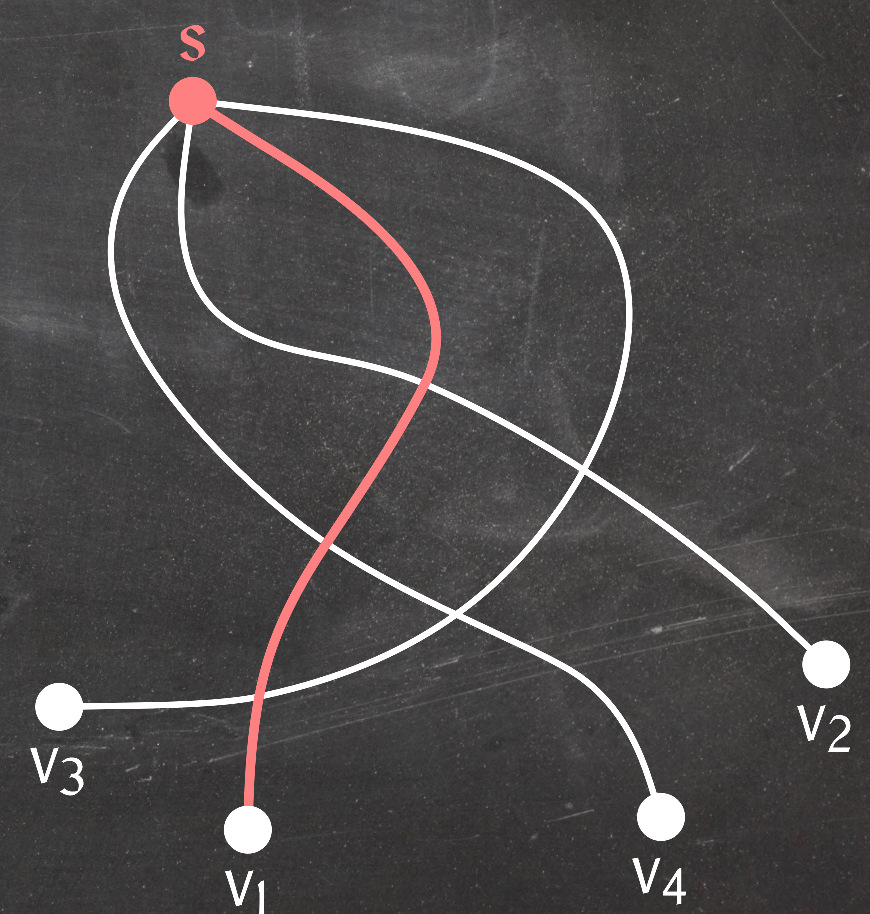
For a vertex $s \in G$, let $R(s)$ be the set of vertices reachable from s: for every vertex $v \in R(s)$, there exists a path from s to v.

Lemma: For every node $s \in G$, there exists a collection of paths $\mathcal{S} = \{P_v \mid v \in R(s)\}$ such that P_v is a shortest path from s to v and $\bigcup_{v \in R(s)} P_v$ is a tree.

Let $R(s) = \{v_1, v_2, \ldots, v_t\}$ and let $\{P'_{v_1}, P'_{v_2}, \ldots, P'_{v_t}\}$ be a collection of shortest paths from s to these vertices.

We define a sequence of trees $\langle T_1, T_2, \ldots, T_t \rangle$ and shortest paths $\langle P_{v_1}, P_{v_2}, \ldots, P_{v_t} \rangle$ as follows:

- $T_1 = P_{v_1} = P'_{v_1}$.
- For $i > 0$, let w be the last vertex in P'_{v_i} that belongs to T_{i-1} and let $T_{i-1}[s, w]$ be the path from s to w in T. Then
  - $P_{v_i} = T[s, w] \circ P'_{v_i}[w, v_i]$
  - $T_i = T_{i-1} \bigcup P'_{v_i}[w, v_i]$



Shortest Path Tree

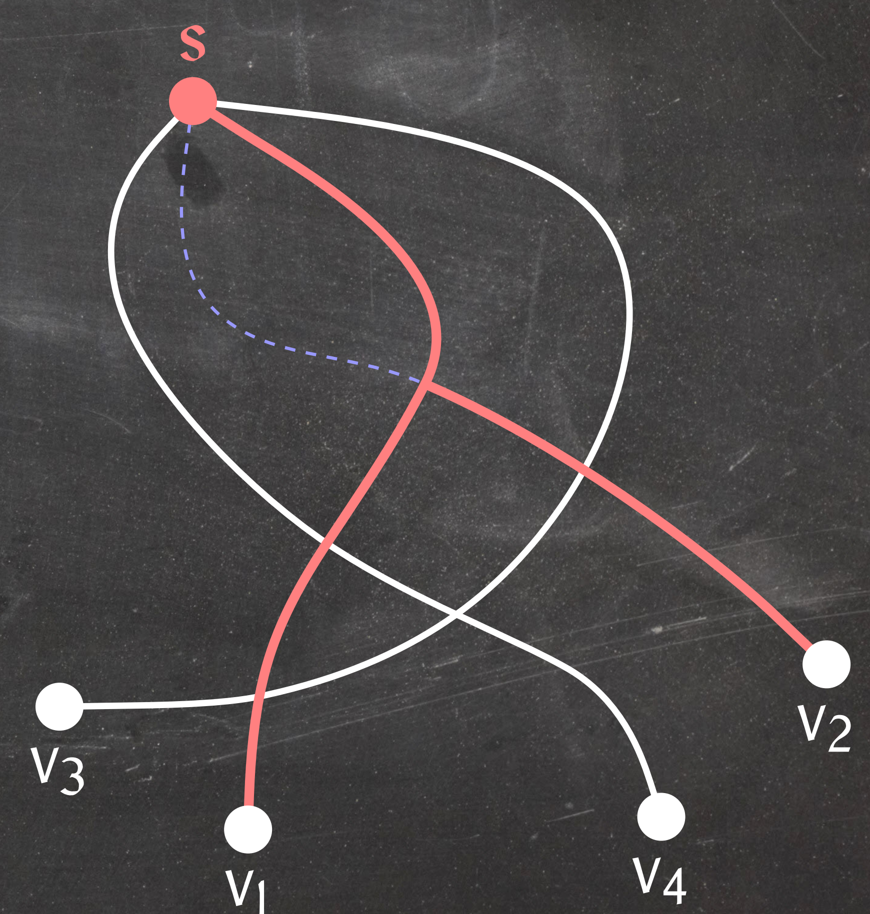
For a vertex $s \in G$, let $R(s)$ be the set of vertices reachable from s: for every vertex $v \in R(s)$, there exists a path from s to v.

Lemma: For every node $s \in G$, there exists a collection of paths $\mathcal{S} = \{P_v \mid v \in R(s)\}$ such that P_v is a shortest path from s to v and $\bigcup_{v \in R(s)} P_v$ is a tree.

Let $R(s) = \{v_1, v_2, \ldots, v_t\}$ and let $\{P'_{v_1}, P'_{v_2}, \ldots, P'_{v_t}\}$ be a collection of shortest paths from s to these vertices.

We define a sequence of trees $\langle T_1, T_2, \ldots, T_t \rangle$ and shortest paths $\langle P_{v_1}, P_{v_2}, \ldots, P_{v_t} \rangle$ as follows:

- $T_1 = P_{v_1} = P'_{v_1}$.
- For $i > 0$, let w be the last vertex in P'_{v_i} that belongs to T_{i-1} and let $T_{i-1}[s, w]$ be the path from s to w in T. Then
  - $P_{v_i} = T[s, w] \circ P'_{v_i}[w, v_i]$
  - $T_i = T_{i-1} \bigcup P'_{v_i}[w, v_i]$

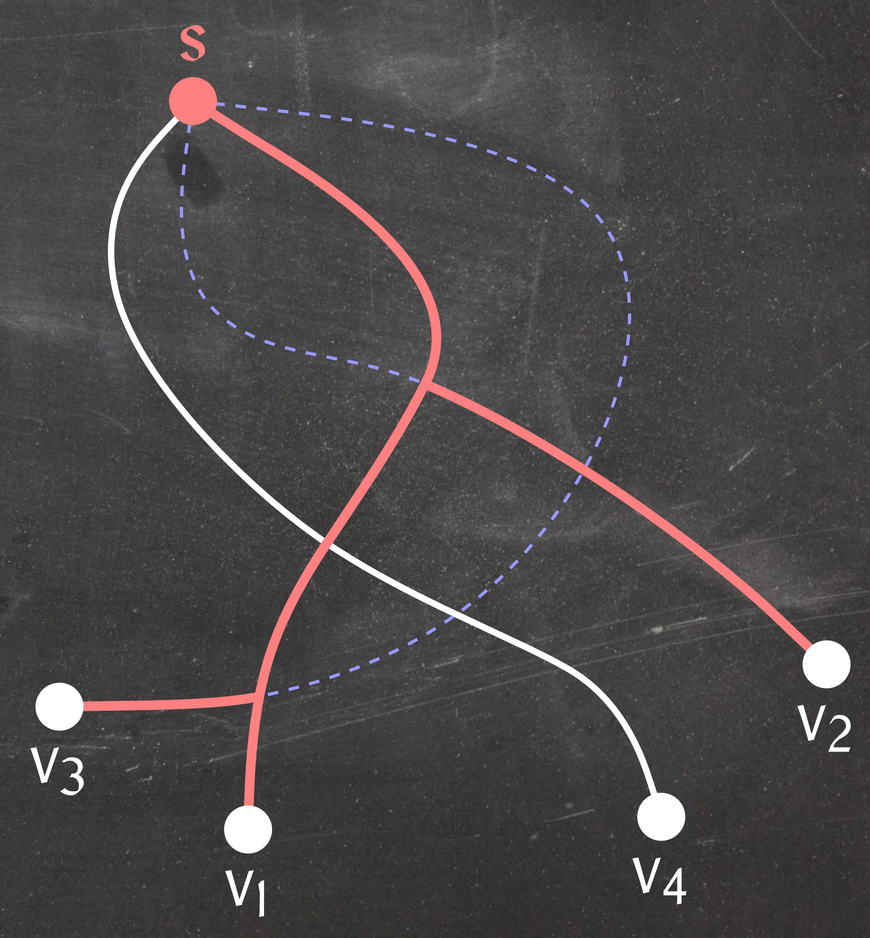


Shortest Path Tree

For a vertex $s \in G$, let $R(s)$ be the set of vertices reachable from s: for every vertex $v \in R(s)$, there exists a path from s to v.

Lemma: For every node $s \in G$, there exists a collection of paths $S = \{P_v \mid v \in R(s)\}$ such that P_v is a shortest path from s to v and $\bigcup_{v \in R(s)} P_v$ is a tree.

Let $R(s) = \{v_1, v_2, \ldots, v_t\}$ and let $\{P'_{v_1}, P'_{v_2}, \ldots, P'_{v_t}\}$ be a collection of shortest paths from s to these vertices.



We define a sequence of trees $\langle T_1, T_2, \ldots, T_t \rangle$ and shortest paths $\langle P_{v_1}, P_{v_2}, \ldots, P_{v_t} \rangle$ as follows:

- $T_1 = P_{v_1} = P'_{v_1}$.
- For $i > 0$, let w be the last vertex in P'_{v_i} that belongs to T_{i-1} and let $T_{i-1}[s, w]$ be the path from s to w in T. Then
  - $P_{v_i} = T[s, w] \circ P'_{v_i}[w, v_i]$
  - $T_i = T_{i-1} \bigcup P'_{v_i}[w, v_i]$

Shortest Path Tree

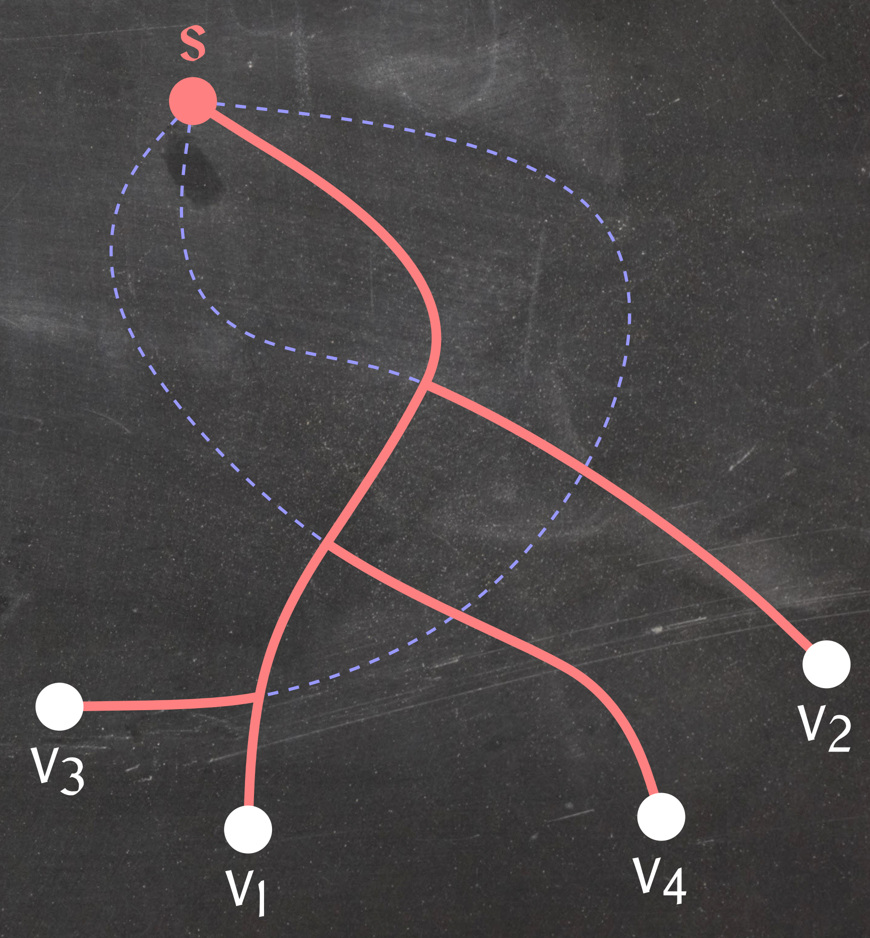
For a vertex $s \in G$, let $R(s)$ be the set of vertices reachable from s: for every vertex $v \in R(s)$, there exists a path from s to v.

Lemma: For every node $s \in G$, there exists a collection of paths $\mathcal{S} = \{P_v \mid v \in R(s)\}$ such that P_v is a shortest path from s to v and $\bigcup_{v \in R(s)} P_v$ is a tree.

Let $R(s) = \{v_1, v_2, \ldots, v_t\}$ and let $\{P'_{v_1}, P'_{v_2}, \ldots, P'_{v_t}\}$ be a collection of shortest paths from s to these vertices.

We define a sequence of trees $\langle T_1, T_2, \ldots, T_t \rangle$ and shortest paths $\langle P_{v_1}, P_{v_2}, \ldots, P_{v_t} \rangle$ as follows:

- $T_1 = P_{v_1} = P'_{v_1}$.
- For $i > 0$, let w be the last vertex in P'_{v_i} that belongs to T_{i-1} and let $T_{i-1}[s, w]$ be the path from s to w in T. Then
  - $P_{v_i} = T[s, w] \circ P'_{v_i}[w, v_i]$
  - $T_i = T_{i-1} \bigcup P'_{v_i}[w, v_i]$

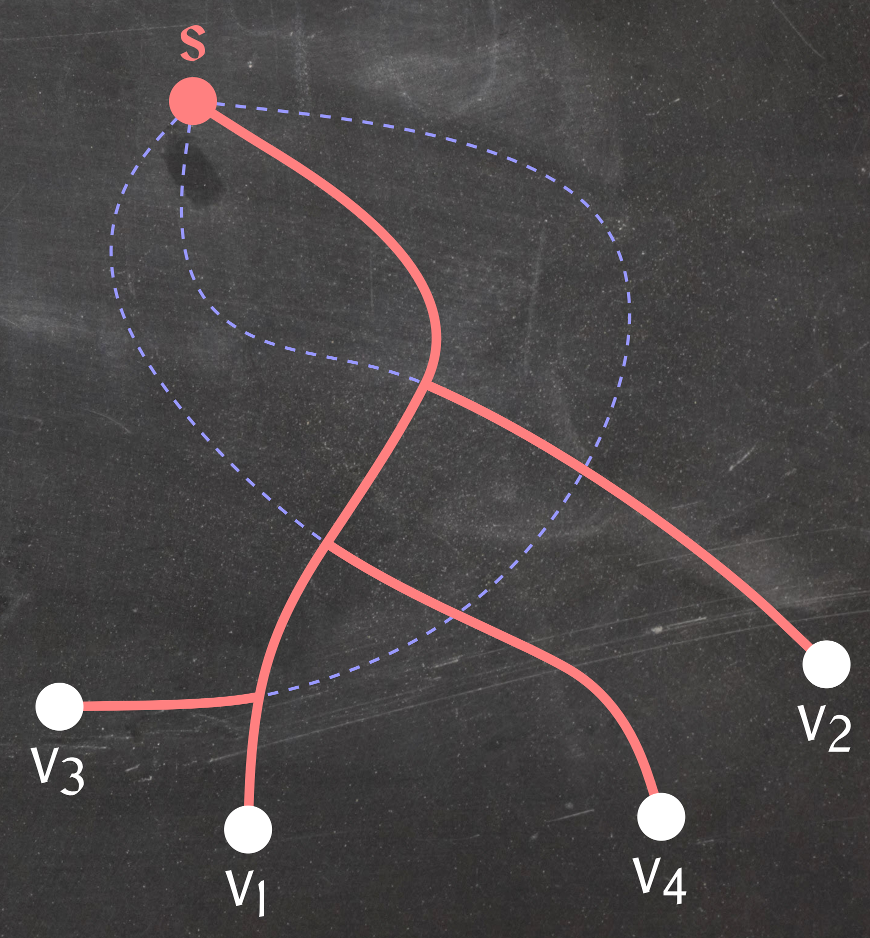


Shortest Path Tree

For a vertex $s \in G$, let $R(s)$ be the set of vertices reachable from s: for every vertex $v \in R(s)$, there exists a path from s to v.

Lemma: For every node $s \in G$, there exists a collection of paths $S = \{P_v \mid v \in R(s)\}$ such that P_v is a shortest path from s to v and $\bigcup_{v \in R(s)} P_v$ is a tree.

$T_t = \bigcup_{v \in R(s)} P_v$



Shortest Path Tree

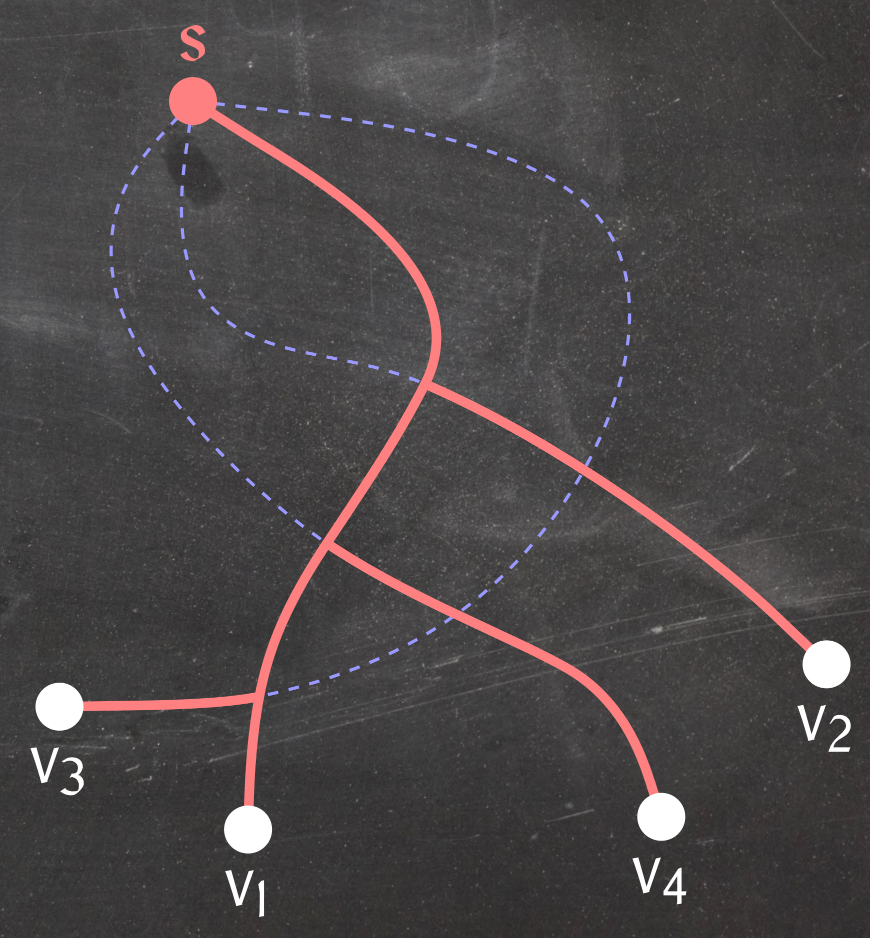
For a vertex $s \in G$, let $R(s)$ be the set of vertices reachable from s: for every vertex $v \in R(s)$, there exists a path from s to v.

Lemma: For every node $s \in G$, there exists a collection of paths $\mathcal{S} = \{P_v \mid v \in R(s)\}$ such that P_v is a shortest path from s to v and $\bigcup_{v \in R(s)} P_v$ is a tree.

$T_t = \bigcup_{v \in R(s)} P_v$

T_t is a tree:

- T_1 is a tree.
- T_i is obtained by adding a path to T_{i-1} that shares only one vertex with T_{i-1}.
- To create a cycle, the added path would have to share two vertices with T_{i-1}.

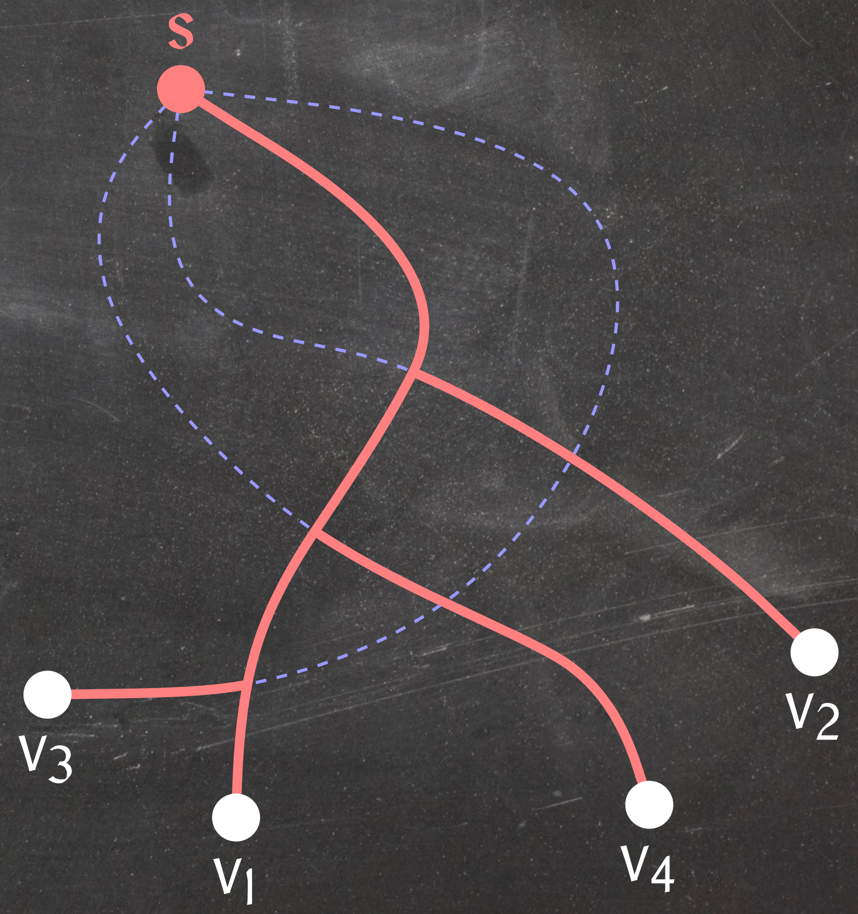


Shortest Path Tree

For a vertex $s \in G$, let $R(s)$ be the set of vertices reachable from s: for every vertex $v \in R(s)$, there exists a path from s to v.

Lemma: For every node $s \in G$, there exists a collection of paths $\mathcal{S} = \{P_v \mid v \in R(s)\}$ such that P_v is a shortest path from s to v and $\bigcup_{v \in R(s)} P_v$ is a tree.

P_v is a shortest path from s to v, for all $v \in R(s)$.



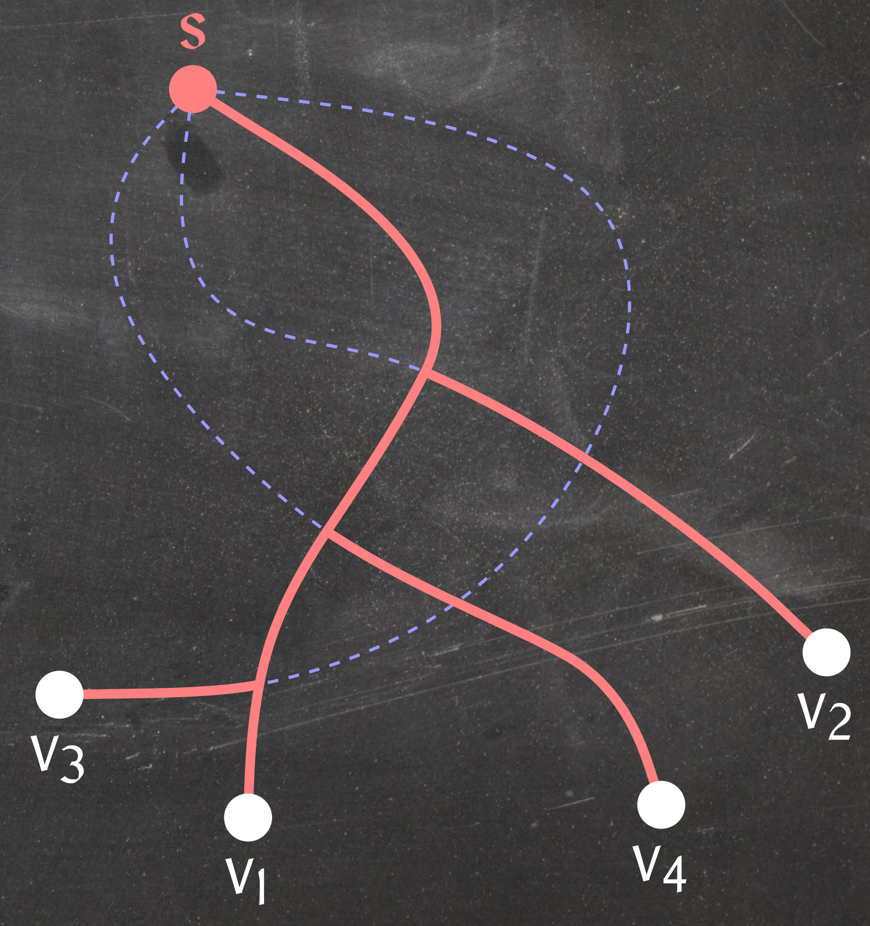
Shortest Path Tree

For a vertex $s \in G$, let $R(s)$ be the set of vertices reachable from s: for every vertex $v \in R(s)$, there exists a path from s to v.

Lemma: For every node $s \in G$, there exists a collection of paths $S = \{P_v \mid v \in R(s)\}$ such that P_v is a shortest path from s to v and $\bigcup_{v \in R(s)} P_v$ is a tree.

P_v is a shortest path from s to v, for all $v \in R(s)$.

Prove by induction on i that $T_i[s, v]$ is a shortest path from s to v, for all $v \in T_i$.



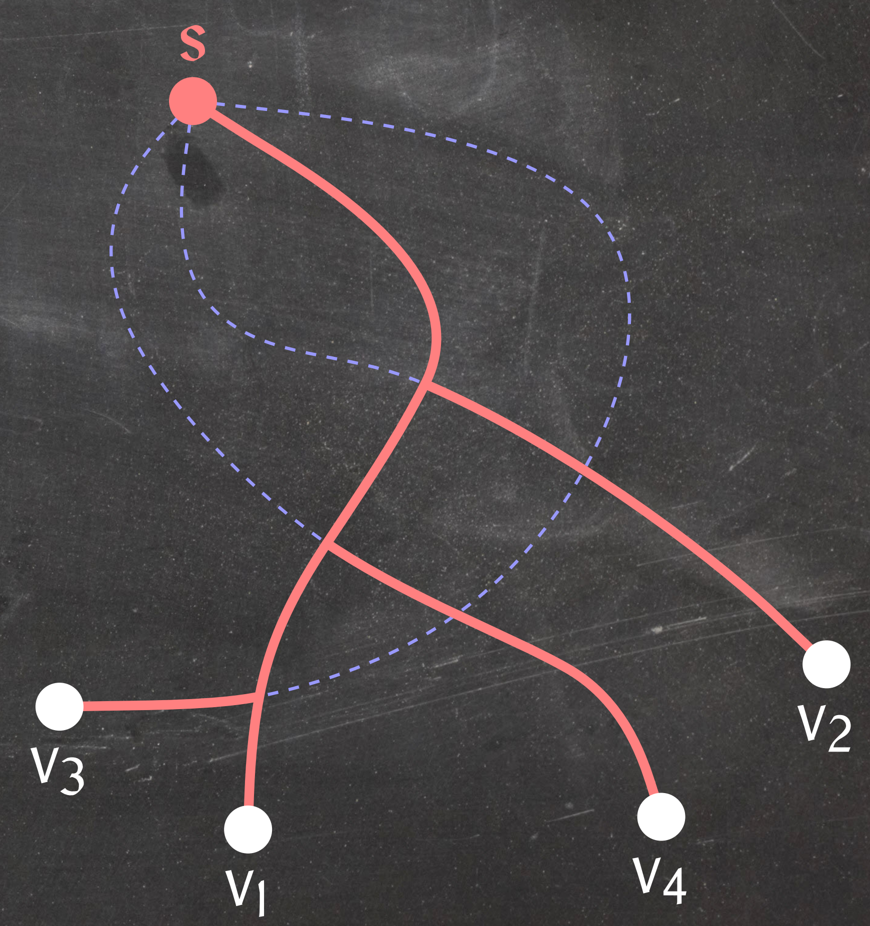
Shortest Path Tree

For a vertex $s \in G$, let $R(s)$ be the set of vertices reachable from s: for every vertex $v \in R(s)$, there exists a path from s to v.

Lemma: For every node $s \in G$, there exists a collection of paths $S = \{P_v \mid v \in R(s)\}$ such that P_v is a shortest path from s to v and $\bigcup_{v \in R(s)} P_v$ is a tree.

P_v is a shortest path from s to v, for all $v \in R(s)$.

For $i = 1$, $T_1 = P_{v_1} = P'_{v_1}$ is a shortest path from s to v_1. By optimal substructure, $T_1[s, v] = P'_{v_1}[s, v]$ is a shortest path from s to v for all $v \in T_1$.



Shortest Path Tree

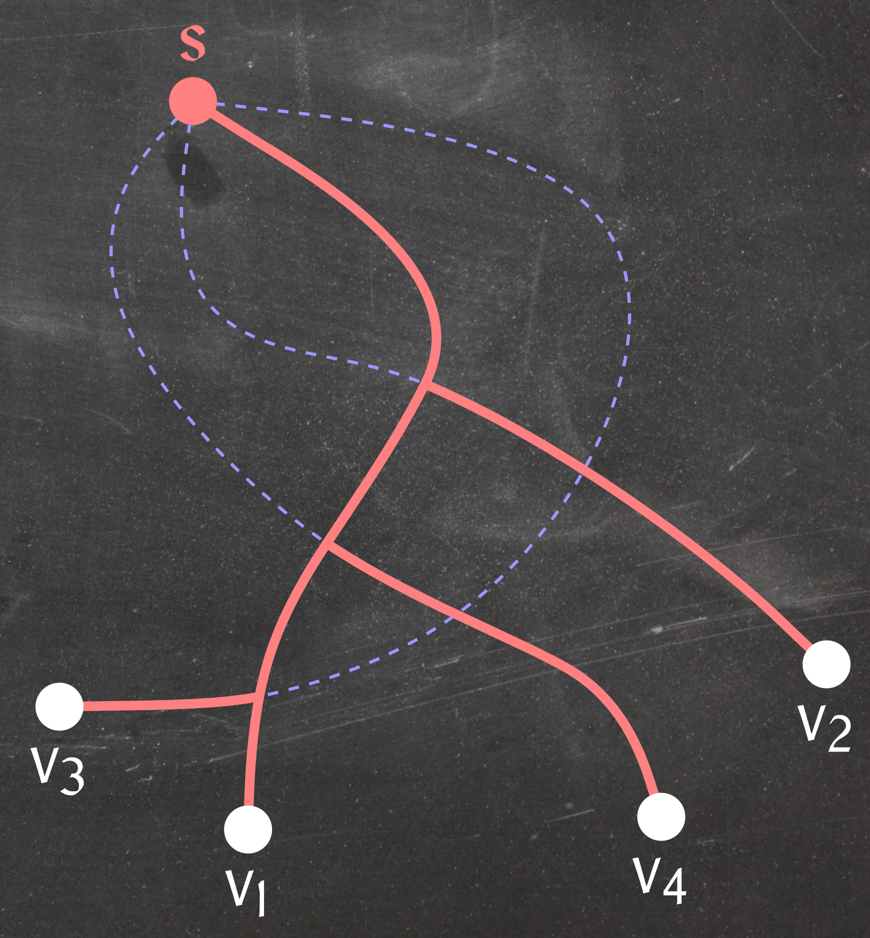
For a vertex $s \in G$, let $R(s)$ be the set of vertices reachable from s: for every vertex $v \in R(s)$, there exists a path from s to v.

Lemma: For every node $s \in G$, there exists a collection of paths $\mathcal{S} = \{P_v \mid v \in R(s)\}$ such that P_v is a shortest path from s to v and $\bigcup_{v \in R(s)} P_v$ is a tree.

P_v is a shortest path from s to v, for all $v \in R(s)$.

For $i = 1$, $T_1 = P_{v_1} = P'_{v_1}$ is a shortest path from s to v_1. By optimal substructure, $T_1[s, v] = P'_{v_1}[s, v]$ is a shortest path from s to v for all $v \in T_1$.

For $i > 1$, $T_{i-1}[s, v]$ is a shortest path from s to v for all $v \in T_{i-1}$, by the inductive hypothesis.



Shortest Path Tree

For a vertex $s \in G$, let $R(s)$ be the set of vertices reachable from s: for every vertex $v \in R(s)$, there exists a path from s to v.

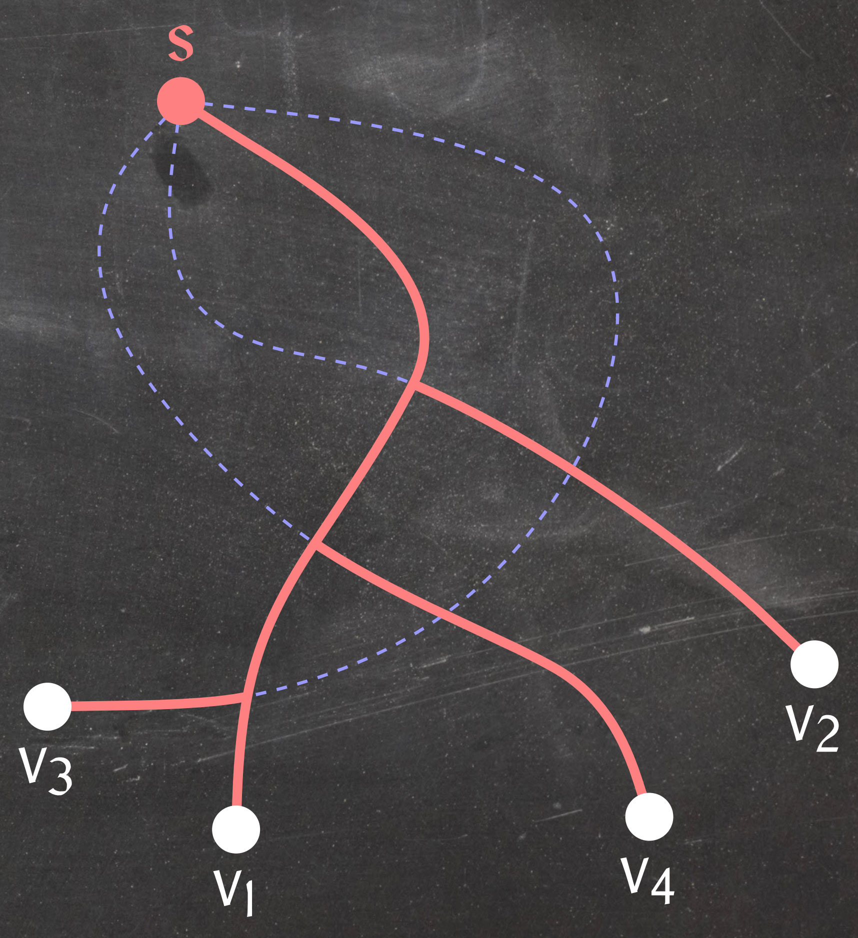
Lemma: For every node $s \in G$, there exists a collection of paths $\mathcal{S} = \{P_v \mid v \in R(s)\}$ such that P_v is a shortest path from s to v and $\bigcup_{v \in R(s)} P_v$ is a tree.

P_v is a shortest path from s to v, for all $v \in R(s)$.

For $i = 1$, $T_1 = P_{v_1} = P'_{v_1}$ is a shortest path from s to v_1. By optimal substructure, $T_1[s, v] = P'_{v_1}[s, v]$ is a shortest path from s to v for all $v \in T_1$.

For $i > 1$, $T_{i-1}[s, v]$ is a shortest path from s to v for all $v \in T_{i-1}$, by the inductive hypothesis.

Thus, $w(T_{i-1}[s, w]) \le w(P'_{v_i}[s, w])$ and therefore $w(P_{v_i}) = w(T_{i-1}[s, w]) + w(P'_{v_i}[w, v_i]) \le w(P'_{v_i})$.



Shortest Path Tree

For a vertex $s \in G$, let $R(s)$ be the set of vertices reachable from s: for every vertex $v \in R(s)$, there exists a path from s to v.

Lemma: For every node $s \in G$, there exists a collection of paths $\mathcal{S} = \{P_v \mid v \in R(s)\}$ such that P_v is a shortest path from s to v and $\bigcup_{v \in R(s)} P_v$ is a tree.

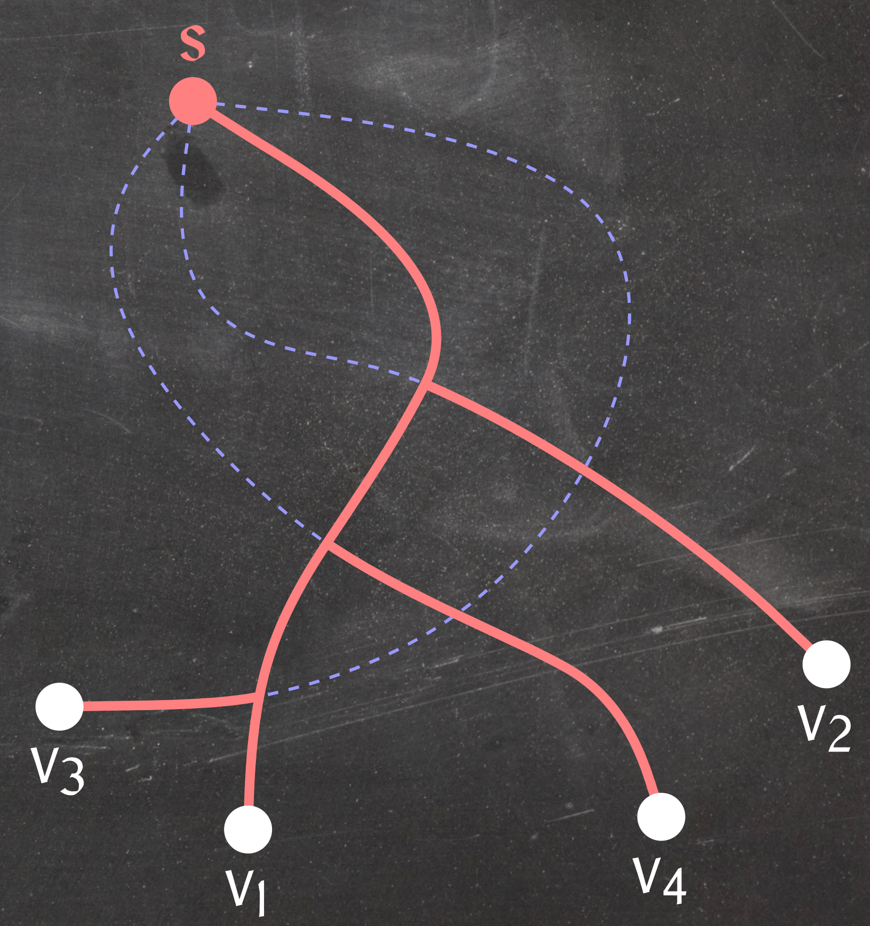
P_v is a shortest path from s to v, for all $v \in R(s)$.

For $i = 1$, $T_1 = P_{v_1} = P'_{v_1}$ is a shortest path from s to v_1. By optimal substructure, $T_1[s, v] = P'_{v_1}[s, v]$ is a shortest path from s to v for all $v \in T_1$.

For $i > 1$, $T_{i-1}[s, v]$ is a shortest path from s to v for all $v \in T_{i-1}$, by the inductive hypothesis.

Thus, $w(T_{i-1}[s, w]) \leq w(P'_{v_i}[s, w])$ and therefore $w(P_{v_i}) = w(T_{i-1}[s, w]) + w(P'_{v_i}[w, v_i]) \leq w(P'_{v_i})$.

Since P'_{v_i} is a shortest path from s to v_i, so is P_{v_i}.



Shortest Path Tree

For a vertex $s \in G$, let $R(s)$ be the set of vertices reachable from s: for every vertex $v \in R(s)$, there exists a path from s to v.

Lemma: For every node $s \in G$, there exists a collection of paths $\mathcal{S} = \{P_v \mid v \in R(s)\}$ such that P_v is a shortest path from s to v and $\bigcup_{v \in R(s)} P_v$ is a tree.

P_v is a shortest path from s to v, for all $v \in R(s)$.

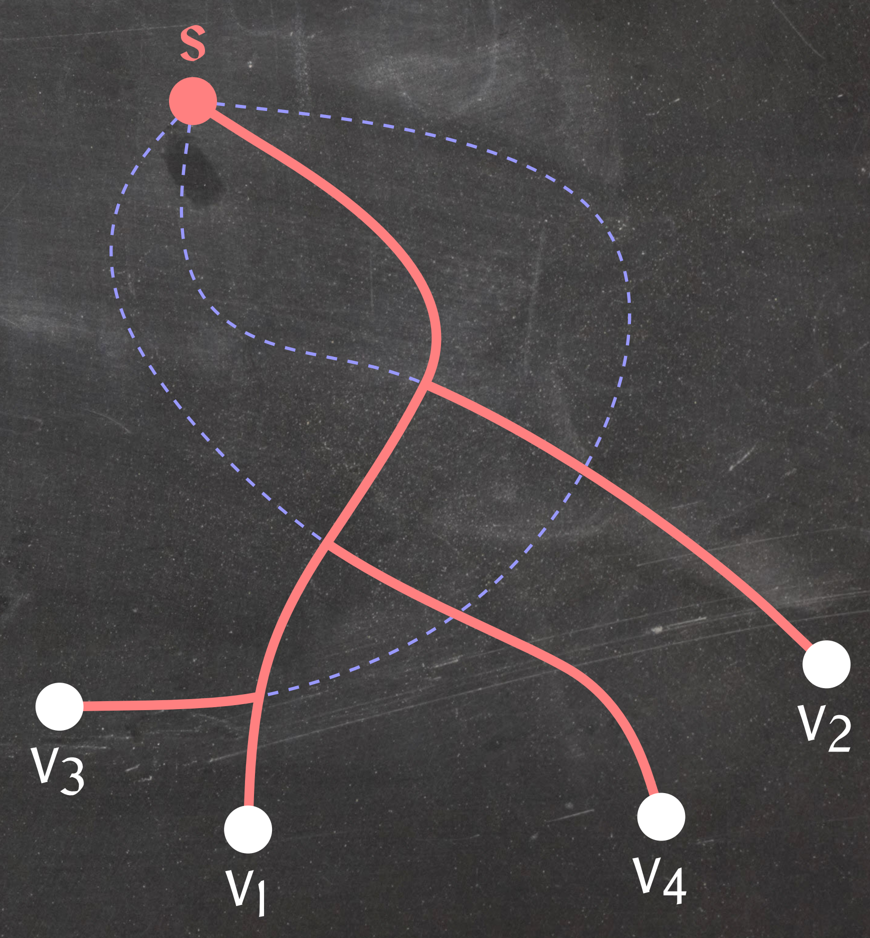
For $i = 1$, $T_1 = P_{v_1} = P'_{v_1}$ is a shortest path from s to v_1. By optimal substructure, $T_1[s, v] = P'_{v_1}[s, v]$ is a shortest path from s to v for all $v \in T_1$.

For $i > 1$, $T_{i-1}[s, v]$ is a shortest path from s to v for all $v \in T_{i-1}$, by the inductive hypothesis.

Thus, $w(T_{i-1}[s, w]) \le w(P'_{v_i}[s, w])$ and therefore $w(P_{v_i}) = w(T_{i-1}[s, w]) + w(P'_{v_i}[w, v_i]) \le w(P'_{v_i})$.

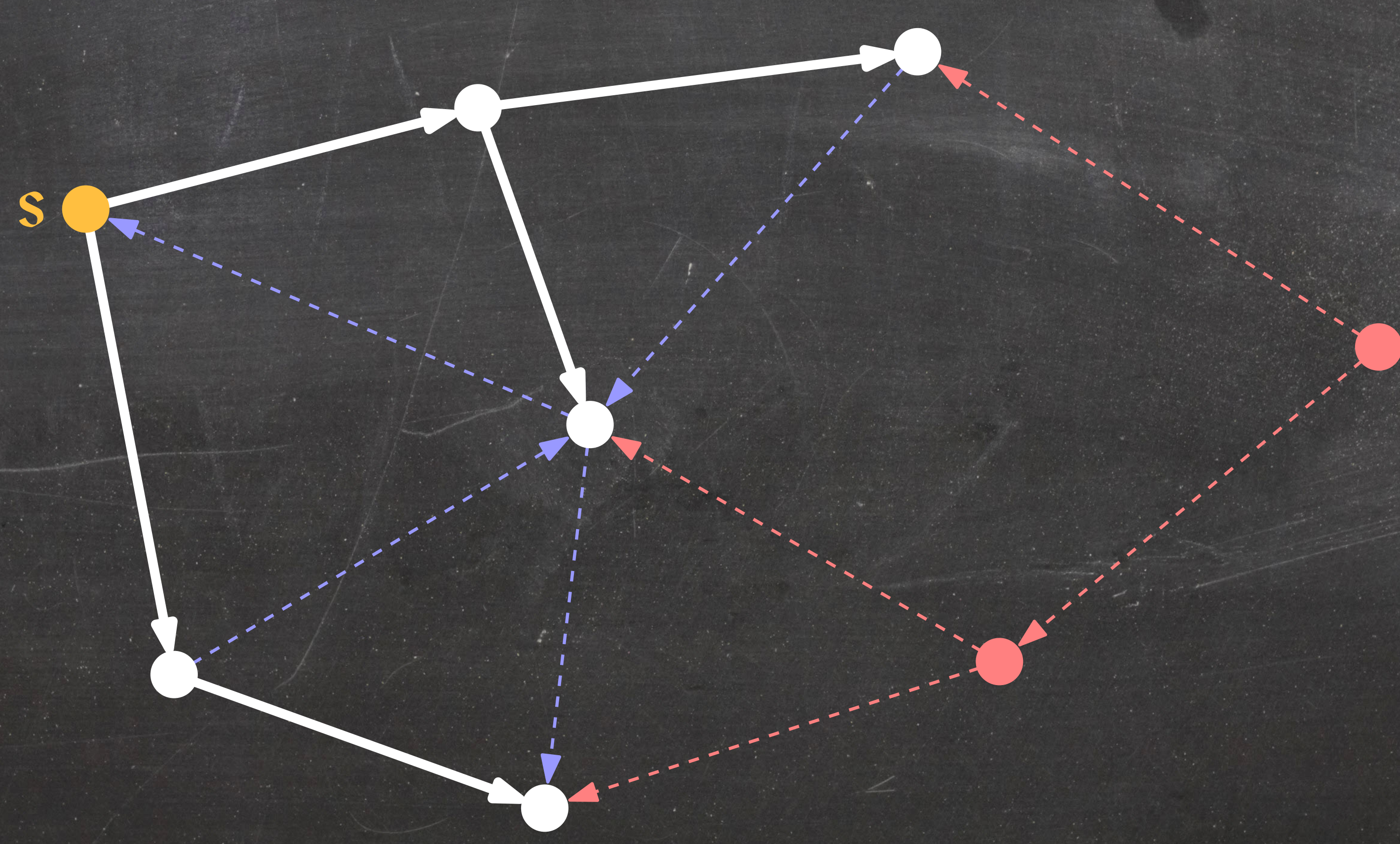
Since P'_{v_i} is a shortest path from s to v_i, so is P_{v_i}.

By optimal substructure $P_{v_i}[s, v]$ is a shortest path from s to v, for all $v \in P_{v_i}$.



A Characterization of Shortest Path Trees

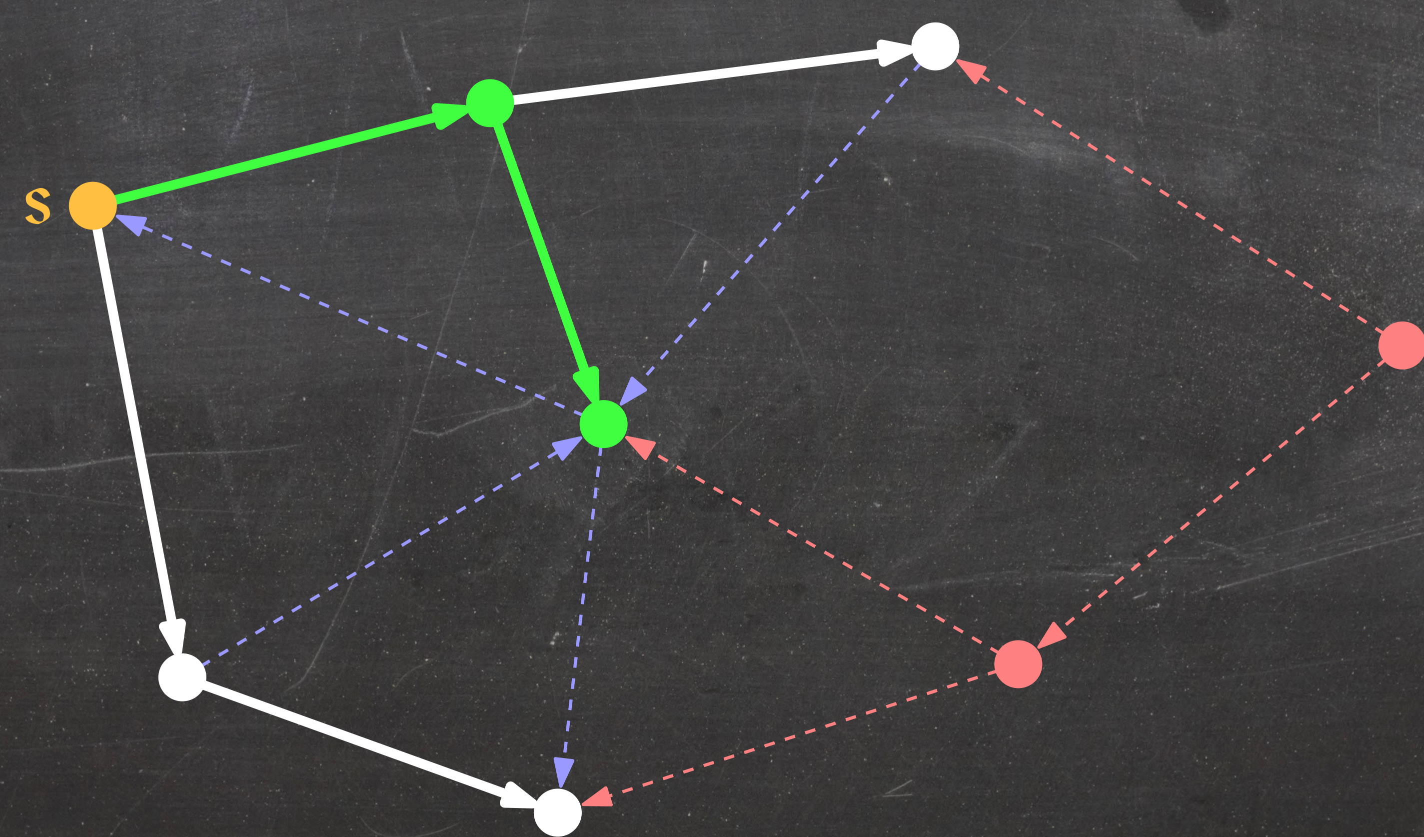
An out-tree of s is a spanning tree T of $G[R(s)] = (R(s), E[R(s)])$, where $E[R(s)] = \{(v, w) \in E \mid v, w \in R(s)\}$, such that there exists a path from s to v in T, for all $v \in R(s)$.



A Characterization of Shortest Path Trees

An out-tree of s is a spanning tree T of $G[R(s)] = (R(s), E[R(s)])$, where $E[R(s)] = \{(v, w) \in E \mid v, w \in R(s)\}$, such that there exists a path from s to v in T, for all $v \in R(s)$.

For an out-tree T of s and every $v \in T$, let $d_T(v) = w(T[s, v])$.

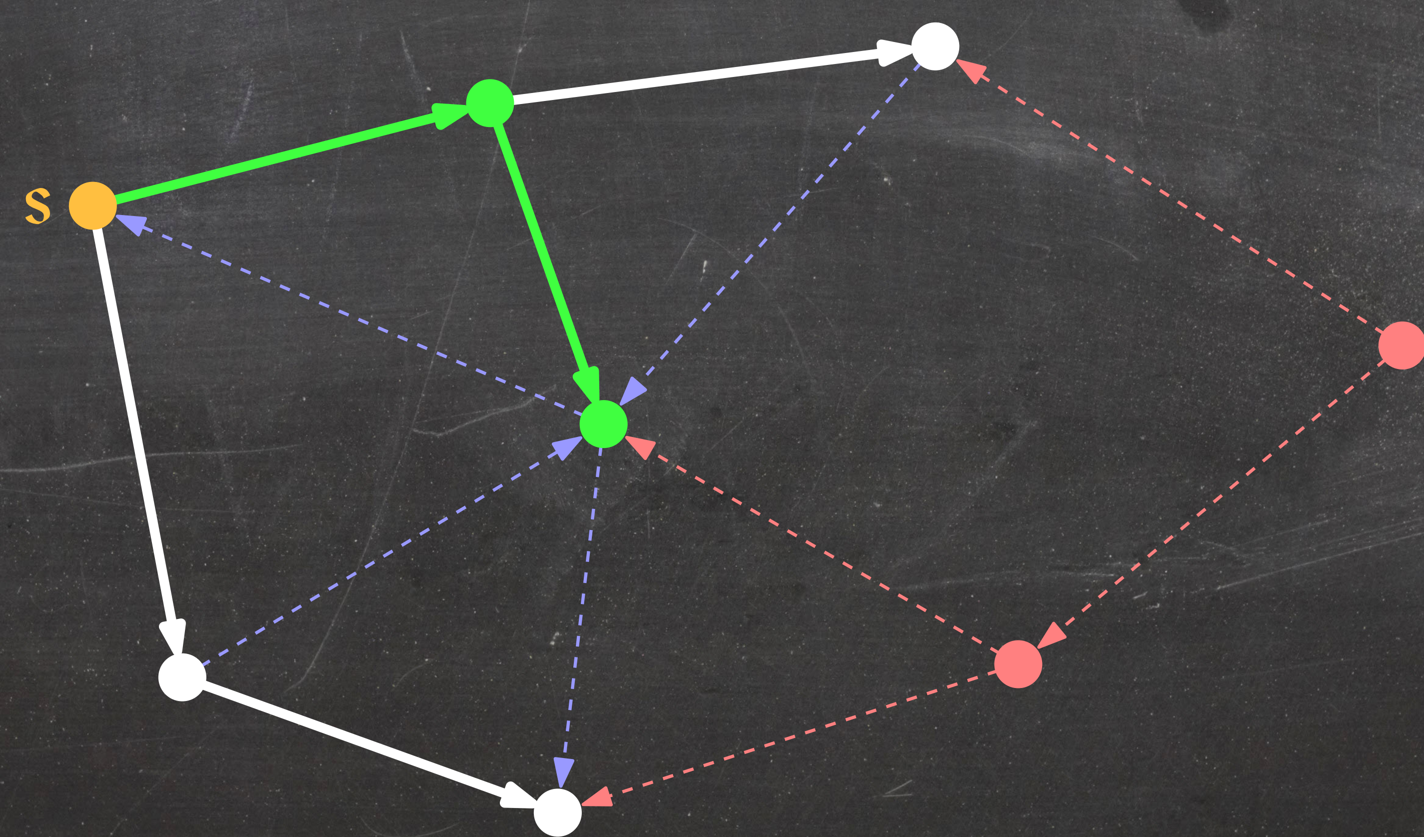


A Characterization of Shortest Path Trees

An out-tree of s is a spanning tree T of $G[R(s)] = (R(s), E[R(s)])$, where $E[R(s)] = \{(v, w) \in E \mid v, w \in R(s)\}$, such that there exists a path from s to v in T, for all $v \in R(s)$.

For an out-tree T of s and every $v \in T$, let $d_T(v) = w(T[s, v])$.

Let $D(T) = \sum_{v \in R(s)} d_T(v)$.



A Characterization of Shortest Path Trees

An out-tree of s is a spanning tree T of $G[R(s)] = (R(s), E[R(s)])$, where $E[R(s)] = \{(v, w) \in E \mid v, w \in R(s)\}$, such that there exists a path from s to v in T, for all $v \in R(s)$.

For an out-tree T of s and every $v \in T$, let $d_T(v) = w(T[s, v])$.

Let $D(T) = \sum_{v \in R(s)} d_T(v)$.

Lemma: An out-tree T of s is a shortest path tree if and only if D(T) is minimal among all out-trees of s.

A Characterization of Shortest Path Trees

An out-tree of s is a spanning tree T of $G[R(s)] = (R(s), E[R(s)])$, where $E[R(s)] = \{(v, w) \in E \mid v, w \in R(s)\}$, such that there exists a path from s to v in T, for all $v \in R(s)$.

For an out-tree T of s and every $v \in T$, let $d_T(v) = w(T[s, v])$.

Let $D(T) = \sum_{v \in R(s)} d_T(v)$.

Lemma: An out-tree T of s is a shortest path tree if and only if D(T) is minimal among all out-trees of s.

Let T and T' be two out-trees of s such that

- T is a shortest path tree and
- $D(T')$ is minimal among all out-trees of s. In particular, $D(T') \leq D(T)$.

A Characterization of Shortest Path Trees

An out-tree of s is a spanning tree T of $G[R(s)] = (R(s), E[R(s)])$, where $E[R(s)] = \{(v, w) \in E \mid v, w \in R(s)\}$, such that there exists a path from s to v in T, for all $v \in R(s)$.

For an out-tree T of s and every $v \in T$, let $d_T(v) = w(T[s, v])$.

Let $D(T) = \sum_{v \in R(s)} d_T(v)$.

Lemma: An out-tree T of s is a shortest path tree if and only if $D(T)$ is minimal among all out-trees of s.

Let T and T' be two out-trees of s such that

- T is a shortest path tree and
- $D(T')$ is minimal among all out-trees of s. In particular, $D(T') \le D(T)$.

If $D(T') < D(T)$, there exists some vertex $v \in R(s)$ such that $d_{T'}(v) < d_T(v)$.

A Characterization of Shortest Path Trees

An out-tree of s is a spanning tree T of $G[R(s)] = (R(s), E[R(s)])$, where $E[R(s)] = \{(v, w) \in E \mid v, w \in R(s)\}$, such that there exists a path from s to v in T, for all $v \in R(s)$.

For an out-tree T of s and every $v \in T$, let $d_T(v) = w(T[s, v])$.

Let $D(T) = \sum_{v \in R(s)} d_T(v)$.

Lemma: An out-tree T of s is a shortest path tree if and only if $D(T)$ is minimal among all out-trees of s.

Let T and T' be two out-trees of s such that

- T is a shortest path tree and
- $D(T')$ is minimal among all out-trees of s. In particular, $D(T') \le D(T)$.

If $D(T') < D(T)$, there exists some vertex $v \in R(s)$ such that $d_{T'}(v) < d_T(v)$.

$\Rightarrow$ T is not a shortest path tree, a contradiction.

A Characterization of Shortest Path Trees

An out-tree of s is a spanning tree T of $G[R(s)] = (R(s), E[R(s)])$, where $E[R(s)] = \{(v, w) \in E \mid v, w \in R(s)\}$, such that there exists a path from s to v in T, for all $v \in R(s)$.

For an out-tree T of s and every $v \in T$, let $d_T(v) = w(T[s, v])$.

Let $D(T) = \sum_{v \in R(s)} d_T(v)$.

Lemma: An out-tree T of s is a shortest path tree if and only if $D(T)$ is minimal among all out-trees of s.

Let T and T' be two out-trees of s such that

- T is a shortest path tree and
- $D(T')$ is minimal among all out-trees of s. In particular, $D(T') \le D(T)$.

If $D(T') < D(T)$, there exists some vertex $v \in R(s)$ such that $d_{T'}(v) < d_T(v)$.

$\Rightarrow$ T is not a shortest path tree, a contradiction.

$\Rightarrow$ $D(T) = D(T')$.

A Characterization of Shortest Path Trees

An out-tree of s is a spanning tree T of $G[R(s)] = (R(s), E[R(s)])$, where $E[R(s)] = \{(v, w) \in E \mid v, w \in R(s)\}$, such that there exists a path from s to v in T, for all $v \in R(s)$.

For an out-tree T of s and every $v \in T$, let $d_T(v) = w(T[s, v])$.

Let $D(T) = \sum_{v \in R(s)} d_T(v)$.

Lemma: An out-tree T of s is a shortest path tree if and only if $D(T)$ is minimal among all out-trees of s.

Let T and T' be two out-trees of s such that $D(T) = D(T')$ is minimal among all out-trees of s and

- T is a shortest path tree,
- T' is not.

A Characterization of Shortest Path Trees

An out-tree of s is a spanning tree T of $G[R(s)] = (R(s), E[R(s)])$, where $E[R(s)] = \{(v, w) \in E \mid v, w \in R(s)\}$, such that there exists a path from s to v in T, for all $v \in R(s)$.

For an out-tree T of s and every $v \in T$, let $d_T(v) = w(T[s, v])$.

Let $D(T) = \sum_{v \in R(s)} d_T(v)$.

Lemma: An out-tree T of s is a shortest path tree if and only if $D(T)$ is minimal among all out-trees of s.

Let T and T' be two out-trees of s such that $D(T) = D(T')$ is minimal among all out-trees of s and

- T is a shortest path tree,
- T' is not.

$\Rightarrow$ There exists a vertex $v \in R(s)$ such that $d_T(v) < d_{T'}(v)$.

A Characterization of Shortest Path Trees

An out-tree of s is a spanning tree T of $G[R(s)] = (R(s), E[R(s)])$, where $E[R(s)] = \{(v, w) \in E \mid v, w \in R(s)\}$, such that there exists a path from s to v in T, for all $v \in R(s)$.

For an out-tree T of s and every $v \in T$, let $d_T(v) = w(T[s, v])$.

Let $D(T) = \sum_{v \in R(s)} d_T(v)$.

Lemma: An out-tree T of s is a shortest path tree if and only if $D(T)$ is minimal among all out-trees of s.

Let T and T' be two out-trees of s such that $D(T) = D(T')$ is minimal among all out-trees of s and

- T is a shortest path tree,
- T' is not.

$\Rightarrow$ There exists a vertex $v \in R(s)$ such that $d_T(v) < d_{T'}(v)$.

$\Rightarrow$ There exists a vertex $v' \in R(s)$ such that $d_{T'}(v') < d_T(v')$, a contradiction.

A Characterization of Shortest Path Trees

An out-tree of s is a spanning tree T of $G[R(s)] = (R(s), E[R(s)])$, where $E[R(s)] = \{(v, w) \in E \mid v, w \in R(s)\}$, such that there exists a path from s to v in T, for all $v \in R(s)$.

For an out-tree T of s and every $v \in T$, let $d_T(v) = w(T[s, v])$.

Let $D(T) = \sum_{v \in R(s)} d_T(v)$.

Lemma: An out-tree T of s is a shortest path tree if and only if $D(T)$ is minimal among all out-trees of s.

Let T and T' be two out-trees of s such that $D(T) = D(T')$ is minimal among all out-trees of s and

- T is a shortest path tree,
- T' is not.

$\Rightarrow$ There exists a vertex $v \in R(s)$ such that $d_T(v) < d_{T'}(v)$.

$\Rightarrow$ There exists a vertex $v' \in R(s)$ such that $d_{T'}(v') < d_T(v')$, a contradiction.

$\Rightarrow$ T' is a shortest path tree.

Dijkstra's Algorithm

Build a shortest-path tree by starting with s and adding vertices in R(s) one by one.

Dijkstra's Algorithm

Build a shortest-path tree by starting with s and adding vertices in R(s) one by one.

In each step, we can only add out-neighbours of vertices already in T.

Dijkstra's Algorithm

Build a shortest-path tree by starting with s and adding vertices in $R(s)$ one by one.

In each step, we can only add out-neighbours of vertices already in T.

A greedy choice:

Add the vertex $v \notin T$ that minimizes $d_T(v)$.

Dijkstra's Algorithm

Build a shortest-path tree by starting with s and adding vertices in R(s) one by one.

In each step, we can only add out-neighbours of vertices already in T.

A greedy choice:

Add the vertex $v \notin T$ that minimizes $d_T(v)$.

Dijkstra(G, s)

```
T = ({s}, ∅)
while some vertex in T has an out-neighbour not in T
    do choose an edge (u, v) such that
         • u ∈ T,
         • v ∉ T, and
         • d_T(u) + w(u, v) is minimized.
       add v and (u, v) to T
return T
```

Dijkstra's Algorithm

Dijkstra(G, s)

```
T = (V, ∅)
mark every vertex of G as unexplored
set d(v) = +∞ and e(v) = nil for every vertex v ∈ G
mark s as explored and set d(v) = 0
Q = an empty priority queue
for every edge (s, v) incident to s
    do Q.insert(v, w(s, v))
       d(v) = w(s, v)
       e(v) = (s, v)
while not Q.isEmpty()
    do u = Q.deleteMin()
       mark u as explored
       add e(u) to T
       for every edge (u, v) incident to u
           do if v is unexplored and (v ∉ Q or d(u) + w(u, v) < d(v))
                 then d(v) = d(u) + w(u, v)
                      e(v) = (u, v)
                      if v ∉ Q
                         then Q.insert(v, d(v))
                         else Q.decreaseKey(v, d(v))
return T
```

Dijkstra's Algorithm

Dijkstra(G, s)

```
T = (V, ∅)
mark every vertex of G as unexplored
set d(v) = +∞ and e(v) = nil for every vertex v ∈ G
mark s as explored and set d(v) = 0
Q = an empty priority queue
for every edge (s, v) incident to s
   do Q.insert(v, w(s, v))
      d(v) = w(s, v)
      e(v) = (s, v)
while not Q.isEmpty()
   do u = Q.deleteMin()
      mark u as explored
      add e(u) to T
      for every edge (u, v) incident to u
         do if v is unexplored and (v ∉ Q or d(u) + w(u, v) < d(v))
               then d(v) = d(u) + w(u, v)
                    e(v) = (u, v)
                    if v ∉ Q
                       then Q.insert(v, d(v))
                       else Q.decreaseKey(v, d(v))
return T
```

This is the same as Prim's algorithm, except that vertex priorities are calculated differently.

Dijkstra's Algorithm

Dijkstra(G, s)

```
T = (V, ∅)
mark every vertex of G as unexplored
set d(v) = +∞ and e(v) = nil for every vertex v ∈ G
mark s as explored and set d(v) = 0
Q = an empty priority queue
for every edge (s, v) incident to s
    do Q.insert(v, w(s, v))
       d(v) = w(s, v)
       e(v) = (s, v)
while not Q.isEmpty()
    do u = Q.deleteMin()
       mark u as explored
       add e(u) to T
       for every edge (u, v) incident to u
           do if v is unexplored and (v ∉ Q or d(u) + w(u, v) < d(v))
                 then d(v) = d(u) + w(u, v)
                      e(v) = (u, v)
                      if v ∉ Q
                         then Q.insert(v, d(v))
                         else Q.decreaseKey(v, d(v))
return T
```

This is the same as Prim's algorithm, except that vertex priorities are calculated differently.

$\Rightarrow$ Dijkstra's algorithm takes $O(n \lg n + m)$ time.

Correctness of Dijkstra's Algorithm

Dijkstra's algorithm does not necessarily produce a shortest path tree if there are edges with negative weights!

Correctness of Dijkstra's Algorithm

Dijkstra's algorithm does not necessarily produce a shortest path tree if there are edges with negative weights!

Lemma: If all edges in G have non-negative weights, then Dijkstra's algorithm computes a shortest path tree of G.

Correctness of Dijkstra's Algorithm

Dijkstra's algorithm does not necessarily produce a shortest path tree if there are edges with negative weights!

Lemma: If all edges in G have non-negative weights, then Dijkstra's algorithm computes a shortest path tree of G.

Assume the contrary and let v be the first vertex added to T such that $d_T(v) > \text{dist}(s, v)$.

Correctness of Dijkstra's Algorithm

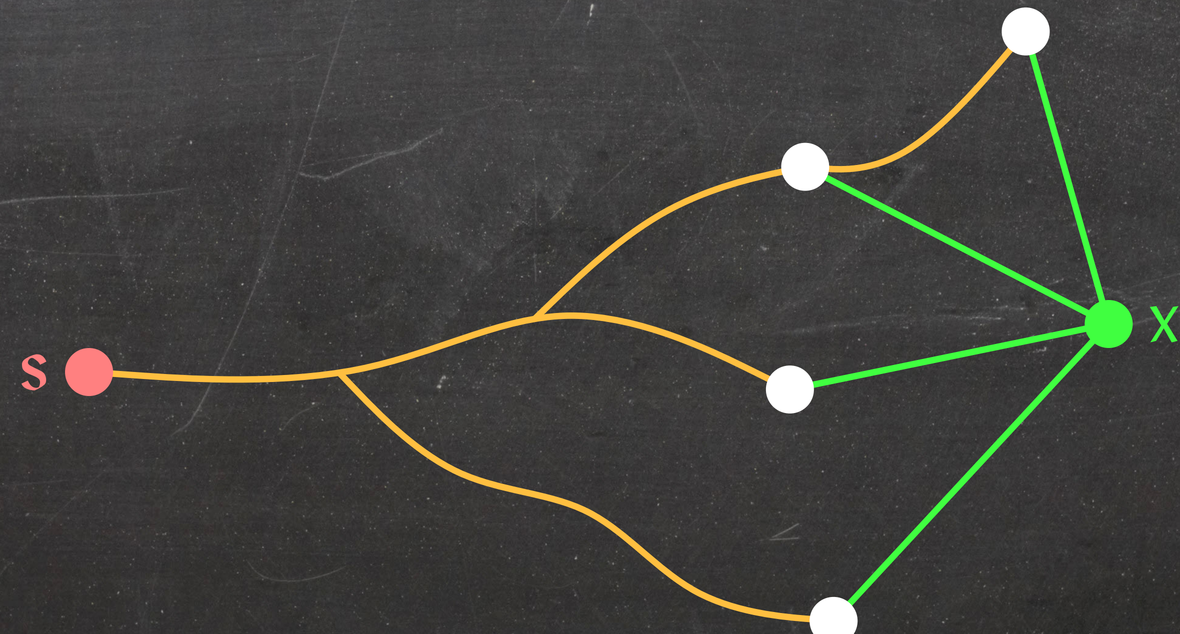
Dijkstra's algorithm does not necessarily produce a shortest path tree if there are edges with negative weights!

Lemma: If all edges in G have non-negative weights, then Dijkstra's algorithm computes a shortest path tree of G.

Assume the contrary and let v be the first vertex added to T such that $d_T(v) > \mathrm{dist}(s, v)$.

For every vertex $x \notin T$, we have

$$d(x) = \min_{\substack{(u,x)\in E\\ u\in T}} d(u) + w(u, x) = \min_{\substack{(u,x)\in E\\ u\in T}} \mathrm{dist}(s, u) + w(u, x).$$



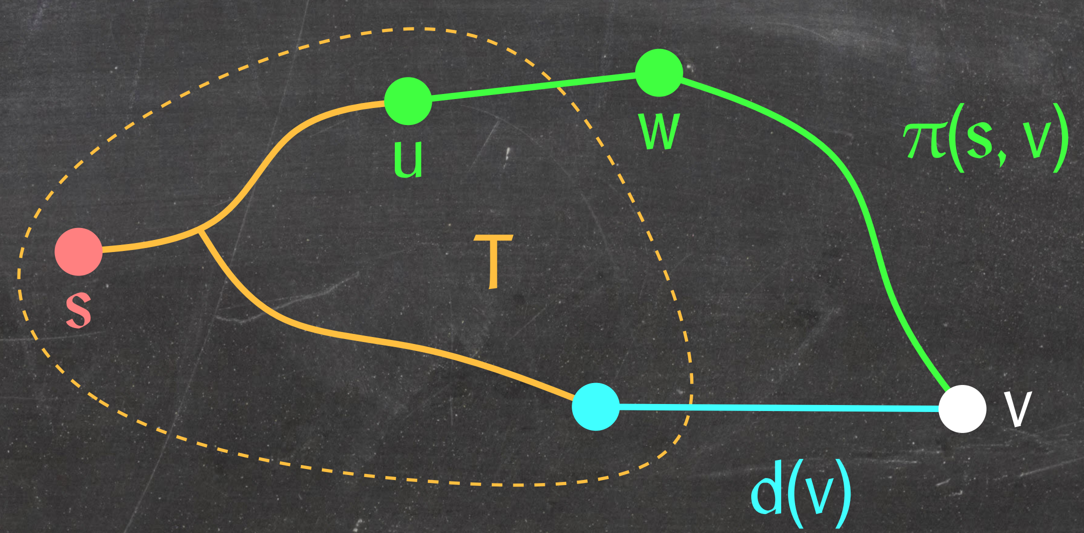
Correctness of Dijkstra's Algorithm

Dijkstra's algorithm does not necessarily produce a shortest path tree if there are edges with negative weights!

Lemma: If all edges in G have non-negative weights, then Dijkstra's algorithm computes a shortest path tree of G.

Assume the contrary and let v be the first vertex added to T such that $d_T(v) > dist(s, v)$.

The shortest path $\pi(s, v)$ from s to v must include a vertex $w \notin T$ whose predecessor u in $\pi(s, v)$ belongs to T.



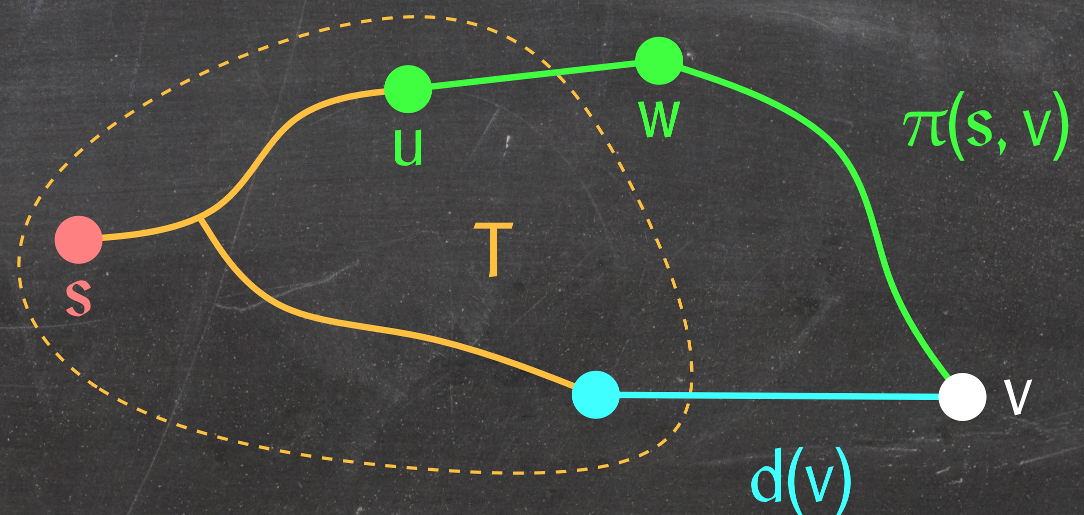
Correctness of Dijkstra's Algorithm

Dijkstra's algorithm does not necessarily produce a shortest path tree if there are edges with negative weights!

Lemma: If all edges in G have non-negative weights, then Dijkstra's algorithm computes a shortest path tree of G.

Assume the contrary and let v be the first vertex added to T such that $d_T(v) > \text{dist}(s, v)$.

The shortest path $\pi(s, v)$ from s to v must include a vertex $w \notin T$ whose predecessor u in $\pi(s, v)$ belongs to T.



$\Rightarrow\ d(w) \leq \text{dist}(s, u) + w(u, w) = \text{dist}(s, w) \leq \text{dist}(s, v) < d(v).$

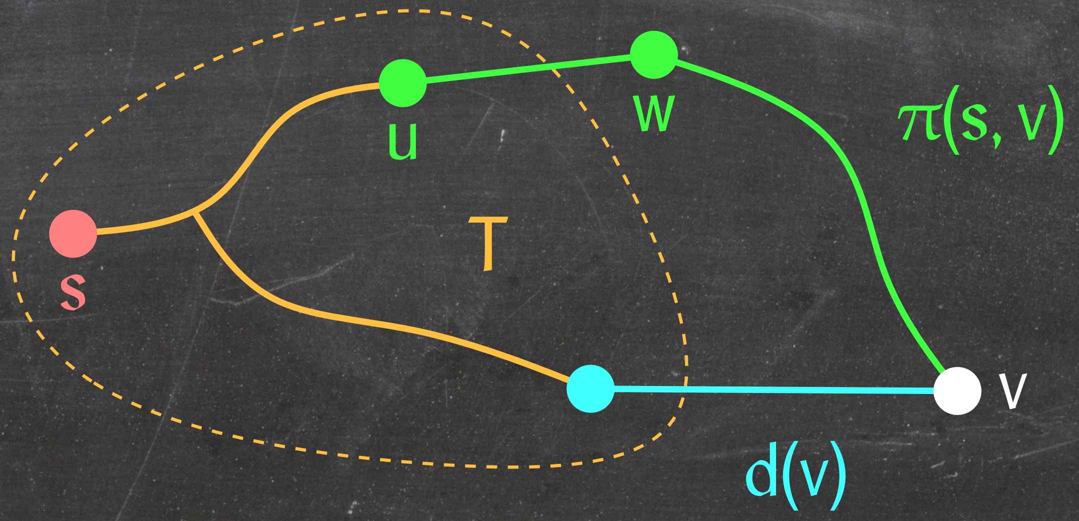
Correctness of Dijkstra's Algorithm

Dijkstra's algorithm does not necessarily produce a shortest path tree if there are edges with negative weights!

Lemma: If all edges in G have non-negative weights, then Dijkstra's algorithm computes a shortest path tree of G.

Assume the contrary and let v be the first vertex added to T such that $d_T(v) > \text{dist}(s, v)$.

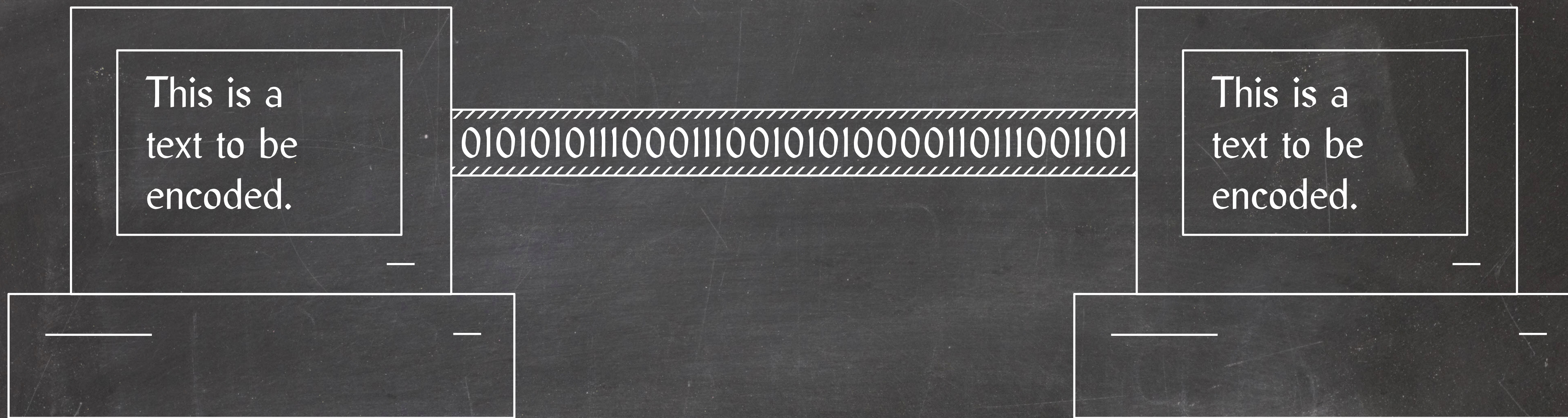
The shortest path $\pi(s, v)$ from s to v must include a vertex $w \notin T$ whose predecessor u in $\pi(s, v)$ belongs to T.

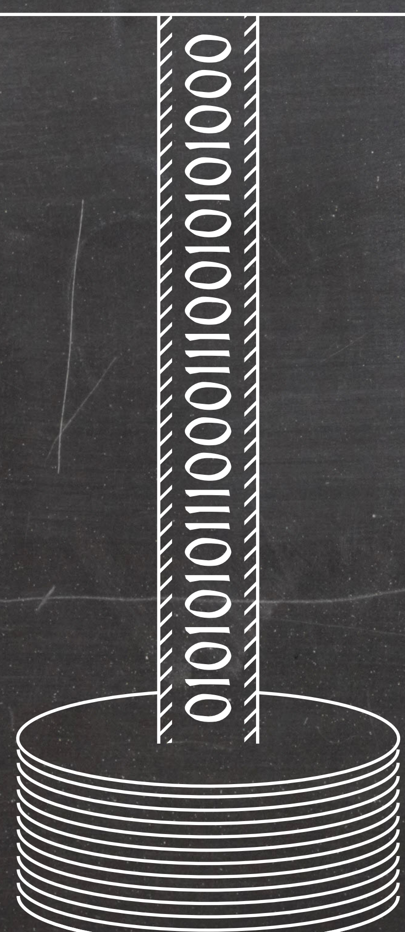


$\Rightarrow$ $d(w) \leq \text{dist}(s, u) + w(u, w) = \text{dist}(s, w) \leq \text{dist}(s, v) < d(v)$.

$\Rightarrow$ v is not the next vertex we add to T, a contradiction.

Minimum Length Codes





Goal:

- Encode a given text using as few bits as possible:
  - Limit amount of disk space required to store the text.
  - Send the text over a potentially slow network.
  - …

Codes That Can Be Decoded

A code is a mapping $C(\cdot)$ that maps every character x to a bit string $C(x)$, called the encoding of x.

Codes That Can Be Decoded

A code is a mapping $C(\cdot)$ that maps every character x to a bit string $C(x)$, called the encoding of x.

	e	f	i	p	r	x	-
C_1	000	001	010	011	100	101	110

Codes That Can Be Decoded

A code is a mapping $C(\cdot)$ that maps every character x to a bit string $C(x)$, called the encoding of x.

For a text $T = \langle x_1, x_2, \ldots, x_n \rangle$, let $C(T) = C(x_1) \circ C(x_2) \circ \cdots \circ C(x_n)$ be the bit string obtained by concatenating the encodings of its characters. We call $C(T)$ the encoding of T.

	e	f	i	p	r	x	-
C_1	000	001	010	011	100	101	110

Codes That Can Be Decoded

A code is a mapping $C(\cdot)$ that maps every character x to a bit string $C(x)$, called the encoding of x.

For a text $T = \langle x_1, x_2, \ldots, x_n \rangle$, let $C(T) = C(x_1) \circ C(x_2) \circ \cdots \circ C(x_n)$ be the bit string obtained by concatenating the encodings of its characters. We call $C(T)$ the encoding of T.

"prefix-free"

	e	f	i	p	r	x	-
C_1	000	001	010	011	100	101	110

C_1(prefix-free) = 011 100 000 001 010 101 110 001 100 000 000 (33 bits)

Codes That Can Be Decoded

A code is a mapping $C(\cdot)$ that maps every character x to a bit string $C(x)$, called the encoding of x.

For a text $T = \langle x_1, x_2, \ldots, x_n \rangle$, let $C(T) = C(x_1) \circ C(x_2) \circ \cdots \circ C(x_n)$ be the bit string obtained by concatenating the encodings of its characters. We call $C(T)$ the encoding of T.

"prefix-free"

	e	f	i	p	r	x	-
C_1	000	001	010	011	100	101	110
C_2	00	010	0110	0111	10	110	111

C_1(prefix-free) = 011 100 000 001 010 101 110 001 100 000 000 (33 bits)

C_2(prefix-free) = 0111 10 00 010 0110 110 111 010 10 00 00 (30 bits)

Codes That Can Be Decoded

A code is a mapping $C(\cdot)$ that maps every character x to a bit string $C(x)$, called the encoding of x.

For a text $T = \langle x_1, x_2, \ldots, x_n \rangle$, let $C(T) = C(x_1) \circ C(x_2) \circ \cdots \circ C(x_n)$ be the bit string obtained by concatenating the encodings of its characters. We call $C(T)$ the encoding of T.

"prefix-free"

	e	f	i	p	r	x	-
C_1	000	001	010	011	100	101	110
C_2	00	010	0110	0111	10	110	111
C_3	0	1	00	01	10	11	000

C_1(prefix-free) = 011 100 000 001 010 101 110 001 100 000 000 (33 bits)

C_2(prefix-free) = 0111 10 00 010 0110 110 111 010 10 00 00 (30 bits)

C_3(prefix-free) = 01 10 0 1 00 11 000 1 10 0 0 (18 bits)

Codes That Can Be Decoded

A code is a mapping $C(\cdot)$ that maps every character x to a bit string $C(x)$, called the encoding of x.

For a text $T = \langle x_1, x_2, \ldots, x_n \rangle$, let $C(T) = C(x_1) \circ C(x_2) \circ \cdots \circ C(x_n)$ be the bit string obtained by concatenating the encodings of its characters. We call $C(T)$ the encoding of T.

"prefix-free"

	e	f	i	p	r	x	-
C_1	000	001	010	011	100	101	110
C_2	00	010	0110	0111	10	110	111
C_3	0	1	00	01	10	11	000

C_1(prefix-free) = 011 100 000 001 010 101 110 001 100 000 000 (33 bits)

C_2(prefix-free) = 0111 10 00 010 0110 110 111 010 10 00 00 (30 bits)

C_3(prefix-free) = 01 10 0 1 00 11 000 1 10 0 0 (18 bits)

A code $C(\cdot)$ is prefix-free if there are no two characters x and y such that $C(x)$ is a prefix of $C(y)$.

Codes That Can Be Decoded

A code is a mapping $C(\cdot)$ that maps every character x to a bit string $C(x)$, called the encoding of x.

For a text $T = \langle x_1, x_2, \ldots, x_n \rangle$, let $C(T) = C(x_1) \circ C(x_2) \circ \cdots \circ C(x_n)$ be the bit string obtained by concatenating the encodings of its characters. We call $C(T)$ the encoding of T.

"prefix-free"

	e	f	i	p	r	x	-
C_1	000	001	010	011	100	101	110
C_2	00	010	0110	0111	10	110	111
C_3	0	1	00	01	10	11	000

C_1(prefix-free) = 011 100 000 001 010 101 110 001 100 000 000 (33 bits)

C_2(prefix-free) = 0111 10 00 010 0110 110 111 010 10 00 00 (30 bits)

C_3(prefix-free) = 01 10 0 1 00 11 000 1 10 0 0 (18 bits)

A code $C(\cdot)$ is prefix-free if there are no two characters x and y such that $C(x)$ is a prefix of $C(y)$.

Non-prefix-free codes cannot always be decoded uniquely!

Codes That Can Be Decoded

Lemma: If $C(\cdot)$ is a prefix-free code and $T \neq T'$, then $C(T) \neq C(T')$.

Codes That Can Be Decoded

Lemma: If $C(\cdot)$ is a prefix-free code and $T \neq T'$, then $C(T) \neq C(T')$.

Let $T = \langle x_1, x_2, \ldots, x_m \rangle$ and $T' = \langle y_1, y_2, \ldots, y_n \rangle$ and assume $C(T) = C(T')$.

$C(T)$ $\boxed{}$

$C(T')$ $\boxed{}$

Codes That Can Be Decoded

Lemma: If $C(\cdot)$ is a prefix-free code and $T \neq T'$, then $C(T) \neq C(T')$.

Let $T = \langle x_1, x_2, \ldots, x_m \rangle$ and $T' = \langle y_1, y_2, \ldots, y_n \rangle$ and assume $C(T) = C(T')$.

Let i be the minimum index such that $x_i \neq y_i$.

$C(T)$

$C(T')$

Codes That Can Be Decoded

Lemma: If $C(\cdot)$ is a prefix-free code and $T \neq T'$, then $C(T) \neq C(T')$.

Let $T = \langle x_1, x_2, \ldots, x_m \rangle$ and $T' = \langle y_1, y_2, \ldots, y_n \rangle$ and assume $C(T) = C(T')$.

Let i be the minimum index such that $x_i \neq y_i$.

$\Rightarrow$ $C(\langle x_1, x_2, \ldots, x_{i-1} \rangle) = C(\langle y_1, y_2, \ldots, y_{i-1} \rangle)$ and
$C(\langle x_i, x_{i+1}, \ldots, x_m \rangle) = C(\langle y_i, y_{i+1}, \ldots, y_n \rangle)$.

$C(T)$	$C(\langle x_1, x_2, \ldots, x_{i-1} \rangle)$	$C(x_i)$	$C(\langle x_{i+1}, x_{i+2}, \ldots, x_m \rangle)$
$C(T')$	$C(\langle y_1, y_2, \ldots, y_{i-1} \rangle)$	$C(y_i)$	$C(\langle y_{i+1}, y_{i+2}, \ldots, y_n \rangle)$

Codes That Can Be Decoded

Lemma: If $C(\cdot)$ is a prefix-free code and $T \neq T'$, then $C(T) \neq C(T')$.

Let $T = \langle x_1, x_2, \ldots, x_m \rangle$ and $T' = \langle y_1, y_2, \ldots, y_n \rangle$ and assume $C(T) = C(T')$.

Let i be the minimum index such that $x_i \neq y_i$.

$$\Rightarrow \begin{array}{l} C(\langle x_1, x_2, \ldots, x_{i-1} \rangle) = C(\langle y_1, y_2, \ldots, y_{i-1} \rangle) \text{ and} \\ C(\langle x_i, x_{i+1}, \ldots, x_m \rangle) = C(\langle y_i, y_{i+1}, \ldots, y_n \rangle). \end{array}$$

Assume w.l.o.g. that $|C(x_i)| \leq |C(y_i)|$.

$C(T)$	$C(\langle x_1, x_2, \ldots, x_{i-1} \rangle)$	$C(x_i)$	$C(\langle x_{i+1}, x_{i+2}, \ldots, x_m \rangle)$
$C(T')$	$C(\langle y_1, y_2, \ldots, y_{i-1} \rangle)$	$C(y_i)$	$C(\langle y_{i+1}, y_{i+2}, \ldots, y_n \rangle)$

Codes That Can Be Decoded

Lemma: If $C(\cdot)$ is a prefix-free code and $T \neq T'$, then $C(T) \neq C(T')$.

Let $T = \langle x_1, x_2, \ldots, x_m \rangle$ and $T' = \langle y_1, y_2, \ldots, y_n \rangle$ and assume $C(T) = C(T')$.

Let i be the minimum index such that $x_i \neq y_i$.

$\Rightarrow$ $C(\langle x_1, x_2, \ldots, x_{i-1} \rangle) = C(\langle y_1, y_2, \ldots, y_{i-1} \rangle)$ and
$C(\langle x_i, x_{i+1}, \ldots, x_m \rangle) = C(\langle y_i, y_{i+1}, \ldots, y_n \rangle)$.

Assume w.l.o.g. that $|C(x_i)| \leq |C(y_i)|$.

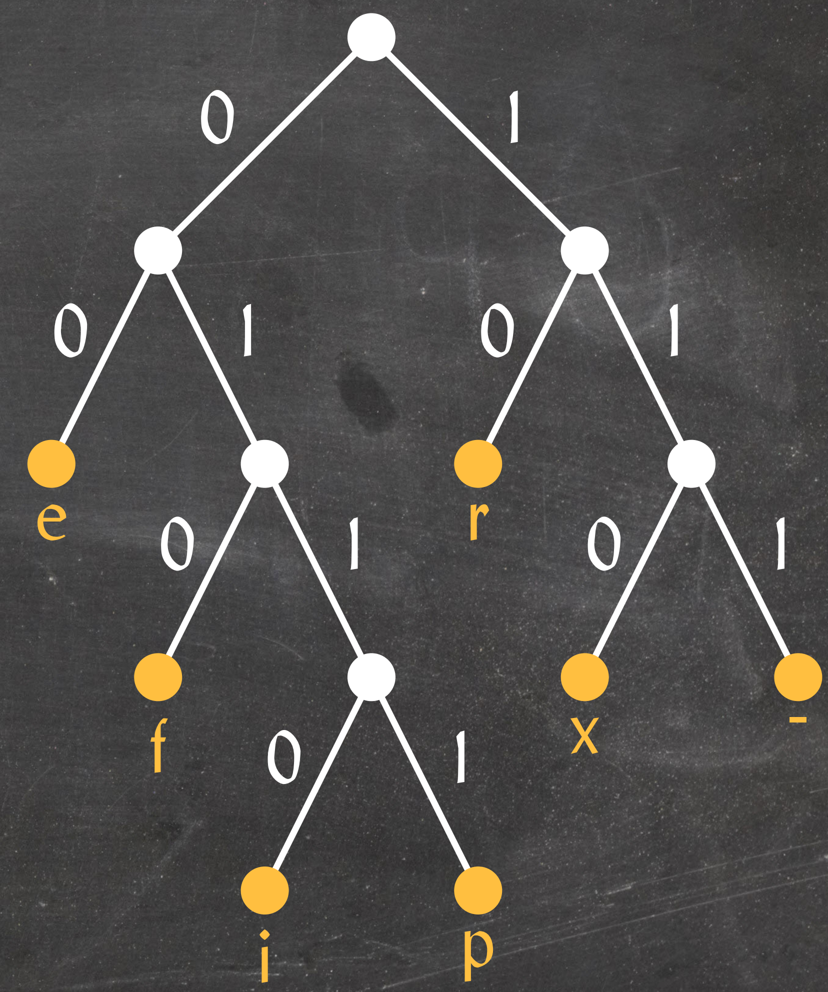
Since both $C(x_i)$ and $C(y_i)$ are prefixes of $C(\langle x_i, x_{i+1}, \ldots, x_m \rangle)$, $C(x_i)$ must be a prefix of $C(y_i)$, a contradiction.

$C(T)$	$C(\langle x_1, x_2, \ldots, x_{i-1} \rangle)$	$C(x_i)$	$C(\langle x_{i+1}, x_{i+2}, \ldots, x_m \rangle)$
$C(T')$	$C(\langle y_1, y_2, \ldots, y_{i-1} \rangle)$	$C(y_i)$	$C(\langle y_{i+1}, y_{i+2}, \ldots, y_n \rangle)$

Prefix Codes and Binary Trees

Observation: Every prefix-free code $C(\cdot)$ can be represented as a binary tree $\mathcal{T}_C$ whose leaves correspond to the letters in the alphabet.

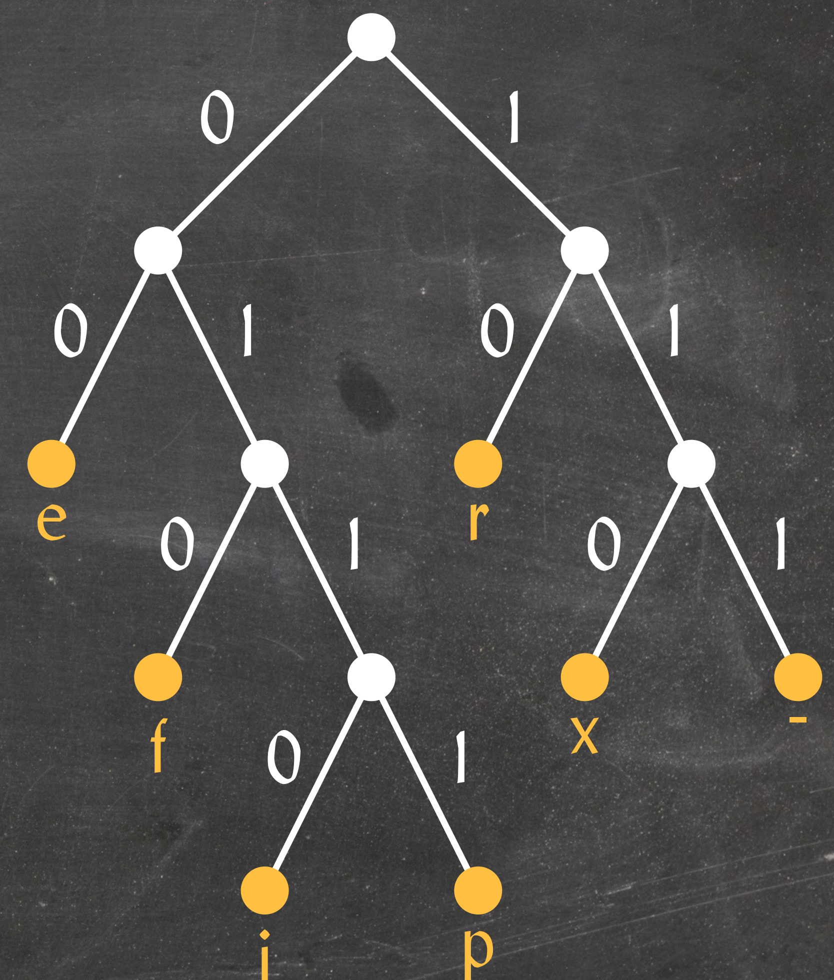
	e	f	i	p	r	x	-
C	00	010	0110	0111	10	110	111



Prefix Codes and Binary Trees

Observation: Every prefix-free code $C(\cdot)$ can be represented as a binary tree $\mathcal{T}_C$ whose leaves correspond to the letters in the alphabet.

	e	f	i	p	r	x	-
C	00	010	0110	0111	10	110	111



The depth of character x in $\mathcal{T}_C$ is the number of bits $|C(x)|$ used to encode x using $C(\cdot)$.

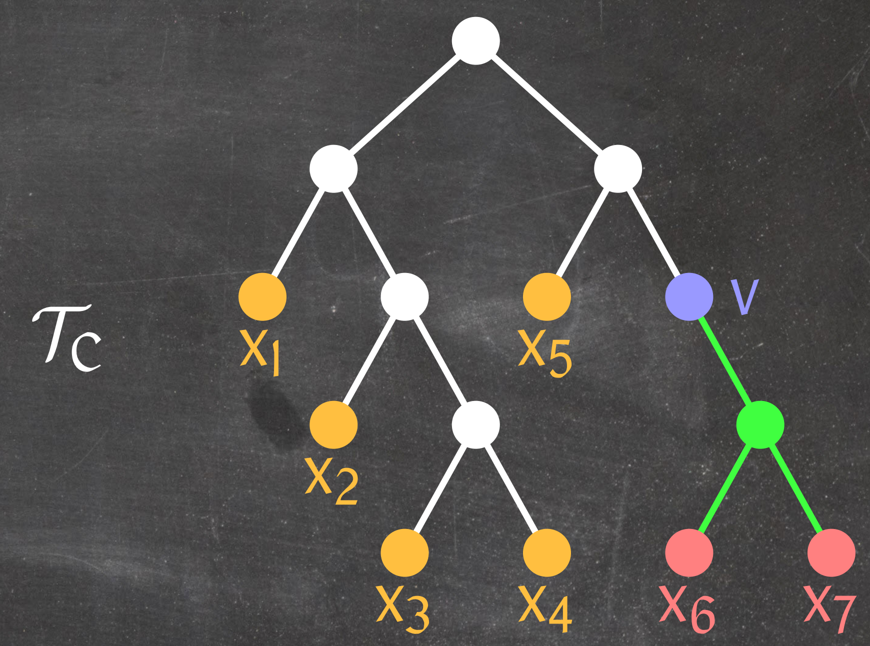
Optimal Prefix Codes and Binary Trees

An optimal prefix-free code for a text T is a prefix-free code C that minimizes $|C(T)|$.

Optimal Prefix Codes and Binary Trees

An optimal prefix-free code for a text T is a prefix-free code C that minimizes |C(T)|.

Lemma: For every text T, there exists an optimal prefix-free code C(·) such that every internal node in $\mathcal{T}_C$ has two children.

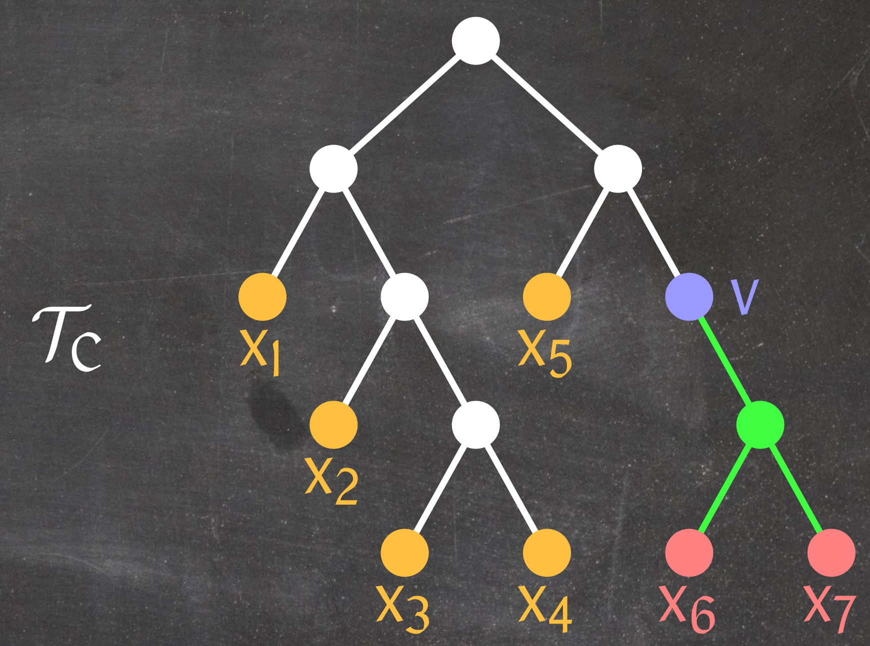


Optimal Prefix Codes and Binary Trees

An optimal prefix-free code for a text T is a prefix-free code C that minimizes |C(T)|.

Lemma: For every text T, there exists an optimal prefix-free code C(·) such that every internal node in $\mathcal{T}_C$ has two children.

Choose C(·) so that $\mathcal{T}_C$ has as few internal nodes with only one child as possible among all optimal prefix-free codes for T.



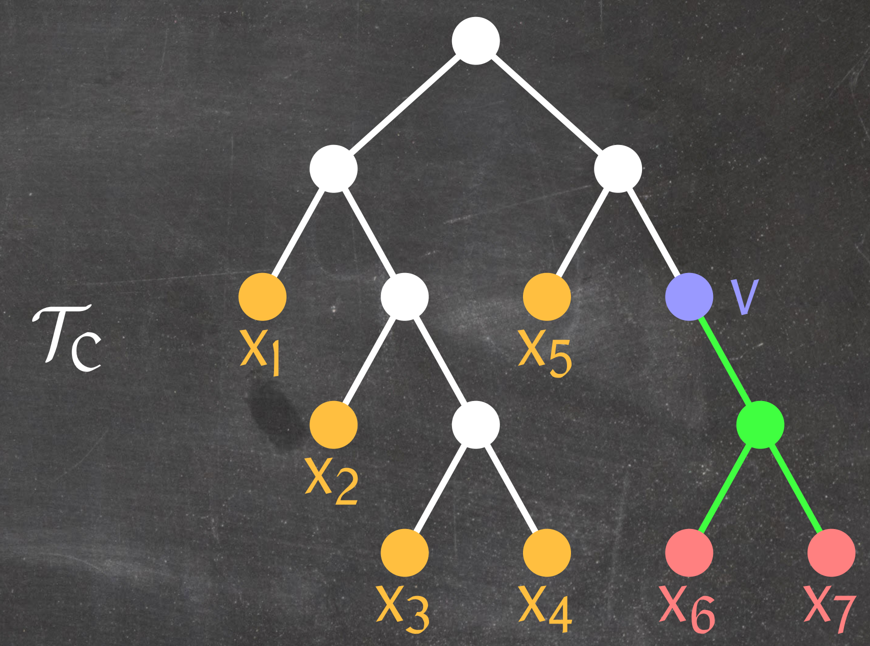
Optimal Prefix Codes and Binary Trees

An optimal prefix-free code for a text T is a prefix-free code C that minimizes |C(T)|.

Lemma: For every text T, there exists an optimal prefix-free code C(·) such that every internal node in $\mathcal{T}_C$ has two children.

Choose C(·) so that $\mathcal{T}_C$ has as few internal nodes with only one child as possible among all optimal prefix-free codes for T.

If $\mathcal{T}_C$ has no internal node with only one child, the lemma holds.



Optimal Prefix Codes and Binary Trees

An optimal prefix-free code for a text T is a prefix-free code C that minimizes $|C(T)|$.

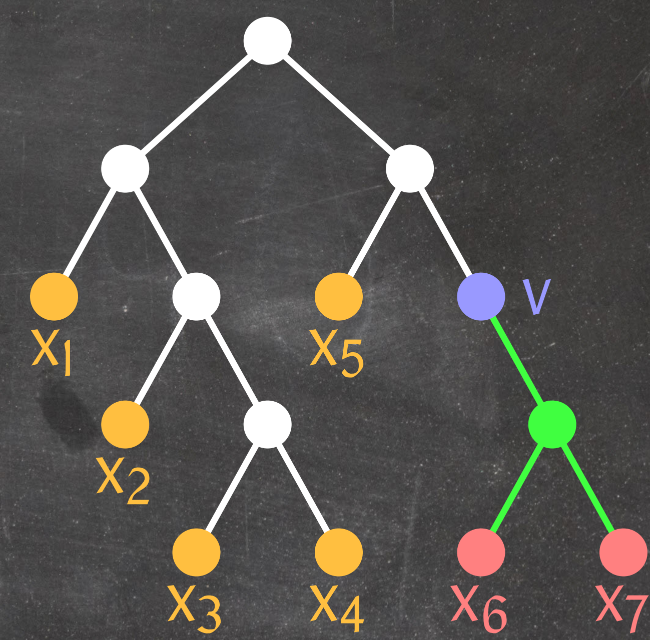
Lemma: For every text T, there exists an optimal prefix-free code $C(\cdot)$ such that every internal node in $\mathcal{T}_C$ has two children.

Choose $C(\cdot)$ so that $\mathcal{T}_C$ has as few internal nodes with only one child as possible among all optimal prefix-free codes for T.

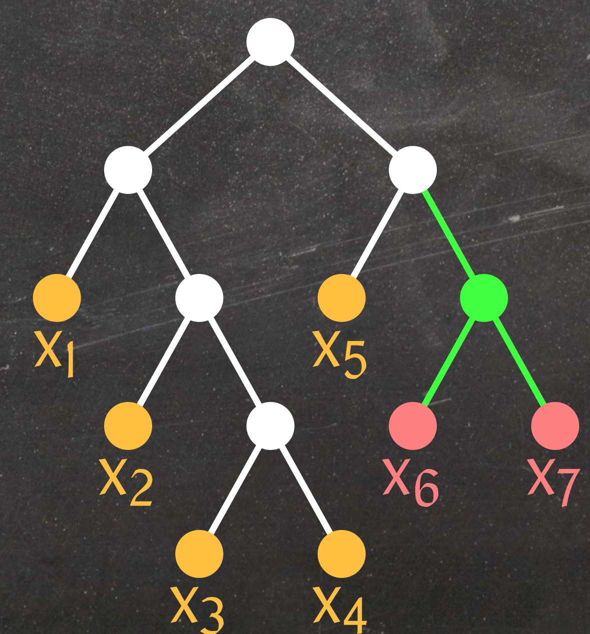
If $\mathcal{T}_C$ has no internal node with only one child, the lemma holds.

Otherwise, choose an internal node v with only one child w and contract the edge (v, w).

$\mathcal{T}_C$



$\mathcal{T}_{C'}$

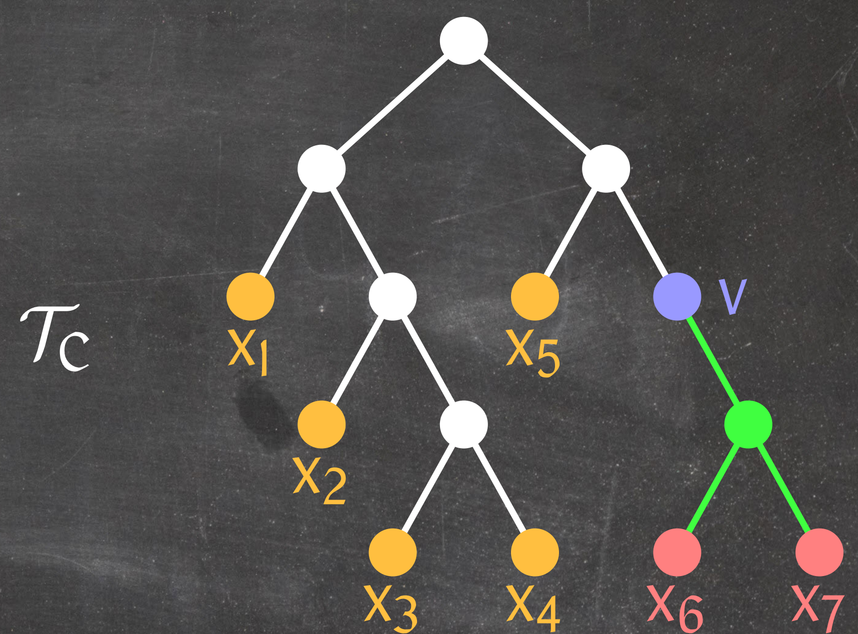


Optimal Prefix Codes and Binary Trees

An optimal prefix-free code for a text T is a prefix-free code C that minimizes $|C(T)|$.

Lemma: For every text T, there exists an optimal prefix-free code $C(\cdot)$ such that every internal node in $\mathcal{T}_C$ has two children.

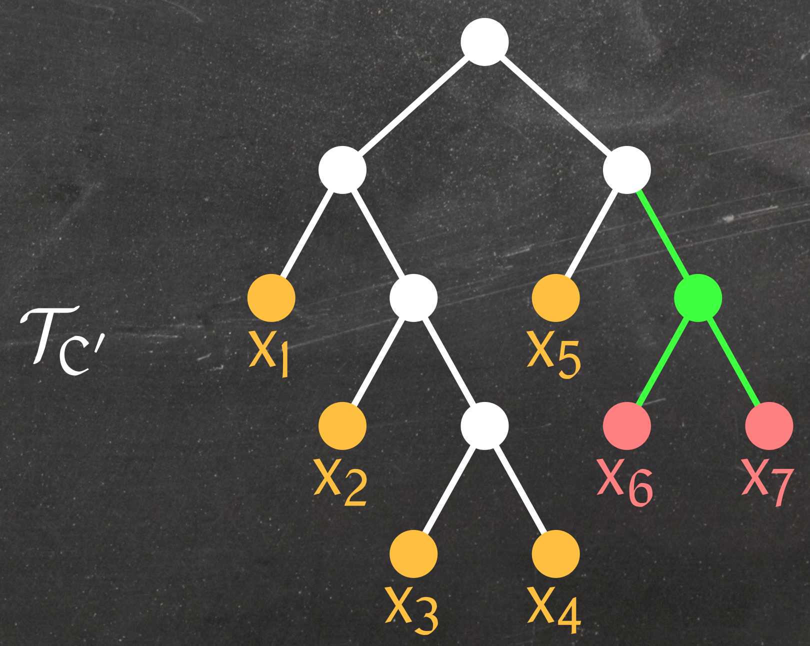
Choose $C(\cdot)$ so that $\mathcal{T}_C$ has as few internal nodes with only one child as possible among all optimal prefix-free codes for T.



If $\mathcal{T}_C$ has no internal node with only one child, the lemma holds.

Otherwise, choose an internal node v with only one child w and contract the edge (v, w).

The resulting tree $\mathcal{T}_{C'}$ has one less internal node with only one child and represents a prefix-free code $C'(\cdot)$ with the property that $|C'(x)| \le |C(x)|$ for every character x.



Optimal Prefix Codes and Binary Trees

An optimal prefix-free code for a text T is a prefix-free code C that minimizes $|C(T)|$.

Lemma: For every text T, there exists an optimal prefix-free code $C(\cdot)$ such that every internal node in $\mathcal{T}_C$ has two children.

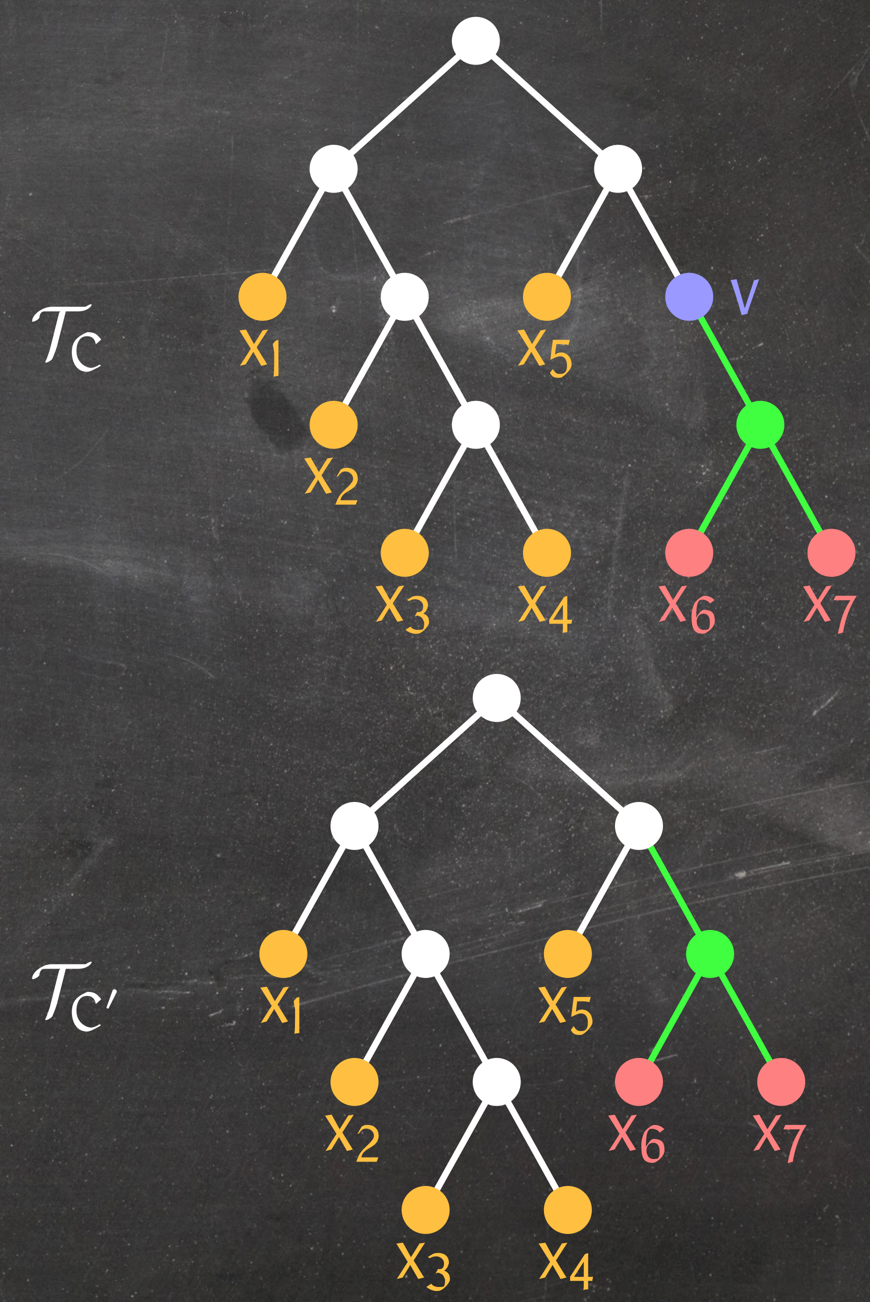
Choose $C(\cdot)$ so that $\mathcal{T}_C$ has as few internal nodes with only one child as possible among all optimal prefix-free codes for T.

If $\mathcal{T}_C$ has no internal node with only one child, the lemma holds.

Otherwise, choose an internal node v with only one child w and contract the edge (v, w).

The resulting tree $\mathcal{T}_{C'}$ has one less internal node with only one child and represents a prefix-free code $C'(\cdot)$ with the property that $|C'(x)| \le |C(x)|$ for every character x.

$\Rightarrow$ $|C'(T)| \le |C(T)|$, contradicting the choice of C.



A Greedy Choice for Optimal Prefix Codes

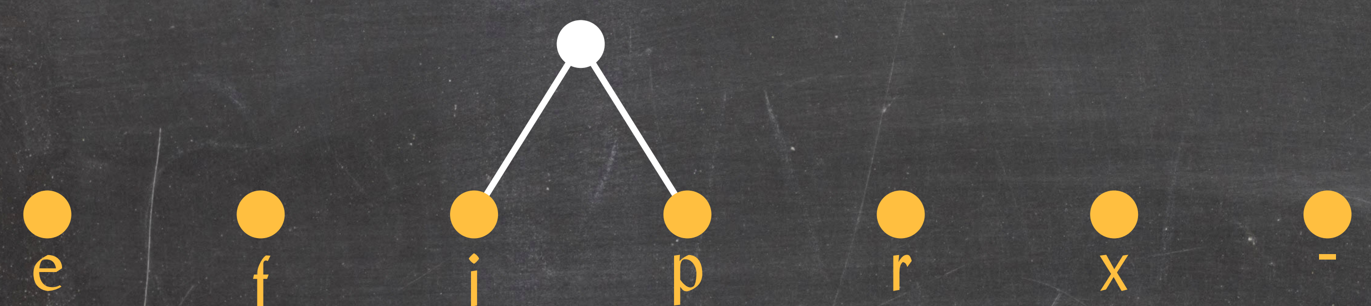
We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.

A Greedy Choice for Optimal Prefix Codes

We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.

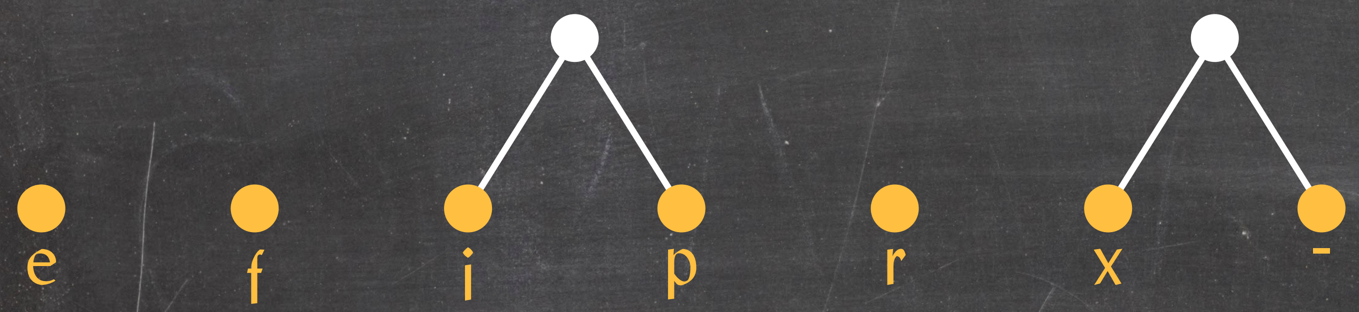
A Greedy Choice for Optimal Prefix Codes

We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.



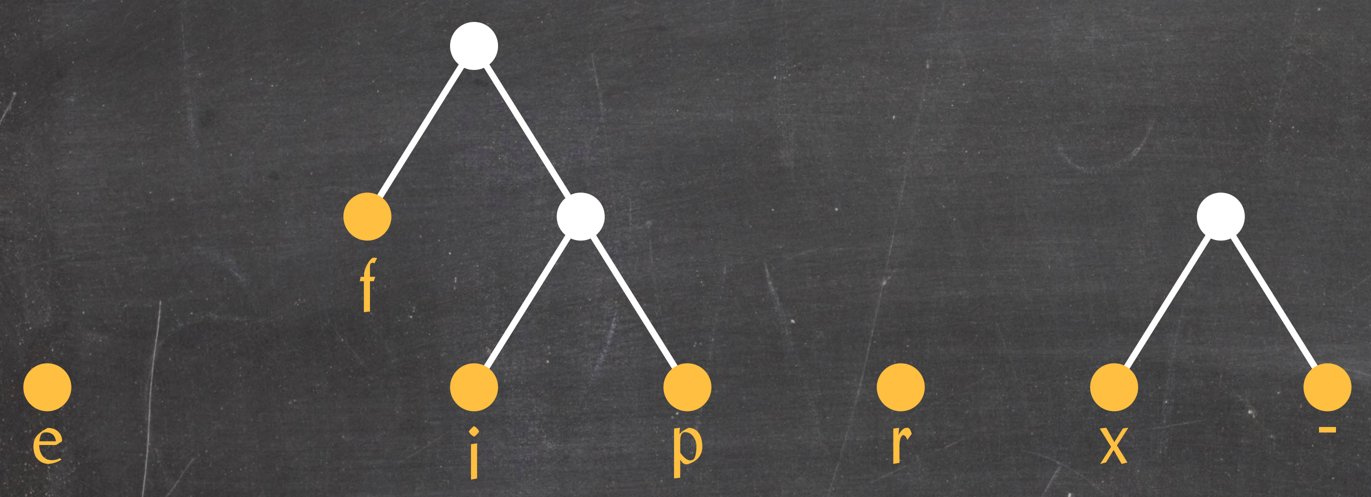
A Greedy Choice for Optimal Prefix Codes

We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.



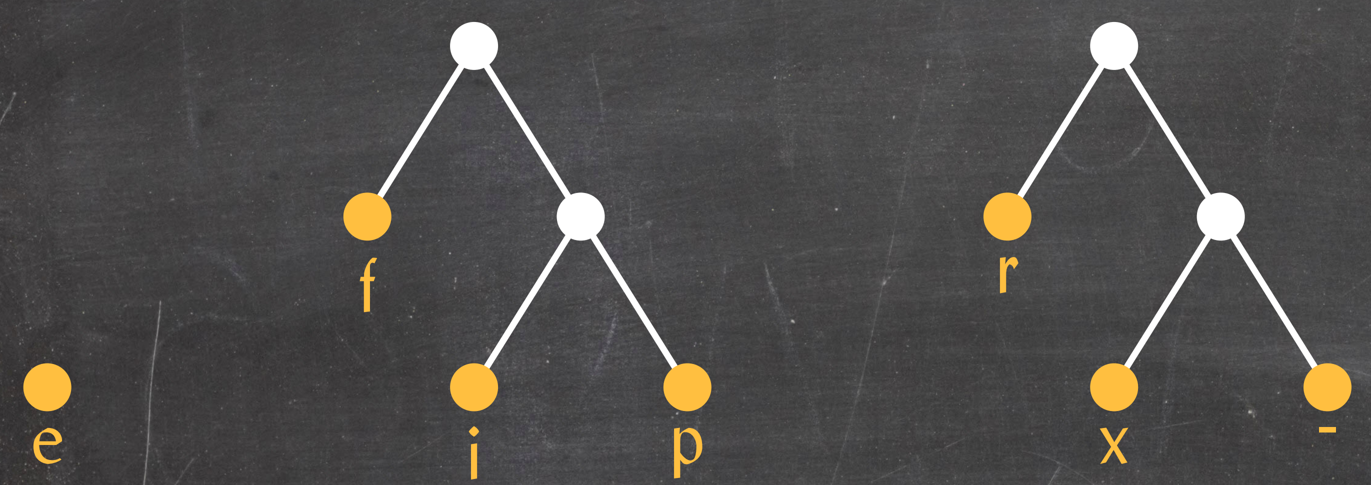
A Greedy Choice for Optimal Prefix Codes

We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.



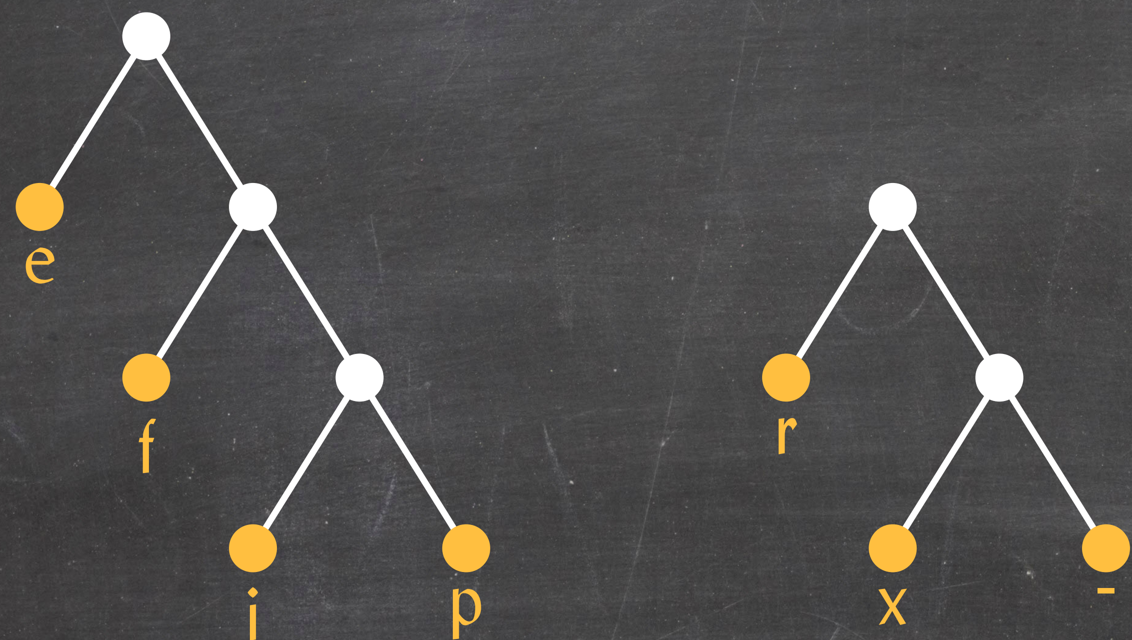
A Greedy Choice for Optimal Prefix Codes

We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.



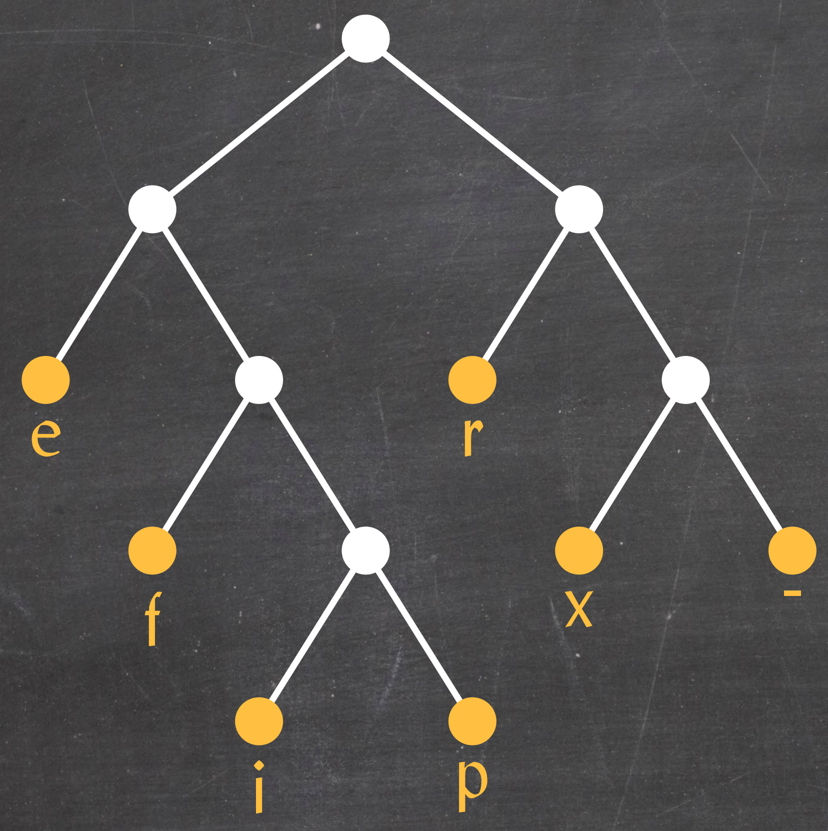
A Greedy Choice for Optimal Prefix Codes

We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.



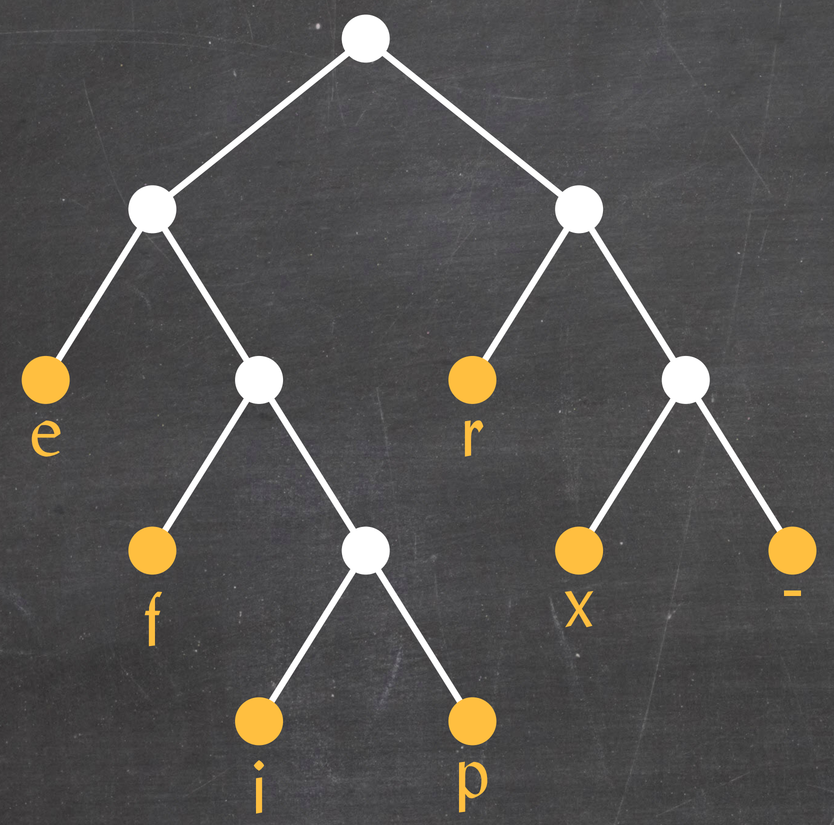
A Greedy Choice for Optimal Prefix Codes

We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.



A Greedy Choice for Optimal Prefix Codes

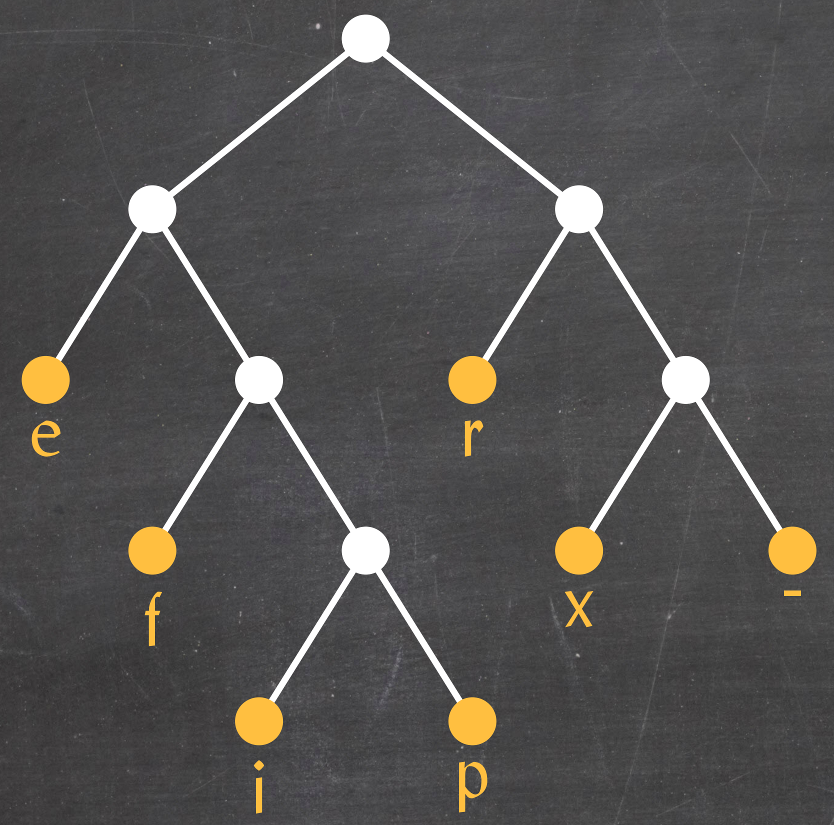
We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.



The length of the encoding of T is $|C(T)| = \sum_x f_T(x)|C(x)|$, where $f_T(x)$ is the frequency of x in T.

A Greedy Choice for Optimal Prefix Codes

We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.

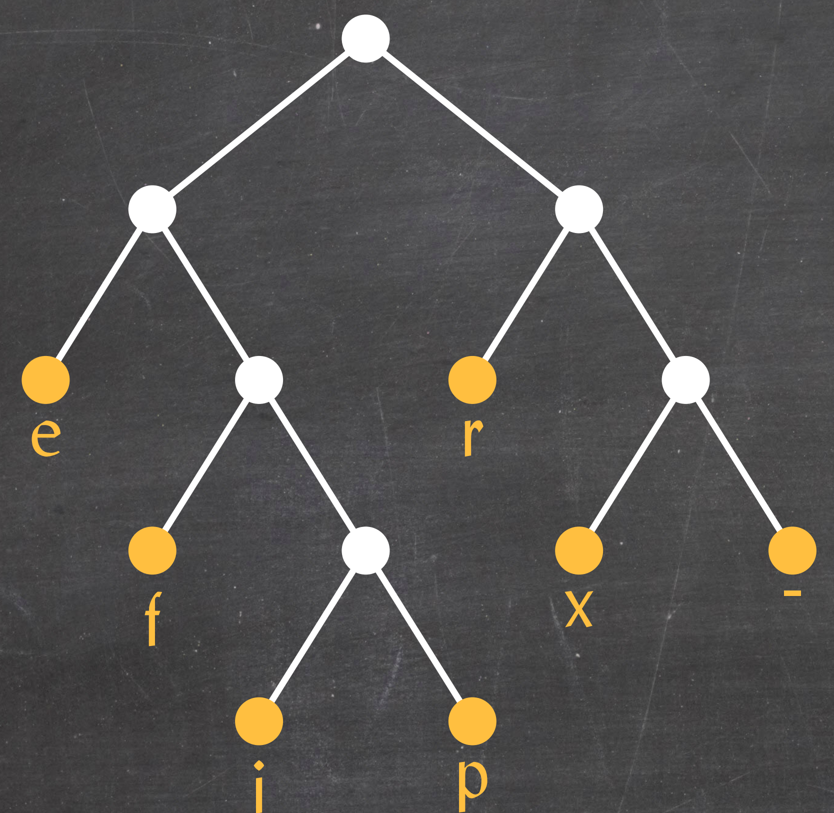


The length of the encoding of T is $|C(T)| = \sum_x f_T(x)|C(x)|$, where $f_T(x)$ is the frequency of x in T.

When making a node r a child of a new parent, we add 1 bit to the encoding C(x) of every descendant leaf x of r.

A Greedy Choice for Optimal Prefix Codes

We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.



The length of the encoding of T is $|C(T)| = \sum_x f_T(x)|C(x)|$, where $f_T(x)$ is the frequency of x in T.

When making a node r a child of a new parent, we add 1 bit to the encoding C(x) of every descendant leaf x of r.

$\Rightarrow$ By choosing the two roots with minimum total frequency of their descendent leaves, we minimize the increase in $|C(T)|$.

A Greedy Choice for Optimal Prefix Codes

We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.

"prefix-free"

x	e	f	i	p	r	x	-
$f_T(x)$	3	2	1	1	2	1	1

e (3) f (2) i (1) p (1) r (2) x (1) - (1)

The length of the encoding of T is $|C(T)| = \sum_x f_T(x)|C(x)|$, where $f_T(x)$ is the frequency of x in T.

When making a node r a child of a new parent, we add 1 bit to the encoding C(x) of every descendant leaf x of r.

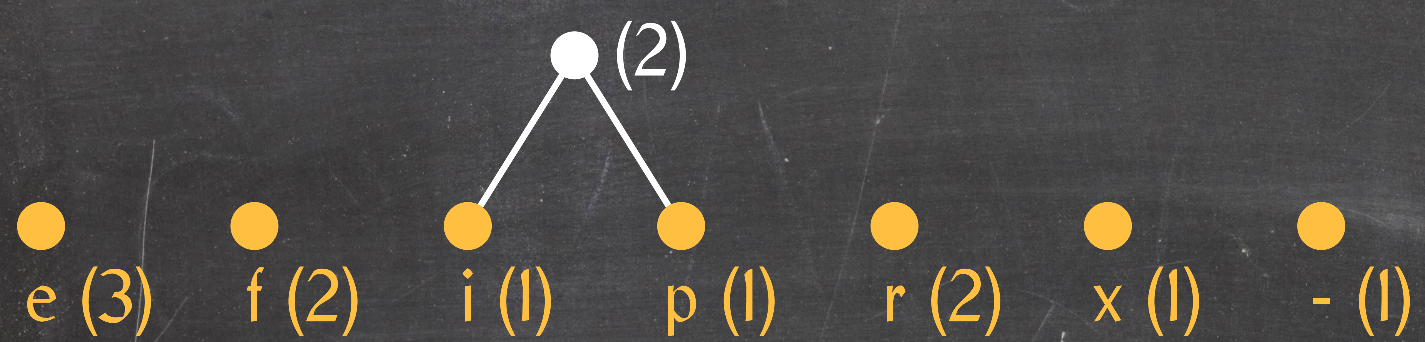
$\Rightarrow$ By choosing the two roots with minimum total frequency of their descendent leaves, we minimize the increase in $|C(T)|$.

A Greedy Choice for Optimal Prefix Codes

We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.

"prefix-free"

x	e	f	i	p	r	x	-
$f_T(x)$	3	2	1	1	2	1	1



The length of the encoding of T is $|C(T)| = \sum_x f_T(x)|C(x)|$, where $f_T(x)$ is the frequency of x in T.

When making a node r a child of a new parent, we add 1 bit to the encoding C(x) of every descendant leaf x of r.

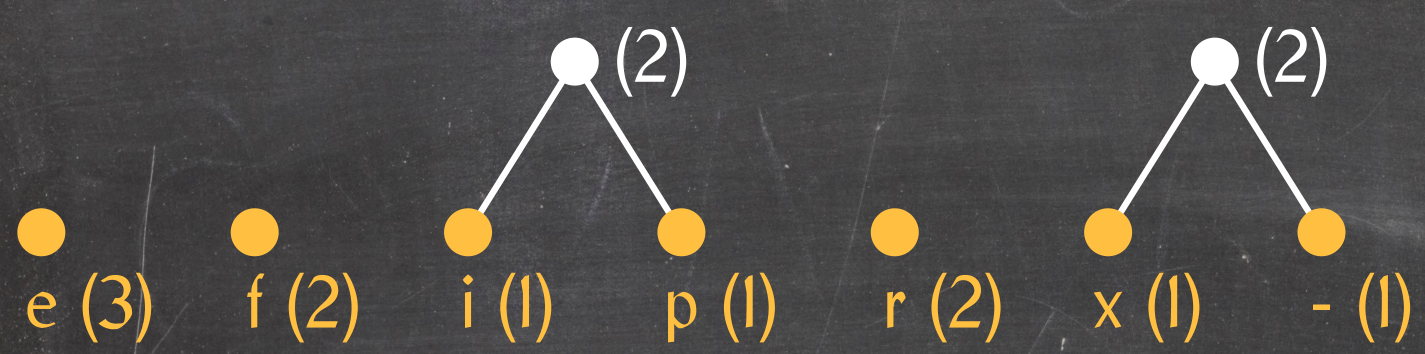
$\Rightarrow$ By choosing the two roots with minimum total frequency of their descendent leaves, we minimize the increase in $|C(T)|$.

A Greedy Choice for Optimal Prefix Codes

We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.

“prefix-free”

x	e	f	i	p	r	x	-
$f_T(x)$	3	2	1	1	2	1	1



The length of the encoding of T is $|C(T)| = \sum_x f_T(x)|C(x)|$, where $f_T(x)$ is the frequency of x in T.

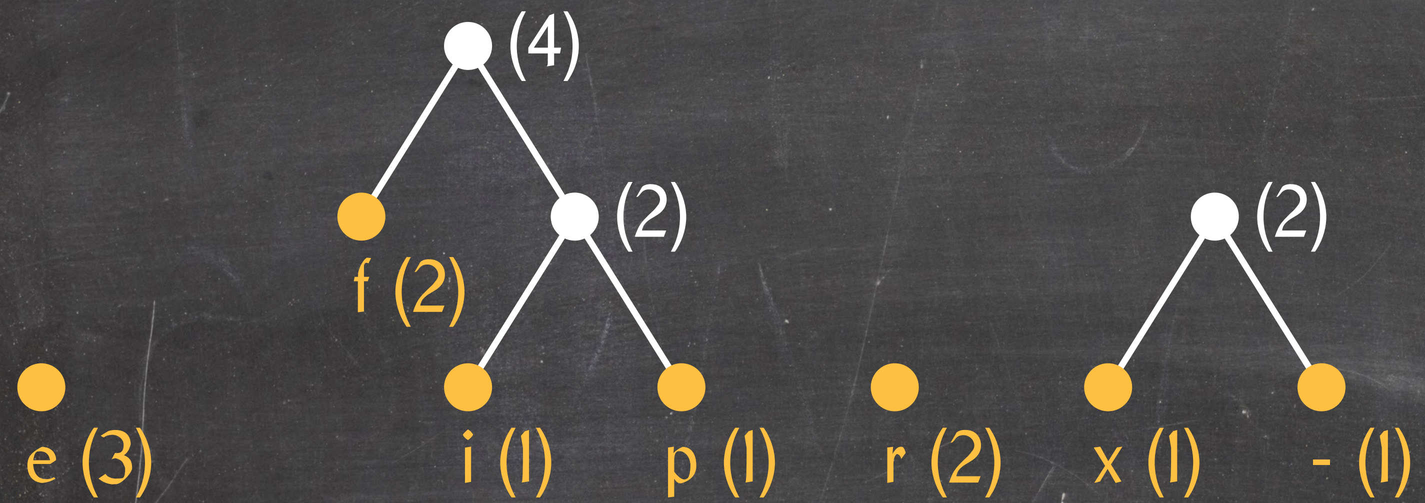
When making a node r a child of a new parent, we add 1 bit to the encoding C(x) of every descendant leaf x of r.

$\Rightarrow$ By choosing the two roots with minimum total frequency of their descendent leaves, we minimize the increase in $|C(T)|$.

A Greedy Choice for Optimal Prefix Codes

We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.

"prefix-free"



x	e	f	i	p	r	x	-
$f_T(x)$	3	2	1	1	2	1	1

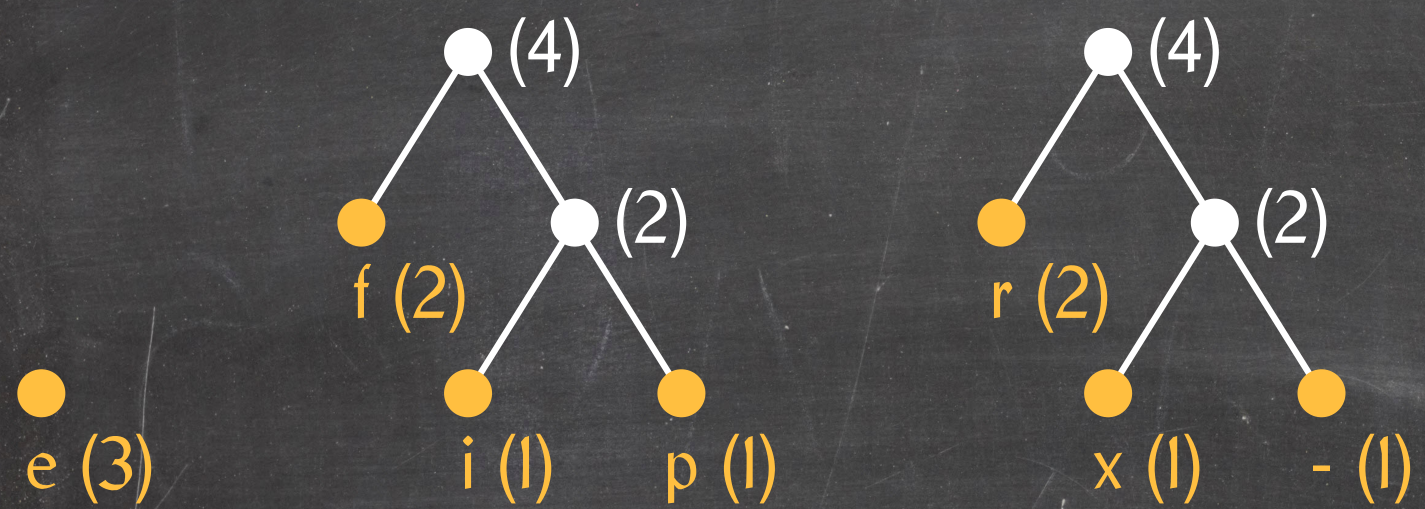
The length of the encoding of T is $|C(T)| = \sum_x f_T(x)|C(x)|$, where $f_T(x)$ is the frequency of x in T.

When making a node r a child of a new parent, we add 1 bit to the encoding C(x) of every descendant leaf x of r.

$\Rightarrow$ By choosing the two roots with minimum total frequency of their descendent leaves, we minimize the increase in $|C(T)|$.

A Greedy Choice for Optimal Prefix Codes

We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.



"prefix-free"

x	e	f	i	p	r	x	-
$f_T(x)$	3	2	1	1	2	1	1

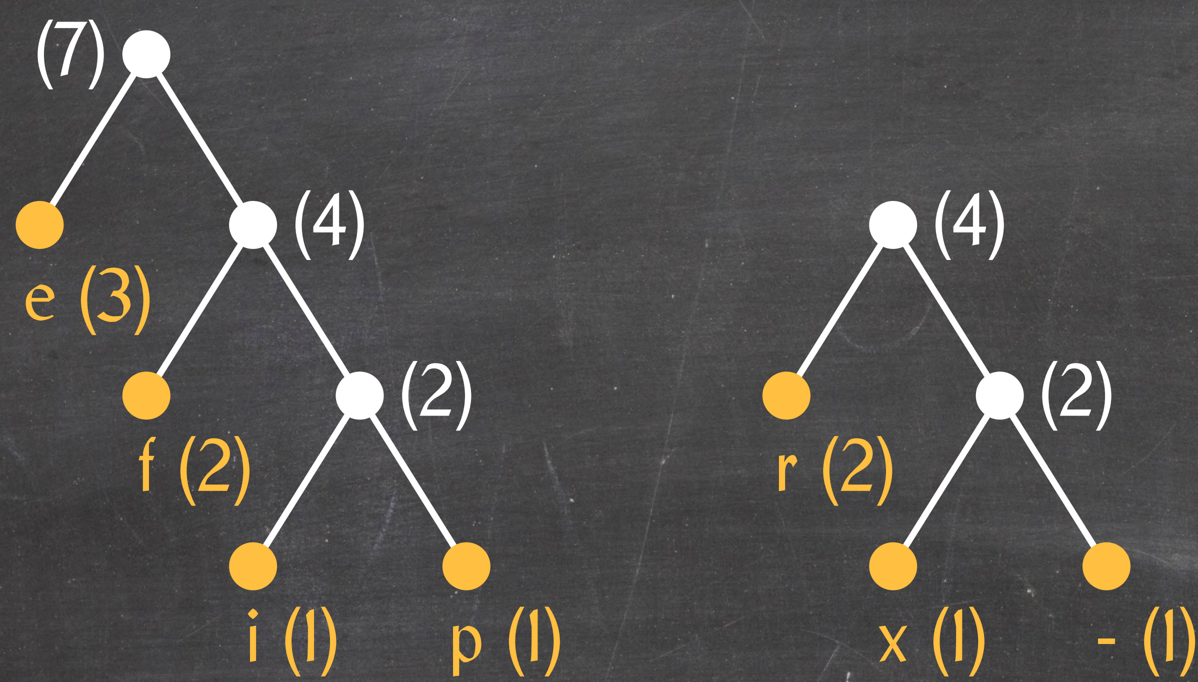
The length of the encoding of T is $|C(T)| = \sum_x f_T(x)|C(x)|$, where $f_T(x)$ is the frequency of x in T.

When making a node r a child of a new parent, we add 1 bit to the encoding C(x) of every descendant leaf x of r.

$\Rightarrow$ By choosing the two roots with minimum total frequency of their descendent leaves, we minimize the increase in $|C(T)|$.

A Greedy Choice for Optimal Prefix Codes

We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.



"prefix-free"

x	e	f	i	p	r	x	-
$f_T(x)$	3	2	1	1	2	1	1

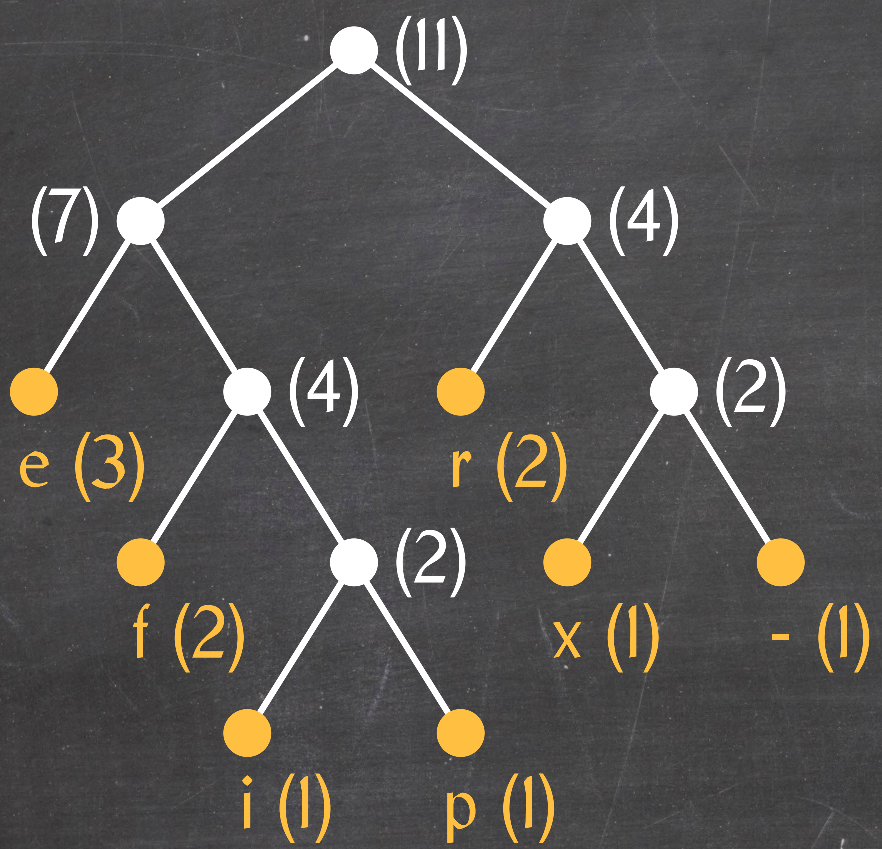
The length of the encoding of T is $|C(T)| = \sum_x f_T(x)|C(x)|$, where $f_T(x)$ is the frequency of x in T.

When making a node r a child of a new parent, we add 1 bit to the encoding C(x) of every descendant leaf x of r.

$\Rightarrow$ By choosing the two roots with minimum total frequency of their descendent leaves, we minimize the increase in $|C(T)|$.

A Greedy Choice for Optimal Prefix Codes

We can build binary trees by starting with each leaf in its own tree, joining two trees under a common parent, and repeating this until only one tree is left.



"prefix-free"

x	e	f	i	p	r	x	-
$f_T(x)$	3	2	1	1	2	1	1

The length of the encoding of T is $|C(T)| = \sum_x f_T(x)|C(x)|$, where $f_T(x)$ is the frequency of x in T.

When making a node r a child of a new parent, we add 1 bit to the encoding C(x) of every descendant leaf x of r.

$\Rightarrow$ By choosing the two roots with minimum total frequency of their descendent leaves, we minimize the increase in $|C(T)|$.

Huffman's Algorithm

Huffman(T)

```
determine the set A of characters that occur in T and their frequencies
Q = an empty priority queue
for every character x ∈ A
    do create a node v associated with x and define f(v) = f(x)
       Q.insert(v, f(v))
while |Q| > 1
    do v = Q.deleteMin()
       w = Q.deleteMin()
       u = a new node with frequency f(u) = f(v) + f(w)
       make v and w children of u
       Q.insert(u, f(u))
return Q.deleteMin()
```

Lemma: Huffman's algorithm runs in $O(m \lg n)$ time, where $m = |T|$ and n is the size of the alphabet.

Correctness of Huffman's Algorithm

Lemma: Huffman's algorithm computes an optimal prefix-free code for its input text T.

Correctness of Huffman's Algorithm

Lemma: Huffman's algorithm computes an optimal prefix-free code for its input text T.

Proof by induction on n.

Correctness of Huffman's Algorithm

Lemma: Huffman's algorithm computes an optimal prefix-free code for its input text T.

Proof by induction on n.

Base case: $n = 2$.

Correctness of Huffman's Algorithm

Lemma: Huffman's algorithm computes an optimal prefix-free code for its input text T.

Proof by induction on n.

Base case: n = 2.

We cannot do better than using one bit per character.

Correctness of Huffman's Algorithm

Lemma: Huffman's algorithm computes an optimal prefix-free code for its input text T.

Proof by induction on n.

Base case: n = 2.

We cannot do better than using one bit per character.

Inductive step: n > 2.

Correctness of Huffman's Algorithm

Lemma: Huffman's algorithm computes an optimal prefix-free code for its input text T.

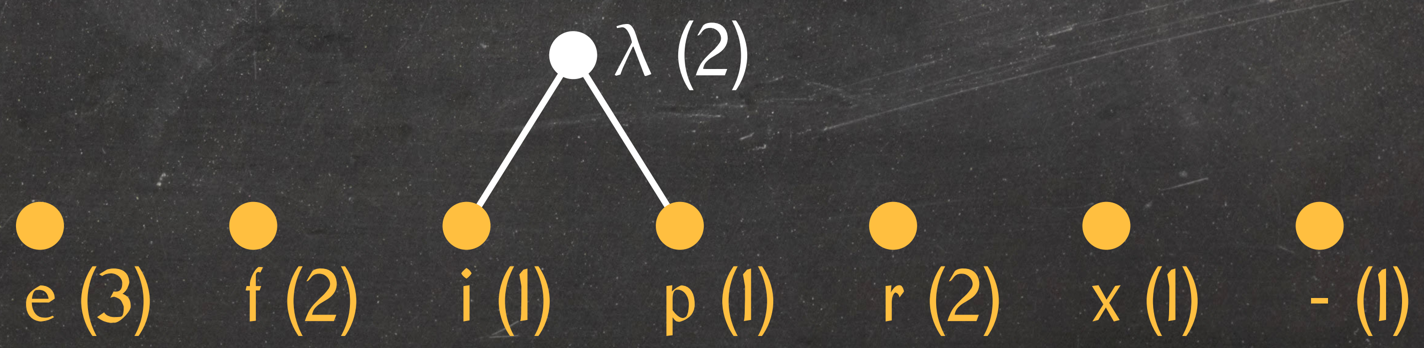
Proof by induction on n.

Base case: n = 2.

We cannot do better than using one bit per character.

Inductive step: n > 2.

Consider the first two characters a and b that are joined under a common parent z with frequency f(z) = f(a) + f(b).



Correctness of Huffman's Algorithm

Lemma: Huffman's algorithm computes an optimal prefix-free code for its input text T.

Proof by induction on n.

Base case: $n = 2$.

We cannot do better than using one bit per character.

Inductive step: $n > 2$.

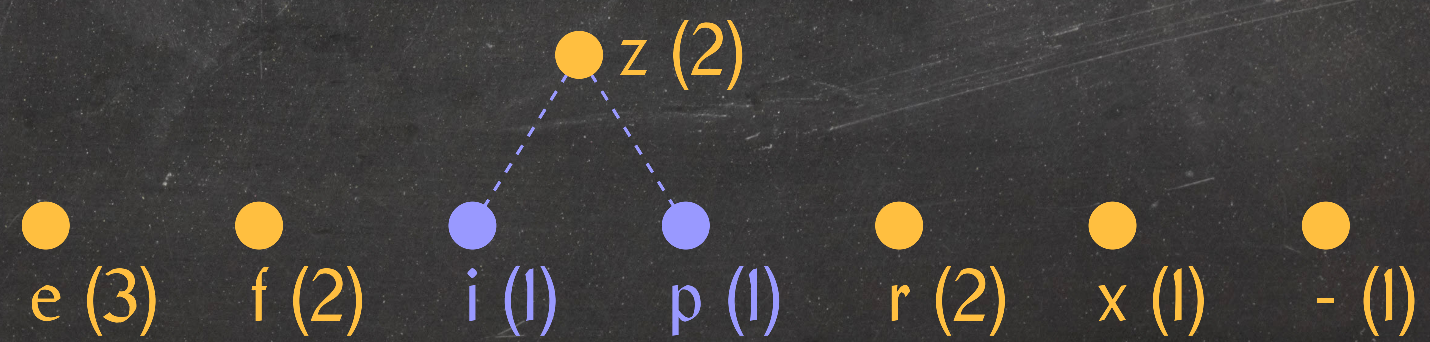
Consider the first two characters a and b that are joined under a common parent z with frequency $f(z) = f(a) + f(b)$.

Replacing a and b with z in T produces a new text T' over an alphabet of size $n - 1$ where z has frequency $f(z)$.

“prefix-free”

$\Downarrow$

“zrefzx-free”



Correctness of Huffman's Algorithm

Lemma: Huffman's algorithm computes an optimal prefix-free code for its input text T.

Proof by induction on n.

Base case: n = 2.

We cannot do better than using one bit per character.

Inductive step: n > 2.

Consider the first two characters a and b that are joined under a common parent z with frequency f(z) = f(a) + f(b).

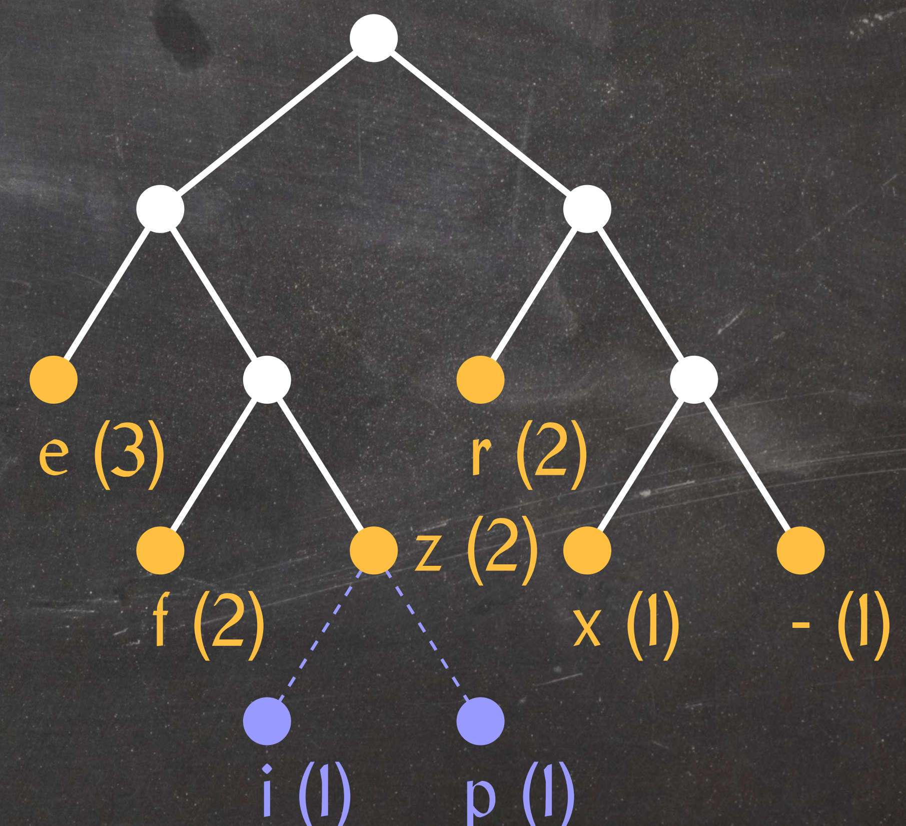
Replacing a and b with z in T produces a new text T' over an alphabet of size $n - 1$ where z has frequency f(z).

After joining a and b under z, Huffman's algorithm behaves exactly as if it was run on T'.

"prefix-free"

$\Downarrow$

"zrefzx-free"



Correctness of Huffman's Algorithm

Lemma: Huffman's algorithm computes an optimal prefix-free code for its input text T.

Proof by induction on n.

Base case: $n = 2$.

We cannot do better than using one bit per character.

Inductive step: $n > 2$.

Consider the first two characters a and b that are joined under a common parent z with frequency $f(z) = f(a) + f(b)$.

Replacing a and b with z in T produces a new text T' over an alphabet of size $n - 1$ where z has frequency $f(z)$.

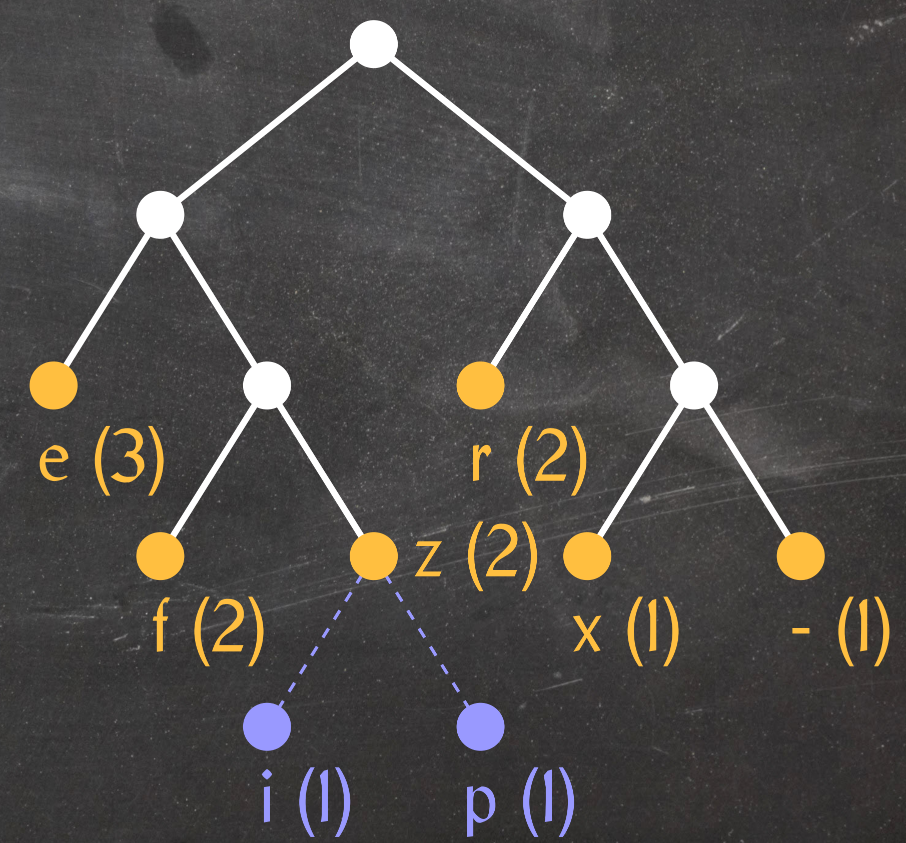
After joining a and b under z, Huffman's algorithm behaves exactly as if it was run on T'.

By induction, it produces an optimal code $C'(\cdot)$ for T'.

“prefix-free”

$\Downarrow$

“zrefzx-free”



Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

$\Rightarrow$ Huffman's algorithm produces an optimal prefix-free code for T.

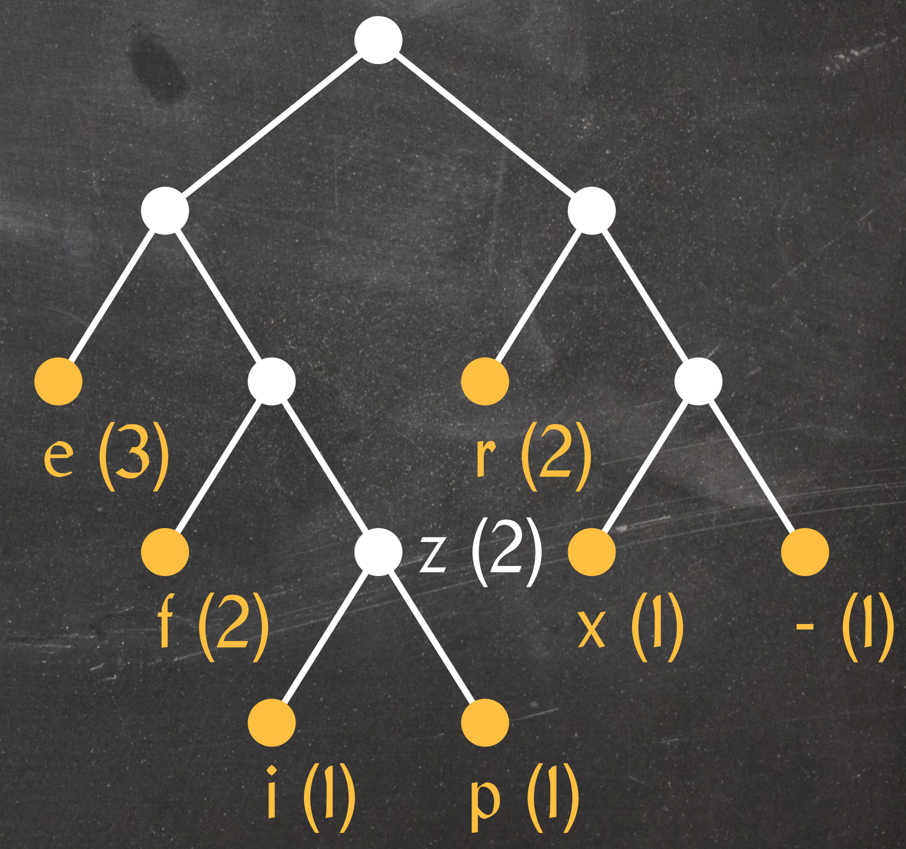
Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

$\Rightarrow$ Huffman's algorithm produces an optimal prefix-free code for T.

Assume there exists a better code $C^*(\cdot)$ such that a and b are siblings in $\mathcal{T}_{C^*}$, that is, $|C^*(T)| < |C(T)|$.

"prefix-free"



Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

$\Rightarrow$ Huffman's algorithm produces an optimal prefix-free code for T.

Assume there exists a better code $C^*(\cdot)$ such that a and b are siblings in $\mathcal{T}_{C^*}$, that is, $|C^*(T)| < |C(T)|$.

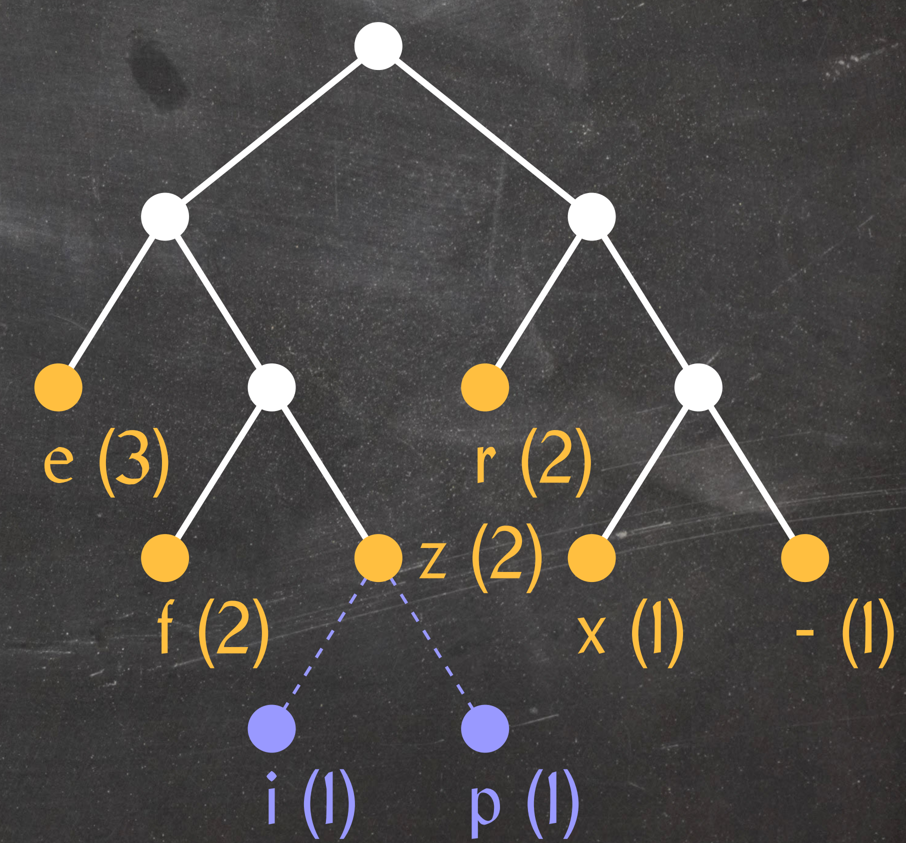
Let $C''(\cdot)$ be the code for T' defined as

$$C''(x) = \begin{cases} C^*(x) & x \neq z \\ \sigma & x = z \text{ and } C^*(a) = \sigma 0 \end{cases}$$

"prefix-free"

$\Downarrow$

"zrefzx-free"



Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

$\Rightarrow$ Huffman's algorithm produces an optimal prefix-free code for T.

Assume there exists a better code $C^*(\cdot)$ such that a and b are siblings in $\mathcal{T}_{C^*}$, that is, $|C^*(T)| < |C(T)|$.

Let $C''(\cdot)$ be the code for T' defined as

$$C''(x) = \begin{cases} C^*(x) & x \neq z \\ \sigma & x = z \text{ and } C^*(a) = \sigma 0 \end{cases}$$

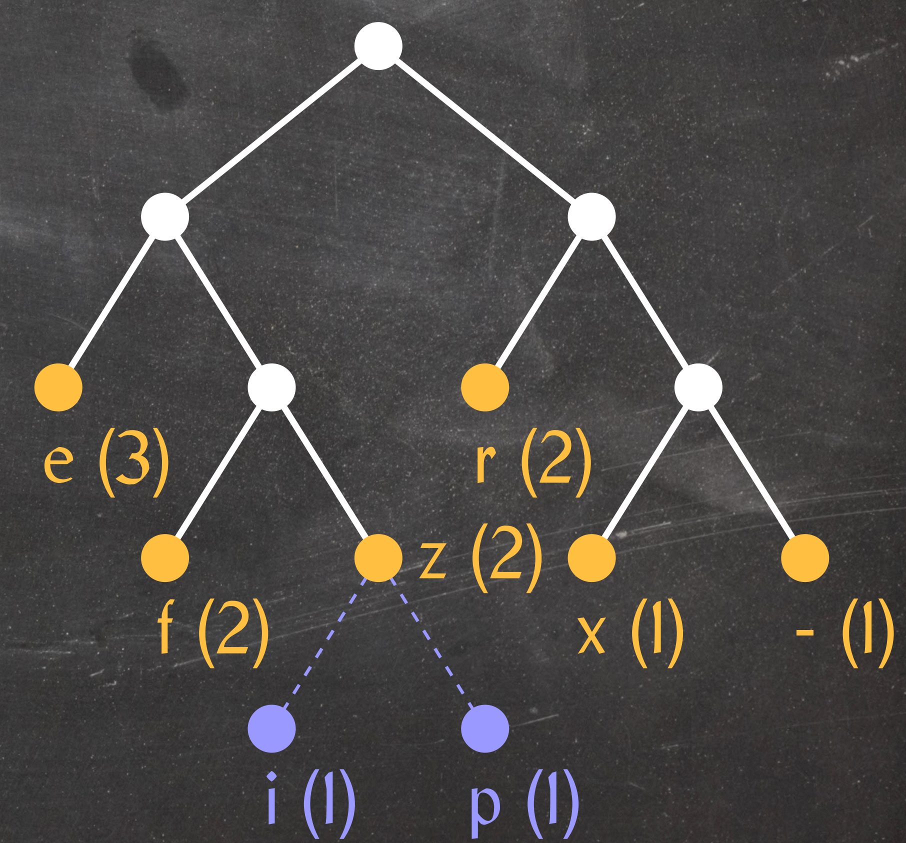
We also have

$$C'(x) = \begin{cases} C(x) & x \neq z \\ \sigma & x = z \text{ and } C(a) = \sigma 0 \end{cases}$$

"prefix-free"

$\Downarrow$

"zrefzx-free"



Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

$\Rightarrow$ Huffman's algorithm produces an optimal prefix-free code for T.

Assume there exists a better code $C^*(\cdot)$ such that a and b are siblings in $\mathcal{T}_{C^*}$, that is, $|C^*(T)| < |C(T)|$.

Let $C''(\cdot)$ be the code for T' defined as

$$C''(x) = \begin{cases} C^*(x) & x \neq z \\ \sigma & x = z \text{ and } C^*(a) = \sigma 0 \end{cases}$$

We also have

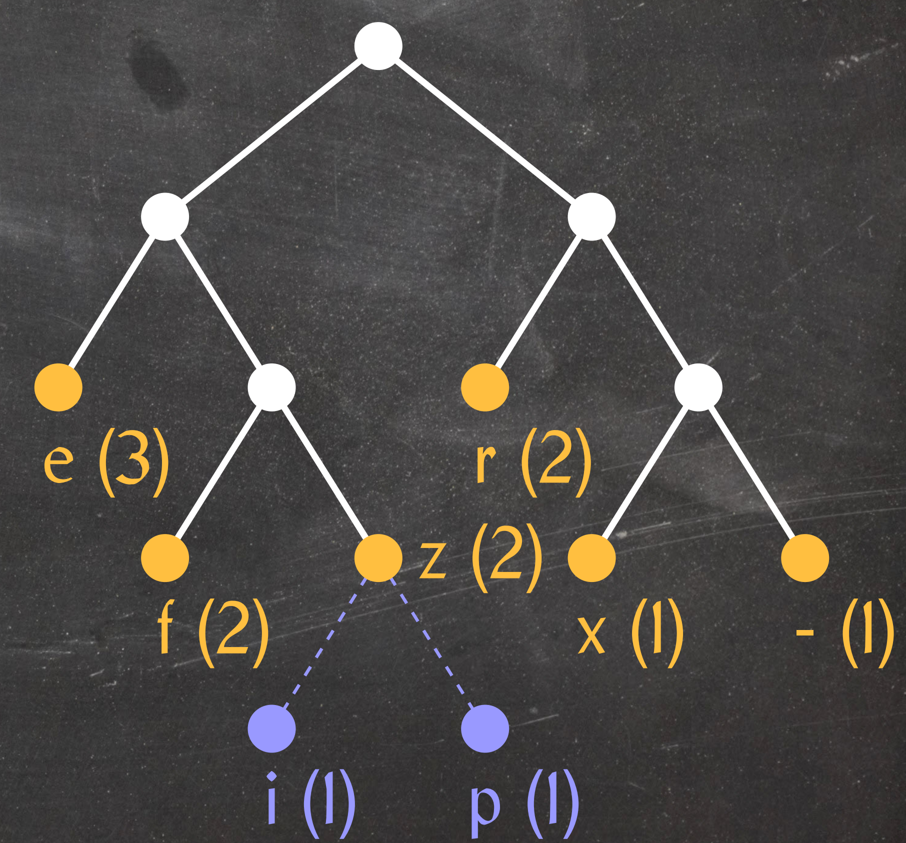
$$C'(x) = \begin{cases} C(x) & x \neq z \\ \sigma & x = z \text{ and } C(a) = \sigma 0 \end{cases}$$

$|C(T)| = |C'(T')| + f(z)$ and $|C^*(T)| = |C''(T')| + f(z)$.

"prefix-free"

$\Downarrow$

"zrefzx-free"



Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

$\Rightarrow$ Huffman's algorithm produces an optimal prefix-free code for T.

Assume there exists a better code $C^*(\cdot)$ such that a and b are siblings in $\mathcal{T}_{C^*}$, that is, $|C^*(T)| < |C(T)|$.

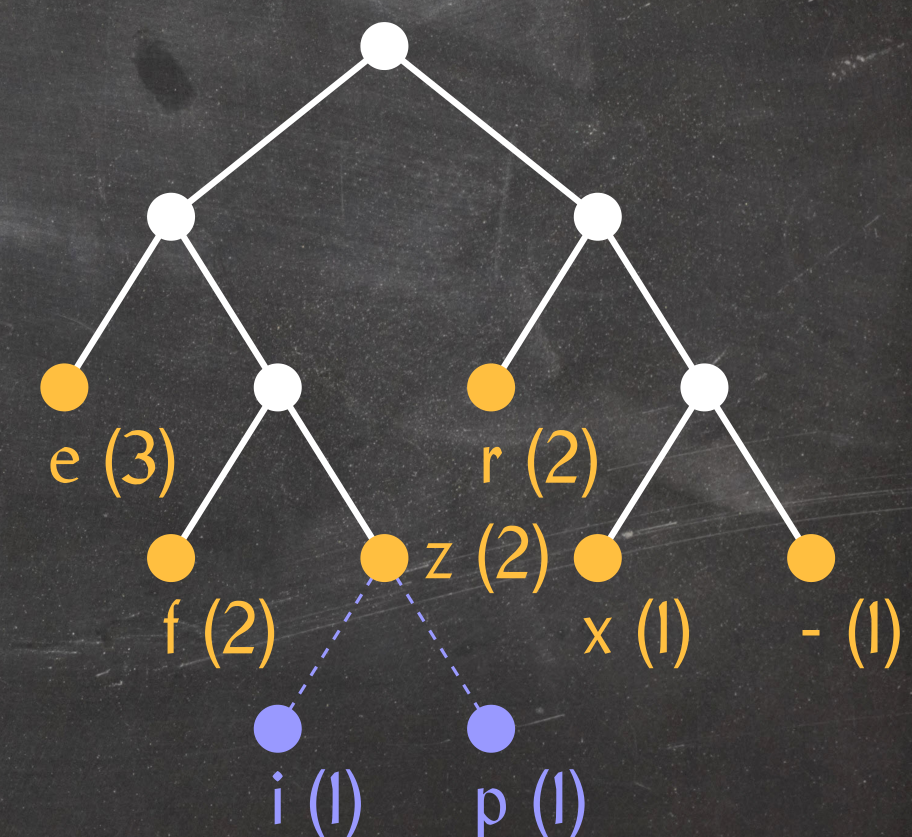
"prefix-free"

$\Downarrow$

"zrefzx-free"

Let $C''(\cdot)$ be the code for T' defined as

$$C''(x) = \begin{cases} C^*(x) & x \neq z \\ \sigma & x = z \text{ and } C^*(a) = \sigma 0 \end{cases}$$



We also have

$$C'(x) = \begin{cases} C(x) & x \neq z \\ \sigma & x = z \text{ and } C(a) = \sigma 0 \end{cases}$$

$|C(T)| = |C'(T')| + f(z)$ and $|C^*(T)| = |C''(T')| + f(z)$.

$\Rightarrow$ $|C''(T')| < |C'(T')|$, a contradiction because $C'(\cdot)$ is optimal for T'.

Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

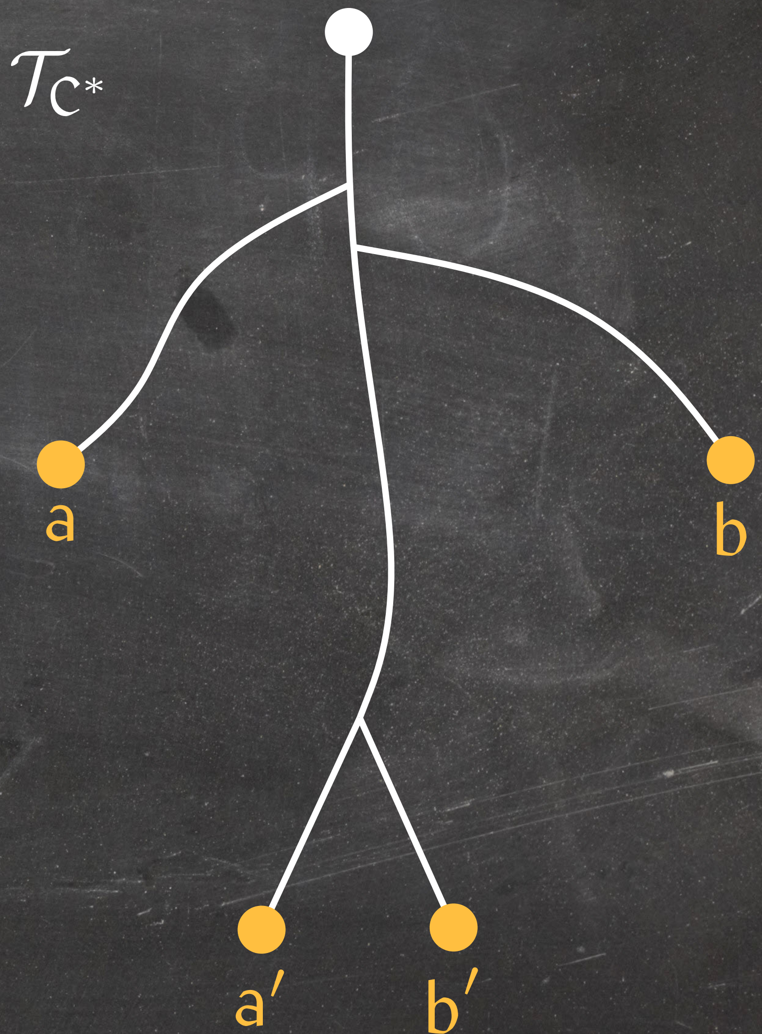
Let $C^*(\cdot)$ be an optimal code for T.

Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

Let $C^*(\cdot)$ be an optimal code for T.

The sibling b' of the deepest leaf a' in $\mathcal{T}_{C^*}$ is also a leaf.



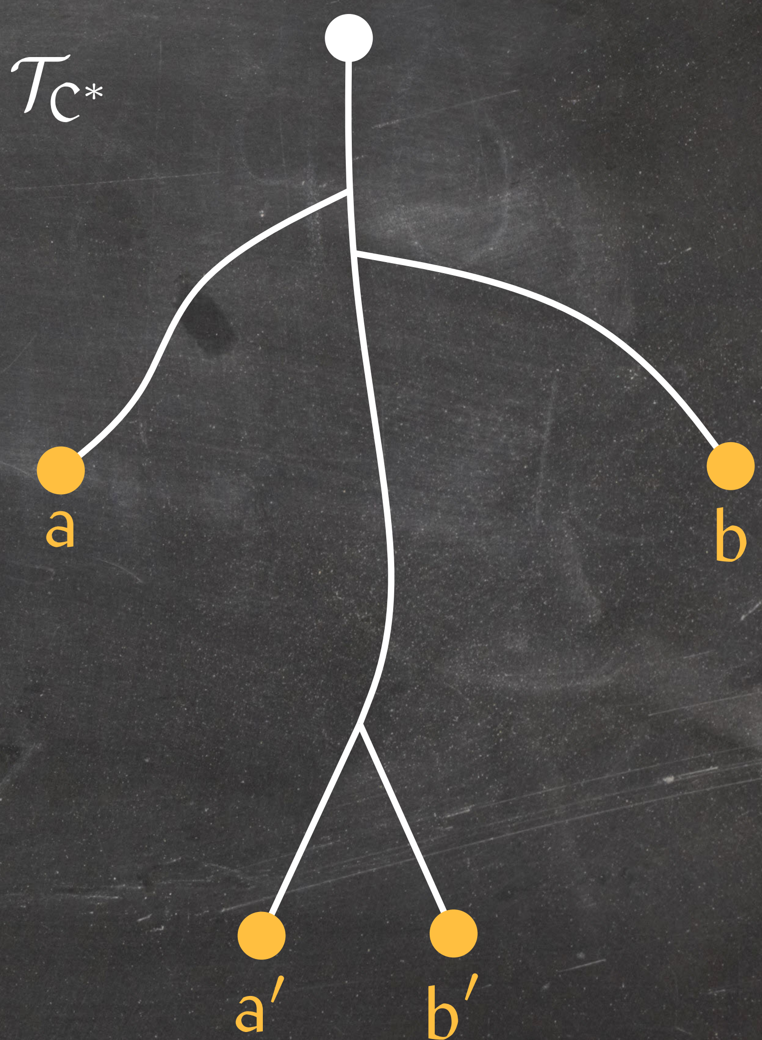
Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

Let $C^*(\cdot)$ be an optimal code for T.

The sibling b' of the deepest leaf a' in $\mathcal{T}_{C^*}$ is also a leaf.

We have $|C^*(a)| \le |C^*(a')|$ and $|C^*(b)| \le |C^*(b')|$.



Correctness of Huffman's Algorithm

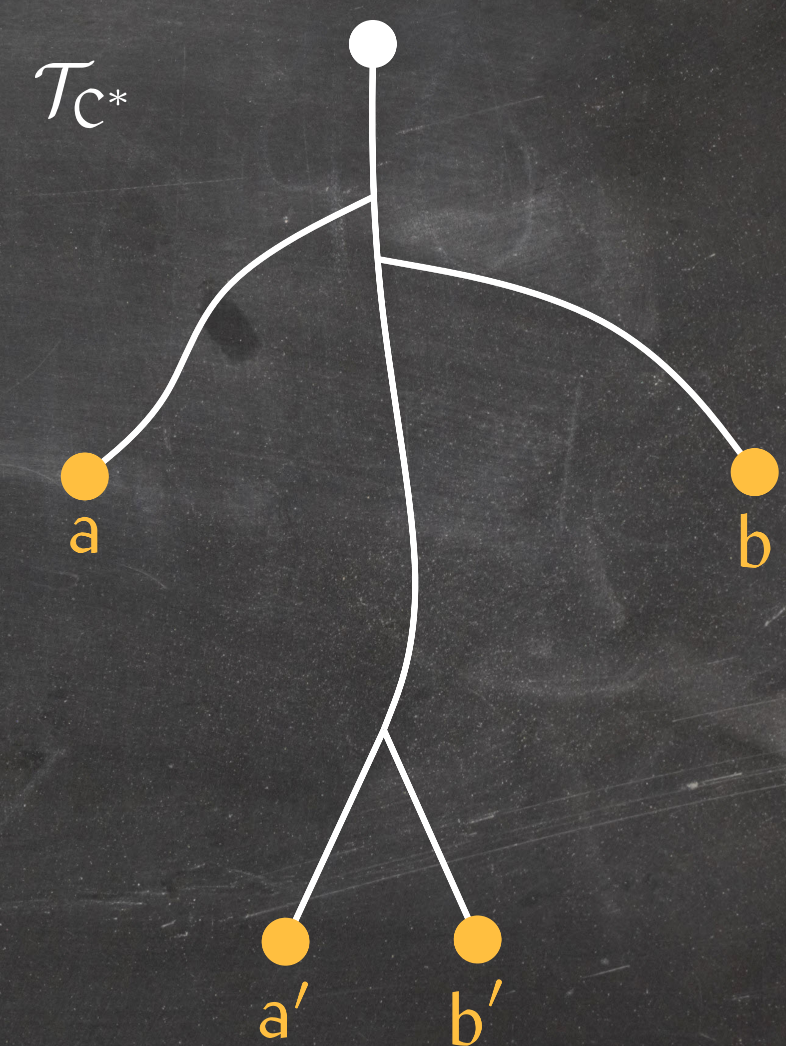
Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

Let $C^*(\cdot)$ be an optimal code for T.

The sibling b' of the deepest leaf a' in $\mathcal{T}_{C^*}$ is also a leaf.

We have $|C^*(a)| \le |C^*(a')|$ and $|C^*(b)| \le |C^*(b')|$.

Now assume $f(a) \le f(b)$ and $f(a') \le f(b')$.



Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

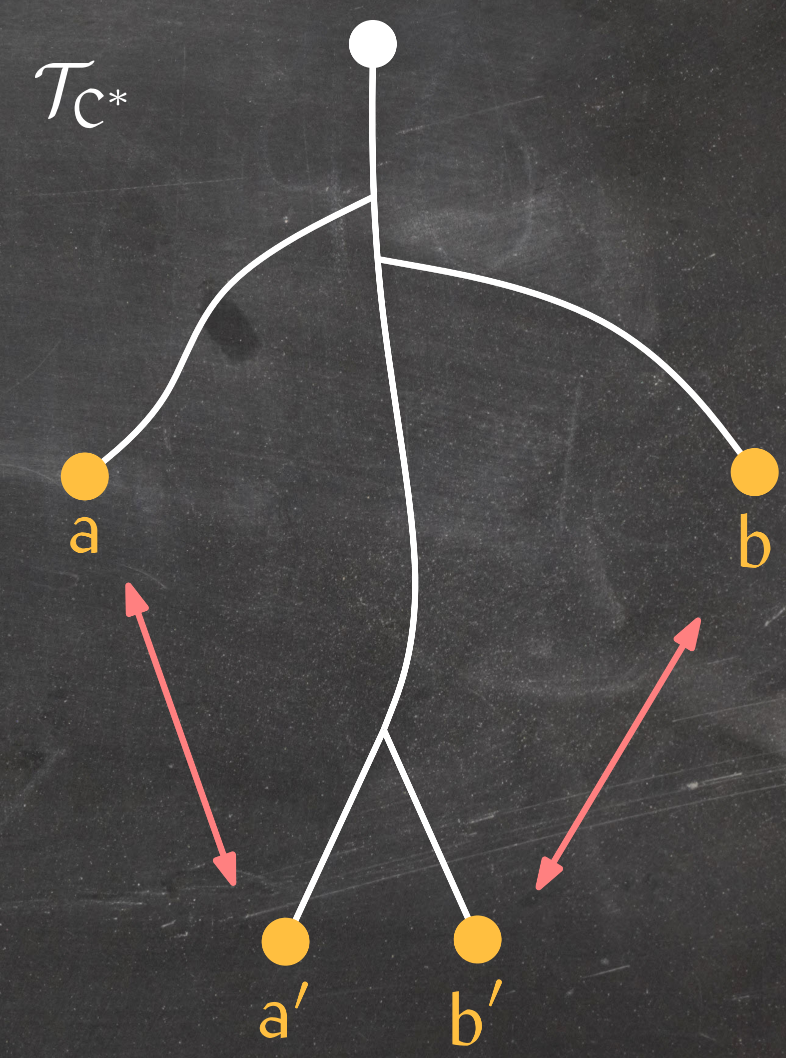
Let $C^*(\cdot)$ be an optimal code for T.

The sibling b' of the deepest leaf a' in $\mathcal{T}_{C^*}$ is also a leaf.

We have $|C^*(a)| \le |C^*(a')|$ and $|C^*(b)| \le |C^*(b')|$.

Now assume $f(a) \le f(b)$ and $f(a') \le f(b')$.

Let $C(\cdot)$ be the code such that $\mathcal{T}_C$ is obtained from $\mathcal{T}_{C^*}$ by swapping a and a', and b and b'.



Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

Let $C^*(\cdot)$ be an optimal code for T.

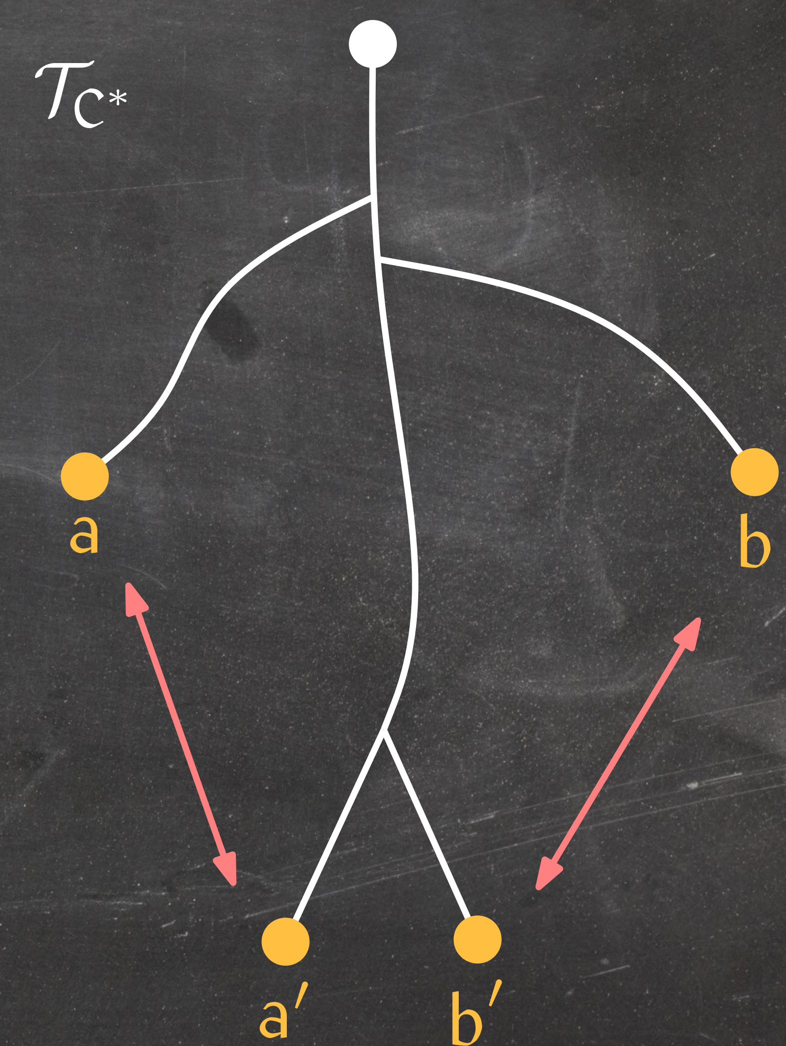
The sibling b' of the deepest leaf a' in $\mathcal{T}_{C^*}$ is also a leaf.

We have $|C^*(a)| \le |C^*(a')|$ and $|C^*(b)| \le |C^*(b')|$.

Now assume $f(a) \le f(b)$ and $f(a') \le f(b')$.

Let $C(\cdot)$ be the code such that $\mathcal{T}_C$ is obtained from $\mathcal{T}_{C^*}$ by swapping a and a', and b and b'.

We prove that $|C(T)| \le |C^*(T)|$, that is, $C(\cdot)$ is an optimal prefix-free code for T.



Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

Let $C^*(\cdot)$ be an optimal code for T.

The sibling b' of the deepest leaf a' in $\mathcal{T}_{C^*}$ is also a leaf.

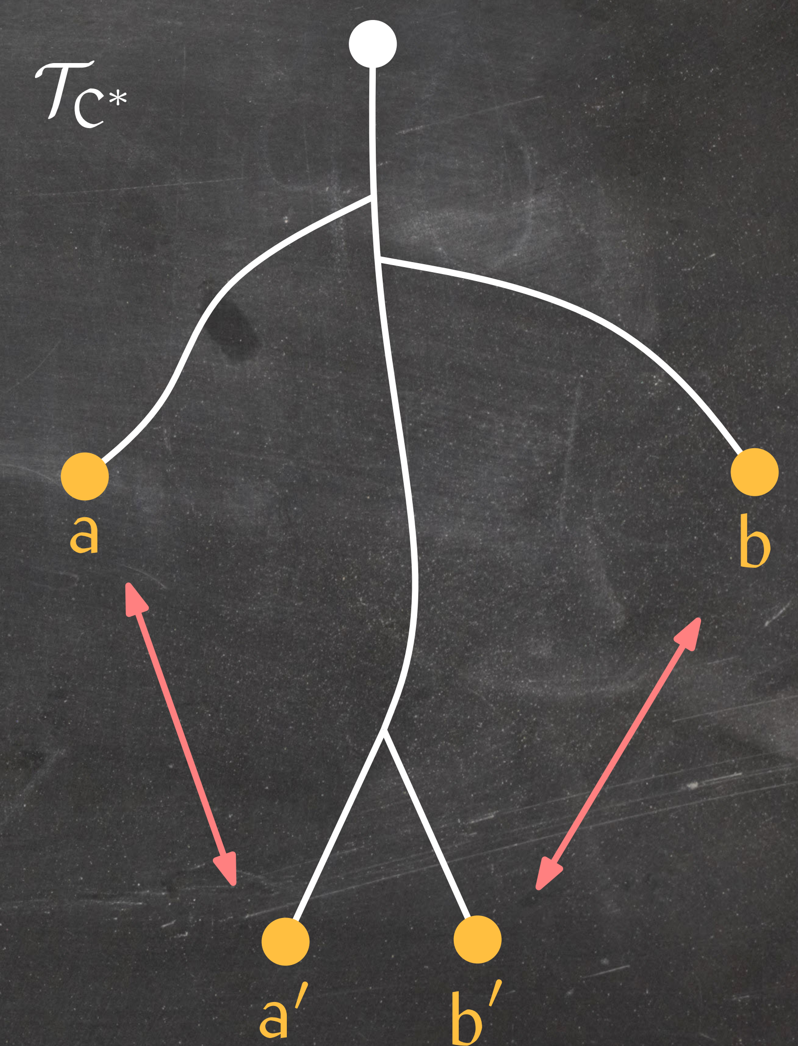
We have $|C^*(a)| \le |C^*(a')|$ and $|C^*(b)| \le |C^*(b')|$.

Now assume $f(a) \le f(b)$ and $f(a') \le f(b')$.

Let $C(\cdot)$ be the code such that $\mathcal{T}_C$ is obtained from $\mathcal{T}_{C^*}$ by swapping a and a', and b and b'.

We prove that $|C(T)| \le |C^*(T)|$, that is, $C(\cdot)$ is an optimal prefix-free code for T.

Since a and b are siblings in $\mathcal{T}_C$, this proves the claim.



Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

Given: $|C^*(a)| \le |C^*(a')|$, $|C^*(b)| \le |C^*(b')|$, $f(a) \le f(b)$, and $f(a') \le f(b')$.

Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

Given: $|C^*(a)| \le |C^*(a')|$, $|C^*(b)| \le |C^*(b')|$, $f(a) \le f(b)$, and $f(a') \le f(b')$.

$\Rightarrow$ $f(a) \le f(a')$ and $f(b) \le f(b')$.

Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

Given: $|C^*(a)| \le |C^*(a')|$, $|C^*(b)| \le |C^*(b')|$, $f(a) \le f(b)$, and $f(a') \le f(b')$.

$\Rightarrow$ $f(a) \le f(a')$ and $f(b) \le f(b')$.

$$\begin{aligned} |C(T)| - |C^*(T)| = {} & f(a)|C(a)| + f(b)|C(b)| + f(a')|C(a')| + f(b')|C(b')| - \\ & f(a)|C^*(a)| - f(b)|C^*(b)| - f(a')|C^*(a')| - f(b')|C^*(b')| \end{aligned}$$

Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

Given: $|C^*(a)| \le |C^*(a')|$, $|C^*(b)| \le |C^*(b')|$, $f(a) \le f(b)$, and $f(a') \le f(b')$.

$\Rightarrow$ $f(a) \le f(a')$ and $f(b) \le f(b')$.

$$\begin{aligned}
|C(T)| - |C^*(T)| &= f(a)|C(a)| + f(b)|C(b)| + f(a')|C(a')| + f(b')|C(b')| - \\
&\quad f(a)|C^*(a)| - f(b)|C^*(b)| - f(a')|C^*(a')| - f(b')|C^*(b')| \\
&= f(a)|C^*(a')| + f(b)|C^*(b')| + f(a')|C^*(a)| + f(b')|C^*(b)| - \\
&\quad f(a)|C^*(a)| - f(b)|C^*(b)| - f(a')|C^*(a')| - f(b')|C^*(b')|
\end{aligned}$$

Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

Given: $|C^*(a)| \le |C^*(a')|$, $|C^*(b)| \le |C^*(b')|$, $f(a) \le f(b)$, and $f(a') \le f(b')$.

$\Rightarrow$ $f(a) \le f(a')$ and $f(b) \le f(b')$.

$$\begin{aligned}
|C(T)| - |C^*(T)| &= f(a)|C(a)| + f(b)|C(b)| + f(a')|C(a')| + f(b')|C(b')| - \\
&\quad\; f(a)|C^*(a)| - f(b)|C^*(b)| - f(a')|C^*(a')| - f(b')|C^*(b')| \\
&= f(a)|C^*(a')| + f(b)|C^*(b')| + f(a')|C^*(a)| + f(b')|C^*(b)| - \\
&\quad\; f(a)|C^*(a)| - f(b)|C^*(b)| - f(a')|C^*(a')| - f(b')|C^*(b')| \\
&= (f(a) - f(a'))\,(|C^*(a')| - |C^*(a)|) + (f(b) - f(b'))\,(|C^*(b')| - |C^*(b)|)
\end{aligned}$$

Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

Given: $|C^*(a)| \le |C^*(a')|$, $|C^*(b)| \le |C^*(b')|$, $f(a) \le f(b)$, and $f(a') \le f(b')$.

$\Rightarrow$ $f(a) \le f(a')$ and $f(b) \le f(b')$.

$$
\begin{aligned}
|C(T)| - |C^*(T)| &= f(a)|C(a)| + f(b)|C(b)| + f(a')|C(a')| + f(b')|C(b')| - \\
&\quad\; f(a)|C^*(a)| - f(b)|C^*(b)| - f(a')|C^*(a')| - f(b')|C^*(b')| \\
&= f(a)|C^*(a')| + f(b)|C^*(b')| + f(a')|C^*(a)| + f(b')|C^*(b)| - \\
&\quad\; f(a)|C^*(a)| - f(b)|C^*(b)| - f(a')|C^*(a')| - f(b')|C^*(b')| \\
&= \underbrace{(f(a) - f(a'))}_{\le 0}\underbrace{(|C^*(a')| - |C^*(a)|)}_{\ge 0} + \underbrace{(f(b) - f(b'))}_{\le 0}\underbrace{(|C^*(b')| - |C^*(b)|)}_{\ge 0}
\end{aligned}
$$

Correctness of Huffman's Algorithm

Claim: There exists an optimal prefix-free code $C(\cdot)$ for T such that the two least frequent characters a and b in T are siblings in $\mathcal{T}_C$.

Given: $|C^*(a)| \le |C^*(a')|$, $|C^*(b)| \le |C^*(b')|$, $f(a) \le f(b)$, and $f(a') \le f(b')$.

$\Rightarrow$ $f(a) \le f(a')$ and $f(b) \le f(b')$.

$$\begin{aligned}
|C(T)| - |C^*(T)| &= f(a)|C(a)| + f(b)|C(b)| + f(a')|C(a')| + f(b')|C(b')| - \\
&\quad\ f(a)|C^*(a)| - f(b)|C^*(b)| - f(a')|C^*(a')| - f(b')|C^*(b')| \\
&= f(a)|C^*(a')| + f(b)|C^*(b')| + f(a')|C^*(a)| + f(b')|C^*(b)| - \\
&\quad\ f(a)|C^*(a)| - f(b)|C^*(b)| - f(a')|C^*(a')| - f(b')|C^*(b')| \\
&= \underbrace{(f(a) - f(a'))}_{\le 0}\underbrace{(|C^*(a')| - |C^*(a)|)}_{\ge 0} + \underbrace{(f(b) - f(b'))}_{\le 0}\underbrace{(|C^*(b')| - |C^*(b)|)}_{\ge 0} \\
&\le 0
\end{aligned}$$

Summary

Greedy algorithms make natural local choices in their search for a globally optimal solution.

Many good heuristics are greedy:

- Simple
- Work well in practice

Proof that a greedy algorithm finds an optimal solution:

- Induction
- Exchange argument

Useful data structures:

- Union-find data structure
- Thin Heap

Analysis of a sequence of data structure operations:

- Amortized analysis
- Potential functions