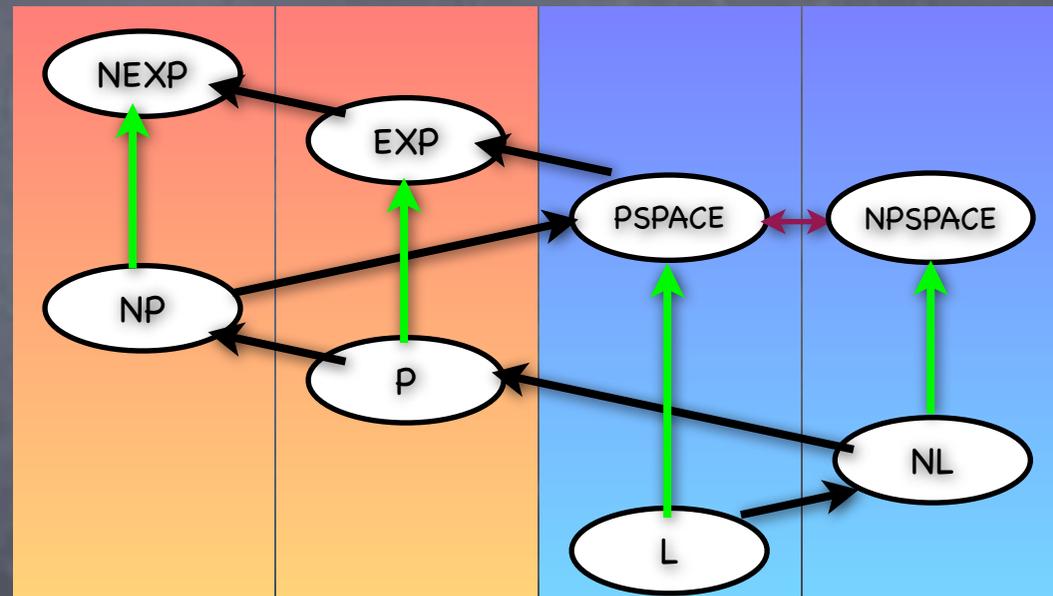


Computational Complexity

Lecture 6

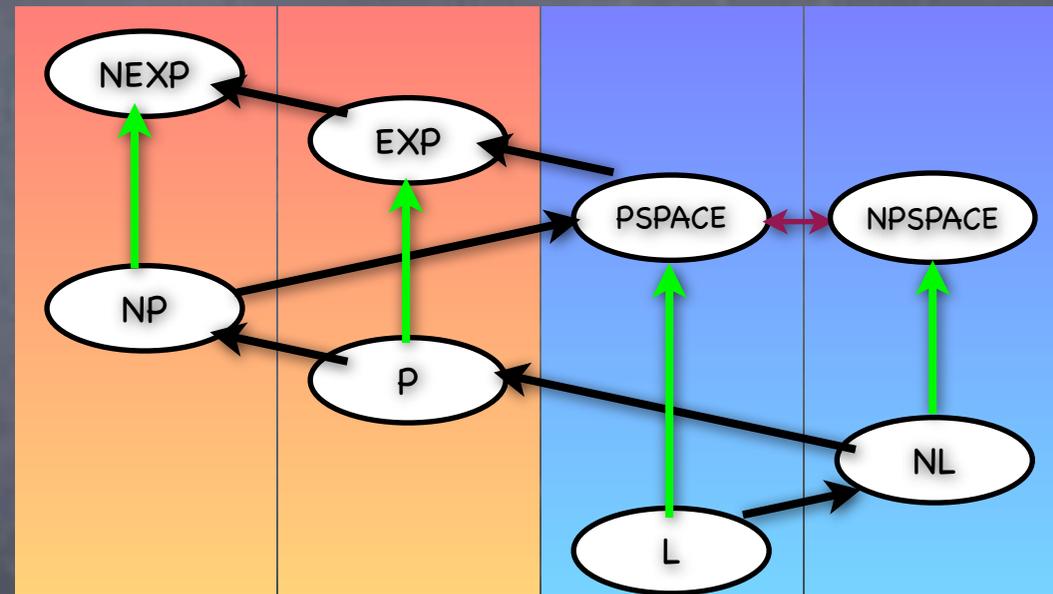
NL-Completeness and $NL=co-NL$

Story, so far



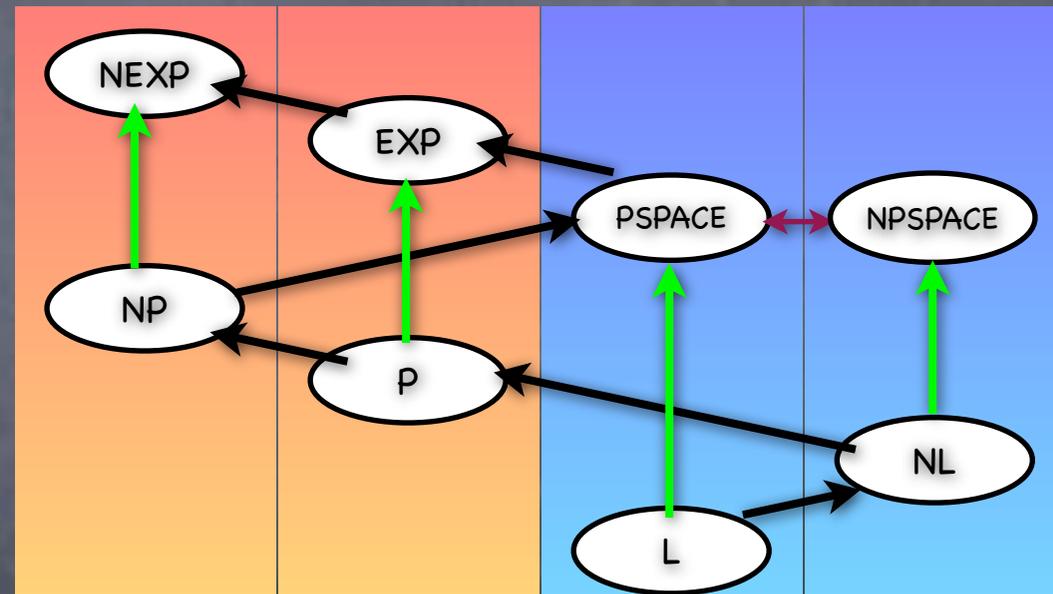
Story, so far

- Time/Space Hierarchies



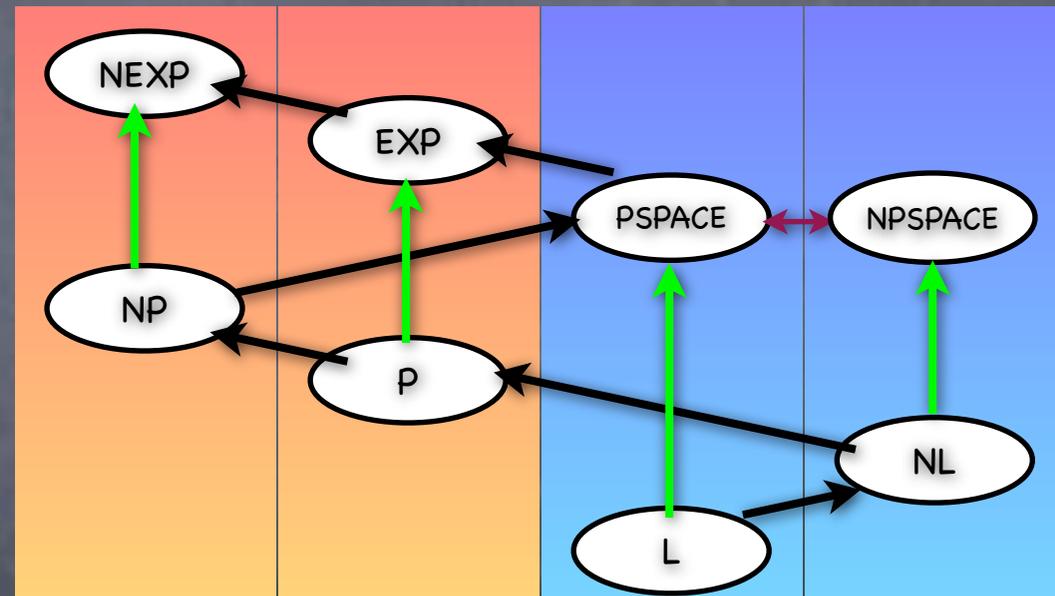
Story, so far

- Time/Space Hierarchies
- Relations across complexity measures



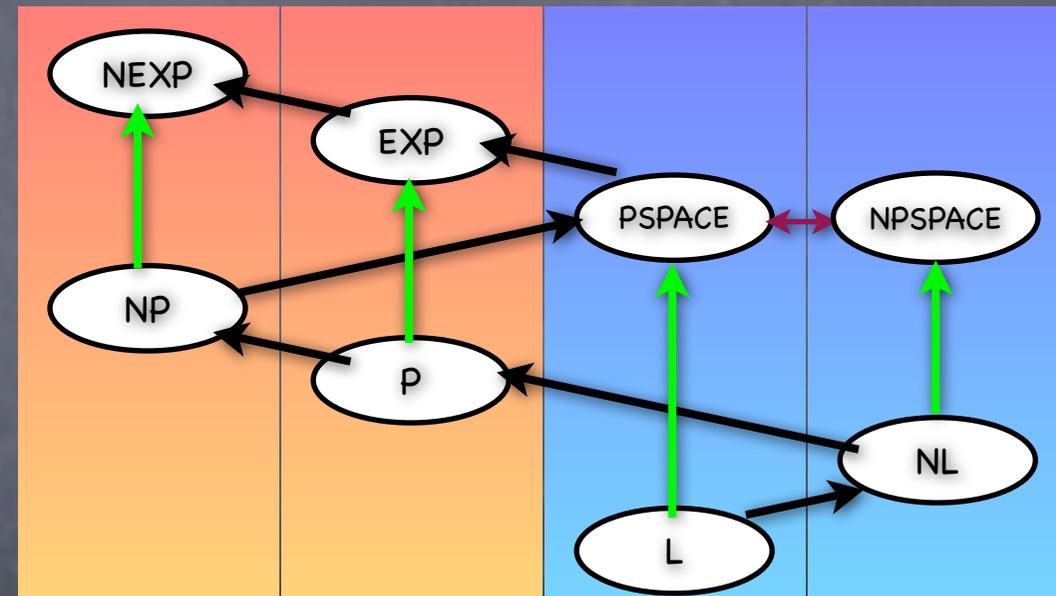
Story, so far

- Time/Space Hierarchies
- Relations across complexity measures
- SAT is NP-complete, TQBF is PSPACE-complete



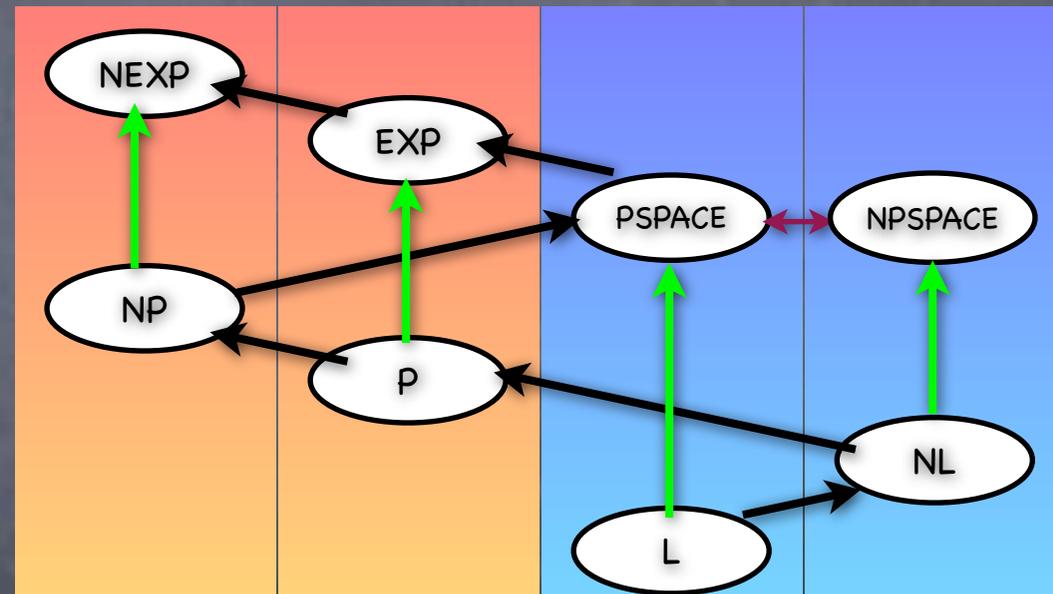
Story, so far

- Time/Space Hierarchies
- Relations across complexity measures
- SAT is NP-complete, TQBF is PSPACE-complete
- Today



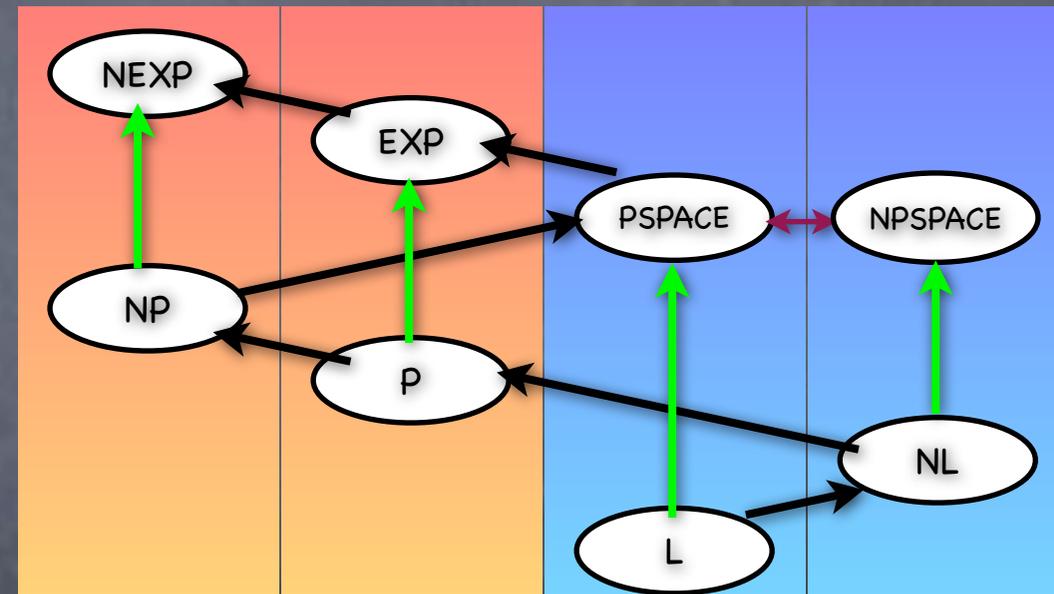
Story, so far

- Time/Space Hierarchies
- Relations across complexity measures
- SAT is NP-complete, TQBF is PSPACE-complete
- Today
 - Log-space reductions



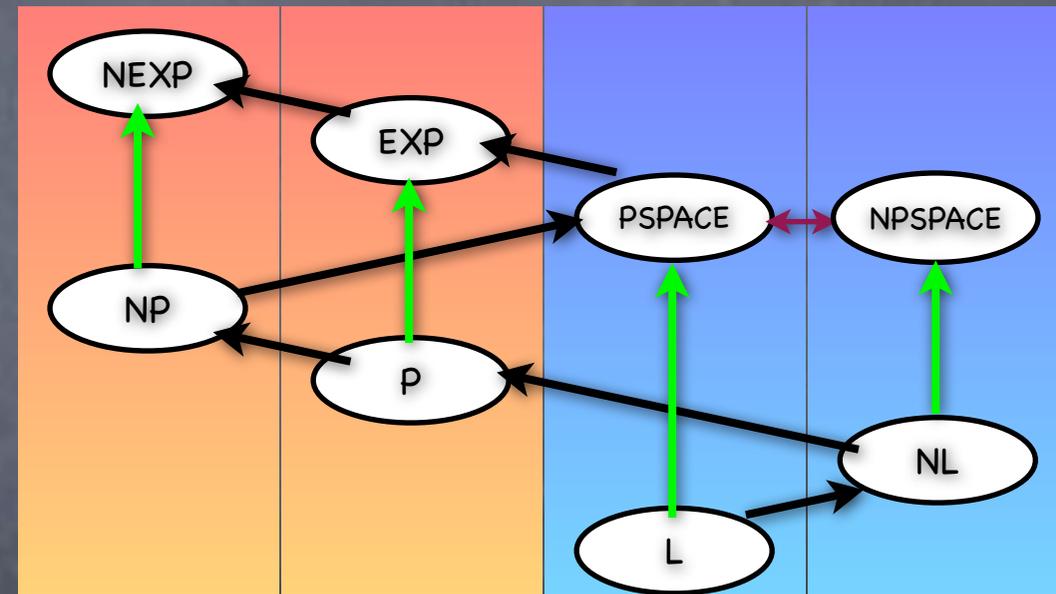
Story, so far

- Time/Space Hierarchies
- Relations across complexity measures
- SAT is NP-complete, TQBF is PSPACE-complete
- Today
 - Log-space reductions
 - An NL-complete language: PATH



Story, so far

- Time/Space Hierarchies
- Relations across complexity measures
- SAT is NP-complete, TQBF is PSPACE-complete
- Today
 - Log-space reductions
 - An NL-complete language: PATH
 - NSPACE = co-NSPACE (one less kind to worry about!)



NL-completeness

NL-completeness

- For any two (non-trivial) languages L_1, L_2 in \mathcal{P} , $L_2 \leq_p L_1$

NL-completeness

- For any two (non-trivial) languages L_1, L_2 in \mathcal{P} , $L_2 \leq_p L_1$
 - So if $X \subseteq \mathcal{P}$, then all non-trivial languages in X are X -complete (w.r.t \leq_p)

NL-completeness

- For any two (non-trivial) languages L_1, L_2 in \mathcal{P} , $L_2 \leq_p L_1$
 - So if $X \subseteq \mathcal{P}$, then all non-trivial languages in X are X -complete (w.r.t \leq_p)
- Need a tighter notion of reduction to capture “(almost) as hard as it gets” within X

Log-Space Reduction

Log-Space Reduction

- Many-one reduction: $L_2 \leq_L L_1$ if there is a TM, M which maps its input x to $f(x)$ such that

Log-Space Reduction

- Many-one reduction: $L_2 \leq_L L_1$ if there is a TM, M which maps its input x to $f(x)$ such that
 - $x \in L_2 \Rightarrow f(x) \in L_1$ and $x \notin L_2 \Rightarrow f(x) \notin L_1$

Log-Space Reduction

- Many-one reduction: $L_2 \leq_L L_1$ if there is a TM, M which maps its input x to $f(x)$ such that
 - $x \in L_2 \Rightarrow f(x) \in L_1$ and $x \notin L_2 \Rightarrow f(x) \notin L_1$
 - M uses only $O(\log|x|)$ work-tape

Log-Space Reduction

- Many-one reduction: $L_2 \leq_L L_1$ if there is a TM, M which maps its input x to $f(x)$ such that
 - $x \in L_2 \Rightarrow f(x) \in L_1$ and $x \notin L_2 \Rightarrow f(x) \notin L_1$
 - M uses **only $O(\log|x|)$ work-tape**
 - Is allowed to have a **write-only output tape**, because $|f(x)|$ may be $\text{poly}(|x|)$

Log-Space Reduction

- Many-one reduction: $L_2 \leq_L L_1$ if there is a TM, M which maps its input x to $f(x)$ such that
 - $x \in L_2 \Rightarrow f(x) \in L_1$ and $x \notin L_2 \Rightarrow f(x) \notin L_1$
 - M uses **only $O(\log|x|)$ work-tape**
 - Is allowed to have a **write-only output tape**, because $|f(x)|$ may be $\text{poly}(|x|)$
 - Equivalently: f “implicitly computable” in log-space

Log-Space Reduction

- Many-one reduction: $L_2 \leq_L L_1$ if there is a TM, M which maps its input x to $f(x)$ such that
 - $x \in L_2 \Rightarrow f(x) \in L_1$ and $x \notin L_2 \Rightarrow f(x) \notin L_1$
 - M uses **only $O(\log|x|)$ work-tape**
 - Is allowed to have a **write-only output tape**, because $|f(x)|$ may be $\text{poly}(|x|)$
 - Equivalently: f “implicitly computable” in log-space
 - A log-space machine M' to output the bit $f_i(x)$ on input (x,i)

Log-Space Reduction

- Many-one reduction: $L_2 \leq_L L_1$ if there is a TM, M which maps its input x to $f(x)$ such that
 - $x \in L_2 \Rightarrow f(x) \in L_1$ and $x \notin L_2 \Rightarrow f(x) \notin L_1$
 - M uses **only $O(\log|x|)$ work-tape**
 - Is allowed to have a **write-only output tape**, because $|f(x)|$ may be $\text{poly}(|x|)$
- Equivalently: f “implicitly computable” in log-space
 - A log-space machine M' to output the bit $f_i(x)$ on input (x,i)
 - M' from M : keep a counter and output only the i^{th} bit

Log-Space Reduction

- Many-one reduction: $L_2 \leq_L L_1$ if there is a TM, M which maps its input x to $f(x)$ such that
 - $x \in L_2 \Rightarrow f(x) \in L_1$ and $x \notin L_2 \Rightarrow f(x) \notin L_1$
 - M uses **only $O(\log|x|)$ work-tape**
 - Is allowed to have a **write-only output tape**, because $|f(x)|$ may be $\text{poly}(|x|)$
- Equivalently: f “implicitly computable” in log-space
 - A log-space machine M' to output the bit $f_i(x)$ on input (x,i)
 - M' from M : keep a counter and output only the i^{th} bit
 - M from M' : keep a counter and repeatedly call M on each i

Log-Space Reduction

- Many-one reduction: $L_2 \leq_L L_1$ if there is a TM, M which maps its input x to $f(x)$ such that
 - $x \in L_2 \Rightarrow f(x) \in L_1$ and $x \notin L_2 \Rightarrow f(x) \notin L_1$
 - M uses **only $O(\log|x|)$ work-tape**
 - Is allowed to have a **write-only output tape**, because $|f(x)|$ may be $\text{poly}(|x|)$
- Equivalently: f "implicitly computable" in log-space
 - A log-space machine M' to output the bit $f_i(x)$ on input (x,i)
 - M' from M : keep a counter and output only the i^{th} bit
 - M from M' : keep a counter and repeatedly call M on each i

Not suitable for use as a subroutine

Log-Space Reduction

Log-Space Reduction

- Log-space reductions “compose”: $L_2 \leq_L L_1 \leq_L L_0 \Rightarrow L_2 \leq_L L_0$

Log-Space Reduction

- Log-space reductions “compose”: $L_2 \leq_L L_1 \leq_L L_0 \Rightarrow L_2 \leq_L L_0$
- Given M_{2-1} and M_{1-0} build M_{2-0} :

Log-Space Reduction

- Log-space reductions “compose”: $L_2 \leq_L L_1 \leq_L L_0 \Rightarrow L_2 \leq_L L_0$
 - Given M_{2-1} and M_{1-0} build M_{2-0} :
 - Start running M_{1-0} without input. When it wants to read i^{th} bit of input, run M_{2-1} (with a counter) to get the i^{th} bit of its output

Log-Space Reduction

- Log-space reductions “compose”: $L_2 \leq_L L_1 \leq_L L_0 \Rightarrow L_2 \leq_L L_0$
 - Given M_{2-1} and M_{1-0} build M_{2-0} :
 - Start running M_{1-0} without input. When it wants to read i^{th} bit of input, run M_{2-1} (with a counter) to get the i^{th} bit of its output
 - Space needed: $O(\log(|f(x)|) + \log(|x|)) = O(\log(|x|))$, because $|f(x)|$ is $\text{poly}(|x|)$

Log-Space Reduction

- Log-space reductions “compose”: $L_2 \leq_L L_1 \leq_L L_0 \Rightarrow L_2 \leq_L L_0$
 - Given M_{2-1} and M_{1-0} build M_{2-0} :
 - Start running M_{1-0} without input. When it wants to read i^{th} bit of input, run M_{2-1} (with a counter) to get the i^{th} bit of its output
 - Space needed: $O(\log(|f(x)|) + \log(|x|)) = O(\log(|x|))$, because $|f(x)|$ is $\text{poly}(|x|)$
- Similarly, L (the class of problems decidable in log-space) is downward closed under log-space reductions

Log-Space Reduction

- Log-space reductions “compose”: $L_2 \leq_L L_1 \leq_L L_0 \Rightarrow L_2 \leq_L L_0$
 - Given M_{2-1} and M_{1-0} build M_{2-0} :
 - Start running M_{1-0} without input. When it wants to read i^{th} bit of input, run M_{2-1} (with a counter) to get the i^{th} bit of its output
 - Space needed: $O(\log(|f(x)|) + \log(|x|)) = O(\log(|x|))$, because $|f(x)|$ is $\text{poly}(|x|)$
- Similarly, L (the class of problems decidable in log-space) is downward closed under log-space reductions
 - $L_2 \leq_L L_1 \in L \Rightarrow L_2 \in L$

NL-completeness

NL-completeness

- NL has languages of the form $L_1 = \{ x \mid \exists w, |w| < \text{poly}(|x|), (x;w) \in L_1' \}$ where L_1' can be decided deterministically in logspace, with w in a read-once tape

NL-completeness

- NL has languages of the form $L_1 = \{ x \mid \exists w, |w| < \text{poly}(|x|), (x;w) \in L_1' \}$ where L_1' can be decided deterministically in logspace, with w in a read-once tape
- L_0 is NL-Hard if for all L_1 in NL, $L_1 \leq_L L_0$

NL-completeness

- NL has languages of the form $L_1 = \{ x \mid \exists w, |w| < \text{poly}(|x|), (x;w) \in L_1' \}$ where L_1' can be decided deterministically in logspace, with w in a read-once tape
- L_0 is NL-Hard if for all L_1 in NL, $L_1 \leq_L L_0$
- L_0 is NL-complete if it is NL-hard and is in NL

NL-completeness

- NL has languages of the form $L_1 = \{ x \mid \exists w, |w| < \text{poly}(|x|), (x;w) \in L_1' \}$ where L_1' can be decided deterministically in logspace, with w in a read-once tape
- L_0 is NL-Hard if for all L_1 in NL, $L_1 \leq_L L_0$
- L_0 is NL-complete if it is NL-hard and is in NL
- Can construct trivial NL-complete language

NL-completeness

- NL has languages of the form $L_1 = \{ x \mid \exists w, |w| < \text{poly}(|x|), (x;w) \in L_1' \}$ where L_1' can be decided deterministically in logspace, with w in a read-once tape
- L_0 is NL-Hard if for all L_1 in NL, $L_1 \leq_L L_0$
- L_0 is NL-complete if it is NL-hard and is in NL
- Can construct trivial NL-complete language
 - $\{ (M, x, 1^n, 1^s) \mid \exists w, |w| < n, M \text{ accepts } (x;w) \text{ in space } \log(s) \}$
(where M takes w in a read-once tape)

NL-completeness

- NL has languages of the form $L_1 = \{ x \mid \exists w, |w| < \text{poly}(|x|), (x;w) \in L_1' \}$ where L_1' can be decided deterministically in logspace, with w in a read-once tape
- L_0 is NL-Hard if for all L_1 in NL, $L_1 \leq_L L_0$
- L_0 is NL-complete if it is NL-hard and is in NL
- Can construct trivial NL-complete language
 - $\{ (M, x, 1^n, 1^s) \mid \exists w, |w| < n, M \text{ accepts } (x;w) \text{ in space } \log(s) \}$
(where M takes w in a read-once tape)
- Interesting NLC language: PATH

Directed Path

Directed Path

- $\text{PATH} = \{(G,s,t) \mid G \text{ a directed graph with a path from } s \text{ to } t\}$

Directed Path

- $\text{PATH} = \{(G,s,t) \mid G \text{ a directed graph with a path from } s \text{ to } t\}$
 - G using some representation, of size say, n^2 ($n=\#\text{vertices}$)

Directed Path

- $\text{PATH} = \{(G,s,t) \mid G \text{ a directed graph with a path from } s \text{ to } t\}$
- G using some representation, of size say, n^2 ($n = \# \text{vertices}$)
- Such that, if two vertices x,y on work-tape, can read the input tape to check for edge (x,y)

Directed Path

- $PATH = \{(G,s,t) \mid G \text{ a directed graph with a path from } s \text{ to } t\}$
 - G using some representation, of size say, n^2 ($n = \# \text{vertices}$)
 - Such that, if two vertices x,y on work-tape, can read the input tape to check for edge (x,y)
- $PATH$ in NL

Directed Path

- $\text{PATH} = \{(G,s,t) \mid G \text{ a directed graph with a path from } s \text{ to } t\}$
 - G using some representation, of size say, n^2 ($n = \# \text{vertices}$)
 - Such that, if two vertices x,y on work-tape, can read the input tape to check for edge (x,y)
- PATH in NL
 - Certificate w is the path ($\text{poly}(n)$ long certificate)

Directed Path

- $\text{PATH} = \{(G,s,t) \mid G \text{ a directed graph with a path from } s \text{ to } t\}$
 - G using some representation, of size say, n^2 ($n = \# \text{vertices}$)
 - Such that, if two vertices x,y on work-tape, can read the input tape to check for edge (x,y)
- PATH in NL
 - Certificate w is the path ($\text{poly}(n)$ long certificate)
 - Need to verify adjacent vertices are connected: need keep only two vertices on the work-tape at a time

Directed Path

- $\text{PATH} = \{(G,s,t) \mid G \text{ a directed graph with a path from } s \text{ to } t\}$
 - G using some representation, of size say, n^2 ($n = \# \text{vertices}$)
 - Such that, if two vertices x,y on work-tape, can read the input tape to check for edge (x,y)
- PATH in NL
 - Certificate w is the path ($\text{poly}(n)$ long certificate)
 - Need to verify adjacent vertices are connected: need keep only two vertices on the work-tape at a time
 - Note: w is scanned only once

Seen PATH before?

Seen PATH before?

- In proving $\text{NSPACE}(S(n)) \subseteq \text{DTIME}(2^{O(S(n))})$ (e.g. $\text{NL} \subseteq \text{P}$)

Seen PATH before?

- In proving $\text{NSPACE}(S(n)) \subseteq \text{DTIME}(2^{O(S(n))})$ (e.g. $\text{NL} \subseteq \text{P}$)
 - Every problem in NL Karp reduces to PATH

Seen PATH before?

- In proving $\text{NSPACE}(S(n)) \subseteq \text{DTIME}(2^{O(S(n))})$ (e.g. $\text{NL} \subseteq \text{P}$)
 - Every problem in NL Karp reduces to PATH
 - $\text{PATH} \in \text{P}$

Seen PATH before?

- In proving $\text{NSPACE}(S(n)) \subseteq \text{DTIME}(2^{O(S(n))})$ (e.g. $\text{NL} \subseteq \text{P}$)
 - Every problem in NL Karp reduces to PATH
 - $\text{PATH} \in \text{P}$
- In Savitch's theorem

Seen PATH before?

- In proving $\text{NSPACE}(S(n)) \subseteq \text{DTIME}(2^{O(S(n))})$ (e.g. $\text{NL} \subseteq \text{P}$)
 - Every problem in NL Karp reduces to PATH
 - $\text{PATH} \in \text{P}$
- In Savitch's theorem
 - $\text{PATH} \in \text{DSPACE}(\log^2(n))$

PATH is NL-complete

PATH is NL-complete

- Log-space reducing any NL language L_1 to PATH

PATH is NL-complete

- Log-space reducing any NL language L_1 to PATH
 - Given input x , output (G,s,t) where G is the configuration graph $G(M,x)$, where M is the NTM accepting L_1 , and s,t are start, accept configurations

PATH is NL-complete

- Log-space reducing any NL language L_1 to PATH
 - Given input x , output (G,s,t) where G is the configuration graph $G(M,x)$, where M is the NTM accepting L_1 , and s,t are start, accept configurations
 - **Outputting G** : Cycle through all pairs of configurations, checking if there is an edge between them, outputting 0 or 1 in the adjacency matrix

PATH is NL-complete

- Log-space reducing any NL language L_1 to PATH
 - Given input x , output (G,s,t) where G is the configuration graph $G(M,x)$, where M is the NTM accepting L_1 , and s,t are start, accept configurations
 - **Outputting G** : Cycle through all pairs of configurations, checking if there is an edge between them, outputting 0 or 1 in the adjacency matrix
 - Edge checking done using M 's transition table

PATH is NL-complete

- Log-space reducing any NL language L_1 to PATH
 - Given input x , output (G,s,t) where G is the configuration graph $G(M,x)$, where M is the NTM accepting L_1 , and s,t are start, accept configurations
 - **Outputting G** : Cycle through all pairs of configurations, checking if there is an edge between them, outputting 0 or 1 in the adjacency matrix
 - Edge checking done using M 's transition table
 - Need to store only two configurations at a time in the work-tape

PATH is NL-complete

- Log-space reducing any NL language L_1 to PATH
 - Given input x , output (G,s,t) where G is the configuration graph $G(M,x)$, where M is the NTM accepting L_1 , and s,t are start, accept configurations
 - **Outputting G** : Cycle through all pairs of configurations, checking if there is an edge between them, outputting 0 or 1 in the adjacency matrix
 - Edge checking done using M 's transition table
 - Need to store only two configurations at a time in the work-tape
- Note: in fact **$O(S)$ -space reduction from $L \in \text{NSPACE}(S)$ to PATH**

If $\text{PATH} \in \text{co-NL}$

If $\text{PATH} \in \text{co-NL}$

- If $\text{PATH} \in \text{co-NL}$, then $\text{co-NL} \subseteq \text{NL}$

If $\text{PATH} \in \text{co-NL}$

- If $\text{PATH} \in \text{co-NL}$, then $\text{co-NL} \subseteq \text{NL}$
 - For any $L \in \text{co-NL}$, we have $L \leq_L \text{PATH}^c$ (as $L^c \leq_L \text{PATH}$), and if $\text{PATH}^c \in \text{NL}$, then $L \in \text{NL}$ (NL is downward closed under \leq_L)

If $\text{PATH} \in \text{co-NL}$

- If $\text{PATH} \in \text{co-NL}$, then $\text{co-NL} \subseteq \text{NL}$
 - For any $L \in \text{co-NL}$, we have $L \leq_L \text{PATH}^c$ (as $L^c \leq_L \text{PATH}$), and if $\text{PATH}^c \in \text{NL}$, then $L \in \text{NL}$ (NL is downward closed under \leq_L)
 - Implies $\text{co-NL} = \text{NL}$ (why?)

If $\text{PATH} \in \text{co-NL}$

- If $\text{PATH} \in \text{co-NL}$, then $\text{co-NL} \subseteq \text{NL}$
 - For any $L \in \text{co-NL}$, we have $L \leq_L \text{PATH}^c$ (as $L^c \leq_L \text{PATH}$), and if $\text{PATH}^c \in \text{NL}$, then $L \in \text{NL}$ (NL is downward closed under \leq_L)
 - Implies $\text{co-NL} = \text{NL}$ (why?)
 - If $Y \subseteq X$, then $\text{co-}Y \subseteq \text{co-}X$. Consider $X = \text{NL}$, $Y = \text{co-NL}$.

If $PATH \in co-NL$

If $\text{PATH} \in \text{co-NL}$

- In fact, $\text{PATH} \in \text{co-NL}$ implies $\text{co-NSPACE}(S) = \text{NSPACE}(S)$

If $\text{PATH} \in \text{co-NL}$

- In fact, $\text{PATH} \in \text{co-NL}$ implies $\text{co-NSPACE}(S) = \text{NSPACE}(S)$
 - Recall: $O(S)$ -space reduction from any $L \in \text{NSPACE}(S)$ to PATH

If $PATH \in co-NL$

- In fact, $PATH \in co-NL$ implies $co-NSPACE(S) = NSPACE(S)$
 - Recall: $O(S)$ -space reduction from any $L \in NSPACE(S)$ to $PATH$
 - i.e., from any $L' \in co-NSPACE(S)$ to $PATH^c$

If $PATH \in co-NL$

- In fact, $PATH \in co-NL$ implies $co-NSPACE(S) = NSPACE(S)$
 - Recall: $O(S)$ -space reduction from any $L \in NSPACE(S)$ to $PATH$
 - i.e., from any $L' \in co-NSPACE(S)$ to $PATH^c$
 - Size of the new instance is at most $N = 2^{O(|S|)}$

If $\text{PATH} \in \text{co-NL}$

- In fact, $\text{PATH} \in \text{co-NL}$ implies $\text{co-NSPACE}(S) = \text{NSPACE}(S)$
 - Recall: $O(S)$ -space reduction from any $L \in \text{NSPACE}(S)$ to PATH
 - i.e., from any $L' \in \text{co-NSPACE}(S)$ to PATH^c
 - Size of the new instance is at most $N = 2^{O(|S|)}$
 - $\text{PATH}^c \in \text{NL}$ implies an NTM that decides if the instance is in PATH^c in $\text{NSPACE}(\log N) = \text{NSPACE}(S)$

If $PATH \in co-NL$

- In fact, $PATH \in co-NL$ implies $co-NSPACE(S) = NSPACE(S)$
 - Recall: $O(S)$ -space reduction from any $L \in NSPACE(S)$ to $PATH$
 - i.e., from any $L' \in co-NSPACE(S)$ to $PATH^c$
 - Size of the new instance is at most $N = 2^{O(|S|)}$
 - $PATH^c \in NL$ implies an NTM that decides if the instance is in $PATH^c$ in $NSPACE(\log N) = NSPACE(S)$
 - Then $L' \in co-NSPACE(S)$ is also in $NSPACE(S)$, by composing space-bounded computations. So, $co-NSPACE(S) \subseteq NSPACE(S)$

If $PATH \in co-NL$

- In fact, $PATH \in co-NL$ implies $co-NSPACE(S) = NSPACE(S)$
 - Recall: $O(S)$ -space reduction from any $L \in NSPACE(S)$ to $PATH$
 - i.e., from any $L' \in co-NSPACE(S)$ to $PATH^c$
 - Size of the new instance is at most $N = 2^{O(|S|)}$
 - $PATH^c \in NL$ implies an NTM that decides if the instance is in $PATH^c$ in $NSPACE(\log N) = NSPACE(S)$
 - Then $L' \in co-NSPACE(S)$ is also in $NSPACE(S)$, by composing space-bounded computations. So, $co-NSPACE(S) \subseteq NSPACE(S)$
 - Hence $co-NSPACE(S) = NSPACE(S)$

If $PATH \in co-NL$

If $\text{PATH} \in \text{co-NL}$

- If $\text{PATH} \in \text{co-NL}$ then $\text{NSPACE}(S) = \text{co-NSPACE}(S)$

If $PATH \in co-NL$

- If $PATH \in co-NL$ then $NSPACE(S) = co-NSPACE(S)$
 - In particular $NL = co-NL$

If $\text{PATH} \in \text{co-NL}$

- If $\text{PATH} \in \text{co-NL}$ then $\text{NSPACE}(S) = \text{co-NSPACE}(S)$
 - In particular $\text{NL} = \text{co-NL}$
- And indeed, $\text{PATH} \in \text{co-NL}$!

If $\text{PATH} \in \text{co-NL}$

- If $\text{PATH} \in \text{co-NL}$ then $\text{NSPACE}(S) = \text{co-NSPACE}(S)$
 - In particular $\text{NL} = \text{co-NL}$
- **And indeed, $\text{PATH} \in \text{co-NL}$!**
 - There is a (polynomial sized) certificate that can be verified in log-space, that there is no path from s to t in a graph G

$\text{PATH}^c \in \text{NL}$

PATH^c ∈ NL

- Certificate for (s,t) connected is just the path

PATH^c ∈ NL

- Certificate for (s,t) connected is just the path
- What is a certificate that (s,t) not connected?

$\text{PATH}^c \in \text{NL}$

- Certificate for (s,t) connected is just the path
- What is a certificate that (s,t) not connected?
 - size c of the connected component of s , C ; a list of all $v \in C$ (with certificates) in order; and (somehow) a certificate for $c = |C|$

PATH^c ∈ NL

- Certificate for (s,t) connected is just the path
- What is a certificate that (s,t) not connected?
 - size c of the connected component of s, C; a list of all v ∈ C (with certificates) in order; and (somehow) a certificate for c = |C|
 - Log-space, one-scan verification of certified C (believing |C|): scan list, checking certificates, counting, ensuring order, and that t not in the list. Verify certificate for |C|.

PATH^c ∈ NL

- Certificate for (s,t) connected is just the path
- What is a certificate that (s,t) not connected?
 - size c of the connected component of s, C; a list of all $v \in C$ (with certificates) in order; and (somehow) a certificate for $c = |C|$
 - Log-space, one-scan verification of certified C (believing $|C|$): scan list, checking certificates, counting, ensuring order, and that t not in the list. Verify certificate for $|C|$.
 - If list has $|C|$ many $v \in C$, without repeating, list must be complete

Certificate for ICI

Certificate for $|C|$

- Let $C_i :=$ set of nodes within distance i of s . Then $C = C_N$

Certificate for $|C|$

- Let $C_i :=$ set of nodes within distance i of s . Then $C = C_N$
- Tail recursion to verify $|C_N|$:

Certificate for $|C|$

- Let $C_i :=$ set of nodes within distance i of s . Then $C = C_N$
- Tail recursion to verify $|C_N|$:
 - Read $|C_{N-1}|$, **believing it verify $|C_N|$** , forget $|C_N|$;

Certificate for $|C|$

- Let $C_i :=$ set of nodes within distance i of s . Then $C = C_N$
- Tail recursion to verify $|C_N|$:
 - Read $|C_{N-1}|$, **believing it verify $|C_N|$** , forget $|C_N|$;
 - Read $|C_{N-2}|$, **believing it verify $|C_{N-1}|$** , forget $|C_{N-1}|$; ...

Certificate for $|C|$

- Let $C_i :=$ set of nodes within distance i of s . Then $C = C_N$
- Tail recursion to verify $|C_N|$:
 - Read $|C_{N-1}|$, **believing it verify $|C_N|$** , forget $|C_N|$;
 - Read $|C_{N-2}|$, **believing it verify $|C_{N-1}|$** , forget $|C_{N-1}|$; ...
 - Base case: $|C_0|=1$

Certificate for $|C|$

- Let $C_i :=$ set of nodes within distance i of s . Then $C = C_N$
- Tail recursion to verify $|C_N|$:
 - Read $|C_{N-1}|$, **believing it verify $|C_N|$** , forget $|C_N|$;
 - Read $|C_{N-2}|$, **believing it verify $|C_{N-1}|$** , forget $|C_{N-1}|$; ...
 - Base case: $|C_0|=1$
- **Believing $|C_{i-1}|$ verify $|C_i|$** : for each vertex v certificate that $v \in C_i$ or **that $v \notin C_i$** (these certificates should be $\text{poly}(N)$ long)

Certificate for $|C|$

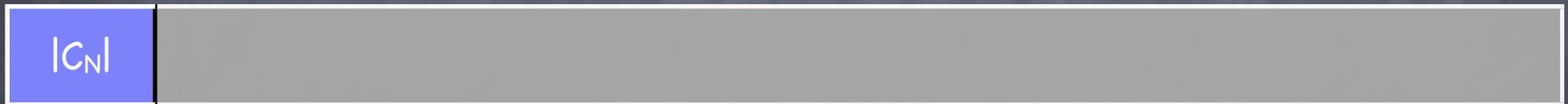
- Let $C_i :=$ set of nodes within distance i of s . Then $C = C_N$
- Tail recursion to verify $|C_N|$:
 - Read $|C_{N-1}|$, **believing it verify $|C_N|$** , forget $|C_N|$;
 - Read $|C_{N-2}|$, **believing it verify $|C_{N-1}|$** , forget $|C_{N-1}|$; ...
 - Base case: $|C_0|=1$
- **Believing $|C_{i-1}|$ verify $|C_i|$** : for each vertex v certificate that $v \in C_i$ or **that $v \notin C_i$** (these certificates should be $\text{poly}(N)$ long)
- **Certificate that $v \notin C_i$ given (i.e., believing) $|C_{i-1}|$** : list of all vertices in C_{i-1} in order, with certificates. As before verify C_{i-1} believing $|C_{i-1}|$ (scan and ensure list is correct/complete), but also check that no node in the list has v as a neighbor

Certificate for $t \notin C_N$

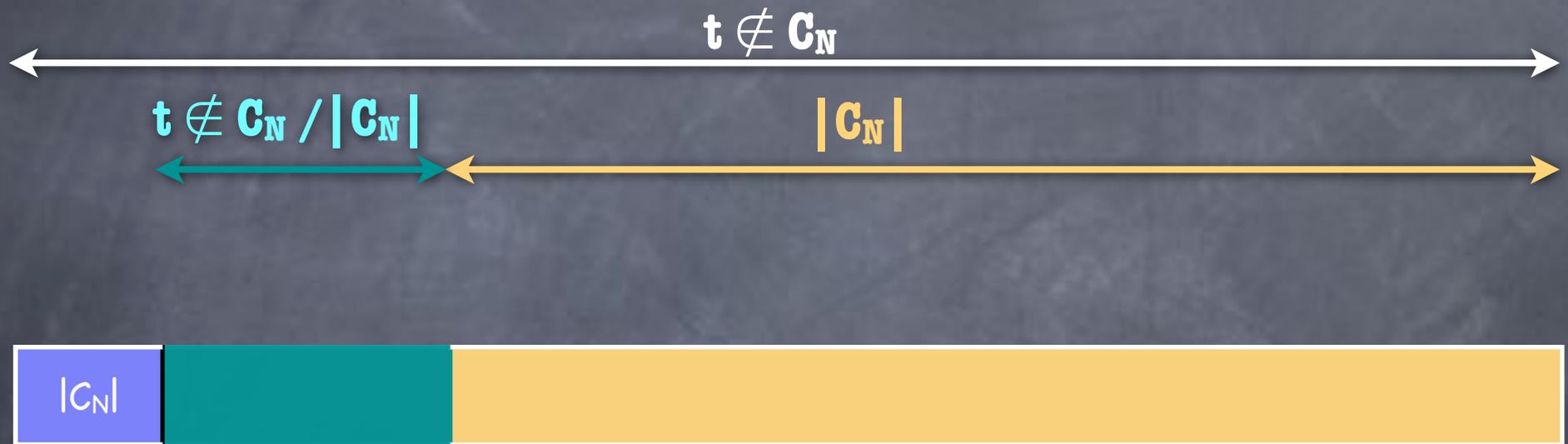
Certificate for $t \notin C_N$



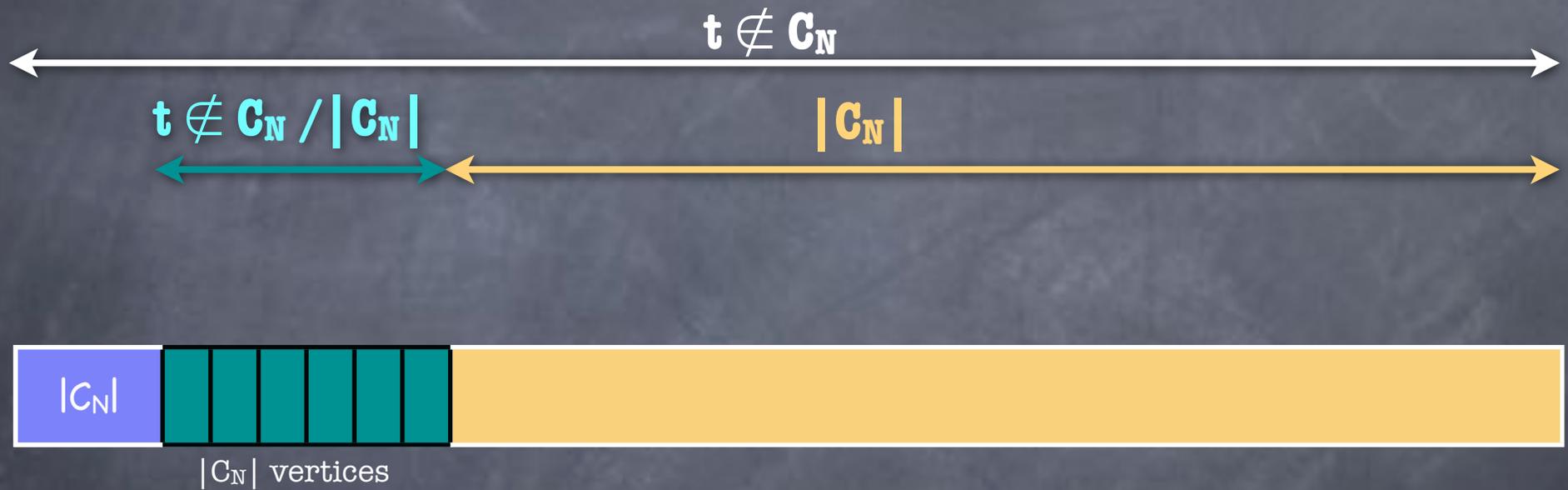
Certificate for $t \notin C_N$



Certificate for $t \notin C_N$



Certificate for $t \notin C_N$



Certificate for $t \notin C_N$



$|C_N|$ vertices



Certificate for $t \notin C_N$



$|C_N|$ vertices



$v_i \neq t$

Certificate for $t \notin C_N$

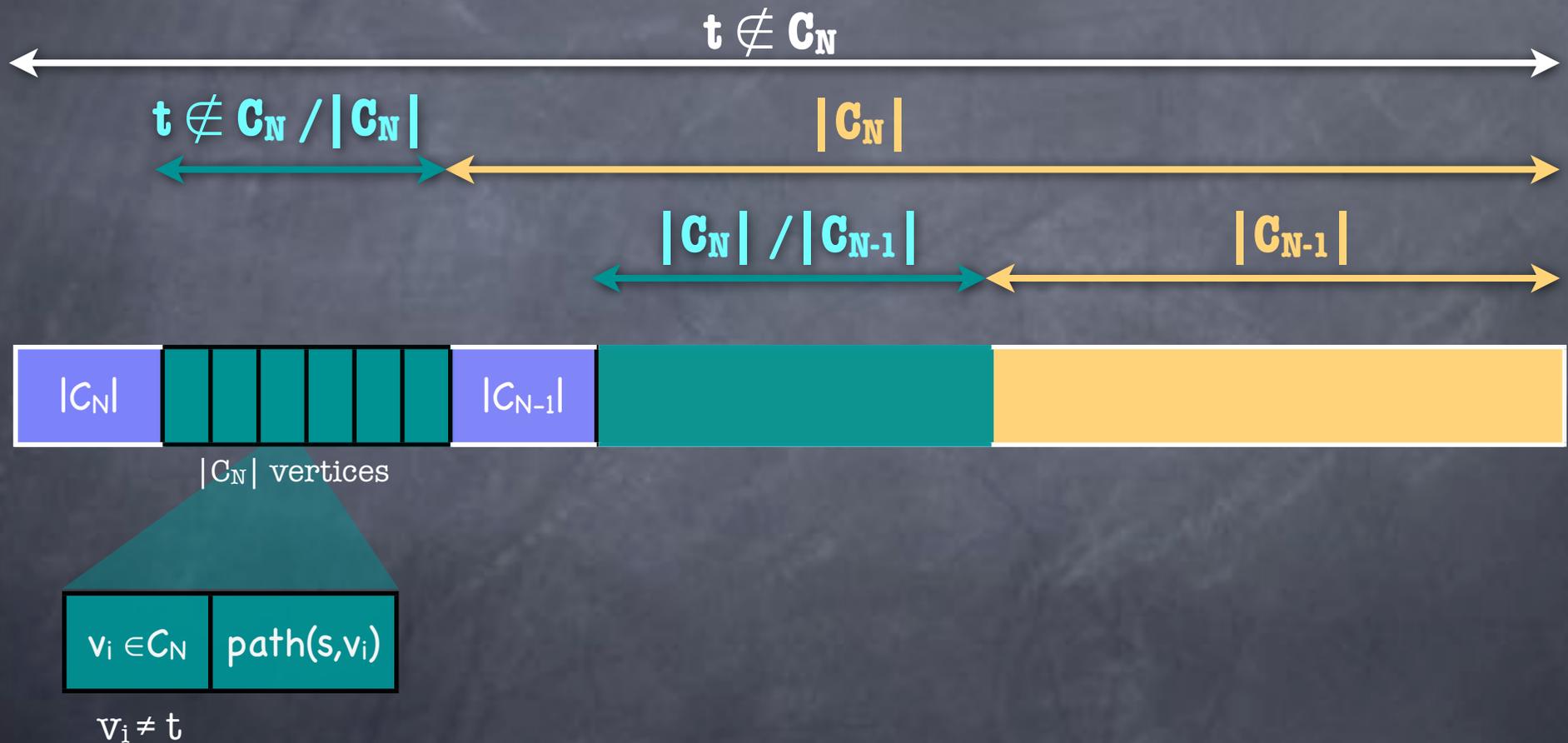


$|C_N|$ vertices

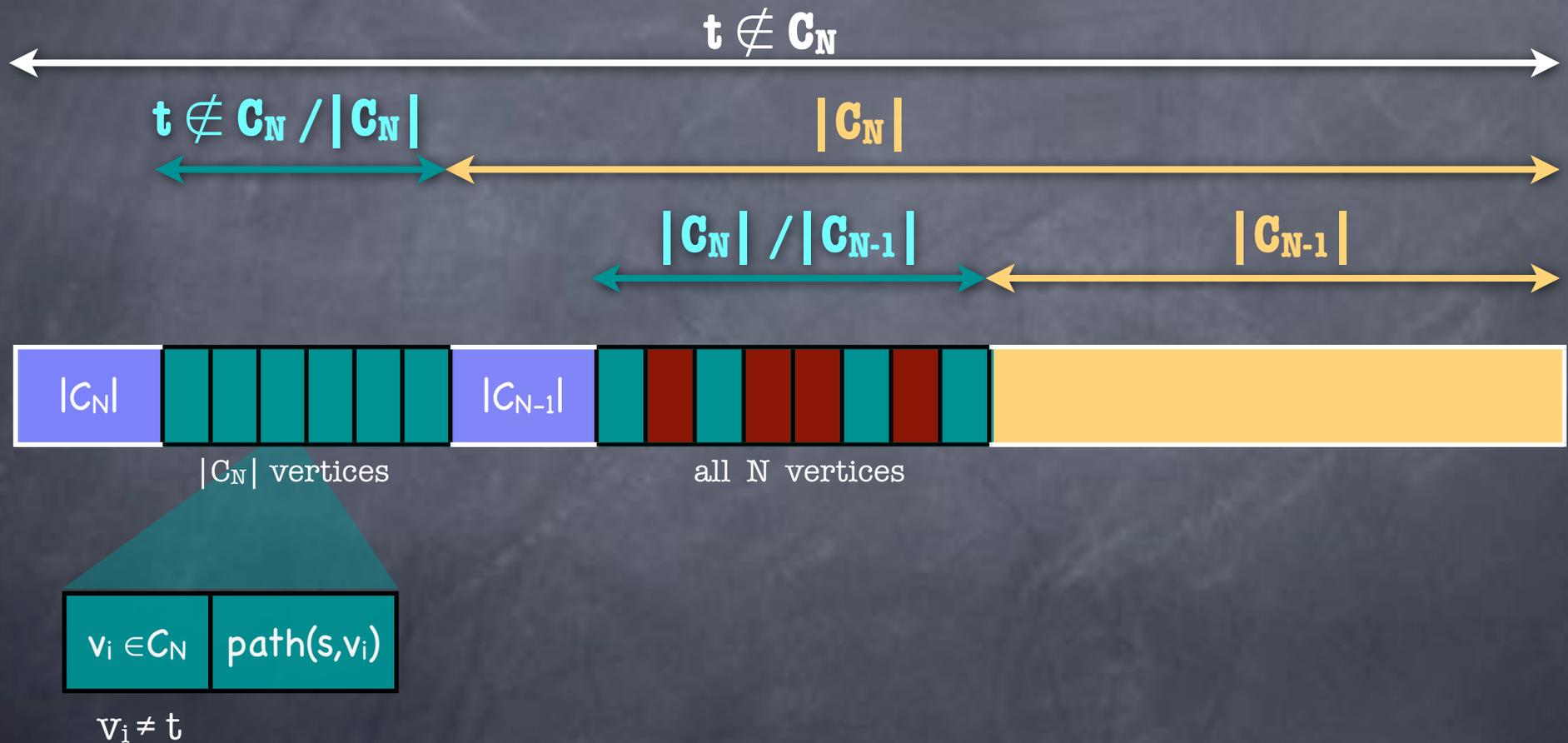


$v_i \neq t$

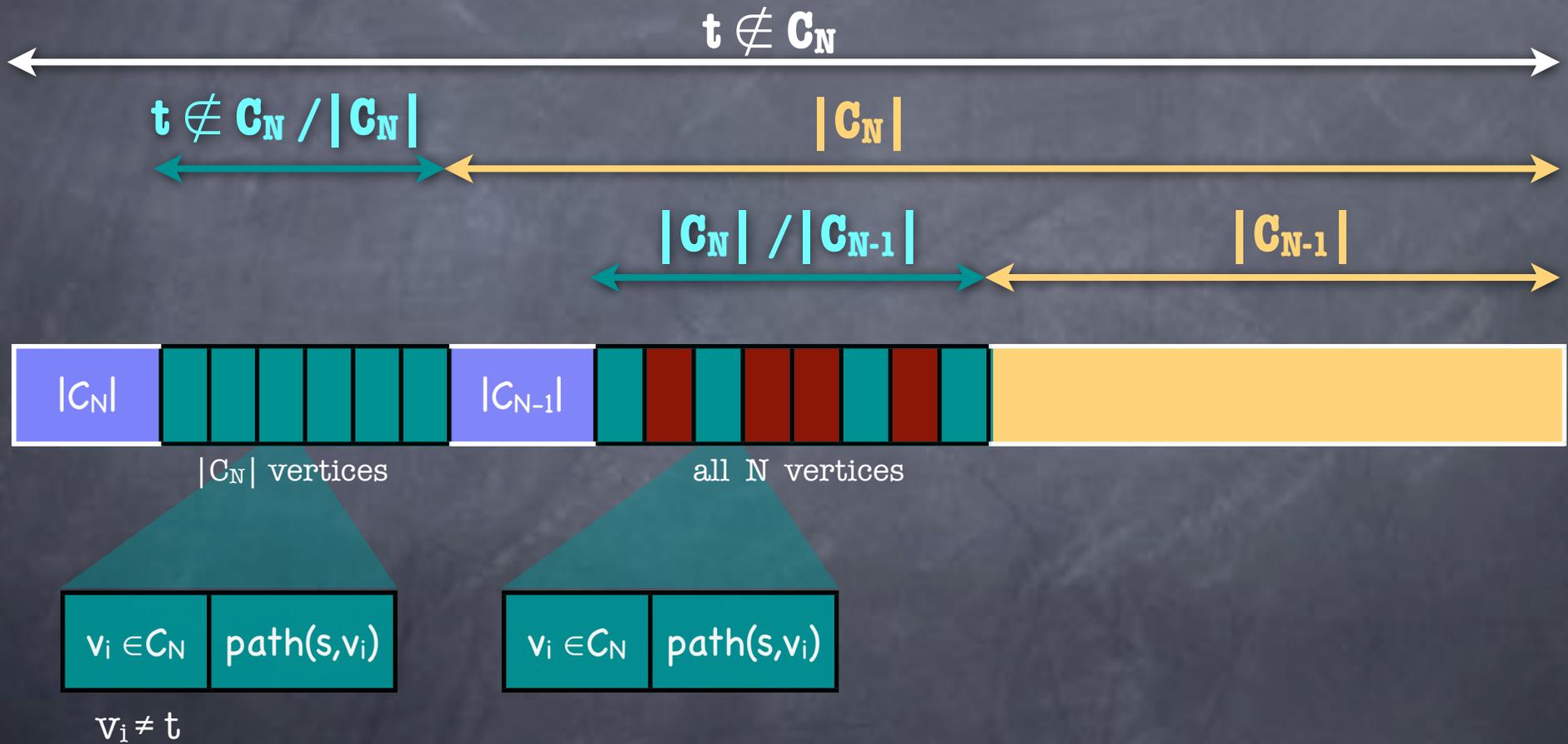
Certificate for $t \notin C_N$



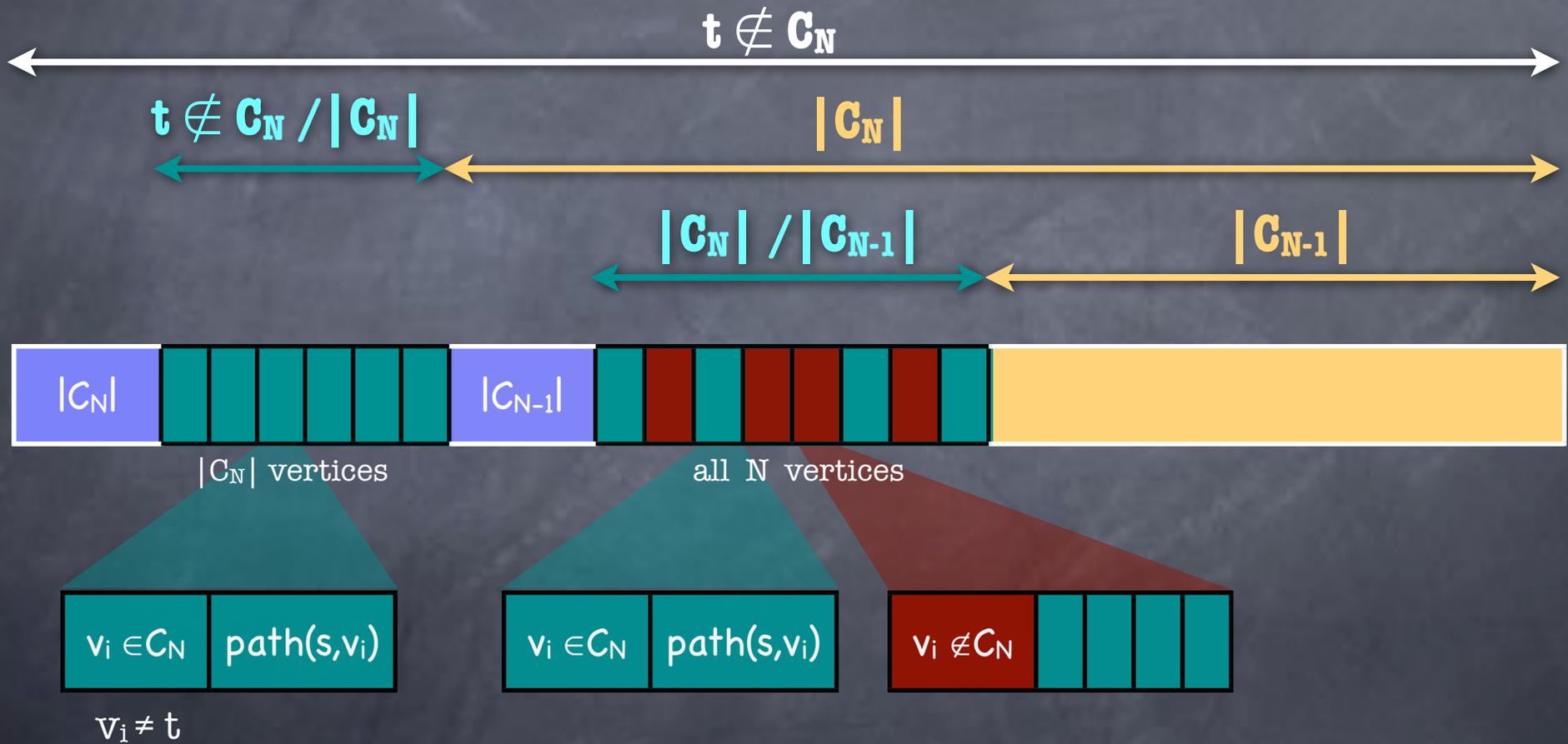
Certificate for $t \notin C_N$



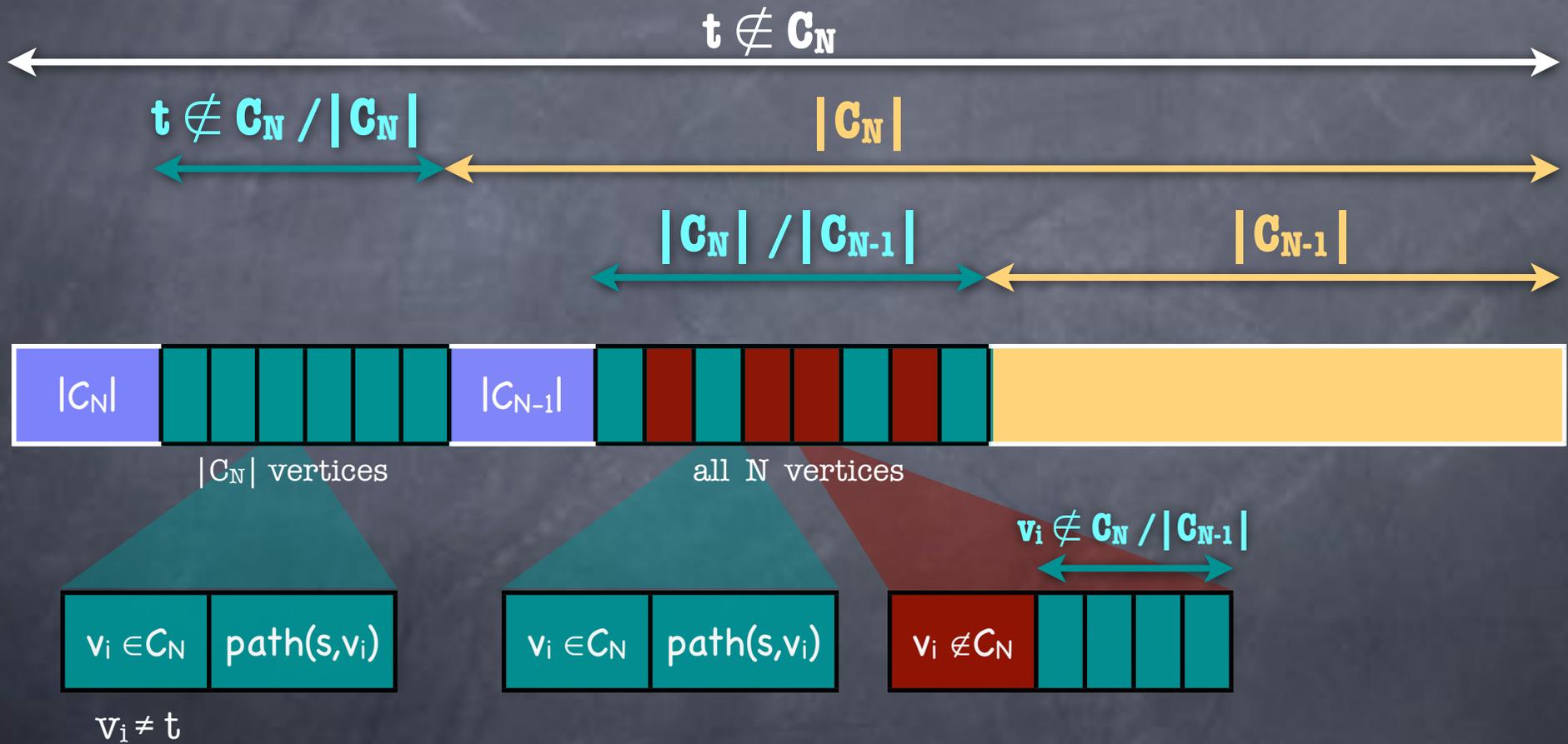
Certificate for $t \notin C_N$



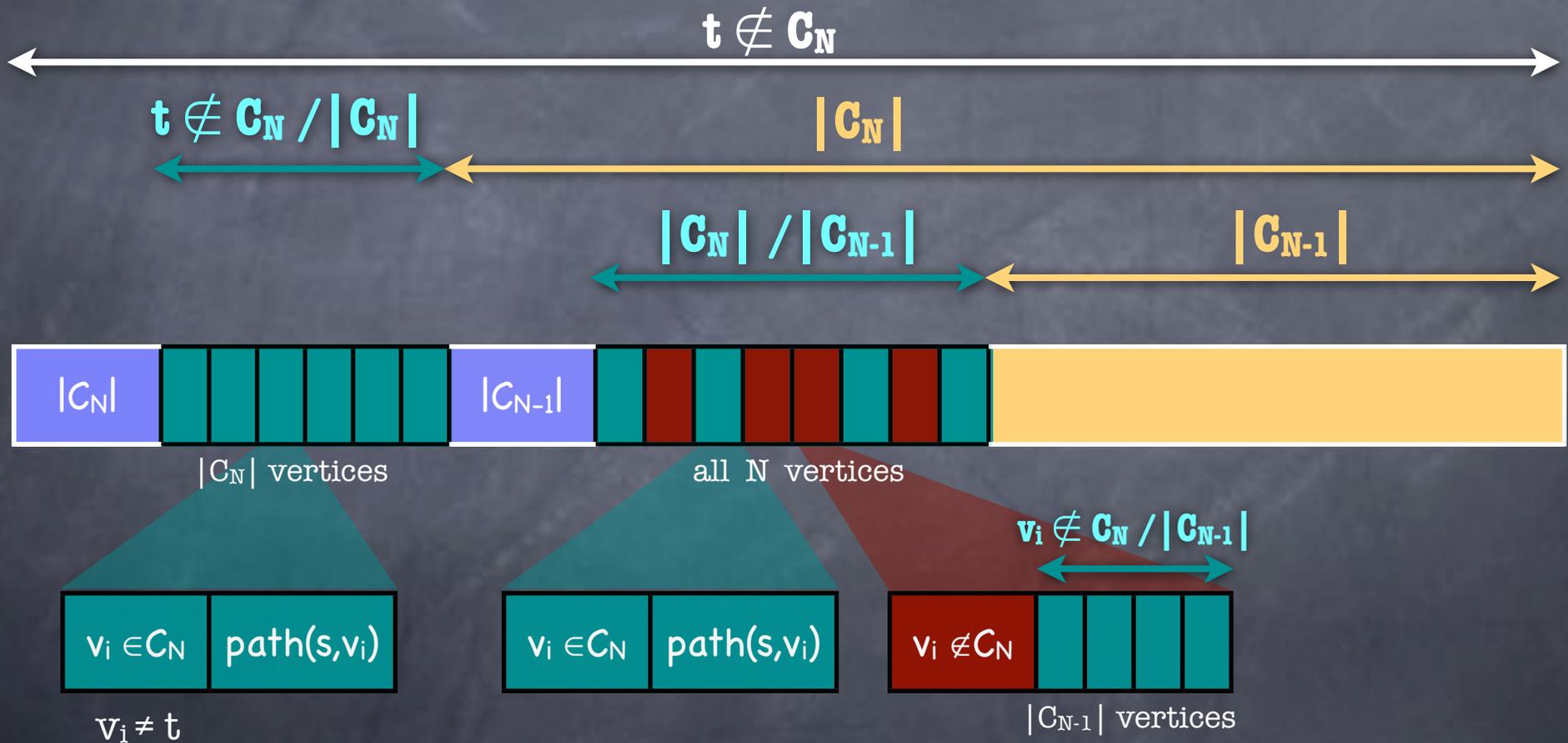
Certificate for $t \notin C_N$



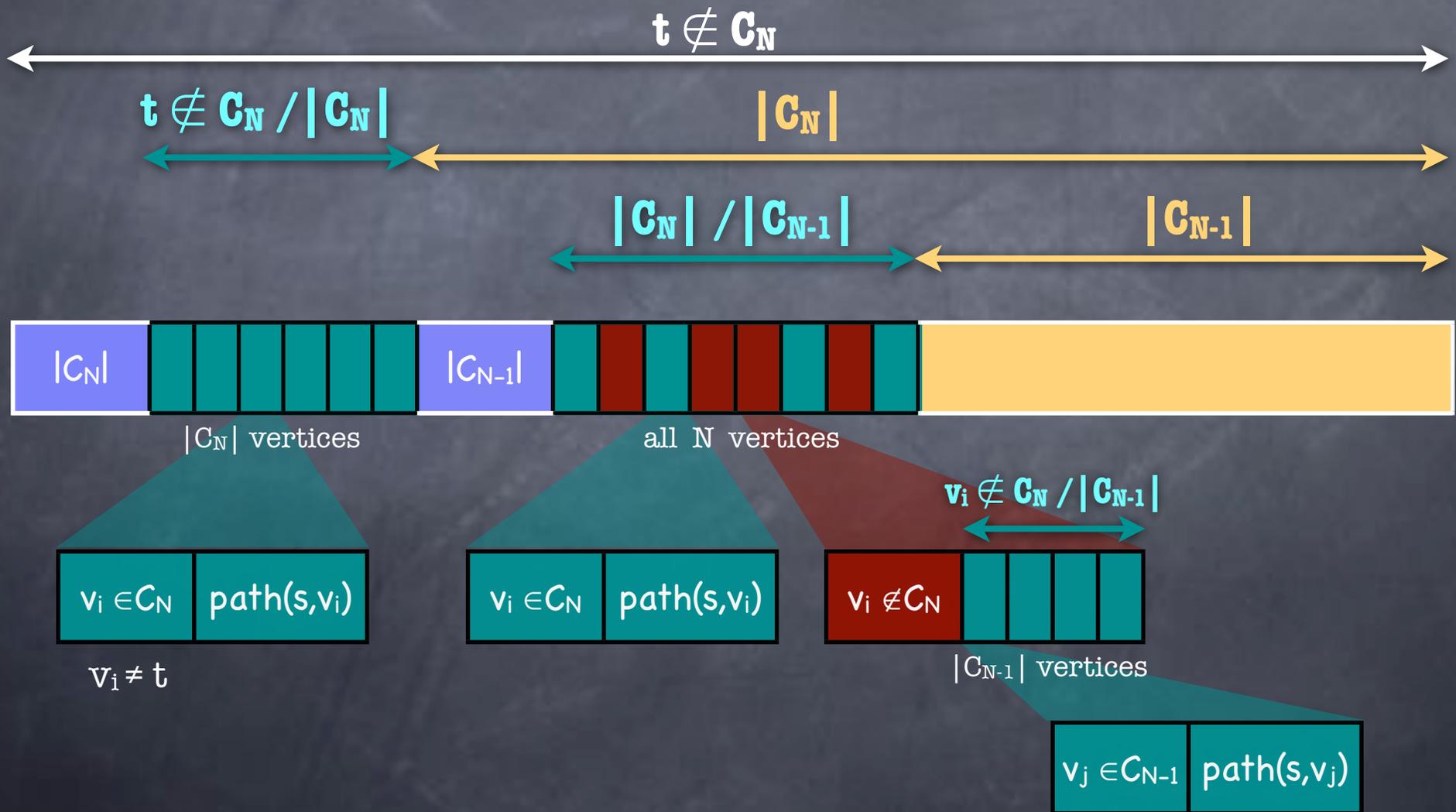
Certificate for $t \notin C_N$



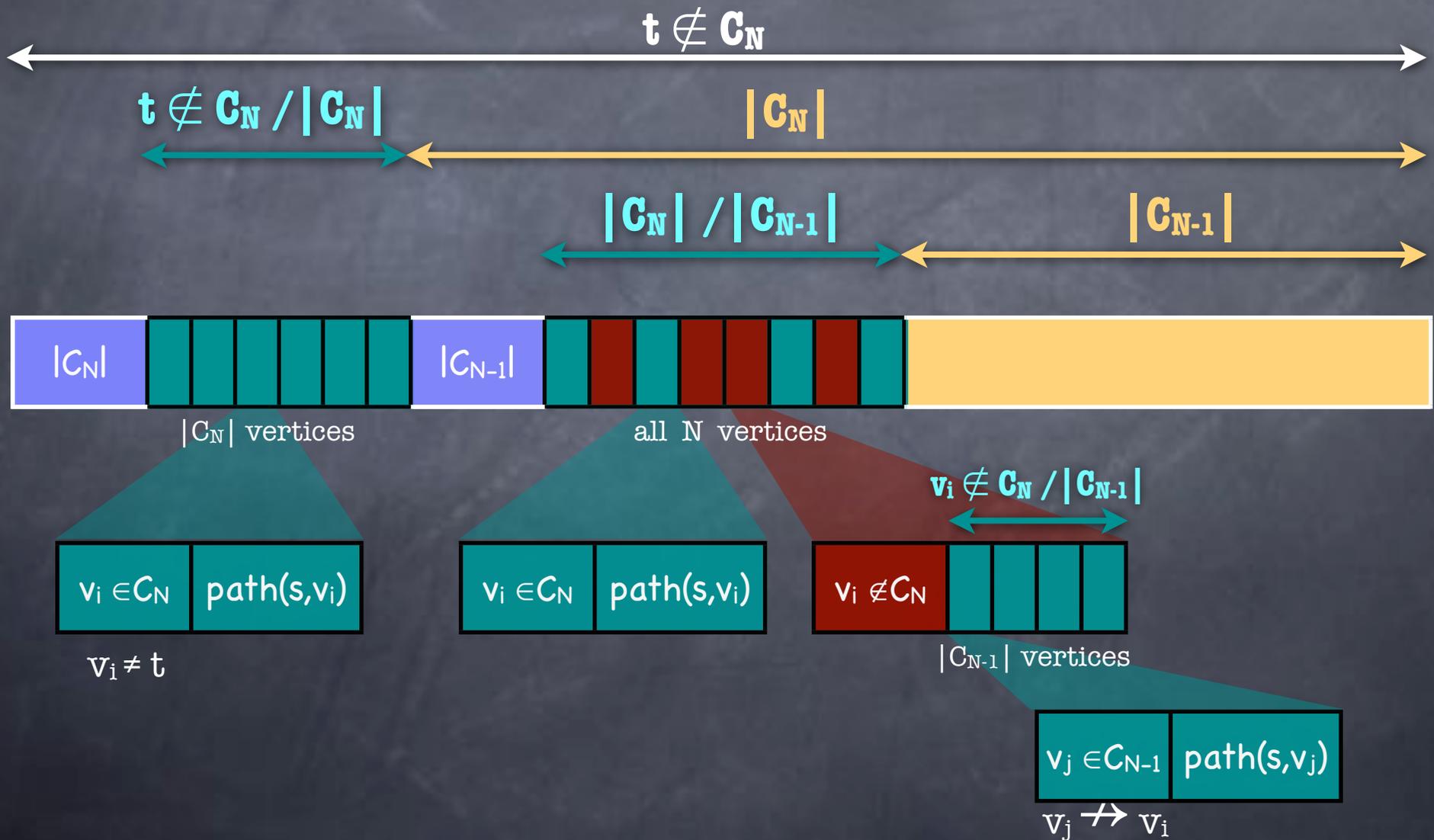
Certificate for $t \notin C_N$



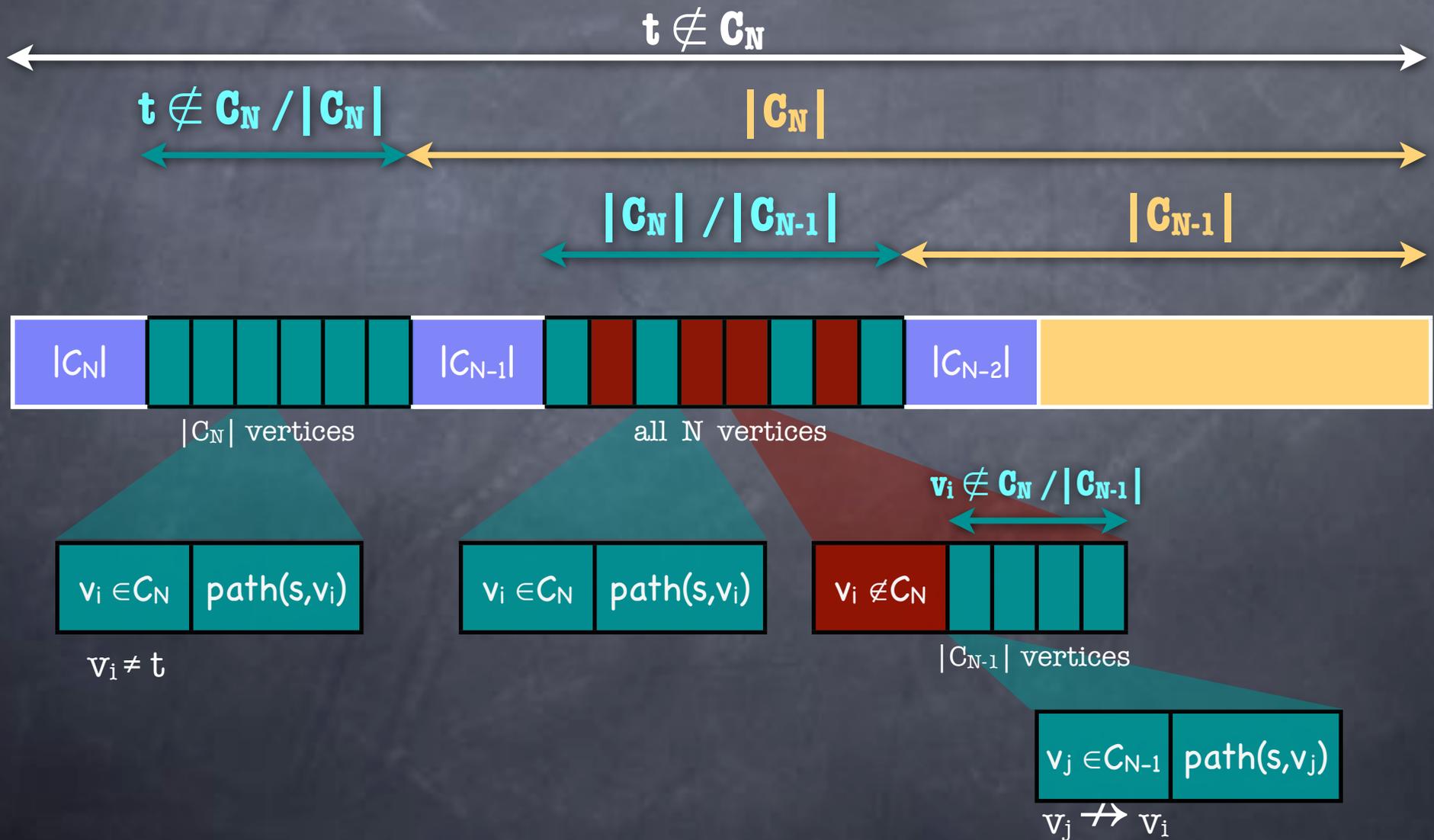
Certificate for $t \notin C_N$



Certificate for $t \notin C_N$



Certificate for $t \notin C_N$



Certificate for $t \notin C_N$

