Reconfiguration Problems and Token Jumping Logic

Kord Eickmeyer,
jww Yota Otachi (Nagoya) and Tatsuya Gima (Sapporo)

AIMoTh, Passau, 3 March 2026

Warm-up: The Clique Problem

The following problem is well known to be NP-complete:

Clique

Given a graph $G = (V, E)$ and a number $k \geq 1$, decide if there is a clique of size k in G.

Warm-up: The Clique Problem

The following problem is well known to be NP-complete:

Clique

Given a graph $G = (V, E)$ and a number $k \geq 1$, decide if there is a clique of size k in G.

equivalent problems (under ptime-Turing reductions):

- search: given G and k, find a clique of size k
- maximisation: given G, find a largest clique / the size of a largest clique

Warm-up: The Clique Problem

The following problem is well known to be NP-complete:

Clique

Given a graph $G = (V, E)$ and a number $k \geq 1$, decide if there is a clique of size k in G.

equivalent problems (under ptime-Turing reductions):

- search: given G and k, find a clique of size k
- maximisation: given G, find a largest clique / the size of a largest clique

parameterised complexity (p-Clique): W[1]-complete

Warm-up: The Clique Problem

The following problem is well known to be NP-complete:

Clique

Given a graph $G = (V, E)$ and a number $k \geq 1$, decide if there is a clique of size k in G.

equivalent problems (under ptime-Turing reductions):

- search: given G and k, find a clique of size k
- maximisation: given G, find a largest clique / the size of a largest clique

parameterised complexity (p-Clique): W[1]-complete
hard to approximate

Reconfiguration Problems

Clique Reconfigurations

Given a graph $G = (V, E)$ and two cliques $C, C' \subseteq V$ with $|C| = |C'|$, a reconfiguration sequence from C to C' is a sequence

$$C = C_1, C_2, \ldots, C_\ell = C'$$

of *cliques* such that $|C_i \setminus C_{i\pm1}| = 1$ for $1 \leq i \leq \ell$.

Reconfiguration Problems

Clique Reconfigurations

Given a graph $G = (V, E)$ and two cliques $C, C' \subseteq V$ with $|C| = |C'|$, a reconfiguration sequence from C to C' is a sequence

$$C = C_1, C_2, \ldots, C_\ell = C'$$

of *cliques* such that $|C_i \setminus C_{i\pm1}| = 1$ for $1 \leq i \leq \ell$.

This is called *token jumping*, in token *sliding* we may only replace a vertex by one of its neighbours.

Reconfiguration Problems

Clique Reconfigurations

Given a graph $G = (V, E)$ and two cliques $C, C' \subseteq V$ with $|C| = |C'|$, a reconfiguration sequence from C to C' is a sequence

$$C = C_1, C_2, \ldots, C_\ell = C'$$

of *cliques* such that $|C_i \setminus C_{i\pm1}| = 1$ for $1 \leq i \leq \ell$.

This is called *token jumping*, in token *sliding* we may only replace a vertex by one of its neighbours.

Ito et al., 2011

The following problem is PSPACE-complete: Given a graph $G = (V, E)$ and two cliques $C, C' \subseteq V$, decide if there is a reconfiguration sequence from C to C'.

Variations on the Theme of Reconfiguration

cliques $\rightarrow$ definable sets (FO, MSO)

Variations on the Theme of Reconfiguration

cliques $\rightarrow$ definable sets (FO, MSO)
restricted graph classes (paths, coloured paths)

Variations on the Theme of Reconfiguration

cliques $\rightarrow$ definable sets (FO, MSO)
restricted graph classes (paths, coloured paths)
parameterised complexity

Variations on the Theme of Reconfiguration

cliques $\rightarrow$ definable sets (FO, MSO)
restricted graph classes (paths, coloured paths)
parameterised complexity

Results in this talk

- FO-Reconfiguration on paths is PSPACE-complete
- with parameter $|X|$,
  FO-Reconfiguration on coloured paths is W[1]-hard
- FO+Token Jumping $\equiv$ FO+symmetric transitive closure

FO/MSO-Reconfiguration

FO-Reconfiguration

Let $\varphi(X)$ be a first-order formula with a free set variable X. Given a graph $G = (V, E)$ and two sets $A, B \subseteq V$ such that both A and B satisfy φ, is there a reconfiguration sequence from A to B?

example: $A \subseteq V$ satisfies

$$\forall x \forall y \left((Xx \wedge Xy) \rightarrow (x \doteq y \vee Exy) \right)$$

if, and only if, A is a clique.

FO/MSO-Reconfiguration

FO-Reconfiguration

Let $\varphi(X)$ be a first-order formula with a free set variable X. Given a graph $G = (V, E)$ and two sets $A, B \subseteq V$ such that both A and B satisfy φ, is there a reconfiguration sequence from A to B?

example: $A \subseteq V$ satisfies

$$\forall x \forall y \left((Xx \wedge Xy) \rightarrow (x \doteq y \vee Exy) \right)$$

if, and only if, A is a clique.
similar: MSO-Reconfiguration.

First Result: FO-Reconfiguration on Paths

FO-Reconfiguration on Paths

There is a first-order formula $\varphi(X)$ for which the following problem is PSPACE-complete: Given a path $P = (V, E)$ and two sets $A, B \subseteq V$ such that both A and B satisfy φ, decide if there is a reconfiguration sequence from A to B.

note: clique reconfiguration on paths is trivial

First Result: FO-Reconfiguration on Paths

FO-Reconfiguration on Paths

There is a first-order formula $\varphi(X)$ for which the following problem is PSPACE-complete: Given a path $P = (V, E)$ and two sets $A, B \subseteq V$ such that both A and B satisfy φ, decide if there is a reconfiguration sequence from A to B.

note: clique reconfiguration on paths is trivial
equivalent formulation: For some locally threshold testable regular language $L \subseteq \{0,1\}^*$, the following problem is PSPACE-complete:

L-Reconfiguration

Given two words $u, v \in L$, decide if there is a sequence

$$u = w_1, w_2, \ldots, w_\ell = v$$

such that $w_i \in L$, $|w_i|_1 = |w_{i+1}|_1$, and $|w_i \oplus w_{i+1}|_1 = 2$ for every $1 \leq i < \ell$.

2-Balanced Symmetric String Rewriting

We reduce from the following problem:

Wrochna, 2018

There is a finite alphabet Σ and a string rewriting system with symmetric rewriting rules of the forms

$$ab \leftrightarrow ab' \qquad \text{and} \qquad ab \leftrightarrow a'b$$

with $a, a', b, b' \in \Sigma$ for which the rewriting problem is PSPACE-complete.

2-Balanced Symmetric String Rewriting

We reduce from the following problem:

Wrochna, 2018

There is a finite alphabet Σ and a string rewriting system with symmetric rewriting rules of the forms

$$ab \leftrightarrow ab' \qquad \text{and} \qquad ab \leftrightarrow a'b$$

with $a, a', b, b' \in \Sigma$ for which the rewriting problem is PSPACE-complete.

encode position of read/write-head and state in the alphabet Σ
symmetry needs some fiddling around

2-Balanced Symmetric String Rewriting

We reduce from the following problem:

Wrochna, 2018

There is a finite alphabet Σ and a string rewriting system with symmetric rewriting rules of the forms

$$ab \leftrightarrow ab' \qquad \text{and} \qquad ab \leftrightarrow a'b$$

with $a, a', b, b' \in \Sigma$ for which the rewriting problem is PSPACE-complete.

encode position of read/write-head and state in the alphabet Σ
symmetry needs some fiddling around

however, we need $\Sigma = \{0, 1\}$ and have to encode the rewriting rules in L

Reducing the Alphabet Size

Let $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$. We encode a string $w_1 w_2 \ldots w_\ell \in \Sigma^*$ as

$$1^3 01^3 00 \underbrace{\alpha_1 00 \beta_1 00 \gamma_1}_{\text{block for } w_1} 001^4 01^4 00 \underbrace{\alpha_2 00 \beta_2 00 \gamma_2}_{\text{block for } w_2} 001^5 01^5 \ldots$$

$$\ldots 00 \underbrace{\alpha_3 00 \beta_3 00 \gamma_3}_{\text{block for } w_3} 001^3 01^3 00 \ldots$$

Reducing the Alphabet Size

Let $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$. We encode a string $w_1 w_2 \ldots w_\ell \in \Sigma^*$ as

$$1^3 0 1^3 00 \underbrace{\alpha_1 00 \beta_1 00 \gamma_1}_{\text{block for } w_1} 00 1^4 0 1^4 00 \underbrace{\alpha_2 00 \beta_2 00 \gamma_2}_{\text{block for } w_2} 00 1^5 0 1^5 \ldots$$

$$\ldots 00 \underbrace{\alpha_3 00 \beta_3 00 \gamma_3}_{\text{block for } w_3} 00 1^3 0 1^3 00 \ldots$$

Every $\alpha_i/\beta_i/\gamma_i$ will contain exactly one 1, so the 1s in these blocks can never be moved and allow us to retrieve the direction from left to right in a path.

Reducing the Alphabet Size

Let $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$. We encode a string $w_1 w_2 \ldots w_\ell \in \Sigma^*$ as

$$1^3 0 1^3 0 0 \underbrace{\alpha_1 00 \beta_1 00 \gamma_1}_{\text{block for } w_1} 0 0 1^4 0 1^4 0 0 \underbrace{\alpha_2 00 \beta_2 00 \gamma_2}_{\text{block for } w_2} 0 0 1^5 0 1^5 \ldots$$

$$\ldots 0 0 \underbrace{\alpha_3 00 \beta_3 00 \gamma_3}_{\text{block for } w_3} 0 0 1^3 0 1^3 0 0 \ldots$$

we start with $\alpha_i = \beta_i = 0^{j-1} 1 0^{k-j}$ if $w_i = \sigma_j$.

If the rules are

$$S_r \colon a_1^{(r)} b_1^{(r)} \leftrightarrow a_2^{(r)} b_2^{(r)}$$

for $r = 1, \ldots, m$ we start with

$$\gamma_i = 10^m.$$

Intended meaning:

$$\gamma_i = 10^m \Rightarrow w_i \text{ and } w_{i+1} \text{ may not be changed}$$

$$\gamma_i = 0^r 10^{m-r} \Rightarrow \text{rule } S_r \text{ is applied to } w_i w_{i+1}.$$

Reducing the Alphabet Size

Let $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$. We encode a string $w_1 w_2 \ldots w_\ell \in \Sigma^*$ as

$$1^3 0 1^3 0 0 \underbrace{\alpha_1 00 \beta_1 00 \gamma_1}_{\text{block for } w_1} 0 0 1^4 0 1^4 0 0 \underbrace{\alpha_2 00 \beta_2 00 \gamma_2}_{\text{block for } w_2} 0 0 1^5 0 1^5 \ldots$$

$$\ldots 0 0 \underbrace{\alpha_3 00 \beta_3 00 \gamma_3}_{\text{block for } w_3} 0 0 1^3 0 1^3 0 0 \ldots$$

We demand:

- at most one γ_i is not equal to 10^m
- if $\gamma_i = 10^m$ then $\alpha_i = \beta_i$ and $\alpha_{i+1} = \beta_{i+1}$.
- if $\gamma_i = 0^r 10^{m-r}$ and $S_r \colon a_1^{(r)} b_1^{(r)} \leftrightarrow a_2^{(r)} b_2^{(r)}$, then α_i, β_i encode $a_1^{(r)}$ or $a_2^{(r)}$ and $\alpha_{i+1}, \beta_{i+1}$ encode $b_1^{(r)}$ or $b_2^{(r)}$.

This is definable in FO.

Second Result: W[1]-hardness

There is an FO-formula $\varphi(X)$ for which the following parameterised problem is W[1]-hard:

p-φ-Reconf on Coloured Paths

Instance:	coloured path P and two subsets A, B of vertices.
Parameter:	$\lvert A\rvert \quad (= \lvert B\rvert)$
Problem:	check if there is a reconfiguration sequence from A to B.

Second Result: W[1]-hardness

There is an FO-formula $\varphi(X)$ for which the following parameterised problem is W[1]-hard:

p-φ-Reconf on Coloured Paths

Instance:	coloured path P and two subsets A, B of vertices.
Parameter:	$\lvert A\rvert \quad (= \lvert B\rvert)$
Problem:	check if there is a reconfiguration sequence from A to B.

- easily seen to be in XP

Second Result: W[1]-hardness

There is an FO-formula $\varphi(X)$ for which the following parameterised problem is W[1]-hard:

p-φ-Reconf on Coloured Paths

Instance: coloured path P and two subsets A, B of vertices.
Parameter: $|A| \quad (= |B|)$
Problem: check if there is a reconfiguration sequence from A to B.

- easily seen to be in XP
- by FO-interpretations: hardness on caterpillar grahs

Second Result: W[1]-hardness

There is an FO-formula $\varphi(X)$ for which the following parameterised problem is W[1]-hard:

p-φ-Reconf on Coloured Paths

Instance: coloured path P and two subsets A, B of vertices.
Parameter: $|A| \quad (= |B|)$
Problem: check if there is a reconfiguration sequence from A to B.

- easily seen to be in XP
- by FO-interpretations: hardness on caterpillar grahs
- FPT-algorithm for MSO with parameter $|\varphi| + |A| +$ shrub-depth (not yet published)

Rephrasing in Terms of Languages

For $w = w_1 w_2 \dots w_\ell \in \Sigma^*$ and $A \subseteq \{1, \dots, \ell\}$ we define $w_A \in (\Sigma \times \{0,1\})^*$ by

$$w_A = v_1 v_2 \dots v_\ell \quad \text{with} \quad v_i = \begin{cases} (w_i, 0) & \text{if } i \neq A \\ (w_i, 1) & \text{if } i = A \text{ (\textit{token} on this position)} \end{cases}$$

Rephrasing in Terms of Languages

For $w = w_1 w_2 \dots w_\ell \in \Sigma^*$ and $A \subseteq \{1, \dots, \ell\}$ we define $w_A \in (\Sigma \times \{0,1\})^*$ by

$$w_A = v_1 v_2 \dots v_\ell \quad \text{with} \quad v_i = \begin{cases} (w_i, 0) & \text{if } i \neq A \\ (w_i, 1) & \text{if } i = A \text{ (\textit{token} on this position)} \end{cases}$$

We show that the following problem is W[1]-hard for some locally threshold testable regular language $L \subseteq (\Sigma \times \{0,1\})^*$:

p-L-Reconfiguration

Instance:	a word $w \in \Sigma^*$, two sets A, B of positions of w such that $w_A, w_B \in L$
Parameter:	$\lvert A \rvert \quad (= \lvert B \rvert)$
Problem:	check if there is a reconfiguration sequence from A to B.

Rephrasing in Terms of Languages

For $w = w_1 w_2 \dots w_\ell \in \Sigma^*$ and $A \subseteq \{1, \dots, \ell\}$ we define $w_A \in (\Sigma \times \{0,1\})^*$ by

$$w_A = v_1 v_2 \dots v_\ell \quad \text{with} \quad v_i = \begin{cases} (w_i, 0) & \text{if } i \neq A \\ (w_i, 1) & \text{if } i = A \text{ (\textit{token} on this position)} \end{cases}$$

We show that the following problem is W[1]-hard for some locally threshold testable regular language $L \subseteq (\Sigma \times \{0,1\})^*$:

p-L-Reconfiguration

Instance:	a word $w \in \Sigma^*$, two sets A, B of positions of w such that $w_A, w_B \in L$
Parameter:	$\lvert A \rvert \quad (= \lvert B \rvert)$
Problem:	check if there is a reconfiguration sequence from A to B.

note: decision problem (given $w \in \Sigma^*$ and k, is there a set A with $w_A \in L$ and $|A| = k$?) can be solved in time $O\big(n \cdot k \cdot f(L)\big)$

Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)

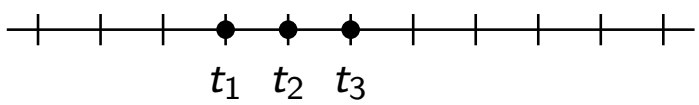
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



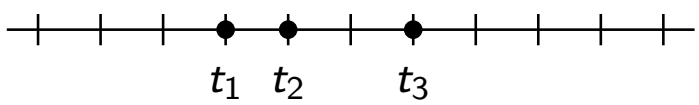
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



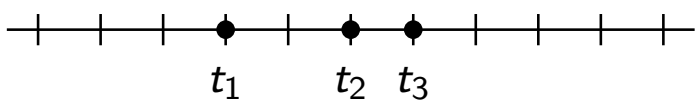
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



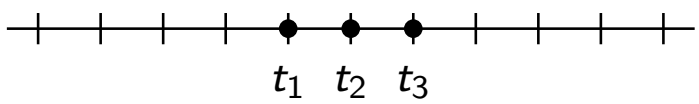
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



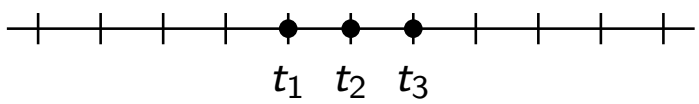
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:

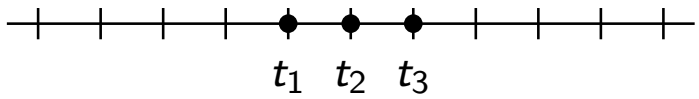
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

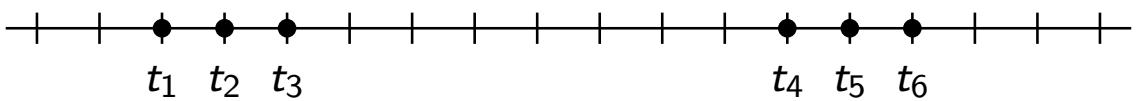
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:

$123 \Rightarrow 45$ $\quad 2\cdot \Rightarrow 4{\cdot\cdot}6$ $\quad \cdot 2 \Rightarrow 56$

$\cdot 5 \Rightarrow 1{\cdot\cdot}3$ $\quad 456 \Rightarrow 23$ $\quad 5\cdot \Rightarrow 12$



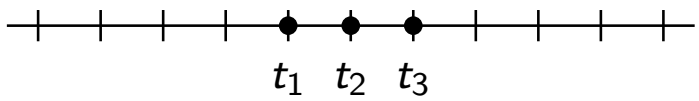
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

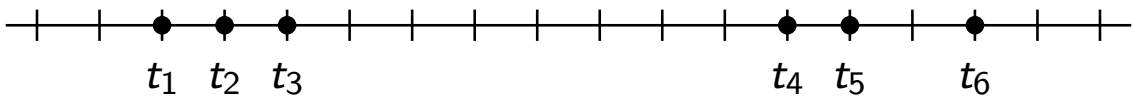
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:

$123 \Rightarrow 45$ $\quad 2\cdot \Rightarrow 4{\cdot\cdot}6$ $\quad \cdot 2 \Rightarrow 56$

$\cdot 5 \Rightarrow 1{\cdot\cdot}3$ $\quad 456 \Rightarrow 23$ $\quad 5\cdot \Rightarrow 12$



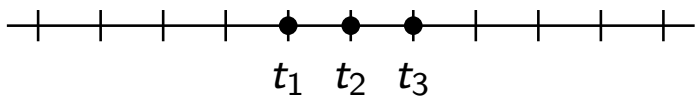
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

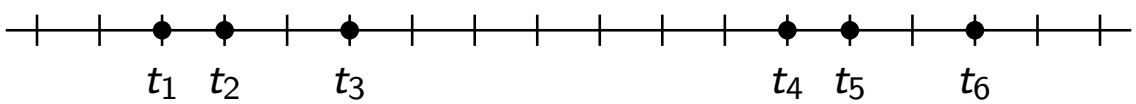
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:

$123 \Rightarrow 45$ $\quad 2\cdot \Rightarrow 4{\cdot\cdot}6$ $\quad \cdot 2 \Rightarrow 56$

$\cdot 5 \Rightarrow 1{\cdot\cdot}3$ $\quad 456 \Rightarrow 23$ $\quad 5\cdot \Rightarrow 12$



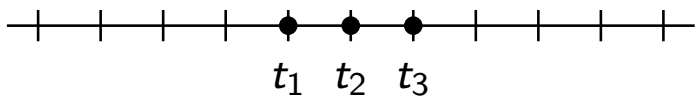
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

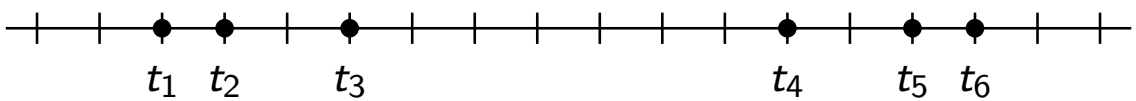
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:

$123 \Rightarrow 45$ $\quad 2\cdot \Rightarrow 4{\cdot\cdot}6$ $\quad \cdot 2 \Rightarrow 56$

$\cdot 5 \Rightarrow 1{\cdot\cdot}3$ $\quad 456 \Rightarrow 23$ $\quad 5\cdot \Rightarrow 12$



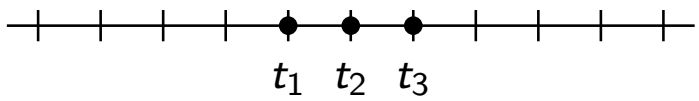
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

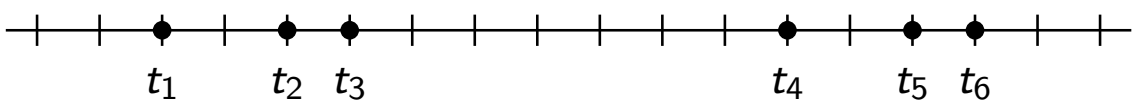
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:

$123 \Rightarrow 45$ $\quad 2\cdot \Rightarrow 4{\cdot\cdot}6$ $\quad \cdot 2 \Rightarrow 56$

$\cdot 5 \Rightarrow 1{\cdot\cdot}3$ $\quad 456 \Rightarrow 23$ $\quad 5\cdot \Rightarrow 12$



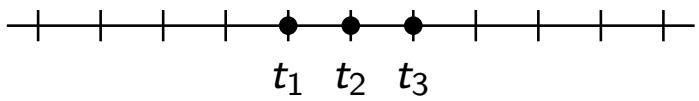
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

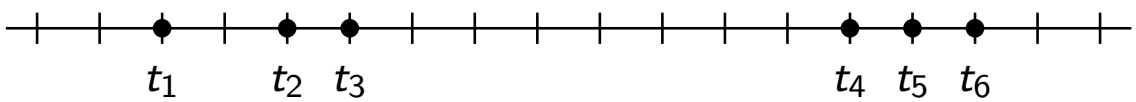
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:

$123 \Rightarrow 45$ $\quad 2\cdot \Rightarrow 4{\cdot\cdot}6$ $\quad \cdot 2 \Rightarrow 56$

$\cdot 5 \Rightarrow 1{\cdot\cdot}3$ $\quad 456 \Rightarrow 23$ $\quad 5\cdot \Rightarrow 12$



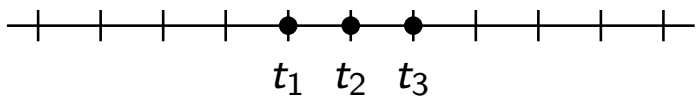
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

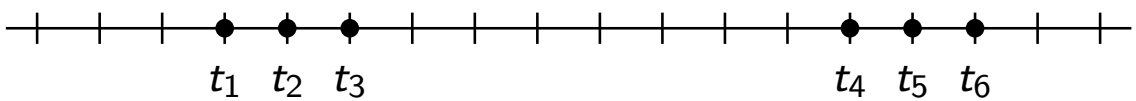
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:

$123 \Rightarrow 45$ $\quad 2\cdot \Rightarrow 4{\cdot\cdot}6$ $\quad \cdot 2 \Rightarrow 56$

$\cdot 5 \Rightarrow 1{\cdot\cdot}3$ $\quad 456 \Rightarrow 23$ $\quad 5\cdot \Rightarrow 12$



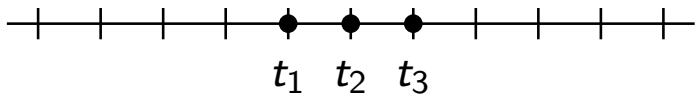
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

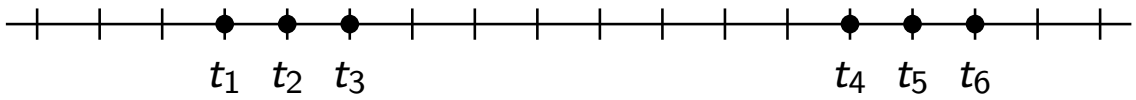
- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

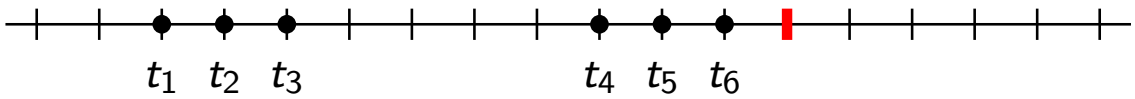
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:



- *partly synchronised sliders*:



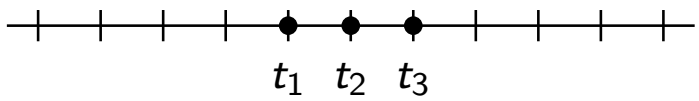
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

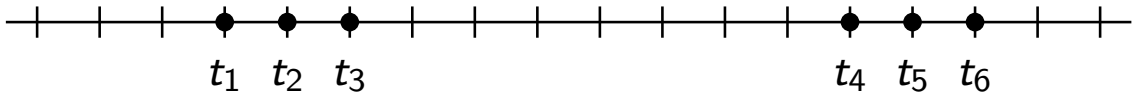
- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

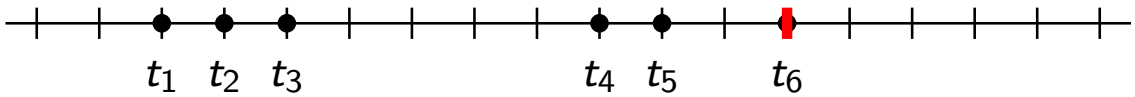
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:



- *partly synchronised sliders*:



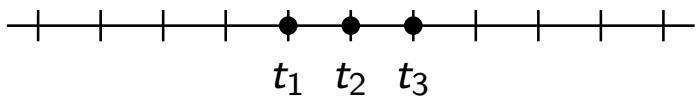
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

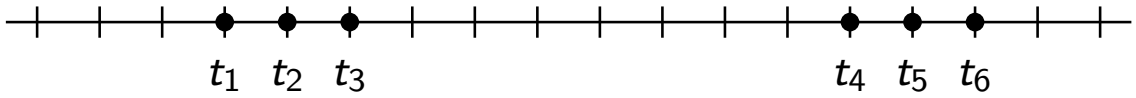
- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

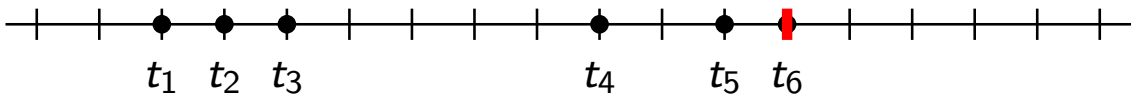
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:



- *partly synchronised sliders*:



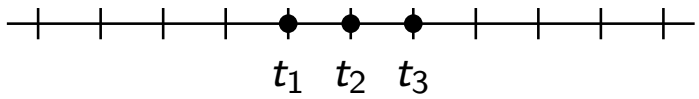
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

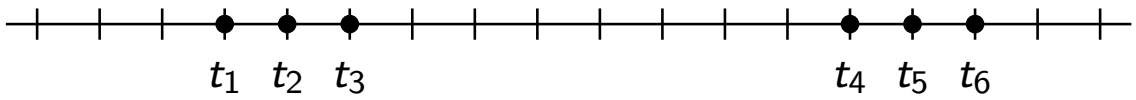
- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

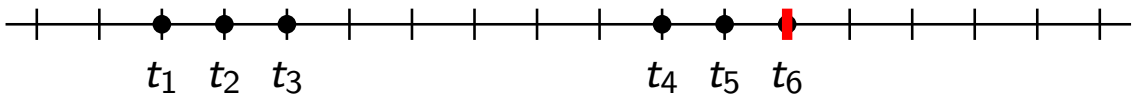
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:



- *partly synchronised sliders*:



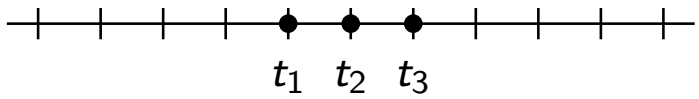
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

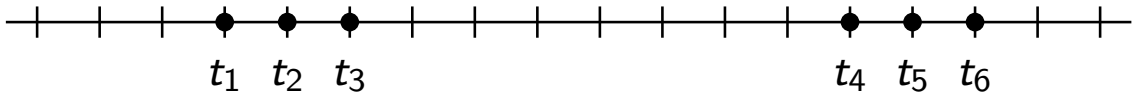
- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

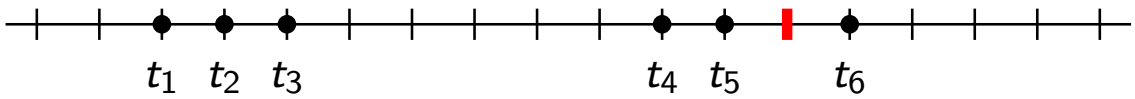
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:



- *partly synchronised sliders*:



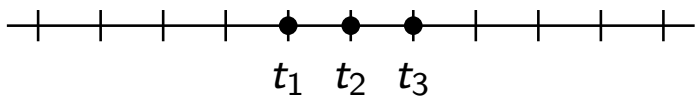
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

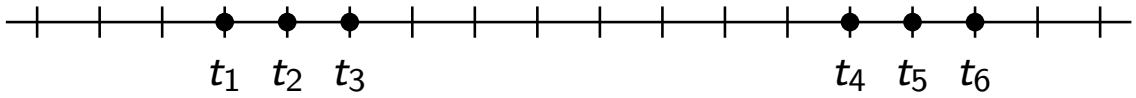
- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

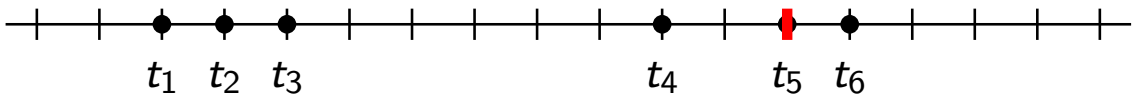
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:



- *partly synchronised sliders*:



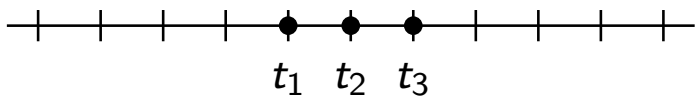
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

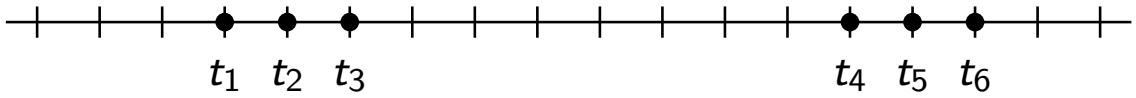
- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

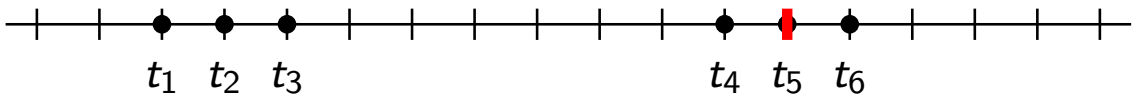
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:



- *partly synchronised sliders*:



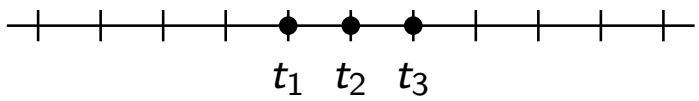
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

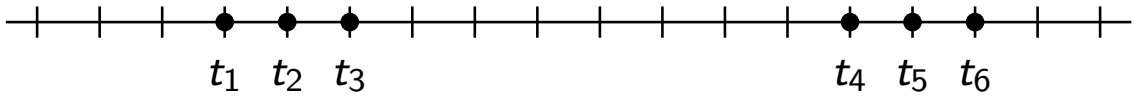
- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

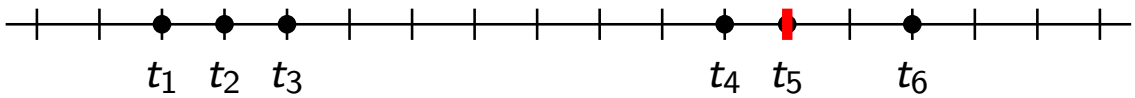
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:



- *partly synchronised sliders*:



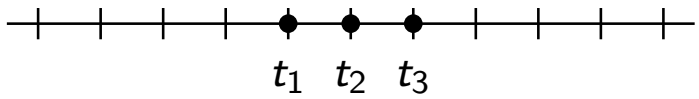
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

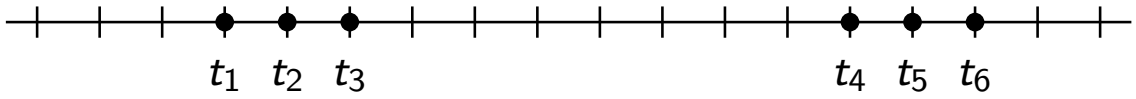
- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

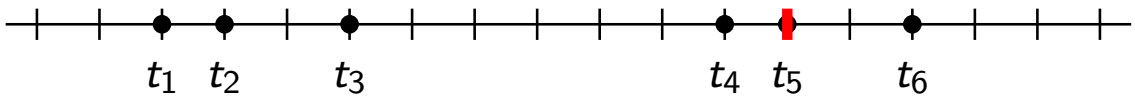
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:



- *partly synchronised sliders*:



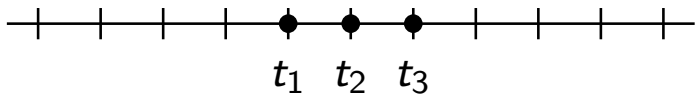
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

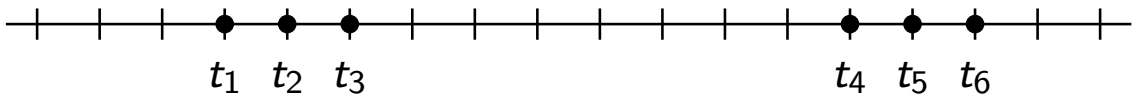
- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

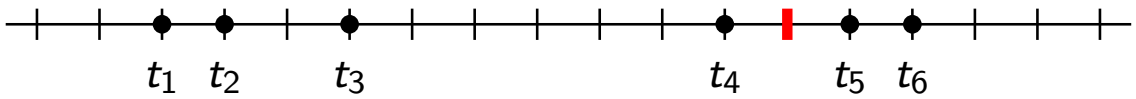
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:



- *partly synchronised sliders*:



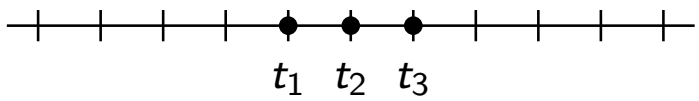
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

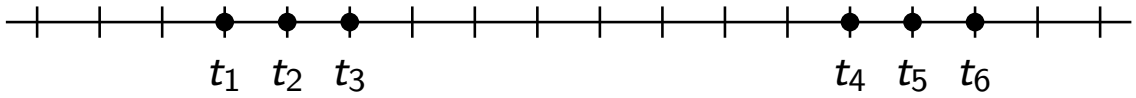
- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

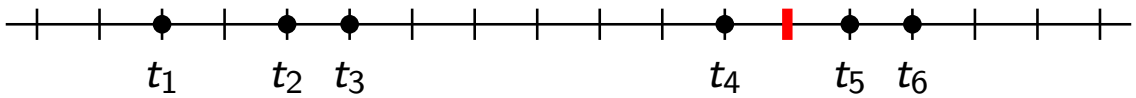
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:



- *partly synchronised sliders*:



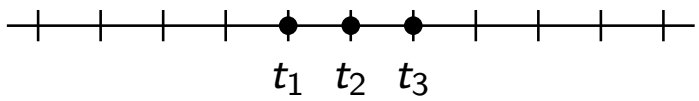
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

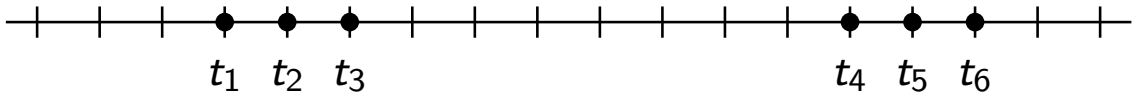
- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

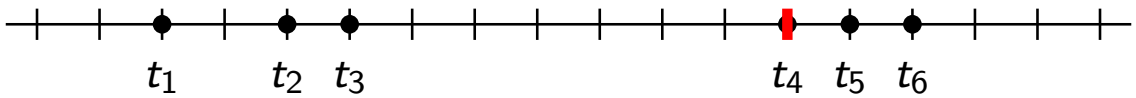
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:



- *partly synchronised sliders*:



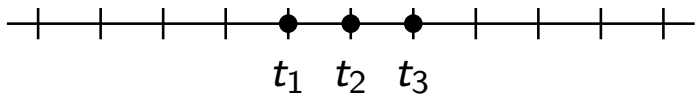
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

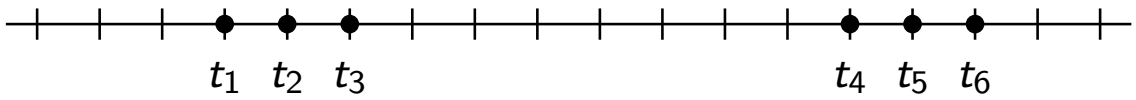
- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

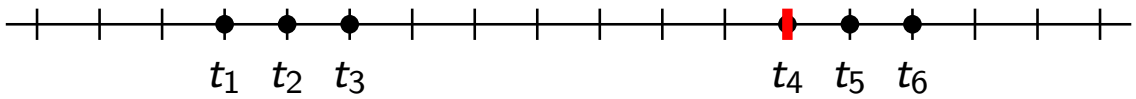
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:



- *partly synchronised sliders*:



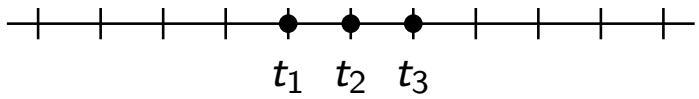
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

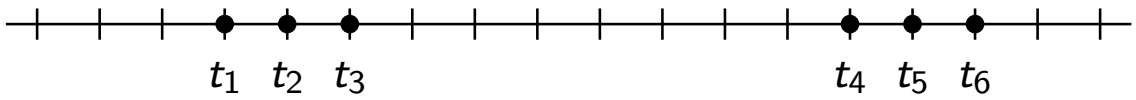
- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

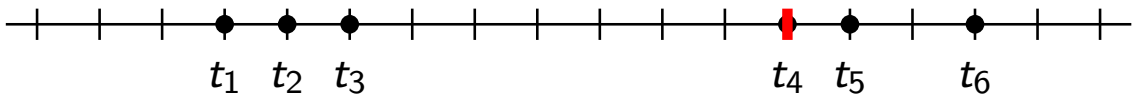
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:



- *partly synchronised sliders*:



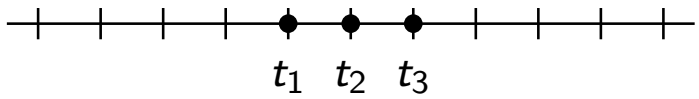
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

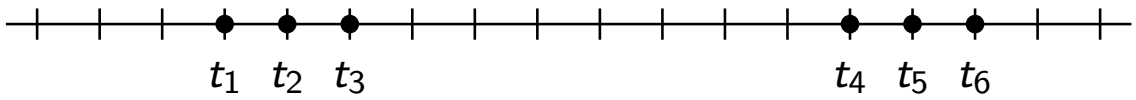
- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

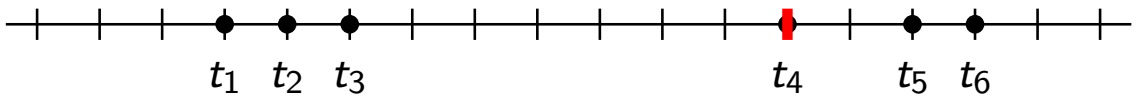
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:



- *partly synchronised sliders*:



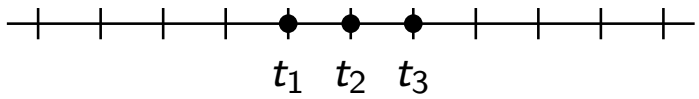
Reducing p-Clique to p-L-Reconfiguration

Basic idea: Given $G = (V, E)$ and $k \geq 1$,

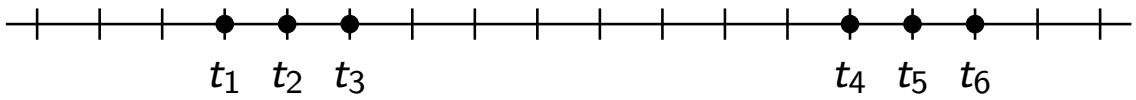
- first place k tokens on vertices
- then read the adjacency matrix of G line by line, checking that any pair of tokenised vertices is adjacent

Some ingredients:

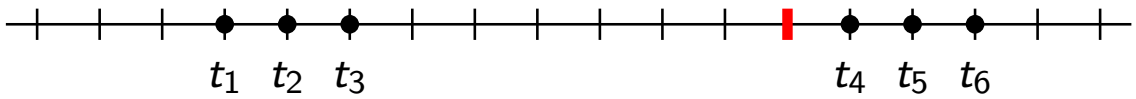
- tokens may have types $1, \ldots, s$
  (s being independent of the number of tokens)
- *sliders*: tokens t_1, t_2, t_3 with $t_1 < t_2 < t_3$ and $d(t_1, t_3) \leq 3$:



- *synchronised sliders*:



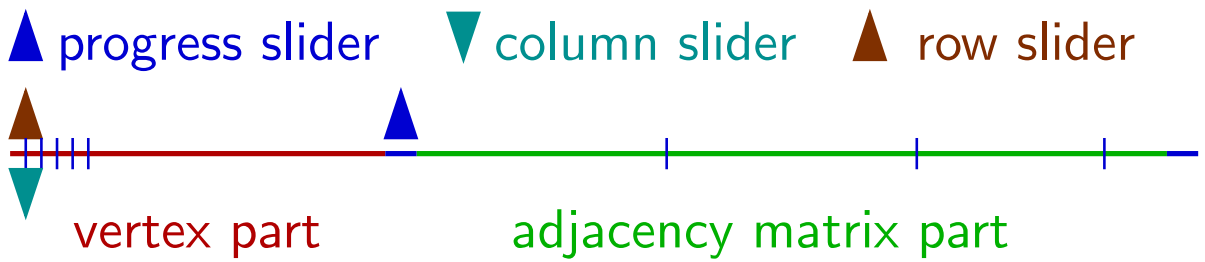
- *partly synchronised sliders*:



Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")



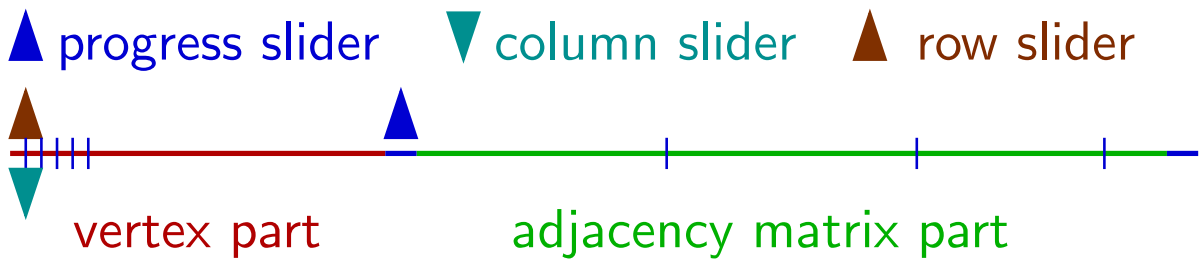
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



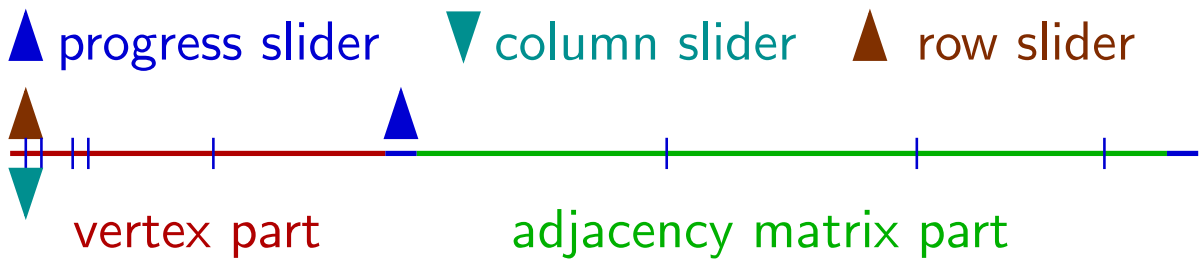
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



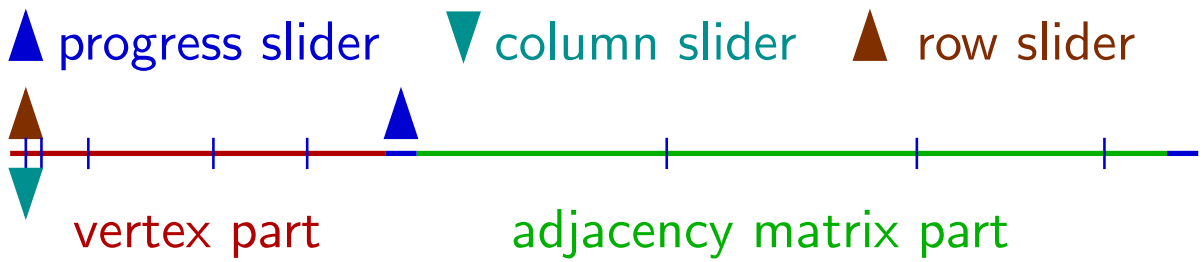
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



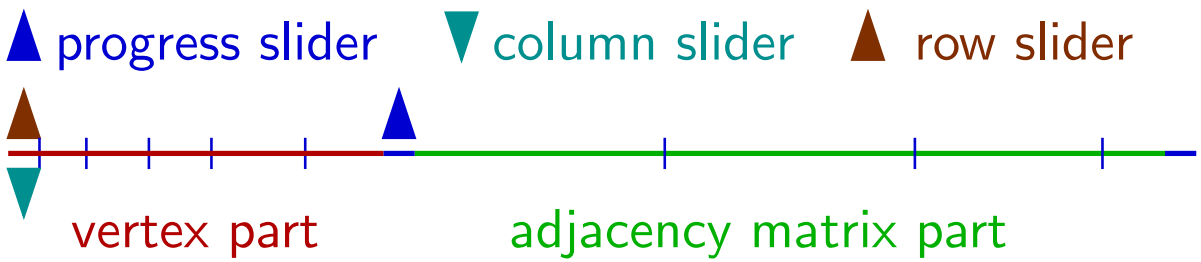
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



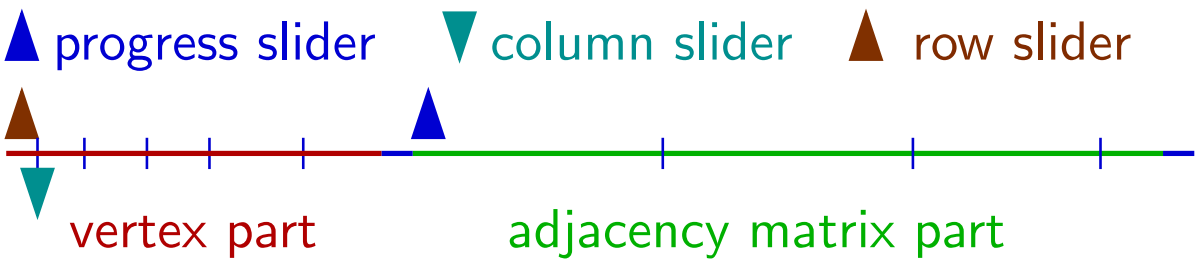
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



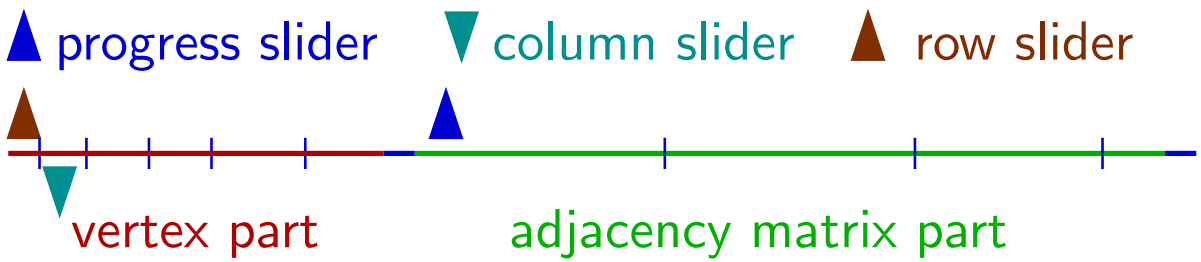
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



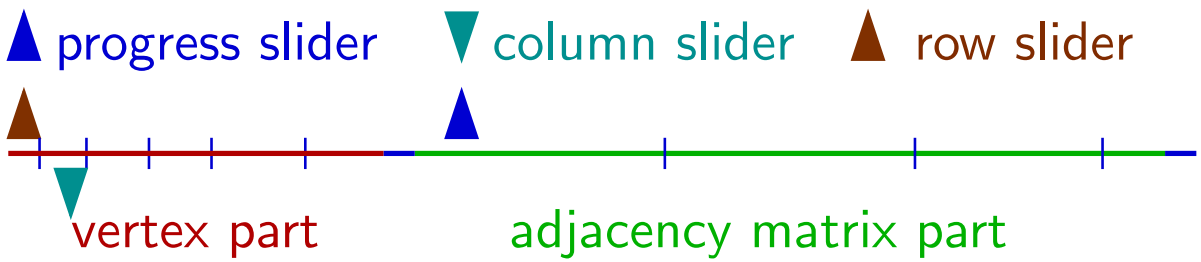
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



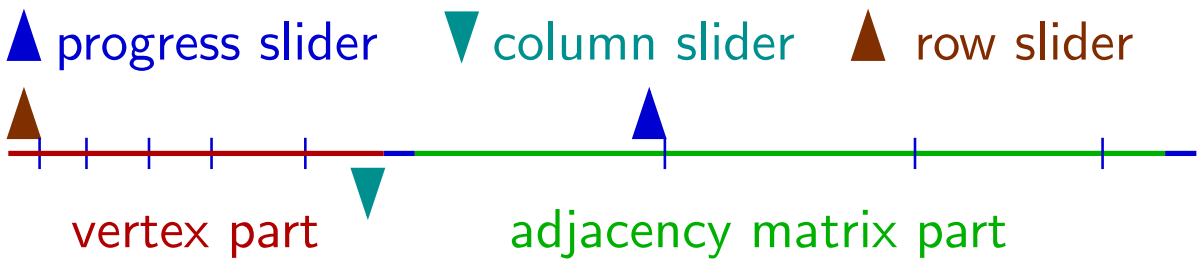
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



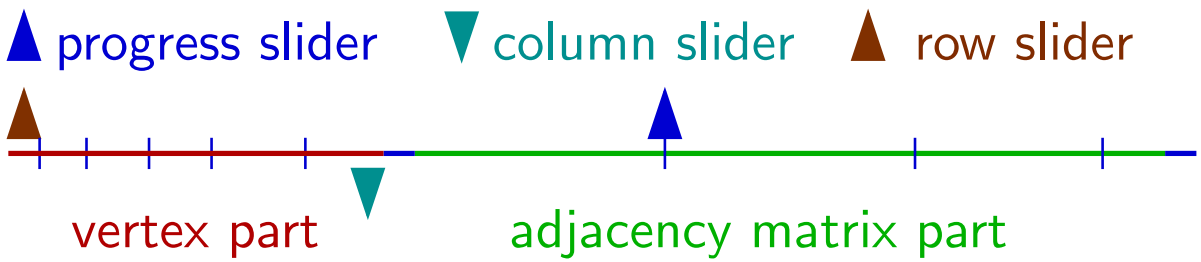
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



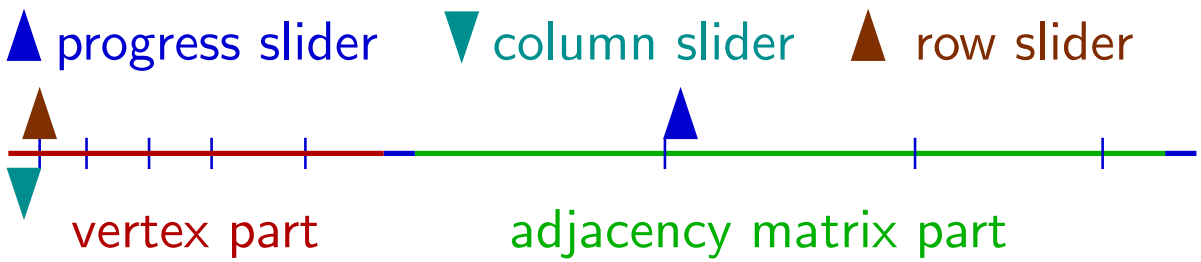
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



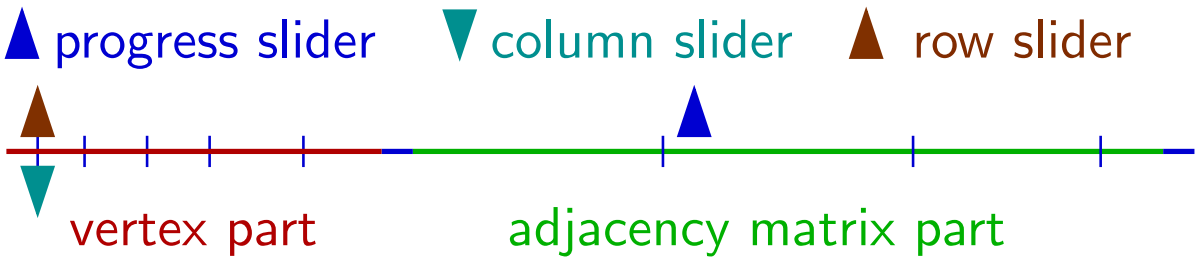
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



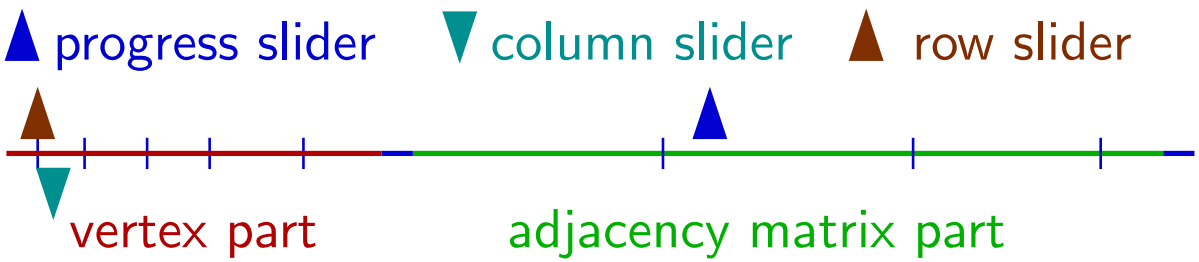
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



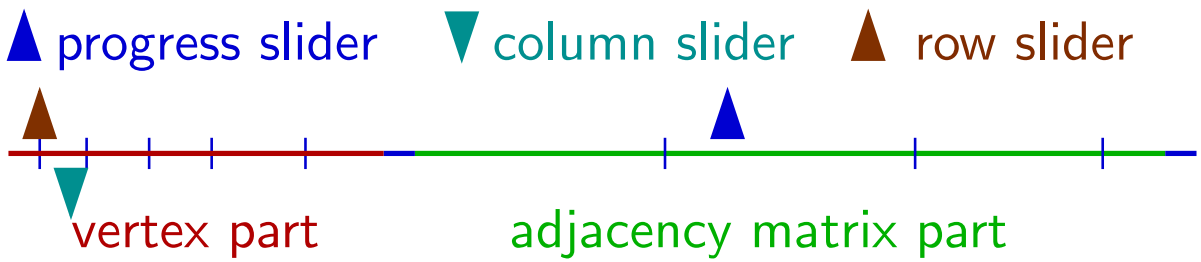
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



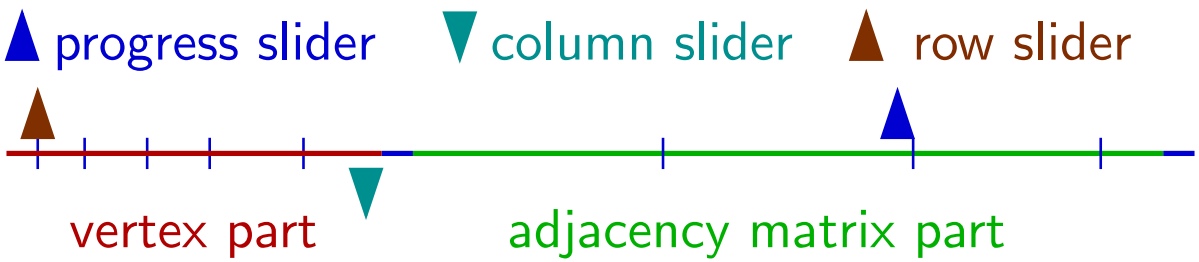
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



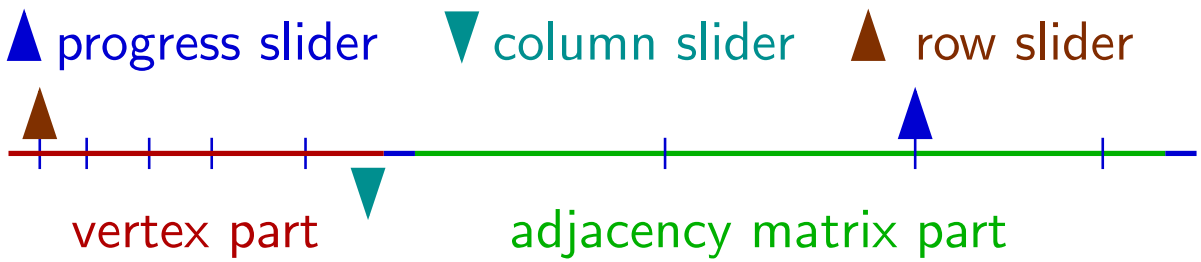
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



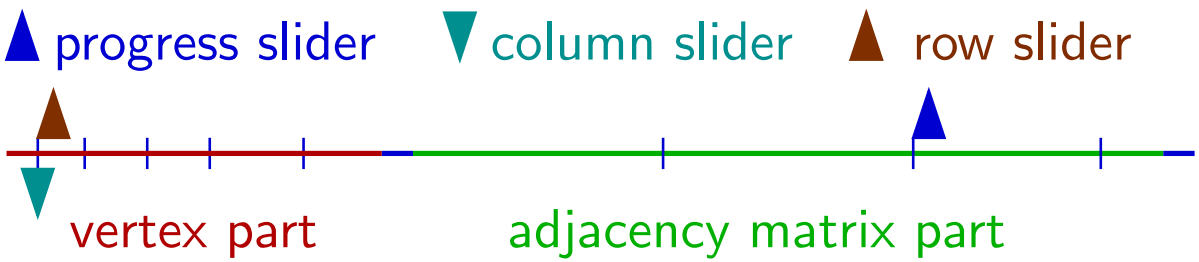
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



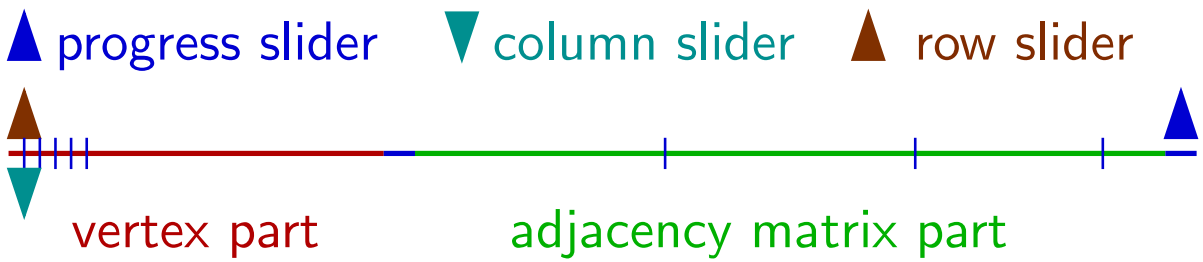
Putting it all together

Two parts of the string:

1. vertex part
   (letters for each vertex, tokenised $\hat{=}$ in clique)
2. adjacency matrix part
   (letters for each pair of vertices, "carriage return gadgets")

Three sliders:

1. progress slider, must be moved all along the adjacency matrix part
2. column slider: moves in sync with progress slider,
   reset by carriage return
3. row slider:
   moves each time the progress slider moves past a carriage return



Token Jumping Logic

FO with Token Jumping – FO(TJ)

For every $\varphi(\bar{x}, \bar{z})$ introduce a formula

$$[\mathrm{TJ}_{\bar{x}}\varphi](\bar{s}, \bar{t})$$

where $\bar{x}$ is a tuple of variables and $\bar{s}$, $\bar{t}$ are tuples of terms of the same length.
In a structure $\mathcal{A}$, for every tuple $\bar{b} \in A^{|\bar{z}|}$ the formula φ defines a set

$$\varphi(A, \bar{b}) := \{\bar{a} \in A^{|x|} \mid \mathcal{A} \models \varphi[\bar{a}, \bar{b}]\}$$

Then

$$\mathcal{A} \models [\mathrm{TJ}_{\bar{x}}\varphi](\bar{s}, \bar{t})[\bar{b}]$$

if there is a reconfiguration sequence from $\bar{s}$ to $\bar{t}$ in $\varphi(A, \bar{b})$.

FO(STC)

FO with symmetric transitive closure – FO(STC)

For every $\varphi(\bar{x}, \bar{y}, \bar{z})$ introduce a formula

$$[\mathrm{STC}_{\bar{x},\bar{y}}\varphi](\bar{s}, \bar{t})$$

where $\bar{x}$ and $\bar{y}$ are tuples of variables and $\bar{s}$, $\bar{t}$ are tuples of terms of the same length.
In a structure $\mathcal{A}$, for every tuple $\bar{b} \in A^{|\bar{z}|}$ the formula φ defines a symmetric binary relation

$$\{(\bar{a}_1, \bar{a}_2) \in \left(A^{|x|}\right)^2 \mid \mathcal{A} \models \varphi[\bar{a}_1, \bar{a}_2, \bar{b}] \text{ or } \mathcal{A} \models \varphi[\bar{a}_2, \bar{a}_1, \bar{b}]\}$$

on $A^{|\bar{x}|}$. Then

$$\mathcal{A} \models [\mathrm{STC}_{\bar{x},\bar{y}}\varphi](\bar{s}, \bar{t})[\bar{b}]$$

if a path between $\bar{s}$ and $\bar{t}$ in this graph.

FO(TJ) $\equiv$ FO(STC)

Third Result: FO(TJ) $\equiv$ FO(STC)

For every $\varphi \in \mathrm{FO}(\mathrm{TJ})$ there is an equivalent $\varphi' \in \mathrm{FO}(\mathrm{STC})$ and vice versa.

FO(TJ) ≡ FO(STC)

Third Result: FO(TJ) ≡ FO(STC)

For every $\varphi \in \mathrm{FO}(\mathrm{TJ})$ there is an equivalent $\varphi' \in \mathrm{FO}(\mathrm{STC})$ and vice versa.

- note that token jumping is inherently symmetric.
- equivalence is not hard to show, need to make sure changes happen one entry at a time
- FO(DTC) (deterministic transitive closure) captures logspace *on ordered structures*, and symmetric logspace is equal to logspace. What can we say about FO(STC)?

Conclusion / Open Questions

We have shown:

- PSPACE-completeness for FO-Reconfiguration on paths
- W[1]-hardness for parameterised FO-Reconfiguration on caterpillars / coloured paths
- equivalence of FO with token jumping and FO with symmetric transitive closure

Conclusion / Open Questions

We have shown:

- PSPACE-completeness for FO-Reconfiguration on paths
- W[1]-hardness for parameterised FO-Reconfiguration on caterpillars / coloured paths
- equivalence of FO with token jumping and FO with symmetric transitive closure

Open Questions

- AW[$*$]-hardness for parameterised MSO-Reconfiguration on caterpillars / coloured paths?
- parameterised complexity of FO-Reconfiguration on paths?
- normal form for FO with token jumping?

Conclusion / Open Questions

We have shown:

- PSPACE-completeness for FO-Reconfiguration on paths
- W[1]-hardness for parameterised FO-Reconfiguration on caterpillars / coloured paths
- equivalence of FO with token jumping and FO with symmetric transitive closure

Open Questions

- AW[$*$]-hardness for parameterised MSO-Reconfiguration on caterpillars / coloured paths?
- parameterised complexity of FO-Reconfiguration on paths?
- normal form for FO with token jumping?

Thank you for your attention!