Vol.2015-AL-155 No.1
2015/11/20

Shortest Reconfiguration of Sliding Tokens on a Caterpillar

Takeshi Yamada[1,a)] Ryuhei Uehara[1,b)]

Abstract: Suppose that we are given two independent sets $\mathbf{I}_b$ and $\mathbf{I}_r$ of a graph such that $|\mathbf{I}_b| = |\mathbf{I}_r|$, and imagine that a token is placed on each vertex in $\mathbf{I}_b$. Then, the SLIDING TOKEN problem is to determine if there exists a sequence of independent sets which transforms $\mathbf{I}_b$ into $\mathbf{I}_r$ so that each independent set in the sequence results from the previous one by sliding exactly one token along an edge in the graph. The sliding token problem is one of the reconfiguration problems that attract the attention from the viewpoint of theoretical computer science. The reconfiguration problems tend to be PSPACE-complete in general, and some polynomial time algorithms are shown in restricted cases. Recently, the problems that aim at finding a shortest reconfiguration sequence are investigated. For the 3SAT problem, a trichotomy for the complexity of finding the shortest sequence has been shown; that is, it is in P, NP-complete, or PSPACE-complete in certain conditions. In general, even if it is polynomial time solvable to decide if two instances are reconfigured with each other, it can be NP-complete to find a shortest sequence between them. We show that the problem for finding a shortest sequence between two independent sets is polynomial time solvable for some graph classes; proper interval graphs, trivially perfect graphs, and caterpillars. As far as the authors know, this is the first polynomial time algorithm for the shortest sliding token problem for a graph class that requires detours.

1. Introduction

Recently, the *reconfiguration problems* attract the attention from the viewpoint of theoretical computer science. The problem arises when we wish to find a step-by-step transformation between two feasible solutions of a problem such that all intermediate results are also feasible and each step abides by a fixed reconfiguration rule, that is, an adjacency relation defined on feasible solutions of the original problem. The reconfiguration problems have been studied extensively for several well-known problems, including INDEPENDENT SET [9], [10], [11], [12], [16], SATISFIABILITY [8], [14], SET COVER, CLIQUE, and MATCHING [11].

The reconfiguration problem can be seen as a natural "puzzle."The *15 puzzle* is one of the most famous classic puzzles, that had the greatest impact on American and European society of any mechanical puzzle the word has ever known in 1880 (see [19] for the details). The 15 puzzle has a parity; for any two placements, we can decide if two placements are reconfigurable or not by the parity. Therefore, we can solve the reconfiguration problem in linear time just by computing their parities. Moreover, we can say that the distance between any two reconfigurable placements is $O(n^3)$, that is, we can reconfigure from one to the other in $O(n^3)$ sliding pieces on a $n \times n$ board. However, surprisingly, for these two reconfigurable placements, finding a shortest path is NP-complete [18]. Another interesting property of the 15 puzzle is in another generalization. In the 15 puzzle, every peace is a unit square of size 1×1. When we allow to use rectangles, we have the other classic puzzles, called "Dad puzzle" and its variants (see Fig. 1). Gardner said that "These puzzles are very much in want of a theory" in 1964 [7], and Hearn and Demaine have gave that after 40 years [9]; they prove that these puzzles are PSPACE-complete in general using their nondeterministic constraint logic model [10]. Therefore, we can characterize these three complexity classes using the model of sliding block puzzles.

From the viewpoint of theoretical computer science, one of the most important problems is the 3SAT problem. For this 3SAT problem, a similar trichotomy for the complexity of finding a shortest sequence has been shown; for the reconfiguration problem of 3SAT, finding a shortest sequence between two satisfiable assignments is in P, NP-complete, or PSPACE-complete in certain conditions [15]. In general, the reconfiguration problems tend to be PSPACE-complete, and some polynomial time algorithms are shown in restricted cases. In the reconfiguration problems, finding a shortest sequence can be a new trend in theoretical computer science because it has a great potential to characterize the class NP from a new viewpoint.

Beside the 3SAT problem, one of the most important problems is the independent set problem. For this notion, the natural reconfiguration problem is called the SLIDING TOKEN problem introduced by Hearn and Demaine [9]: Suppose that we are given two independent sets $\mathbf{I}_b$ and $\mathbf{I}_r$ with $|\mathbf{I}_b| = |\mathbf{I}_r|$ of a graph $G = (V, E)$, and imagine that a token is placed on each vertex in $\mathbf{I}_b$. Then, the SLIDING TOKEN problem is to determine if there exists a sequence $\langle \mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_\ell \rangle$ of independent sets of G such that (a) $\mathbf{I}_1 = \mathbf{I}_b$, $\mathbf{I}_\ell = \mathbf{I}_r$, and $|\mathbf{I}_b| = |\mathbf{I}_i|$ for all i, $1 \le i \le \ell$; and (b) for each i, $2 \le i \le \ell$, there is an edge $\{u, v\}$ in G such that $\mathbf{I}_{i-1} \setminus \mathbf{I}_i = \{u\}$ and $\mathbf{I}_i \setminus \mathbf{I}_{i-1} = \{v\}$. Figure 2 illustrates a sequence $\langle \mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_5 \rangle$ of independent sets which transforms $\mathbf{I}_b = \mathbf{I}_1$ into $\mathbf{I}_r = \mathbf{I}_5$. Hearn and Demaine proved that the SLIDING TOKEN problem is PSPACE-

[1] School of Information Science, Japan Advanced Institute of Science and Technology, Nomi, Ishikawa 923–1292, Japan
[a)] tyama@jaist.ac.jp
[b)] uehara@jaist.ac.jp

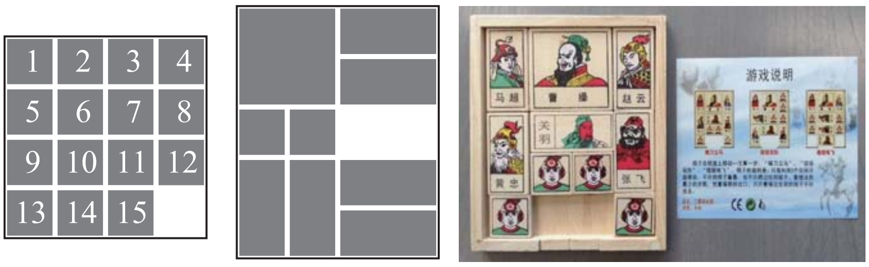

Fig. 1 The 15 puzzle, Dad's puzzle, and its Chinese variant.

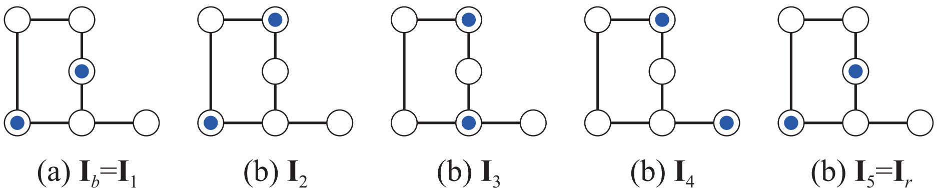
(a) $\mathbf{I}_b$=$\mathbf{I}_1$
(b) $\mathbf{I}_2$
(b) $\mathbf{I}_3$
(b) $\mathbf{I}_4$
(b) $\mathbf{I}_5$=$\mathbf{I}_r$

Fig. 2 A sequence $\langle \mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_5 \rangle$ of independent sets of the same graph, where the vertices in independent sets are depicted by small black circles (tokens).

complete for planar graphs.

For the SLIDING TOKEN problem, some polynomial time algorithms are investigated as follows: Linear time algorithms have been shown for cographs (also known as P_4-free graphs) [12] and trees [4]. Polynomial time algorithms are shown for bipartite permutation graphs [6], and claw-free graphs [2]. On the other hand, PSPACE-completeness is also shown for graphs of bounded treewidth [17], and planar graphs [10].

In this context, we investigate for finding a shortest sequence of the SLIDING TOKEN problem, which is called the SHORTEST SLIDING TOKEN problem. That is, our problem is formalized as follows:

Input: A graph $G = (V, E)$ and two independent sets $\mathbf{I}_b, \mathbf{I}_r$ with $|\mathbf{I}_b| = |\mathbf{I}_r|$.

Output: A *shortest* reconfiguration sequence $\mathbf{I}_b = \mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_\ell = \mathbf{I}_r$ such that $\mathbf{I}_i$ can be obtained from $\mathbf{I}_{i-1}$ by sliding exactly one token on a vertex $u \in \mathbf{I}_{i-1}$ to its adjacent vertex v along $\{u, v\} \in E$ for each i, $2 \leq i \leq \ell$.

We note that ℓ is not necessarily in polynomial of $|V|$; this is an issue how we formalize the problem, and if we do not know that ℓ is in polynomial or not. If the length k is given as a part of input, we may be able to decide if $\ell \leq k$ in polynomial time even if ℓ itself is not in polynomial. However, if we have to output the sequence itself, it cannot be solved in polynomial time if ℓ is not in polynomial.

In this paper, we will show that the SHORTEST SLIDING TOKEN problem is solvable in polynomial time for the following graph classes:

1.1 Proper interval graphs

We first prove that every proper interval graph with two independent sets $\mathbf{I}_b$ and $\mathbf{I}_r$ is a yes-instance if $|\mathbf{I}_b| = |\mathbf{I}_r|$. Furthermore, we can find the ordering of tokens to be slid in a shortest sequence in $O(n)$ time (implicitly), even though there exists an infinite family of independent sets on paths (and hence on proper interval graphs) for which any sequence requires $\Omega(n^2)$ length.

1.2 Trivially perfect graphs

We give an $O(n)$-time algorithm for trivially perfect graphs that actually finds a shortest sequence if such a sequence exists. In contrast to proper interval graphs, any shortest sequence is of length $O(n)$ for trivially perfect graphs. Note that trivially perfect graphs form a subclass of cographs, and hence its polynomial time solvability has been known [12].

1.3 Caterpillars

We finally give an $O(n^2)$-time algorithm for a caterpillar for the shortest sliding token problem. To make self-contained, we first show a linear time algorithm for decision problem that asks if two independent sets are reconfigurable or not. (We note that this problem can be solved in linear time for a tree [4].) For a yes-instance, we show an algorithm that finds a shortest sequence of token sliding between them.

We here remark that, since the problem is PSPACE-complete in general, an instance of the SLIDING TOKEN problem may require superpolynomial number of independent sets to transform. In such a case, tokens should make detours to avoid violating to be independent (as shown in Fig. 2). As we will see, caterpillars certainly require to make detours to transform, but it can be bounded by a polynomial.

As far as the authors know, this is the first polynomial time algorithm for the shortest sliding token problem for a graph class that requires detours.

2. Preliminaries

In this section, we introduce some basic terms and notations. In the SLIDING TOKEN problem, we may assume without loss of generality that graphs $G = (V, E)$ are simple and connected, and $|V| = n$ and $|E| = m$.

2.1 SLIDING TOKEN

For two independent sets $\mathbf{I}_i$ and $\mathbf{I}_j$ of the same cardinality in a graph $G = (V, E)$, if there exists exactly one edge $\{u, v\}$ in G such that $\mathbf{I}_i \setminus \mathbf{I}_j = \{u\}$ and $\mathbf{I}_j \setminus \mathbf{I}_i = \{v\}$, then we say that $\mathbf{I}_j$ can be obtained from $\mathbf{I}_i$ by *sliding* a token on the vertex $u \in \mathbf{I}_i$ to its adjacent vertex v along the edge $\{u, v\}$, and denote it by $\mathbf{I}_i \vdash \mathbf{I}_j$. We remark that the tokens are unlabeled, while the vertices in a graph are labeled.

A *reconfiguration sequence* between two independent sets $\mathbf{I}_1$ and $\mathbf{I}_\ell$ of G is a sequence $\langle \mathbf{I}_1, \mathbf{I}_2, \ldots, \mathbf{I}_\ell \rangle$ of independent sets of G such that $\mathbf{I}_{i-1} \vdash \mathbf{I}_i$ for $i = 2, 3, \ldots, \ell$. We denote by $\mathbf{I}_1 \vdash^* \mathbf{I}_\ell$ if there exists a reconfiguration sequence between them. We note that a reconfiguration sequence is *reversible*, that is, we have $\mathbf{I}_1 \vdash^* \mathbf{I}_\ell$ iff $\mathbf{I}_\ell \vdash^* \mathbf{I}_1$. Thus we say that two independent sets $\mathbf{I}_1$ and $\mathbf{I}_\ell$ are *reconfigurable* into each other if $\mathbf{I}_1 \vdash^* \mathbf{I}_\ell$. The *length* of a reconfiguration sequence $\mathcal{S}$ is defined as the number of independent sets contained in $\mathcal{S}$.

The SLIDING TOKEN problem is to determine if two given independent sets $\mathbf{I}_b$ and $\mathbf{I}_r$ of a graph G are reconfigurable into each other. We assume that $|\mathbf{I}_b| = |\mathbf{I}_r|$ without loss of generality; otherwise the answer is clearly "no." Note that the SLIDING TOKEN problem is a decision problem asking for the existence of a reconfiguration sequence between $\mathbf{I}_b$ and $\mathbf{I}_r$, and it does not ask an actual reconfiguration sequence. In this paper, we will consider the SHORTEST SLIDING TOKEN problem that computes the length of a shortest reconfiguration sequence between two independent sets. Note that the length of a reconfiguration sequence may not be in polynomial since the sequence may contain detours of tokens.

We always denote by $\mathbf{I}_b$ and $\mathbf{I}_r$ the initial and target independent sets of G, respectively, as an instance of the (SHORTEST) SLIDING TOKEN problem; we wish to slide tokens on the vertices in $\mathbf{I}_b$ to the vertices in $\mathbf{I}_r$. We sometimes call the vertices in $\mathbf{I}_b$ *blue*, and in $\mathbf{I}_r$ *red*; each vertex in $\mathbf{I}_b \cap \mathbf{I}_r$ is blue *and* red.

2.2 Target-assignment

We here give another notation of the SLIDING TOKEN problem, which is useful to explain our algorithm.

Let $\mathbf{I}_b = \{b_1, b_2, \ldots, b_k\}$ be an initial independent set of a graph G. For the sake of convenience, we label the tokens on the vertices in $\mathbf{I}_b$; let t_i be the token placed on b_i for each i, $1 \le i \le k$. Let $\mathcal{S}$ be a reconfiguration sequence between $\mathbf{I}_b$ and an independent set $\mathbf{I}$ of G. Then, for each token t_i, $1 \le i \le k$, we denote by $f_{\mathcal{S}}(t_i)$ the vertex in $\mathbf{I}$ on which the token t_i is placed via the reconfiguration sequence $\mathcal{S}$.

Let $\mathbf{I}_r$ be a target independent set of G, which is not necessarily reconfigurable from $\mathbf{I}_b$. Then, we call a mapping $g : \mathbf{I}_b \to \mathbf{I}_r$ a *target-assignment* between $\mathbf{I}_b$ and $\mathbf{I}_r$. The target-assignment g is said to be *proper* if there exists a reconfiguration sequence $\mathcal{S}$ such that $f_{\mathcal{S}}(t_i) = g(b_i)$ for all i, $1 \le i \le k$. Therefore, the SLIDING TOKEN problem can be seen as the problem of determining if there exists at least one proper target-assignment between $\mathbf{I}_b$ and $\mathbf{I}_r$.

2.3 Interval graphs and subclasses

The *neighborhood* of a vertex v in a graph $G = (V, E)$ is the set of all vertices adjacent to v, and denoted by $N(v) = \{u \in V \mid \{u, v\} \in E\}$. Let $N[v] = N(v) \cup \{v\}$. For any graph $G = (V, E)$, two vertices u and v are called *strong twins* if $N[u] = N[v]$, and *weak twins* if $N(u) = N(v)$. In our problem, strong twins have no meaning: when u and v are strong twins, only one of them can be used by a token. Therefore, we only consider the graphs without strong twins. (We have to take care about weak twins; see Section 5 for the details.)

A graph $G = (V, E)$ with $V = \{v_1, v_2, \ldots, v_n\}$ is an *interval graph* if there exists a set $\mathcal{I}$ of intervals $I_1, I_2, \ldots, I_n$ such that $\{v_i, v_j\} \in E$ if and only if $I_i \cap I_j \neq \emptyset$ for each i and j with $1 \le i, j \le n$.[*1] We call the set $\mathcal{I}$ of intervals an *interval representation* of the graph, and sometimes identify a vertex $v_i \in V$ with its corresponding interval $I_i \in \mathcal{I}$. We denote by $L(I)$ and $R(I)$ the left and right endpoints of an interval $I \in \mathcal{I}$, respectively. That is, we always have $L(I) \le R(I)$ for any interval $I = [L(I), R(I)]$.

To specify the bottleneck of the running time of our algorithms, we suppose that an interval graph $G = (V, E)$ is given as an input by its interval representation using $O(n)$ space. (If necessary, an interval representation of G can be found in $O(n + m)$ time [13].) Precisely, G is given by a string of length $2n$ over alphabets $\{L(I_1), L(I_2), \ldots, L(I_n), R(I_1), R(I_2), \ldots, R(I_n)\}$.

An interval graph is *proper* if it has an interval representation such that no interval properly contains another. The class of proper interval graphs is also known as the class of unit interval graphs [1]: an interval graph is *unit* if it has an interval representation such that every interval has unit length. Hereafter, we assume that each proper interval graph is given in the interval representation of intervals of unit length. In the context of the interval representation, an interval graph is proper iff $L(I_i) < L(I_j)$ if and only if $R(I_i) < R(I_j)$.

An interval graph is *trivially perfect* if it has an interval representation such that the relationship between any two intervals is either disjoint or inclusion. That is, for any two intervals I_i and I_j with $L(I_i) < L(I_j)$, we have either $L(I_i) < L_(I_j) < R(I_j) < R(I_i)$ or $L(I_i) < R(I_i) < L(I_j) < R(I_j)$.

A *caterpillar* $G = (V, E)$ is a tree (i.e., a connected acyclic graph) that consists of two subsets S and L of V as follows. The vertex set S induces a path $(s_1, \ldots, s_{n'})$ in G, and each vertex v in L has degree 1, and its unique neighbor is in S. We call the path $(s_1, \ldots, s_{n'})$ *spine*, and each vertex in L *leaf*. In this paper, without loss of generality, we assume that $n' \ge 2$, $\deg(s_1) \ge 2$, and $\deg(s_{n'}) \ge 2$. That is, the endpoints s_1 and $s_{n'}$ of the spine $(s_1, \ldots, s_{n'})$ should have at least one leaf. The class of caterpillars is a proper subset of the class of interval graphs.

3. Proper Interval Graphs

We first show that the answer of SLIDING TOKEN is always "yes" for proper interval graphs. We give a constructive proof of the claim, and it certainly finds a shortest sequence in linear time.

Theorem 1 For a (connected) proper interval graph $G = (V, E)$, any two independent sets $\mathbf{I}_b$ and $\mathbf{I}_r$ with $|\mathbf{I}_b| = |\mathbf{I}_r|$ satisfy $\mathbf{I}_b \vdash^* \mathbf{I}_r$. Moreover, the shortest reconfiguration sequence can be found in linear time.

We give an algorithm which actually finds a shortest reconfiguration sequence between any two independent sets $\mathbf{I}_b$ and $\mathbf{I}_r$. A proper interval graph $G = (V, E)$ has a unique interval representation (up to reversal), and we can assume that each interval is of unit length in the representation [5]. Therefore, by renumbering the vertices, we can fix an interval representation $\mathcal{I} =$

[*1] In this paper, a bold $\mathbf{I}$ denotes an "independent set," an italic I denotes an "interval," and calligraphy $\mathcal{I}$ denotes "a set of intervals."

$\{I_1, I_2, \ldots, I_n\}$ of G so that $L(I_i) < L(I_{i+1})$ (and $R(I_i) < R(I_{i+1})$) for each i, $1 \le i \le n-1$, and each interval $I_i \in \mathcal{I}$ corresponds to the vertex $v_i \in V$.

Let $\mathbf{I}_b = \{b_1, b_2, \ldots, b_k\}$ and $\mathbf{I}_r = \{r_1, r_2, \ldots, r_k\}$ be any given initial and target independent sets of G, respectively. Without loss of generality, we assume that the blue vertices $b_1, b_2, \ldots, b_k$ and the red vertices $r_1, r_2, \ldots, r_k$ are labeled from left to right according to the interval representation $\mathcal{I}$ of G, that is, $L(b_i) < L(b_j)$ and $L(r_i) < L(r_j)$ if $i < j$. Then, we define a target-assignment $g : \mathbf{I}_b \to \mathbf{I}_r$ as follows: for each blue vertex $b_i \in \mathbf{I}_b$

$$g(b_i) = r_i. \tag{1}$$

To prove Theorem 1, it suffices to show that g is proper, and each token takes no detours.

3.1 String representation

By traversing the interval representation $\mathcal{I}$ of a connected proper interval graph G from left to right, we can obtain a string $S = s_1 s_2 \cdots s_{2k}$ which is a superstring of both $b_1 b_2 \cdots b_k$ and $r_1 r_2 \cdots r_k$, that is, each letter s_i in S is one of the vertices in $\mathbf{I}_b \cup \mathbf{I}_r$ and s_i appears in S before s_j if $L(s_i) < L(s_j)$. We assume without loss of generality that $s_1 = b_1$. If a vertex is contained in both $\mathbf{I}_b$ and $\mathbf{I}_r$, as b_i and r_j, then we define that it appears as $b_i r_j$ in S. Then, for each i, $1 \le i \le 2k$, we define the *height $h(i)$ at i* by the number of blue vertices appeared in the substring $s_1 s_2 \cdots s_i$ minus the number of red vertices appeared in $s_1 s_2 \cdots s_i$ (we define $h(0) = 0$). Then $h(i)$ is recursively computed by

$$h(i) = \begin{cases} 0 & \text{if } i = 0; \\ h(i-1) + 1 & \text{if } s_i \text{ is blue;} \\ h(i-1) - 1 & \text{if } s_i \text{ is red.} \end{cases} \tag{2}$$

Since $|\mathbf{I}_b| = |\mathbf{I}_r|$, $h(2k) = 0$ for any string S.

Using the notion of height, we split the string S into substrings $S_1, S_2, \ldots, S_h$ at every point of height 0, i.e., in each substring $S_j = s_{2p+1} s_{2p+2} \cdots s_{2q}$, we have $h(2q) = 0$ and $h(i) \neq 0$ for all i, $2p+1 \le i \le 2q-1$. Then, the substrings $S_1, S_2, \ldots, S_h$ form a partition of S, and each substring S_j contains the same number of blue and red tokens. We call such a partition the *partition of S at height* 0.

Lemma 2 Let $S_j = s_{2p+1} s_{2p+2} \cdots s_{2q}$ be a substring in the partition of the string S at height 0. Then, (a) the blue vertices $b_{p+1}, b_{p+2}, \ldots, b_q$ appear in S_j, and their corresponding red vertices $r_{p+1}, r_{p+2}, \ldots, r_q$ appear in S_j; (b) if S_j starts with the blue vertex b_{p+1}, each blue vertex b_i $(p+1 \le i \le q)$ appears in S_j before its corresponding red vertex r_i; and (c) if S_j starts with the red vertex r_{p+1}, each blue vertex b_i $(p+1 \le i \le q)$ appears in S_j after its corresponding red vertex r_i.

Proof. By the definitions, the claim (a) clearly holds. We thus show that the claim (b) holds (the claim (c) is symmetric). Since $h(2p) = 0$ and S_j starts with a blue vertex, we have $h(2p+1) = 1 > 0$. We now suppose for a contradiction that there exists a blue vertex $s_x = b_{i'}$ which appears in S_j after its corresponding red vertex $s_y = r_{i'}$. Then $y < x$. We assume that y is the minimum index among such blue vertices in S_j. Then, in the substring $s_1 s_2 \cdots s_y$ of S, there are exactly i' red vertices. On the other hand, since $y < x$, the substring $s_1 s_2 \cdots s_y$ contains at most $i'-1$ blue vertices. Therefore, by the definition of height, we have $h(y) < 0$. Since $h(2p+1) = 1 > 0$ and $h(y) < 0$, by Eq. (2) there exists an index z with $h(z) = 0$ and $2p < z < y$. This contradicts the fact that S_j is a substring in the partition of S at height 0. ∎

3.2 Algorithm

Since all intervals in $\mathcal{I}$ have unit length, the following proposition clearly holds.

Proposition 3 For two vertices v_i and v_j in G such that $i < j$, there is a path P in G which passes through only intervals (vertices) contained in $[L(I_i), R(I_j)]$. Furthermore, if $I_{i'} \cap I_i = \emptyset$ for some index i' with $i' < i$, no vertex in $v_1, v_2, \ldots, v_{i'}$ is adjacent to any vertex in P. If $I_j \cap I_{j'} = \emptyset$ for some index j' with $j < j'$, no vertex in $v_{j'}, v_{j'+1}, \ldots, v_n$ is adjacent to any vertex in P.

Let S be the string of length $2k$ obtained from two given independent sets $\mathbf{I}_b$ and $\mathbf{I}_r$ of a proper interval graph G, where $k = |\mathbf{I}_b| = |\mathbf{I}_r|$. Let $S_1, S_2, \ldots, S_h$ be the partition of S at height 0. The following lemma shows that the tokens in each substring S_j can always reach their corresponding red vertices. (We sometimes denote simply by S_j the set of all vertices appeared in the substring S_j, $1 \le j \le h$.)

Lemma 4 Let $S_j = s_{2p+1} s_{2p+2} \cdots s_{2q}$ be a substring in the partition of S at height 0. Then, there exists a reconfiguration sequence between $\mathbf{I}_b \cap S_j$ and $\mathbf{I}_r \cap S_j$ such that tokens are slid along edges only in the subgraph of G induced by the vertices contained in $[L(s_{2p+1}), R(s_{2q})]$.

Proof. We first consider the case where S_j starts with the blue vertex b_{p+1}, that is, $s_{2p+1} = b_{p+1}$. Then, by Lemma 2(b) each blue vertex b_i $(p+1 \le i \le q)$ appears in S_j before the corresponding red vertex r_i. Therefore, we know that $s_{2q} = r_q$, and hence it is red. Suppose that $s_x = b_q$, then all vertices appeared in $s_{x+1} s_{x+2} \cdots s_{2q}$ are red. Intuitively, we slide the tokens $t_q, t_{q-1}, \ldots, t_{p+1}$ from left to right in this order.

We first claim that the token t_q can be slid from b_q $(= s_x)$ to r_q $(= s_{2q})$. By Proposition 3 there is a path P between b_q and r_q which passes through only intervals contained in $[L(b_q), R(r_q)]$. Since $\mathbf{I}_b$ is an independent set of G, the vertex b_q is not adjacent to any other vertices $b_{p+1}, b_{p+2}, \ldots, b_{q-1}$ in $\mathbf{I}_b \cap S_j$. Since $L(b_{p+1}) < L(b_{p+2}) < \cdots < L(b_{q-1}) < L(b_q)$, by Proposition 3 all vertices in P are not adjacent to any of tokens $t_{p+1}, t_{p+2}, \ldots, t_{q-1}$ that are now placed on $b_{p+1}, b_{p+2}, \ldots, b_{q-1}$, respectively. Therefore, we can slide the token t_q from b_q to r_q. We fix the token t_q on $r_q = s_{2q}$, and will not move it anymore.

We then slide the next token t_{q-1} on b_{q-1} to r_{q-1} along a path P' which passes through only intervals contained in $[L(b_{q-1}), R(r_{q-1})]$. Since $\mathbf{I}_r$ is an independent set of G, the corresponding red vertex r_{q-1} is not adjacent to r_q on which the token t_q is now placed. Recall that $L(r_{q-1}) < L(r_q)$, and hence by Proposition 3, r_q is not adjacent to any vertex in P'. Similarly as above, the tokens $t_{p+1}, t_{p+2}, \ldots, t_{q-2}$ are not adjacent to any vertex in P'. Therefore, we can slide the token t_{q-1} from b_{q-1} to r_{q-1}.

Repeat this process until the token t_{p+1} on b_{p+1} is slid to r_{p+1}. In this way, there is a reconfiguration sequence between $\mathbf{I}_b \cap S_j$ and $\mathbf{I}_r \cap S_j$ such that tokens are slid along edges only in the subgraph of G induced by the vertices contained in $[L(b_{p+1}), R(r_q)]$.

The symmetric arguments prove the case where S_j starts with the red vertex r_{p+1}. Note that, in this case, we slide the tokens $t_{p+1}, t_{p+2}, \ldots, t_q$ from right to left in this order. ∎

Proof of Theorem 1. We now give an algorithm for sliding all tokens on the vertices in $\mathbf{I}_b$ to the vertices in $\mathbf{I}_r$. Recall that $S_1, S_2, \ldots, S_h$ are the substrings in the partition of S at height 0. Intuitively, the algorithm repeatedly picks up one substring S_j, and slides all tokens in $\mathbf{I}_b \cap S_j$ to $\mathbf{I}_r \cap S_j$. By Lemma 4 it works locally in each substring S_j, but a token in S_j may be adjacent to another token in S_{j-1} or S_{j+1} at the boundary of the substrings. To avoid this, we define a partial order over the substrings $S_1, S_2, \ldots, S_h$.

Consider any two consecutive substrings S_j and S_{j+1}, and let $S_j = s_{2p+1}s_{2p+2}\cdots s_{2q}$. Then, the first letter of S_{j+1} is s_{2q+1}. We first consider the case where both s_{2q} and s_{2q+1} have the same color. Then, since s_{2q} and s_{2q+1} are in the same independent set, they are not adjacent on G. Therefore, by Proposition 3 and Lemma 4, we can deal with S_j and S_{j+1} independently. In this case, we do not define the ordering between S_j and S_{j+1}. We then consider the case where s_{2q} and s_{2q+1} have different colors. Suppose that s_{2q} is blue and s_{2q+1} is red; then we have $s_{2q} = b_q$ and $s_{2q+1} = r_{q+1}$. By Lemma 4 the token t_q on s_{2q} is slid to left, and the token t_{q+1} will reach r_{q+1} from right. Therefore, the algorithm has to deal with S_j before S_{j+1}. Note that, after sliding all tokens $t_{p+1}, t_{p+2}, \ldots, t_q$ in S_j, they are on the red vertices $r_{p+1}, r_{p+2}, \ldots, r_q$, respectively, and hence the tokens in S_{j+1} are not adjacent to any of them. By the symmetric argument, if s_{2q} is red and s_{2q+1} is blue, S_{j+1} should be dealt with before S_j.

Such an ordering is defined only for two consecutive substrings S_j and S_{j+1} with $1 \leq j \leq h-1$. Therefore, the partial order over the substrings $S_1, S_2, \ldots, S_h$ is acyclic, and hence there exists a total order which is consistent with the partial order. The algorithm certainly slides all tokens from $\mathbf{I}_b$ to $\mathbf{I}_r$ according to the total order. Therefore, the target-assignment g defined in Eq. (1) is proper, and hence $\mathbf{I}_b \vdash^* \mathbf{I}_r$. Thus there always exists a reconfiguration sequence between two independent sets $\mathbf{I}_b$ and $\mathbf{I}_r$ of a connected proper interval graph G.

We now discuss the length of reconfiguration sequences between $\mathbf{I}_b$ and $\mathbf{I}_r$, together with the running time of our algorithm.

Proposition 5 For two given independent sets $\mathbf{I}_b$ and $\mathbf{I}_r$ of a proper interval graph G with n vertices, (1) the ordering of tokens to be slid in a shortest reconfiguration sequence between them can be computed in $O(n)$ time and $O(n)$ space; and (2) a shortest reconfiguration sequence between them can be output in $O(n^2)$ time and $O(n)$ space.

Proof. We first modify our algorithm so that it finds a shortest reconfiguration sequence between $\mathbf{I}_b$ and $\mathbf{I}_r$. To do that, it suffices to slide each token t_i, $1 \leq i \leq k$, from the blue vertex b_i to its corresponding red vertex r_i along the shortest path between b_i and r_j. We may assume without loss of generality that $L(b_i) < L(r_i)$, that is, the token t_i will be slid from left to right. Then, for the interval b_i, we choose an interval $I_j \in \mathcal{I}$ such that $b_i \cap I_j \neq \emptyset$ and $L(I_j)$ is the maximum among all $I_{j'} \in \mathcal{I}$. If $L(r_i) \leq L(I_j)$, we can slide t_i from b_i to r_i directly; otherwise we slide t_i to the vertex I_j, and repeat.

We then prove the claim (1). If we simply want to compute the ordering of tokens to be slid in a shortest reconfiguration sequence, it suffices to compute the partial order over the substrings $S_1, S_2, \ldots, S_h$ in the partition of the string S at height 0. It is not difficult to implement our algorithm in Section 3.2 to run in $O(n)$ time and $O(n)$ space.

We finally prove the claim (2). Remember that each token t_i, $1 \leq i \leq k$, is slid along the shortest path from b_i to r_i. Furthermore, once the token t_i reaches r_i, it is not slid anymore. Therefore, the length of a shortest reconfiguration sequence between $\mathbf{I}_b$ and $\mathbf{I}_r$ is given by the sum of all lengths of the shortest paths between b_i and r_i. It is clear that this sum is $O(kn) = O(n^2)$. We output only the shortest paths between b_i and r_i, together with the ordering of the tokens to be slid. ∎

This proposition completes the proof of Theorem 1. ∎

It is remarkable that there exists an infinite family of instances for which any reconfiguration sequence requires $\Omega(n^2)$ length. Simple example is: G is a path $(v_1, v_2, \ldots, v_{8k})$ of length $n = 8k$ for any positive integer k, $\mathbf{I}_b = \{v_1, v_3, v_5, \ldots, v_{2k-1}\}$, and $\mathbf{I}_r = \{v_{6k+2}, v_{6k+4}, \ldots, v_{8k}\}$. In this instance, each token t_i must be slid $\Theta(n)$ times, and hence it requires $\Theta(n^2)$ time to output them all. A path belongs to proper interval graphs and caterpillars.

4. Trivially perfect graphs

The main result of this section is the following theorem.

Theorem 6 The SLIDING TOKEN problem for a trivially perfect graph $G = (V, E)$ can be solved in $O(n)$ time and $O(n)$ space. Furthermore, one can find a shortest reconfiguration sequence between two given independent sets $\mathbf{I}_b$ and $\mathbf{I}_r$ in $O(n)$ time and $O(n)$ space if there exists.

We explicitly give such an algorithm as a proof of Theorem 6. Note that there are no-instances for trivially perfect graphs. However, for trivially perfect graphs, we construct a proper target-assignment between $\mathbf{I}_b$ and $\mathbf{I}_r$ efficiently if it exists.

4.1 MPQ-tree for trivially perfect graphs

The MPQ-tree of an interval graph G is a kind of decomposition tree, developed by Korte and Möhring [13], which represents the set of all feasible interval representations of G. For the notion of MPQ-trees, the following theorem is known:

Theorem 7 ([13]) For any interval graph $G = (V, E)$, its corresponding MPQ-tree can be constructed in $O(n + m)$ time.

We here give a simplified definition of MPQ-tree only for trivially perfect graphs. Let $G = (V, E)$ be a trivially perfect graph. Then, the MPQ-*tree* $\mathcal{T}$ of G is a rooted tree such that each node, called a P-*node*, in $\mathcal{T}$ is associated with a non-empty set of vertices in G such that (a) each vertex $v \in V$ appears in exactly one P-node in $\mathcal{T}$, and (b) if a vertex $v_i \in V$ is in an ancestor node of another node that contains $v_j \in V$, then $L(I_i) \leq L(I_j) \leq R(I_j) \leq R(I_i)$ in any interval representation of G, where v_i and v_j correspond to the intervals I_i and I_j, respectively (see Fig. 3 as an example). By (b), the ancestor/descendant relationship on $\mathcal{T}$ corresponds to the inclusion relationship in the interval representation of G. Thus, $N[v_j] \subseteq N[v_i]$ iff v_i is in an ancestor of the node containing v_j in the MPQ-tree.

Let $\mathcal{T}$ be the (unique) MPQ-tree of a (connected) trivially perfect graph $G = (V, E)$. For two vertices u and w in G, we denote

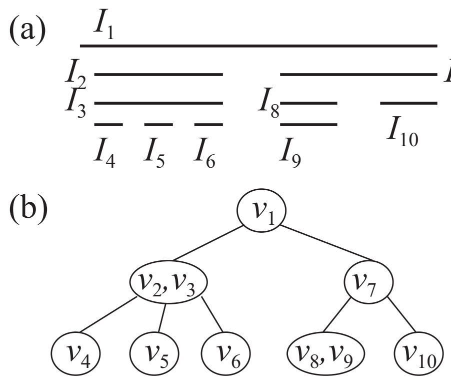

Fig. 3 (a) A trivially perfect graph in an interval representation, and (b) its MPQ-tree.

by $\mathsf{LCA}(u, w)$ the least common ancestor in $\mathcal{T}$ for the nodes containing u and w. By (a), the node $\mathsf{LCA}(u, w)$ can be uniquely defined.

4.2 Basic properties and key lemma

Any interval representation of a trivially perfect graph has only disjoint or inclusion relationship. This fact implies the following:

Observation 8 Every pair of vertices u and w in a trivially perfect graph G has a path of length at most two via a vertex in $\mathsf{LCA}(u, w)$.

Proof. Omitted. ∎

Let $\mathsf{LCA}^*(u, w)$ be the set of vertices in V appearing in the P-nodes on the unique path from $\mathsf{LCA}(u, w)$ to the root of the MPQ-tree. By the definition of MPQ-tree, we clearly have the following observation.

Observation 9 Consider an arbitrary reconfiguration sequence $\mathcal{S}$ which slides a token t_i from $b_i \in \mathbf{I}_b$ to some vertex r_i. Then, t_i must pass through at least one vertex in $\mathsf{LCA}^*(b_i, r_i)$, that is, there exists at least one independent set $\mathbf{I}'$ in $\mathcal{S}$ such that $\mathbf{I}' \cap \mathsf{LCA}^*(b_i, r_i) \neq \emptyset$.

We are now ready to give the key lemma for trivially perfect graphs.

Lemma 10 Let $g : \mathbf{I}_b \to \mathbf{I}_r$ be a target-assignment between $\mathbf{I}_b$ and $\mathbf{I}_r$. Then, g is proper iff the nodes $\mathsf{LCA}(b_i, g(b_i))$ and $\mathsf{LCA}(b_j, g(b_j))$ are not in the ancestor/descendant relationship on $\mathcal{T}$ for every pair of vertices $b_i, b_j \in \mathbf{I}_b$.

Proof. Omitted. ∎

4.3 Algorithm and its correctness

We now describe our linear-time algorithm for a trivially perfect graph. Let $\mathcal{T}$ be the MPQ-tree of a connected trivially perfect graph $G = (V, E)$. Let $\mathbf{I}_b = \{b_1, b_2, \ldots, b_k\}$ and $\mathbf{I}_r = \{r_1, r_2, \ldots, r_k\}$ be given initial and target independent sets of G, respectively. Then, we determine if $\mathbf{I}_b \vdash^* \mathbf{I}_r$ as follows: (A) construct some particular target-assignment g^* between $\mathbf{I}_b$ and $\mathbf{I}_r$; and (B) check if g^* is proper or not using Lemma 10. We will show later in Lemma 12 that it suffices to check only g^* in order to determine if $\mathbf{I}_b \vdash^* \mathbf{I}_r$ or not. Indeed, our linear-time algorithm executes (A) and (B) above at the same time, in the bottom-up manner based on $\mathcal{T}$.

4.3.1 Description of the algorithm

Since the vertex-set associated to each P-node in $\mathcal{T}$ induces a clique, for any independent set $\mathbf{I}$ of G, each P-node contains at most one vertex in $\mathbf{I}$, and hence contains at most one token. We put a "blue token" for each P-node containing a blue vertex in $\mathbf{I}_b$, and also put a "red token" for each P-node containing a red vertex in $\mathbf{I}_r$. Note that a P-node may contain a pair of blue and red tokens. Our algorithm lifts up the tokens from the leaves to the root of $\mathcal{T}$, and if a blue token b meets a red token r at their least common ancestor $\mathsf{LCA}(b, r)$ in $\mathcal{T}$, then we replace them by a single "green token." This corresponds to setting $g^*(b) = r$. Precisely, at initialization step, the algorithm first collects all leaves of $\mathcal{T}$ in a queue, called *frontier*. The algorithm marks the nodes in the frontier, and lifts up each token to its parent P-node. Each P-node P is put into the frontier if its all children are marked, and then, all children of P are removed from the frontier after the following procedure at P:

Case (1): P contains at most one token: the algorithm has nothing to do.

Case (2): P contains only one pair of blue token b and red token r: the algorithm replaces them by a single green token, and let $g^*(b) = r$.

Case (3): P contains only green tokens: the algorithm replaces them by a single green token.

Case (4): P contains two or more blue tokens, or two or more red tokens: the algorithm outputs "no" and halts (i.e., $\mathbf{I}_b \nvdash^* \mathbf{I}_r$).

Case (4): P contains at least one green token and at least one blue or red token: the algorithm outputs "no" and halts (i.e., $\mathbf{I}_b \nvdash^* \mathbf{I}_r$).

Repeating this process, and the algorithm outputs "yes" if and only when the frontier contains only the root P-node r of $\mathcal{T}$ which is in one of Cases (1)–(3) above.

4.3.2 Correctness of the algorithm

It is not difficult to implement our algorithm to run in $O(n)$ time and $O(n)$ space. Therefore, we here prove the correctness of the algorithm. We first show that $\mathbf{I}_b \vdash^* \mathbf{I}_r$ if the algorithm outputs "yes." In this case, the algorithm is in Cases (1), (2), or (3) at each P-nodes in $\mathcal{T}$ (including the root r). Then, the target-assignment g^* has been (completely) constructed: for each blue vertex $b_i \in \mathbf{I}_b$, $g^*(b_i)$ is the red vertex in $\mathbf{I}_r$ such that $\mathsf{LCA}(b_i, v_{i'})$ has the minimum height in $\mathcal{T}$ among all vertices $v_{i'} \in \mathbf{I}_r$. Then, we have the following lemma.

Lemma 11 If the algorithm outputs "yes," then $\mathbf{I}_b \vdash^* \mathbf{I}_r$.

Proof. Omitted. ∎

The following lemma completes the correctness proof of our algorithm.

Lemma 12 If the algorithm outputs "no," then $\mathbf{I}_b \nvdash^* \mathbf{I}_r$.

Proof. Omitted. ∎

4.4 Shortest reconfiguration sequence

To complete the proof of Theorem 6, we finally show that our algorithm in Section 4.3 can be modified so that it actually finds a shortest reconfiguration sequence between $\mathbf{I}_b$ and $\mathbf{I}_r$. Roughly, when the algorithm put a green token at a vertex w for a blue token b coming from a vertex u and a red token r coming from a vertex v, we can slide a token on u to v through w. Therefore, we need no detour, and the number of slides is at most $2k$, which completes the proof of Theorem 6.

5. Caterpillars

The main result of this section is the following theorem.

Theorem 13 The SLIDING TOKEN problem for a connected caterpillar $G = (V, E)$ and two independent sets $\mathbf{I}_b$ and $\mathbf{I}_r$ of G can be solved in $O(n)$ time and $O(n)$ space. Moreover, for a yes-instance, a shortest reconfiguration sequence between them can be output in $O(n^2)$ time and $O(n)$ space.

Let $G = (S \cup L, E)$ be a caterpillar with spine S which induces the path $(s_1, \ldots, s_{n'})$, and leaf set L. We assume that $n' \geq 2$, $\deg(s_1) \geq 2$, and $\deg(s_{n'}) \geq 2$. First we show that we can assume that each spine vertex has at most one leaf without loss of generality.

Lemma 14 For any given caterpillar $G = (S \cup L, E)$ and two independent sets $\mathbf{I}_b$ and $\mathbf{I}_r$ on G, there is a linear time reduction from them to another caterpillar $G' = (S' \cup L', E')$ and two independent sets $\mathbf{I}'_b$ and $\mathbf{I}'_r$ such that (1) G, $\mathbf{I}_b$, and $\mathbf{I}_r$ are a yes-instance of the SLIDING TOKEN problem iff G', $\mathbf{I}'_b$, and $\mathbf{I}_r$ are a yes-instance of the SLIDING TOKEN problem, (2) the maximum degree of G' is at most 3, and (3) $\deg(s_1) = \deg(s_{n'}) = 2$, where $n' = |S'|$. In other words, the SLIDING TOKEN problem on a caterpillar is sufficient to consider only caterpillars of maximum degree 3.

Proof. On G, let s_i be any vertex in S with $\deg(s_i) > 3$. Then there exist at least two leaves ℓ_i and ℓ'_i attached to s_i (note that they are weak twins). Now we consider the case that two tokens in $\mathbf{I}_b$ are on ℓ_i and ℓ'_i. Then, we cannot slide these two tokens at all, and any other token cannot pass through s_i since it is blocked by them. If $\mathbf{I}_r$ contains these two tokens also, we can split the problem into two subproblems by removing s_i and its leaves from G, and solve it separately. Otherwise, the answer is "no" (remind that the problem is reversible; that is, if tokens cannot be slid, there are no other tokens which slide into the situation). Therefore, if at least two tokens are placed on the leaves of a vertex of the original graph, we can reduce the case in linear time. Thus we assume that every spine vertex with its leaves contains at most one token in $\mathbf{I}_b$ and $\mathbf{I}_r$, respectively. Then, by the same reason, we can remove all leaves but one of each spine vertex. More precisely, regardless whether $\mathbf{I}_b \vdash^* \mathbf{I}_r$ or $\mathbf{I}_b \nvdash^* \mathbf{I}_r$, at most one leaf for each spine vertex is used for the transitions. Therefore, we can remove all other useless leaves but one from each spine vertex. Especially, removing all useless leaves, we have $\deg(s_1) = \deg(s_{n'}) = 2$. ∎

Hereafter, we only consider the caterpillars stated in Lemma 14. That is, for any given caterpillar $G = (S \cup L, E)$ with spine $(s_1, \ldots, s_{n'})$, we assume that $\deg(s_1) = \deg(s_{n'}) = 2$ and $2 \leq \deg(s_i) \leq 3$ for each $1 < i < n'$. Then, we denote the unique leaf of s_i by ℓ_i if it exists.

We here introduce a key notion of the problem on these caterpillars that is named *locked* path. Let G and $\mathbf{I}$ be a caterpillar and an independent set of G, respectively. A path $P = (p_1, p_2, \ldots, p_k)$ on G is *locked* by $\mathbf{I}$ iff (a) k is odd and greater than 2, (b) $\mathbf{I} \cap P = \{p_1, p_3, p_5, \ldots, p_k\}$, (c)$\deg(p_1) = \deg(p_k) = 1$ (in other words, they are leaves), and $\deg(p_3) = \deg(p_5) = \cdots = \deg(p_{k-2}) = 2$. This notion is simplified version of a *locked* tree used in [4]. Using the discussion in [4], we obtain the condition for the immovable independent set on a caterpillar:

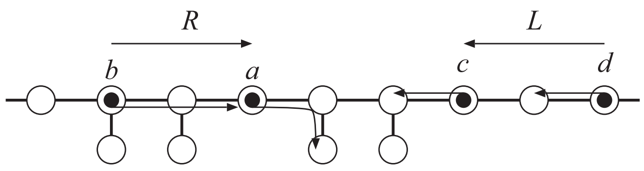

Fig. 4 The most right R token a has to precede the most left L token c.

Theorem 15 ([4]) Let G and $\mathbf{I}$ be a caterpillar and an independent set of G, respectively. Then we cannot slide any token in $\mathbf{I}$ on G at all if and only if there exist a set of locked paths $P_1, \ldots, P_h$ for some h such that $\mathbf{I}$ is a union of them.

The proof can be found in [4], and omitted here. Intuitively, for any caterpillar G and its independent set $\mathbf{I}$, if $\mathbf{I}$ contains a locked path P, we cannot slide any token through the vertices in P. Therefore, P splits G into two subgraphs, and we obtain two completely separated subproblems. Therefore, we obtain the following lemma:

Lemma 16 For any given caterpillar $G = (S \cup L, E)$ and two independent sets $\mathbf{I}_b$ and $\mathbf{I}_r$ on G, there is a linear time reduction from them to another caterpillar $G' = (S' \cup L', E')$ and two independent sets $\mathbf{I}'_b$ and $\mathbf{I}'_r$ such that (1) G, $\mathbf{I}_b$, and $\mathbf{I}_r$ are a yes-instance of the SLIDING TOKEN problem if and only if G', $\mathbf{I}'_b$, and $\mathbf{I}_r$ are a yes-instance of the SLIDING TOKEN problem, and (2) both of $\mathbf{I}'_b$ and $\mathbf{I}'_r$ contain no locked path.

Proof. Omitted. ∎

Hereafter, without loss of generality, we assume that the caterpillar G with two independent sets $\mathbf{I}_b$ and $\mathbf{I}_r$ satisfies the conditions in Lemmas 14 and 16. That is, each spine vertex s_i has at most one leaf ℓ_i, s_1 and $s_{n'}$ have one leaf ℓ_1 and $\ell_{n'}$, respectively, both of $\mathbf{I}_b$ and $\mathbf{I}_r$ contain no locked path, and $|\mathbf{I}_b| = |\mathbf{I}_r|$. By the result in [4], this is a yes-instance. Thus, it is sufficient to show an $O(n^2)$ time algorithm that computes a shortest reconfiguration sequence between $\mathbf{I}_b$ and $\mathbf{I}_r$.

It is clear that each pair (s_i, ℓ_i) can have at most one token. Therefore, without loss of generality, we can assume that the blue vertices $b_1, b_2, \ldots, b_k$ in $\mathbf{I}_b$ (and the red vertices $r_1, r_2, \ldots, r_k$) are labeled from left to right according to the order (s_1, ℓ_1), (s_2, ℓ_2), $\ldots$, $(s_{n'}, \ell_{n'})$ of G; that is, $L(b_i) < L(b_j)$ if $i < j$. Then, we define a target-assignment $g : \mathbf{I}_b \to \mathbf{I}_r$, as $g(b_i) = r_i$ for each blue vertex $b_i \in \mathbf{I}_b$. To prove Theorem 13, we show that we can slide tokens with fewest detours in case analysis.

Now we introduce *direction* of a token t denoted by $dir(t)$ as follows: when t moves from $v_i \in \{s_i, \ell_i\}$ in $\mathbf{I}_b$ to $v_j \in \{s_j, \ell_j\}$ in $\mathbf{I}_r$ with $i < j$, the direction of t is said to be R and denoted by $dir(t) = R$. If $i > j$, it is said to be L and denoted by $dir(t) = L$. If $i = j$, the direction of t is said to be C and denoted by $dir(t) = C$.

We first consider a simple case: all directions are either R or L. In this case, we can use the same idea appearing in the algorithm for a proper interval graph in Section 3. We can introduce a partial order over the tokens, and move them straightforwardly using the same idea in Section 3.2. Intuitively, a sequence of R tokens are moved from left to right, and a sequence of L tokens are moved from right to left, and we can define a partial order over the sequences of different directions. The only additional considerable case is shown in Fig. 4. That is, when the token a moves to ℓ_i from left and the other token c moves to s_{i+1} from right, a should precede c. It is not difficult to see that this (and its symmetric case) is the only exception than the algorithm in Section 3.2 when all tokens move to right or left. In other words, in this case, need detour is required.

We next suppose that $\mathbf{I}_b$ (and hence $\mathbf{I}_r$) contains some token t with $dir(t) = C$. In other words, t is put on s_i or ℓ_i for some i in both of $\mathbf{I}_b$ and $\mathbf{I}_r$. We have five cases. Here we show one case, and the other simpler four cases are omitted.

We assume that t is put on s_i in $\mathbf{I}_b$ and $\mathbf{I}_r$, and ℓ_i does not exist. By assumption, $1 < s < n'$ (since ℓ_1 and $\ell_{n'}$ exist). Without loss of generality, we suppose t is the leftmost spine with the condition. We first observe that $|\mathbf{I}_b \cap \{s_{i-1}, \ell_{i-1}, s_{i+1}, \ell_{i+1}\}|$ is at most 1. Clearly, we have no token on s_{i-1} and s_{i+1}. When we have two tokens on ℓ_{i-1} and ℓ_{i+1}, the path $(\ell_{i-1}, s_{i-1}, s_i, s_{i+1}, \ell_{i+1})$ is a locked path, which contradicts the assumption. We also have $|\mathbf{I}_r \cap \{s_{i-1}, \ell_{i-1}, s_{i+1}, \ell_{i+1}\}| \leq 1$ by the same argument.

Now we consider the most serious case since the other cases are simpler and easier. The most serious case is that $\mathbf{I}_b$ contains ℓ_{i-1} and $\mathbf{I}_r$ contains ℓ_{i+1}. Since any token cannot bypass the other, $\mathbf{I}_b$ contains an L token on ℓ_{i-1}, and $\mathbf{I}_r$ contains an L token on ℓ_{i+1}. In this case, by the L token on ℓ_{i-1}, first, t should make a detour to right, and by the L token in $\mathbf{I}_r$, t next should make a detour to left twice after the first detour. This three slides should not be avoided, and this ordering of three slides cannot be violated. Therefore, t itself should slide at least four times to return to the original position, and t can done it in four slides. During this slides, since t is the leftmost spine with this condition, the tokens on $s_1, \ell_1, s_2, \ell_2, \ldots, s_{i-1}, \ell_{i-1}$ do not make any detours. Thus we focus on the tokens on $s_{i+1}, \ell_{i+1}, \ldots$. Let t' be the token that should be on ℓ_{i+1} in $\mathbf{I}_r$. Since t is on s_i, t' is not on $\{s_{i+1}, \ell_{i+1}\}$. If t' is on one of $\ell_{i+2}, s_{i+3}, \ell_{i+3}, s_{i+4}, \ldots$ in $\mathbf{I}_b$, we have nothing to do; just make a detour for only t. The problem occurs when t' is on s_{i+2} in $\mathbf{I}_b$. If there exists ℓ_{i+2}, we first slide t' to it; this detour for t' is unavoidable. If ℓ_{i+2} does not exist, we have to slide t' to s_{i+3} before slide of t. This can be done immediately except the similar situation that the only considerable case is that we have another L or S token t'' on s_{i+3}. We can repeat this analysis and confirm that each detour is unavoidable. Since G with $\mathbf{I}_b$ and $\mathbf{I}_r$ contains no locked path, this process will halts. Therefore, traversing this process, we can construct the shortest reconfiguration sequence.

Proof of Theorem 13. For a given independent set $\mathbf{I}_b$ on a caterpillar $G = (V, E)$, we can check if each vertex is a part of locked path in $O(n)$ time. Thus, we first check twice for $(G, \mathbf{I}_b)$ and $(G, \mathbf{I}_r)$ in $O(n)$ time, and check if the sets of locked paths coincide with each other. If not, the algorithm outputs "no". We assume that they coincide. Then the algorithm splits the caterpillar G into subgraphs $G_1, G_2, \ldots, G_h$ by removing all locked paths. For each subgraph $G_1, \ldots, G_h$, the algorithm next checks if each subgraph contains the same number of tokens from $\mathbf{I}_b$ and $\mathbf{I}_r$. If they do not coincide, the algorithm outputs "no."After this process, we have a yes-instance. The correctness of the algorithm so far follows from Theorem 15 with results in [4] immediately. It is also easy to implement the algorithm to run in $O(n)$ time and space.

It is not difficult to modify the algorithm to output a shortest sequence based on the previous case analysis. For each token, the number of detours made by the token is bounded above by $O(n)$, the number of slide of the token itself is also bounded above by $O(n)$, and the computation for the token can be done in $O(n)$ time. Therefore, the algorithm runs in $O(n^2)$ time, and the length of the sequence is $O(n^2)$. ■

6. Concluding Remarks

In this paper, we showed that the SHORTEST SLIDING TOKEN problem can be solved in polynomial time for three subclasses of interval graphs. The computational complexity of the problem for chordal graphs, interval graphs, and trees are still open. Especially, tree seems to be the next target. We can decide if two independent sets are reconfigurable in linear time [4], then can we find a shortest sequence for a yes-instance? As in the 15-puzzle, finding a shortest one can be NP-hard. Even we do not know that the length can be bounded by any polynomial or not for a tree. It is an interesting open question whether there is any instance on some graph classes whose reconfiguration sequence requires super-polynomial length.

References

[1] Bogart, K.P., West, D.B.: A short proof that 'proper=unit'. Discrete Mathematics 201, pp. 21–23 (1999)

[2] Bonsma, P., Kamiński, M., Wrochna M.: Reconfiguration Independent Sets in Claw-Free Graphs arXiv:1403.0359, 2014.

[3] Brandstädg, A., Le, V.B., Spinrad, J.P.: Graph Classes: A Survey, SIAM (1999)

[4] Demaine, E.D., Demaine, M.L., Fox-Epstein, E., Hoang, D.A., Ito T., Ono, H., Otachi, Y., Uehara, R., Yamada, T.: Linear-Time Algorithm for Sliding Tokens on Trees. TCS 600, pp. 132–142 (2015)

[5] Deng, X., Hell, P., Huang., J.: Linear-time representation algorithms for proper circular-arc graphs and proper interval graphs. SIAM J. Computing 25, pp. 390–403 (1996)

[6] Fox-Epstein, E., Hoang, D.A., Otachi, Y., Uehara, R.: Sliding Token on Bipartite Permutation Graphs. In Proc. of ISAAC, accepted, 2015.

[7] Gardner, M.: The Hypnotic Fascination of Sliding-Block Puzzles. Scientific American 210, pp. 122–130 (1964).

[8] Gopalan, P., Kolaitis, P.G., Maneva, E.N., Papadimitriou, C.H.: The connectivity of Boolean satisfiability: computational and structural dichotomies. SIAM J. Computing 38, pp. 2330–2355 (2009)

[9] Hearn, R.A., Demaine, E.D.: PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. TCS 343, pp. 72–96 (2005)

[10] Hearn, R.A., Demaine, E.D.: Games, Puzzles, and Computation. A K Peters (2009)

[11] Ito, T., Demaine, E.D., Harvey, N.J.A., Papadimitriou, C.H., Sideri, M., Uehara, R., Uno, Y.: On the complexity of reconfiguration problems. TCS 412, pp. 1054–1065 (2011)

[12] Kamiński, M., Medvedev, P., Milanič, M.: Complexity of independent set reconfigurability problems. TCS 439, pp. 9–15 (2012)

[13] Korte, N., Möhring, R.: An incremental linear-time algorithm for recognizing interval graphs. SIAM J. Computing 18, pp. 68–81 (1989)

[14] Makino, K., Tamaki, S., Yamamoto, M.: An exact algorithm for the Boolean connectivity problem for k-CNF. TCS 412, pp. 4613–4618 (2011)

[15] Mouawad, A.E., Nishimura, N., Pathak, V., Raman, V.: Shortest Reconfiguration Paths in the Solution Space of Boolean Formulas. In Proc. of ICALP 2015, LNCS 9134, pp. 985–996 (2015)

[16] Mouawad, A.E., Nishimura, N., Raman, V., Simjour, N., Suzuki, A.: On the parameterized complexity of reconfiguration problems. In Proc. of IPEC 2013, LNCS 8296, pp. 281–294 (2013)

[17] Mouawad, A.E., Nishimura, N., Raman, V., Wrochna, M.: Reconfiguration over tree decompositions. In Proc. of IPEC 2014, LNCS 8894, pp. 246–257 (2014)

[18] Ratner, R., Warmuth, M.: Finding a shortest solution for the $N \times N$-extension of the 15-puzzle is intractable. J. Symb. Comp., Vol. 10, pp. 111–137, 1990.

[19] Slocum, J.: The 15 Puzzle Book: How it Drove the World Crazy. Slocum Puzzle Foundation, 2006.