# 贪心策略
# Greedy Strategy

钮鑫涛
Nanjing University
2023 Fall

*The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne. Thanks for their supports!*

# The Greedy Strategy

- For many games, you should think ahead, a strategy which focuses on immediate advantage could easily lead to defeat.
    - Such as playing chess.
- But for many other games, you can do quite well by simply making whichever move seems best at the moment, without worrying too much about future consequences.
    - Such as building an MST.

# The Greedy Strategy

- **The Greedy Algorithmic Strategy**: given a problem, build up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.
  - Sometimes it gives optimal solution.
  - Sometimes it gives near-optimal solution.
  - Or, it simply fails…

# An Activity-Selection Problem

- Assume we have one hall and $n$ activities $S = \{a_1, \cdots, a_n\}$.
  - Each activity has a start time $s_i$ and a finish time $f_i$.
  - Two activities cannot happen simultaneously in the hall.
  - Maximum number of activities we can schedule?

# An Activity-Selection Problem

- Let's start with "divide-and-conquer"
    - Define $S_i$ to be the set of activities start after $a_i$ finishes;
    - Define $F_i$ to be the set of activities finish before $a_i$ starts.
    - $OPT(S) = \max_{1 \le i \le n} \{OPT(F_i) + 1 + OPT(S_i)\}$

In any solution, some activity is the first to finish.

$$OPT(S) = \max_{1 \le i \le n} \{1 + OPT(S_i)\}$$

Observation: To make $\mathrm{OPT}(S)$ as large as possible, the activity that finishes first should finish as early as possible!

# An Activity-Selection Problem

- A greedy strategy to solve this problem:

ActivitySelection(S):

*Sort S into increasing order of finish time*

$SOL := \{a_1\},\ a' = a_1$

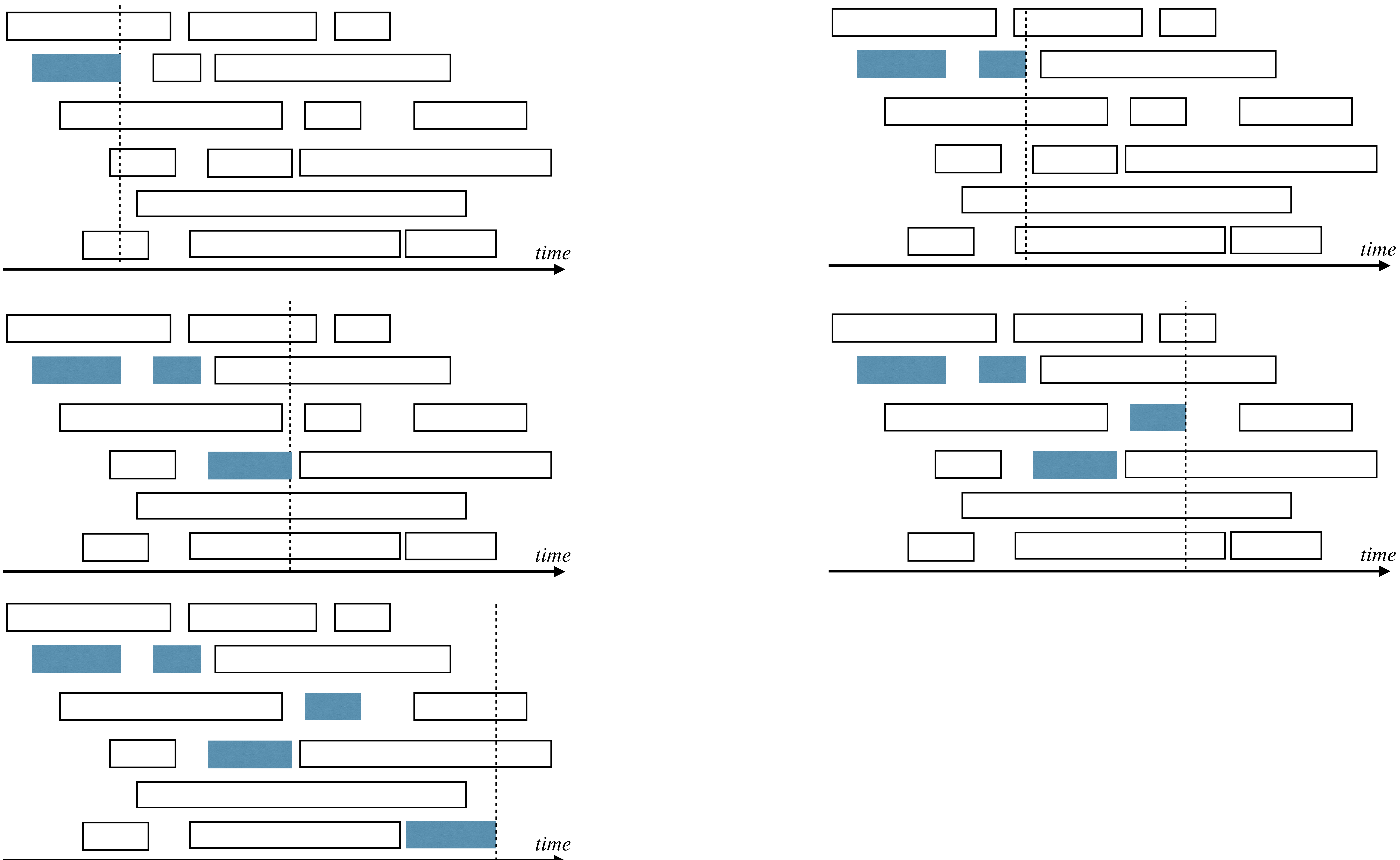
**for** $i := 2$ **to** $n$

  **If** $a_i.start\_time > a'.finish\_time$

    $SOL := SOL \cup \{a_i\}$

    $a' := a_i$

**return** $SOL$


time
time
time
time
time

# Correctness of the greedy strategy for this problem

- The Greedy Algorithm for the Activity-Selection Problem:
  - Add earliest finish activity $a'$ to solution, remove ones overlapping with $a'$.
  - Repeat until all activities are processed.
- How to formally prove this algorithm is correct?
  - The firstly selected activity is in some optimal solution.
  - The following selection is correct to this optimal solution.

# Correctness of the greedy strategy for this problem

Lemma 1 let $a'$ be the earliest finishing activity in $S$, then $a'$ is in some optimal solution of the problem.

- Proof:
  - Let $OPT(S)$ be an optimal solution to the problem, let $a$ be the earliest finishing activity in $OPT(S)$.
  - Assume $a' \notin OPT(S)$, otherwise we are done.
  - Then $SOL(S) = OPT(S) + a' - a$ is also a feasible solution, and it has same size as $OPT(S)$.
  - So $SOL(S)$ is also an optimal solution.

# Correctness of the greedy strategy for this problem

Lemma 2 let $a'$ be the earliest finishing activity in $S$, let $S'$ be the activities starting after $a'$, then $OPT(S') \cup \{a'\}$ is an optimal solution of the problem.

- Proof:
  - Let $OPT(S)$ be an optimal solution to the original problem, and $a' \in OPT(S)$. (Lemma 1 ensures such solution exists.)
  - Thus, $OPT(S) = SOL(S') \cup \{a'\}$.
  - If $OPT(S') \cup \{a'\}$ is not an optimal solution to the original problem, then it must be the case that $|SOL(S')| > |OPT(S')|$.
  - But this contradicts that $OPT(S')$ is an optimal solution for problem $S'$.

# Correctness of the greedy strategy for this problem

**Theorem** The greedy algorithm for the activity-selection problem is correct.

- Proof:
  - By induction on size of $S$.
  - When $|S| = 1$, the algorithm clearly is correct.
  - When $|S| = n$. Due to Lemma 2, $OPT(S) = OPT(S') \cup \{a'\}$
  - By induction hypothesis, the algorithm correctly finds $OPT(S')$. So we are done.

# Elements of the Greedy Strategy

# Elements of the Greedy Strategy

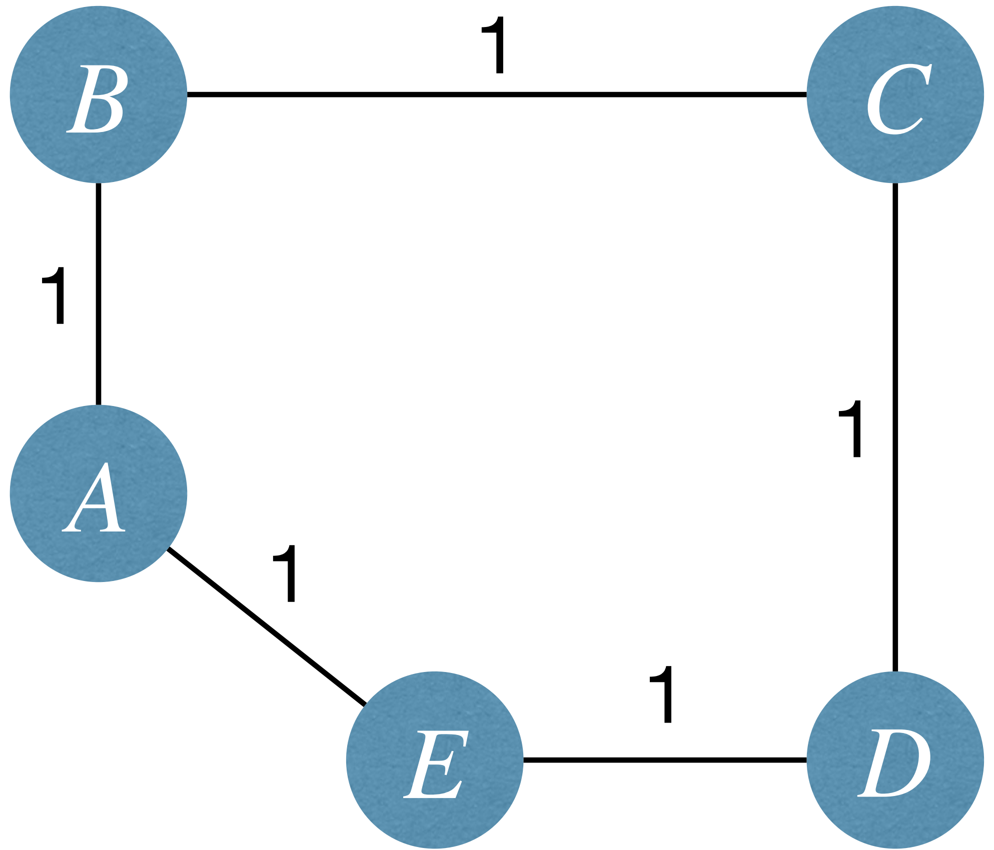
- If an (optimization) problem has following two properties, then the greedy strategy usually works for it:
  - **Optimal substructure.**
  - **Greedy property**.

# Optimal Substructure

- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solution(s) to subproblem(s):
  - Size $n$ problem $P(n)$, and optimal solution of $P(n)$ is $OPT_{P(n)}$.
  - Solving $P(n)$ needs to solve size $n' < n$ subproblem $P(n')$.
  - Optimal solution of $P(n')$: $OPT_{P(n')}$
  - $OPT_{P(n)}$ contains a solution of $P(n')$: $SOL_{P(n')}$
  - Optimal Substructure Property: $SOL_{P(n')}$ = $OPT_{P(n')}$
    - Or these two solutions provide same “utility” under certain metric.

# Optimal Substructure

- Example:
  - Lemma 2 in activity selection: let $a'$ be the earliest finishing activity in $S$, let $S'$ be the activities starting after $a'$, then $OPT(S') \cup \{a'\}$ is some $OPT(S)$.
- There are problems that do **NOT** exhibit optimal substructure property!
  - E.g., find the longest path between two vertices without repeating an edge.



# Greedy-Choice Property

- At each step when building a solution, make the choice that looks best for the *current* problem, *without* considering results from subproblems. That is, make local greedy choice at each step.
  - To solve $P(n)$, currently have $k$ choices $a_1$ to $a_k$. If we choose $a_i$, the problem is reduced to a smaller size $n_i$ subproblem $P(n_i)$.
  - If the problem only admits optimal structure:
    - Find $i$ that maximize, Utility$(a_i + OPT_{P(n_i)})$.
    - We have to compute $OPT_{P(n_i)}$ for all $i$ first.

# Greedy-Choice Property

Identifying a greedy-choice property is the challenging part!

- With greedy choice:
  - We **have a way** to pick correct $i$, without knowing any $OPT_{P(n_i)}$.
- Example:
  - Lemma 1 in activity selection: let $a'$ be the earliest finishing activity in $S$, then $a'$ is in some optimal solution of the problem.

# Fractional Knapsack Problem

- A thief robbing a warehouse finds $n$ items $A = \{a_1, \ldots, a_n\}$.
- Item $a_i$ is worth $v_i$ dollars and weighs $w_i$ pounds.
- The thief can carry at most $W$ pounds in his knapsack.
- The thief can carry **fraction of items.**
- What should the thief take to maximize his profit?

# Fractional Knapsack Problem

- A greedy strategy:
  - keep taking the most cost efficient item (i.e., $\max\{\frac{v_i}{w_i}\}$ ) until the knapsack is full.
- The greedy solution is optimal!
  - Greedy-choice
  - Optimal substructure

# Correctness of the greedy algorithm

- Lemma 1 [greedy-choice]: let $a_m$ be a most cost efficient item in $A$, then in some optimal solution, at least $w_{m'} = \min\{w_m, W\}$ pounds of $a_m$ are taken.

- Proof:

  - Consider an optimal solution, assume $w' < w_{m'}$ pounds of $a_m$ are taken.

  - Now, substitute $w_{m'} - w'$ pounds of other items with $a_m$.

  - Since $a_m$ is most cost-efficient, the new solution cannot be worse.

# Correctness of the greedy algorithm

- Lemma 2 [optimal substructure]: let $a_m$ be a most cost efficient item in $A$, then "$OPT_{W-\min\{w_m,W\}}(A-a_m)$ with $\min\{w_m, W\}$ pounds of $a_m$" is an optimal solution of the problem.

- Proof:

  - Consider some $OPT_{W(A)}$ containing $\min\{w_m, W\}$ pounds of $a_m$.

  - If optimal substructure does not hold, then $OPT_{W(A)}$ gives $SOL_{W-\min\{w_m,W\}}(A-a_m) > OPT_{W-\min\{w_m,W\}}(A-a_m)$.

  - But this contradicts the optimality of $OPT_{W-\min\{w_m,W\}}(A-a_m)$.

# 0-1 Knapsack Problem

- A thief robbing a warehouse finds $n$ items $A = \{a_1, \ldots, a_n\}$.
- Item $a_i$ is worth $v_i$ dollars and weighs $w_i$ pounds.
- The thief can carry at most $W$ pounds in his knapsack.
- The thief ***cannot*** carry fraction of items!
- What should the thief take to maximize his profit?

# 0-1 Knapsack Problem

- A greedy strategy:
  - keep taking the most cost efficient item (i.e., $\max\{\frac{v_i}{w_i}\}$ ) until the knapsack is full.
- **The greedy solution is NOT optimal!**
- A simple **counterexample**:
  - There are only two items.
  - Item One has value 2 and weighs 1 pound.
  - Item Two has value $W$ and weighs $W$ pounds.

The greedy solution can be arbitrarily bad!

# Why greedy strategy fail?

- Lemma 1 [greedy-choice]: let $a_m$ be a most cost efficient item that can fit into the bag, then in some optimal solution, this item is taken.

- Thus, this lemma cannot be proven!
  - Consider an optimal solution, assume $a_m$ is NOT taken.

    can $w' < w_m$ ?

  - Now, substitute $w' \geq w_m$ pounds of other items with $a_m$ (all $w_m$ pounds).
  - However, these $w'$ pounds of items may have aggregate value larger than $v_m$, since it may $w' > w_m$.

What about the optimal substructure property? That is, is $OPT_{W-w_x}(A - a_x)$ with $w_x$ pounds of $a_x$ is the optimal solution?

# A data compression problem

- Assume we have a data file containing 100k characters.
  - Further assume the file only uses 6 characters.
  - How to store this file to save space?
- Simplest way: use 3 bits to encode each char.
  - `a=000,b=001,…,f=101`
  - This costs 300k **bits** in total.
- Can we do better?

# A data compression problem

- How to store this file to save space?
  - Instead of using fixed-length codeword for each char, we should **let frequent chars use shorter codewords.** That is, use a *variable-length code.*

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| **Frequency** | 45k | 13k | 12k | 16k | 9k | 5k |
| **Fixed-length code** | 000 | 001 | 010 | 011 | 100 | 101 |
| **varaible-length code** | 0 | 00 | 01 | 1 | 10 | 11 |

How to decode bit string `000`?

# A data compression problem

- How to store this file to save space?
  - Instead of using fixed-length codeword for each char, we should **let frequent chars use shorter codewords.** That is, use a *variable-length code.*
  - To avoid ambiguity in decoding, variable-length code should be **prefix-free**: **no codeword is also a prefix** of some other codeword.

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| **Frequency** | 45k | 13k | 12k | 16k | 9k | 5k |
| **Fixed-length code** | 000 | 001 | 010 | 011 | 100 | 101 |
| **varaible-length code** | 0 | 101 | 100 | 111 | 1101 | 1100 |

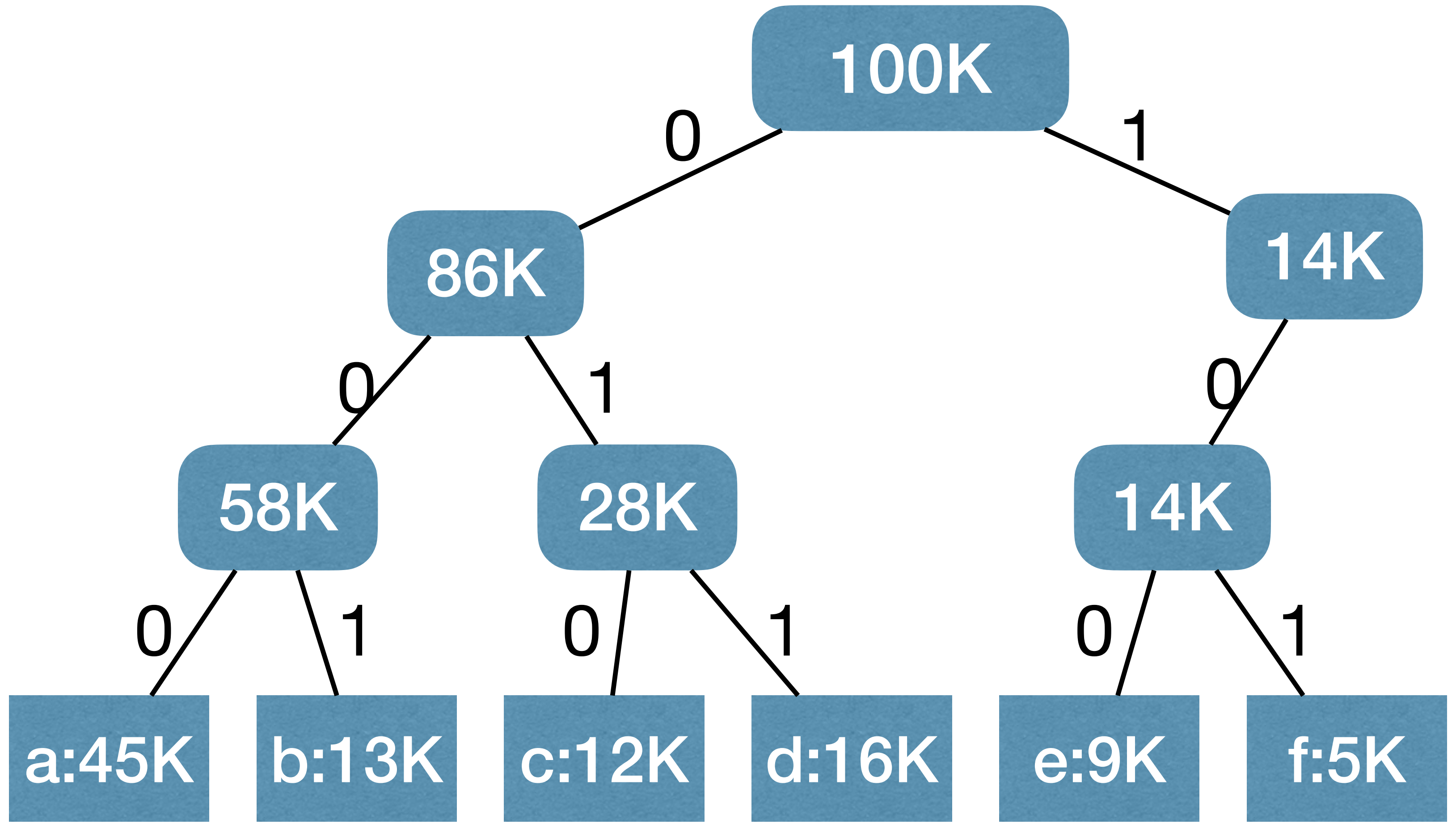
Fixed-length code vs Variable-length code: 300k vs 224k. This is ≈25% saving.

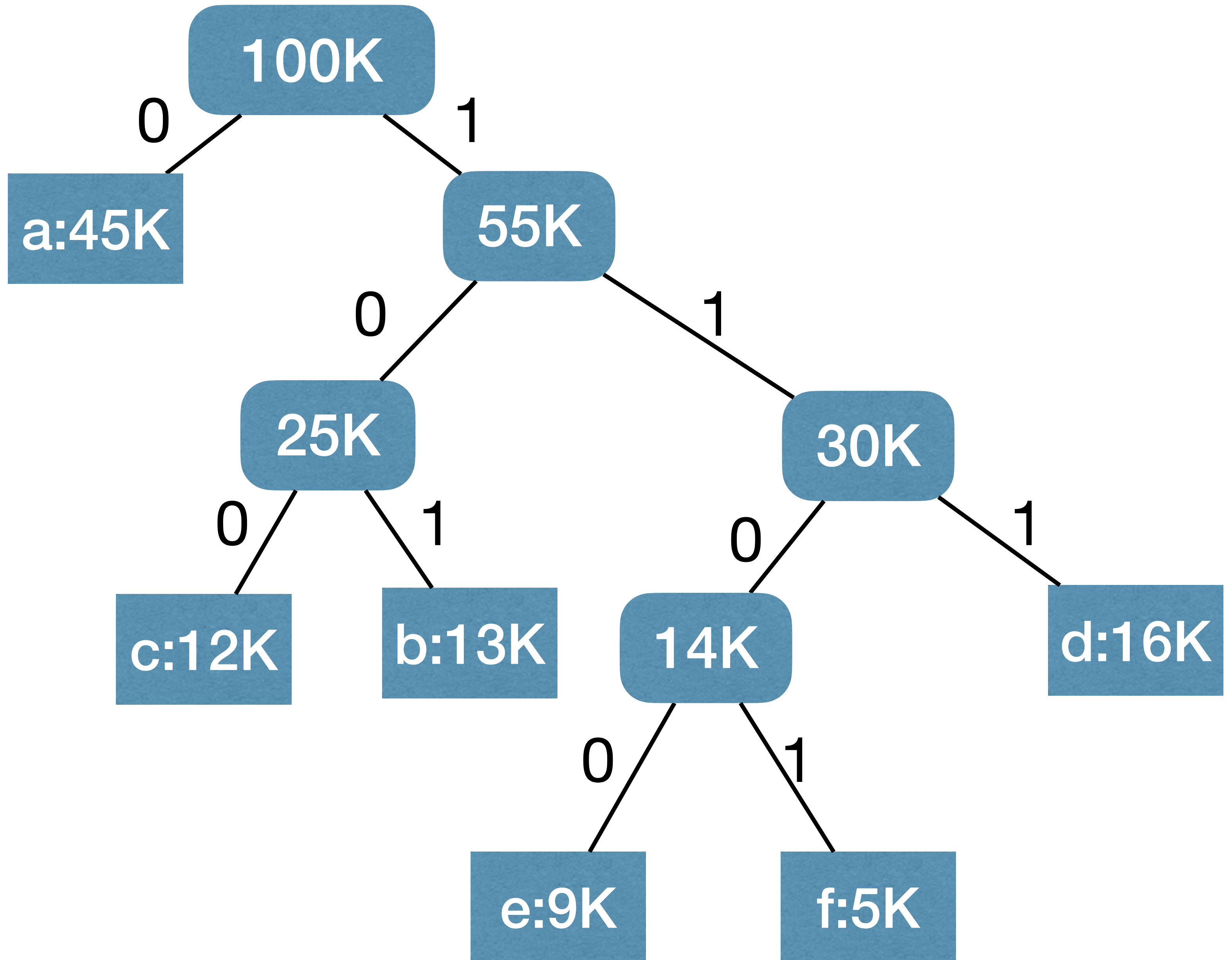
Is it optimal?

# Properties of prefix-free code

- Use a binary tree to visualize a prefix-free code.
  - Each leaf denotes a char.
  - Each internal node: left branch is 0, right branch is 1.
  - Path from root to leaf is the codeword of that char.
  - **Optimal code must be represented by a *full binary tree*: a tree each node having zero or two children.**

WHY?




| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| **Frequency** | 45k | 13k | 12k | 16k | 9k | 5k |
| **Fixed-length code** | 000 | 001 | 010 | 011 | 100 | 101 |
| **varaible-length code** | 0 | 101 | 100 | 111 | 1101 | 1100 |

# Length of encoded message

- Consider a file using a size $n$ alphabet $C = \{c_1, \ldots, c_n\}$. For each character, let $f_i$ be the frequency of char $c_i$.

- Let $T$ be a full binary tree representing a prefix-free code.For each character $c_i$, let $d_T(i)$ be the depth of $c_i$ in $T$.

  - Length of encoded message is $\sum_{i=1}^{n} f_i \cdot d_T(i)$

- Alternatively, recursively (bottom-up) define each internal node's frequency to be sum of its two children.

  - Length of encoded message is $\sum_{u \in tree \setminus root} f_u$

# Huffman Codes

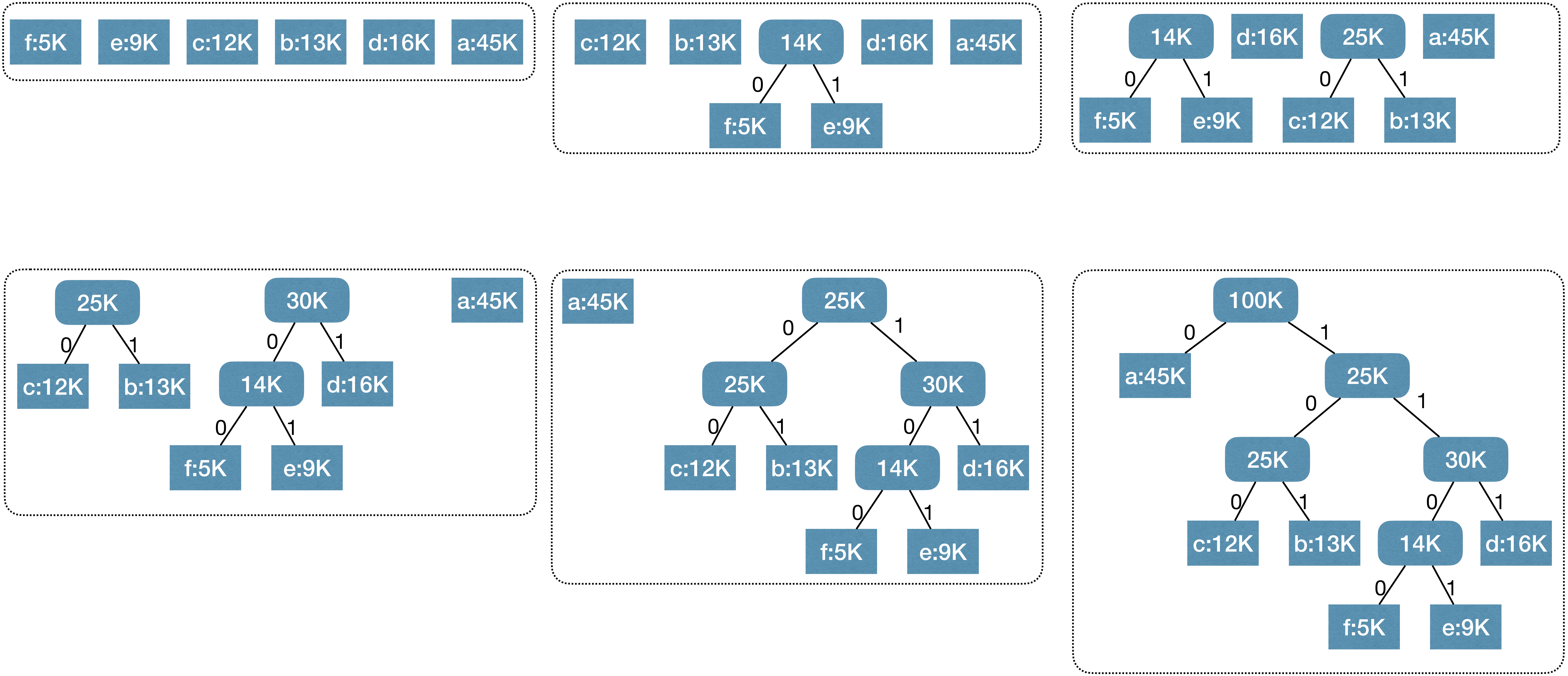
- How to construct optimal prefix-free code?
- Huffman Codes: Merge the two least frequent chars and recurse.

<u>Huffman(C):</u>

```
Build a priority queue Q based on frequency
for i := 1 to n - 1
    Allocate new node z
    x := z.left := Q.ExtractMin()
    y := z.right := Q.ExtractMin()
    z.frequency := x.frequency + y.frequency
    Q.Insert(z)
return Q.ExtractMin()
```

→ Time complexity is $O(n \log n)$

# Huffman Codes



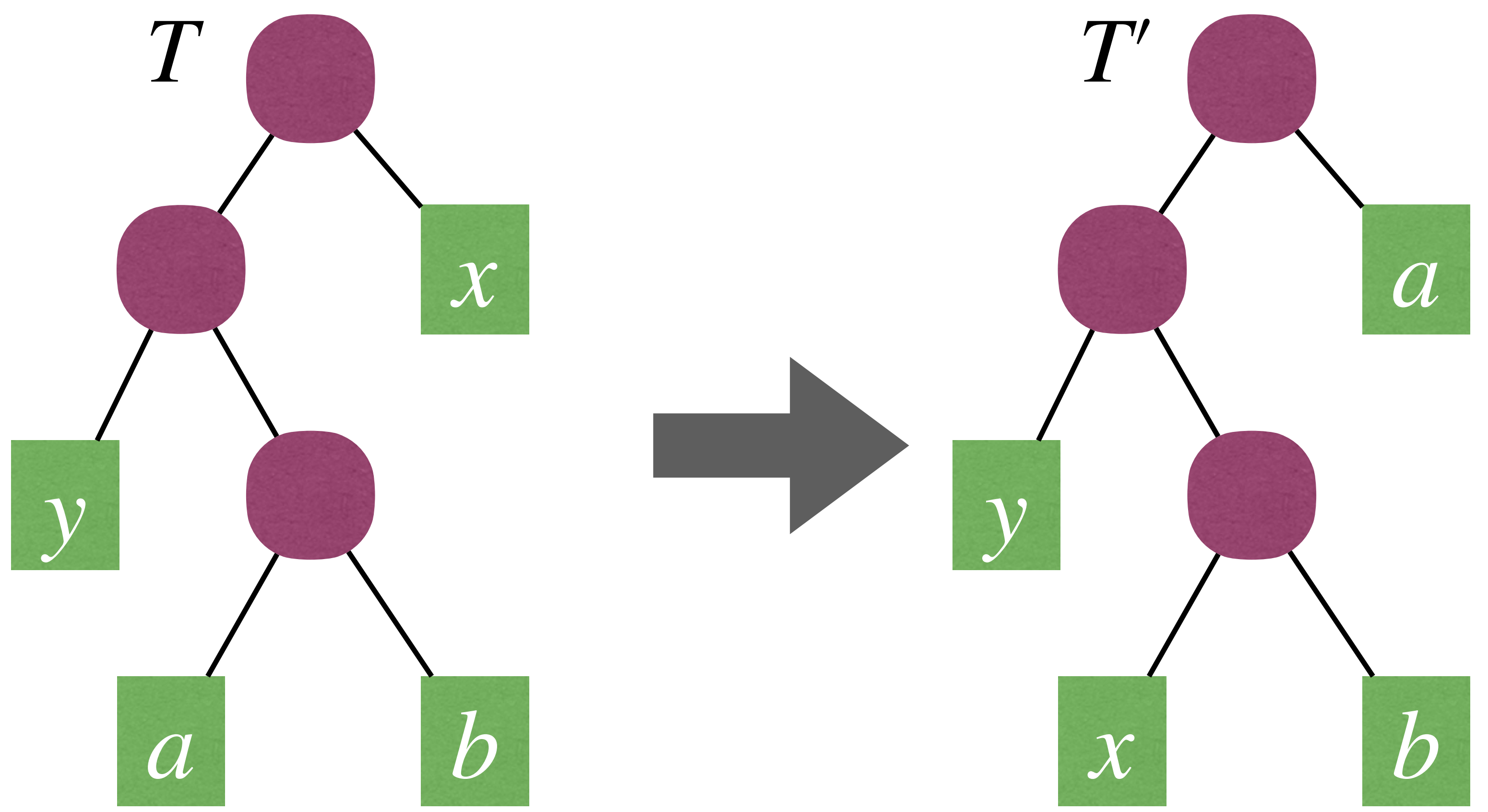
# Correctness of Huffman Codes

- Length of encoded message is computed by $\sum_{i=1}^{n} f_i \cdot d_T(i)$ or $\sum_{u \in tree \setminus root} f_u$
- Huffman Codes: Merge the two least frequent chars and recurse.
- Lemma 1 [greedy choice]: Let $x$ and $y$ be two least frequent chars, then in some optimal code tree, $x$ and $y$ are siblings and have largest depth.
- Lemma 2 [optimal substructure]: Let $x$ and $y$ be two least frequent chars in $C$. Let $C_z = C - \{x, y\} + \{z\}$ with $f_z = f_x + f_y$. Let $T_z$ be an optimal code tree for $C_z$. Let $T$ be a code tree obtained from $T_z$ by replacing leaf node $z$ with an internal node having $x$ and $y$ as children. Then, $T$ is an optimal code tree for $C$.

# Correctness of Huffman Codes

Lemma 1 [greedy choice]: Let $x$ and $y$ be two least frequent chars, then in some optimal code tree, $x$ and $y$ are siblings and have largest depth.

- Proof sketch:
  - Let $T$ be an optimal code tree with depth $d$.
  - Let $a$ and $b$ be siblings with depth $d$. (Recall $T$ is a full binary tree.)
  - Assume $a$ and $b$ are **not** $x$ and $y$. (Otherwise we are done.)
  - Let $T'$ be the code tree obtained by swapping $a$ and $x$.
  - $cost(T') = cost(T) + (d - d_T(x)) \cdot f_x - (d - d_T(x)) \cdot f_a = cost(T) + (d - d_T(x)) \cdot (f_x - f_a) \leq cost(T)$
  - Swapping $b$ and $y$, obtaining $T''$, further reduces the total cost.
  - So $T''$ must also be an optimal code tree.



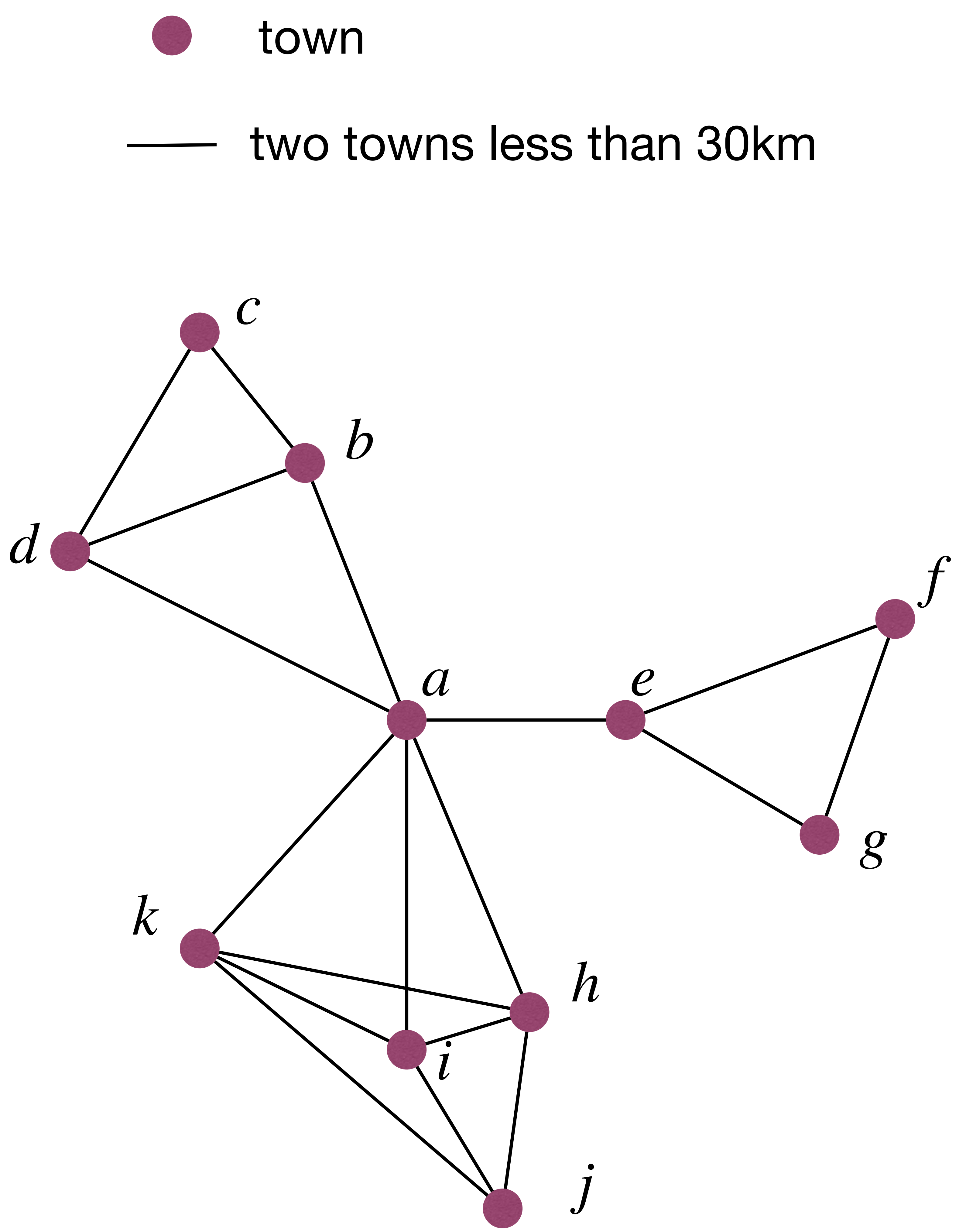
# Correctness of Huffman Codes

Lemma 2 [optimal substructure]: Let $x$ and $y$ be two least frequent chars in $C$. Let $C_z = C - \{x, y\} + \{z\}$ with $f_z = f_x + f_y$ . Let $T_z$ be an optimal code tree for $C_z$. Let $T$ be a code tree obtained from $T_z$ by replacing leaf node $z$ with an internal node having $x$ and $y$ as children. Then, $T$ is an optimal code tree for $C$.

- Proof sketch:
- Let $T'$ be an optimal code tree for $C$, with $x$ and $y$ being sibling leaves.
- $$Cost(T') = f_x + f_y + \sum_{u \in T' \setminus root\ and\ u \notin \{x,y\}} f_u = f_x + f_y + cost(T'_z) \geq f_x + f_y + cost(T_z) = cost(T)$$
- So T must be an optimal code tree for C.

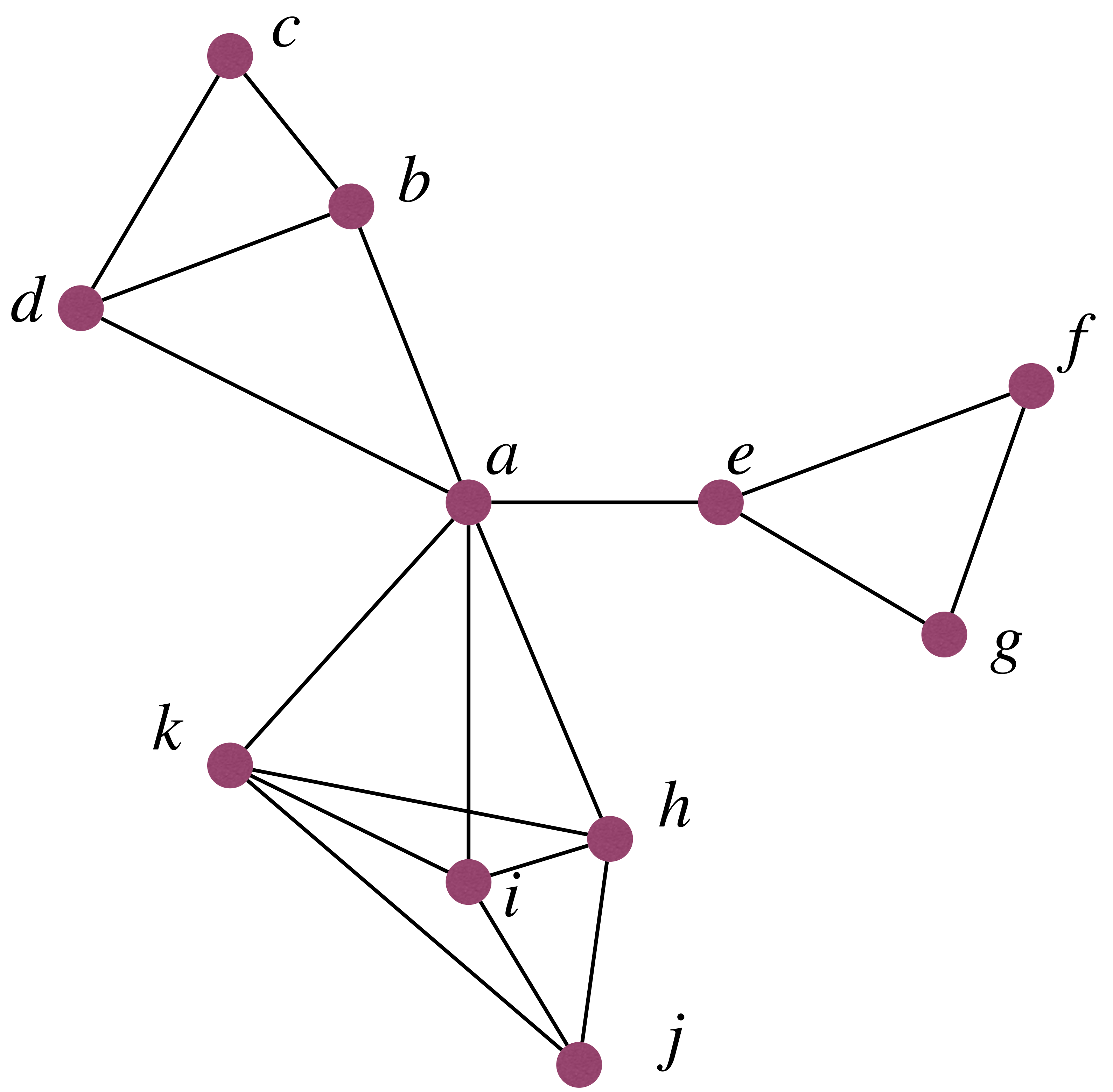
# Set Cover

- Suppose we need to build schools for $n$ towns.
- Each school must be in a town, no child should travel more than 30km to reach a school.
- Minimum number of schools we need to build?



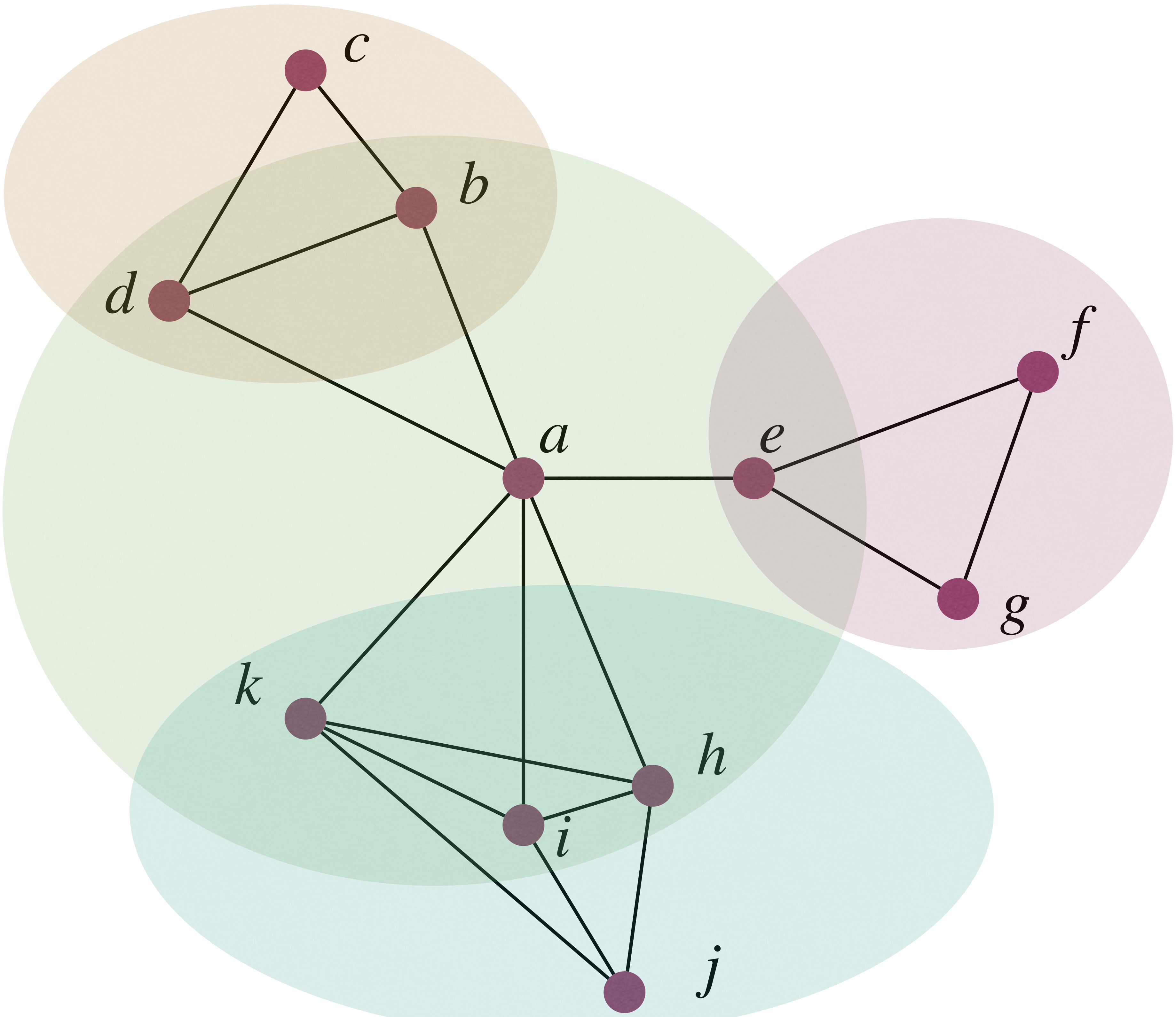
# Set Cover

- The Set Cover Problem:
- **Input:** a universe $U$ of $n$ elements; and $\mathcal{S} = \{S_1, \ldots, S_m\}$ where each $S_i \subseteq U$.
- **Output:** $\mathscr{C} \subseteq \mathcal{S}$ such that $\bigcup_{S_i \in \mathscr{C}} S_i = U$
  - That is, a subset of $\mathcal{S}$ that "covers" $U$.
- **Goal:** minimize $|\mathscr{C}|$



# Set Cover

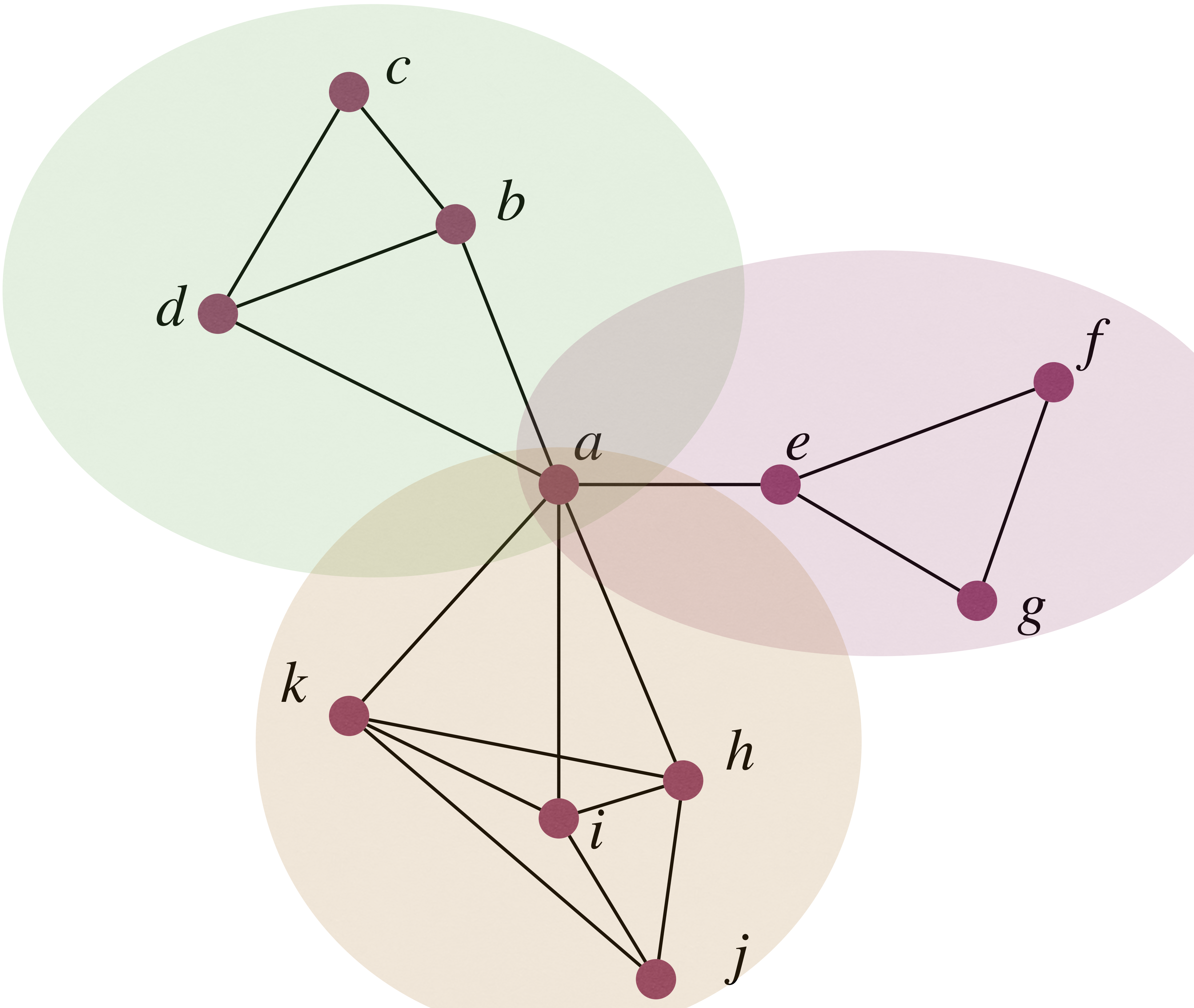
- **Simple greedy strategy**:
- Keep picking the town that covers most remaining uncovered towns, until we are done.
  - Pick the set that covers most uncovered elements, until all elements are covered.
- Greedy solution: $a, f, c, j$




Can we do better?

# Set Cover

- The optimal solution is $b, e, i$
- Nevertheless, the greedy solution $a, f, c, j$ is very close!
  - But, how close?



# Greedy solution of Set Cover is close to optimal

**Theorem** Suppose the optimal solution uses $k$ sets, then the greedy strategy will use at most $k \ln n$ sets.

- **Proof:**
- Let $n_t$ be number of uncovered elements after $t$ iterations. (Thus $n_0 = n$.)
- These $n_t$ elements can be covered by some $k$ sets. (The optimal solution will do)
- So one of the remaining sets will cover at least $\frac{n_t}{k}$ of these uncovered elements.
- Thus $n_{t+1} \leq n_t - \frac{n_t}{k} = n_t(1 - \frac{1}{k})$
- $n_t \leq n_0(1 - \frac{1}{k})^t < n_0(e^{-\frac{1}{k}})^t = n \cdot e^{-\frac{t}{k}}$

$e^x = \lim_{n\to\infty}(1 + \frac{x}{n})^n \geq 1 + x$, for $x \geq -1$, and when $x \neq 0$, the inequality holds

- With $t = k \ln n$ we have $n_t < 1$, by then we must have done!

# Greedy solution of Set Cover is close to optimal

- Simple greedy strategy: Keep picking the set the covers most uncovered elements, until all elements are covered.

- Theorem Suppose the optimal solution uses $k$ sets, then the greedy strategy will use at most $k \ln n$ sets.

- So the greedy strategy gives a $\ln n$ **approximation algorithm**, and it has efficient implementation. (Polynomial runtime.)

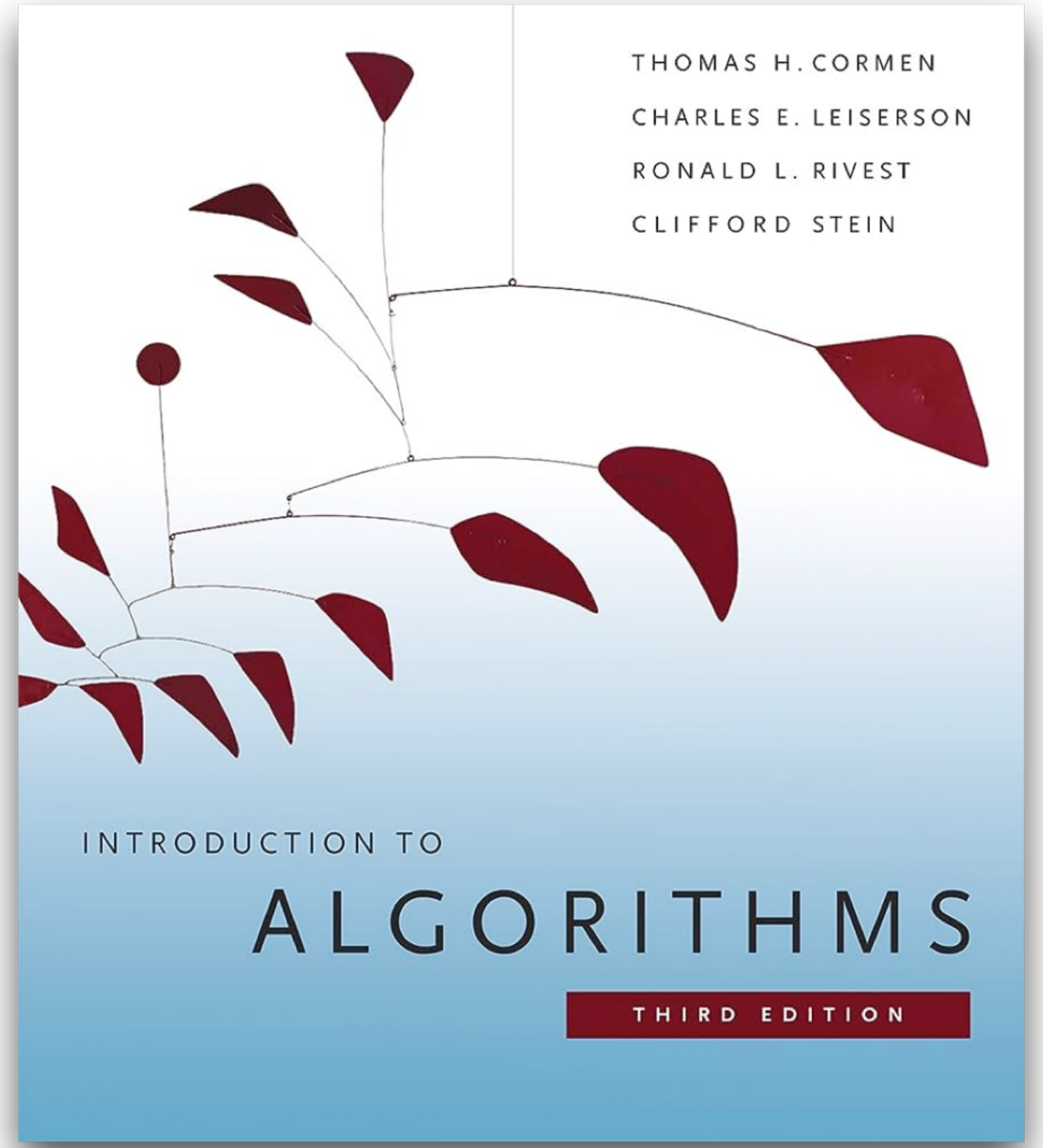
- Can we do better?
  - Most likely, **NO**! If we only care about efficient algorithms.
    - [Dinur & Steuer STOC14] There is no polynomial-runtime $\big(1 - o(1)\big) \cdot \ln n$ approximation algorithm unless **P** = **NP**.

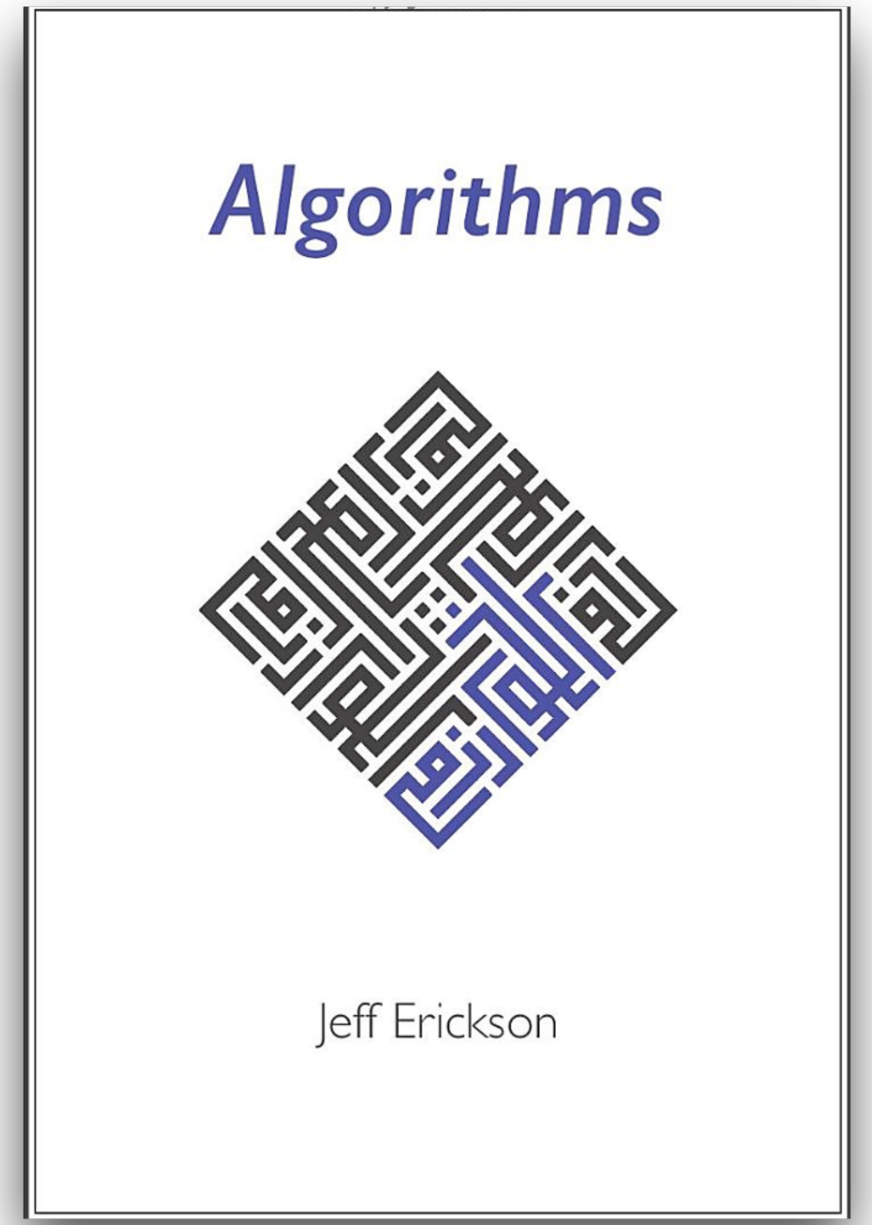
# Summary

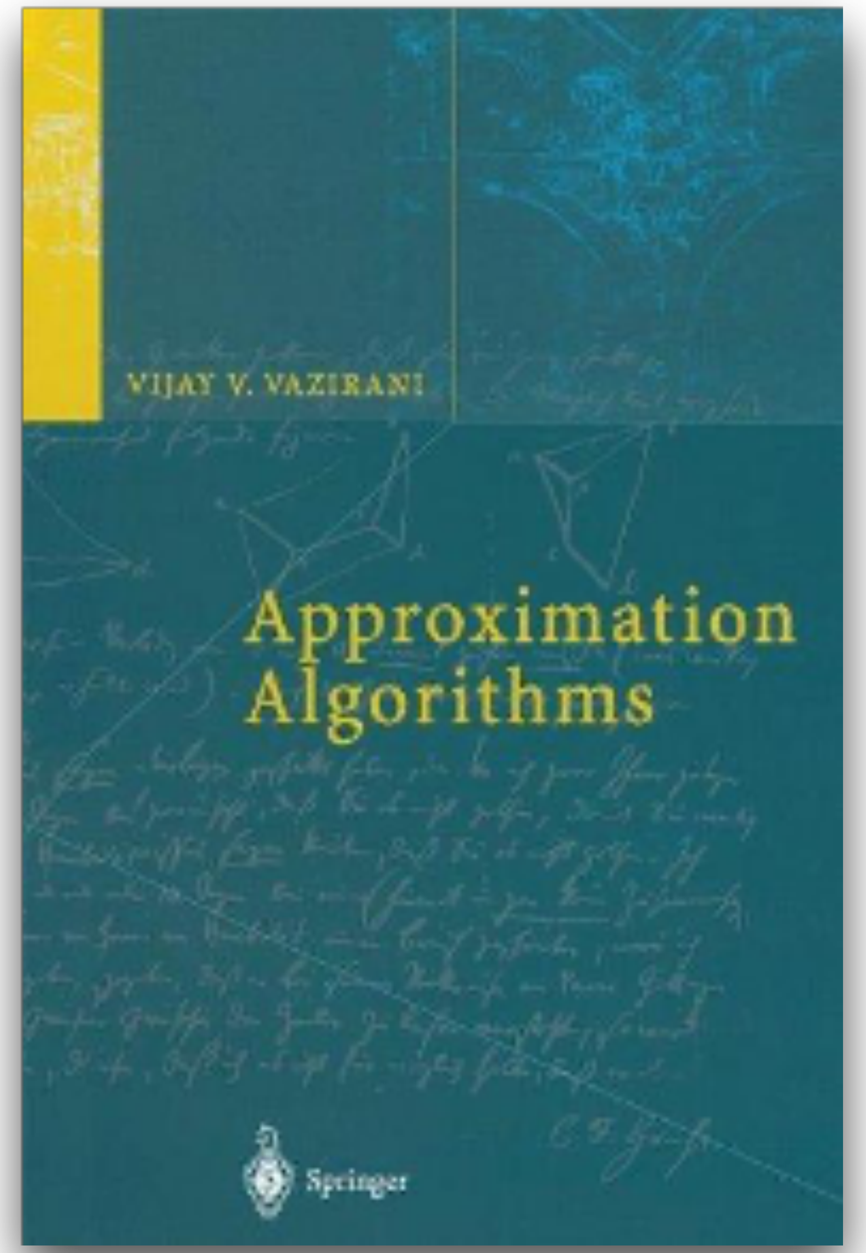
- **Basic idea of greedy strategy**: At each step when building a solution, make the choice that looks best at that moment, based on some metric.
- **Properties that make greedy strategy work**:
  - **Optimal substructure** [usually easy to prove]: optimal solution to the problem contains within it optimal solution(s) to subproblem(s).
  - **Greedy choice** [could be hard to identify and prove]: the greedy choice is contained within some optimal solution.
- Greed gives **optimal** solutions: MST, Huffman codes, …
- Greed gives **near-optimal** solutions: Set cover, …
- Greed gives **arbitrarily bad** solutions: 0-1 knapsack, …

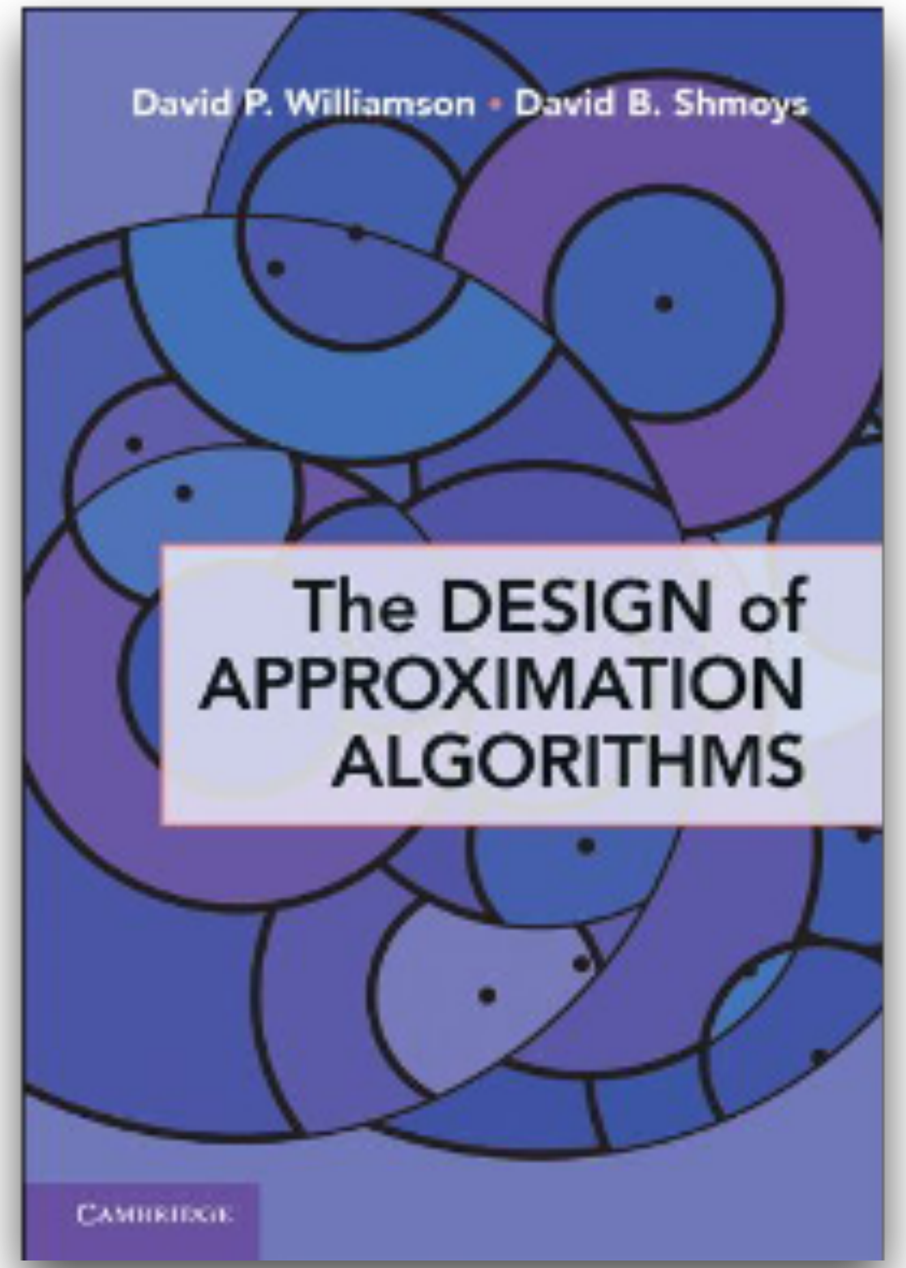
# Further reading

- [CLRS] Ch.16 (16.1-16.3, 35.3)
- [Erickson v1] Ch.4 (4.5)




Refer to [Vazirani ] and [Williamson & Shmoys] for more approximation algorithms