形式语言与计算复杂性

第 4 章 Complexity Theory

4.1 Time Complexity

黄文超

https://faculty.ustc.edu.cn/huangwenchao

$\longrightarrow$ 教学课程 $\longrightarrow$ 形式语言与计算复杂性

4.1 Time Complexity

Outline

4.1 Time Complexity

Outline

4.1 Time Complexity

1. Introduction

问: Is a problem *solvable* in *practice*, when the problem is *decidable*?
答: No, consider the case that the solution requires an inordinate amount of *time* or *memory*

问: How to deal with it?
答: Introducing the Complexity Theory

1. Introduce a way of *measuring the time* used to solve a problem
2. *Classify* problems according to the amount of *time* required
3. Discuss the possibility that *certain* decidable *problems* require *enormous amounts of time*
   - Discuss how to determine when you are faced with such a problem

4.1 Time Complexity

1. Introduction

问: Is a problem *solvable* in *practice*, when the problem is *decidable*?
答: No, consider the case that the solution requires an inordinate amount of *time* or *memory*

问: How to deal with it?
答: Introducing the Complexity Theory

1. Introduce a way of *measuring the time* used to solve a problem
2. *Classify* problems according to the amount of *time* required
3. Discuss the possibility that *certain* decidable *problems* require *enormous amounts of time*
   - Discuss how to determine when you are faced with such a problem

4.1 Time Complexity

Outline

例: How much *time* does a *single-tape* Turing machine need to decide A, where $A = \{0^k 1^k \mid k \geq 0\}$?

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

答: First introduce some *terminology* and *notation* for this purpose, e.g.,

- *worst-case analysis*: the form we consider here, we consider the *longest* running *time* of all *inputs* of a *particular length*
- *average-case analysis*: we consider the *average* of all the running *times* of *inputs* of a *particular length*

例: How much *time* does a *single-tape* Turing machine need to decide A, where $A = \{0^k 1^k \mid k \geq 0\}$?

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

答: First introduce some *terminology* and *notation* for this purpose, e.g.,

- *worst-case analysis*: the form we consider here, we consider the *longest* running *time* of all *inputs* of a *particular length*
- *average-case analysis*: we consider the *average* of all the running *times* of *inputs* of a *particular length*

定义: Time complexity, Running time

Let M be a *deterministic* Turing machine that *halts* on all inputs.

- The *running time* or *time complexity* of M: is the function $f : \mathcal{N} \to \mathcal{N}$, where $f(n)$ is the ***maximum** number* of steps that M uses on ***any** input of length n*.
- If $f(n)$ is the running time of M, we say that M *runs in time $f(n)$* and that M is an *$f(n)$ time Turing machine*.
- Customarily we use n to represent the *length of the input*.

4.1 Time Complexity

Outline

问: Then?

答: Just *estimate* the *running time*, because it is often is a complex expression.

问: How to estimate?

答: *Asymptotic (渐进) analysis*: understand the *running time* of the algorithm when it is run on *large inputs*

- considering only the *highest order term* of the *expression* for the *running time*
- *disregarding* both the *coefficient* of that term and any *lower order terms*, because the highest order term ***dominates*** the other terms on large inputs
- e.g., for $f(n) = 6n^3 + 2n^2 + 20n + 45$, the *highest order term* is $6n^3$.
  - The *asymptotic notation* or *big-O notation* for describing this relationship is $f(n) = O(n^3)$

问: Then?
答: Just *estimate* the *running time*, because it is often is a complex expression.

问: How to estimate?
答: *Asymptotic (渐进) analysis*: understand the *running time* of the algorithm when it is run on *large inputs*

- considering only the *highest order term* of the *expression* for the *running time*
- *disregarding* both the *coefficient* of that term and any *lower order terms*, because the highest order term ***dominates*** the other terms on large inputs
- e.g., for $f(n) = 6n^3 + 2n^2 + 20n + 45$, the *highest order term* is $6n^3$.
  - The *asymptotic notation* or *big-O notation* for describing this relationship is $f(n) = O(n^3)$

问: Then?
答: Just *estimate* the *running time*, because it is often is a complex expression.

问: How to estimate?
答: *Asymptotic (渐进) analysis*: understand the *running time* of the algorithm when it is run on *large inputs*

- considering only the *highest order term* of the *expression* for the *running time*
- *disregarding* both the *coefficient* of that term and any *lower order terms*, because the highest order term ***dominates*** the other terms on large inputs
- e.g., for $f(n) = 6n^3 + 2n^2 + 20n + 45$, the *highest order term* is $6n^3$.
  - The *asymptotic notation* or *big-O notation* for describing this relationship is $f(n) = O(n^3)$

问: Then?
答: Just *estimate* the *running time*, because it is often is a complex expression.

问: How to estimate?
答: *Asymptotic (渐进) analysis*: understand the *running time* of the algorithm when it is run on *large inputs*

- considering only the *highest order term* of the *expression* for the *running time*
- *disregarding* both the *coefficient* of that term and any *lower order terms*, because the highest order term ***dominates*** the other terms on large inputs
- e.g., for $f(n) = 6n^3 + 2n^2 + 20n + 45$, the *highest order term* is $6n^3$.
  - The *asymptotic notation* or *big-O notation* for describing this relationship is $f(n) = O(n^3)$

问: Then?
答: Just *estimate* the *running time*, because it is often is a complex expression.

问: How to estimate?
答: *Asymptotic (渐进) analysis*: understand the *running time* of the algorithm when it is run on *large inputs*

- considering only the *highest order term* of the *expression* for the *running time*
- *disregarding* both the *coefficient* of that term and any *lower order terms*, because the highest order term ***dominates*** the other terms on large inputs
- e.g., for $f(n) = 6n^3 + 2n^2 + 20n + 45$, the *highest order term* is $6n^3$.
  - The *asymptotic notation* or *big-O notation* for describing this relationship is $f(n) = O(n^3)$

定义: Upper bound, Big-O notation, $O(g(n))$

Let f and g be functions $f, g : \mathcal{N} \to \mathcal{R}^+$.
Say that $f(n) = O(g(n))$ if positive integers c and n_0 *exist* such that for *every integer* $n \geq n_0$,

$$f(n) \leq cg(n)$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an *upper bound* for $f(n)$, or more precisely, that $g(n)$ is an *asymptotic upper bound* for $f(n)$, to emphasize that we are suppressing constant factors.

定义: Upper bound, Big-O notation, $O(g(n))$

Let f and g be functions $f, g : \mathcal{N} \to \mathcal{R}^+$.
Say that $f(n) = O(g(n))$ if positive integers c and n_0 *exist* such that for *every integer* $n \geq n_0$,

$$f(n) \leq cg(n)$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an *upper bound* for $f(n)$, or more precisely, that $g(n)$ is an *asymptotic upper bound* for $f(n)$, to emphasize that we are suppressing constant factors.

例 1

Let $f_1(n)$ be the function $5n^3 + 2n^2 + 22n + 6$.

- $f_1(n) = O(n^3)$
- $f_1(n) = O(n^4)$
- But $f_1(n) \neq O(n^2)$

定义: Upper bound, Big-O notation, $O(g(n))$

Let f and g be functions $f, g : \mathcal{N} \to \mathcal{R}^+$.
Say that $f(n) = O(g(n))$ if positive integers c and n_0 *exist* such that for *every integer* $n \geq n_0$,

$$f(n) \leq cg(n)$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an *upper bound* for $f(n)$, or more precisely, that $g(n)$ is an *asymptotic upper bound* for $f(n)$, to emphasize that we are suppressing constant factors.

例 1

Let $f_1(n)$ be the function $5n^3 + 2n^2 + 22n + 6$.

- $f_1(n) = O(n^3)$
- $f_1(n) = O(n^4)$
- But $f_1(n) \neq O(n^2)$

定义: Upper bound, Big-O notation, $O(g(n))$

Let f and g be functions $f, g : \mathcal{N} \to \mathcal{R}^+$.
Say that $f(n) = O(g(n))$ if positive integers c and n_0 *exist* such that for *every integer* $n \geq n_0$,

$$f(n) \leq cg(n)$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an *upper bound* for $f(n)$, or more precisely, that $g(n)$ is an *asymptotic upper bound* for $f(n)$, to emphasize that we are suppressing constant factors.

例 1

Let $f_1(n)$ be the function $5n^3 + 2n^2 + 22n + 6$.

- $f_1(n) = O(n^3)$ let $c = 6$, $n_0 = 10$
- $f_1(n) = O(n^4)$
- But $f_1(n) \neq O(n^2)$

定义: Upper bound, Big-O notation, $O(g(n))$

Let f and g be functions $f, g : \mathcal{N} \to \mathcal{R}^+$.
Say that $f(n) = O(g(n))$ if positive integers c and n_0 *exist* such that for *every integer* $n \geq n_0$,

$$f(n) \leq cg(n)$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an *upper bound* for $f(n)$, or more precisely, that $g(n)$ is an *asymptotic upper bound* for $f(n)$, to emphasize that we are suppressing constant factors.

例 1

Let $f_1(n)$ be the function $5n^3 + 2n^2 + 22n + 6$.

- $f_1(n) = O(n^3)$
- $f_1(n) = O(n^4)$
- But $f_1(n) \neq O(n^2)$

定义: Upper bound, Big-O notation, $O(g(n))$

Let f and g be functions $f, g : \mathcal{N} \to \mathcal{R}^+$.
Say that $f(n) = O(g(n))$ if positive integers c and n_0 *exist* such that for *every integer* $n \geq n_0$,

$$f(n) \leq cg(n)$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an *upper bound* for $f(n)$, or more precisely, that $g(n)$ is an *asymptotic upper bound* for $f(n)$, to emphasize that we are suppressing constant factors.

例 1

Let $f_1(n)$ be the function $5n^3 + 2n^2 + 22n + 6$.

- $f_1(n) = O(n^3)$
- $f_1(n) = O(n^4)$
- But $f_1(n) \neq O(n^2)$

定义: Upper bound, Big-O notation, $O(g(n))$

Let f and g be functions $f, g : \mathcal{N} \to \mathcal{R}^+$.
Say that $f(n) = O(g(n))$ if positive integers c and n_0 *exist* such that for *every integer* $n \geq n_0$,

$$f(n) \leq cg(n)$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an *upper bound* for $f(n)$, or more precisely, that $g(n)$ is an *asymptotic upper bound* for $f(n)$, to emphasize that we are suppressing constant factors.

例 2

Let $f_2(n) = 3n \log_2 n + 5n \log_2 \log_2 n + 2$.

- $f_2(n) = O(n \log n)$ because $\log n$ *dominates* $\log \log n$

定义: Upper bound, Big-O notation, $O(g(n))$

Let f and g be functions $f, g : \mathcal{N} \to \mathcal{R}^+$.
Say that $f(n) = O(g(n))$ if positive integers c and n_0 *exist* such that for *every integer* $n \geq n_0$,

$$f(n) \leq cg(n)$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an *upper bound* for $f(n)$, or more precisely, that $g(n)$ is an *asymptotic upper bound* for $f(n)$, to emphasize that we are suppressing constant factors.

定义: Polynomial bounds, Exponential bounds

- *Polynomial bounds*: bounds of the form n^c for c greater than 0
- *Exponential bounds*: Bounds of the form $2^{(n^\delta)}$, when δ is a real number greater than 0.

定义: Upper bound, Big-O notation, $O(g(n))$

Let f and g be functions $f, g : \mathcal{N} \to \mathcal{R}^+$.
Say that $f(n) = O(g(n))$ if positive integers c and n_0 *exist* such that for *every integer* $n \geq n_0$,

$$f(n) \leq cg(n)$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an *upper bound* for $f(n)$, or more precisely, that $g(n)$ is an *asymptotic upper bound* for $f(n)$, to emphasize that we are suppressing constant factors.

对比:

- *Big-O* notation: one function is asymptotically *no more than* another
- *Small-o* notation: one function is asymptotically *less than* another
- The difference between the big-O and small-o notations is analogous to the difference between $\leq$ and $<$.

定义: Small-o notation, $o(g(n))$

Let f and g be functions $f, g : \mathcal{N} \to \mathcal{R}^+$. Say that $f(n) = o(g(n))$, if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

In other words, $f(n) = o(g(n))$ means that for *any* real number $c > 0$, a number n_0 *exists*, where $f(n) < cg(n)$ for *all* $n \geq n_0$.

定义: Small-o notation, $o(g(n))$

Let f and g be functions $f, g : \mathcal{N} \to \mathcal{R}^+$. Say that $f(n) = o(g(n))$, if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

In other words, $f(n) = o(g(n))$ means that for *any* real number $c > 0$, a number n_0 *exists*, where $f(n) < cg(n)$ for *all* $n \geq n_0$.

例:

1. $\sqrt{n} = o(n)$.
2. $n = o(n \log\log n)$.
3. $n \log\log n = o(n \log n)$.
4. $n \log n = o(n^2)$.
5. $n^2 = o(n^3)$.

However, $f(n)$ is never $o(f(n))$.

4.1 Time Complexity

Outline

原先问题: 例: How much *time* does a *single-tape* Turing machine need to decide A, where $A = \{0^k 1^k \mid k \geq 0\}$?

M_1 = "On input string w:

1. Scan across the tape and *reject* if a `0` is found to the right of a `1`.
2. Repeat if both `0`s and `1`s remain on the tape:
3. Scan across the tape, crossing off a single `0` and a single `1`.
4. If `0`s still remain after all the `1`s have been crossed off, or if `1`s still remain after all the `0`s have been crossed off, *reject*. Otherwise, if neither `0`s nor `1`s remain on the tape, *accept*."

原先问题: 例: How much *time* does a *single-tape* Turing machine need to decide A, where $A = \{0^k 1^k \mid k \geq 0\}$?

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

Stage 1:

- Performing this scan uses n steps
- Repositioning the head at the left-hand end of the tape uses another n steps.
- So, this stage uses $O(n)$ steps

原先问题: 例: How much *time* does a *single-tape* Turing machine need to decide A, where $A = \{0^k 1^k \mid k \geq 0\}$?

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

Stages 2 and 3:

- *Each scan* uses $O(n)$ steps
- Because *each scan* crosses off two symbols, at most *n/2* scans *can occur*
- So the total time taken by stages 2 and 3 is $(n/2)O(n) = O(n^2)$ steps

原先问题: 例: How much *time* does a *single-tape* Turing machine need to decide A, where $A = \{0^k 1^k \mid k \geq 0\}$?

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

Stage 4:

- The time taken in this stage is at most $O(n)$.

原先问题: 例: How much *time* does a *single-tape* Turing machine need to decide A, where $A = \{0^k 1^k \mid k \geq 0\}$?

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

The *total time* of M_1 on an input of length n is $O(n) + O(n^2) + O(n)$, or $O(n^2)$

原先问题: 例: How much *time* does a *single-tape* Turing machine need to decide A, where $A = \{0^k 1^k \mid k \geq 0\}$?

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

The *total time* of M_1 on an input of length n is $O(n) + O(n^2) + O(n)$, or $O(n^2)$

问: What can the notations be used for?
答: *Classifying* languages

定义: Time complexity class

Let $t : \mathcal{N} \to \mathcal{R}^+$ be a function.
Define the *time complexity class*, $TIME(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

So, $\underline{A} \in \text{Time}(n^2)$

问: Is there a machine that decides $\underline{A}$ asymptotically more quickly?
答: Yes, The following machine, M_2, uses a different method to decide A asymptotically faster. It shows that $\underline{A} \in \text{Time}(n \log n)$

定义: Time complexity class

Let $t : \mathcal{N} \to \mathcal{R}^+$ be a function.
Define the *time complexity class*, $TIME(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

So, $\underline{A} \in \text{Time}(n^2)$

问: Is there a machine that decides $\underline{A}$ asymptotically more quickly?
答: Yes, The following machine, M_2, uses a different method to decide A asymptotically faster. It shows that $\underline{A} \in \text{Time}(n \log n)$

定义: Time complexity class

Let $t : \mathcal{N} \to \mathcal{R}^+$ be a function.
Define the *time complexity class*, $TIME(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

So, $\underline{A} \in \text{Time}(n^2)$

问: Is there a machine that decides $\underline{A}$ asymptotically more quickly?
答: Yes, The following machine, M_2, uses a different method to decide A asymptotically faster. It shows that $\underline{A} \in \text{Time}(n \log n)$

答: M_2 that uses a different method to decide $\underline{A}$

M_2 = "On input string w:

1. Scan across the tape and *reject* if a `0` is found to the right of a `1`.
2. Repeat as long as some `0`s and some `1`s remain on the tape:
3. Scan across the tape, checking whether the total number of `0`s and `1`s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other `0` starting with the first `0`, and then crossing off every other `1` starting with the first `1`.
5. If no `0`s and no `1`s remain on the tape, *accept*. Otherwise, *reject*."

答: M_2 that uses a different method to decide $\underline{A}$

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

Stage 1: $O(n)$

答: M_2 that uses a different method to decide $\underline{A}$

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

Stage 3: $O(n)$

答: M_2 that uses a different method to decide $\underline{A}$

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

Stage 4: $O(n)$

答: M_2 that uses a different method to decide $\underline{A}$

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

Stage 2: At most $1 + \log_2 n$ iterations

- *Stage 4* crosses off at least half the 0s and 1s each time it is executed

答: M_2 that uses a different method to decide $\underline{A}$

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

Stage 2-4: $O(n \log n)$

答: M_2 that uses a different method to decide $\underline{A}$

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

Stage 5: $O(n)$

答: M_2 that uses a different method to decide $\underline{A}$

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

So $A \in TIME(n \log n)$

答: M_2 that uses a different method to decide $\underline{A}$

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

So $A \in TIME(n \log n)$

问: Can it still be optimized?

答: Yes, M_3 decides $\underline{A}$ in *linear time*, $O(n)$, by having a *second tape*.

So, $A \in TIME(n)$,

答: Yes, M_3 decides $\underline{A}$ in *linear time*, $O(n)$, by having a *second tape*.

M_3 = "On input string w:

1. Scan across tape 1 and *reject* if a `0` is found to the right of a `1`.
2. Scan across the `0`s on tape 1 until the first `1`. At the same time, copy the `0`s onto tape 2.
3. Scan across the `1`s on tape 1 until the end of the input. For each `1` read on tape 1, cross off a `0` on tape 2. If all `0`s are crossed off before all the `1`s are read, *reject*.
4. If all the `0`s have now been crossed off, *accept*. If any `0`s remain, *reject*."

答: Yes, M_3 decides $\underline{A}$ in *linear time*, $O(n)$, by having a *second tape*.

M_3 = "On input string w:

1. Scan across tape 1 and *reject* if a 0 is found to the right of a 1.
2. Scan across the 0s on tape 1 until the first 1. At the same time, copy the 0s onto tape 2.
3. Scan across the 1s on tape 1 until the end of the input. For each 1 read on tape 1, cross off a 0 on tape 2. If all 0s are crossed off before all the 1s are read, *reject*.
4. If all the 0s have now been crossed off, *accept*. If any 0s remain, *reject*."

注意: The *complexity* of A *depends on* the *model* of computation selected

- In *computability* theory: all reasonable *models* of computation are *equivalent*
- In *complexity* theory: the *choice* of *model affects* the *time complexity* of languages

4.1 Time Complexity

Outline

问: How can the *choice* of computational *model* affect the *time complexity* of languages

- single-tape Turing machine
- multitape Turing machine
- nondeterministic Turing machine.

定理: Multitape and Single-tape

定理: Non-deterministic single-tape and Deterministic single-tape

问: How can the *choice* of computational *model* affect the *time complexity* of languages

- single-tape Turing machine
- multitape Turing machine
- nondeterministic Turing machine.

定理: Multitape and Single-tape

定理: Non-deterministic single-tape and Deterministic single-tape

定理: Multitape and Single-tape

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time *multitape* Turing machine has an equivalent $O(t^2(n))$ time *single-tape* Turing machine.

定理: Multitape and Single-tape

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time *multitape* Turing machine has an equivalent $O(t^2(n))$ time *single-tape* Turing machine.

证明: (Proof by construction)

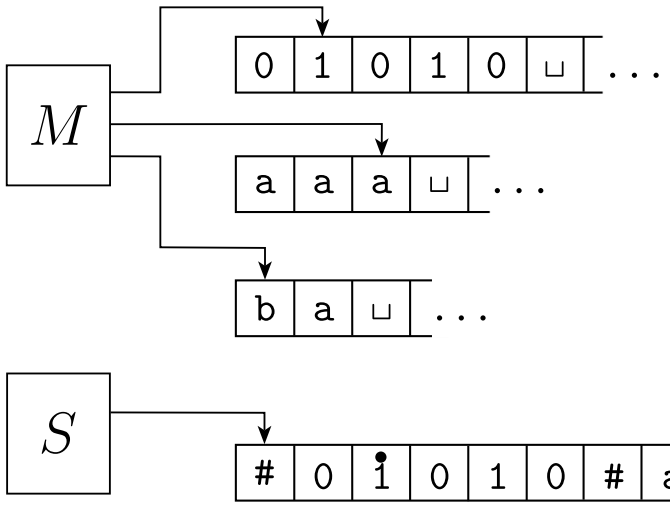
Let M be a *k-tape* TM that runs in $t(n)$ time. We *construct* a single-tape TM S that runs in $O(t^2(n))$ time.

定理: Multitape and Single-tape

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time *multitape* Turing machine has an equivalent $O(t^2(n))$ time *single-tape* Turing machine.

证明: (Proof by construction)

Let M be a *k-tape* TM that runs in $t(n)$ time. We *construct* a single-tape TM S that runs in $O(t^2(n))$ time.



For *each step* of M, machine S makes *two passes* over the active portion of its tape

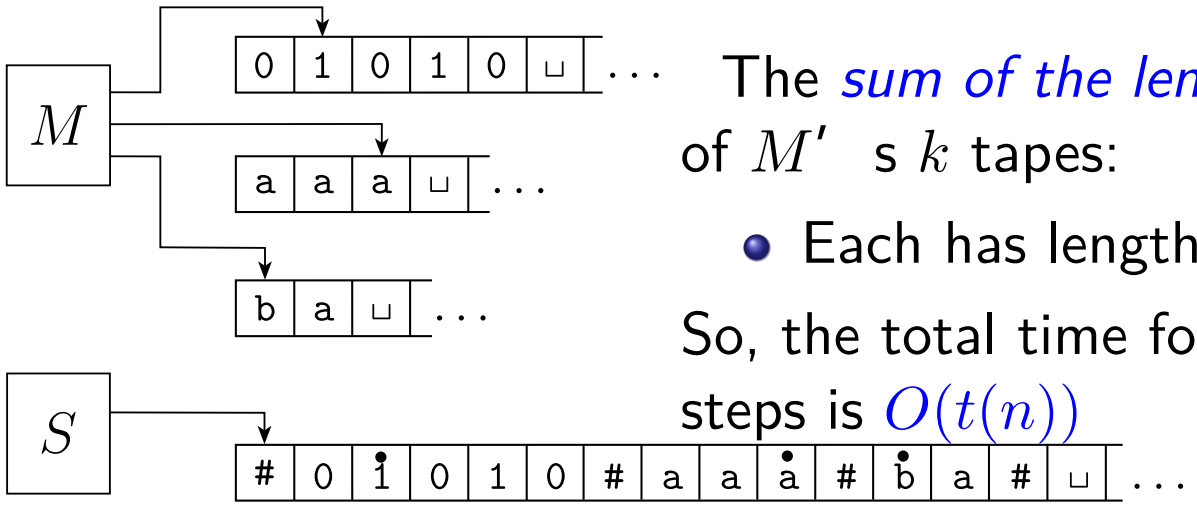
- The *first obtains* the *information* necessary to determine the next move
- The *second carries* it *out*

定理: Multitape and Single-tape

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time *multitape* Turing machine has an equivalent $O(t^2(n))$ time *single-tape* Turing machine.

证明: (Proof by construction)

Let M be a *k-tape* TM that runs in $t(n)$ time. We *construct* a single-tape TM S that runs in $O(t^2(n))$ time.



The *sum of the lengths* of the active portions of M' s k tapes:

- Each has length at most $t(n)$

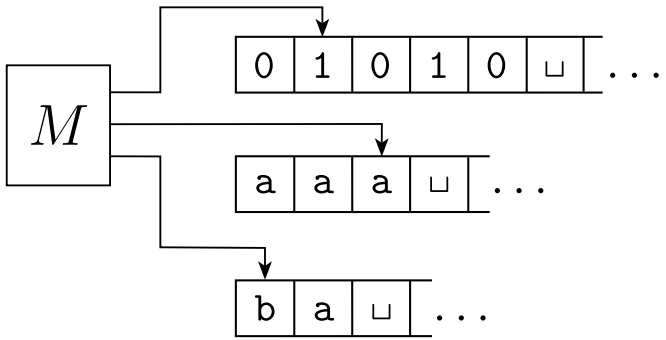
So, the total time for S to simulate *one of* M's steps is $O(t(n))$

定理: Multitape and Single-tape

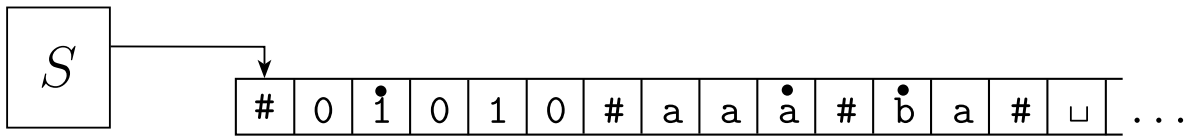
Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time *multitape* Turing machine has an equivalent $O(t^2(n))$ time *single-tape* Turing machine.

证明: (Proof by construction)

Let M be a *k-tape* TM that runs in $t(n)$ time. We *construct* a single-tape TM S that runs in $O(t^2(n))$ time.



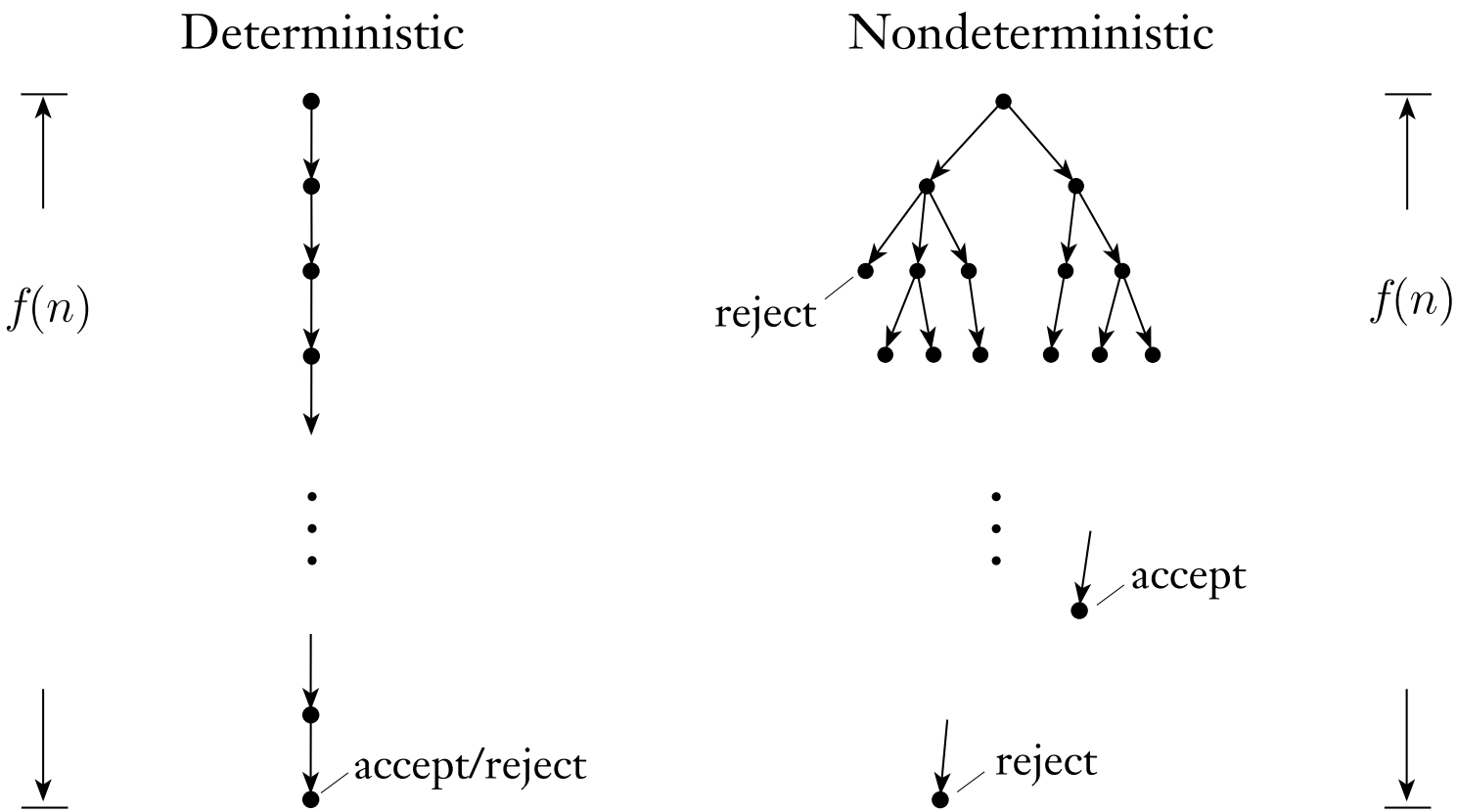
So, the part of the simulation uses $t(n) \times O(t(n)) = O(t^2(n))$ steps



定义: Running time of a nondeterministic TM

Let N be a *nondeterministic* Turing machine that is a decider. The *running time* of N is the function $f : \mathcal{N} \to \mathcal{N}$, where $f(n)$ is the *maximum number* of steps that N uses on *any branch* of its computation on *any input of length* n, as shown in the following figure.

定义: Running time of a nondeterministic TM

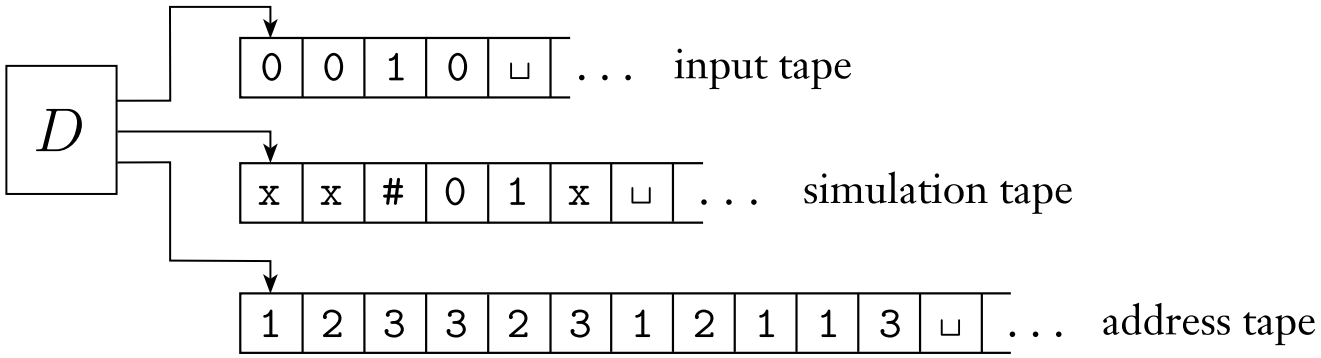


定理: Non-deterministic single-tape and Deterministic single-tape

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time *nondeterministic* single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

定理: Non-deterministic single-tape and Deterministic single-tape

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time *nondeterministic* single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

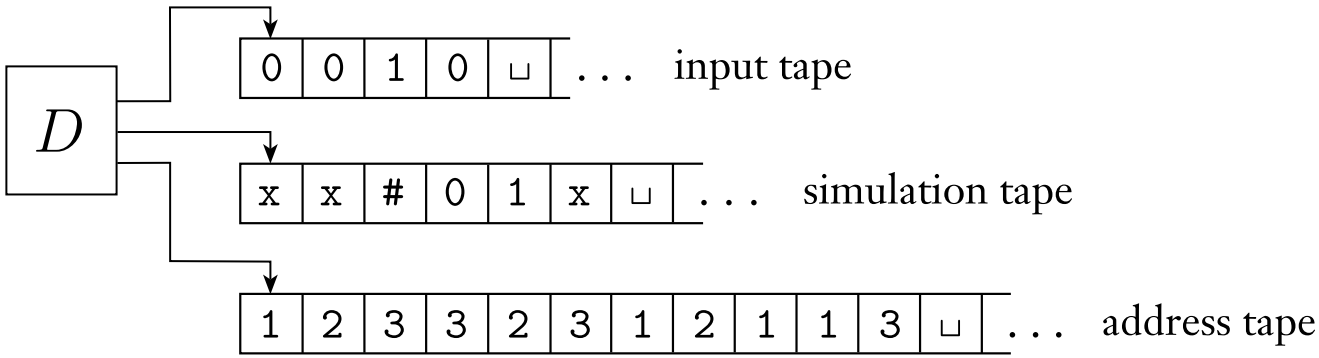


证明

Let N be a nondeterministic TM running in $t(n)$ time. We construct a deterministic TM D that simulates N

定理: Non-deterministic single-tape and Deterministic single-tape

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time *nondeterministic* single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.



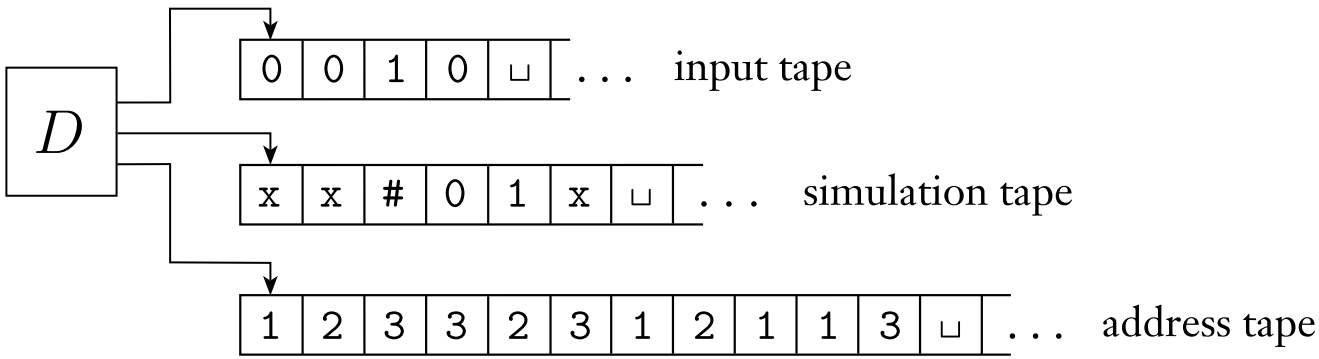
证明

Let N be a nondeterministic TM running in $t(n)$ time. We construct a deterministic TM D that simulates N

- Every branch of N' s nondeterministic computation tree has a length of at most $t(n)$.

定理: Non-deterministic single-tape and Deterministic single-tape

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time *nondeterministic* single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.



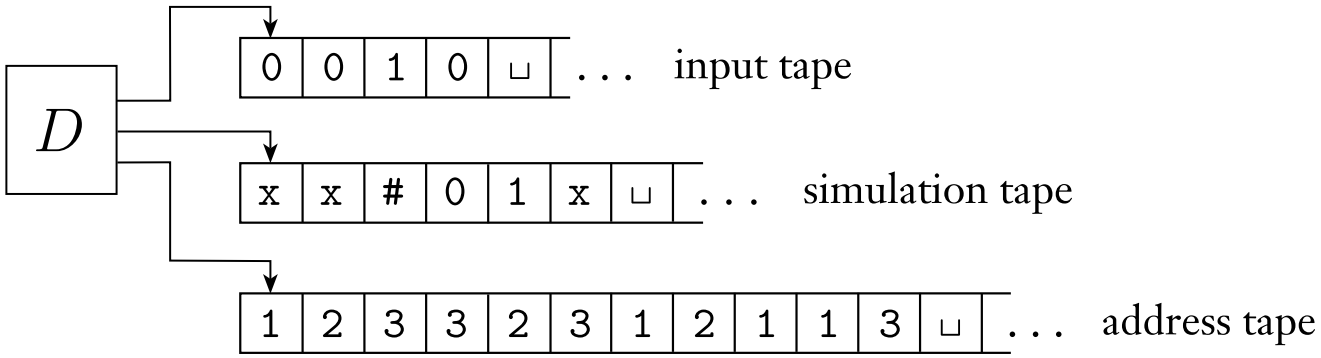
证明

Let N be a nondeterministic TM running in $t(n)$ time. We construct a deterministic TM D that simulates N

- Every node in the tree can have at most b children, where b is the maximum number of legal choices given by N's transition function.

定理: Non-deterministic single-tape and Deterministic single-tape

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time *nondeterministic* single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.



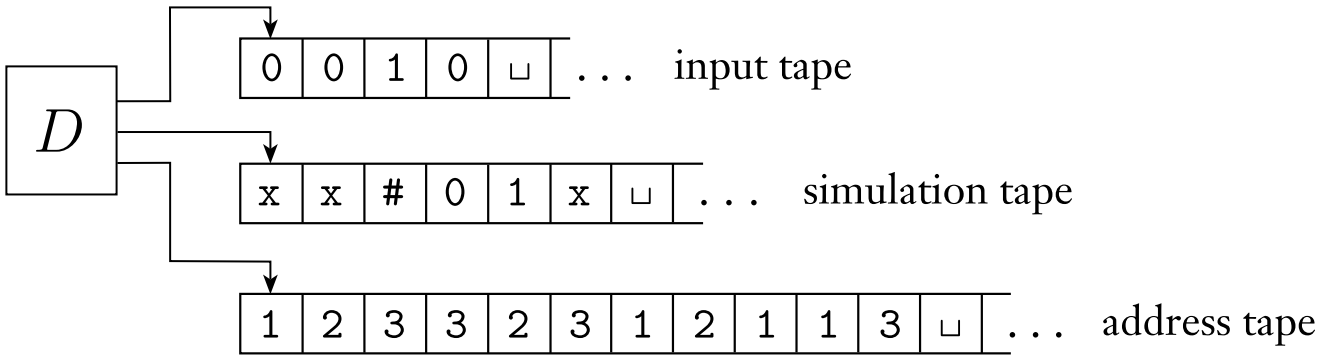
证明

Let N be a nondeterministic TM running in $t(n)$ time. We construct a deterministic TM D that simulates N

- Thus, the total number of leaves in the tree is at most $b^{t(n)}$

定理: Non-deterministic single-tape and Deterministic single-tape

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time *nondeterministic* single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.



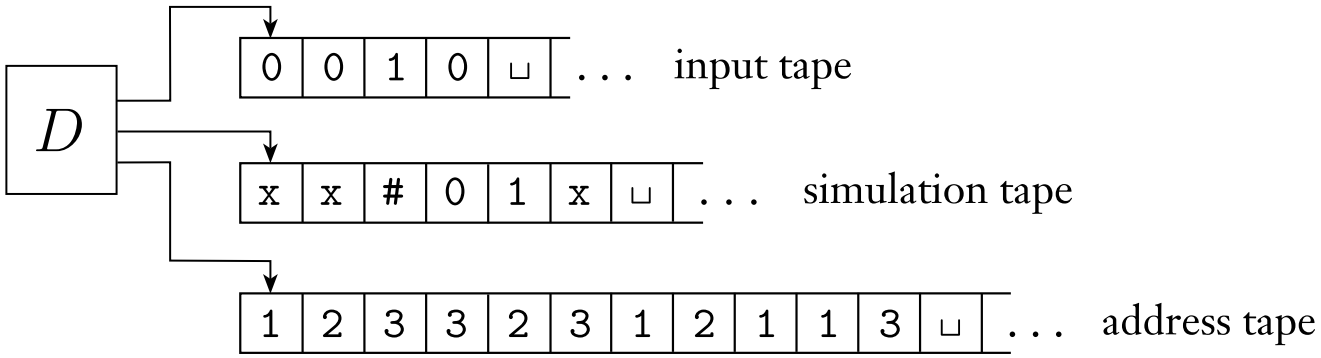
证明

Let N be a nondeterministic TM running in $t(n)$ time. We construct a deterministic TM D that simulates N

- The time it takes to start from the root and travel down to a node is $O(t(n))$

定理: Non-deterministic single-tape and Deterministic single-tape

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time *nondeterministic* single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.



证明

Let N be a nondeterministic TM running in $t(n)$ time. We construct a deterministic TM D that simulates N

- The running time of D is $O(t(n)b^{t(n)}) = 2^{O(t(n))}$

问: What do we learn from previous theorems?
答: polynomial difference and exponential difference

- *polynomial* difference: between the time complexity of problems measured on deterministic *single-tape* and *multitape* Turing machines.
- *exponential* difference: between the time complexity of problems on *deterministic* and *nondeterministic* Turing machines.

问: How to compare polynomial difference with exponential difference?
答:

- polynomial difference: small
- exponential difference: large

问: What will learn next?
答: The class P, NP, NP-completeness

问: What do we learn from previous theorems?
答: polynomial difference and exponential difference

- *polynomial* difference: between the time complexity of problems measured on deterministic *single-tape* and *multitape* Turing machines.
- *exponential* difference: between the time complexity of problems on *deterministic* and *nondeterministic* Turing machines.

问: How to compare polynomial difference with exponential difference?
答:

- polynomial difference: small
- exponential difference: large

问: What will learn next?
答: The class P, NP, NP-completeness

问: What do we learn from previous theorems?
答: polynomial difference and exponential difference

- *polynomial* difference: between the time complexity of problems measured on deterministic *single-tape* and *multitape* Turing machines.
- *exponential* difference: between the time complexity of problems on *deterministic* and *nondeterministic* Turing machines.

问: How to compare polynomial difference with exponential difference?
答:

- polynomial difference: small
- exponential difference: large

问: What will learn next?
答: The class P, NP, NP-completeness

4.1 Time Complexity

Outline

问: Why do we choose to make this separation between *polynomials* and *exponentials*?
答:

- The *dramatic difference* between the *growth rate* of typically occurring polynomials such as n^3 and typically occurring exponentials such as 2^n
- *Exponential time* algorithms typically *arise* when we solve problems by exhaustively searching through a *space of solutions*, called ***brute-force search***
- *All* reasonable *deterministic* computational models are *polynomially equivalent*
  - e.g., the deterministic single-tape and multitape Turing machine models
- Note: Our decision to *disregard* polynomial differences *doesn't imply* that we consider such differences *unimportant*.

问: Why do we choose to make this separation between *polynomials* and *exponentials*?
答:

- The *dramatic difference* between the *growth rate* of typically occurring polynomials such as n^3 and typically occurring exponentials such as 2^n
- *Exponential time* algorithms typically *arise* when we solve problems by exhaustively searching through a *space of solutions*, called ***brute-force search***
- *All* reasonable *deterministic* computational models are *polynomially equivalent*
  - e.g., the deterministic single-tape and multitape Turing machine models
- Note: Our decision to *disregard* polynomial differences *doesn't imply* that we consider such differences *unimportant*.

问: Why do we choose to make this separation between *polynomials* and *exponentials*?

答:

- The *dramatic difference* between the *growth rate* of typically occurring polynomials such as n^3 and typically occurring exponentials such as 2^n
- *Exponential time* algorithms typically *arise* when we solve problems by exhaustively searching through a *space of solutions*, called ***brute-force search***
- *All* reasonable *deterministic* computational models are *polynomially equivalent*
  - e.g., the deterministic single-tape and multitape Turing machine models
- Note: Our decision to *disregard* polynomial differences *doesn't imply* that we consider such differences *unimportant*.

问: Why do we choose to make this separation between *polynomials* and *exponentials*?

答:

- The *dramatic difference* between the *growth rate* of typically occurring polynomials such as n^3 and typically occurring exponentials such as 2^n
- *Exponential time* algorithms typically *arise* when we solve problems by exhaustively searching through a *space of solutions*, called ***brute-force search***
- *All* reasonable *deterministic* computational models are *polynomially equivalent*
  - e.g., the deterministic single-tape and multitape Turing machine models
- Note: Our decision to *disregard* polynomial differences *doesn't imply* that we consider such differences *unimportant*.

问: Why do we choose to make this separation between *polynomials* and *exponentials*?

答:

- The *dramatic difference* between the *growth rate* of typically occurring polynomials such as n^3 and typically occurring exponentials such as 2^n
- *Exponential time* algorithms typically *arise* when we solve problems by exhaustively searching through a *space of solutions*, called ***brute-force search***
- *All* reasonable *deterministic* computational models are *polynomially equivalent*
  - e.g., the deterministic single-tape and multitape Turing machine models
- Note: Our decision to *disregard* polynomial differences *doesn't imply* that we consider such differences *unimportant*.

定义: Class P, Polynomial time

P is the class of languages that are decidable in *polynomial time* on *a deterministic single-tape* Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k)$$

问: Why do we introduce class P?

答: The class P plays a central role in our theory

1. P is *invariant* for *all deterministic models* of computation that are *polynomially equivalent* to the *deterministic single-tape Turing machine*
2. P *roughly* corresponds to the class of problems that are *realistically solvable* on a computer

定义: Class P, Polynomial time

P is the class of languages that are decidable in *polynomial time* on *a deterministic single-tape* Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k)$$

问: Why do we introduce class P?

答: The class P plays a central role in our theory

1. P is *invariant* for *all deterministic models* of computation that are *polynomially equivalent* to the *deterministic single-tape Turing machine*
2. P *roughly* corresponds to the class of problems that are *realistically solvable* on a computer

问: How to *prove* that an algorithm runs in *polynomial time*?

答: 2 things

- Give a *polynomial upper bound* on the *number of stages* that the algorithm uses when it runs on an *input of length n*.
- *Examine* the *individual stages* that each can be implemented in *polynomial time* on a *reasonable deterministic model*.

Note 1: The *encoding method* used for problems.

- A *reasonable* method: allows for polynomial time *encoding* and decoding of *objects* into natural internal representations or into other reasonable encodings
  - e.g., not reasonable encoding: number 17 encoded by the unary string 11111111111111

Note 2: Reasonable encodings of graphs

- a list of its nodes and edges
- *adjacency matrix*, where the (i, j)th entry is 1 if there is an edge from node i to node j and 0 if not.

问: How to *prove* that an algorithm runs in *polynomial time*?

答: 2 things

- Give a *polynomial upper bound* on the *number of stages* that the algorithm uses when it runs on an *input of length n*.
- *Examine* the *individual stages* that each can be implemented in *polynomial time* on a *reasonable deterministic model*.

Note 1: The *encoding method* used for problems.

- A *reasonable* method: allows for polynomial time *encoding* and decoding of *objects* into natural internal representations or into other reasonable encodings
  - e.g., not reasonable encoding: number 17 encoded by the unary string 11111111111111111

Note 2: Reasonable encodings of graphs

- a list of its nodes and edges
- *adjacency matrix*, where the (i, j)th entry is 1 if there is an edge from node i to node j and 0 if not.

问: How to *prove* that an algorithm runs in *polynomial time*?

答: 2 things

- Give a *polynomial upper bound* on the *number of stages* that the algorithm uses when it runs on an *input of length n*.
- *Examine* the *individual stages* that each can be implemented in *polynomial time* on a *reasonable deterministic model*.

Note 1: The *encoding method* used for problems.

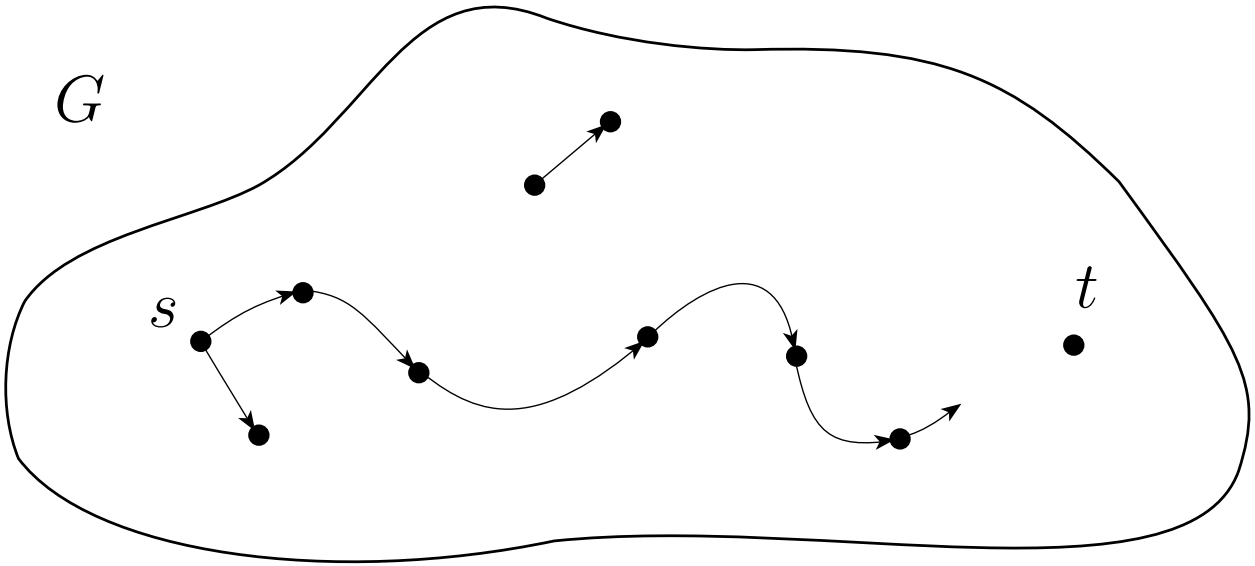
- A *reasonable* method: allows for polynomial time *encoding* and decoding of *objects* into natural internal representations or into other reasonable encodings
  - e.g., not reasonable encoding: number 17 encoded by the unary string 11111111111111

Note 2: Reasonable encodings of graphs

- a list of its nodes and edges
- *adjacency matrix*, where the (i, j)th entry is 1 if there is an edge from node i to node j and 0 if not.

定理: $PATH \in \mathrm{P}$

$PATH \in \mathrm{P}$, where $PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph that has a directed path from s to $t\}$



定理: $PATH \in$ P

$PATH \in$ P, where $PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph that has a directed path from s to $t\}$

证明: (Proof by Construction)

$M =$ "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

定理: $PATH \in$ P

$PATH \in$ P, where $PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph that has a directed path from s to $t\}$

证明: (Proof by Construction)

$M =$ "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

Assume m is the number of nodes in G

定理: $PATH \in$ P

$PATH \in$ P, where $PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph that has a directed path from s to $t\}$

证明: (Proof by Construction)

$M =$ "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

Assume m is the number of nodes in G

- *Stage 1 and 4*: executed once
- *Stage 3*: runs at most m times

定理: $PATH \in$ P

$PATH \in$ P, where $PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph that has a directed path from s to $t\}$

证明: (Proof by Construction)

$M =$ "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

Assume m is the number of nodes in G

Stages 1 and 4 of M are easily implemented in *polynomial time* on any reasonable *deterministic model*.

定理: $PATH \in$ P

$PATH \in$ P, where $PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph that has a directed path from s to $t\}$

证明: (Proof by Construction)

$M =$ "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

Assume m is the number of nodes in G

Proved. ▸ How to prove

定理: *RELPRIME* $\in$ P

RELPRIME $\in$ P,
where *RELPRIME* $= \{\langle x, y\rangle \mid x$ and y are relatively prime$\}$

定理: $RELPRIME \in$ P

$RELPRIME \in$ P,
where $RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime}\}$

证明: (Proof by Construction)

$E =$ "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

1. Repeat until $y = 0$:
2. Assign $x \leftarrow x \bmod y$.
3. Exchange x and y.
4. Output x."

Algorithm R solves $RELPRIME$, using E as a subroutine.

$R =$ "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

1. Run E on $\langle x, y\rangle$.
2. If the result is 1, *accept*. Otherwise, *reject*."

定理: *RELPRIME* $\in$ P

RELPRIME $\in$ P,
where *RELPRIME* $= \{\langle x, y\rangle \mid x$ and y are relatively prime$\}$

证明: (Proof by Construction)

$E =$ "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

1. Repeat until $y = 0$:
2. Assign $x \leftarrow x \bmod y$.
3. Exchange x and y.
4. Output x."

定理: *RELPRIME* $\in$ P

RELPRIME $\in$ P,
where *RELPRIME* $= \{\langle x, y\rangle \mid x$ and y are relatively prime$\}$

证明: (Proof by Construction)

$E =$ "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

1. Repeat until $y = 0$:
2. Assign $x \leftarrow x \bmod y$.
3. Exchange x and y.
4. Output x."

Every execution of *stage 2* (except possibly the first) cuts the value of x by *at least half*.

定理: *RELPRIME* $\in$ P

RELPRIME $\in$ P,
where *RELPRIME* $= \{\langle x, y\rangle \mid x$ and y are relatively prime$\}$

证明: (Proof by Construction)

$E =$ "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

1. Repeat until $y = 0$:
2. Assign $x \leftarrow x \bmod y$.
3. Exchange x and y.
4. Output x."

So each of the original values of x *and* y are reduced by *at least half* every other time through the loop.

定理: *RELPRIME* ∈ P

RELPRIME ∈ P,
where $RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime}\}$

证明: (Proof by Construction)

$E =$ "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

1. Repeat until $y = 0$:
2. Assign $x \leftarrow x \bmod y$.
3. Exchange x and y.
4. Output x."

Number of stages executed: $O(n)$,

- The maximum number of times that stages 2 and 3 are executed is the lesser of $2\log_2 x$ and $2\log_2 y$
- These logarithms are proportional to the lengths of the representations

定理: *RELPRIME* $\in$ P

RELPRIME $\in$ P,
where *RELPRIME* $= \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime}\}$

证明: (Proof by Construction)

$E =$ "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

1. Repeat until $y = 0$:
2. Assign $x \leftarrow x \bmod y$.
3. Exchange x and y.
4. Output x."

Each stage of E uses only *polynomial time*

定理: *RELPRIME* $\in$ P

RELPRIME $\in$ P,
where *RELPRIME* $= \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime}\}$

证明: (Proof by Construction)

$E =$ "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

1. Repeat until $y = 0$:
2. Assign $x \leftarrow x \bmod y$.
3. Exchange x and y.
4. Output x."

Proved. ▸ How to prove

定理: Every Context-free language $\in$ P

Every context-free language is a member of P

定理: Every Context-free language $\in$ P

Every context-free language is a member of P

证明: (Proof by Construction)

First of all, CFL is decidable

定理: Every Context-free language $\in$ P

Every context-free language is a member of P

证明: (Proof by Construction)

Let G be a CFG in Chomsky normal form generating the CFL L

定理: Every Context-free language $\in$ P

Every context-free language is a member of P

证明: (Proof by Construction)

Let G be a CFG in Chomsky normal form generating the CFL L

Introduce a powerful technique called *dynamic programming*

- This technique uses the *accumulation* of information about *smaller subproblems* to solve larger problems
- consider the *subproblems* of determining whether each variable in G generates each *substring* of w
- enters the solution to this subproblem in an $n \times n$ *table*
  - For $i \leq j$, the (i, j)th entry of the table contains the collection of variables that generate the substring $w_i w_{i+1} \cdots w_j$
  - For $i > j$, the table entries are unused

定理: Every Context-free language $\in$ P

Every context-free language is a member of P

证明: (Proof by Construction)

Let G be a CFG in Chomsky normal form generating the CFL L

Introduce a powerful technique called *dynamic programming*

- This technique uses the *accumulation* of information about *smaller subproblems* to solve larger problems
- consider the *subproblems* of determining whether each variable in G generates each *substring* of w
- enters the solution to this subproblem in an $n \times n$ *table*
  - For $i \leq j$, the (i, j)th entry of the table contains the collection of variables that generate the substring $w_i w_{i+1} \cdots w_j$
  - For $i > j$, the table entries are unused

定理: Every Context-free language $\in$ P

Every context-free language is a member of P

证明: (Proof by Construction)

Let G be a CFG in Chomsky normal form generating the CFL L

Introduce a powerful technique called *dynamic programming*

- This technique uses the *accumulation* of information about *smaller subproblems* to solve larger problems
- consider the *subproblems* of determining whether each variable in G generates each *substring* of w
- enters the solution to this subproblem in an $n \times n$ *table*
  - For $i \leq j$, the (i, j)th entry of the table contains the collection of variables that generate the substring $w_i w_{i+1} \cdots w_j$
  - For $i > j$, the table entries are unused

定理: Every Context-free language $\in$ P

Every context-free language is a member of P

证明: (Proof by Construction)

Let G be a CFG in Chomsky normal form generating the CFL L

Introduce a powerful technique called *dynamic programming*

- This technique uses the *accumulation* of information about *smaller subproblems* to solve larger problems
- consider the *subproblems* of determining whether each variable in G generates each *substring* of w
- enters the solution to this subproblem in an $n \times n$ *table*
  - For $i \leq j$, the (i, j)th entry of the table contains the collection of variables that generate the substring $w_i w_{i+1} \cdots w_j$
  - For $i > j$, the table entries are unused

定理: Every Context-free language $\in$ P

Every context-free language is a member of P

证明: (Proof by Construction)

Let G be a CFG in Chomsky normal form generating the CFL L

Introduce a powerful technique called *dynamic programming*

- This technique uses the *accumulation* of information about *smaller subproblems* to solve larger problems
- consider the *subproblems* of determining whether each variable in G generates each *substring* of w
- enters the solution to this subproblem in an $n \times n$ *table*
  - For $i \leq j$, the (i, j)th entry of the table contains the collection of variables that generate the substring $w_i w_{i+1} \cdots w_j$
  - For $i > j$, the table entries are unused

定理: Every Context-free language $\in$ P

Every context-free language is a member of P

证明: (Proof by Construction)

Let G be a CFG in Chomsky normal form generating the CFL L

Introduce a powerful technique called *dynamic programming*

- This technique uses the *accumulation* of information about *smaller subproblems* to solve larger problems
- consider the *subproblems* of determining whether each variable in G generates each *substring* of w
- enters the solution to this subproblem in an $n \times n$ *table*
  - For $i \leq j$, the (i, j)th entry of the table contains the collection of variables that generate the substring $w_i w_{i+1} \cdots w_j$
  - For $i > j$, the table entries are unused

定理: Every Context-free language $\in$ P

Every context-free language is a member of P

证明: (Proof by Construction)

Let G be a CFG in Chomsky normal form generating the CFL L

$D =$ "On input $w = w_1 \cdots w_n$:

1. For $w = \varepsilon$, if $S \to \varepsilon$ is a rule, *accept*; else, *reject*. [[$w = \varepsilon$ case]]
2. For $i = 1$ to n: [[examine each substring of length 1]]
3. For each variable A:
4. Test whether $A \to \mathtt{b}$ is a rule, where $\mathtt{b} = w_i$.
5. If so, place A in $table(i, i)$.

定理: Every Context-free language ∈ P

Every context-free language is a member of P

证明: (Proof by Construction)

Let G be a CFG in Chomsky normal form generating the CFL L

6. For $l = 2$ to n: $[\![l$ is the length of the substring $]\!]$

7. For $i = 1$ to $n - l + 1$: $[\![i$ is the start position of the substring $]\!]$

8. Let $j = i + l - 1$. $[\![j$ is the end position of the substring $]\!]$

9. For $k = i$ to $j - 1$: $[\![k$ is the split position $]\!]$

10. For each rule $A \to BC$:

11. If $table(i, k)$ contains B and $table(k + 1, j)$ contains C, put A in $table(i, j)$.

12. If S is in $table(1, n)$, *accept*; else, *reject*."

定理: Every Context-free language ∈ P

Every context-free language is a member of P

证明: (Proof by Construction)

Let G be a CFG in Chomsky normal form generating the CFL L

6. For $l = 2$ to n: ⟦l is the length of the substring⟧
7. For $i = 1$ to $n - l + 1$: ⟦i is the start position of the substring⟧
8. Let $j = i + l - 1$. ⟦j is the end position of the substring⟧
9. For $k = i$ to $j - 1$: ⟦k is the split position⟧
10. For each rule $A \to BC$:
11. If $table(i, k)$ contains B and $table(k + 1, j)$ contains C, put A in $table(i, j)$.
12. If S is in $table(1, n)$, *accept*; else, *reject*."

Finally, D executes $O(n^3)$ stages. Proved.

4.1 Time Complexity

Outline

问: Can we avoid brute force in certain other problems, including many interesting and useful ones?
答: Unfortunately, *polynomial time algorithms* that solve them *aren't* known to *exist*

问: How to deal with it?
答: One *remarkable discovery* concerning this question shows that the *complexities* of *many problems are linked*

- A *polynomial time algorithm* for *one such problem* can be used to solve *an entire class of problems*, i.e., *NP-complete* Problems.

问: What is *NP-complete*?
答: Let us *first* study *NP*, and then study NP-complete.

问: What is NP?
答: Let us begin with examples.

问: Can we avoid brute force in certain other problems, including many interesting and useful ones?
答: Unfortunately, *polynomial time algorithms* that solve them *aren't* known to *exist*

问: How to deal with it?
答: One *remarkable discovery* concerning this question shows that the *complexities* of *many problems are linked*

- A *polynomial time algorithm* for *one such problem* can be used to solve *an entire class of problems*, i.e., *NP-complete* Problems.

问: What is *NP-complete*?
答: Let us *first* study *NP*, and then study NP-complete.

问: What is NP?
答: Let us begin with examples.

问: Can we avoid brute force in certain other problems, including many interesting and useful ones?
答: Unfortunately, *polynomial time algorithms* that solve them *aren't* known to *exist*

问: How to deal with it?
答: One *remarkable discovery* concerning this question shows that the *complexities* of *many problems are linked*

- A *polynomial time algorithm* for *one such problem* can be used to solve *an entire class of problems*, i.e., *NP-complete* Problems.

问: What is *NP-complete*?
答: Let us *first* study *NP*, and then study NP-complete.

问: What is NP?
答: Let us begin with examples.

问: Can we avoid brute force in certain other problems, including many interesting and useful ones?
答: Unfortunately, *polynomial time algorithms* that solve them *aren't* known to *exist*

问: How to deal with it?
答: One *remarkable discovery* concerning this question shows that the *complexities* of *many problems are linked*

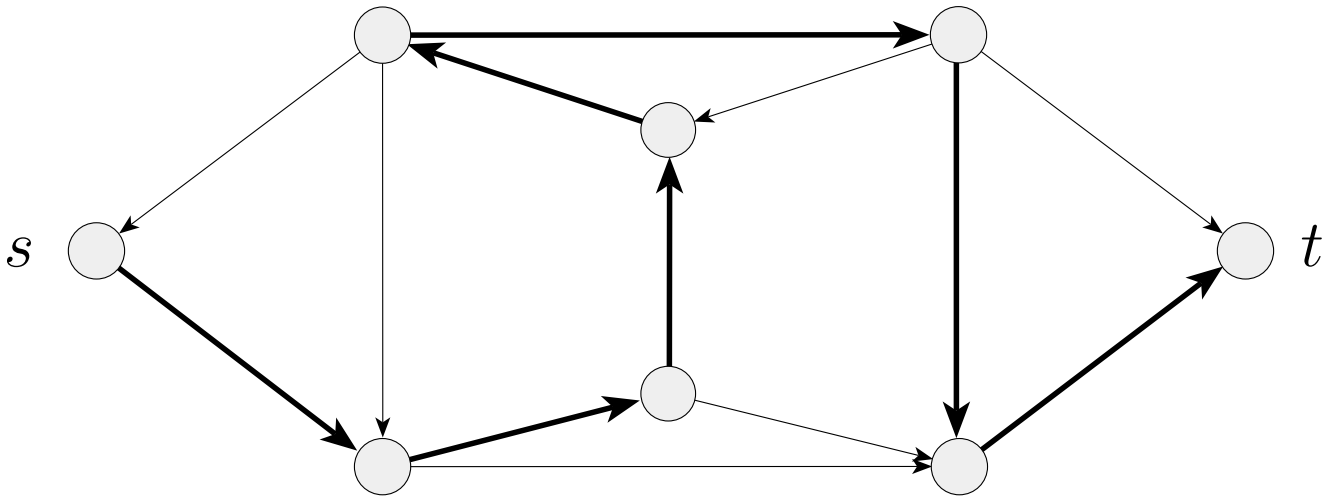
- A *polynomial time algorithm* for *one such problem* can be used to solve *an entire class of problems*, i.e., *NP-complete* Problems.

问: What is *NP-complete*?
答: Let us *first* study *NP*, and then study NP-complete.

问: What is NP?
答: Let us begin with examples.

定义: Hamiltonian path

A *Hamiltonian path* in a directed graph G is a directed *path* that goes through each *node exactly once*.



例 1: HAMPATH

HAMPATH $=\{\langle G, s, t\rangle \mid G$ is a directed graph with a Hamiltonian path from s to $t\}$

分析: The *HAMPATH* problem has a feature called *polynomial verifiability*

- *Bad news*: we *don't know* of a fast (i.e., *polynomial time*) way *to determine* whether a graph contains a Hamiltonian path
- *Good news*: *if* such *a path were discovered* somehow (perhaps using the exponential time algorithm), we could *easily convince* someone else of its existence *simply by presenting it*.
- In conclusion: *verifying* the existence of a Hamiltonian path may be *much easier* than *determining* its existence.

例 1: HAMPATH

HAMPATH $=\{\langle G, s, t\rangle \mid G$ is a directed graph with a Hamiltonian path from s to $t\}$

分析: The *HAMPATH* problem has a feature called *polynomial verifiability*

- *Bad news*: we *don't know* of a fast (i.e., *polynomial time*) way *to determine* whether a graph contains a Hamiltonian path
- *Good news*: *if* such *a path were discovered* somehow (perhaps using the exponential time algorithm), we could *easily convince* someone else of its existence *simply by presenting it*.
- In conclusion: *verifying* the existence of a Hamiltonian path may be *much easier* than *determining* its existence.

例 2: *COMPOSITES*

COMPOSITES $= \{x \mid x = pq, \text{ for integers } p, q > 1\}$

分析: *COMPOSITES* is also *polynomial verifiable*

Note: Some problems are even *not polynomial verifiable*

- $\overline{\textit{COMPOSITES}}$

例 2: *COMPOSITES*

$COMPOSITES = \{x \mid x = pq, \text{ for integers } p, q > 1\}$

分析: *COMPOSITES* is also *polynomial verifiable*

Note: Some problems are even *not polynomial verifiable*

- $\overline{COMPOSITES}$

例 2: *COMPOSITES*

$COMPOSITES = \{x \mid x = pq, \text{ for integers } p, q > 1\}$

分析: *COMPOSITES* is also *polynomial verifiable*

Note: Some problems are even *not polynomial verifiable*

- $\overline{COMPOSITES}$

定义: Verifier

A *verifier* for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$$

We measure the time of a verifier only in terms of the length of w

- A *polynomial time verifier* runs in polynomial time in the length of w
- A language A is *polynomially verifiable* if it has a polynomial time verifier
- This information c is called a *certificate*, or *proof*, of *membership in A*.

分析: Certificate

- For the *HAMPATH* problem, a *certificate* for a string $\langle G, s, t \rangle \in$ *HAMPATH* simply is *a Hamiltonian path* from s to t.
- For the *COMPOSITES* problem, a *certificate* for the composite number x simply is one of its *divisors*.

定义: Verifier

A *verifier* for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$$

We measure the time of a verifier only in terms of the length of w

- A *polynomial time verifier* runs in polynomial time in the length of w
- A language A is *polynomially verifiable* if it has a polynomial time verifier
- This information c is called a *certificate*, or *proof*, of *membership in A*.

分析: Certificate

- For the *HAMPATH* problem, a *certificate* for a string $\langle G, s, t \rangle \in$ *HAMPATH* simply is *a Hamiltonian path* from s to t.
- For the *COMPOSITES* problem, a *certificate* for the composite number x simply is one of its *divisors*.

定义: Verifier

A *verifier* for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$$

We measure the time of a verifier only in terms of the length of w

- A *polynomial time verifier* runs in polynomial time in the length of w
- A language A is *polynomially verifiable* if it has a polynomial time verifier
- This information c is called a *certificate*, or *proof*, of *membership in A*.

分析: Certificate

- For the *HAMPATH* problem, a *certificate* for a string $\langle G, s, t \rangle \in$ *HAMPATH* simply is *a Hamiltonian path* from s to t.
- For the *COMPOSITES* problem, a *certificate* for the composite number x simply is one of its *divisors*.

定义: NP

NP is the *class of languages* that have *polynomial time verifiers*

分析:

- Both *HAMPATH* and *COMPOSITES* are members of NP
- Problems in NP are sometimes called *NP-problems*

问: What does the name NP come from?
答: *Nondeterministic polynomial time*
问: Why?
答: Decides the *HAMPATH* problem in *nondeterministic polynomial time*

定义: NP

NP is the *class of languages* that have *polynomial time verifiers*

分析:

- Both *HAMPATH* and *COMPOSITES* are members of NP
- Problems in NP are sometimes called *NP-problems*

问: What does the name NP come from?
答: *Nondeterministic polynomial time*
问: Why?
答: Decides the *HAMPATH* problem in *nondeterministic polynomial time*

定义: NP

NP is the *class of languages* that have *polynomial time verifiers*

分析:

- Both *HAMPATH* and *COMPOSITES* are members of NP
- Problems in NP are sometimes called *NP-problems*

问: What does the name NP come from?
答: *Nondeterministic polynomial time*
问: Why?
答: Decides the *HAMPATH* problem in *nondeterministic polynomial time*

定义: NP

NP is the *class of languages* that have *polynomial time verifiers*

分析:

- Both *HAMPATH* and *COMPOSITES* are members of NP
- Problems in NP are sometimes called *NP-problems*

问: What does the name NP come from?

答: *Nondeterministic polynomial time*

问: Why?

答: Decides the *HAMPATH* problem in *nondeterministic polynomial time*

N_1 = "On input $\langle G, s, t \rangle$, where G is a directed graph with nodes s and t:

1. Write a list of m numbers, $p_1, \ldots, p_m$, where m is the number of nodes in G. Each number in the list is nondeterministically selected to be between 1 and m.
2. Check for repetitions in the list. If any are found, *reject*.
3. Check whether $s = p_1$ and $t = p_m$. If either fail, *reject*.
4. For each i between 1 and $m - 1$, check whether (p_i, p_{i+1}) is an edge of G. If any are not, *reject*. Otherwise, all tests have been passed, so *accept*."

定理

A language is in NP, iff
it is *decided* by some *nondeterministic polynomial time* Turing machine.

定理

A language is in NP, iff
it is *decided* by some *nondeterministic polynomial time* Turing machine.

证明: (Proof by Construction)

定理

A language is in NP, iff
it is *decided* by some *nondeterministic polynomial time* Turing machine.

证明: (Proof by Construction)

(1) *Forward direction*: let $A \in$ NP and show that A is decided by a polynomial time NTM N

定理

A language is in NP, iff
it is *decided* by some *nondeterministic polynomial time* Turing machine.

证明: (Proof by Construction)

(1) *Forward direction*: let $A \in$ NP and show that A is decided by a polynomial time NTM N
Let V be the polynomial time verifier for A.

定理

A language is in NP, iff
it is *decided* by some *nondeterministic polynomial time* Turing machine.

证明: (Proof by Construction)

(1) *Forward direction*: let $A \in$ NP and show that A is decided by a polynomial time NTM N
Let V be the polynomial time verifier for A.
Assume that V is a TM that runs in time n^k and construct N as follows

定理

A language is in NP, iff
it is *decided* by some *nondeterministic polynomial time* Turing machine.

证明: (Proof by Construction)

(1) *Forward direction*: let $A \in$ NP and show that A is decided by a polynomial time NTM N

Let V be the polynomial time verifier for A.

Assume that V is a TM that runs in time n^k and construct N as follows

$N =$ "On input w of length n:

1. Nondeterministically select string c of length at most n^k.
2. Run V on input $\langle w, c \rangle$.
3. If V accepts, *accept*; otherwise, *reject*."

定理

A language is in NP, iff
it is *decided* by some *nondeterministic polynomial time* Turing machine.

证明: (Proof by Construction)

(2) *Reverse direction*: Assume that A is decided by a polynomial time NTM N and construct a polynomial time verifier V as follows:

定理

A language is in NP, iff
it is *decided* by some *nondeterministic polynomial time* Turing machine.

证明: (Proof by Construction)

(2) *Reverse direction*: Assume that A is decided by a polynomial time NTM N and construct a polynomial time verifier V as follows:
$V =$ "On input $\langle w, c \rangle$, where w and c are strings:

1. Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice to make at each step (as in the proof of Theorem 3.16).
2. If this branch of N's computation accepts, *accept*; otherwise, *reject*."

定义: NTIME

NTIME$(t(n)) = \{L \mid L$ is a language decided by an $O(t(n))$ time *nondeterministic Turing machine*$\}$

推论

NP $= \bigcup_k$ NTIME (n^k)

问: Any more problems in NP?
答: Yes

- *CLIQUE*
- *SUBSET-SUM*

定义: NTIME

NTIME$(t(n)) = \{L \mid L$ is a language decided by an $O(t(n))$ time *nondeterministic Turing machine*$\}$

推论

NP $= \bigcup_k$ NTIME (n^k)

问: Any more problems in NP?
答: Yes

- *CLIQUE*
- *SUBSET-SUM*

定义: NTIME

NTIME$(t(n)) = \{L \mid L$ is a language decided by an $O(t(n))$ time *nondeterministic Turing machine*$\}$

推论

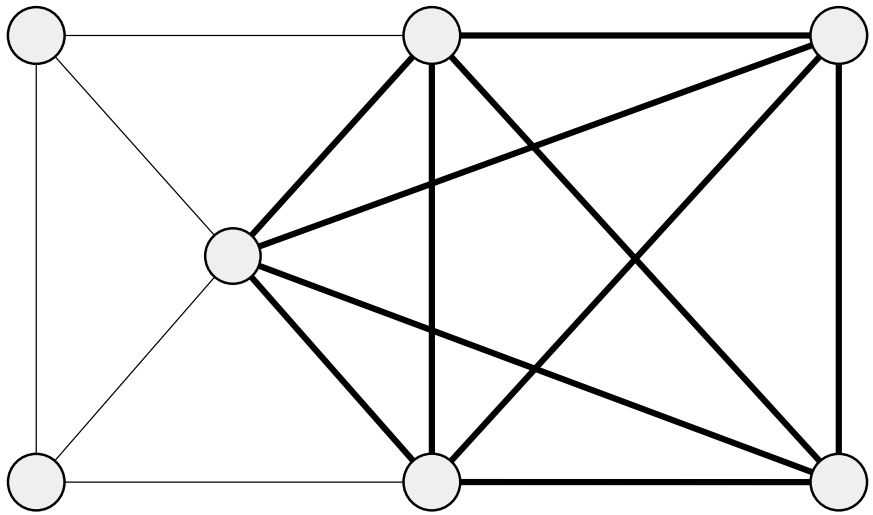
NP $= \bigcup_k$ NTIME (n^k)

问: Any more problems in NP?
答: Yes

- *CLIQUE*
- *SUBSET-SUM*

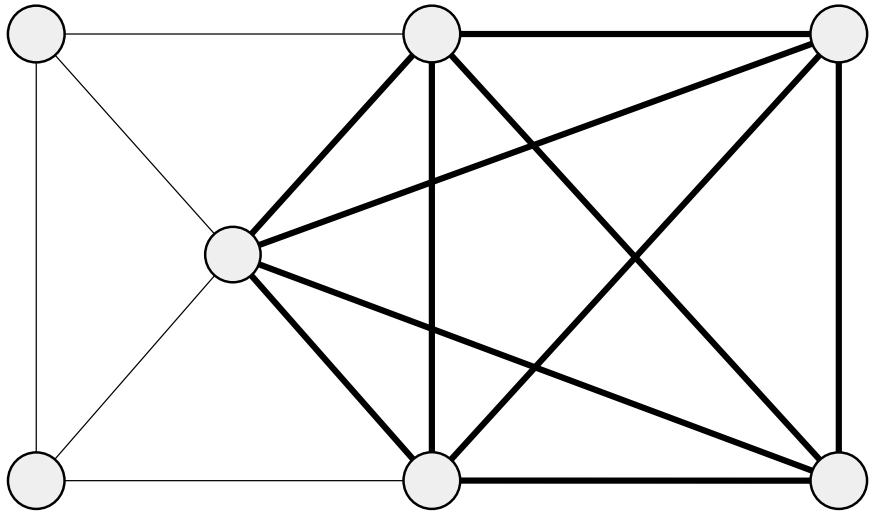
定义: Clique (团)

- A *clique* in an undirected graph is a subgraph, wherein every *two nodes are connected by an edge*.
- A *k-clique* is a clique that contains k nodes.



定义: Clique (团)

- A *clique* in an undirected graph is a subgraph, wherein every *two nodes are connected by an edge*.
- A *k-clique* is a clique that contains k nodes.



定理: *CLIQUE* is in NP

CLIQUE is in NP, where

$$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique } \}$$

定理: *CLIQUE* is in NP

CLIQUE is in NP, where

$$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique } \}$$

证明方法 1: (Proof by Construction)

The following is a *verifier* V for *CLIQUE*:

$V =$ "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a subgraph with k nodes in G.
2. Test whether G contains all edges connecting nodes in c.
3. If both pass, *accept*; otherwise, *reject*."

定理: *CLIQUE* is in NP

CLIQUE is in NP, where

$$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique } \}$$

证明方法 2: (Proof by Construction)

The following is a *nondeterministic TM* N for *CLIQUE*:

$N =$ "On input $\langle G, k\rangle$, where G is a graph:

1. Nondeterministically select a subset c of k nodes of G.
2. Test whether G contains all edges connecting nodes in c.
3. If yes, *accept*; otherwise, *reject*."

定理: *SUBSET-SUM* is in NP

SUBSET-SUM is in NP, where
SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}$, and for some $\{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}$, we have $\sum y_i = t\}$

定理: *SUBSET-SUM* is in NP

SUBSET-SUM is in NP, where
SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}$, and for some $\{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}$, we have $\sum y_i = t\}$

例

$\langle\{4, 11, 16, 21, 27\}, 25\rangle \in$ *SUBSET-SUM* because $4 + 21 = 25$

定理: *SUBSET-SUM* is in NP

SUBSET-SUM is in NP, where
SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}$, and for some $\{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}$, we have $\sum y_i = t\}$

例

$\langle\{4, 11, 16, 21, 27\}, 25\rangle \in$ *SUBSET-SUM* because $4 + 21 = 25$

证明方法 1: (Construction by Proof)

$V =$ "On input $\langle\langle S, t\rangle, c\rangle$:

1. Test whether c is a collection of numbers that sum to t.
2. Test whether S contains all the numbers in c.
3. If both pass, *accept*; otherwise, *reject*."

定理: *SUBSET-SUM* is in NP

SUBSET-SUM is in NP, where
SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}$, and for some $\{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}$, we have $\sum y_i = t\}$

例

$\langle\{4, 11, 16, 21, 27\}, 25\rangle \in$ *SUBSET-SUM* because $4 + 21 = 25$

证明方法 2: (Construction by Proof)

$N =$ "On input $\langle S, t\rangle$:

1. Nondeterministically select a subset c of the numbers in S.
2. Test whether c is a collection of numbers that sum to t.
3. If the test passes, *accept*; otherwise, *reject*."

问: Comprehensively compare P and NP?

答:

- P = the class of languages for which membership can be *decided* quickly.
- NP = the class of languages for which membership can be *verified* quickly.

问: Whether P = NP?

答: One of the greatest *unsolved problems* in theoretical computer science and contemporary mathematics

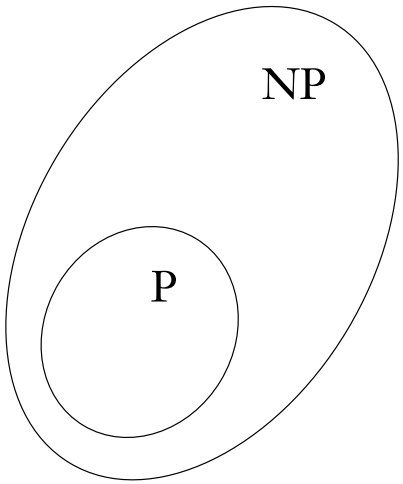
问: Comprehensively compare P and NP?

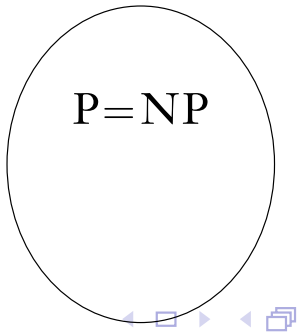
答:

- P = the class of languages for which membership can be *decided* quickly.
- NP = the class of languages for which membership can be *verified* quickly.

问: Whether P = NP?

答: One of the greatest *unsolved problems* in theoretical computer science and contemporary mathematics





问: Why?
答: We are *unable* to *prove* the existence of a *single* language in *NP* that is *not in P*

问: What can we prove?
答: Deciding languages in NP uses exponential time

$$\mathrm{NP} \subseteq \mathrm{EXPTIME} = \bigcup_k \mathrm{TIME}(2^{n^k})$$

问: Is *every* NP problem hard?
答: No

问: What is hard?
答: A certain subset of NP problems, called *NP-complete* problems

问: Why?
答: We are *unable* to *prove* the existence of a *single* language in *NP* that is *not in P*

问: What can we prove?
答: Deciding languages in NP uses exponential time

$$\mathrm{NP} \subseteq \mathrm{EXPTIME} = \bigcup_k \mathrm{TIME}(2^{n^k})$$

问: Is *every* NP problem hard?
答: No

问: What is hard?
答: A certain subset of NP problems, called *NP-complete* problems

问: Why?
答: We are *unable* to *prove* the existence of a *single* language in *NP* that is *not in P*

问: What can we prove?
答: Deciding languages in NP uses exponential time

$$\mathrm{NP} \subseteq \mathrm{EXPTIME} = \bigcup_k \mathrm{TIME}(2^{n^k})$$

问: Is *every* NP problem hard?
答: No

问: What is hard?
答: A certain subset of NP problems, called *NP-complete* problems

问: Why?
答: We are *unable* to *prove* the existence of a *single* language in *NP* that is *not in P*

问: What can we prove?
答: Deciding languages in NP uses exponential time

$$\mathrm{NP} \subseteq \mathrm{EXPTIME} = \bigcup_k \mathrm{TIME}(2^{n^k})$$

问: Is *every* NP problem hard?
答: No

问: What is hard?
答: A certain subset of NP problems, called *NP-complete* problems

Outline

问: What is NP-complete?
答: If a *polynomial time* algorithm exists for *any of* these problems in *NP-complete*, *all problems* in *NP* would be *polynomial* time solvable

- *theoretical* side: a researcher trying to show that P is unequal to NP may *focus on an NP-complete problem*
- *practical* side: *prevent wasting time searching* for a nonexistent polynomial time algorithm to solve a particular problem

问: An example of problem in NP-complete?
答: SAT in Boolean Logic

定义: Boolean formula, Satisfiable

A *Boolean formula* is an expression involving Boolean variables and operations.

问: What is NP-complete?
答: If a *polynomial time* algorithm exists for *any of* these problems in *NP-complete*, *all problems* in *NP* would be *polynomial* time solvable

- *theoretical* side: a researcher trying to show that P is unequal to NP may *focus on an NP-complete problem*
- *practical* side: *prevent wasting time searching* for a nonexistent polynomial time algorithm to solve a particular problem

问: An example of problem in NP-complete?
答: SAT in Boolean Logic

定义: Boolean formula, Satisfiable

A *Boolean formula* is an expression involving Boolean variables and operations.

问: What is NP-complete?
答: If a *polynomial time* algorithm exists for *any of* these problems in *NP-complete*, *all problems* in *NP* would be *polynomial* time solvable

- *theoretical* side: a researcher trying to show that P is unequal to NP may *focus on an NP-complete problem*
- *practical* side: *prevent wasting time searching* for a nonexistent polynomial time algorithm to solve a particular problem

问: An example of problem in NP-complete?
答: SAT in Boolean Logic

定义: Boolean formula, Satisfiable

A *Boolean formula* is an expression involving Boolean variables and operations.

问: What is NP-complete?
答: If a *polynomial time* algorithm exists for *any of* these problems in *NP-complete*, *all problems* in *NP* would be *polynomial* time solvable

- *theoretical* side: a researcher trying to show that P is unequal to NP may *focus on an NP-complete problem*
- *practical* side: *prevent wasting time searching* for a nonexistent polynomial time algorithm to solve a particular problem

问: An example of problem in NP-complete?
答: SAT in Boolean Logic

定义: Boolean formula, Satisfiable

A *Boolean formula* is an expression involving Boolean variables and operations.

定义: Satisfiability problem, SAT

$\mathrm{SAT} = \{\langle\varphi\rangle \mid \varphi \text{ is a satisfiable Boolean formula}\}$

定理

$\mathrm{SAT} \in \mathrm{P}$ iff $\mathrm{P} = \mathrm{NP}$

问: How to prove the theorem?
答: 见下页

定义: Satisfiability problem, SAT

$\mathrm{SAT} = \{\langle\varphi\rangle \mid \varphi \text{ is a satisfiable Boolean formula}\}$

定理

$\mathrm{SAT} \in \mathrm{P}$ iff $\mathrm{P} = \mathrm{NP}$

问: How to prove the theorem?
答: 见下页

定义: Satisfiability problem, SAT

$\mathrm{SAT} = \{\langle\varphi\rangle \mid \varphi \text{ is a satisfiable Boolean formula}\}$

定理

$\mathrm{SAT} \in \mathrm{P}$ iff $\mathrm{P} = \mathrm{NP}$

问: How to prove the theorem?
答: 见下页

Outline

回顾: Map-reducibility

回顾: Map-reducibility

定义: Polynomial time computable function

A function $f : \Sigma^* \to \Sigma^*$ is a *polynomial time computable function* if some polynomial time Turing machine M exists that *halts* with just $f(w)$ on its tape, when started on *any input* w.

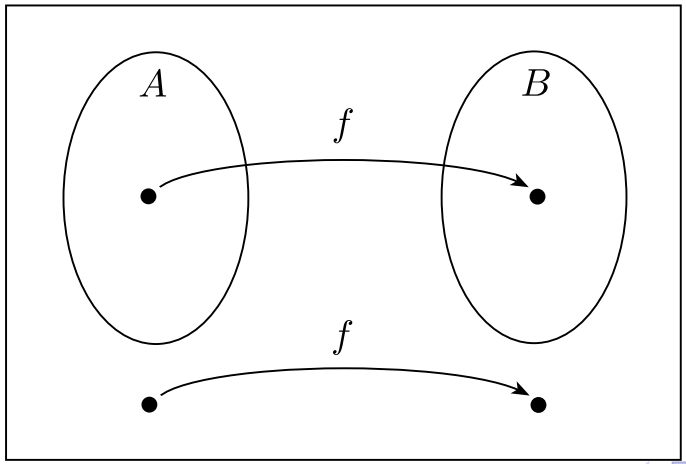
回顾: Map-reducibility

定义: Polynomial Time Reducibility

Language A is *polynomial time mapping reducible*, or simply *polynomial time reducible*, to language B, written $A \leq_{\mathrm{P}} B$, if a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ exists, where for every w,

$$w \in A \Leftrightarrow f(w) \in B$$

The function f is called the *polynomial time reduction* of A to B



定理

If $A \leq_{\mathrm{P}} B$ and $B \in P$,then $A \in P$

证明

- Let M be the polynomial time algorithm deciding B
- Let f be the polynomial time reduction from A to B
- We describe a polynomial time algorithm N deciding A as follows.

$N =$ "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

定理

If $A \leq_{\mathrm{P}} B$ and $B \in P$,then $A \in P$

证明

- Let M be the polynomial time algorithm deciding B
- Let f be the polynomial time reduction from A to B
- We describe a polynomial time algorithm N deciding A as follows.

$N =$ "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

定义: 3cnf-formula

- *Literal*: a *Boolean variable* or a *negated Boolean variable*, as in x or $\overline{x}$
- *Clause*: several literals connected with $\vee$s, as in $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$
- *cnf-formula*: A Boolean formula is in *conjunctive normal form*, called a *cnf-formula*, if it comprises several clauses connected with $\wedge$s, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee x_5 \vee x_6) \wedge (x_3 \vee x_6)$$

- *3cnf-formula*: if all the clauses have three literals, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee x_5 \vee x_6) \wedge (\overline{x_2} \vee x_3 \vee x_6) \wedge (\overline{x_4} \vee x_5 \vee x_6)$$

定义: *3SAT*

$3SAT = \{\langle\varphi\rangle \mid \varphi \text{ is a satisfiable 3cnf-formula}\}$

- A Boolean formula is *satisfiable* if *some assignment* of 0s and 1s to the variables makes the *formula evaluate to 1*.

定义: 3cnf-formula

- *Literal*: a *Boolean variable* or a *negated Boolean variable*, as in x or $\overline{x}$
- *Clause*: several literals connected with $\vee$s, as in $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$
- *cnf-formula*: A Boolean formula is in *conjunctive normal form*, called a *cnf-formula*, if it comprises several clauses connected with $\wedge$s, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee x_5 \vee x_6) \wedge (x_3 \vee x_6)$$

- *3cnf-formula*: if all the clauses have three literals, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee x_5 \vee x_6) \wedge (\overline{x_2} \vee x_3 \vee x_6) \wedge (\overline{x_4} \vee x_5 \vee x_6)$$

定义: *3SAT*

$3SAT = \{\langle\varphi\rangle \mid \varphi$ is a satisfiable 3cnf-formula$\}$

- A Boolean formula is *satisfiable* if *some assignment* of 0s and 1s to the variables makes the *formula evaluate to 1*.

定理

3SAT is polynomial time reducible to *CLIQUE*

定理

3SAT is polynomial time reducible to *CLIQUE*

证明

Let φ be a 3cnf-formula with k *clauses*.
The reduction f generates the string $\langle G, k \rangle$, where G is an undirected graph defined as follows.

定理

3SAT is polynomial time reducible to *CLIQUE*

证明

Let φ be a 3cnf-formula with k *clauses*.
The reduction f generates the string $\langle G, k \rangle$, where G is an undirected graph defined as follows.

- The nodes in G are organized into k *groups* of three nodes each called the *triples*, $t_1, \ldots, t_k$
- Each *triple* corresponds to *one of the clauses* in φ
- Each *node* in a triple corresponds to a *literal* in the associated clause
- The *edges* of G connect all but *two types* of pairs of nodes in G

定理

3SAT is polynomial time reducible to *CLIQUE*

证明

Let φ be a 3cnf-formula with k *clauses*.
The reduction f generates the string $\langle G, k \rangle$, where G is an undirected graph defined as follows.

- The nodes in G are organized into k *groups* of three nodes each called the *triples*, $t_1, \ldots, t_k$
- Each *triple* corresponds to *one of the clauses* in φ
- Each *node* in a triple corresponds to a *literal* in the associated clause
- The *edges* of G connect all but *two types* of pairs of nodes in G

定理

3SAT is polynomial time reducible to *CLIQUE*

证明

Let φ be a 3cnf-formula with k *clauses*.
The reduction f generates the string $\langle G, k \rangle$, where G is an undirected graph defined as follows.

- The nodes in G are organized into k *groups* of three nodes each called the *triples*, $t_1, \ldots, t_k$
- Each *triple* corresponds to *one of the clauses* in φ
- Each *node* in a triple corresponds to a *literal* in the associated clause
- The *edges* of G connect all but *two types* of pairs of nodes in G

定理

3SAT is polynomial time reducible to *CLIQUE*

证明

Let φ be a 3cnf-formula with k *clauses*.
The reduction f generates the string $\langle G, k\rangle$, where G is an undirected graph defined as follows.

- The nodes in G are organized into k *groups* of three nodes each called the *triples*, $t_1, \ldots, t_k$
- Each *triple* corresponds to *one of the clauses* in φ
- Each *node* in a triple corresponds to a *literal* in the associated clause
- The *edges* of G connect all but *two types* of pairs of nodes in G

定理

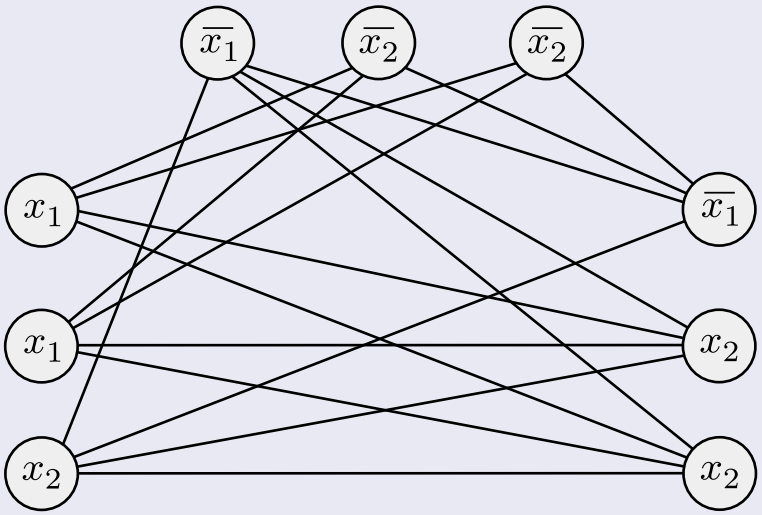
3SAT is polynomial time reducible to *CLIQUE*

证明

Let φ be a 3cnf-formula with k *clauses*.

The reduction f generates the string $\langle G, k\rangle$, where G is an undirected graph defined as follows.

Example: when $\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$



定理

3SAT is polynomial time reducible to *CLIQUE*

证明

Let φ be a 3cnf-formula with k *clauses*.
The reduction f generates the string $\langle G, k \rangle$, where G is an undirected graph defined as follows.

(1) Suppose that φ has a satisfying assignment

- *At least one literal* is *true* in every clause
- In *each* triple of G, we select *one node* corresponding to a *true* literal in the satisfying assignment
- The number of nodes selected is k
- G contains a k-*clique*

定理

3SAT is polynomial time reducible to *CLIQUE*

证明

Let φ be a 3cnf-formula with k *clauses*.
The reduction f generates the string $\langle G, k \rangle$, where G is an undirected graph defined as follows.

(2) Suppose that G has a k-clique

- *No two* of the clique′ s nodes occur *in the same triple*
- Each of the k triples contains exactly one of the k clique nodes
- We *assign truth values* to the variables of φ so that each literal *labeling* a *clique node* is made *true*
- φ is satisfiable

Outline

定义: NP-Completeness

A language B is *NP-complete* if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is *polynomial time reducible* to B.

If B merely satisfies condition 2, we say that it is *NP-hard*

定理

If B is NP-complete and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$

定理

If B is NP-complete and $B \leq_{\mathrm{P}} C$ for C in NP, then C is NP-complete.

问: Which problem is NP-complete and how to prove it?
答: 见下页

定义: NP-Completeness

A language B is *NP-complete* if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is *polynomial time reducible* to B.

If B merely satisfies condition 2, we say that it is *NP-hard*

定理

If B is NP-complete and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$

定理

If B is NP-complete and $B \leq_{\mathrm{P}} C$ for C in NP, then C is NP-complete.

问: Which problem is NP-complete and how to prove it?
答: 见下页

定义: NP-Completeness

A language B is *NP-complete* if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is *polynomial time reducible* to B.

If B merely satisfies condition 2, we say that it is *NP-hard*

定理

If B is NP-complete and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$

定理

If B is NP-complete and $B \leq_{\mathrm{P}} C$ for C in NP, then C is NP-complete.

问: Which problem is NP-complete and how to prove it?

答: 见下页

定义: NP-Completeness

A language B is *NP-complete* if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is *polynomial time reducible* to B.

If B merely satisfies condition 2, we say that it is *NP-hard*

定理

If B is NP-complete and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$

定理

If B is NP-complete and $B \leq_{\mathrm{P}} C$ for C in NP, then C is NP-complete.

问: Which problem is NP-complete and how to prove it?
答: 见下页

Outline

定理: Cook–Levin theorem

SAT is NP-complete

定理: Cook–Levin theorem

SAT is NP-complete

证明

(1) Prove that SAT is in NP

- A *nondeterministic polynomial* time machine can guess an assignment to a given formula φ
- and *accept* if the assignment satisfies φ

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides* A *in* n^k *time* for some constant k, and construct a tableau for N

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides* A *in* n^k *time* for some constant k, and construct a tableau for N
  - A tableau for N on w is an $n^k \times n^k$ table whose *rows* are the *configurations* of *a branch* of the computation of N on input w

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
  - A tableau for N on w is an $n^k \times n^k$ table whose *rows* are the *configurations* of *a branch* of the computation of N on input w
  - A tableau is *accepting* if *any row* of the tableau is an *accepting configuration*.

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides* A *in* n^k *time* for some constant k, and construct a tableau for N
  - A tableau for N on w is an $n^k \times n^k$ table whose *rows* are the *configurations* of *a branch* of the computation of N on input w
  - A tableau is *accepting* if *any row* of the tableau is an *accepting configuration*.
  - the problem of determining whether N *accepts* w is *equivalent to* the problem of determining whether an *accepting* tableau for N on w exists.

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides* A *in* n^k *time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}} SAT$ 1 2 3

- Let N be a nondeterministic Turing machine that *decides* A *in* n^k *time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - Say that Q and Γ are the *state set* and *tape alphabet* of N, respectively
  - Let $C = Q \cup \Gamma \cup \{\#\}$
  - For each i and j between 1 and n^k and for each s in C, we have a variable, $x_{i,j,s}$.
  - Each of the $(n^k)^2$ entries of a tableau is called a *cell*
  - The cell in row i and column j is called $cell[i,j]$ and contains a symbol

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}} SAT$ 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - Say that Q and Γ are the *state set* and *tape alphabet* of N, respectively
  - Let $C = Q \cup \Gamma \cup \{\#\}$
  - For each i and j between 1 and n^k and for each s in C, we have a variable, $x_{i,j,s}$.
  - Each of the $(n^k)^2$ entries of a tableau is called a *cell*
  - The cell in row i and column j is called $cell[i,j]$ and contains a symbol

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}} SAT$ 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - Say that Q and Γ are the *state set* and *tape alphabet* of N, respectively
  - Let $C = Q \cup \Gamma \cup \{\#\}$
  - For each i and j between 1 and n^k and for each s in C, we have a variable, $x_{i,j,s}$.
  - Each of the $(n^k)^2$ entries of a tableau is called a *cell*
  - The cell in row i and column j is called $cell[i,j]$ and contains a symbol

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}} SAT$ 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - Say that Q and Γ are the *state set* and *tape alphabet* of N, respectively
  - Let $C = Q \cup \Gamma \cup \{\#\}$
  - For each i and j between 1 and n^k and for each s in C, we have a variable, $x_{i,j,s}$.
  - Each of the $(n^k)^2$ entries of a tableau is called a *cell*
  - The cell in row i and column j is called $cell[i,j]$ and contains a symbol

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}} SAT$ 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - Say that Q and Γ are the *state set* and *tape alphabet* of N, respectively
  - Let $C = Q \cup \Gamma \cup \{\#\}$
  - For each i and j between 1 and n^k and for each s in C, we have a variable, $x_{i,j,s}$.
  - Each of the $(n^k)^2$ entries of a tableau is called a *cell*
  - The cell in row i and column j is called $cell[i,j]$ and contains a symbol

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}} SAT$ 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - Say that Q and Γ are the *state set* and *tape alphabet* of N, respectively
  - Let $C = Q \cup \Gamma \cup \{\#\}$
  - For each i and j between 1 and n^k and for each s in C, we have a variable, $x_{i,j,s}$.
  - Each of the $(n^k)^2$ entries of a tableau is called a *cell*
  - The cell in row i and column j is called $cell[i,j]$ and contains a symbol

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides* A *in* n^k *time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\mathrm{cell}} \wedge \phi_{\mathrm{start}} \wedge \phi_{\mathrm{accept}} \wedge \phi_{\mathrm{move}}$

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}}$

$$\phi_{\text{cell}} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right]$$

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}}$

$$\phi_{\text{cell}} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right]$$

1st part: *at least one* variable that is associated with *each cell* is on

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}}$

$$\phi_{\text{cell}} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right]$$

2nd part: *no more than one variable* is on for *each cell*

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}}$

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}\end{aligned}$$

the *first row* of the table is the *starting configuration* of N on w

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\mathrm{cell}} \wedge \phi_{\mathrm{start}} \wedge \phi_{\mathrm{accept}} \wedge \phi_{\mathrm{move}}$

$$\phi_{\mathrm{accept}} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{\mathrm{accept}}}$$

an *accepting configuration* occurs in the tableau

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}}$

$$\phi_{\text{move}} = \bigwedge_{1 \leq i < n^k,\ 1 < j < n^k} (\text{the } (i,j)\text{-window is legal}).$$

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\mathrm{cell}} \wedge \phi_{\mathrm{start}} \wedge \phi_{\mathrm{accept}} \wedge \phi_{\mathrm{move}}$

$$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{is a legal window}}} \left(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}\right)$$

$$\delta(q_1, \mathtt{a}) = \{(q_1,\mathtt{b},\mathrm{R})\} \text{ and } \delta(q_1, \mathtt{b}) = \{(q_2,\mathtt{c},\mathrm{L}), (q_2,\mathtt{a},\mathrm{R})\}$$

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides* A *in* n^k *time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}}$

Examples of *legal* windows:

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides* A *in* n^k *time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\mathrm{cell}} \wedge \phi_{\mathrm{start}} \wedge \phi_{\mathrm{accept}} \wedge \phi_{\mathrm{move}}$

Examples of *illegal* windows:

(a)

a	b	a
a	a	a

(b)

a	q_1	b
q_2	a	a

(c)

b	q_1	b
q_2	b	q_2

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\mathrm{cell}} \wedge \phi_{\mathrm{start}} \wedge \phi_{\mathrm{accept}} \wedge \phi_{\mathrm{move}}$
- Analyze the Complexity of Reduction 3.1 3.2 3.3

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\mathrm{cell}} \wedge \phi_{\mathrm{start}} \wedge \phi_{\mathrm{accept}} \wedge \phi_{\mathrm{move}}$
- Analyze the Complexity of Reduction 3.1 3.2 3.3
  - The total number of variables is $O(n^{2k})$
    - tableau is an $n^k \times n^k$ table, so it contains n^{2k} cells

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\mathrm{cell}} \wedge \phi_{\mathrm{start}} \wedge \phi_{\mathrm{accept}} \wedge \phi_{\mathrm{move}}$
- Analyze the Complexity of Reduction 3.1 3.2 3.3
  - The total number of variables is $O(n^{2k})$
    - tableau is an $n^k \times n^k$ table, so it contains n^{2k} cells

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides* A *in* n^k *time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\mathrm{cell}} \wedge \phi_{\mathrm{start}} \wedge \phi_{\mathrm{accept}} \wedge \phi_{\mathrm{move}}$
- Analyze the Complexity of Reduction 3.1 3.2 3.3
  - ϕ' s total size is $O(n^{2k})$

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\mathrm{cell}} \wedge \phi_{\mathrm{start}} \wedge \phi_{\mathrm{accept}} \wedge \phi_{\mathrm{move}}$
- Analyze the Complexity of Reduction 3.1 3.2 3.3
  - ϕ' s total size is $O(n^{2k})$
    - ϕ_{cell}: contains a fixed-size fragment of the formula for each cell of the tableau, so its size is $O(n^{2k})$

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\mathrm{cell}} \wedge \phi_{\mathrm{start}} \wedge \phi_{\mathrm{accept}} \wedge \phi_{\mathrm{move}}$
- Analyze the Complexity of Reduction 3.1 3.2 3.3
  - ϕ' s total size is $O(n^{2k})$
    - ϕ_{start} has a fragment for each cell in the top row, so its size is $O(n^k)$

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\mathrm{cell}} \wedge \phi_{\mathrm{start}} \wedge \phi_{\mathrm{accept}} \wedge \phi_{\mathrm{move}}$
- Analyze the Complexity of Reduction 3.1 3.2 3.3
  - ϕ' s total size is $O(n^{2k})$
    - ϕ_{move} and ϕ_{accept} each contain a fixed-size fragment of the formula for each cell of the tableau, so their size is $O(n^{2k})$

定理: Cook–Levin theorem

SAT is NP-complete

证明

(2) Take any language A in NP and show that $A \leq_{\mathrm{P}}$ *SAT* 1 2 3

- Let N be a nondeterministic Turing machine that *decides A in n^k time* for some constant k, and construct a tableau for N
- Reduction from w to a formula ϕ: 2.1 2.2
  - If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s
  - $\phi = \phi_{\mathrm{cell}} \wedge \phi_{\mathrm{start}} \wedge \phi_{\mathrm{accept}} \wedge \phi_{\mathrm{move}}$
- Analyze the Complexity of Reduction 3.1 3.2 3.3
  - We may easily construct a reduction that produces ϕ in polynomial time from the input w
    - Each component of the formula is composed of many nearly identical fragments

推论

3SAT is NP-complete

证明方法 1

Prove that SAT polynomial time reduces to 3SAT. Theorem Tseitin Transformation

证明方法 2

Modify the proof of The Cook-Levin Theorem

推论

3SAT is NP-complete

证明方法 1

Prove that SAT polynomial time reduces to 3SAT. Theorem Tseitin Transformation

证明方法 2

Modify the proof of The Cook-Levin Theorem

推论

3SAT is NP-complete

证明方法 1

Prove that SAT polynomial time reduces to 3SAT. Theorem Tseitin Transformation

证明方法 2

Modify the proof of The Cook-Levin Theorem

4.1 Time Complexity

Outline

推论: *CLIQUE* is NP-complete

CLIQUE is NP-complete

证明

Use the following theorems

- *CLIQUE* is NP
- Reducibility from 3SAT to *CLIQUE*
- Theorems of NP-Complete

推论: *CLIQUE* is NP-complete

CLIQUE is NP-complete

证明

Use the following theorems

- *CLIQUE* is NP
- Reducibility from 3SAT to *CLIQUE*
- Theorems of NP-Complete

定义: Vertex cover

A *vertex cover* of G is a *subset* of the *nodes* where *every edge* of G touches *one* of those *nodes*.
VERTEX-COVER $= \{\langle G, k\rangle \mid G$ is an undirected graph that has a k-node vertex cover$\}$

定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(1) Prove: *VERTEX-COVER* is NP

- A certificate is simply a vertex cover of size k

定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *VERTEX-COVER* 2.1 Construction 2.2 Proof 1 2.3 Proof 2

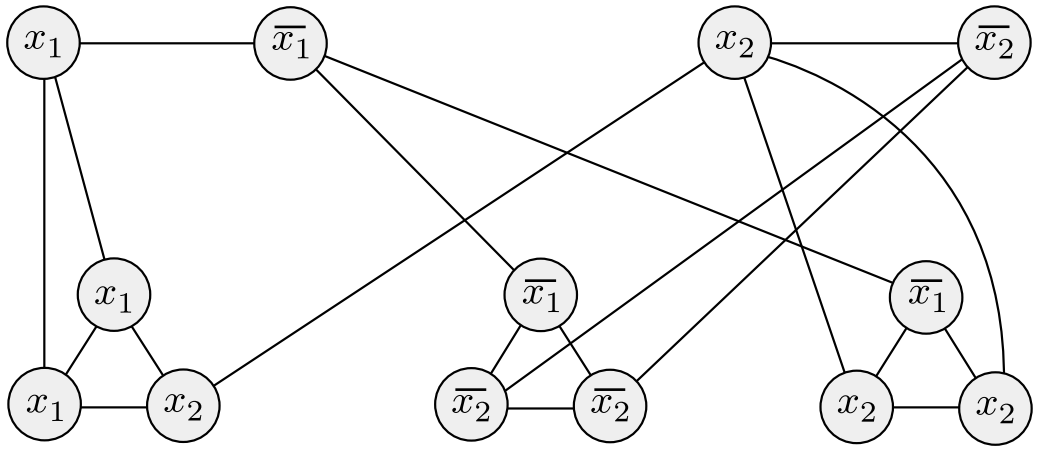
定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: $3SAT \leq_{\mathrm{P}} VERTEX\text{-}COVER$ 2.1 Construction 2.2 Proof 1 2.3 Proof 2

- (2.1) The graph contains *gadgets* that mimic the *variables* and *clauses* of the formula: the *variable gadget* and the *clause gadget*



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

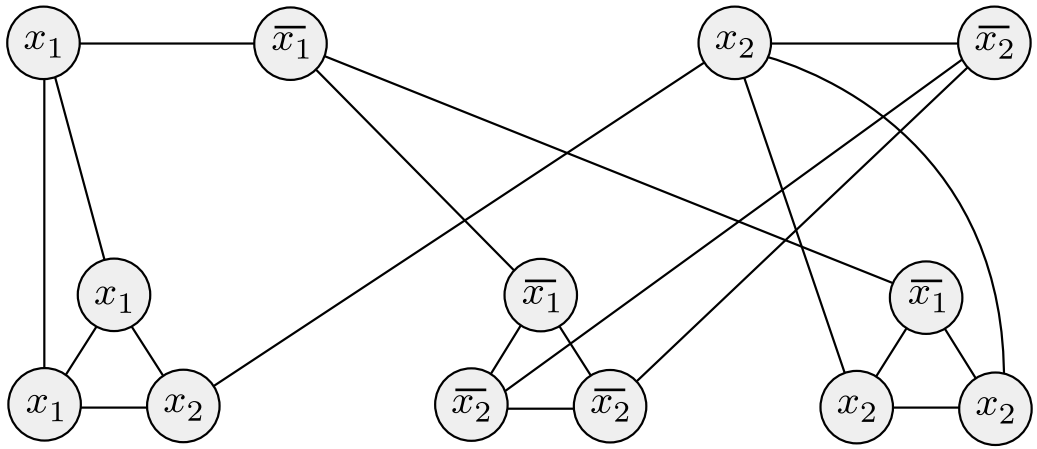
定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: $3SAT \leq_{\mathrm{P}} VERTEX\text{-}COVER$ 2.1 Construction 2.2 Proof 1 2.3 Proof 2

- The *variable gadget* contains *two nodes* connected by an edge
  - *One of these nodes* must appear in the vertex cover.



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

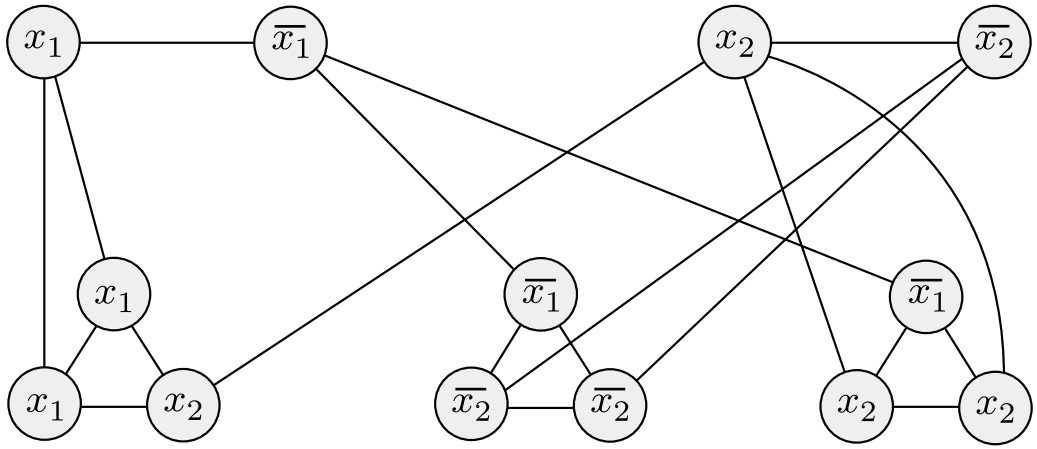
定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: $3SAT \leq_{\mathrm{P}}$ *VERTEX-COVER* 2.1 Construction 2.2 Proof 1 2.3 Proof 2

- The *clause gadget* contains *three nodes* and additional edges
  - *Any vertex cover* must include *at least two of the nodes*, or possibly all three.



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

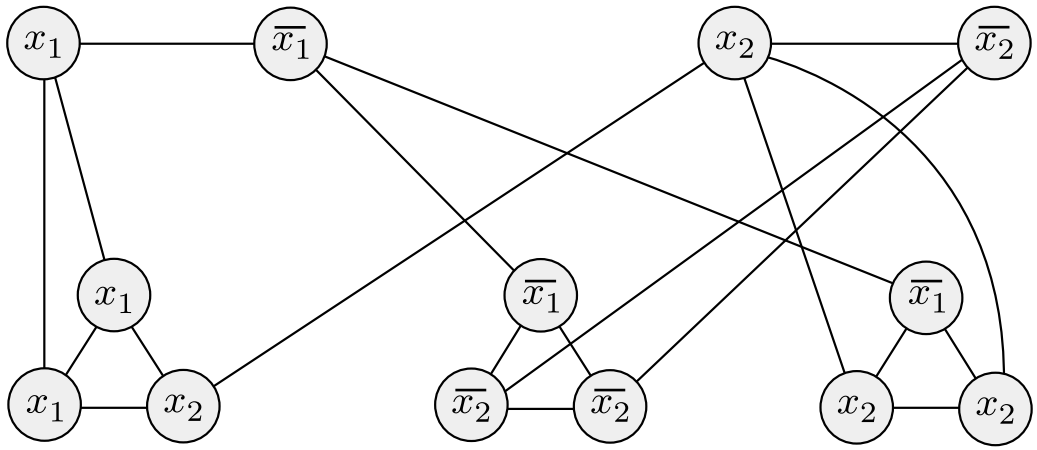
定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: $3SAT \leq_{\mathrm{P}}$ *VERTEX-COVER* 2.1 Construction 2.2 Proof 1 2.3 Proof 2

- Thus, the total *number of nodes* that appear in G is $2m + 3l$, where ϕ has *m variables* and l *clauses*
  - Let k *be* $m + 2l$



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

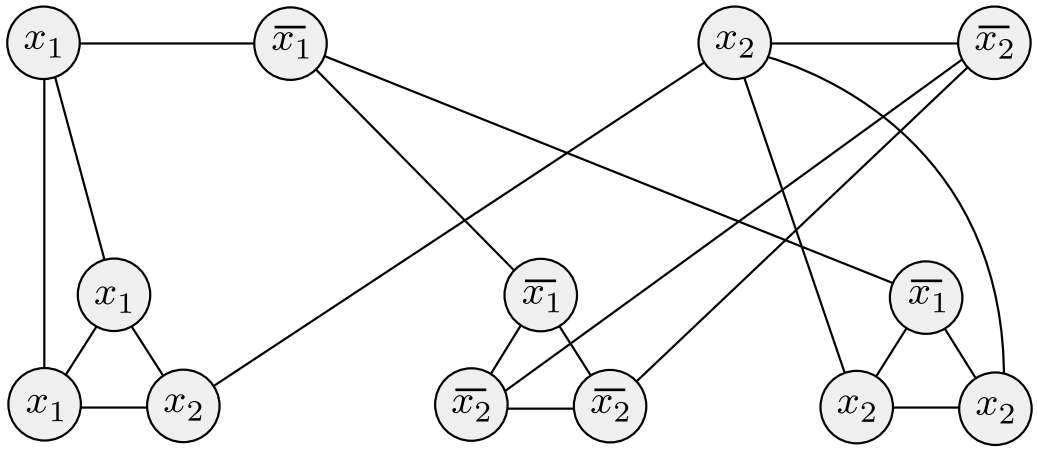
定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *VERTEX-COVER* 2.1 Construction 2.2 Proof 1 2.3 Proof 2

- We *chose* k so that the sought-after vertex cover has
  - *one node* per *variable gadget*
  - *two nodes* per *clause gadget*



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

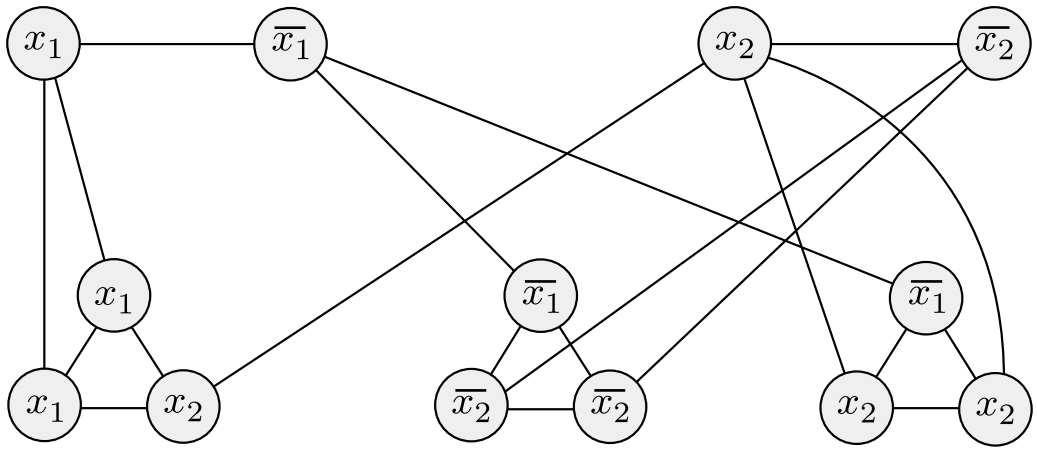
定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *VERTEX-COVER* 2.1 Construction 2.2 Proof 1 2.3 Proof 2

- (2.2) Prove: If ϕ is satisfiable, G has a vertex cover with k nodes



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

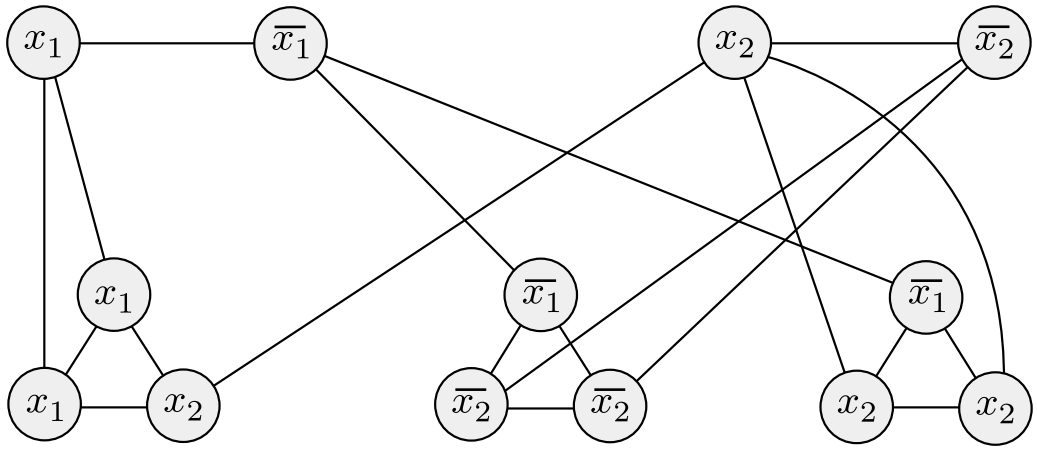
定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: $3SAT \leq_{\mathrm{P}} VERTEX\text{-}COVER$ 2.1 Construction 2.2 Proof 1 2.3 Proof 2

- First, put the nodes of the ***variable gadgets*** that correspond to the *true literals* in the assignment into the vertex cover



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

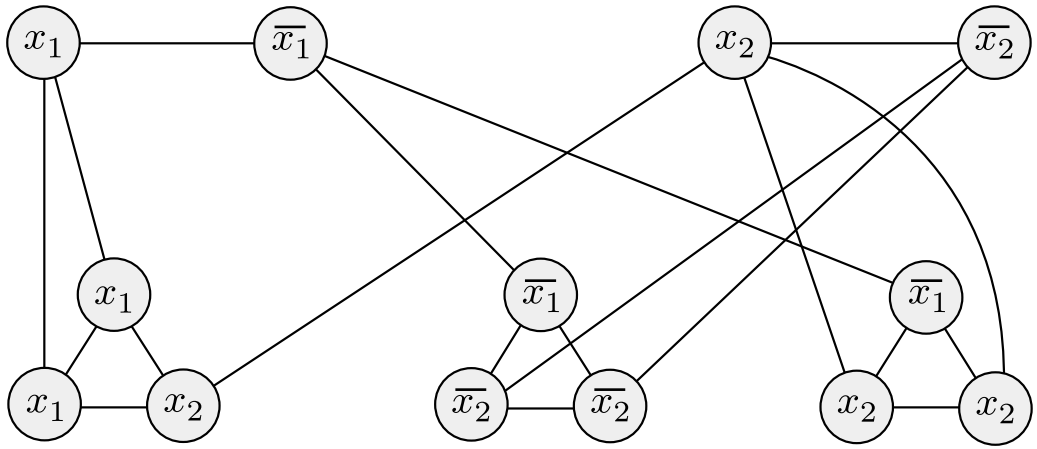
定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: $3SAT \leq_{\mathrm{P}} VERTEX\text{-}COVER$ 2.1 Construction 2.2 Proof 1 2.3 Proof 2

- Then, select *one true literal* in every clause and put the *remaining two nodes* from every ***clause gadget*** *into* the *vertex cover*.



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

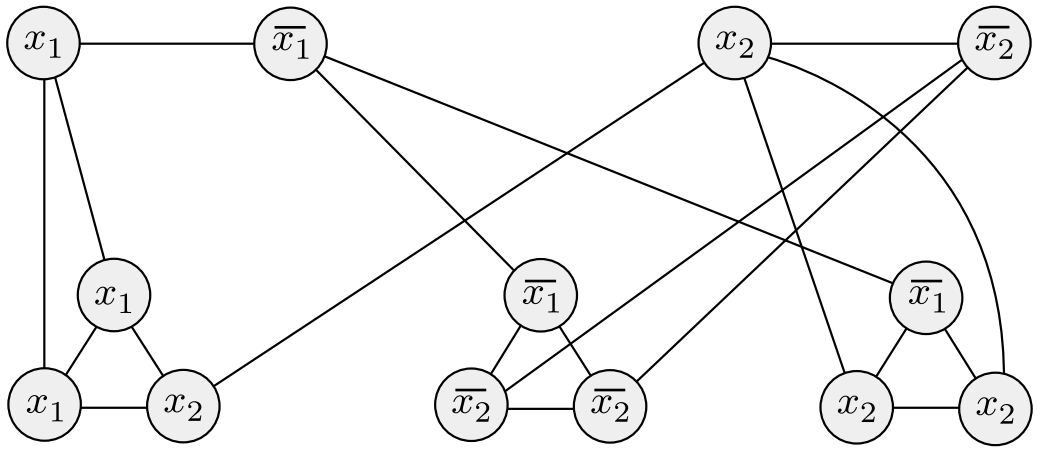
定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *VERTEX-COVER* 2.1 Construction 2.2 Proof 1 2.3 Proof 2

- Now we have a total of k nodes.



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

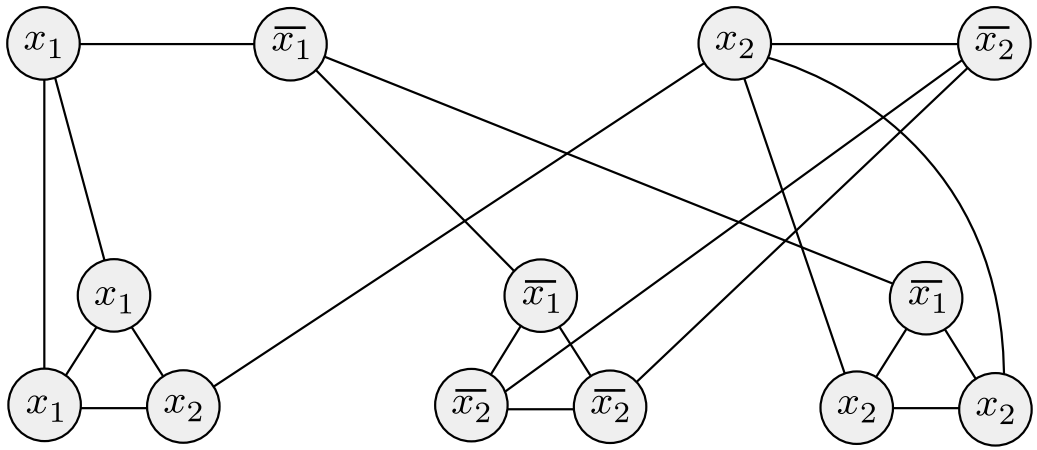
定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *VERTEX-COVER* 2.1 Construction 2.2 Proof 1 2.3 Proof 2

- (2.3) Prove: ϕ is satisfiable, if G has a vertex cover with k nodes



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

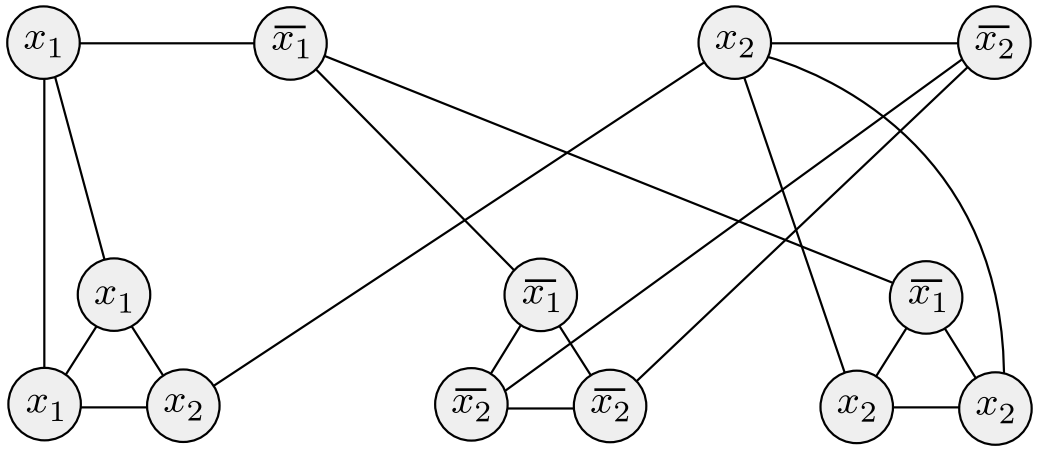
定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: $3SAT \leq_{\mathrm{P}} VERTEX\text{-}COVER$ 2.1 Construction 2.2 Proof 1 2.3 Proof 2

- The vertex cover *must* contain *one* node in each *variable gadget* and *two* in every *clause gadget*



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

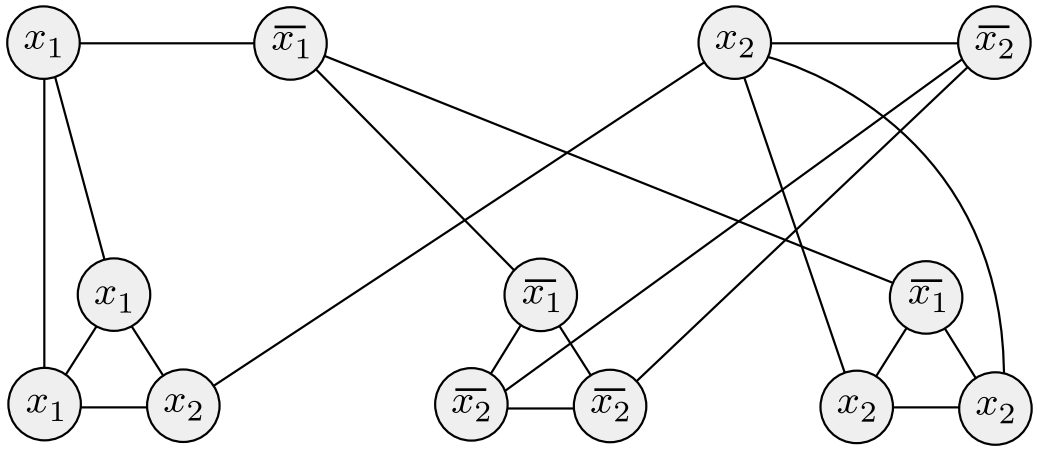
定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: $3SAT \leq_{\mathrm{P}} VERTEX\text{-}COVER$ 2.1 Construction 2.2 Proof 1 2.3 Proof 2

- We take the nodes of the *variable gadgets* that are in the *vertex cover* and assign *TRUE* to the corresponding literals.



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

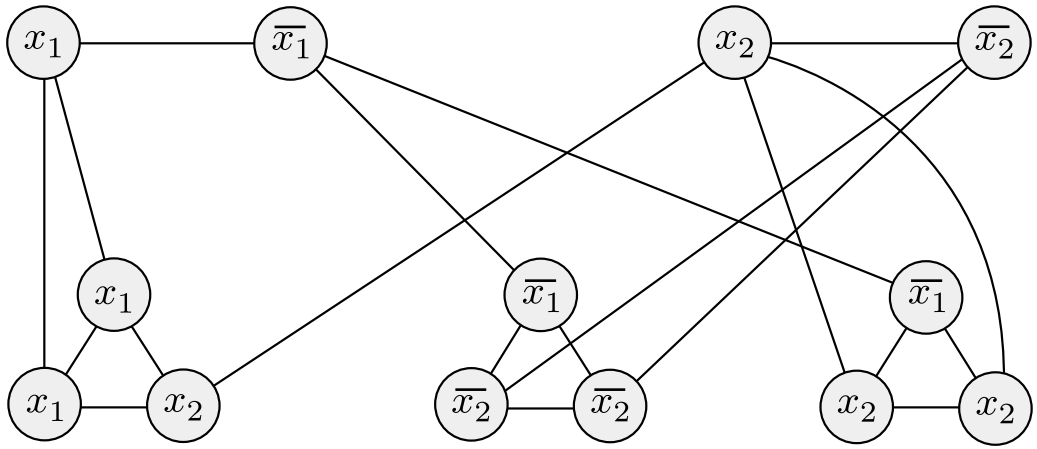
定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *VERTEX-COVER* 2.1 Construction 2.2 Proof 1 2.3 Proof 2

- That assignment *satisfies* ϕ: because each of the ***three*** *edges connecting* the *variable gadgets* with each *clause gadget* is *covered*



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

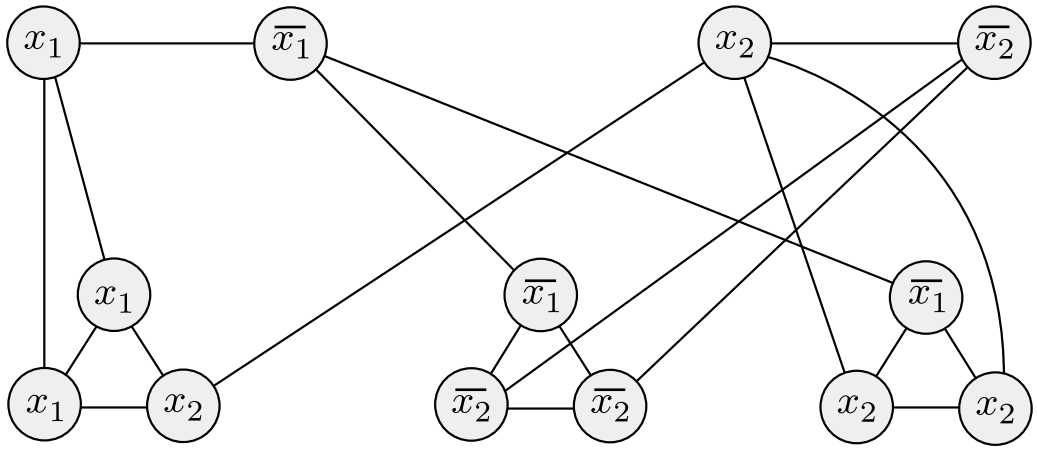
定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: $3SAT \leq_{\mathrm{P}}$ *VERTEX-COVER* 2.1 Construction 2.2 Proof 1 2.3 Proof 2

- That assignment *satisfies* ϕ: and only ***two*** *nodes* of the clause gadget are in the *vertex cover*



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

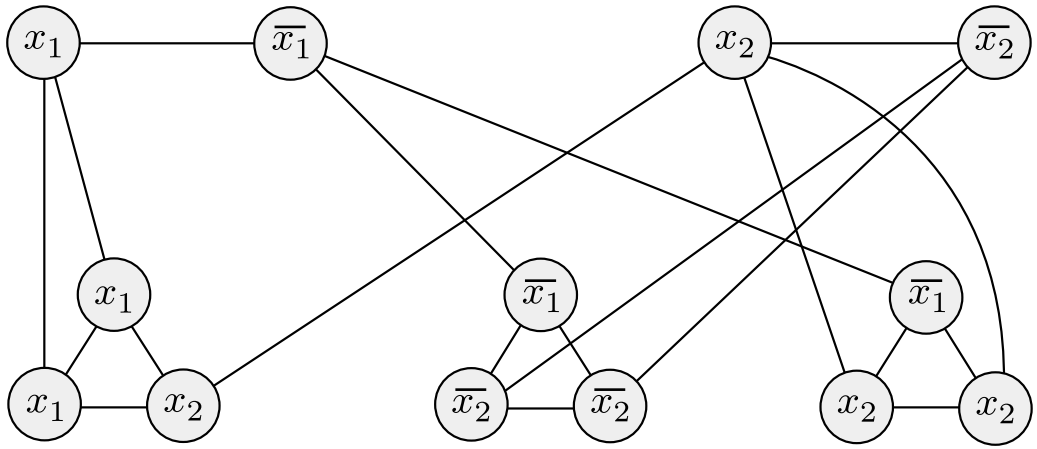
定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: $3SAT \leq_{\mathrm{P}} VERTEX\text{-}COVER$ 2.1 Construction 2.2 Proof 1 2.3 Proof 2

- That assignment *satisfies* ϕ: ***one*** of the edges *must* be *covered* by a node from a *variable gadget*



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

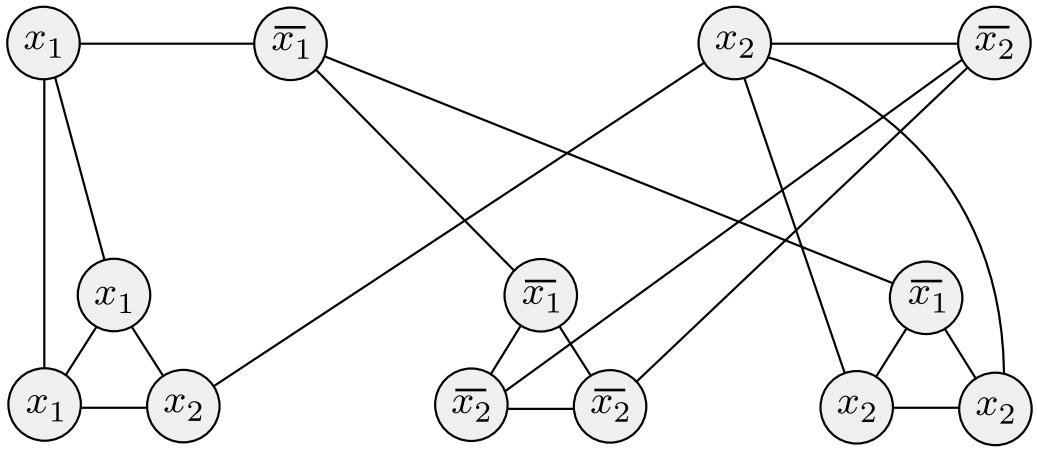
定理: *VERTEX-COVER* is NP-complete

VERTEX-COVER is NP-complete

证明: (Proof by Construction)

(2) Prove: $3SAT \leq_{\mathrm{P}} VERTEX\text{-}COVER$ 2.1 Construction 2.2 Proof 1 2.3 Proof 2

- Proved



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(1) Prove: *HAMPATH* is NP. Proved

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.1) Construction with a 3cnf-formula ϕ containing k clauses

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$$

- each a, b, and c is a literal x_i or $\overline{x_i}$
- Let $x_1, \ldots, x_l$ be the l variables of ϕ

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.2) Convert ϕ to G Basic Grouping More Edges: ▸ x_i in c_j ▸ $\overline{x_i}$ in c_j

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.2) Convert ϕ to G Basic Grouping More Edges: ▸ x_i in c_j ▸ $\overline{x_i}$ in c_j

- Represent each variable x_i with a diamond-shaped structure
- Represent each clause of ϕ as a single node
- The high-level structure of G

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.2) Convert ϕ to G Basic Grouping More Edges: ▸ x_i in c_j ▸ $\overline{x_i}$ in c_j

- Represent each variable x_i with a diamond-shaped structure
- Represent each clause of ϕ as a single node
- The high-level structure of G

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.2) Convert ϕ to G Basic Grouping More Edges: ▸ x_i in c_j ▸ $\overline{x_i}$ in c_j

- Represent each variable x_i with a diamond-shaped structure
- Represent each clause of ϕ as a single node
- The high-level structure of G

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.2) Convert ϕ to G Basic Grouping More Edges: ▸ x_i in c_j ▸ $\overline{x_i}$ in c_j

- *Grouping:* Connect the *diamonds* representing the *variables* to the *nodes* representing the *clauses*
  - Each *diamond* structure contains a *horizontal row* of nodes connected by *edges* running in *both directions*.
  - The *horizontal row* contains $3k + 1$ *nodes* in *addition* to the *two nodes* on the *ends* belonging to the diamond
  - These nodes are *grouped* into *adjacent pairs*, one for *each clause*, with *extra separator nodes* next to the pairs

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: $3SAT \leq_{\mathrm{P}} HAMPATH$ 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.2) Convert ϕ to G Basic Grouping More Edges: ▸ x_i in c_j ▸ $\overline{x_i}$ in c_j

- *Grouping:* Connect the *diamonds* representing the *variables* to the *nodes* representing the *clauses*
  - Each *diamond* structure contains a *horizontal row* of nodes connected by *edges* running in *both directions*.
  - The *horizontal row* contains $3k + 1$ *nodes* in *addition* to the *two nodes* on the *ends* belonging to the diamond
  - These nodes are *grouped* into *adjacent pairs*, one for *each clause*, with *extra separator nodes* next to the pairs

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: $3SAT \leq_{\mathrm{P}} HAMPATH$ 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.2) Convert ϕ to G Basic Grouping More Edges: ▸ x_i in c_j ▸ $\overline{x_i}$ in c_j

- *Grouping:* Connect the *diamonds* representing the *variables* to the *nodes* representing the *clauses*
  - Each *diamond* structure contains a *horizontal row* of nodes connected by *edges* running in *both directions*.
  - The *horizontal row* contains $3k + 1$ *nodes* in *addition* to the *two nodes* on the *ends* belonging to the diamond
  - These nodes are *grouped* into *adjacent pairs*, one for *each clause*, with *extra separator nodes* next to the pairs

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.2) Convert ϕ to G Basic Grouping More Edges: ▸ x_i in c_j ▸ $\overline{x_i}$ in c_j

- *Grouping:* Connect the *diamonds* representing the *variables* to the *nodes* representing the *clauses*
  - Each *diamond* structure contains a *horizontal row* of nodes connected by *edges* running in *both directions*.
  - The *horizontal row* contains $3k + 1$ *nodes* in *addition* to the *two nodes* on the *ends* belonging to the diamond
  - These nodes are *grouped* into *adjacent pairs*, one for *each clause*, with *extra separator nodes* next to the pairs

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.2) Convert ϕ to G Basic Grouping More Edges: ▸ x_i in c_j ▸ $\overline{x_i}$ in c_j

- *More Edges*:
  - If variable x_i appears in clause c_j, we add the following two edges *from* the jth pair in the ith diamond *to* the jth clause node.
  - If variable $\overline{x_i}$ appears in clause c_j, we add the following two edges *from* the jth pair in the ith diamond *to* the jth clause node.
  - *Construction Complete*

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.2) Convert ϕ to G Basic Grouping More Edges: ▸ x_i in c_j ▸ $\overline{x_i}$ in c_j

- *More Edges*:
  - If variable x_i appears in clause c_j, we add the following two edges *from* the jth pair in the ith diamond *to* the jth clause node.
  - If variable $\overline{x_i}$ appears in clause c_j, we add the following two edges *from* the jth pair in the ith diamond *to* the jth clause node.
  - *Construction Complete*

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.2) Convert ϕ to G Basic Grouping More Edges: ▸ x_i in c_j ▸ $\overline{x_i}$ in c_j

- *More Edges*:
  - If variable x_i appears in clause c_j, we add the following two edges *from* the jth pair in the ith diamond *to* the jth clause node.
  - If variable $\overline{x_i}$ appears in clause c_j, we add the following two edges *from* the jth pair in the ith diamond *to* the jth clause node.
  - *Construction Complete*

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.2) Convert ϕ to G Basic Grouping More Edges: ▸ x_i in c_j ▸ $\overline{x_i}$ in c_j

- *More Edges*:
  - If variable x_i appears in clause c_j, we add the following two edges *from* the jth pair in the ith diamond *to* the jth clause node.
  - If variable $\overline{x_i}$ appears in clause c_j, we add the following two edges *from* the jth pair in the ith diamond *to* the jth clause node.
  - *Construction Complete*

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.3) Prove: If ϕ is satisfiable, G has a Hamiltonian path from s to t

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.3) Prove: If ϕ is satisfiable, G has a Hamiltonian path from s to t

- First, *ignore* the *clause* nodes
  - The path begins at s, goes through each diamond in turn, and ends up at t.

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.3) Prove: If ϕ is satisfiable, G has a Hamiltonian path from s to t

- First, *ignore* the *clause* nodes
  - The path begins at s, goes through each diamond in turn, and ends up at t.
- Include the *clause nodes* by adding detours at the horizontal nodes
  - If we selected x_i in clause c_j, we can detour at the jth pair in the ith diamond.
  - If we selected $\overline{x_i}$ in clause c_j, we can detour at the jth pair in the ith diamond.

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.4) Prove: ϕ is satisfiable, if G has a Hamiltonian path from s to t

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.4) Prove: ϕ is satisfiable, if G has a Hamiltonian path from s to t

Case 1: The hamiltonian path is *normal*

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.4) Prove: ϕ is satisfiable, if G has a Hamiltonian path from s to t

Case 1: The hamiltonian path is *normal*

- Define *normal*: it *goes through* the *diamonds* in order from the *top one* to the *bottom one*, *except* for the *detours* to the clause nodes

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.4) Prove: ϕ is satisfiable, if G has a Hamiltonian path from s to t

Case 1: The hamiltonian path is *normal*

We can easily obtain the satisfying assignment:

- If the path zig-zags through the diamond, we assign the corresponding *variable* TRUE
- If it zag-zigs, we assign FALSE

Construction Completed.

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.4) Prove: ϕ is satisfiable, if G has a Hamiltonian path from s to t

Case 2: The hamiltonian path is *not normal*

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.4) Prove: ϕ is satisfiable, if G has a Hamiltonian path from s to t

Case 2: The hamiltonian path is *not normal*

In this case, prove it by *contradiction*:

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.4) Prove: ϕ is satisfiable, if G has a Hamiltonian path from s to t

Case 2: The hamiltonian path is *not normal*

- It can be inferred that
  the path enters a clause from one diamond but returns to another

定理: *HAMPATH* is NP-complete

HAMPATH is NP-complete

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH* 2.1 ϕ 2.2 G 2.3 Proof 1 2.4 Proof 2

(2.4) Prove: ϕ is satisfiable, if G has a Hamiltonian path from s to t

Case 2: The hamiltonian path is *not normal*

- It can be inferred that
  the path enters a clause from one diamond but returns to another
- Hence a Hamiltonian path *must be normal*.

定理: *SUBSET-SUM* is NP-complete

SUBSET-SUM is NP-complete.

定理: *SUBSET-SUM* is NP-complete

SUBSET-SUM is NP-complete.

证明: (Proof by Construction)

(1) Prove: *SUBSET-SUM* is NP. *Proved*

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *SUBSET-SUM* 2.1 Construction 2.2 Proof 1 2.3 Proof 2

定理: *SUBSET-SUM* is NP-complete

SUBSET-SUM is NP-complete.

证明: (Proof by Construction)

(2) Prove: $3SAT \leq_{\mathrm{P}} SUBSET\text{-}SUM$ 2.1 Construction 2.2 Proof 1 2.3 Proof 2

定理: *SUBSET-SUM* is NP-complete

SUBSET-SUM is NP-complete.

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *SUBSET-SUM* 2.1 Construction 2.2 Proof 1 2.3 Proof 2

(2.1) *Construction* with a 3cnf-formula ϕ containing k clauses

- Let ϕ be a Boolean formula with variables $x_1, \ldots, x_l$ and clauses $c_1, \ldots, c_k$
- The reduction converts ϕ to an instance of the SUBSET-SUM problem $\langle S, t \rangle$
  - *Elements of S*: rows labeled with $y_1, z_1, y_2, z_2, \ldots, y_l, z_l$ and $g_1, h_1, g_2, h_2, \ldots, g_k, h_k$
  - t: row below the double line
- Time complexity of reduction: $O(n^2)$

定理: *SUBSET-SUM* is NP-complete

SUBSET-SUM is NP-complete.

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *SUBSET-SUM* 2.1 Construction 2.2 Proof 1 2.3 Proof 2

(2.2) Prove: If ϕ is satisfiable, construct a *subset* of S that sums to t

- Select y_i if x_i is assigned *TRUE* in the satisfying assignment
- Select z_i if x_i is assigned *FALSE*
- Select *enough* of the g and h numbers to bring each of the last k digits up to 3

定理: *SUBSET-SUM* is NP-complete

SUBSET-SUM is NP-complete.

证明: (Proof by Construction)

(2) Prove: *3SAT* $\leq_{\mathrm{P}}$ *SUBSET-SUM* 2.1 Construction 2.2 Proof 1 2.3 Proof 2

(2.3) Prove: Construct a satisfying assignment to ϕ, if a *subset* of S that sums to t

- If the subset contains y_i, we assign x_i TRUE
- If the subset contains z_i, we assign x_i FALSE.

Analysis

- In column c_j, *at most 2* can come from g_j and h_j, so at *least 1* in this column must come from some y_i *or* z_i in the subset.
  - If it is y_i, then x_i appears in c_j and is assigned TRUE, so c_j is satisfied.
  - If it is z_i, then $\overline{x_i}$ appears in c_j and x_i is assigned FALSE, so c_j is satisfied.

4.1 Time Complexity

Outline

总结

- Class P
  - *PATH*
  - *RELPRIME*
  - *Context-free Language*
- Class NP
  - *HAMPATH*, *COMPOSITES*, *CLIQUE*, *SUBSET-SUM*
- Reduction
  - $\textit{3SAT} \leq_{\mathrm{P}} \textit{CLIQUE}$
  - $\textit{SAT} \leq_{\mathrm{P}} \textit{3SAT}$
  - $\textit{3SAT} \leq_{\mathrm{P}} \textit{HAMPATH}$
  - $\textit{3SAT} \leq_{\mathrm{P}} \textit{SUBSET-SUM}$
- NP-complete
  - *SAT*, *3SAT*, *CLIQUE*, *VERTEX-COVER*, *HAMPATH*, *SUBSET-SUM*

4.1 Time Complexity

Outline

作业

7.1 Answer each part TRUE or FALSE.

a. $2n = O(n)$.

b. $n^2 = O(n)$.

A c. $n^2 = O(n \log^2 n)$.

A d. $n \log n = O(n^2)$.

e. $3^n = 2^{O(n)}$.

f. $2^{2^n} = O(2^{2^n})$.

7.2 Answer each part TRUE or FALSE.

a. $n = o(2n)$.

b. $2n = o(n^2)$.

A c. $2^n = o(3^n)$.

A d. $1 = o(n)$.

e. $n = o(\log n)$.

f. $1 = o(1/n)$.

7.5 Is the following formula satisfiable?

$$(x \vee y) \wedge (x \vee \overline{y}) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee \overline{y})$$

7.9 A ***triangle*** in an undirected graph is a 3-clique. Show that $TRIANGLE \in \mathrm{P}$, where $TRIANGLE = \{\langle G \rangle \mid G \text{ contains a triangle}\}$.

回顾

定义: Chomsky Normal Form

A context-free grammar is in *Chomsky normal form* if every rule is of the form

$$A \to BC$$
$$A \to a$$

where a is any *terminal* and A, B, and C are *any variables*—except that B and C *may not* be the *start variable*. In addition, we permit the rule $S \to \varepsilon$, where S is the *start variable*.

定理

Any context-free language is *generated* by a context-free grammar in *Chomsky normal form*.

回顾: 3.2 Decidability

定理

Every context-free language is decidable

证明

Let A be a CFL.
Let G be a CFG for A.

Design a TM M_G that decides A.
$M_G =$ "On input w:

1. Run TM $\underline{S}$ on input $\langle G, w \rangle$.
2. If this machine accepts, accept; if it rejects, reject."

Outline

定义: Boolean logic, Boolean value, Boolean operations, Operands

- *Boolean logic*: a mathematical system built around the two values
  - **TRUE** and **FALSE**: The values TRUE and FALSE are called the *Boolean values* and are often represented by the values *1 and 0*.
- *Boolean operations*: which manipulate Boolean values, i.e.,
  - *negation* $\neg$: $\neg 0 = 1$ and $\neg 1 = 0$
  - *conjunction* or *AND* $\wedge$: The conjunction of two Boolean values is 1 if both of those values are 1.
  - *disjunction* or *OR* $\vee$: The disjunction of two Boolean values is 1 if either of those values is 1.
- *Operands*: Inputs of the operations

$$\begin{array}{lll} 0 \wedge 0 = 0 & 0 \vee 0 = 0 & \overline{0} = 1 \\ 0 \wedge 1 = 0 & 0 \vee 1 = 1 & \overline{1} = 0 \\ 1 \wedge 0 = 0 & 1 \vee 0 = 1 & \\ 1 \wedge 1 = 1 & 1 \vee 1 = 1 & \end{array}$$

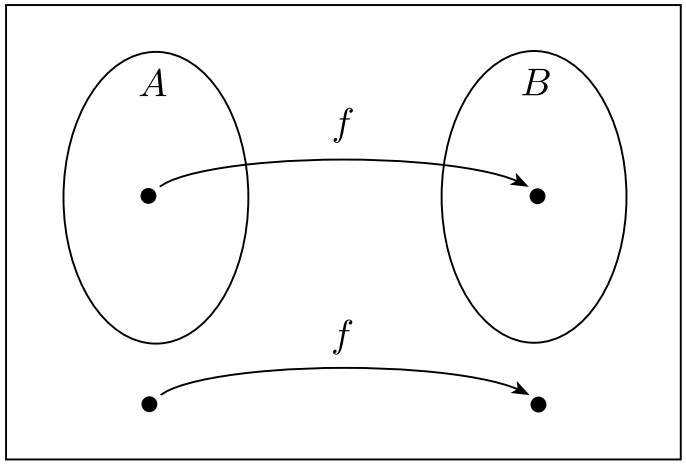
定义: Mapping Reducibility

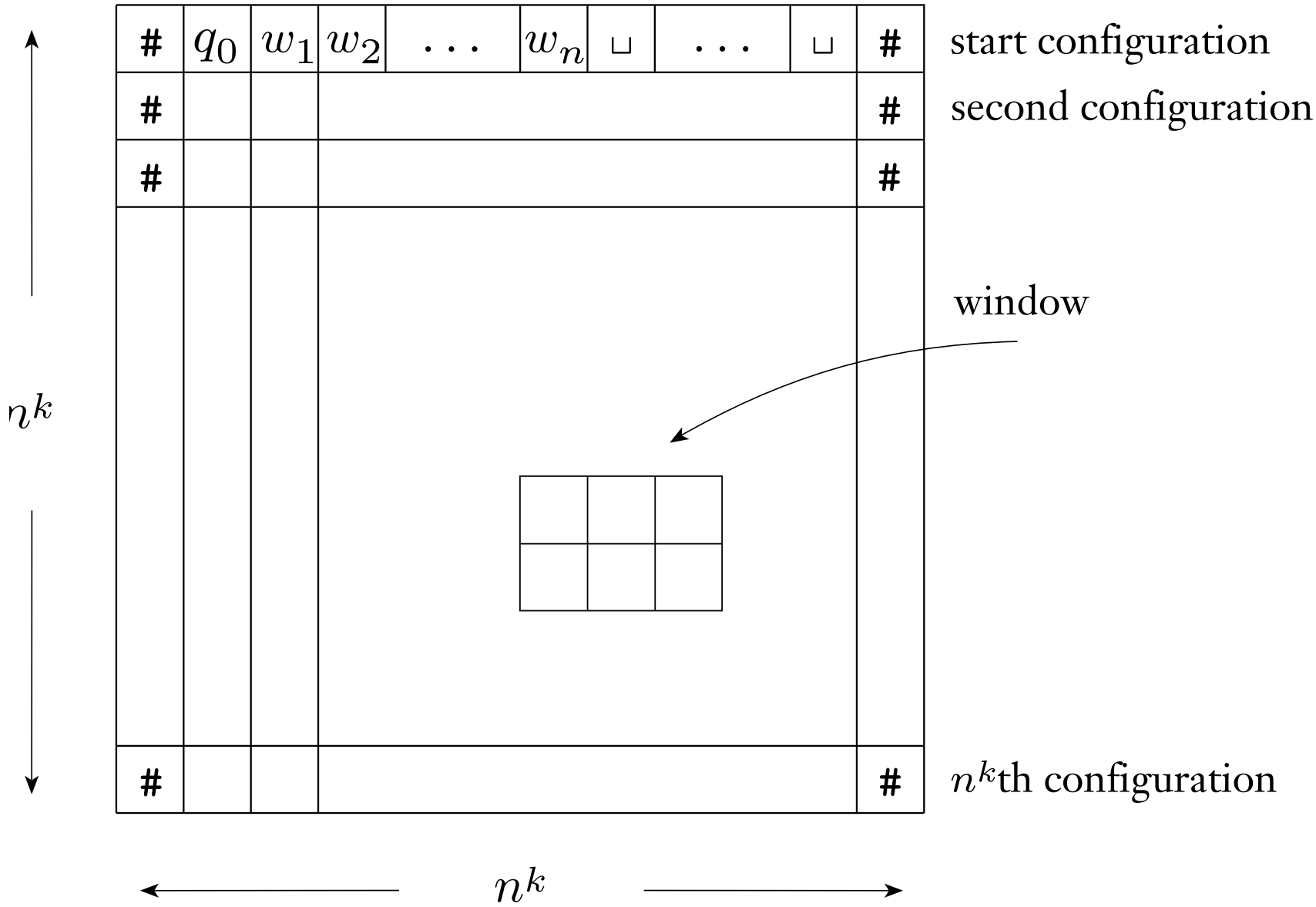
Language A is *mapping reducible* to language B, written $A \leq_{\mathrm{m}} B$, if there is a *computable function* $f : \Sigma^* \to \Sigma^*$, where for every w,

$$w \in A \Leftrightarrow f(w) \in B$$

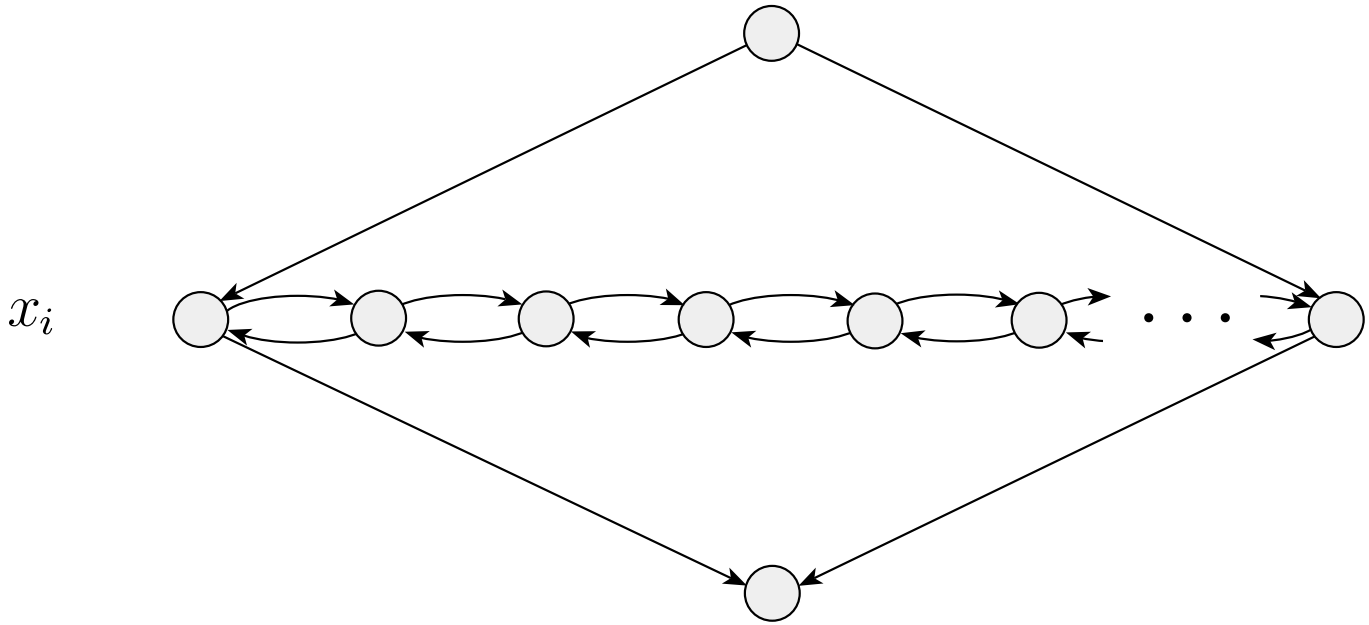
The function f is called the *reduction* from A to B

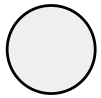
To test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$



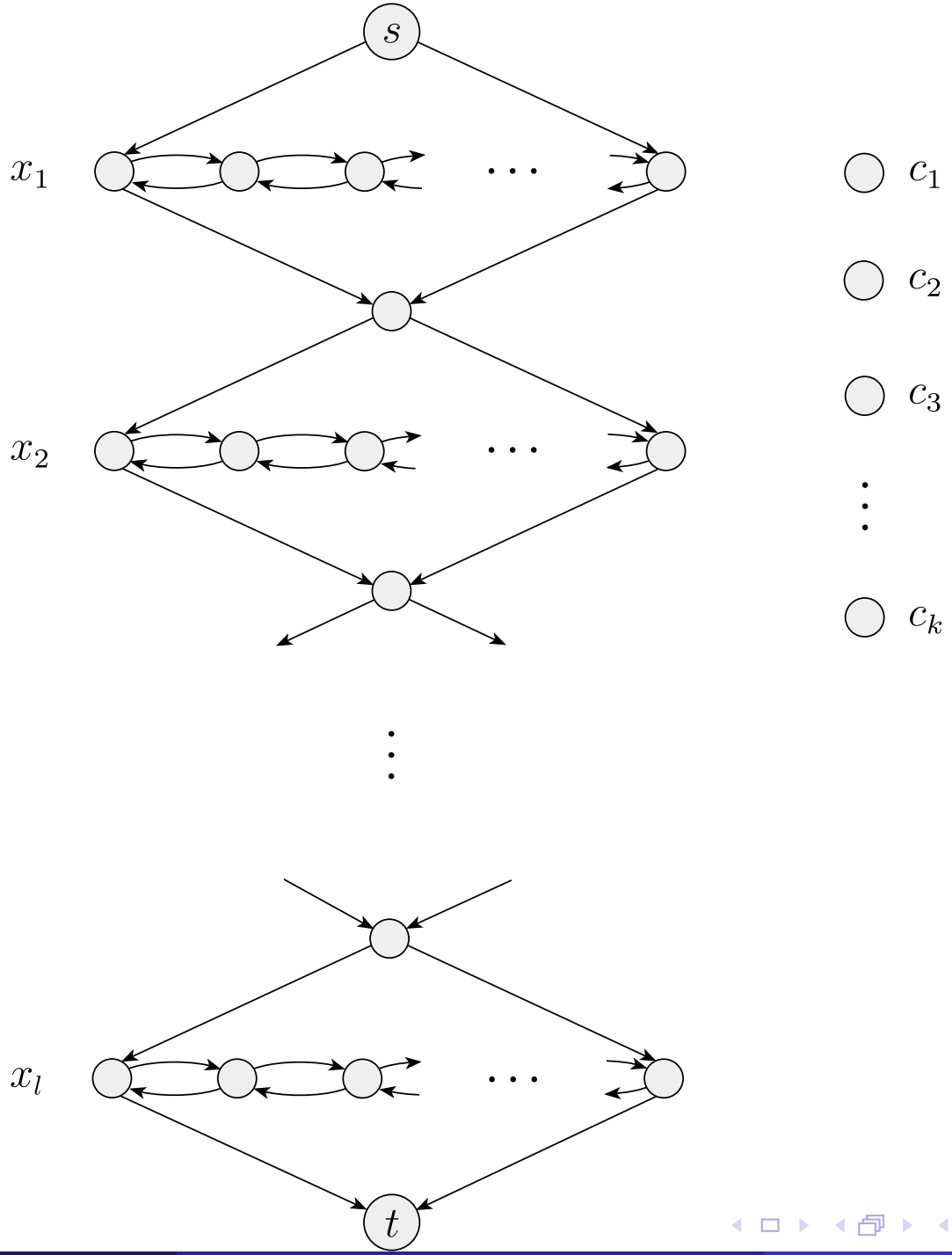

#
q_0
w_1
w_2
. . .
w_n
␣
. . .
␣
#
start configuration
second configuration
window
n^k
n^kth configuration
n^k

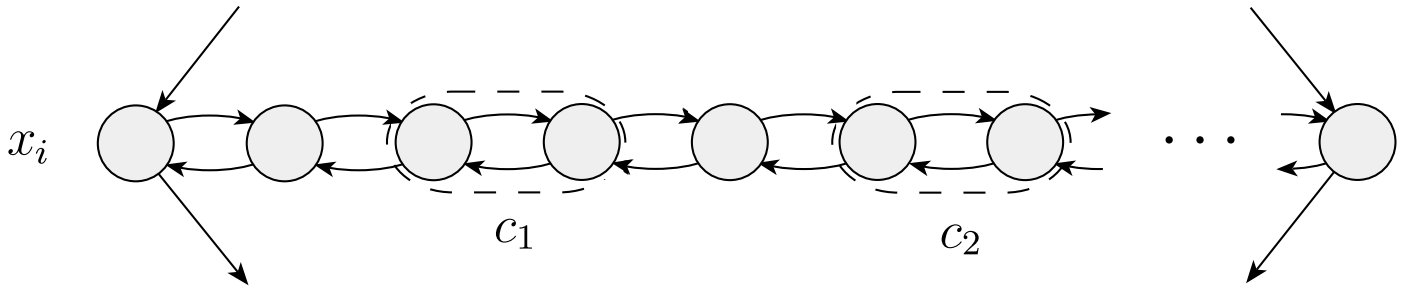
TEST

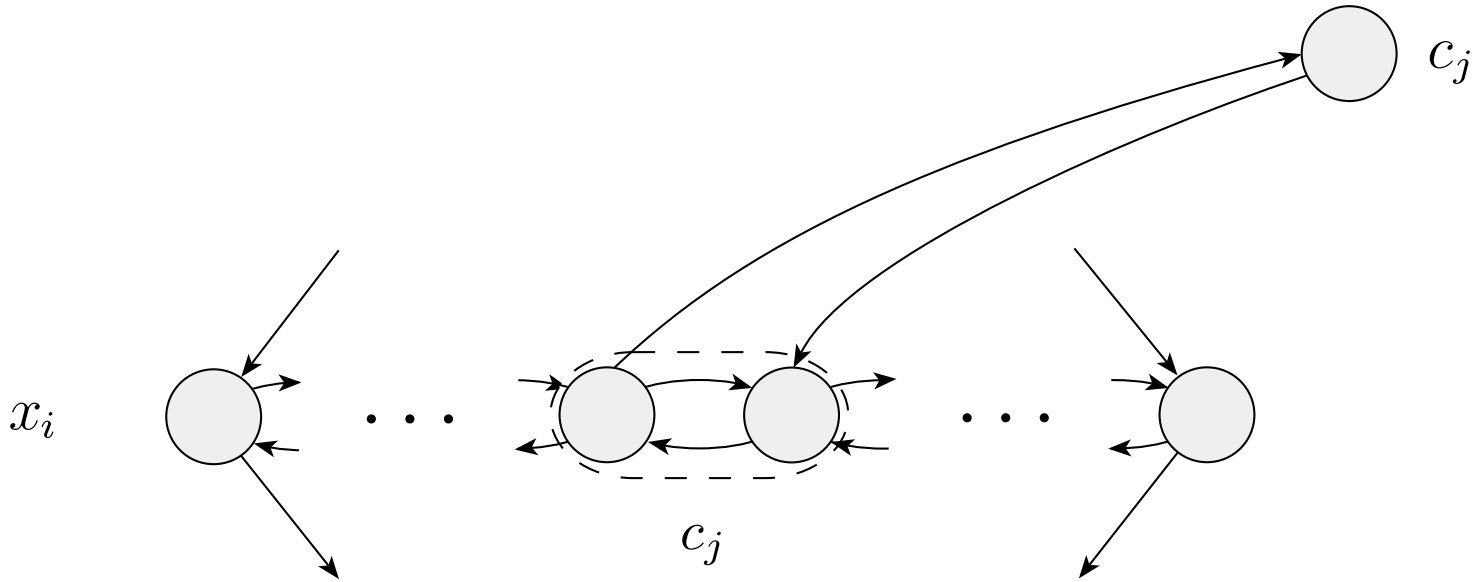

x_i



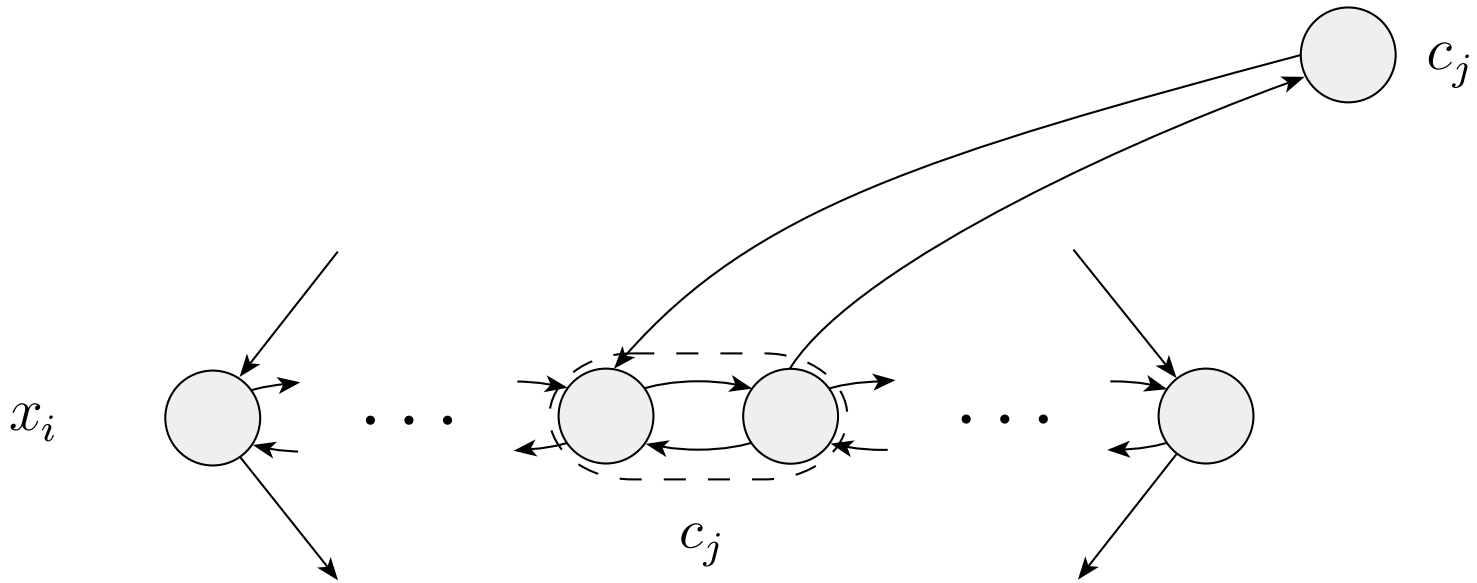
c_j


s
x_1
x_2
x_l
t
c_1
c_2
c_3
c_k

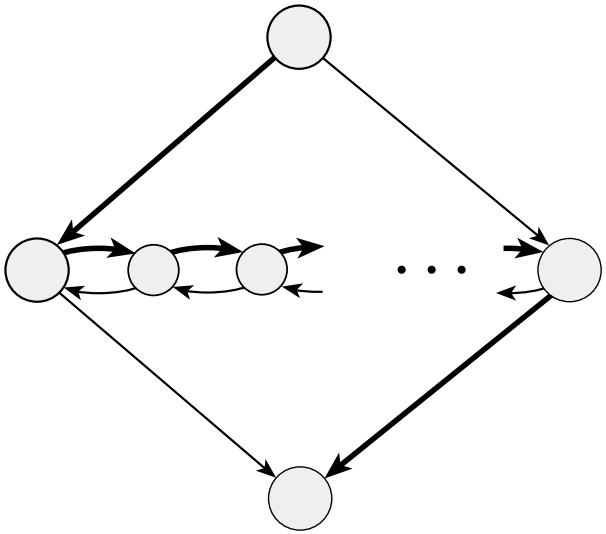

x_i
c_1
c_2
. . .



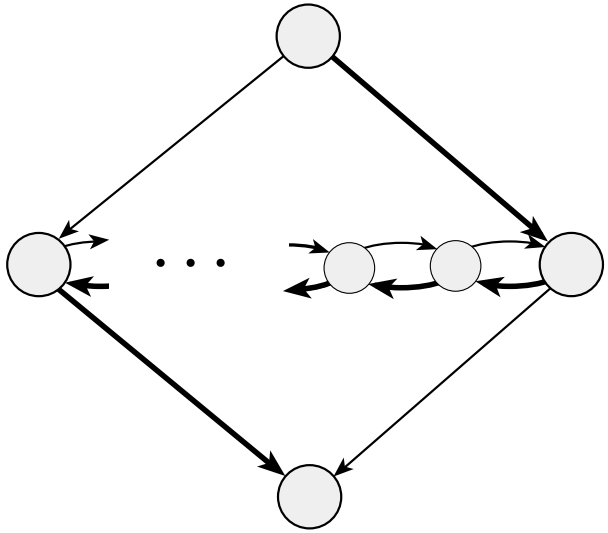
x_i in c_j



$\overline{x_i}$ in c_j

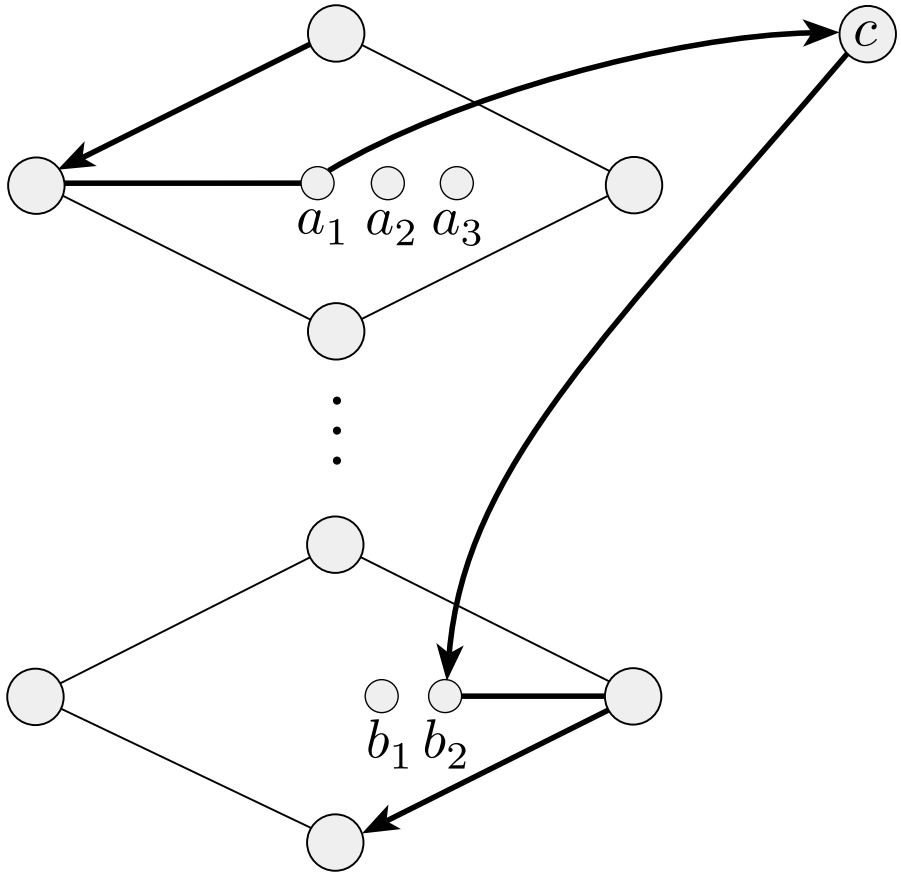


zig-zag

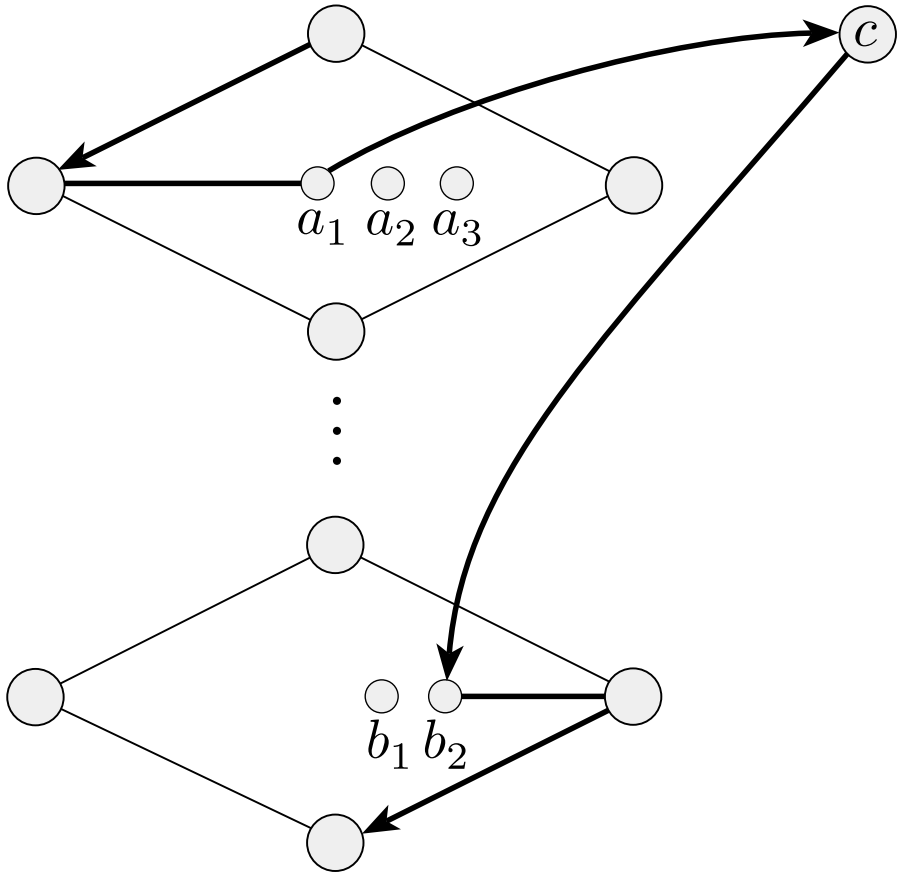


zag-zig

- If x_i is assigned *TRUE*, the path *zig-zags* through the corresponding diamond.
- If x_i is assigned *FALSE*, the path *zag-zigs*.

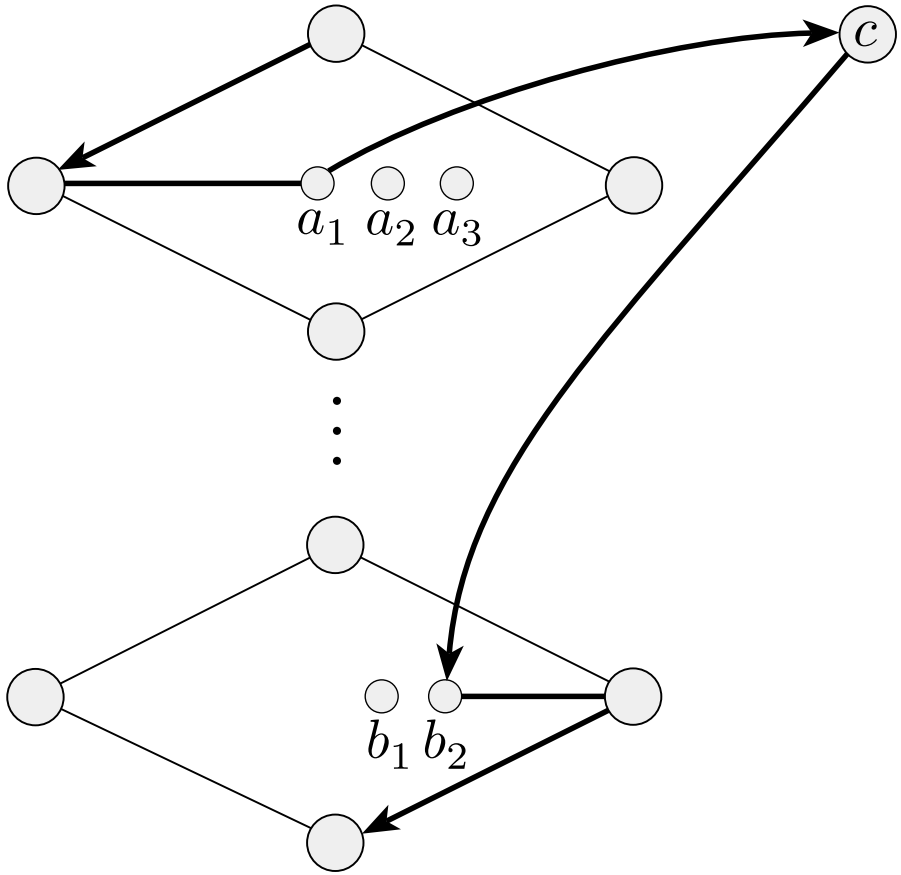


If that occurs, either a_2 or a_3 must be a separator node. Back



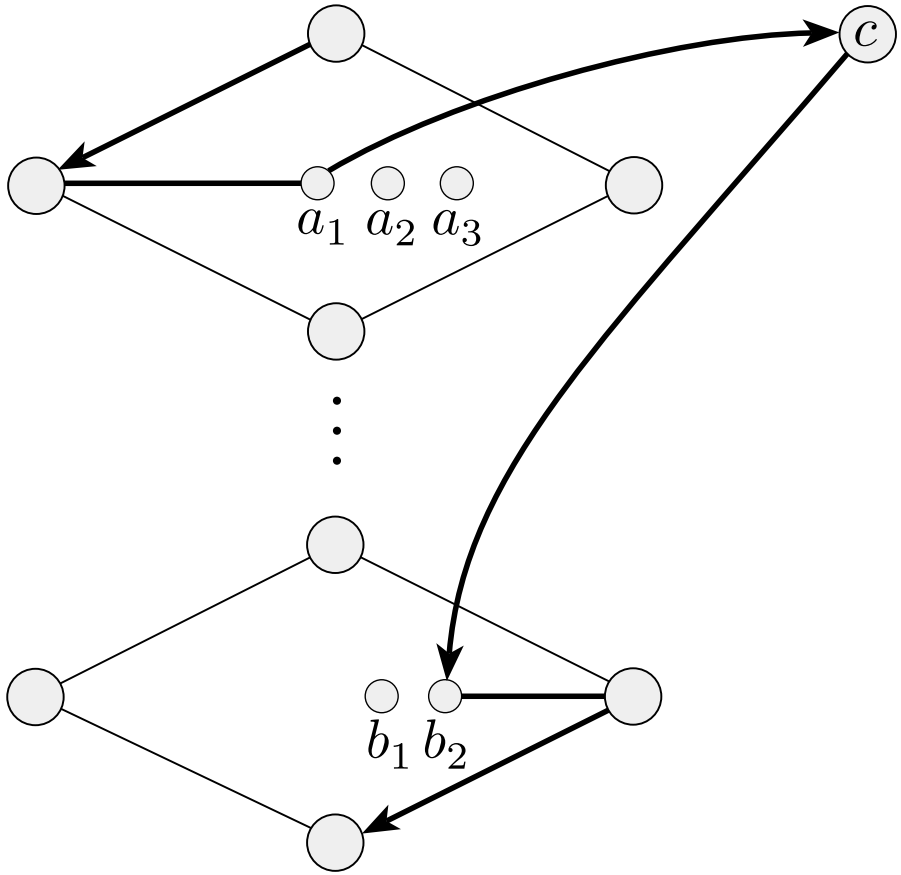
If that occurs, either a_2 or a_3 must be a separator node. Back

- If a_2 were a separator node, the only edges entering a_2 would be from a_1 and a_3
- If a_3 were a separator node, a_1 and a_2 would be in the same clause pair, and hence the only edges entering a_2 would be from a_1, a_3, and c.



If that occurs, either a_2 or a_3 must be a separator node. Back

- If a_2 were a separator node, the only edges entering a_2 would be from a_1 and a_3
- If a_3 were a separator node, a_1 and a_2 would be in the same clause pair, and hence the only edges entering a_2 would be from a_1, a_3, and c.



If that occurs, either a_2 or a_3 must be a separator node. Back

In either case, the path could not contain node a_2.

	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

- The digit of y_i in column c_j is 1, if clause c_j contains literal x_i
- The digit of z_i in column c_j is 1, if clause c_j contains literal $\overline{x_i}$

Sample clauses:

$(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3 \vee \dots) \wedge \dots$
$\cdots \wedge (\overline{x_3} \vee \cdots \vee \dots)$