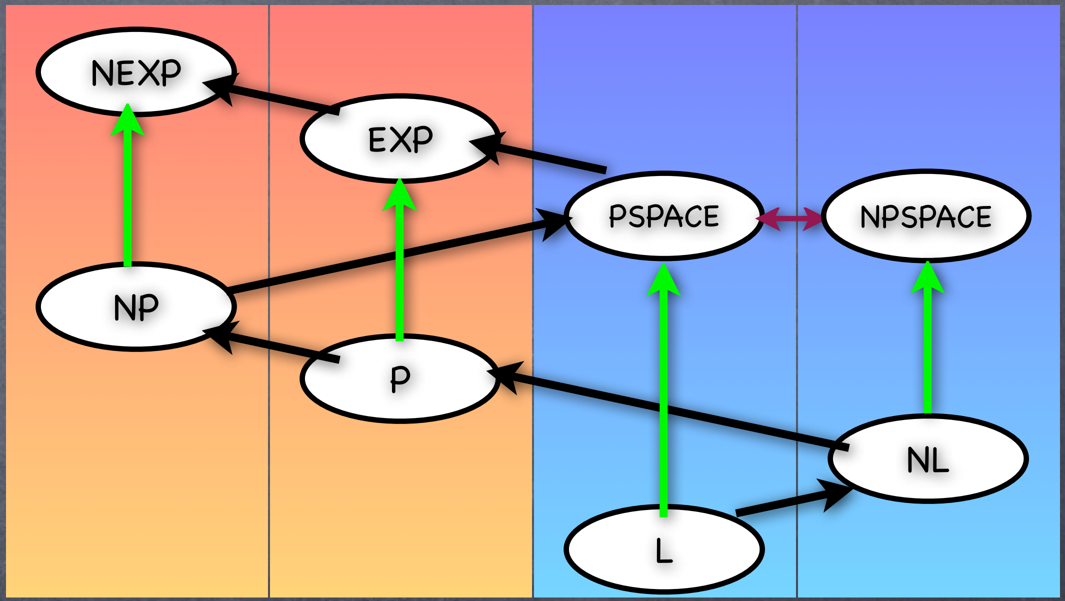
Computational Complexity

Lecture 6

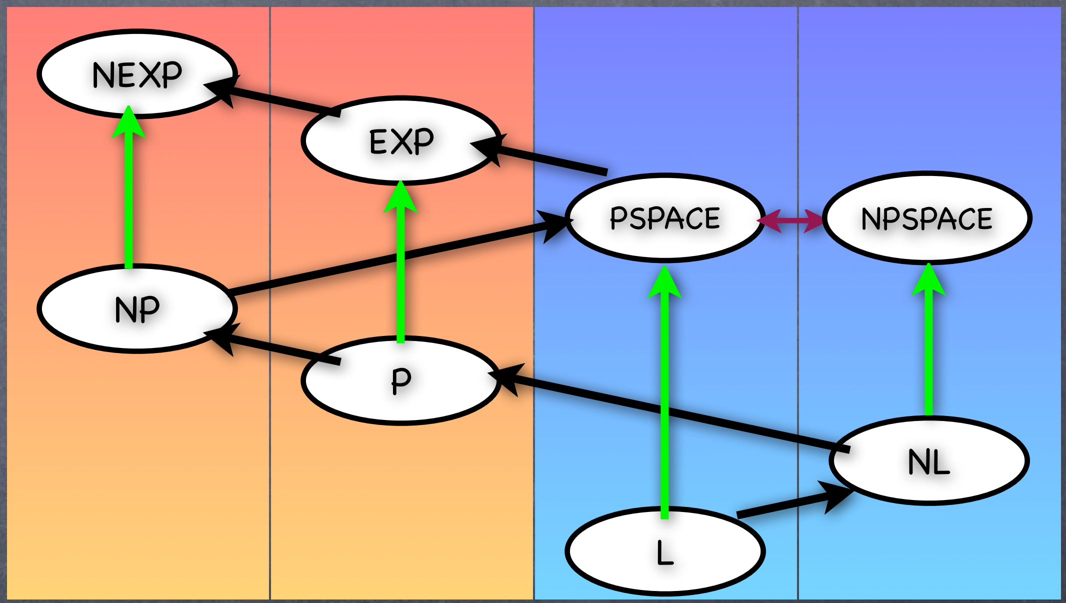
NL-Completeness and NL=co-NL

Story, so far



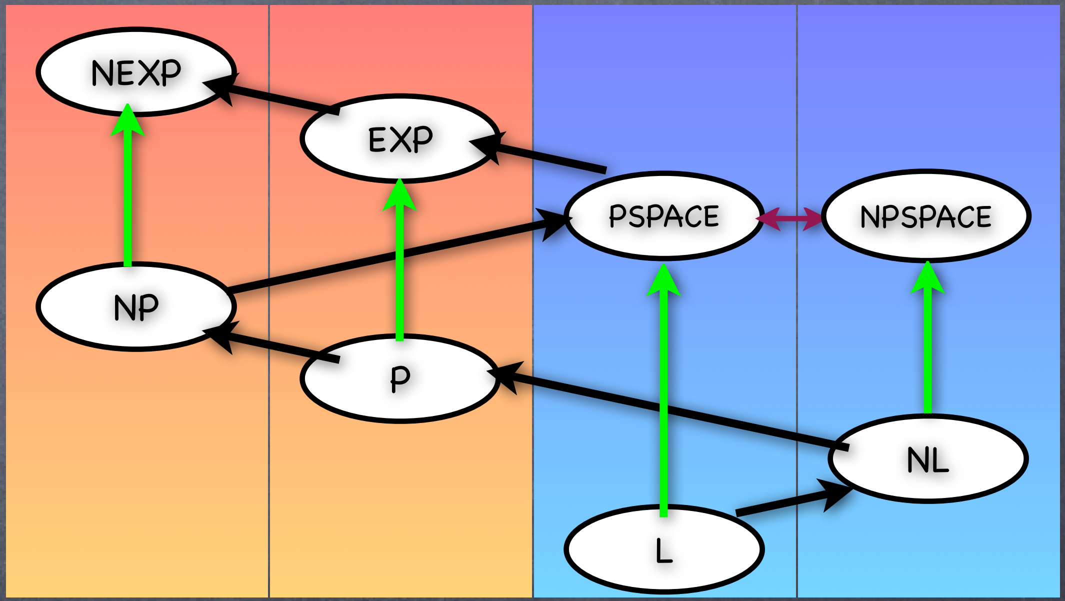
Story, so far

- Time/Space Hierarchies



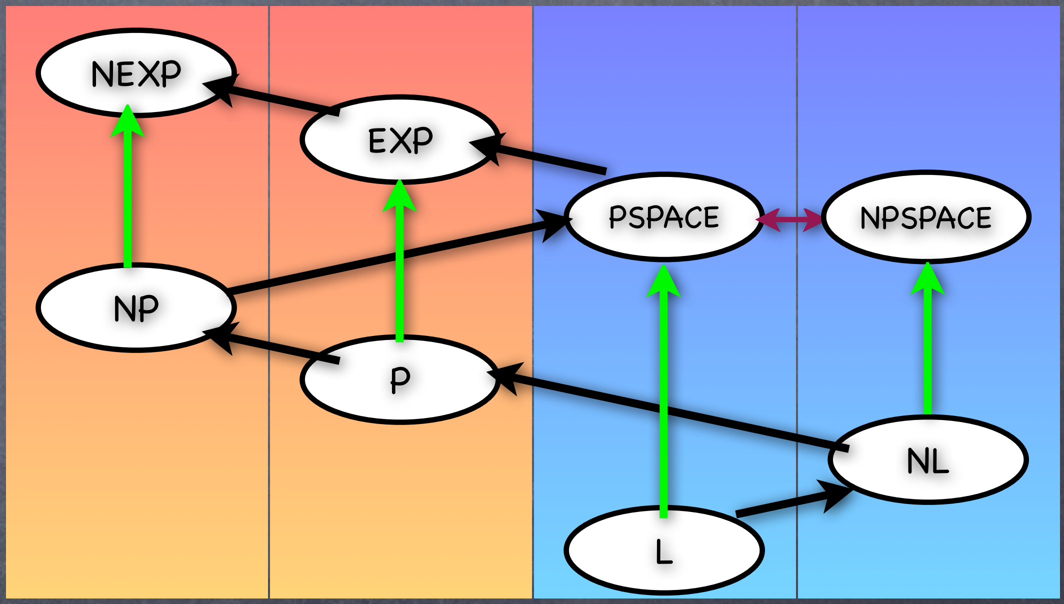
Story, so far

- Time/Space Hierarchies
- Relations across complexity measures



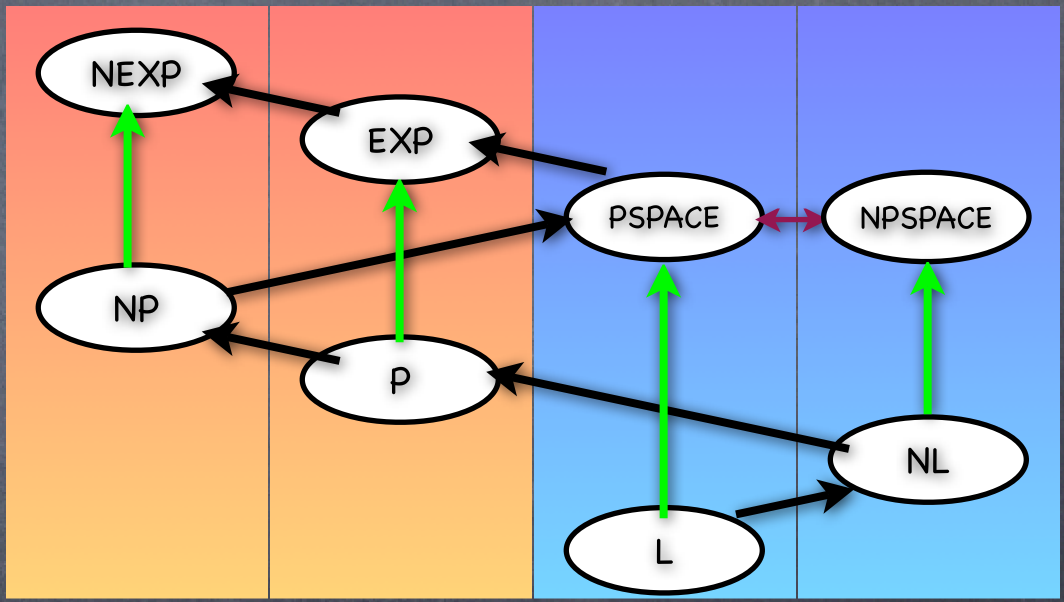
Story, so far

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



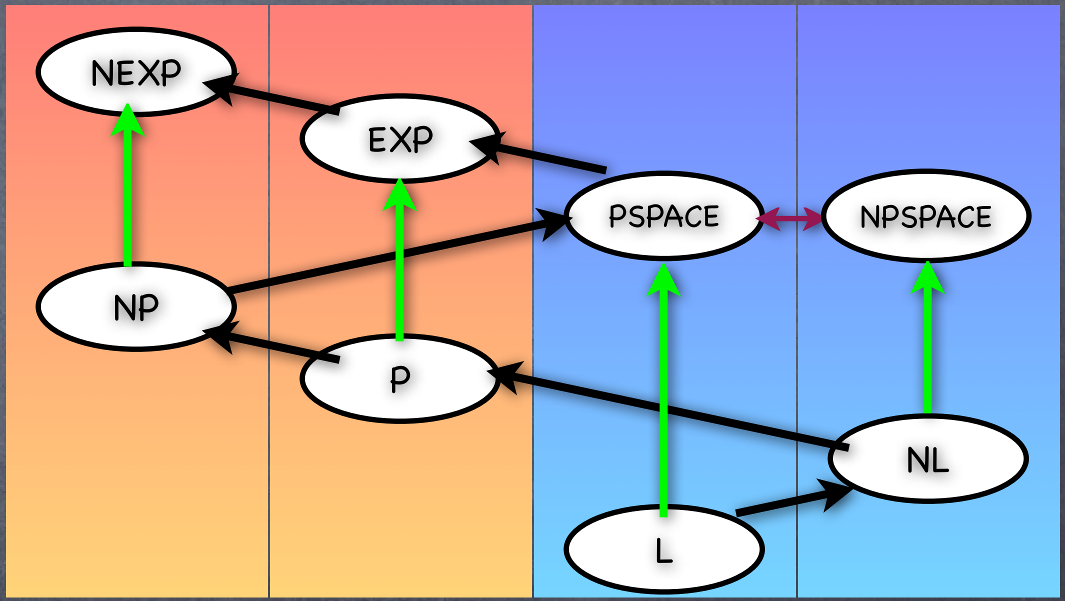
Story, so far

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



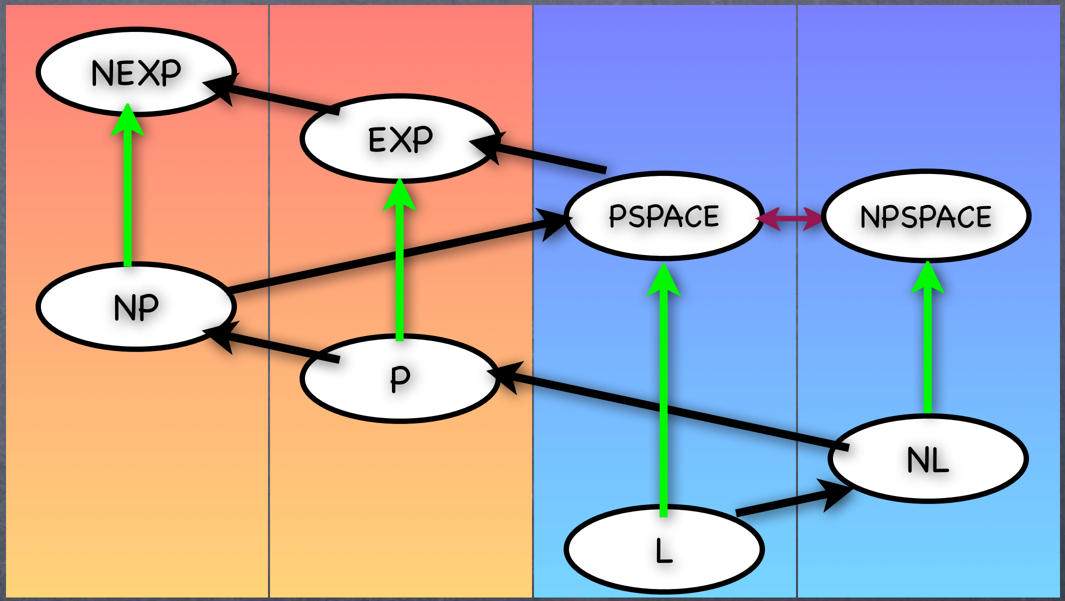
Story, so far

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



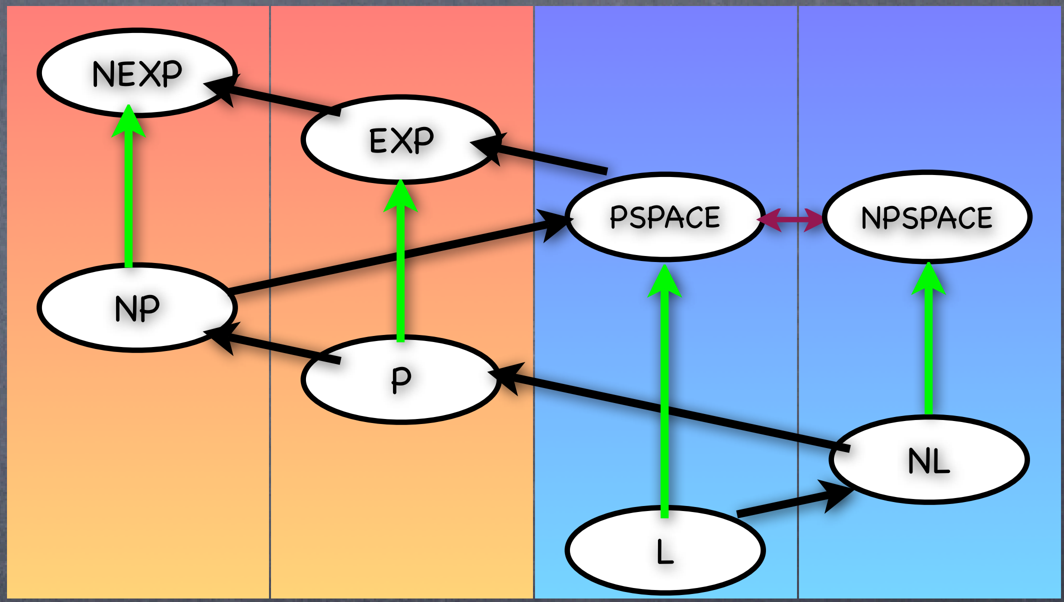
Story, so far

- 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 P, $L_2 \leq_p L_1$

NL-completeness

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

NL-completeness

- For any two (non-trivial) languages L_1, L_2 in P, $L_2 \leq_p L_1$
  - So if $X \subseteq P$, all 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 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 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 $\mathrm{poly}(|x|)$
  - 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 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: to 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 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: to 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

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 $M_{2\text{-}1}$ and $M_{1\text{-}0}$ build $M_{2\text{-}0}$:

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 $M_{2\text{-}1}$ and $M_{1\text{-}0}$ build $M_{2\text{-}0}$:
  - Start running $M_{1\text{-}0}$ without input. When it wants to read i^{th} bit of input, run $M_{2\text{-}1}$ (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 $M_{2\text{-}1}$ and $M_{1\text{-}0}$ build $M_{2\text{-}0}$:
  - Start running $M_{1\text{-}0}$ without input. When it wants to read i^{th} bit of input, run $M_{2\text{-}1}$ (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 $\mathrm{poly}(|x|)$

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 $M_{2\text{-}1}$ and $M_{1\text{-}0}$ build $M_{2\text{-}0}$:
  - Start running $M_{1\text{-}0}$ without input. When it wants to read i^{th} bit of input, run $M_{2\text{-}1}$ (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 $M_{2\text{-}1}$ and $M_{1\text{-}0}$ build $M_{2\text{-}0}$:
  - Start running $M_{1\text{-}0}$ without input. When it wants to read i^{th} bit of input, run $M_{2\text{-}1}$ (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

- L_0 is NL-Hard if for all L_1 in NL, $L_1 \leq_L L_0$

NL-completeness

- 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

- 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

- 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 accepts $(x;w)$ in space $\log(s) \}$ (where M takes w in a read-once tape)

NL-completeness

- 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 accepts $(x;w)$ in space $\log(s) \}$ (where M takes w in a read-once tape)
- Interesting NLC language: PATH

Directed Path

Directed Path

- PATH = {(G,s,t) | G a directed graph with a path from s to t}

Directed Path

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

Directed Path

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

Directed Path

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

Directed Path

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

Directed Path

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

- PATH = {(G,s,t) | G a directed graph with a path from s to t}
  - G using some representation, of size say, n^2 (n=#vertices)
    - Such that, if two vertices x,y on work-tape, can check for edge (x,y)
- PATH in NL
  - Certificate w is the path (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 $\mathrm{NSPACE}(S(n)) \subseteq \mathrm{DTIME}(2^{O(S(n))})$ (e.g. $\mathrm{NL} \subseteq \mathrm{P}$)

Seen PATH before?

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

Seen PATH before?

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

Seen PATH before?

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

Seen PATH before?

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

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 $\in$ NSPACE(S) to PATH

If PATH $\in$ co-NL

If PATH $\in$ co-NL

- If PATH $\in$ co-NL, then co-NL $\subseteq$ NL

If PATH $\in$ co-NL

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

If PATH $\in$ co-NL

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

If PATH $\in$ co-NL

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

If PATH $\in$ co-NL

If PATH $\in$ co-NL

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

If PATH $\in$ co-NL

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

If PATH $\in$ co-NL

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

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

If PATH $\in$ co-NL

If PATH $\in$ co-NL

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

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

If PATH $\in$ co-NL

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

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

$PATH^c \in NL$

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

PATH$^c \in$ NL

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

PATH$^c \in$ 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 $c = |C|$

$PATH^c \in NL$

- 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 count.

$PATH^c \in NL$

- 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 count.
      - List has $|C|$ many $v \in C$, without repeating

Certificate for |C|

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 are poly(N) long)

Certificate for |C|

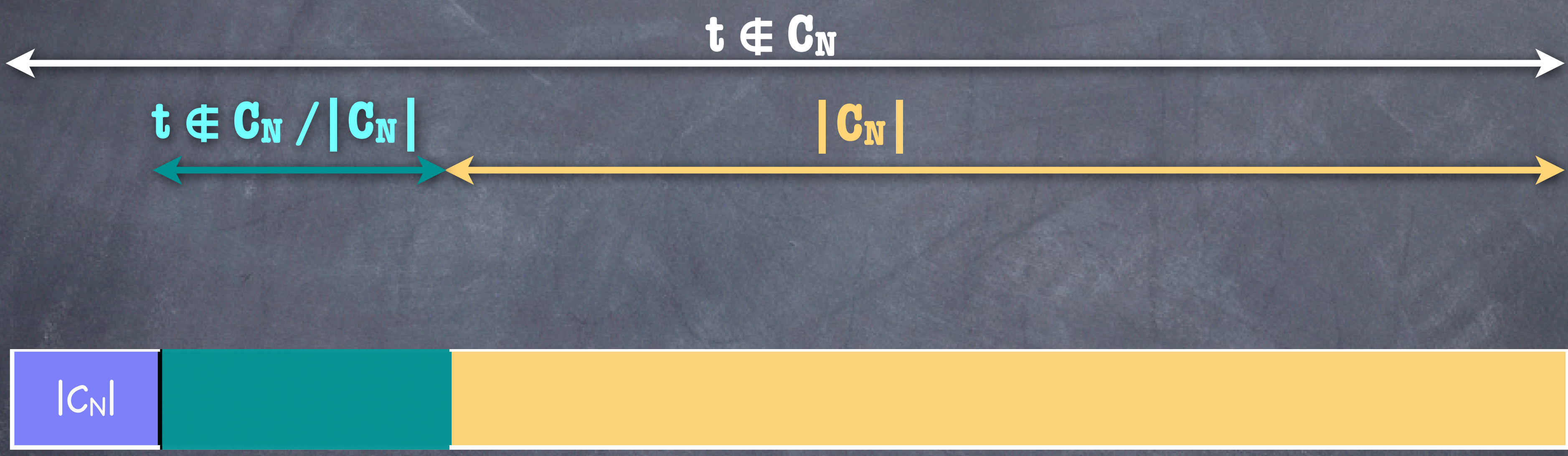
- Let C_i := set of nodes within distance i of s. Then $C = C_N$
- Tail recursion to verify $|C_N|$:
  - Read $|C_{N-1}|$, believing it verify $|C_N|$, forget $|C_N|$;
  - Read $|C_{N-2}|$, believing it verify $|C_{N-1}|$, forget $|C_{N-1}|$; ...
  - Base case: $|C_0|=1$
- Believing $|C_{i-1}|$ verify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

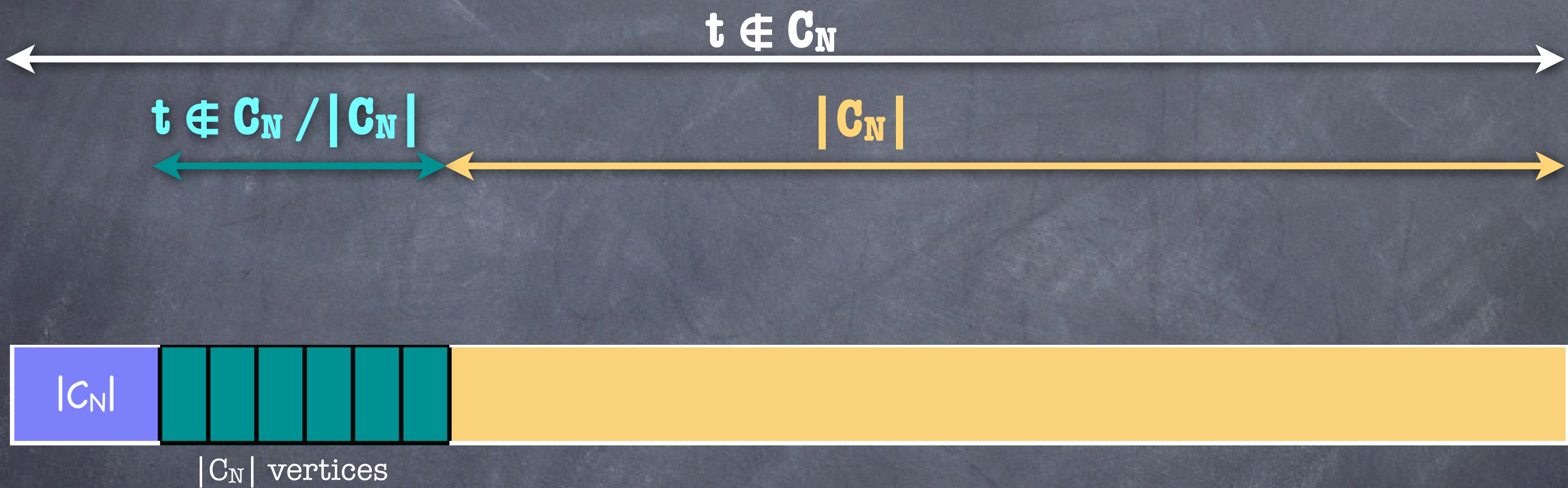
Certificate for $t \notin C_N$

Certificate for $t \notin C_N$

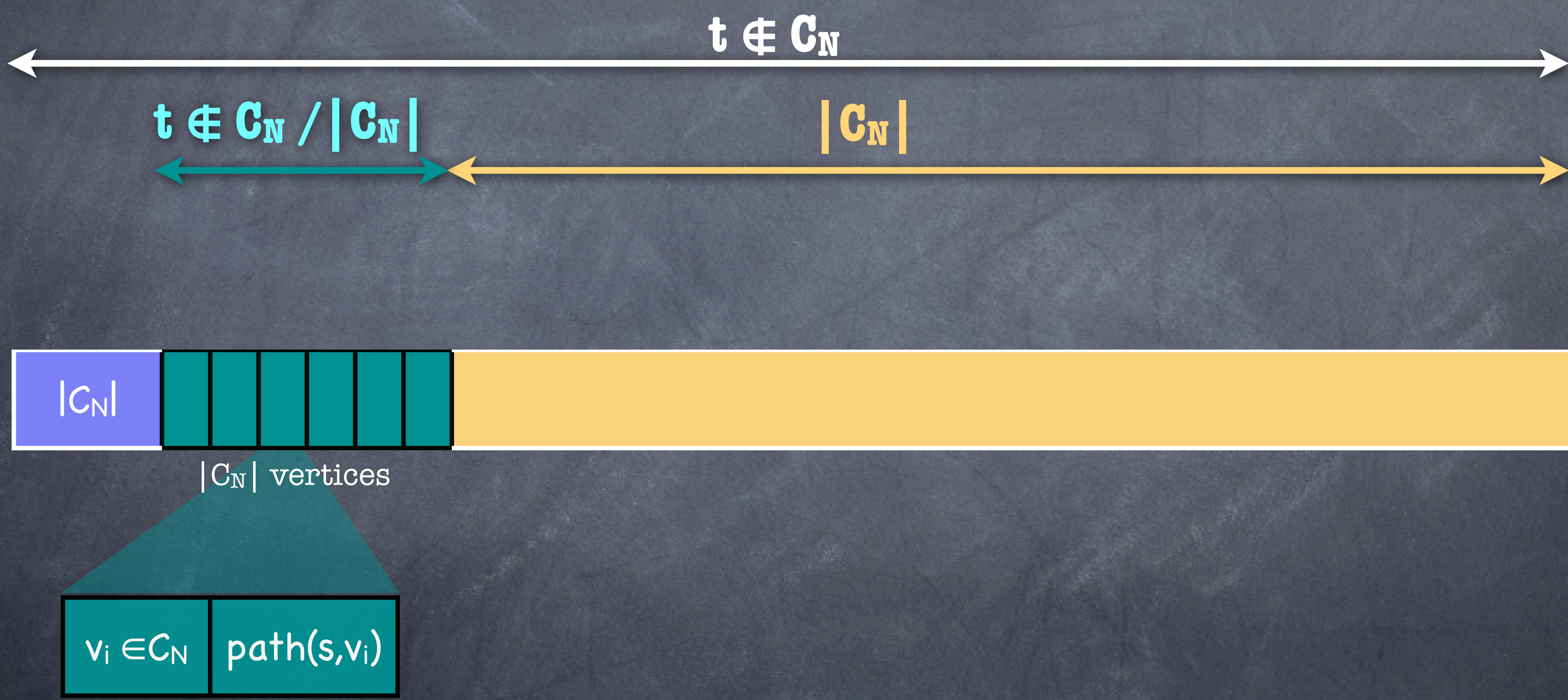
Certificate for $t \notin C_N$



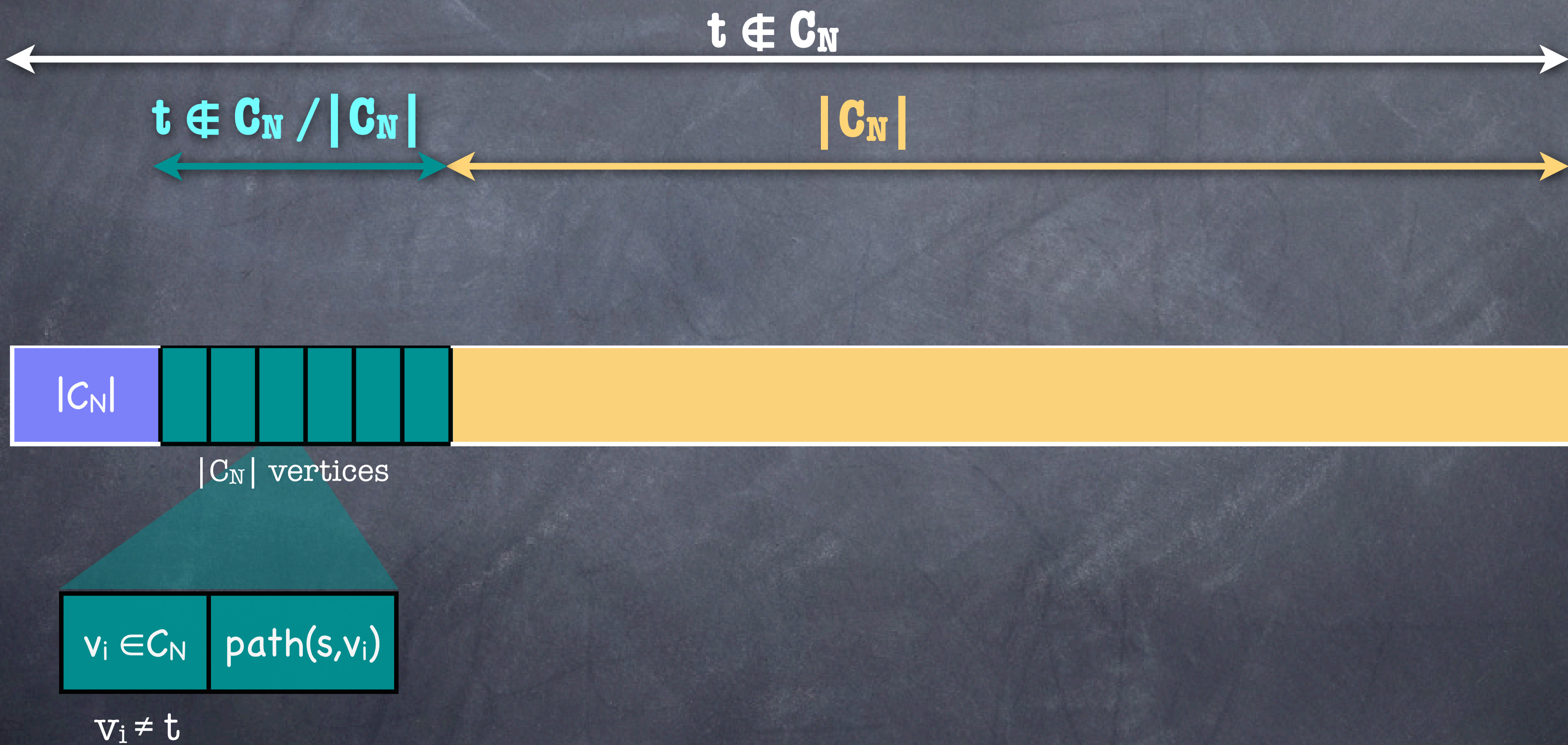
Certificate for $t \notin C_N$



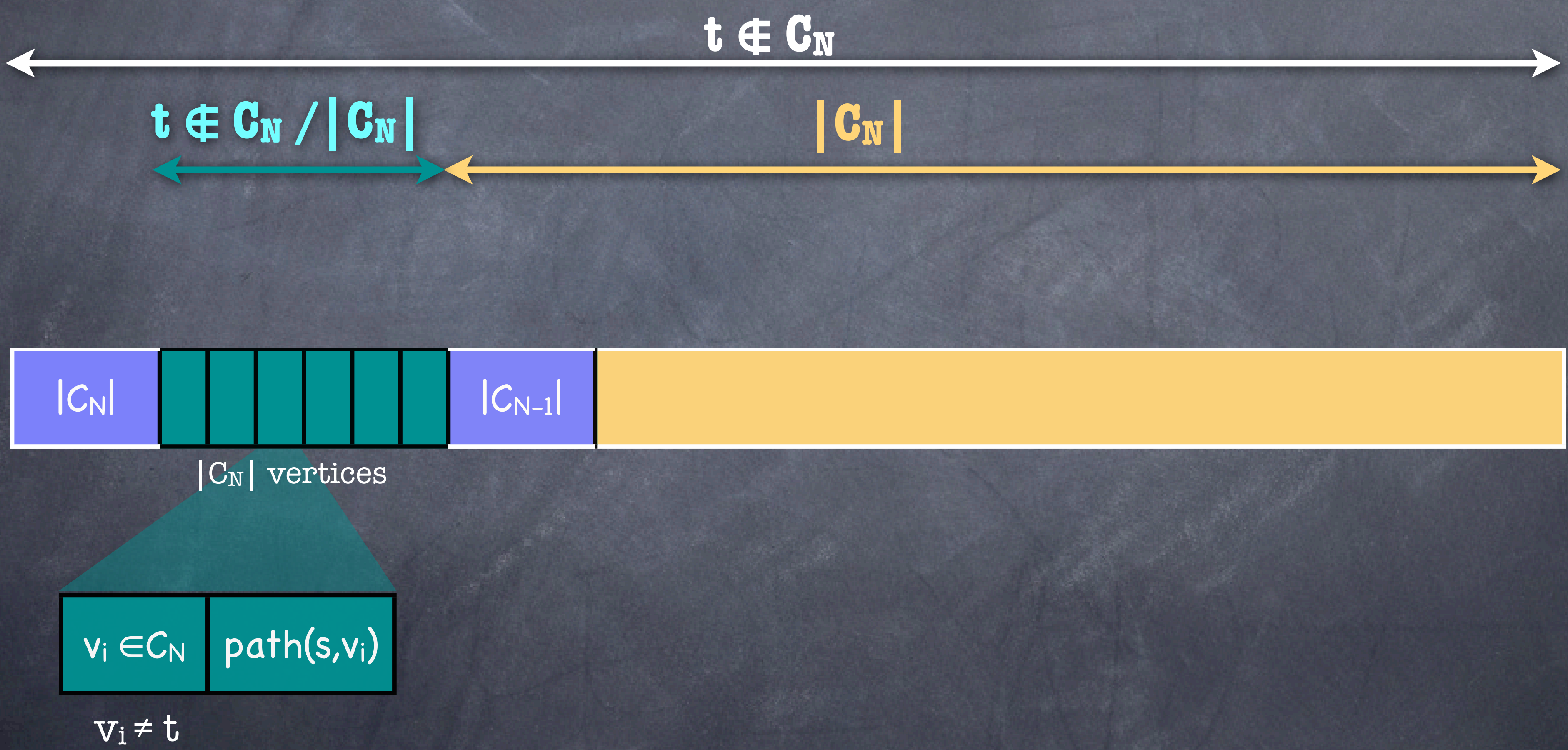
Certificate for $t \notin C_N$



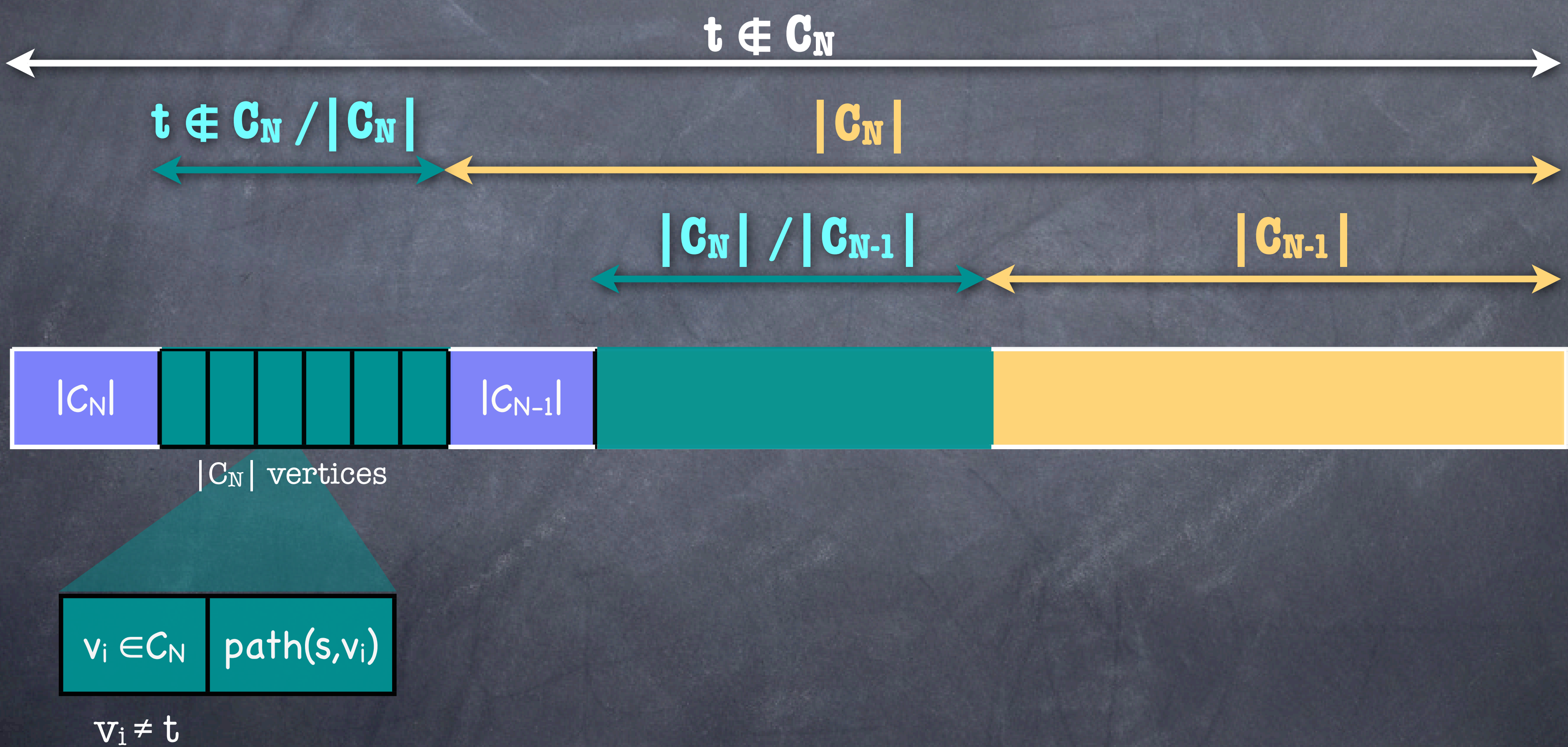
Certificate for $t \notin C_N$



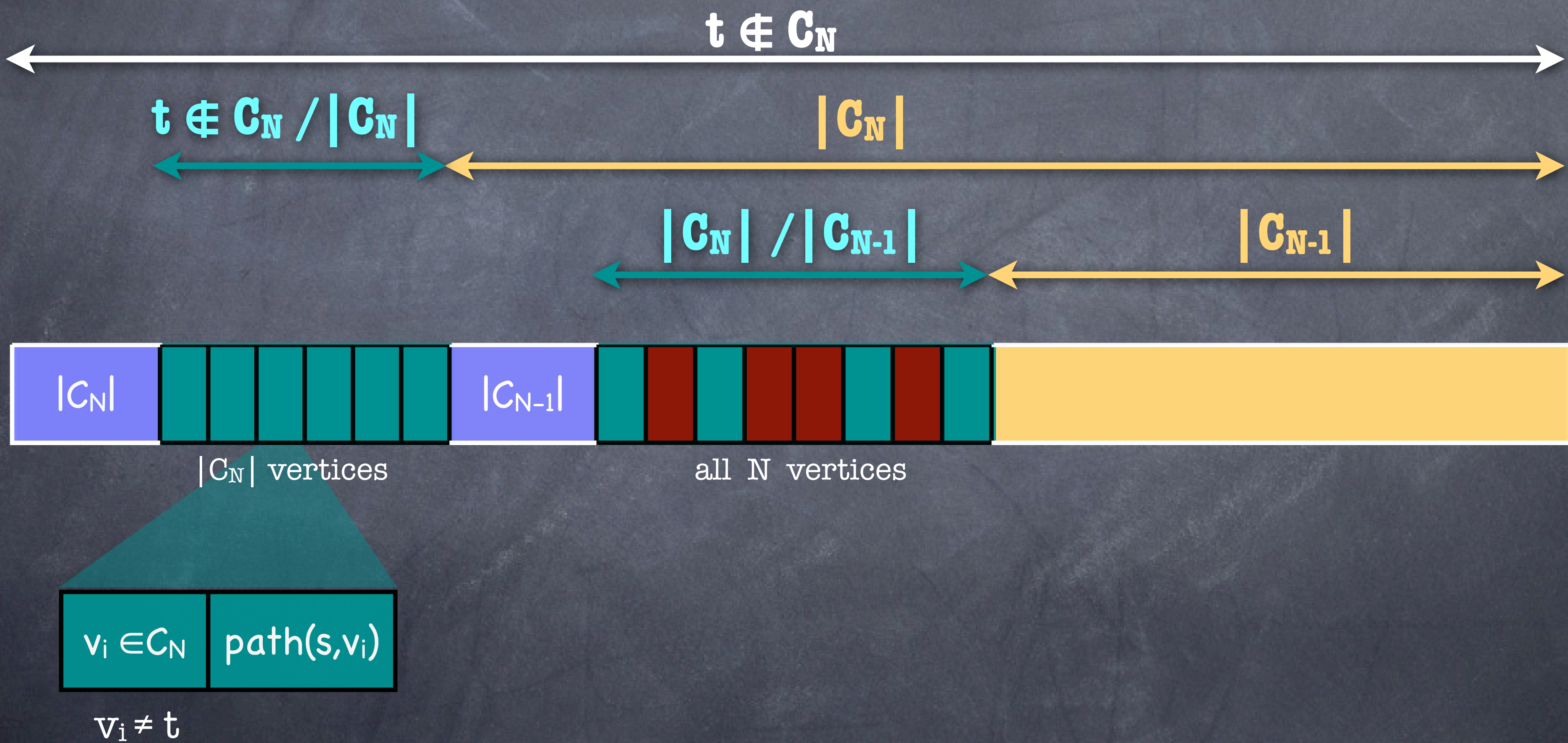
Certificate for $t \notin C_N$



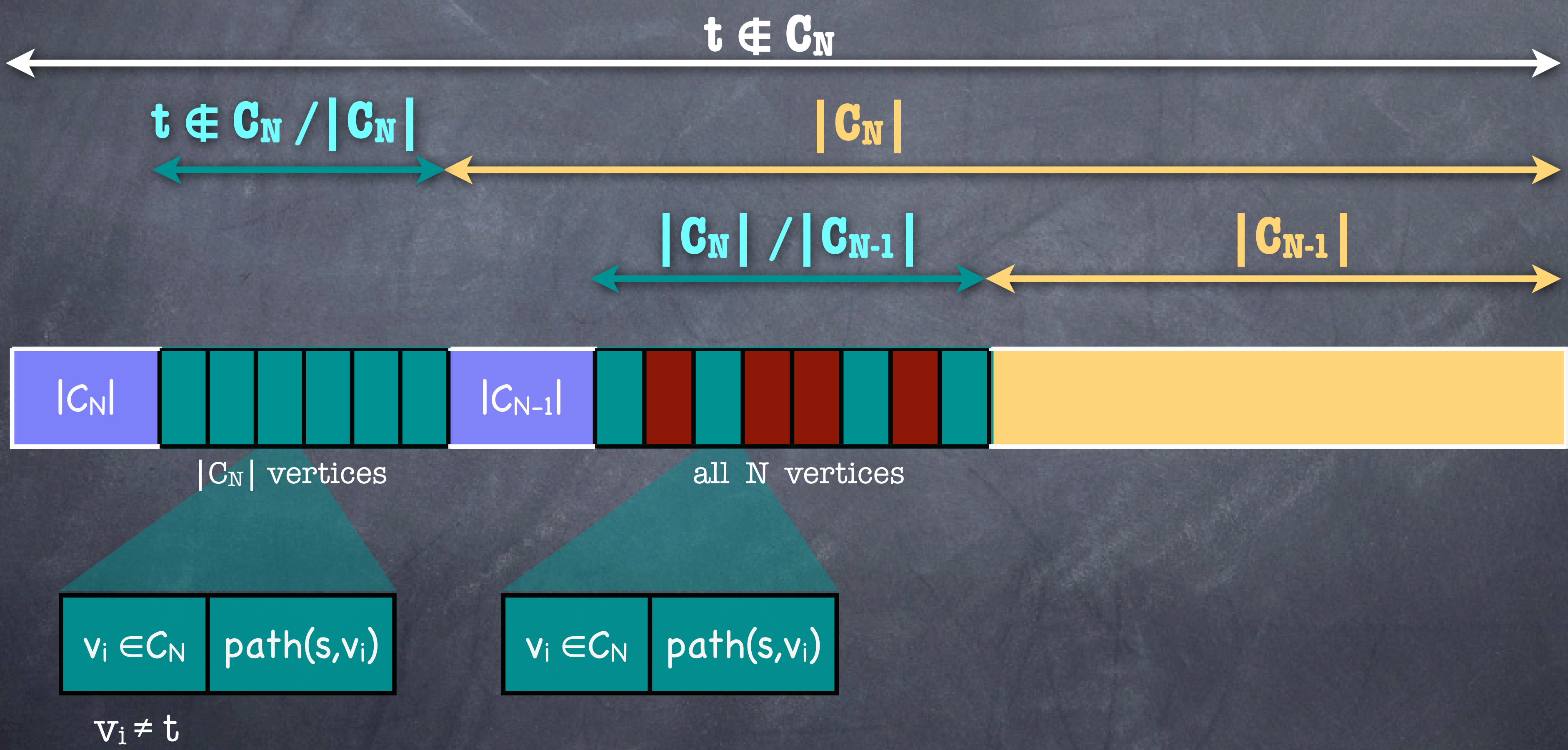
Certificate for $t \notin C_N$



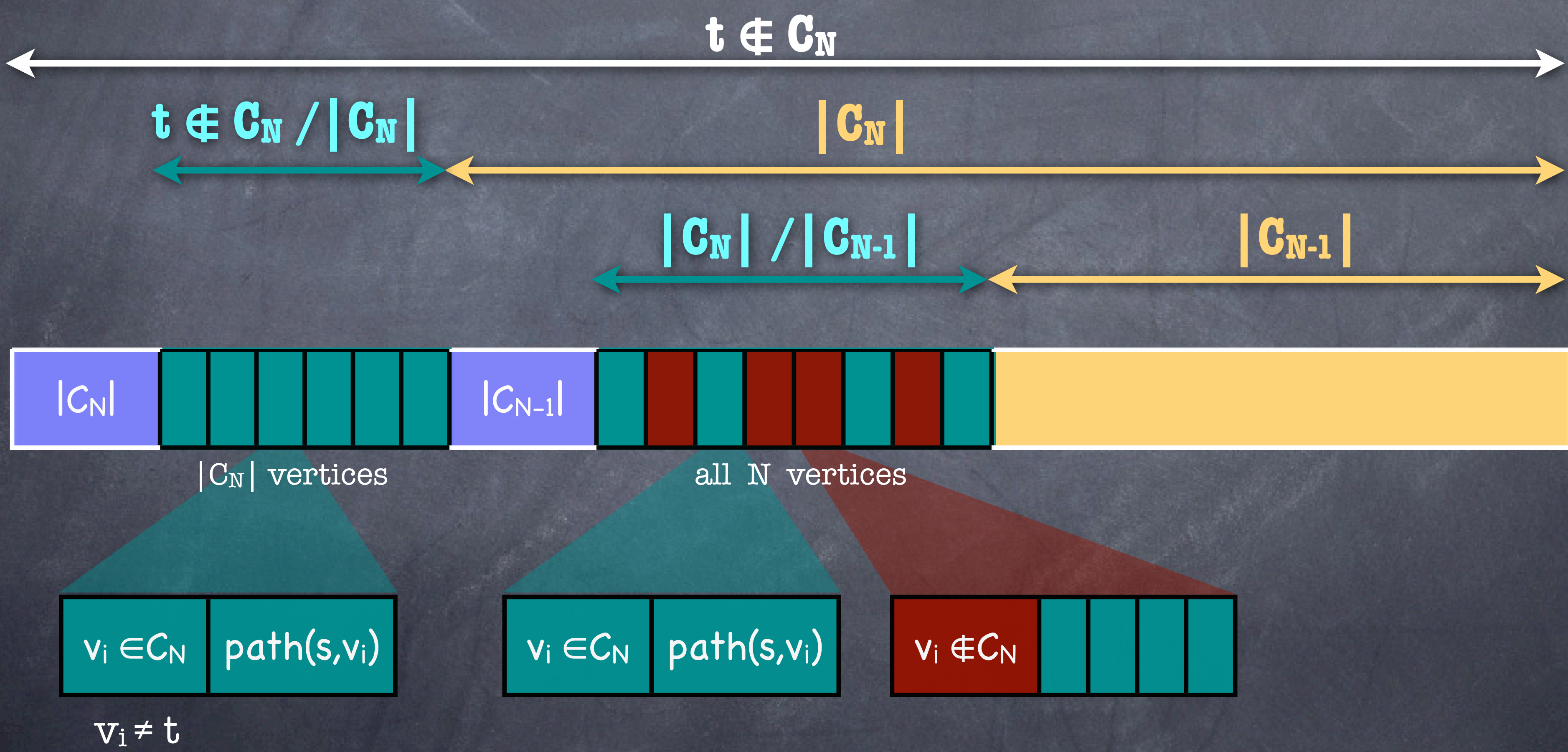
Certificate for $t \notin C_N$



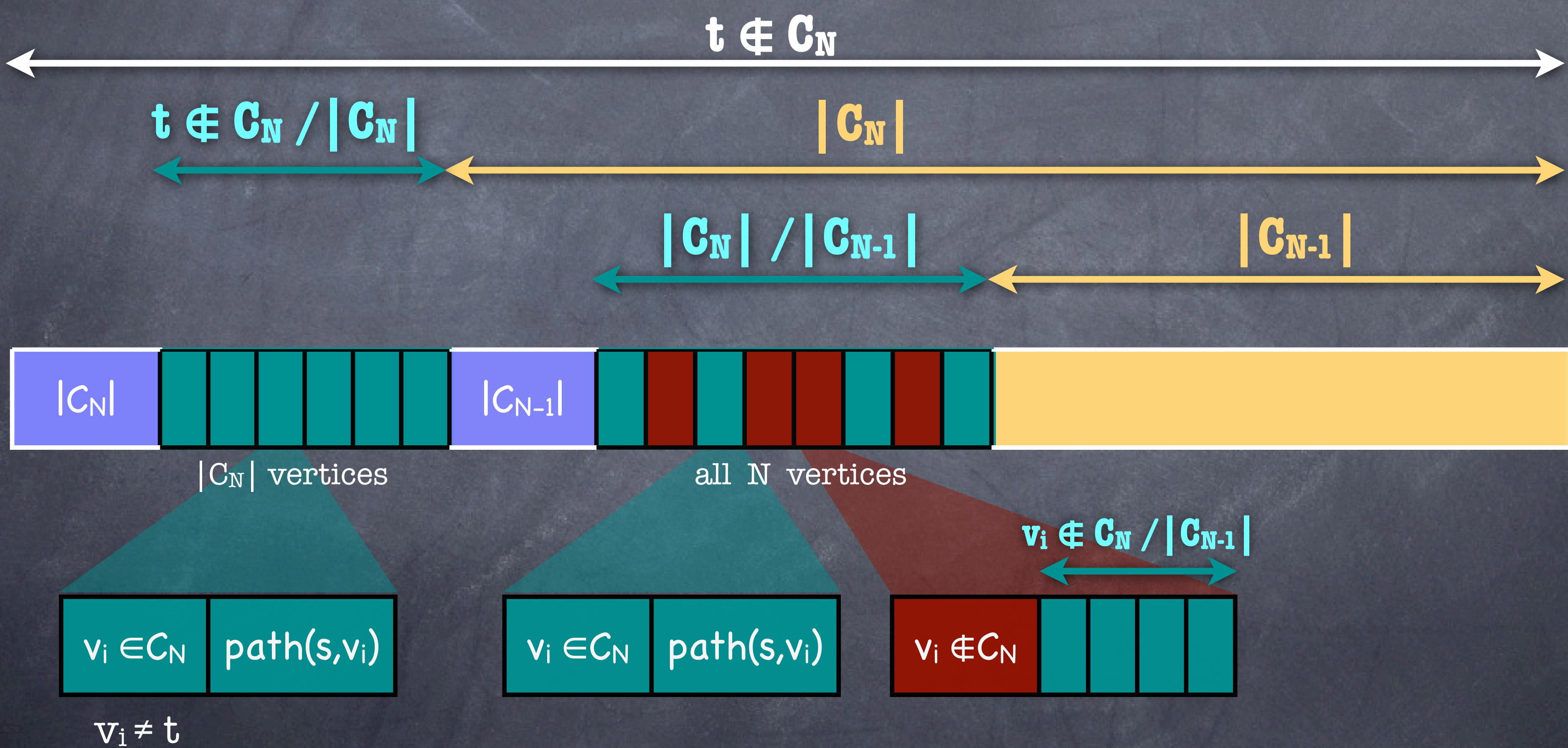
Certificate for $t \notin C_N$



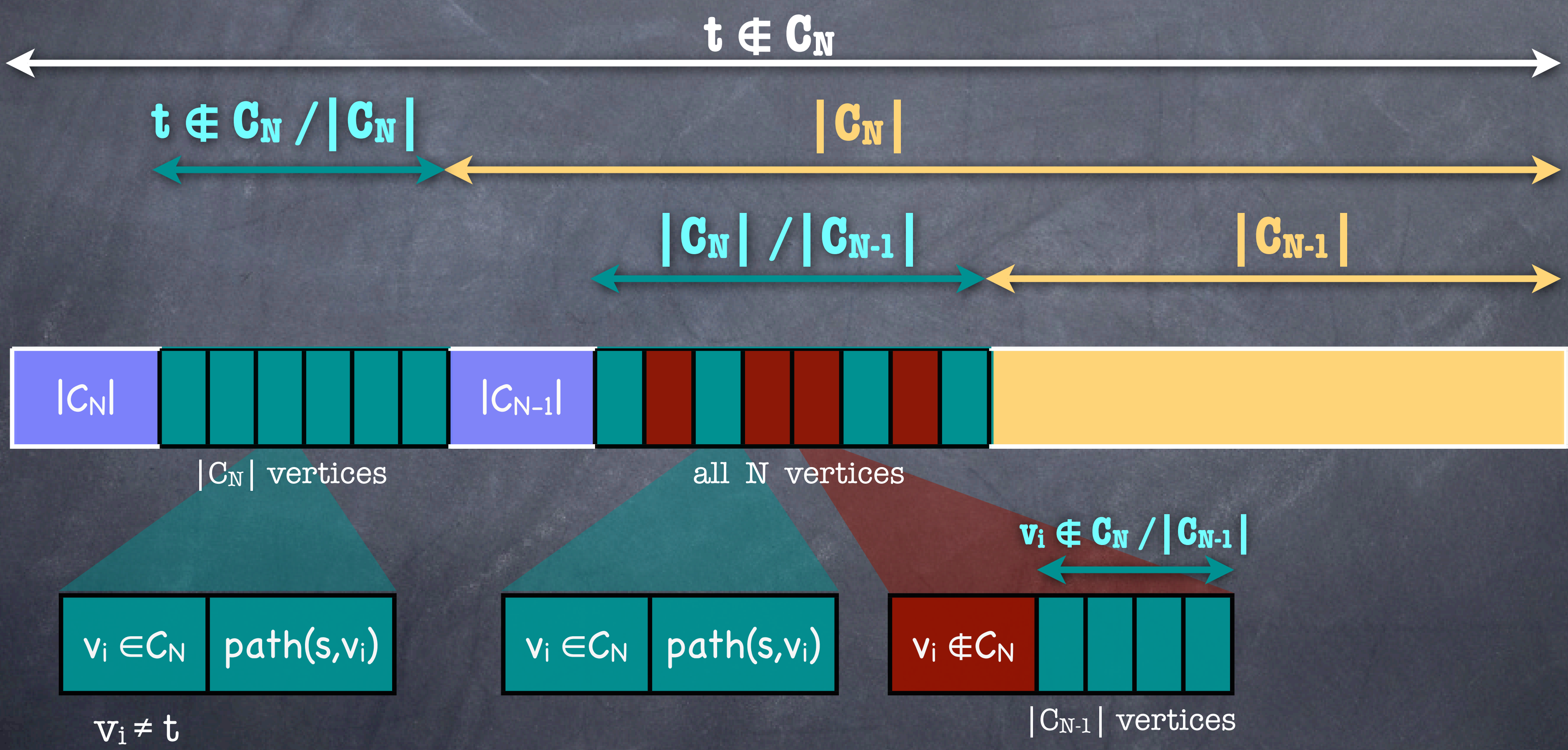
Certificate for $t \notin C_N$



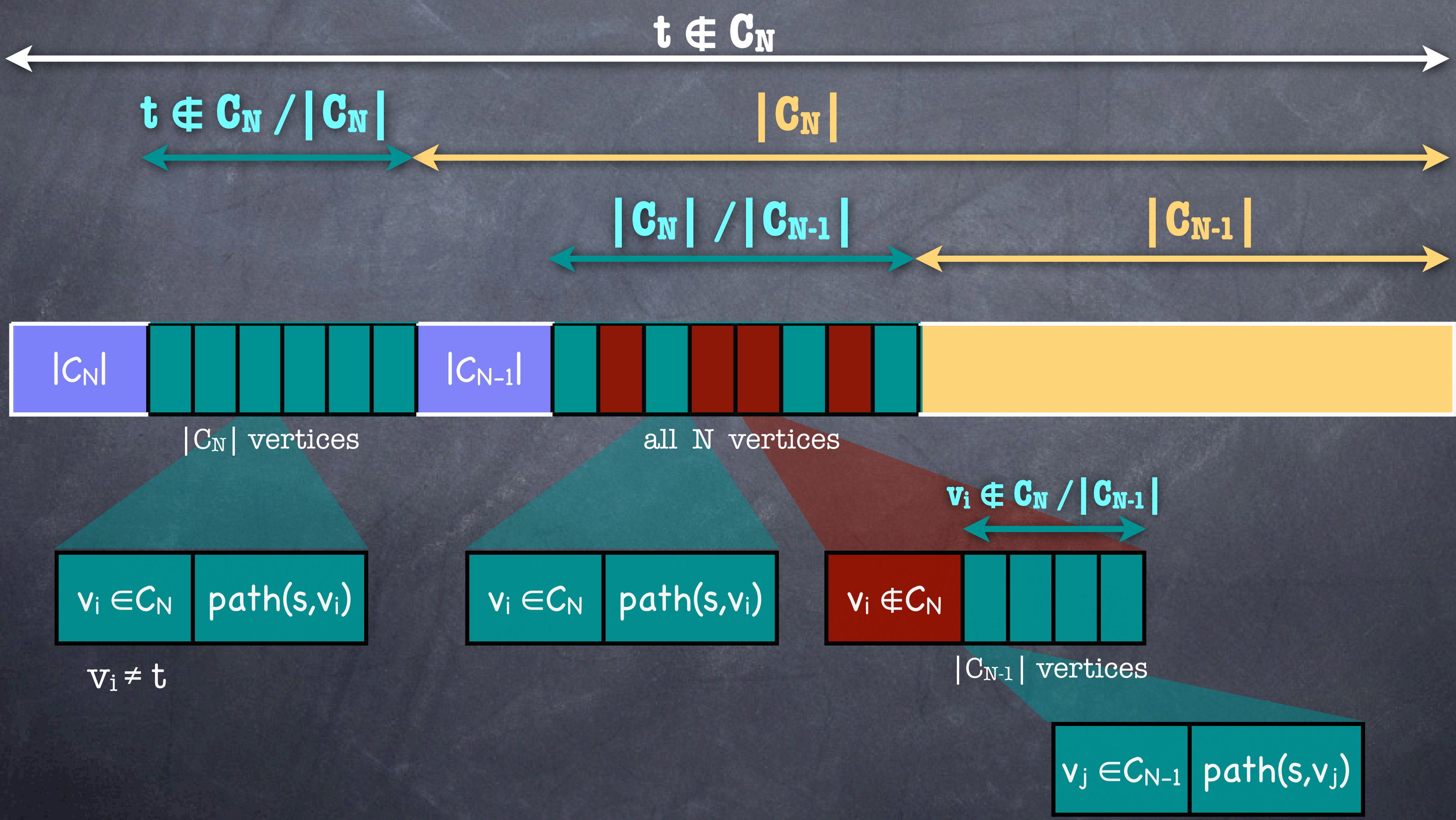
Certificate for $t \notin C_N$



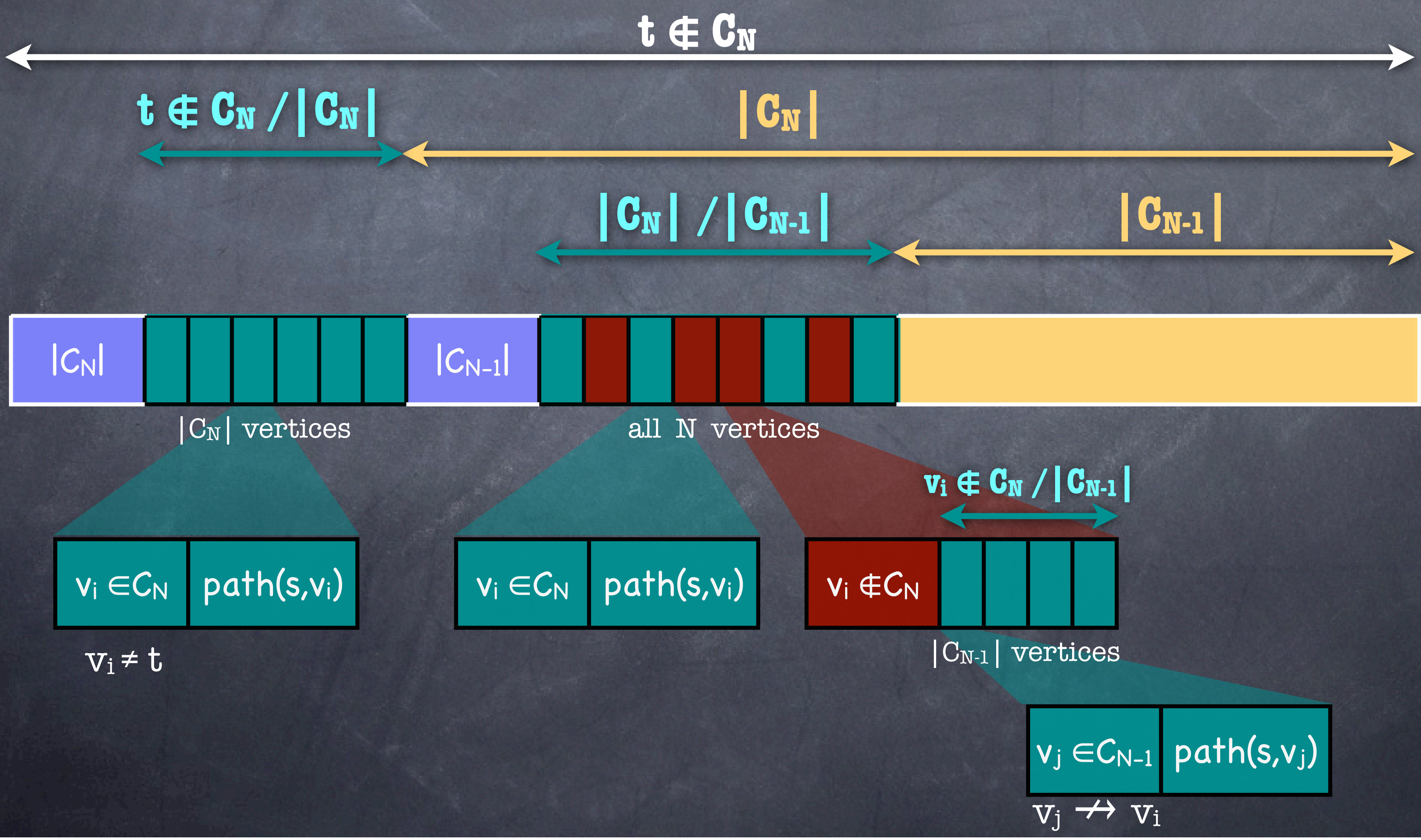
Certificate for $t \notin C_N$



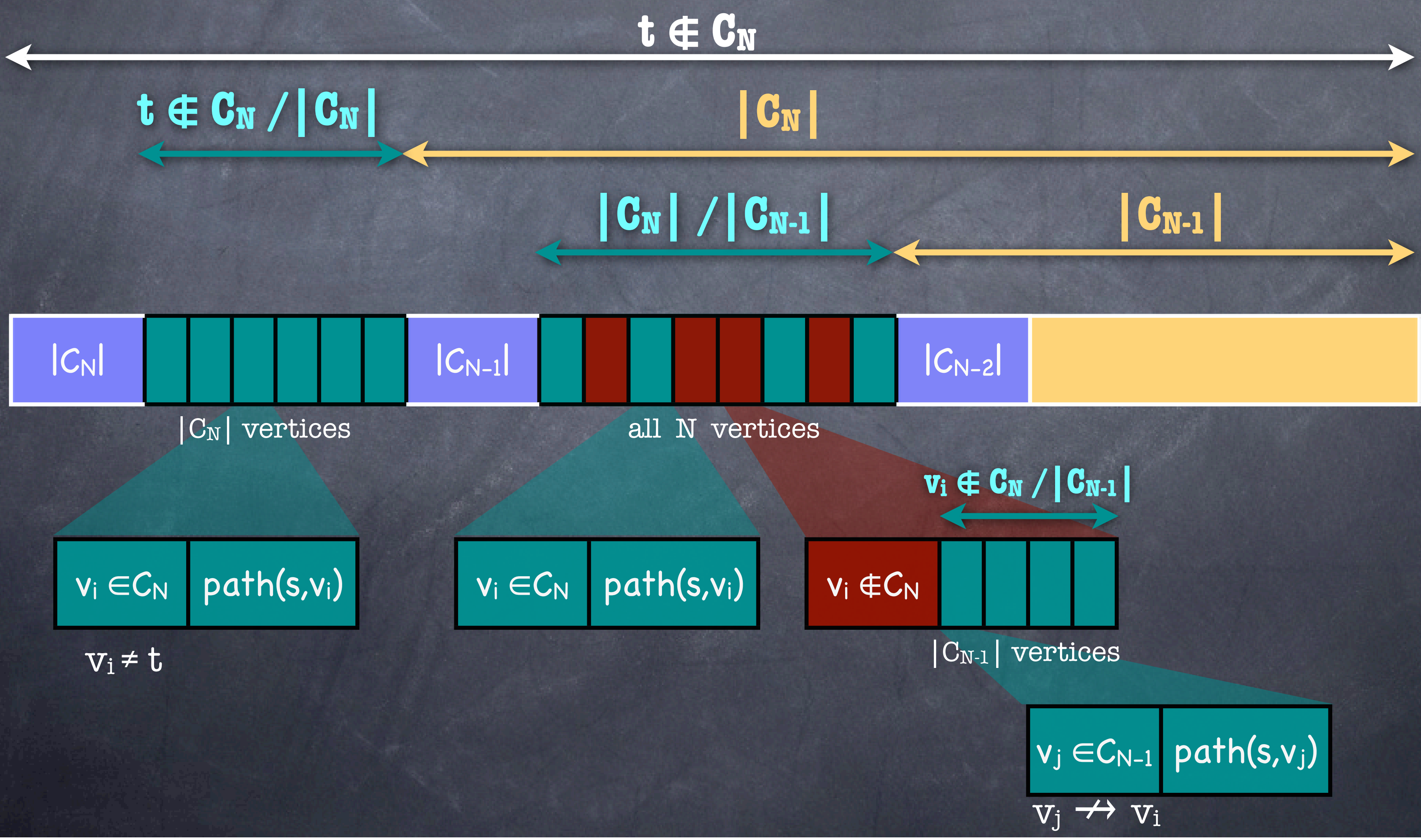
Certificate for $t \notin C_N$



Certificate for $t \notin C_N$



Certificate for $t \notin C_N$



Certificate for $t \notin C_N$