# Sliding Token on Trees in Linear Time

Erik Demaine[*] Martin Demaine[†] Eli Fox-Epstein[‡] Duc Hoang[§]
Takehiro Ito[¶] Hirotaka Ono[‖] Yota Otachi[**] Ryuhei Uehara[††]
Takeshi Yamada[‡‡]

**概 要**

Sliding Token is a well-studied reconfiguration problem over independent sets in which one must transform one independent set into another by iteratively replacing a vertex with a neighbor. We give a linear-time algorithm for Sliding Token on trees.

## 1 Introduction

*Reconfiguration problems* have been subject to much recent attention and study. For references to many reconfiguration problems and motivation for this line of study, we refer the reader to Demaine et al. [1]. We focus on just one reconfiguration problem, Sliding Token, which is a natural reconfiguration problem over independent sets on graphs. Recall, an *independent set* of a graph is a subset of its vertices such that no two are adjacent.

Independent set $I$ *slides to* independent set $J$ if $|I| = |J|$ and there exists an edge $uv \in E(G)$ where $I \triangle J = \{u, v\}$. Here, we also say $u$ slides to $v$ in $I$. A *reconfiguration sequence* is a sequence of independent sets $I_1, I_2, \cdots, I_k$ such that for all $i \in \{1, \cdots, k-1\}$, $I_i$ slides to $I_{i+1}$. For independent sets $I$ and $J$ on graph $G$, we write $I \overset{G}{\leftrightsquigarrow} J$ if there exists a reconfiguration sequence that starts with $I$ and ends with $J$. In general, Sliding Token takes as input a graph $G$ and two independent sets $I \subseteq V(G)$ and $J \subseteq V(G)$. It is a yes-instance if and only if there is a reconfiguration sequence from $I$ to $J$.

Our main result is that Sliding Token can be solved in linear time on trees.

### 1.1 Preliminaries

Let $\mathbb{I}(x)$ be the indicator function: 1 if $x$ is true and 0 otherwise. The degree of a vertex $v$ is given by $\delta(v)$. Given a graph $G$ and vertex-subset $S$, denote by $G[S]$ the subgraph induced by $S$. Let $T^u_v$ denote the subtree of tree $T$ rooted at $v$ when $u$ is root of $T$, i.e. the subgraph induced by the set of all vertices whose unique path to $u$ contains $v$. $N(G, v)$ is the set of all vertices adjacent to $v$ in $G$ and $N[G, v] = N(G, v) \cup \{v\}$. $N[G, S] = \cup_{v \in S} N[G, v]$ for vertex-subset $S$. When the graph is unambiguous, $N(v)$, $N[v]$, or $N[S]$ may be written. Finally, given an independent set $I$, let $I_{uv} = \mathbb{I}(u \in I \vee v \in I)$ be syntactic sugar for indicating that one of $u$ or $v$ is in $I$.

[*]MIT
[†]MIT
[‡]Brown University
[§]JAIST
[¶]Tohoku University
[‖]Kyushu University
[**]JAIST
[††]JAIST
[‡‡]JAIST

## 1.2 Related Results

Demaine et al. [1] give a quadratic-time algorithm to compute a reconfiguration sequence between two given independent sets or decide that one does not exist. Furthermore, they state that just deciding if a reconfiguration sequence exists without producing a witness could potentially be subquadratic. Here, we give a linear algorithm for the decision problem.

A vertex $v$ is *$(G, I)$-rigid* for an independent set $I$ on graph $G$ if $v \in I$ and $I \overset{G}{\leftrightsquigarrow} J$ only if $v \in J$. Demaine et al. give two powerful lemmas about rigid vertices.

**Lemma 1** ([1], Lemma 6)**.** *Let $I$ and $J$ be independent sets of tree $T$ of the same size. If there are no $(T, I)$- or $(T, J)$-rigid vertices then $I \overset{T}{\leftrightsquigarrow} J$.*

From this, we immediately get:

**Corollary 1.** *$I \overset{G}{\leftrightsquigarrow} J$ if and only if the set $X$ of $(T, I)$-rigid vertices is exactly the same as the $(T, J)$-rigid vertices and for each component $C$ of $T[V(T) \setminus N[T, X]]$, we have $|V(C) \cap I| = |V(C) \cap J|$.* □

This reduces the problem to identifying rigid vertices. To assist with this, we have:

**Lemma 2** ([1], Lemma 1)**.** *Let $I$ be an independent set of tree $T$ and let vertex $u \in I$.*

*(a) Suppose that $V(T) = \{u\}$. Then $u$ is $(T, I)$-rigid.*

*(b) Suppose that $|V(T)| \geq 2$. Then $u$ is $(T, I)$-rigid if and only if, for all neighbors $v \in N(T, u)$, there exists a vertex $w \in I \cap N(T^u_v, v)$ that is $(T^v_w, I \cap T^v_w)$-rigid.*

# 2 Rigid Vertices

From Lemma 2, we extract a recurrence that will be used to efficiently find rigid vertices. Define $\Phi(u, v)$ and $\phi(v)$ as follows:

$$\Phi(u,v) = \begin{cases} \mathbb{I}(v \in I) & v \text{ is a leaf} \\ \mathbb{I}\left(\sum\{\Phi(v,w) \mid w \neq u, w \in N(T,v)\} = \delta(v) - 1\right) & v \in I \text{ is not a leaf} \\ \mathbb{I}\left(\sum\{\Phi(v,w) \mid w \neq u, w \in N(T,v) \cap I\} > 0\right) & v \notin I \text{ is not a leaf} \end{cases}$$

$$\phi(v) = \mathbb{I}\left(\sum_{w \in N(T,v)} \Phi(v,w) = \delta(v)\right)$$

For the purposes of efficient computation, we give an equivalent definition that more closely mirrors how the algorithm functions, despite the apparent circularity in its definition.

$$C(v) = \sum\{\Phi(v,w) \mid w \in N(G,v), I_{vw} = 1\} \tag{1}$$

$$\Phi(u,v) = \begin{cases} \mathbb{I}(v \in I) & v \text{ is a leaf} \\ \mathbb{I}(C(v) - \Phi(v,u) = \delta(v) - 1) & v \in I \text{ is not a leaf} \\ \mathbb{I}(C(v) - I_{uv}\Phi(v,u) > 0) & v \notin I \text{ is not a leaf} \end{cases}$$

$$\phi(v) = \mathbb{I}(C(v) = \delta(v))$$

These are equivalent definitions of $\Phi$ and $\phi$, seen by expanding $C(v)$ and cancelling from the summation the term corresponding to $w = u$.

**Lemma 3.** *Vertex $u \in I$ is $(T, I)$-rigid if and only if $\phi(u) = 1$.*

*Proof.* The recurrences for $\phi$ and $\Phi$ directly mirror the computation implied by Lemma 2. □

## 3 Main Algorithm

**Algorithm 1:** DecideSlidingToken

```
Input: Tree $T$, independent sets $I$ and $J$
Output: “yes” or “no”
1 $F \leftarrow$ DeleteRigidVertices$(T, I)$
2 $F' \leftarrow$ DeleteRigidVertices$(T, J)$
3 if $F = F'$ then
4     foreach componenent $C$ of $F$ do
5         if $|V(C) \cap I| \neq |V(C) \cap J|$ then
6             return “no”
7     return “yes”
8 else
9     return “no”
```

Corollary 1 implies a simple algorithm: DecideSlidingToken (Algorithm 1). Note that for DecideSlidingToken to run in linear time, the loop on line 4 of Algorithm 1 should be implemeneted by keeping a set of unvisited vertices and running a graph search over each component, starting from an arbitrary unvisited vertex whenever the search gets stuck.

Once the subroutines used by the algorithm have been explained, proof is given of the main result:

**Theorem 1.** *In linear time,* DecideSlidingToken *(Algorithm 1) returns “yes” if $I \overset{T}{\leftrightsquigarrow} J$ and “no” otherwise.*

## 4 Deleting Rigid Vertices

DeleteRigidVertices defers to an auxiliary procedure ComputePhi to implement the recurrence implied in Lemma 2. This subroute performs the bulk of the work by populating a table $C(\cdot)$ with the information necessary to decide if a vertex is rigid. The table $P(\cdot)$ is used to bound the number of calls to ComputePhi.

**Lemma 4.** *Suppose that* ComputePhi *(Algorithm 3) runs in amortized constant time per call. Then* DeleteRigidVertices *(Algorithm 2) runs in linear time.* □

**Lemma 5.** *Suppose that right before the first iteration of the loop on line 7 of Algorithm 2, for any $v \in I$, $C(v)$ is correct according to the value in recurrence (1). Then* DeleteRigidVertices *(Algorithm 2) deletes exactly the set of vertices that are $(T, I)$-rigid or adjacent to a $(T, I)$-rigid vertex.*

*Proof.* Follows directly from the definition of $\phi(v)$ and Lemma 3. □

**Algorithm 2:** DeleteRigidVertices

```
Input: Tree $T = (V, E)$, independent set $I$
Output: Forest derived from deleting all $(T, I)$-rigid vertices and neighbors
1  $\Phi(\cdot,\cdot) \leftarrow$ table from $E$ to $\{0,1\}$, initially with $\Phi(u,v) \leftarrow 0\ \forall uv \in E$
2  $C(\cdot) \leftarrow$ table from $V$ to $\{0,1,\dots,|V|\}$, initially with $C(v) \leftarrow 0\ \forall v \in V$
3  $P(\cdot) \leftarrow$ table from $V$ to subsets of $V$, initially with $P(v) \leftarrow N(T,v)\ \forall v \in V$
4  foreach edge $uv \in E$ do
5      ComputePhi$(T, I, \Phi, C, P, u, v)$
6      ComputePhi$(T, I, \Phi, C, P, v, u)$
7  foreach vertex $v \in I$ do
8      if $C(v) = \delta(v)$ then
9          delete $N[T, v]$ from $T$
10 return the resulting forest
```

**Algorithm 3:** ComputePhi

```
Input: Forest $T$, independent set $I$, tables $\Phi$, $C$, $P$, vertices $u$, $v$
1  if ComputePhi$(T, I, \Phi, C, P, u, v)$ was ever executed before then
2      return
3  foreach neighbor $w \in P(v)$ do
4      if $w \neq u$ then
5          ComputePhi$(T, I, \Phi, C, P, v, w)$
6          remove $w$ from $P(v)$
7  $y \leftarrow C(v) - I_{uv} \cdot \Phi(v,u)$
8  $\Phi(u,v) \leftarrow \begin{cases} \mathbb{I}(y = \delta(v) - 1) & v \in I \\ \mathbb{I}(y > 0) & v \notin I \end{cases}$
9  if $I_{uv} = 1$ and $\Phi(u,v) = 1$ then
10     $C(u) \leftarrow C(u) + 1$
```

## 5 ComputePhi in linear total time

**Lemma 6.** ComputePhi *(Algorithm 3), when invoked with vertices $u$ and $v$, assigns the correct values to $\Phi(u,v)$ and $C(u)$ according to the alternate recurrence (1).*

*Proof.* First, notice that if $v$ is a leaf then the $y$-value on line 7 is 0, so $\Phi(u,v) = 1$ if and only if $v \in I$ because $\delta(v) - 1 = 0 = y$.

Now suppose, toward contradiction, that some $\Phi$-value is not correctly computed. Say the induced subtree of an ordered pair $(u,v)$, with $v$ an internal vertex, is $T_w^u$. Let $(u,v)$ be the pair with a smallest (in number of vertices) induced subtree such that $\Phi(u,v)$ is computed incorrectly. Then the value of $C(v)$ is wrong. $C(v)$ is the sum of the appropriate $\Phi$-values for vertex pairs with strictly smaller induced subtrees. Thus, one of these $\Phi$-values, say $\Phi(v,w)$, must be incorrect. Notice that if $w \neq u$, ComputePhi$(T, I, \Phi, C, P, v, w)$ must have already been invoked to compute $\Phi(v,w)$. This violates the assumption that $\Phi(u,v)$ was a minimal counterexample: $(v,w)$ induces a smaller subtree than $(u,v)$.

Consequently, $\Phi(u, v)$ is correct.

The last line of the algorithm ensures $C(u)$ is tracking that which is claimed. □

**Proposition 1.** *There are a linear number of calls to* ComputePhi *in an execution of* DeleteRigidVertices.

*Proof.* DeleteRigidVertices is directly responsible for $2|E|$ calls. Every other call can be charged to a distinct item in a set in $P(\cdot)$, and in total there are only $2|E|$ items in all the sets in $P(\cdot)$. Thus, ComputePhi is called at most $4|E|$ times. □

**Lemma 7.** *Each invocation of* ComputePhi *performs an amount of work that is linear in the number of recursive calls it makes.*

*Proof.* Except potentially one iteration where $u = w$ (that takes constant time), each iteration of the loop on line 3 can be charged to a distinct self-recursive call. The rest consists of a constant number of atomic operations. □

## 6 Putting it all together

*Proof of Theorem 1.* Linear runtime follows from Lemma 4, Lemma 7, and Proposition 1, along with the observation that both labelled forest isomorphism and set intersections are trivially computed in linear time. Correctness follows from Corollary 1, Lemma 5, and Lemma 6. □

## 参考文献

[1] E. D. Demaine, M. L. Demaine, E. Fox-Epstein, D. A. Hoang, T. Ito, H. Ono, Y. Otachi, R. Uehara, and T. Yamada. Polynomial-Time Algorithm for Sliding Tokens on Trees. *ArXiv e-prints*, June 2014.