IV. Approximation Algorithms: Covering Problems

Thomas Sauerwald

Easter 2016

UNIVERSITY OF CAMBRIDGE

Outline

Introduction

Vertex Cover

Motivation

Many fundamental problems are **NP-complete**, yet they are too important to be abandoned.

Motivation

Many fundamental problems are **NP-complete**, yet they are too important to be abandoned.

Examples: HAMILTON, 3-SAT, VERTEX-COVER, KNAPSACK,...

Motivation

Many fundamental problems are **NP-complete**, yet they are too important to be abandoned.

Examples: HAMILTON, 3-SAT, VERTEX-COVER, KNAPSACK,...

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Motivation

Many fundamental problems are **NP-complete**, yet they are too important to be abandoned.

Examples: HAMILTON, 3-SAT, VERTEX-COVER, KNAPSACK,...

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Motivation

Many fundamental problems are **NP-complete**, yet they are too important to be abandoned.

Examples: HAMILTON, 3-SAT, VERTEX-COVER, KNAPSACK,...

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

We will call these **approximation algorithms**.

Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size n, the cost C of the returned solution and optimal cost C^* satisfy:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size n, the cost C of the returned solution and optimal cost C^* satisfy:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

This covers both maximization and minimization problems.

Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size n, the cost C of the returned solution and optimal cost C^* satisfy:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

- Maximization problem: $\frac{C^*}{C} \geq 1$

This covers both maximization and minimization problems.

Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size n, the cost C of the returned solution and optimal cost C^* satisfy:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

- Maximization problem: $\frac{C^*}{C} \geq 1$
- Minimization problem: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size n, the cost C of the returned solution and optimal cost C^* satisfy:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

- Maximization problem: $\frac{C^*}{C} \geq 1$
- Minimization problem: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size n, the cost C of the returned solution and optimal cost C^* satisfy:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

- Maximization problem: $\frac{C^*}{C} \geq 1$
- Minimization problem: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

Approximation Schemes

Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size n, the cost C of the returned solution and optimal cost C^* satisfy:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

- Maximization problem: $\frac{C^*}{C} \geq 1$
- Minimization problem: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

Approximation Schemes

An approximation scheme is an approximation algorithm, which given any input and $\epsilon > 0$, is a $(1+\epsilon)$-approximation algorithm.

Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size n, the cost C of the returned solution and optimal cost C^* satisfy:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

- Maximization problem: $\frac{C^*}{C} \geq 1$
- Minimization problem: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

Approximation Schemes

An approximation scheme is an approximation algorithm, which given any input and $\epsilon > 0$, is a $(1+\epsilon)$-approximation algorithm.

- It is a polynomial-time approximation scheme (PTAS) if for any fixed $\epsilon > 0$, the runtime is polynomial in n.

Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size n, the cost C of the returned solution and optimal cost C^* satisfy:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

- Maximization problem: $\frac{C^*}{C} \geq 1$
- Minimization problem: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

Approximation Schemes

An approximation scheme is an approximation algorithm, which given any input and $\epsilon > 0$, is a $(1+\epsilon)$-approximation algorithm.

- It is a polynomial-time approximation scheme (PTAS) if for any fixed $\epsilon > 0$, the runtime is polynomial in n. For example, $O(n^{2/\epsilon})$.

Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size n, the cost C of the returned solution and optimal cost C^* satisfy:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

- Maximization problem: $\frac{C^*}{C} \geq 1$
- Minimization problem: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

Approximation Schemes

An approximation scheme is an approximation algorithm, which given any input and $\epsilon > 0$, is a $(1+\epsilon)$-approximation algorithm.

- It is a polynomial-time approximation scheme (PTAS) if for any fixed $\epsilon > 0$, the runtime is polynomial in n. For example, $O(n^{2/\epsilon})$.
- It is a fully polynomial-time approximation scheme (FPTAS) if the runtime is polynomial in both $1/\epsilon$ and n.

Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size n, the cost C of the returned solution and optimal cost C^* satisfy:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

- Maximization problem: $\frac{C^*}{C} \geq 1$
- Minimization problem: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

Approximation Schemes

An approximation scheme is an approximation algorithm, which given any input and $\epsilon > 0$, is a $(1+\epsilon)$-approximation algorithm.

- It is a polynomial-time approximation scheme (PTAS) if for any fixed $\epsilon > 0$, the runtime is polynomial in n. For example, $O(n^{2/\epsilon})$.
- It is a fully polynomial-time approximation scheme (FPTAS) if the runtime is polynomial in both $1/\epsilon$ and n. For example, $O((1/\epsilon)^2 \cdot n^3)$.

Outline

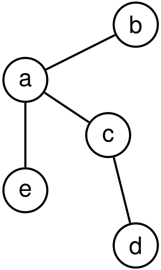
Introduction

Vertex Cover

The Vertex-Cover Problem

Vertex Cover Problem

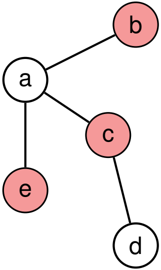
- Given: Undirected graph $G = (V, E)$
- Goal: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.



The Vertex-Cover Problem

Vertex Cover Problem

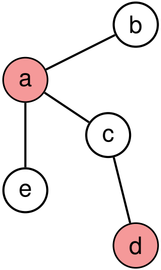
- Given: Undirected graph $G = (V, E)$
- Goal: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.



The Vertex-Cover Problem

Vertex Cover Problem

- Given: Undirected graph $G = (V, E)$
- Goal: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

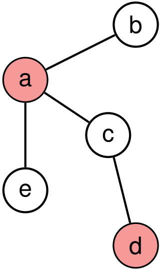


The Vertex-Cover Problem

We are covering edges by picking vertices!

Vertex Cover Problem

- Given: Undirected graph $G = (V, E)$
- Goal: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.



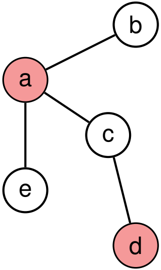
The Vertex-Cover Problem

We are covering edges by picking vertices!

Vertex Cover Problem

- Given: Undirected graph $G = (V, E)$
- Goal: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is an NP-hard problem.



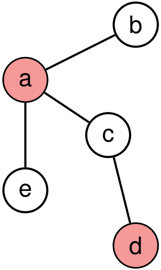
The Vertex-Cover Problem

We are covering edges by picking vertices!

Vertex Cover Problem

- Given: Undirected graph $G = (V, E)$
- Goal: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is an NP-hard problem.



Applications:

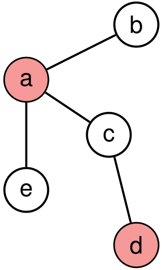
The Vertex-Cover Problem

We are covering edges by picking vertices!

Vertex Cover Problem

- Given: Undirected graph $G = (V, E)$
- Goal: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is an NP-hard problem.



Applications:

- Every edge forms a task, and every vertex represents a person/machine which can execute that task

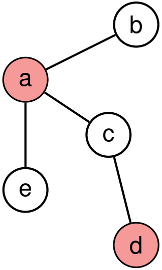
The Vertex-Cover Problem

We are covering edges by picking vertices!

Vertex Cover Problem

- Given: Undirected graph $G = (V, E)$
- Goal: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is an NP-hard problem.



Applications:

- Every edge forms a task, and every vertex represents a person/machine which can execute that task
- Perform all tasks with the minimal amount of resources

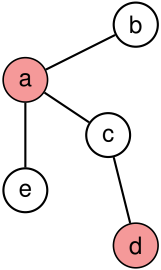
The Vertex-Cover Problem

We are covering edges by picking vertices!

Vertex Cover Problem

- Given: Undirected graph $G = (V, E)$
- Goal: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is an NP-hard problem.



Applications:

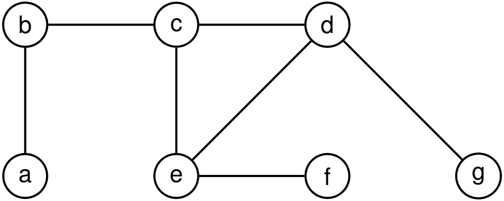
- Every edge forms a task, and every vertex represents a person/machine which can execute that task
- Perform all tasks with the minimal amount of resources
- Extensions: weighted vertices or hypergraphs ($\rightsquigarrow$ Set-Covering Problem)

An Approximation Algorithm based on Greedy

```
APPROX-VERTEX-COVER(G)
C = ∅
E' = G.E
while E' ≠ ∅
    let (u, v) be an arbitrary edge of E'
    C = C ∪ {u, v}
    remove from E' every edge incident on either u or v
return C
```

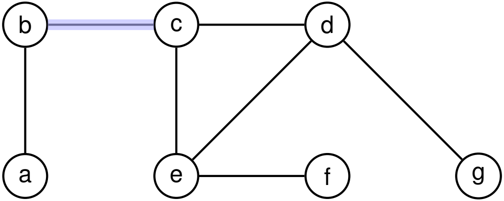
An Approximation Algorithm based on Greedy

```
APPROX-VERTEX-COVER(G)
C = ∅
E' = G.E
while E' ≠ ∅
    let (u, v) be an arbitrary edge of E'
    C = C ∪ {u, v}
    remove from E' every edge incident on either u or v
return C
```



An Approximation Algorithm based on Greedy

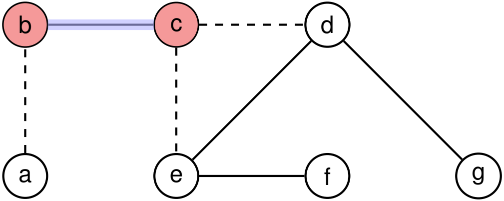
```
APPROX-VERTEX-COVER(G)
C = ∅
E' = G.E
while E' ≠ ∅
    let (u, v) be an arbitrary edge of E'
    C = C ∪ {u, v}
    remove from E' every edge incident on either u or v
return C
```



An Approximation Algorithm based on Greedy

APPROX-VERTEX-COVER(G)

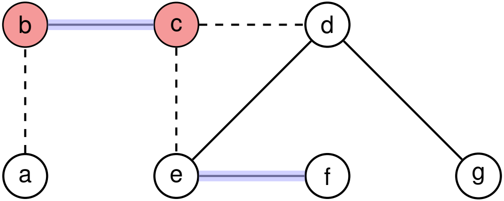
$C = \emptyset$
$E' = G.E$
while $E' \neq \emptyset$
    let (u, v) be an arbitrary edge of E'
    $C = C \cup \{u, v\}$
    remove from E' every edge incident on either u or v
return C



An Approximation Algorithm based on Greedy

APPROX-VERTEX-COVER(G)

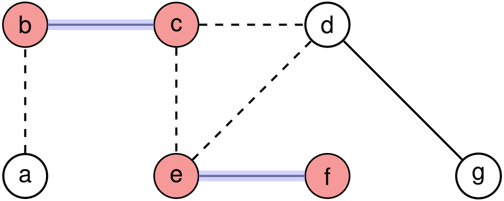
$C = \emptyset$
$E' = G.E$
while $E' \neq \emptyset$
  let (u, v) be an arbitrary edge of E'
  $C = C \cup \{u, v\}$
  remove from E' every edge incident on either u or v
return C



An Approximation Algorithm based on Greedy

APPROX-VERTEX-COVER(G)

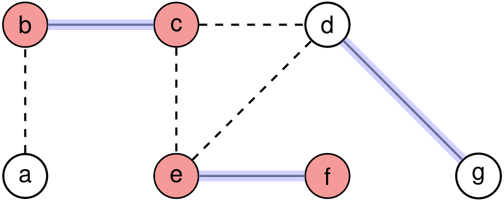
```
C = ∅
E' = G.E
while E' ≠ ∅
    let (u, v) be an arbitrary edge of E'
    C = C ∪ {u, v}
    remove from E' every edge incident on either u or v
return C
```



An Approximation Algorithm based on Greedy

APPROX-VERTEX-COVER(G)

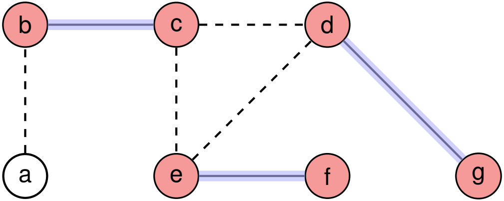
$C = \emptyset$
$E' = G.E$
while $E' \neq \emptyset$
    let (u, v) be an arbitrary edge of E'
    $C = C \cup \{u, v\}$
    remove from E' every edge incident on either u or v
return C



An Approximation Algorithm based on Greedy

APPROX-VERTEX-COVER(G)

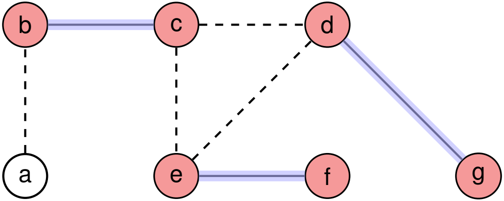
```
C = ∅
E' = G.E
while E' ≠ ∅
    let (u, v) be an arbitrary edge of E'
    C = C ∪ {u, v}
    remove from E' every edge incident on either u or v
return C
```



An Approximation Algorithm based on Greedy

APPROX-VERTEX-COVER(G)

```
C = ∅
E′ = G.E
while E′ ≠ ∅
    let (u, v) be an arbitrary edge of E′
    C = C ∪ {u, v}
    remove from E′ every edge incident on either u or v
return C
```

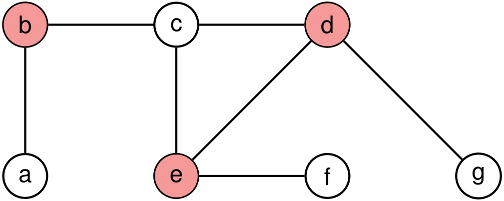


APPROX-VERTEX-COVER produces a set of size 6.

An Approximation Algorithm based on Greedy

APPROX-VERTEX-COVER(G)

$C = \emptyset$
$E' = G.E$
while $E' \neq \emptyset$
    let (u, v) be an arbitrary edge of E'
    $C = C \cup \{u, v\}$
    remove from E' every edge incident on either u or v
return C



The optimal solution has size 3.

Analysis of Greedy for Vertex Cover

```
APPROX-VERTEX-COVER(G)
C = ∅
E' = G.E
while E' ≠ ∅
    let (u, v) be an arbitrary edge of E'
    C = C ∪ {u, v}
    remove from E' every edge incident on either u or v
return C
```

Analysis of Greedy for Vertex Cover

APPROX-VERTEX-COVER(G)

```
C = ∅
E' = G.E
while E' ≠ ∅
    let (u, v) be an arbitrary edge of E'
    C = C ∪ {u, v}
    remove from E' every edge incident on either u or v
return C
```

Theorem 35.1

APPROX-VERTEX-COVER is a poly-time 2-approximation algorithm.

Analysis of Greedy for Vertex Cover

APPROX-VERTEX-COVER(G)

```
C = ∅
E' = G.E
while E' ≠ ∅
    let (u, v) be an arbitrary edge of E'
    C = C ∪ {u, v}
    remove from E' every edge incident on either u or v
return C
```

Theorem 35.1

APPROX-VERTEX-COVER is a poly-time 2-approximation algorithm.

Proof:

Analysis of Greedy for Vertex Cover

```
APPROX-VERTEX-COVER(G)
C = ∅
E' = G.E
while E' ≠ ∅
      let (u, v) be an arbitrary edge of E'
      C = C ∪ {u, v}
      remove from E' every edge incident on either u or v
return C
```

Theorem 35.1

APPROX-VERTEX-COVER is a poly-time 2-approximation algorithm.

Proof:

- Running time is $O(V + E)$ (using adjacency lists to represent E')

Analysis of Greedy for Vertex Cover

```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```

Theorem 35.1

APPROX-VERTEX-COVER is a poly-time 2-approximation algorithm.

Proof:

- Running time is $O(V + E)$ (using adjacency lists to represent E')
- Let $A \subseteq E$ denote the set of edges picked in line 4

Analysis of Greedy for Vertex Cover

APPROX-VERTEX-COVER(G)

```
1  C = ∅
2  E′ = G.E
3  while E′ ≠ ∅
4      let (u, v) be an arbitrary edge of E′
5      C = C ∪ {u, v}
6      remove from E′ every edge incident on either u or v
7  return C
```

Theorem 35.1

APPROX-VERTEX-COVER is a poly-time 2-approximation algorithm.

Proof:

- Running time is $O(V + E)$ (using adjacency lists to represent E')
- Let $A \subseteq E$ denote the set of edges picked in line 4
- Every optimal cover C^* must include at least one endpoint of edges in A,

Analysis of Greedy for Vertex Cover

```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```

Theorem 35.1

APPROX-VERTEX-COVER is a poly-time 2-approximation algorithm.

Proof:

- Running time is $O(V + E)$ (using adjacency lists to represent E')
- Let $A \subseteq E$ denote the set of edges picked in line 4
- Every optimal cover C^* must include at least one endpoint of edges in A, and edges in A do not share a common endpoint:

Analysis of Greedy for Vertex Cover

APPROX-VERTEX-COVER(G)

```
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```

Theorem 35.1

APPROX-VERTEX-COVER is a poly-time 2-approximation algorithm.

Proof:

- Running time is $O(V + E)$ (using adjacency lists to represent E')
- Let $A \subseteq E$ denote the set of edges picked in line 4
- Every optimal cover C^* must include at least one endpoint of edges in A, and edges in A do not share a common endpoint: $\boxed{|C^*| \geq |A|}$

Analysis of Greedy for Vertex Cover

```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```

Theorem 35.1

APPROX-VERTEX-COVER is a poly-time 2-approximation algorithm.

Proof:

- Running time is $O(V + E)$ (using adjacency lists to represent E')
- Let $A \subseteq E$ denote the set of edges picked in line 4
- Every optimal cover C^* must include at least one endpoint of edges in A, and edges in A do not share a common endpoint: $\boxed{|C^*| \geq |A|}$
- Every edge in A contributes 2 vertices to $|C|$:

Analysis of Greedy for Vertex Cover

APPROX-VERTEX-COVER(G)

```
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```

Theorem 35.1

APPROX-VERTEX-COVER is a poly-time 2-approximation algorithm.

Proof:

- Running time is $O(V + E)$ (using adjacency lists to represent E')
- Let $A \subseteq E$ denote the set of edges picked in line 4
- Every optimal cover C^* must include at least one endpoint of edges in A, and edges in A do not share a common endpoint: $\boxed{|C^*| \geq |A|}$
- Every edge in A contributes 2 vertices to $|C|$: $\boxed{|C| = 2|A|}$

Analysis of Greedy for Vertex Cover

APPROX-VERTEX-COVER(G)

```
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```

Theorem 35.1

APPROX-VERTEX-COVER is a poly-time 2-approximation algorithm.

Proof:

- Running time is $O(V+E)$ (using adjacency lists to represent E')
- Let $A \subseteq E$ denote the set of edges picked in line 4
- Every optimal cover C^* must include at least one endpoint of edges in A, and edges in A do not share a common endpoint: $\boxed{|C^*| \geq |A|}$
- Every edge in A contributes 2 vertices to $|C|$: $\boxed{|C| = 2|A| \leq 2|C^*|.}$

Analysis of Greedy for Vertex Cover

```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E′ = G.E
3  while E′ ≠ ∅
4      let (u, v) be an arbitrary edge of E′
5      C = C ∪ {u, v}
6      remove from E′ every edge incident on either u or v
7  return C
```

Theorem 35.1

APPROX-VERTEX-COVER is a poly-time 2-approximation algorithm.

Proof:

- Running time is $O(V + E)$ (using adjacency lists to represent E')
- Let $A \subseteq E$ denote the set of edges picked in line 4
- Every optimal cover C^* must include at least one endpoint of edges in A, and edges in A do not share a common endpoint: $\boxed{|C^*| \geq |A|}$
- Every edge in A contributes 2 vertices to $|C|$: $\boxed{|C| = 2|A| \leq 2|C^*|.}$ $\square$

Analysis of Greedy for Vertex Cover

APPROX-VERTEX-COVER(G)

```
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```

We can bound the size of the returned solution without knowing the (size of an) optimal solution!

Theorem 35.1

APPROX-VERTEX-COVER is a poly-time 2-approximation algorithm.

Proof:

- Running time is $O(V+E)$ (using adjacency lists to represent E')
- Let $A \subseteq E$ denote the set of edges picked in line 4
- Every optimal cover C^* must include at least one endpoint of edges in A, and edges in A do not share a common endpoint: $\boxed{|C^*| \geq |A|}$
- Every edge in A contributes 2 vertices to $|C|$: $\boxed{|C| = 2|A| \leq 2|C^*|.}$ $\square$

Analysis of Greedy for Vertex Cover

APPROX-VERTEX-COVER(G)

1 $C = \emptyset$
2 $E' = G.E$
3 **while** $E' \neq \emptyset$
4 let (u, v) be an arbitrary edge of E'
5 $C = C \cup \{u, v\}$
6 remove from E' every edge incident on either u or v
7 **return** C

A "vertex-based" Greedy that adds **one** vertex at each iteration fails to achieve an approximation ratio of 2 (Exercise)!

We can bound the size of the returned solution without knowing the (size of an) optimal solution!

Theorem 35.1

APPROX-VERTEX-COVER is a poly-time 2-approximation algorithm.

Proof:

- Running time is $O(V + E)$ (using adjacency lists to represent E')
- Let $A \subseteq E$ denote the set of edges picked in line 4
- Every optimal cover C^* must include at least one endpoint of edges in A, and edges in A do not share a common endpoint: $\boxed{|C^*| \geq |A|}$
- Every edge in A contributes 2 vertices to $|C|$: $\boxed{|C| = 2|A| \leq 2|C^*|.}$ $\square$

Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Solving Special Cases

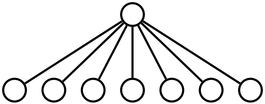
Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Solving Special Cases

Strategies to cope with NP-complete problems

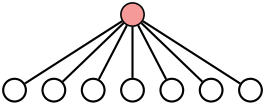
1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.



Solving Special Cases

Strategies to cope with NP-complete problems

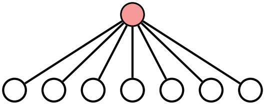
1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

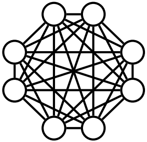


Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

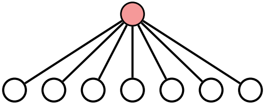


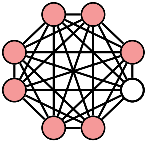


Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

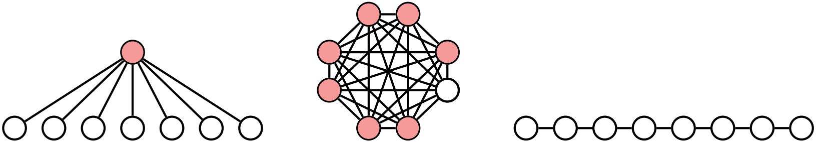




Solving Special Cases

Strategies to cope with NP-complete problems

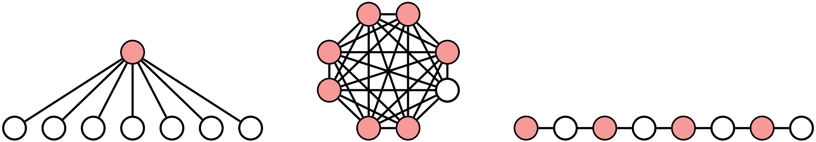
1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.



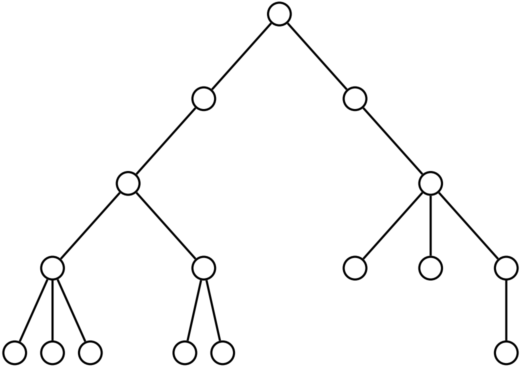
Solving Special Cases

Strategies to cope with NP-complete problems

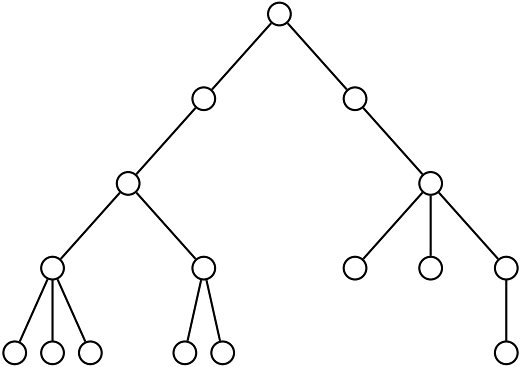
1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.



Vertex Cover on Trees

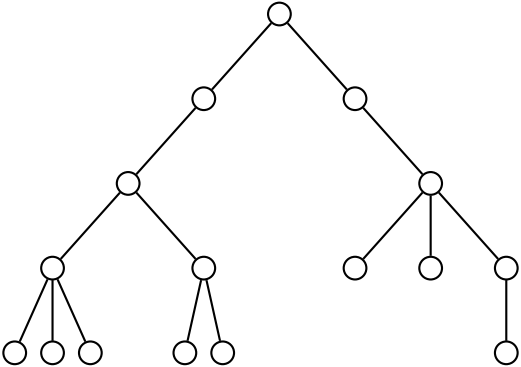


Vertex Cover on Trees



There exists an optimal vertex cover which does not include any leaves.

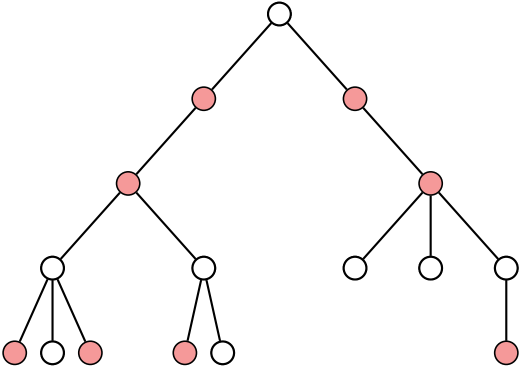
Vertex Cover on Trees



There exists an optimal vertex cover which does not include any leaves.

Exchange-Argument: Replace any leaf in the cover by its parent.

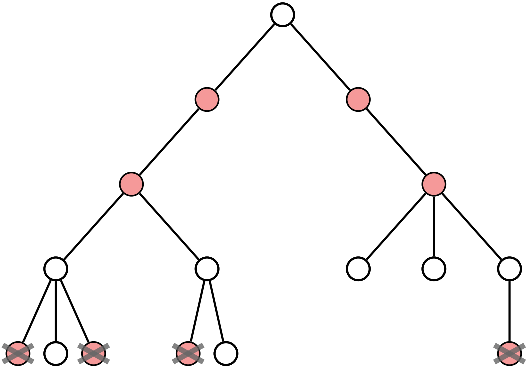
Vertex Cover on Trees



There exists an optimal vertex cover which does not include any leaves.

Exchange-Argument: Replace any leaf in the cover by its parent.

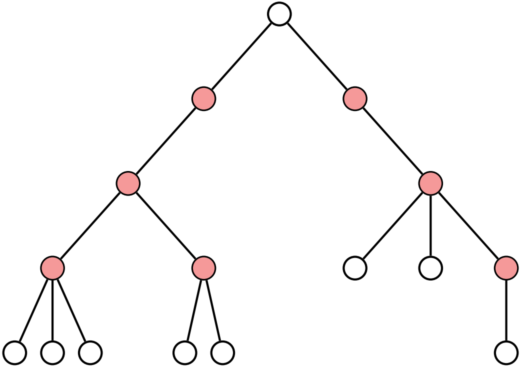
Vertex Cover on Trees



There exists an optimal vertex cover which does not include any leaves.

Exchange-Argument: Replace any leaf in the cover by its parent.

Vertex Cover on Trees



There exists an optimal vertex cover which does not include any leaves.

Exchange-Argument: Replace any leaf in the cover by its parent.

Solving Vertex Cover on Trees

There exists an optimal vertex cover which does not include any leaves.

Solving Vertex Cover on Trees

There exists an optimal vertex cover which does not include any leaves.

VERTEX-COVER-TREES(G)

$C = \emptyset$
while $\exists$ leaves in G
    Add all parents to C
    Remove all leaves and their parents from G
return C

Solving Vertex Cover on Trees

There exists an optimal vertex cover which does not include any leaves.

VERTEX-COVER-TREES(G)

$C = \emptyset$
while $\exists$ leaves in G
    Add all parents to C
    Remove all leaves and their parents from G
return C

Clear: Running time is $O(V)$, and the returned solution is a vertex cover.

Solving Vertex Cover on Trees

There exists an optimal vertex cover which does not include any leaves.

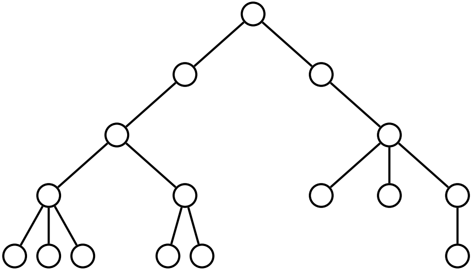
VERTEX-COVER-TREES(G)

1: $C = \emptyset$
2: **while** $\exists$ leaves in G
3: Add all parents to C
4: Remove all leaves and their parents from G
5: **return** C

Clear: Running time is $O(V)$, and the returned solution is a vertex cover.

Solution is also optimal. (Use inductively the existence of an optimal vertex cover without leaves)

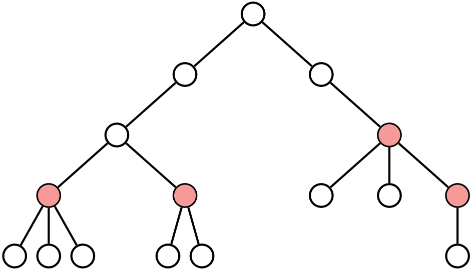
Execution on a Small Example



VERTEX-COVER-TREES(G)

```
C = ∅
while ∃ leaves in G
    Add all parents to C
    Remove all leaves and their parents from G
return C
```

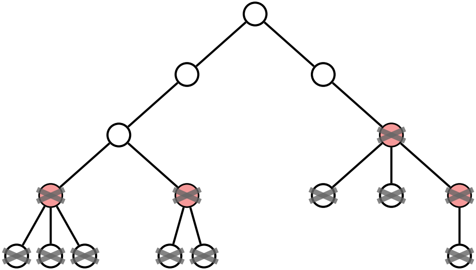
Execution on a Small Example



VERTEX-COVER-TREES(G)

```
C = ∅
while ∃ leaves in G
    Add all parents to C
    Remove all leaves and their parents from G
return C
```

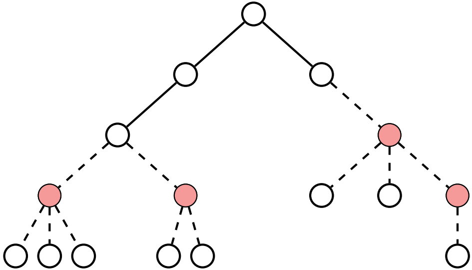
Execution on a Small Example



VERTEX-COVER-TREES(G)

1: $C = \emptyset$
2: **while** $\exists$ leaves in G
3: Add all parents to C
4: Remove all leaves and their parents from G
5: **return** C

Execution on a Small Example



VERTEX-COVER-TREES(G)

$C = \emptyset$

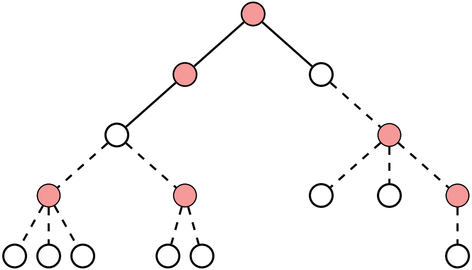
while $\exists$ leaves in G

    Add all parents to C

    Remove all leaves and their parents from G

return C

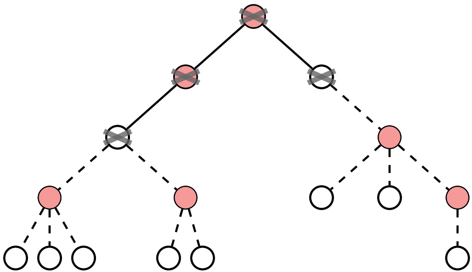
Execution on a Small Example



VERTEX-COVER-TREES(G)

```
C = ∅
while ∃ leaves in G
    Add all parents to C
    Remove all leaves and their parents from G
return C
```

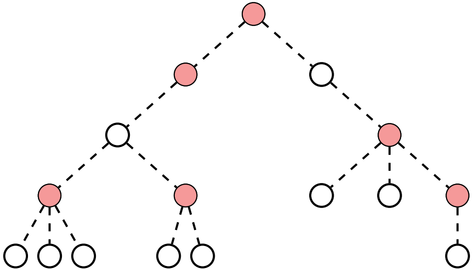
Execution on a Small Example



VERTEX-COVER-TREES(G)

1: $C = \emptyset$
2: **while** $\exists$ leaves in G
3: Add all parents to C
4: Remove all leaves and their parents from G
5: **return** C

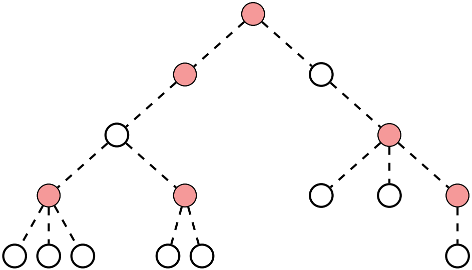
Execution on a Small Example



VERTEX-COVER-TREES(G)

$C = \emptyset$
while $\exists$ leaves in G
    Add all parents to C
    Remove all leaves and their parents from G
return C

Execution on a Small Example



VERTEX-COVER-TREES(G)

1: $C = \emptyset$
2: **while** $\exists$ leaves in G
3: Add all parents to C
4: Remove all leaves and their parents from G
5: **return** C

Problem can be also solved on bipartite graphs, using Max-Flows and Min-Cuts.

Exact Algorithms

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Exact Algorithms

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Exact Algorithms

Such algorithms are called exact algorithms.

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Exact Algorithms

Such algorithms are called exact algorithms.

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Focus on instances where the minimum vertex cover is small, that is, **less or equal** than some given integer k.

Exact Algorithms

Such algorithms are called exact algorithms.

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Focus on instances where the minimum vertex cover is small, that is, **less or equal** than some given integer k.

Simple Brute-Force Search would take $\approx \binom{n}{k} = \Theta(n^k)$ time.

Towards a more efficient Search

Substructure Lemma

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let G_u be the graph obtained by deleting u and its incident edges (G_v is defined similarly). Then G has a vertex cover of size k if and only if G_u or G_v (or both) have a vertex cover of size $k - 1$.

Towards a more efficient Search

Substructure Lemma

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let G_u be the graph obtained by deleting u and its incident edges (G_v is defined similarly). Then G has a vertex cover of size k if and only if G_u or G_v (or both) have a vertex cover of size $k - 1$.

Reminiscent of Dynamic Programming.

Towards a more efficient Search

Substructure Lemma

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let G_u be the graph obtained by deleting u and its incident edges (G_v is defined similarly). Then G has a vertex cover of size k if and only if G_u or G_v (or both) have a vertex cover of size $k - 1$.

Proof:

$\Leftarrow$ Assume G_u has a vertex cover C_u of size $k - 1$.

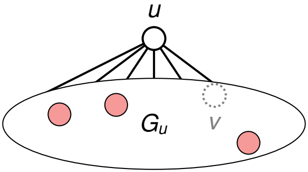
Towards a more efficient Search

Substructure Lemma

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let G_u be the graph obtained by deleting u and its incident edges (G_v is defined similarly). Then G has a vertex cover of size k if and only if G_u or G_v (or both) have a vertex cover of size $k - 1$.

Proof:

$\Leftarrow$ Assume G_u has a vertex cover C_u of size $k-1$.



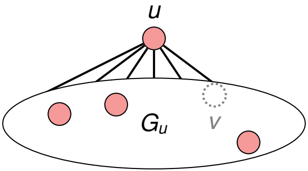
Towards a more efficient Search

Substructure Lemma

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let G_u be the graph obtained by deleting u and its incident edges (G_v is defined similarly). Then G has a vertex cover of size k if and only if G_u or G_v (or both) have a vertex cover of size $k - 1$.

Proof:

$\Leftarrow$ Assume G_u has a vertex cover C_u of size $k - 1$.
Adding u yields a vertex cover of G which is of size k



Towards a more efficient Search

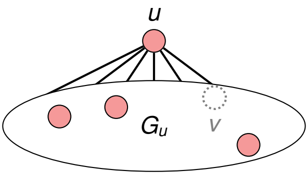
Substructure Lemma

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let G_u be the graph obtained by deleting u and its incident edges (G_v is defined similarly). Then G has a vertex cover of size k if and only if G_u or G_v (or both) have a vertex cover of size $k - 1$.

Proof:

$\Leftarrow$ Assume G_u has a vertex cover C_u of size $k - 1$.
Adding u yields a vertex cover of G which is of size k

$\Rightarrow$ Assume G has a vertex cover C of size k, which contains, say u.



Towards a more efficient Search

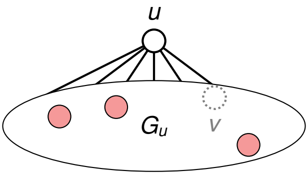
Substructure Lemma

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let G_u be the graph obtained by deleting u and its incident edges (G_v is defined similarly). Then G has a vertex cover of size k if and only if G_u or G_v (or both) have a vertex cover of size $k-1$.

Proof:

$\Leftarrow$ Assume G_u has a vertex cover C_u of size $k-1$.
Adding u yields a vertex cover of G which is of size k

$\Rightarrow$ Assume G has a vertex cover C of size k, which contains, say u.
Removing u from C yields a vertex cover of G_u which is of size $k-1$. $\square$



A More Efficient Search Algorithm

VERTEX-COVER-SEARCH(G, k)

If $E = \emptyset$ **return** $\emptyset$
If $k = 0$ and $E \neq \emptyset$ **return** $\perp$
Pick an arbitrary edge $(u, v) \in E$
$S_1 =$ VERTEX-COVER-SEARCH$(G_u, k-1)$
$S_2 =$ VERTEX-COVER-SEARCH$(G_v, k-1)$
if $S_1 \neq \perp$ **return** $S_1 \cup \{u\}$
if $S_2 \neq \perp$ **return** $S_2 \cup \{v\}$
return $\perp$

A More Efficient Search Algorithm

$\textsc{Vertex-Cover-Search}(G, k)$

If $E = \emptyset$ **return** $\emptyset$
If $k = 0$ and $E \neq \emptyset$ **return** $\perp$
Pick an arbitrary edge $(u, v) \in E$
$S_1 = \textsc{Vertex-Cover-Search}(G_u, k-1)$
$S_2 = \textsc{Vertex-Cover-Search}(G_v, k-1)$
if $S_1 \neq \perp$ **return** $S_1 \cup \{u\}$
if $S_2 \neq \perp$ **return** $S_2 \cup \{v\}$
return $\perp$

Correctness follows by the Substructure Lemma and induction.

A More Efficient Search Algorithm

$\textsc{Vertex-Cover-Search}(G, k)$

If $E = \emptyset$ **return** $\emptyset$
If $k = 0$ and $E \neq \emptyset$ **return** $\perp$
Pick an arbitrary edge $(u, v) \in E$
$S_1 = \textsc{Vertex-Cover-Search}(G_u, k-1)$
$S_2 = \textsc{Vertex-Cover-Search}(G_v, k-1)$
if $S_1 \neq \perp$ **return** $S_1 \cup \{u\}$
if $S_2 \neq \perp$ **return** $S_2 \cup \{v\}$
return $\perp$

Running time:

A More Efficient Search Algorithm

```
VERTEX-COVER-SEARCH(G, k)
If E = ∅ return ∅
If k = 0 and E ≠ ∅ return ⊥
Pick an arbitrary edge (u, v) ∈ E
S_1 = VERTEX-COVER-SEARCH(G_u, k − 1)
S_2 = VERTEX-COVER-SEARCH(G_v, k − 1)
if S_1 ≠ ⊥ return S_1 ∪ {u}
if S_2 ≠ ⊥ return S_2 ∪ {v}
return ⊥
```

Running time:

- Depth k, branching factor 2

A More Efficient Search Algorithm

$\textsc{Vertex-Cover-Search}(G, k)$

If $E = \emptyset$ **return** $\emptyset$
If $k = 0$ and $E \neq \emptyset$ **return** $\perp$
Pick an arbitrary edge $(u, v) \in E$
$S_1 = \textsc{Vertex-Cover-Search}(G_u, k-1)$
$S_2 = \textsc{Vertex-Cover-Search}(G_v, k-1)$
if $S_1 \neq \perp$ **return** $S_1 \cup \{u\}$
if $S_2 \neq \perp$ **return** $S_2 \cup \{v\}$
return $\perp$

Running time:

- Depth k, branching factor 2 $\Rightarrow$ total number of calls is $O(2^k)$

A More Efficient Search Algorithm

$\textsc{Vertex-Cover-Search}(G, k)$

If $E = \emptyset$ **return** $\emptyset$

If $k = 0$ and $E \neq \emptyset$ **return** $\perp$

Pick an arbitrary edge $(u, v) \in E$

$S_1 = \textsc{Vertex-Cover-Search}(G_u, k - 1)$

$S_2 = \textsc{Vertex-Cover-Search}(G_v, k - 1)$

if $S_1 \neq \perp$ **return** $S_1 \cup \{u\}$

if $S_2 \neq \perp$ **return** $S_2 \cup \{v\}$

return $\perp$

Running time:

- Depth k, branching factor 2 $\Rightarrow$ total number of calls is $O(2^k)$
- $O(E)$ work per recursive call

A More Efficient Search Algorithm

VERTEX-COVER-SEARCH(G, k)

If $E = \emptyset$ **return** $\emptyset$
If $k = 0$ and $E \neq \emptyset$ **return** $\perp$
Pick an arbitrary edge $(u, v) \in E$
$S_1 =$ VERTEX-COVER-SEARCH$(G_u, k - 1)$
$S_2 =$ VERTEX-COVER-SEARCH$(G_v, k - 1)$
if $S_1 \neq \perp$ **return** $S_1 \cup \{u\}$
if $S_2 \neq \perp$ **return** $S_2 \cup \{v\}$
return $\perp$

Running time:

- Depth k, branching factor 2 $\Rightarrow$ total number of calls is $O(2^k)$
- $O(E)$ work per recursive call
- Total runtime: $O(2^k \cdot E)$.

A More Efficient Search Algorithm

$\textsc{Vertex-Cover-Search}(G, k)$

If $E = \emptyset$ **return** $\emptyset$
If $k = 0$ and $E \neq \emptyset$ **return** $\perp$
Pick an arbitrary edge $(u, v) \in E$
$S_1 = \textsc{Vertex-Cover-Search}(G_u, k-1)$
$S_2 = \textsc{Vertex-Cover-Search}(G_v, k-1)$
if $S_1 \neq \perp$ **return** $S_1 \cup \{u\}$
if $S_2 \neq \perp$ **return** $S_2 \cup \{v\}$
return $\perp$

Running time:

- Depth k, branching factor 2 $\Rightarrow$ total number of calls is $O(2^k)$
- $O(E)$ work per recursive call
- Total runtime: $O(2^k \cdot E)$.

exponential in k, but much better than $\Theta(n^k)$ (i.e., still polynomial for $k = O(\log n)$)