# Dynamic Unit Disk Graph Recognition
# Internship Report

Neven Villani
supervised by Arnaud Casteigts, LaBRI

June 8 to July 16, 2021

## Acknowledgement

Thanks to Arnaud Casteigts for his tireless supervision, enthusiasm, and kind words; to Timothée Corsini for his patience and his valuable feedback in making my presentation more accessible; to Jason Schoeters, Eleni Akrida and George Mertzios for their support.

I also express my gratitude to the LaBRI team for the warm welcome I received and for organizing so many opportunities for discussion, which very much helped me grasp the extent of the daily activities of a researcher.

# Contents

# 1 Introduction

Unit Disk Graphs (UDGs) in $n$ dimensions are undirected graphs for which there exists a mapping from the vertices to points in $\mathbb{R}^n$ such that two points are linked if and only if they are close enough. This is interpreted as them being within communication range of one another and can be used to model entities that are able to communicate remotely. There are a few ways to define formally UDG, which are introduced in Section 2.
UDG arise in particular when a group of entities (e.g. mobile phones, drones, broadcast towers) can communicate with each other within a certain range. The radius $R$ of the disks models the maximum communication distance, and two vertices can communicate if their corresponding points are close enough. As for dynamic graphs, they can model the evolution of relationships by means of a discrete sequence of graphs with the same vertex set, with successive graphs (called *snapshots*) being interpreted as successive instants. These two notions of temporal graphs and unit disk graphs can be used together to study the temporal evolution of a group of communicating entities. There could be hope to reverse-engineer the movements of such entities from a record of their contacts, the interest for this approach being that contact traces are cheaper to obtain and store than accurate positional data: there is no need for costly measuring instruments, and there is less data to store. In addition, inferred positional data allows for realistic transformations: adding vertices and increasing the communication range in a way that is consistent.

Such is the path taken by [**Plausible**] (more information available at [**Plausible_Web**]), to produce the library [**DITL**] in which heuristics are presented to infer movements of people or objects given their contact trace (the times at which they were able to communicate). In other words, the goal is to construct a dynamic proximity model from a temporal graph.
However in that paper it is assumed that there exists such a proximity model and some inaccuracies (vertices being linked despite being far away, or being separated yet close) are tolerated. The effect of additional information on the accuracy is measured. This additional information includes knowing the position of some vertices at all times, knowing the position of all vertices initially, or making assumptions on the movement of disks. The heuristics proposed show that the movements reconstructed from a contact trace are often a decent approximation of actual movements, but there is much room for improvement if inaccuracies are not tolerated. This has to be done conjointly with the trace being perfect : there is no opportunity for contact that is missed by inaccurate measuring instruments.
The paper in question also introduces the notion of a *perfect contact trace*, in which no contact opportunities are missed, and these perfect traces are used mostly to evaluate the performance of the heuristics. This work however assumes perfect traces throughout.

In the static case, UDG recognition is known to be NP-hard in two dimensions [**UDG_NPHard**]. Restriction to trees is also NP-hard, but [**Caterpillar**] recently showed a linear-time algorithm for caterpillars (trees with all leaves at distance 1 of a central path).
The one-dimensional static problem (in which disks are unit intervals) has also been extensively studied, and at least three linear-time recognition algorithms have been proposed [**Linear_PIG**], [**Simple_PIG**], [**Consecutive_Ones**].

The objective of this intership is to take a more theoretical look at the dynamic UDG recognition problem, with the goal of studying the influence of the additional time parameter in the general case and finding restrictions of the problem that are tractable yet physically plausible. More specifically, the aim is to:

- simplify the proof of NP-hardness in the general case (since the problem is more difficult we can expect the proof of hardness to be simpler);
- study some restrictions of movements (integer coordinates, bounded speed, bounded acceleration);
- see if the caterpillar constraint can be adapted to a temporal setting;
- analyze the one-dimensional version of the problem.

# 2 Preliminaries

## 2.1 Unit Disk Graphs

A first formal definition of UDG is to describe a mapping from vertices to disks such that vertices are linked iff the corresponding disks intersect

**Definition 1** (UDG – intersection model)**.**
*$G = (E, V)$ has a UDG intersection model with radius $R$ when there exists $\iota : V \to \left\{\overline{B}(x, R) \mid x \in \mathbb{R}^n\right\} \subset \mathcal{P}(\mathbb{R}^n)$ such that for all $v, v' \in V$, $\{v, v'\} \in E \iff \iota(v) \cap \iota(v') \neq \emptyset$.*

Of course if $\iota(v)$ can take any value in $\mathbb{R}^n$, the actual value of the radius is irrelevant since an intersection model for a radius can be rescaled for any other radius.

As mentioned in [**UDG_Clique**], UDGs can also be defined as the graphs which have a *proximity model*:

**Definition 2** (UDG – proximity model)**.**
*$G = (E, V)$ has a UDG proximity model with radius $R$ when there exists $\iota : V \to \mathbb{R}^n$ such that for all $v, v' \in V$, $\{v, v'\} \in E \iff \|\iota(v) - \iota(v')\| \leqslant R$*

i.e. two vertices are linked iff they are separated by at most a fixed distance. A third definition is the *containment model*: vertices are again mapped to unit disks, but they are considered linked when the center of one disk is contained in the other disk.

**Definition 3** (UDG – containment model)**.**
*$G = (E, V)$ has a UDG containment model with radius $R$ when there exists $\iota : V \to \mathbb{R}^n$ such that for all $v, v' \in V$, $\{v, v'\} \in E \iff \iota(v) \in \overline{B}(\iota(v'), R)$*

It should be obvious that not only is the radius also irrelevant in these two alternative definitions, these three kinds of models can be easily transformed into one another (possibly with some rescaling of embeddings).
In most of what follows, computations are performed using a proximity model for $R = 1$, and figures use either a containment model with $R = 2$ or an intersection model with $R = 1$ depending on which one is the most readable.

## 2.2 Statement of the problem

**Input**: $\mathcal{G} = (V, E_0, \cdots, E_\tau)$ a temporal graph
**Known**: constraints on values which $\iota_{i+1}$ can take depending on $\iota_i$
**Output**: **true**/**false** depending on the existence of a mapping $\iota_\bullet : \mathbb{N} \to V \to \mathbb{R}^2$ that satisfies known constraints and so that each $\iota_i$ is a proximity model for $(V, E_i)$, i.e. **true** iff there exists a temporal model
**Output (alternative)**: a representation of one (or several) such mapping if it exists.

The input is to be interpreted as a sequence of graphs on the same vertex set, with each $(V, E_i)$ a different instant in time.

Example of known constraints: $\forall i, \forall v, |\iota_i(v) - \iota_{i+1}(v)| \leqslant 1|$, i.e. vertices have bounded speed. This is the constraint that will be used in Section 3.
Section 4 introduces constraints that are more difficult to express concisely, but they can also be interpreted as the trace being perfect.

# 3 NP-hardness in two dimensions

The static problem is already known to be NP-hard [**UDG_NPHard**]. This automatically makes the dynamic unrestricted problem NP-hard, however a new proof of hardness can have advantages. It may provide insights about the added complexity of the temporal setting, and it can be extended to different settings, some of which are irrelevant for the static case (initial positions known, one event at a time).
Given that the proof in the static case is quite difficult, there is also an opportunity to simplify it in a dynamic setting.

## 3.1 Intuition

A reduction from 3-SAT is constructed. It uses a group of disks for each variable and these variables assemble with each other to successively form clauses. This makes the number of disks linear with regard to the input instead of quadratic in the static proof.
It is ensured that variables can only take two configurations, which are interpreted as **true** and **false**. Constraints are added so that variables cannot change configuration during the lifetime of the graph: it is assumed that the speed of disks is bounded, in particular the maximum distance that a disk can move between two snapshots does not exceed its radius. This prevents disks from crossing each other without an encounter, and should be the baseline requirement for what can be considered "physically plausible" ([**Plausible**] also makes this assumption that the detection range is greater than the distance travelled in one instant). Moreover it would be uninteresting for a temporal graph to be dynamic UDG if and only if (iff) all of its snapshots are UDG, hence the restriction.
This is also in accordance to the definition of *Plausible Mobility* in [**Plausible**]: "In order to be plausible, the inferred movement must realistically limit the speed of the nodes".

One major cause of difficulty can be identified to come from the fact that among several embeddings for each instant, only a few are compatible with all future snapshots, and determining which ones exactly can take exponential time.

## 3.2 Using "blinking" links to add constraints

A key phenomenon that appears in the dynamic case is that of "blinking" links. A link between two vertices $u$ and $v$ is present if $d(u, v) \leqslant 1$ and absent if $d(u, v) > 1$, hence the static case can only constrain two nodes to be either closer or further that one unit apart.

In the dynamic case however, a link $u - v$ that is present on even timestamps and absent on odd timestamps forces $u$ and $v$ to remain separated by a distance within $[1 - 2\delta, 1 + 2\delta]$ with $\delta$ being the distance each node can travel in one unit of time.
Indeed, at $t = 0$, $d(u, v) \leqslant 1$ hence at $t = 1$ $d(u, v) \leqslant 1 + 2\delta$ if both moved in opposite directions at their maximum speed, the constraint that at the next timestamp they be once again within 1 unit of each other prevents their distance from being any bigger.
The same reasoning sets the lower bound to $1 - 2\delta$.

More generally, if during any $K$ consecutive timestamps a link is absent at least once and present at least once then the nodes in question must remain at all times within $1 \pm K\delta$ of each other (the above was for $K = 2$). Given a sufficiently low maximum speed (a small but constant fraction of the radius) it is possible by setting $K$ small enough to force certain pairs of nodes to remain within $1 \pm \varepsilon$ of each other.
What follows uses this property, although $K$ will only be decided later.

### 3.3 Structure for a variable

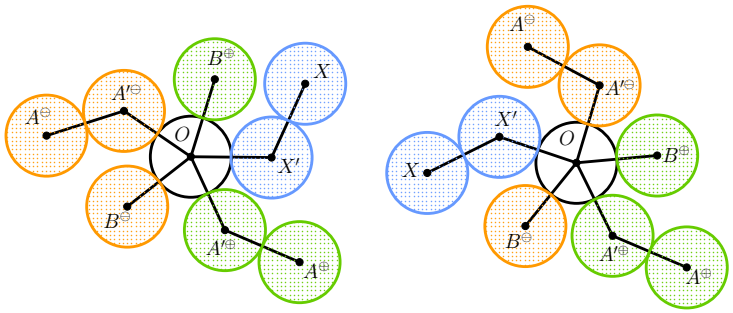


Figure 1: Intersection models of the structure representing a variable
Left: variable set to true. Right: variable set to false.
Both are identical except for the relative position of the $X$ branch

There is one copy of the structure shown in Figure 1 for each variable of the 3-SAT instance. These structures are built in several steps detailed in Section I which guarantee that only the two configurations shown (and small variations that do not change the relative positions of vertices) can exist. These two configurations are arbitrarily interpreted as the variable being set to **true** or **false**.

As the links shown in Figure 1 are permanent (more accurately, from now on they appear at least once every $K$ timestamps), the configuration of each variable is fixed once and for all.
This is crucial for what follows, since it guarantees that the **true**/**false** value of a variable does not change during the construction of the clauses.
A similar structure is built to represent bottom ($\bot$), and is designed to behave as a variable which is always false. This structure can be seen on the bottom right of Figure 2.

### 3.4 Structure for a clause

A clause with 3 literals is assembled by connecting together the structures that represent each literal, in a way that guarantees that each unsatisfied literal adds a pair of disks on the inside of the 12-cycle that constitutes the clause. By packing arguments which are also widely used in [**UDG_NPHard**], the cycle in question can contain at most two pairs of extra disks. Therefore there exists a realization for the clause iff at least one structure has its pair of extra disks outside of the cycle, i.e. if and only if at least one literal is satisfied.

Clauses are handled successively, each over an interval of consecutive snapshots : during $t \in [0, t_0]$ the structures are assembled, then there exists $0 < t_0 < t_1 < \cdots < t_n = \tau$ such that the structure corresponding to clause $C_i$ is assembled during the period $t \in [t_i + 1, t_{i+1}]$. The number of vertices is linear w.r.t. the initial size of the problem (since it is linear w.r.t. the number of variables), and so is the number of instants (since it is linear w.r.t. the number of clauses).
The constraint that structures retain their configuration during the lifetime of the temporal graph ensures that the corresponding variables do not change value, and hence there exists a realization iff the formula is satisfiable.

### 3.5 Adapting the proof to different constraints

This result can be extended to classes of temporal graphs that have very restricted snapshots: it still holds for the graphs in which all snapshots have connected components of size at most 2 (which includes the class of graphs for which all snapshots can be partitionned into caterpillars).

Indeed, instead of links alternating between on and off at each step as described in Section 3.2, they can be desynchronized so that no two links adjacent to the same vertex are active at the same time (see Figure 3). Since all structures constructed are of constant size, it is always possible to leave no link turned off for more than a constant number of steps. Note however that proving this requires an unbounded number of links to be active at the same time. The difficulty of the problem in the case of a very restricted footprint (the footprint is the union of all snapshots, "very restricted" would mean at least a caterpillar), or when there are a bounded

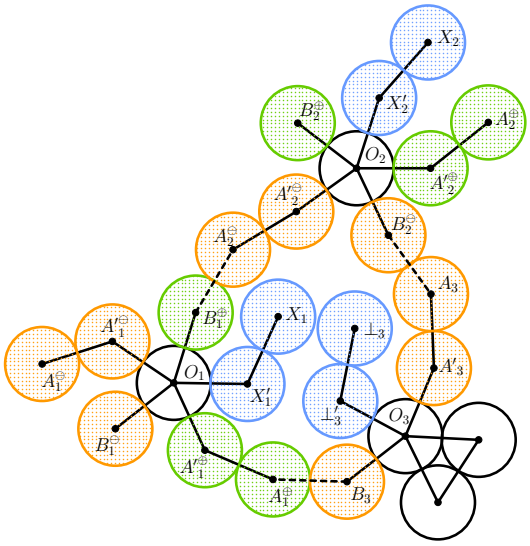


Figure 2: Three variables assembled to form a clause $C = \neg x_1 \vee x_2 \vee \perp$
Represented with $x_1$ and $x_2$ both set to true.
In this particular instance $C$ is satisfied thanks to $x_2$, and the graph has a disk realization.
Dashed lines only for visual clarity to emphasize the boundaries between components.

number of interactions at each snapshot is still open. Both of these constraints can nevertheless be deemed too restrictive for any applications, as they are uncommon in the field of temporal graphs : it is more common to place lower bounds on the footprint (e.g. require that it be connected, but not that it be at most a tree) or upper bounds on local properties of the snapshots (e.g. bound the maximum number of interactions a single entity can have during one instant, but not require that each snapshot be connected or that there be a maximum number of interactions globally).

Section II shows how to adapt the construction when coordinates of all disks are restricted to integers.

The difficulty still holds if all the initial positions of disks are known, or if part of the disks have their positions known at all times. Although [**Plausible**] mentions that these two factors greatly improve the accuracy of existing heuristics, this additional information cannot be said to make the problem significantly easier in the case of a perfect contact trace.
Perhaps more accurate heuristics will emerge, but an efficient exact algorithm is unlikely for any meaningful restriction of the problem.
This does not necessarily contradict the hopes formulated in [**Plausible**] that "perhaps the plausible mobility obtained from the techniques in this paper could be used as a first-pass in a more complete algorithm that actually solves the large constraint system": it is possible that an algorithm with exponential runtime in the general case would be efficient in practice on real-world traces.

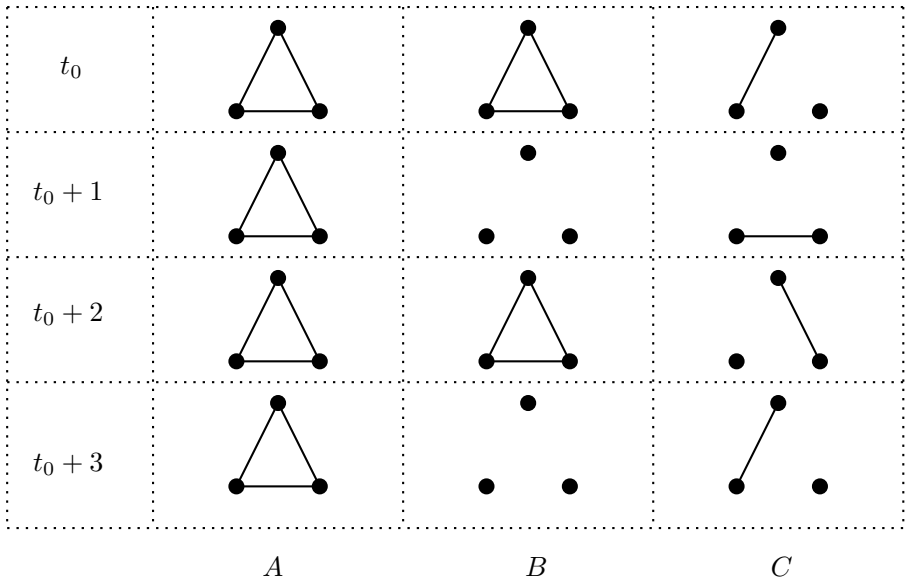


Figure 3: Blinking phenomenon, including desynchronization
All three contact traces $A$, $B$ and $C$ force the vertices to remain close to each other (at most $1 + \varepsilon$ apart). $B$ and $C$ have the added property that they force the vertices to remain separated by at least a distance $1 - \varepsilon$. $C$ has the added property that its connected components are of size at most 2 for each snapshot.

# 4 Tractability in one dimension

A new kind of restriction is now explored: reducing the dimension of the space in which entities can move. This constraint is relevant from a theoretical point of view, since the 1-dimensional UDG are well-studied and their recognition in the static case is easy. From a practical point of view, the argument can be made that many mobile entities operate in contexts in which their movement is effectively one-dimensional, e.g. cars on a highway, passengers in a train.

Vertices are thereby constrained to a single dimension and can take any position in $\mathbb{R}$.
This is equivalently defined as the recognition problem for dynamic Unit Interval Graphs (also called *Proper Interval Graphs*, or PIG, a subclass of *Interval Graphs* , or IG), which are known to be linear-time recognizable in a static setting ([**Consecutive_Ones**], [**Simple_PIG**], [**Linear_PIG**] for PIG; [**Ultimate_IG**], [**Incremental_IG**] for IG).
The trace is assumed to be perfect: events are distinguishable from one another, i.e. $|E_i \Delta E_{i+1}| = 1$, and a continuous transition (defined formally in Definition 8) is expected from one embedding to the next in order for it to be physically realizable. It is argued in Section VII.1 that these transformations are also physically reasonable in that they naturally place an upper bound on the movement speed of vertices.
Section VII.2 discusses allowing several simultaneous events.

The properties in Section 4.1 and Section 4.2 are essential to the final result, because they enable abstracting away the underlying physical model. They state that the set of all possible embeddings can be efficiently represented by a set of permutations (Theorem 1), and provide criteria to determine which manipulations to perform on these sets of permutations to simulate movements compatible with the physical restrictions (Theorem 2).

In what follows, $v$, $\iota(v)$, and the data structure which represents $v$ in the algorithm are all called "a vertex". It should be obvious from the context which of them the term refers to. e.g. "$v$ and $v'$ are at distance at most 1" instead of the more accurate "$|\iota(v) - \iota(v')| \leqslant 1$"

## 4.1 Neighborhood compatibility

$G = (V, E)$ is a PIG.

**Definition 4** (Neighborhood)**.**
*The neighborhood $N[v]$ of $v \in V$ is $\{v\} \cup \{v' \mid \{v, v'\} \in E\}$.*

*The class $C[v]$ of $v \in V$ is $\{v' \mid N[v] = N[v']\}$*

**Definition 5** (Realizable permutation)**.**
*A realizable permutation of $G$ is an ordering $\sigma \in \mathfrak{S}(V)$ such that there exists an injective model $\iota$ with*

$$\forall v, v',\ \sigma(v) < \sigma(v') \iff \iota(v) < \iota(v')$$

**Definition 6** (Permutation compatible with neighborhoods)**.**
*An ordering $\sigma$ is said to be a permutation compatible with neighborhoods of $G$ when*

$$\forall v_0, \exists v^+, v^-\ s.t.\ \forall v,\ \sigma(v^-) \leqslant \sigma(v) \leqslant \sigma(v^+) \iff v \in N[v_0]$$

That is, the neighborhood of each vertex is contiguous.
This can be equivalently defined as $\forall \{v_1, v_2\} \in E,\ \forall v,\ \sigma(v_1) < \sigma(v) < \sigma(v_2) \implies v \in N[v_1] \cap N[v_2]$: if two vertices are neighbors then all vertices in between are also their neighbors.

**Theorem 1** (Compatibility is realizability)**.**
*An ordering is a realizable permutation of $G$ iff it is compatible with neighborhoods of $G$.*

Proof: Section III
Idea: ($\implies$) geometric arguments and triangular inequalities; ($\impliedby$) explicitly construct a model.

## 4.2 Temporal compatibility

Let us now introduce a temporal aspect.
In this section, $\mathcal{G} = (G_i)_{0 \leqslant i \leqslant \tau}$ is a sequence of UDG, with $G_i = (V, E_i)$.
$E_i$ and $E_{i+1}$ differ only by a single event $L_{i+1}$.
The trace is assumed to be initially empty: $E_0 = \emptyset$. See Section VII.3 for how to adapt the algorithm to a specific initial configuration.

Neighborhoods will be indexed by either the snapshot $N_G$ or the instant $N_i$.

**Definition 7** (Event)**.**
*An event is a pair $L_i = \{v_i, v_i'\}$ with $v_i \neq v_i'$ such that $E_{i-1} \,\Delta\, \{L_i\} = E_i$.*

i.e. $L_i \in E_i \iff L_i \notin E_{i-1}$ and $\forall L \neq L_i,\ L \in E_i \iff L \in E_{i-1}$: the connection between $v_i$ and $v_i'$ is destroyed if it already existed and created otherwise, and all other connections remain unchanged.

**Definition 8** (Continuous transition)**.**
*Given two models $\iota$, $\iota'$ of $G$ and $G'$, a continuous transition from $\iota$ to $\iota'$ with event $\{v_1, v_2\}$, written $\iota \xrightarrow{\{v_1,v_2\}} \iota'$, is a continuous function $\varphi : [0,1] \times V \to \mathbb{R}$ such that there exists $0 < t_0 < 1$:*

1. $\varphi(0, \bullet) = \iota$
2. $\varphi(1, \bullet) = \iota'$
3. $\forall \{v, v'\} \neq \{v_1, v_2\}, \forall t,\ |\varphi(t, v) - \varphi(t, v')| \leqslant 1 \iff D_\iota(v, v') \leqslant 1$
4. $\forall t < t_0,\ |\varphi(t, v_1) - \varphi(t, v_2)| \leqslant 1 \iff D_\iota(v_1, v_2) \leqslant 1$
5. $\forall t > t_0,\ |\varphi(t, v_1) - \varphi(t, v_2)| \leqslant 1 \iff D_{\iota'}(v_1, v_2) \leqslant 1$
6. $|\varphi(t_0, v_1) - \varphi(t_0, v_2)| = 1$

*If conditions 4, 5, 6 are removed and condition 3 is extended to also include $\{v_1, v_2\}$, the transition is said to be without event and is written $\iota \to \iota'$.*

That is, no event other than $\{v_1, v_2\}$ may happen during the continuous transformation of $\iota$ into $\iota'$ (conditions 1, 2, 3), and $\{v_1, v_2\}$ occurs exactly at time $t_0$ (conditions 4, 5, 6).
This is meant to express the movements that are physically possible knowing the two instants $t = 0$ and $t = 1$ and assuming that all contacts are properly detected.

**Definition 9** (Temporal compatibility)**.**
*Any padded model of $G_0$ is temporally compatible with an empty sequence of events.*
*If $\iota$ is compatible with events $L_1, \cdots, L_i$, $\iota'$ is a padded model of $G_{i+1}$, and there exists a continuous transition $\iota \xrightarrow{L_{i+1}} \iota'$ then $\iota'$ is temporally compatible with events $L_1, \cdots, L_{i+1}$.*

*A sequence $\iota_0 \xrightarrow{L_1} \iota_1 \xrightarrow{L_2} \cdots \xrightarrow{L_i} \iota_i$ is called a temporal model of $\mathcal{G}_i$.*

**Definition 10** (Discrete transition)**.**
*When $\sigma$ is compatible with the neighborhoods of $G$, a discrete transition from $\sigma$ compatible with $G$ is a permutation $\sigma'$ such that $\forall v, v',\ N_G[v] \neq N_G[v'] \Longrightarrow (\sigma(v) < \sigma(v') \iff \sigma'(v) < \sigma(v'))$.*

*It is written $\sigma \xrightarrow{G} \sigma'$.*

That is, $\sigma'$ may differ from $\sigma$ only in the relative positions of vertices that have the same class in $G$.
The main result of this section establishes an equivalence between physically plausible movements and manipulations on permutations:

**Theorem 2** (Discrete decomposition of a continuous transition)**.**
*Let $\sigma, \sigma'$ two permutations compatible with the neighborhoods of $G$ and $G'$ respectively. Let $\iota$, $\iota'$ padded models of $G$ and $G'$ that have the orderings of $\sigma$ and $\sigma'$.*
*There exists a continuous transition $\iota \xrightarrow{\{v_1,v_2\}} \iota'$ iff there exists $\sigma''$ such that $\sigma \xrightarrow{G} \sigma'' \xrightarrow{G'} \sigma'$.*

*Note that in particular $\sigma''$ must be compatible with the neighborhoods of both $G$ and $G'$.*

Proof: Section IV
Idea: ($\Longrightarrow$) intermediate value theorem to isolate the event; ($\Longleftarrow$) naive interpolation.

## 4.3 Representation by a $PQ$-forest

The algorithm keeps track of the possible configurations using a $PQ$-forest.

**Definition 11** ($PQ$-forest)**.**
*A $PQ$-forest is a tree with the following constraints:*

- *children of the root are labeled $Q$;*
- *each leaf corresponds to a vertex;*
- *direct parents of the leaves are labeled $P$.*

**Notation 1.**
*A $P$ node is written ${}^P(v_1, \cdots, v_n)$; A $Q$ node is written ${}^Q[c_1, \cdots, c_n]$; a $PQ$-forest is written ${}^F\{q_1, \cdots, q_n\}$. Where $v_i$ are vertices, $c_i$ are $P$- or $Q$-nodes, $q_i$ are $Q$-nodes.*

*When $q$ is a $Q$-node in $F$ a $PQ$-forest, $F_q$ is the subtree rooted in $q$.*

This is to be understood as a forest of $PQ$-trees as in **[Consecutive_Ones]**, but with a few small differences:

- the $PQ$-trees are not necessarily normalized since they can have a subtree of the form ${}^Q[{}^P(v)]$ which is disallowed in standard $PQ$-trees
- they only have $P$-nodes as direct parents of leaves;
- they contain at least one $Q$-node.

**Definition 12** (Associated permutations of a $PQ$-forest)**.**
*The set of associated permutations $\Sigma(F)$ of a $PQ$-forest $F$ is defined recursively as:*

- $\Sigma({}^P(v_1, \cdots, v_n)) = \mathfrak{S}(v_1, \cdots, v_n)$
- $\Sigma({}^Q[c_1, \cdots, c_n]) = \{(\Sigma(c_1) \cdots \Sigma(c_n))\} \cup \{(\Sigma(c_n) \cdots \Sigma(c_1))\}$
- $\Sigma({}^F\{q_1, \cdots, q_n\}) = \mathfrak{S}(\Sigma(q_1), \cdots, \Sigma(q_n))$

Intuitively, $P$-nodes represent an arbitrary permutation that can change with time. $Q$ nodes represent two sets of permutation that have a 1-1 mirror correspondence in order to indicate knowledge of relative positions but not orientation. A forest is an arbitrary fixed permutation of sets of permutations when there is no known information on their relative position.

The main useful property of PQ-forests is that they are able to represent all possible sets of permutations that vertices in a dynamic 1-dimensional UDG can take.
This allows the algorithm to manipulate a $PQ$-forest as an efficient discrete representation of all possible models. More precisely, it will at all times retain a $PQ$-forest of all the temporally compatible orderings of vertices given past events, and at each new event it uses the criterion developed in Theorem 2 to calculate all the new temporally compatible permutations.

See Section V for some properties of $PQ$-forests useful in the proof of the algorithm.

## 4.4 Algorithm (summary)

The input given the algorithm consists of the number of vertices $n$ and the sequence of events $\mathcal{L} = (L_i)_{1 \leqslant i \leqslant \tau}$.
Procedures operate on a single $PQ$-forest $F$.
At the end of each loop iteration the following properties (some redundant) are ensured:

- vertices that share the same $P$-node are those in the same class;
- vertices that do not have the same direct $Q$ parent cannot be linked;
- the neighborhood of each vertex is contiguous.

These are then used to prove that $\Sigma(F)$ is always exactly the set of valid permutations for the last step of a temporal model of $\mathcal{G}_i$.

See Section VI.1 for a list of the functions assumed to be implemented as part of the data structure, Section VI.2 and Section VI.3 for the auxiliairy functions used to handle LinkUp and LinkDown events, and Section VI.4 for the main loop.

A complexity analysis is provided in Section VII.5 and Section VII.6. The main result is that with an efficient representation of children as a height-balanced binary tree, the algorithm can perform amortized $O(\lg n)$ operations for each event, making the total runtime $O(\tau \cdot \lg n)$. Since this is also the expected size of the input, the algorithm can be considered to run in linear-time (this is a bit of a simplification, since unique identifiers are considered to be of size $O(\lg n)$ in the input, yet the algorithm assumes they can be read and compared in $O(1)$).
It also happens to be an online algorithm: it is able to incrementally perform all computations in real time, without access to future events. Section VII discusses some extensions of the algorithm and the physical model, and Section IX is an overview of forbidden patterns.

# 5 Conclusion

Having so far only been studied by [**Plausible**], the problem of inferring plausible mobility from contacts is still in its infancy. The negative results shown here in the two-dimensional setting are not incompatible with the possibility of having an algorithm that performs well in practice on real-life traces, and the one-dimensional tractability is encouraging, although applying it as is to messy real-world contact traces would require making it somehow more fault-tolerant.

## Approximate timeline

- Week 1: 50% 2D bibliography, 40% 2D unrestricted, 10% lab activities
- Week 2: 30% 1D bibliography, 30% 1D exploration, 20% 2D integer, 20% lab activities
- Week 3: 90% 1D exploration + algorithm + proof, 10% lab activities
- Week 4: 70% 1D proof, 20% beamer, 10% lab activities
- Week 5: 40% beamer, 30% lab activities, 15% 1D forbidden patterns, 15% 1D improvements
- Week 6: 50% forbidden patterns, 20% report, 20% AATG, 10% lab activities

Lab activities included a dozen conferences (20min to 1h), a thesis defense, a general lab assembly, and working on a problem unrelated to my internship with my supervisor and two other researchers.

## Future prospects

A question that remains open in the two-dimensional case is that of the tractability of the problem when the footprint is a caterpillar. It is worth noting that the reductions proposed produce snapshots that are simpler than in the static proof, but footprints that are more complex. Managing to prove hardness with a footprint that is more restrictive than those that make the 2-dimensional problem tractable would show that the increase in difficulty when temporality is introduced is much greater than shown here.

Developing an exact algorithm for the 2-dimensional problem also remains to be done, but a first step would likely be to find an algorithm for the static case first. Perhaps the restriction to integer coordinates should be attempted beforehand.

As for the one-dimensional case, a more in-depth study of forbidden patterns could lead to a characterization as for PIG.

Finally, the static 1-dimensional recognition problem was also first solved using $PQ$-trees, and simpler algorithms were then discovered. It is possible that simpler recognition procedures will emerge for the dynamic case, perhaps also some that would tolerate simultaneous events.

# I Assembling variables

Section 3.3 skimmed over how to actually ensure that there can only be two possible realizations of each structure for variables. This process is detailed in what follows.
A single first component shown in Figure 4 is constructed, which will determine the orientation of all other components shown in Figure 1 and Figure 2.

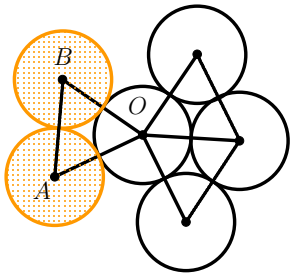


Figure 4: Polarizer

In any realization there can be only one of two possible orientations of $OAB$, and the physical restrictions prevent this orientation from changing during the lifetime of the graph.

"$A^{\oplus}$ branch" (resp. $A^{\ominus}$, $B^{\oplus}$, $B^{\ominus}$, $X$) refers to $A^{\oplus}$ and $A'^{\oplus}$ (resp. $A^{\ominus}$ and $A'^{\ominus}$, $B^{\oplus}$, $B^{\ominus}$, $X$ and $X'$).
The $A$ (resp. $B$, $\oplus$, $\ominus$ branches are the $A^{\oplus}$ and $A^{\ominus}$ branches (resp. $B^{\oplus}$ and $B^{\ominus}$ $A^{\oplus}$ and $B^{\oplus}$, $A^{\ominus}$ and $B^{\ominus}$).
Variables as in Figure 1 are built in several steps. First the $A$ and $B$ branches are connected to $O$, then their relative positioning is set once and for all by successively connecting nodes of the variable with nodes of the polarizer:

1. $A^{\ominus}$ to $A$ and $B^{\ominus}$ to $B$
2. $A^{\oplus}$ to $A$ and $B^{\oplus}$ to $B$
3. $A^{\ominus}$ and $A'^{\ominus}$ to $B^{\oplus}$
4. $A^{\oplus}$ and $A'^{\oplus}$ to $B^{\ominus}$

Each of these (sometimes redundant) connections further restricts the relative positions that the branches can take: 1. and 2. uniquely determine the orientation of $OA'^{\ominus}A^{\ominus}B^{\ominus}$ and $OA'^{\oplus}A^{\oplus}B^{\oplus}$, 3. and 4. make it so that $B$ and $A$ branches must alternate.
Only two valid configurations are left, only one of which is compatible with each configuration of the polarizer. Once this step is over all variables are identical and the $X$ branches have not yet been placed.

They are added by connecting $O$ with and only with $X'$ all the while $A'^{\ominus}$ is connected to $B^{\oplus}$ and $A'^{\oplus}$ is connected to $B^{\ominus}$. This forces the $X$ branch to place itself in one of two possible positions: between the $\oplus$ branches or between the $\ominus$ branches. These two configurations are illustrated in Figure 1 for one of the two orientations of the polarizer.

# II Discrete movements

Coordinates are now restricted to be integers. Since this invalidates the equivalence between containment and intersection models, three different geometries shown below have been studied. All of them leave the problem NP-hard.

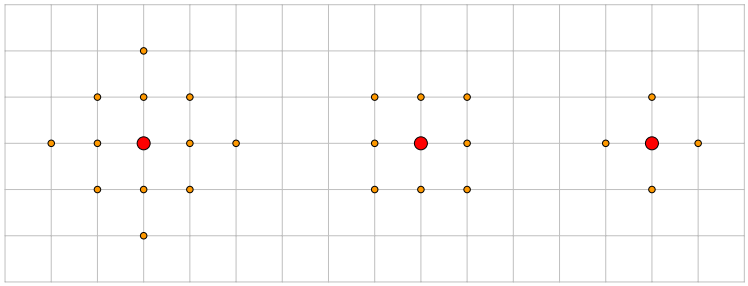

Figure 5: Three different geometries for integer coordinates
In each, the neighborhood of the red point are the orange points.
Left: containment $R = 2$ (or equivalently, intersection $R = 1$).
Middle: usual definition of the neighborhood, can be seen as containment $R = \sqrt{2}$
Right: containment $R = 1$ (or equivalently, intersection $R = \frac{1}{2}$).

The allowed movements are at least the four nearest neighbors, and at most the considered neighborhood for the chosen geometry. This ensures that two points cannot exchange positions in one step without encountering each other.
What follows considers the specific case of containment $R = 2$, but the proof can easily be adapted to any of the other two geometries represented in Figure 5 with similar structures.

Once again, the proof relies on the existence of contact traces which can correspond to several configurations, different enough from one another so that the configuration taken initially has to stay the same during the whole lifetime of the graph.

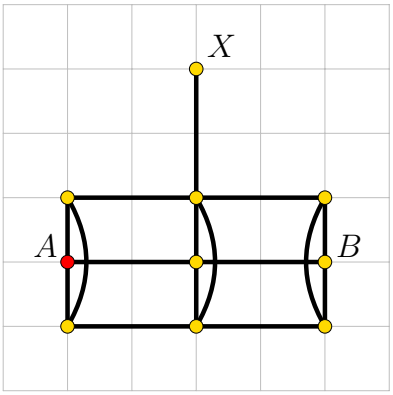


Figure 6: A variable
The bottom 9-disk structure is rigid and can be extended to form a structure as large as required that has only one valid configuration excluding symmetries.

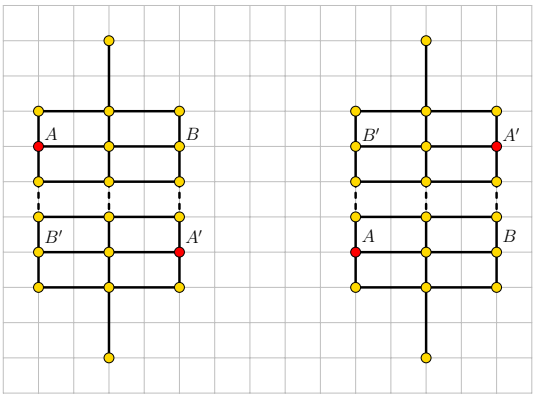


Figure 7: The two possible states of a pair of literals
Left: $x$ set to true. $l = x$ top, $l = \neg x$ bottom.
Right: $x$ set to false. $l = x$ bottom, $l = \neg x$ top.
Once the initialization is done the dashed links are separated and the two clauses can be used independently.
A satisfied literal always has the extra disk on top.
A non-negated literal always has its $A$ node (red) to the left.
Six more configurations can be obtained with a vertical symmetry and a 90-degree rotation,
but they will be rejected when the literal is connected to the rest of the structure.

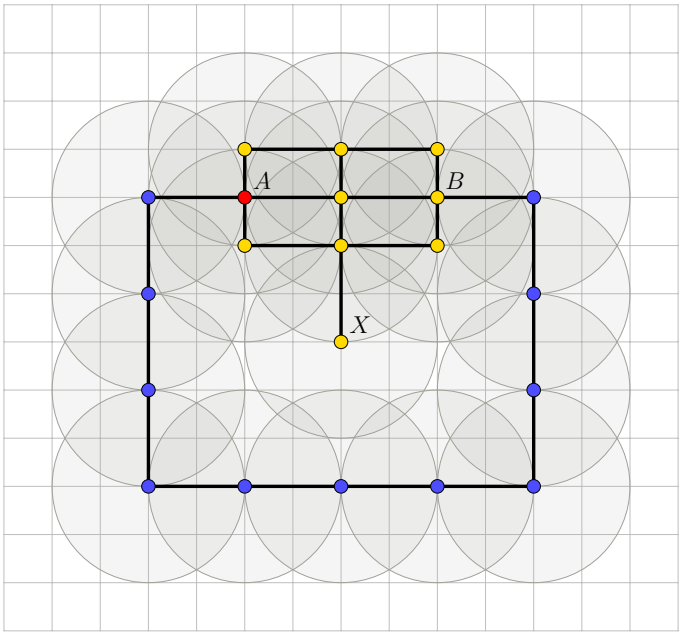


Figure 8: An unsatisfied literal
The $X$ disk leaves no room for any more disks within the enclosed space.
(represented with a containment model)
For now the blue disks are assumed to be immovable, an outer structure will later be set up to make them so.

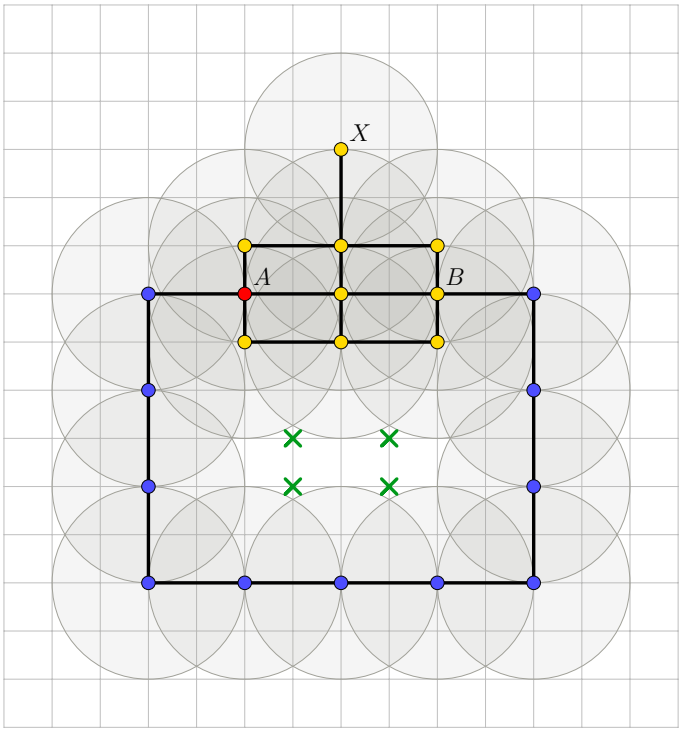


Figure 9: A satisfied literal
There are four available positions (green crosses) that an extra lone disk can occupy

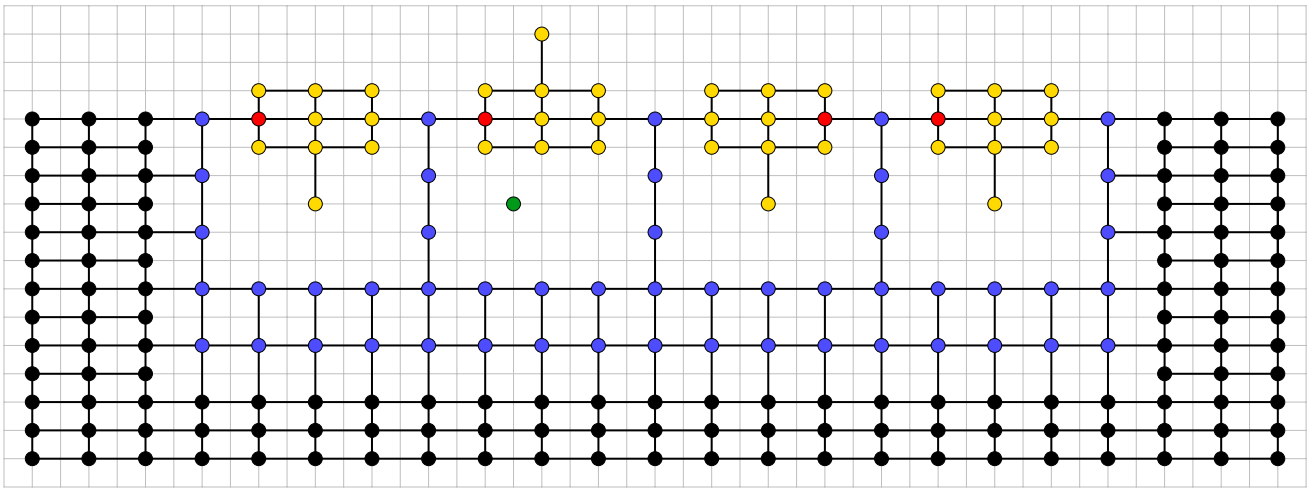

Figure 10: A clause
Here $C = x_1 \vee x_2 \vee \neg x_3 \vee x_4$, with $x_1, x_4$ set to false and $x_2, x_3$ set to true ($x_i$ from left to right).
Satisfied thanks to $x_2$, the extra disk (green) occupies the free space below the corresponding structure. There happens to be only one satisfied literal, otherwise it could be below any of them.
Variables have their $A$ node colored red,
either to the left in the case of non-negated literals, or to the right otherwise.
The outer structure (black) has only one realization excluding symmetries and rotations, and guarantees that there also is only one realization for blue disks.

# III Proof of Theorem 1

**Definition 13** (Padded model)**.**
*A model is said to be padded if it is injective and no two vertices are at distance exactly 1.*

**Notation 2.**
$D_\iota(v_1, v_2) \triangleq |\iota(v_1) - \iota(v_2)|$ *the distance according to* $\iota$ *of two vertices.*

$\varepsilon_\iota(x) \triangleq \min_{D_\iota(d,v')>x} D_\iota(d, v') - x$
*In a padded model,* $\min(\varepsilon_\iota(0), \varepsilon_\iota(1))$ *is intuitively the distance that vertices can move in both directions without any event or permutation.*
*Notice that* $\varepsilon_\iota(x) > 0$ *for any* $x$*.*

Two permutations or injective models $f, g$ are said to have the same ordering when $\forall v, v',\ f(v) < f(v') \iff g(v) < g(v')$. For two permutations this is equivalent to them being equal, but there may be several models with the same ordering.
In particular it is unambiguous to refer to "*the* permutation with the same ordering as $\iota$", (when $\iota$ is an injective model), but the other way around would be "*an* injective model with the same ordering as $\sigma$"

**Lemma 1** (Padding models)**.**
*Any model* $\iota$ *can be transformed into a padded model* $\iota'$ *in the following way:*
*if* $\sigma$ *satisfies* $\iota(v) < \iota(v') \implies \sigma(v) < \sigma(v')$ *then there exists* $\iota'$ *such that* $\iota'(v) < \iota'(v') \iff \sigma(v) < \sigma(v')$*.*

i.e. any strict ordering that does not contradict $\iota$ is realizable.

*Proof.*
Let $\sigma : V \hookrightarrow [1, n]$ a target ordering that satisfies $\iota(v_1) < \iota(v_2) \implies \sigma(v_1) < \sigma(v_2)$.

If there exists $v, v'$ such that $\iota(v) + 1 = \iota(v')$, let $\varepsilon = \min(1, \varepsilon_\iota(0), \varepsilon_\iota(1))$.
Notice that $\varepsilon > 0$ since there are finitely many pairs of vertices. Let $\iota_0$ defined as $\iota_0(v'') = \iota(v'')$ when $\iota(v'') < \iota(v')$ and $\iota_0(v'') = \iota(v'') - \varepsilon/2$ otherwise, i.e. move all vertices to the right of $v'$ towards the left by a small amount.
If $v_1, v_2$ are both such that $\iota(v_1), \iota(v_2) < \iota(v')$ then $D_{\iota_0}(v_1, v_2) = D_\iota(v_1, v_2)$.
If $v_1, v_2$ are both such that $\iota(v_1), \iota(v_2) \geqslant \iota(v')$ then $D_{\iota_0}(v_1, v_2) = D_\iota(v_1, v_2)$.
If $\iota(v_1) < \iota(v') \leqslant \iota(v_2)$ then $D_{\iota_0}(v_1, v_2) = |D_\iota(v_1, v_2) - \varepsilon/2|$
If $D_\iota(v_1, v_2) \leqslant 1$ then $D_{\iota_0}(v_1, v_2) < 1$. If $D_\iota(v_1, v_2) > 1$ then by definition of $\varepsilon \leqslant \varepsilon_\iota(1)$, $D_{\iota_0}(v_1, v_2) > 1$.
Therefore $\iota_0$ is a model of $G$.
Since $\varepsilon \leqslant \varepsilon_\iota(0)$, it follows that $\iota(v_1) < \iota(v_2) \implies \iota_0(v_1) < \iota_0(v_2)$, as well as $\iota(v_1) = \iota(v_2) \iff \iota_0(v_1) = \iota_0(v_2)$.
$\iota_0$ has at least one fewer pair of nodes satisfying $D_{\iota_0}(v, v') = 1$, by iterating this process the rest of the proof can assume that there exists no such pair.

If there exists $v, v'$ such that $D_\iota(v, v') = 0$, with $v'$ having the greatest image by $\sigma$ of all pairs satisfying this condition, then let $\varepsilon = \min(1, \varepsilon_\iota(0), \varepsilon_\iota(1))$. By the previous process, $\varepsilon > 0$.
Let $\iota_0$ differing from $\iota$ only in that $\iota_0(v') = \iota(v') + \varepsilon/2$.
All distances that do not involve $v'$ are unchanged, and all $|D_{\iota_0}(v', v'') - D_\iota(v', v'')| = \varepsilon/2 < \varepsilon_\iota(1)$ guarantee that $D_\iota(v', v'') \leqslant 1 \iff D_\iota(v', v'') < 1 \iff D_{\iota_0}(v', v'') < 1 \iff D_{\iota_0}(v', v'') \leqslant 1$.
In addition $0 < \varepsilon/2 < \varepsilon_\iota(0)$ implies that $\iota_0(v') \neq \iota_0(v'')$ for all $v''$.
By $\varepsilon \leqslant \varepsilon_\iota(0)$, it must be that $\iota(v_1) < \iota(v_2) \implies \iota_0(v_1) < \iota_0(v_2)$. It also happens that $\iota(v_1) = \iota(v') \implies \iota_0(v_1) < \iota_0(v')$ coincides with $\sigma(v_1) < \sigma(v')$.
$\iota_0$ has at least one fewer pair of nodes having $\iota(v) = \iota(v')$, iterating this process yields $\iota'$ injective and that satisfies $\iota'(v_1) < \iota'(v_2) \iff \sigma(v_1) < \sigma(v_2)$.

□

Theorem 1

*Proof.*
($\implies$) Let $\iota$ an injective model s.t. $\forall v, v',\ \sigma(v) < \sigma(v') \iff \iota(v) < \iota(v')$.
Let $v$ any vertex and $v^+ = \text{argmax}\{\sigma(v') \mid v' \in N[v]\}$ and $v^- = \text{argmin}\{\sigma(v') \mid v' \in N[v]\}$.

They must satisfy $\sigma(v^-) \leqslant \sigma(v) \leqslant \sigma(v^+)$ since $v \in N[v]$.
In addition $\forall v',\ \sigma(v^-) \leqslant \sigma(v') \leqslant \sigma(v^+) \iff \iota(v^-) \leqslant \iota(v') \leqslant \iota(v^+) \iff v' \in N[v]$.
Therefore $\sigma$ is compatible with the neighborhood of $v$. As this is valid for any $v$, $\sigma$ is compatible with neighborhoods of $G$.

($\Longleftarrow$) A connected component of size $k$ must span a distance smaller than $k$ in any model (apply triangular inequality on any path), therefore each connected component can be handled separately by spacing them by more than $n$ units of distance. The rest of the proof assumes only one connected component. By Lemma 1, a non-injective model can afterwards be transformed into an injective one, hence it can be assumed that no two vertices belong to the same class: if they do then consider them to be only one vertex for the construction that follows, then apply Lemma 1 to obtain the desired ordering.

Let $\sigma$ any ordering that is compatible with neighborhoods of $G$.
Assume a numbering $1 \leqslant i \leqslant n$, in which $v_i$ is such that $\sigma(v_i) = i$.
A model can be constructed inductively by assuming that $i < j \iff \iota(i) < \iota(j)$ as well as $\forall d, v',\ D_\iota(d, v') \neq 1$ and $\delta \triangleq \max_{d,v' \in N(v_n)} D_\iota(d, v') \leqslant 1 - 2^{-n}$.

For $n = 1$ this is easy: $\iota(v_1) \triangleq 0$ has only one vertex, and $\delta = 0 \leqslant 1 - 2^{-1}$.

Inductive step: let $i^- = \min\{j \mid v_j \in N(v_n)\}$ the smallest neighbor of $v_n$. Let $\varepsilon = \min(2^{-n}, \varepsilon_\iota(1), \varepsilon_\iota(0))$.
Choose $\iota'(v_n) = \iota(v_{i^-}) + 1 - \varepsilon/2 < 1$.

The constraint $\varepsilon < \varepsilon_\iota(0)$ guarantees that $D_\iota(v_{i^- - 1}, v_n) \geqslant 1 - \varepsilon/2 + \varepsilon > 1$.
In addition $\delta \leqslant 1 - 2^{-n+1}$ and the fact that neighborhoods are contiguous provide that $v_{i^-} \in N(v_{n-1})$ hence $\iota(v_{n-1}) \leqslant \iota(v_{i^-}) + 1 - 2^{-n+1} < \iota'(v_n)$.
The ordering is correct, and $\max_{d,v' \in N(v_n)} D_{\iota'}(d, v') \leqslant 1 - 2^{-n}$ is preserved.
A model can be thus iteratively constructed until all vertices are accounted for.

The final ordering is $\iota_f(v_i) < \iota_f(v_j) \iff i < j$. □

# IV Proof of Theorem 2

**Notation 3.**
*When $L_i \in E_i$, $L_i$ is written* LinkUp$(v_i, v_i')$.
*When $L_i \notin E_i$, $L_i$ is written* LinkDown$(v_i, v_i')$.

*$\mathcal{G}_i$ is the restriction of $\mathcal{G}$ to the first $i+1$ instants: $\mathcal{G}_i = (G_0, \cdots, G_i)$.*
*In particular $\mathcal{G}_\tau = \mathcal{G}$.*

$D_\varphi(t, v, v') \triangleq |\varphi(t, v) - \varphi(t, v')|$

**Lemma 2** (Interpolation of a discrete transition)**.**
*Let $\sigma \xrightarrow{G} \sigma'$ a discrete transition.*
*Let $\iota, \iota'$ padded models of $G$ with the orderings of $\sigma$ and $\sigma'$ respectively.*
*Then $\psi_{\iota,\iota'}$ defined as $\psi_{\iota,\iota'}(t, v) = (1-t) \cdot \iota(v) + t \cdot \iota'(v)$ is a continuous transition from $\iota$ to $\iota'$ without event.*

*Proof.*
Continuity is obvious. (Definition 8.1) is a direct consequence of $\psi_{\iota,\iota'}(0, v) = \iota(v)$, the same is true for (Definition 8.2) which comes from $\psi_{\iota,\iota'}(1, v) = \iota'(v)$.

Since both $\iota$ and $\iota'$ are padded models of $G$, $D_\iota(v, v') \leqslant 1 \iff D_{\iota'}(v, v') \leqslant 1$. Since the derivative of $\psi_{\iota,\iota'}$ for a fixed $v$ with respects to $t$ is a constant, for all pair $v, v'$, the function $f : t \mapsto D_{\psi_{\iota,\iota'}}(t, v, v')$ is monotonous, hence $f(0) \leqslant 1 \iff f(1) \leqslant 1$ yields $\forall t,\ f(t) \leqslant 1 \iff f(0) \leqslant 1$. There occurs thus no event. □

**Lemma 3** (Snapshot switch)**.**
*Given $G$ and $G'$ two snapshots differing only by the event $\{v_1, v_2\}$, $\sigma''$ an ordering compatible with both, then for any $\iota, \iota'$ padded models of $G$ and $G'$ respectively and with the ordering of $\sigma''$, the function $\psi_{\iota,\iota'}$ defined in the same way as before is a continuous transition from $\iota$ to $\iota'$ with event $\{v_1, v_2\}$.*

*Proof.*
Again, continuity is obvious, and the same argument of constant derivative applies to pairs that are not $\{v_1, v_2\}$. For $\{v_1, v_2\}$, the intermediate value theorem applied to a strictly monotonous function yields conditions 4 to 6 from Definition 8. □

**Lemma 4** (Eventless transitions)**.**
*Given two padded models $\iota$, $\iota'$ of $G$, if the continuous transition from $\iota$ to $\iota'$ is without events then $\iota$ and $\iota'$ differ only in the order of vertices within the same class.*

*Proof.*
By contrapositive, assume that $\iota(v) < \iota(v')$ and $\iota'(v) > \iota'(v')$ yet $v'' \in N[v] \setminus N[v']$ (which without loss of generality also covers the symetric case $v'' \in N[v'] \setminus N[v]$).
Let $\varphi$ be a continuous transition. By continuity, since $\varphi(0, v) < \varphi(0, v')$ and $\varphi(1, v) > \varphi(1, v')$ there must be some $t'$ for which $\varphi(t', v) = \varphi(t', v')$.
$\varphi(t', v'')$ cannot satisfy both $D_\varphi(t', v, v'') \leqslant 1$ and $D_\varphi(t', v', t'') > 1$ since $D_\varphi(t', v, v'') = D_\varphi(t', v', v'')$, hence there must be at least one event that occurs within $[0, t']$. □

Theorem 2

*Proof.*
($\Longrightarrow$) Let $\varphi$ a continuous transition from $\iota$ to $\iota'$. Let $\sigma$, $\sigma'$ the orderings of each.
Consider $t_0$ at which $|\varphi(t_0, v_1) - \varphi(t_0, v_2)| = 1$ occurs.
For any $\delta > 0$, $v \mapsto \varphi(t_0 - \delta, v)$ is a model of $G$, and $v \mapsto \varphi(t_0 + \delta, v)$ is a model of $G'$.
Therefore by continuity, $v \mapsto \varphi(t_0, v)$ must be compatible with the neighborhoods of both $G$ and $G'$.
If $v, v'$ are such that $D_\varphi(t_0, v, v') = 0$ then they must have the same neihborhood in both $G$ and $G'$. If a pair $\{v, v'\} \neq \{v_1, v_2\}$ is such that $D_\varphi(t_0, v, v') = 1$ then since no event occurs other than $\{v_1, v_2\}$, they must satisfy $D_\varphi(t_0 - \delta, v, v') \leqslant 1 \iff D_\varphi(t_0 + \delta, v, v') \leqslant 1$.

Hence $v \mapsto \varphi(t_0 - \delta, v)$ can be turned into a padded model $\iota^-$, and $v \mapsto \varphi(t_0 + \delta, v)$ can be turned into a padded model $\iota^+$ with the same ordering as $\iota^-$.
Let $\sigma''$ this ordering, then since $\varphi(0, \bullet) \to \iota^- \xrightarrow{\{v_1, v_2\}} \iota^+ \to \varphi(1, \bullet)$ it follows by Lemma 4 that $\sigma \xrightarrow{G} \sigma'' \xrightarrow{G'} \sigma'$.

($\Longleftarrow$) Let $\iota_0, \iota'_0$ padded models of the snapshots $G$, $G'$ respectively, both with the ordering of $\sigma''$.
Consider the function $\varphi$ defined as $\psi_{\iota,\iota_0}$ on $[0, 1/3]$, $\psi_{\iota_0,\iota'_0}$ on $[1/3, 2/3]$ and $\psi_{\iota'_0,\iota'}$ on $[2/3, 1]$, rescaled to a third of their original interval of definition.
By Lemma 2 and Lemma 3, $\varphi$ is a continuous transition from $\iota$ to $\iota'$ with event $\{v_1, v_2\}$. □

# V Properties of $PQ$-forests

Related to Section 4.3 which defines $PQ$-forests.

**Definition 14** (Equivalent $PQ$-forests)**.**
*Two $PQ$-forests $F_1$, $F_2$ are equivalent when $\Sigma(F_1) = \Sigma(F_2)$.*
*(reminder: Definition 12)*

**Definition 15** (Current permutations)**.**
*The current permutations $\Pi(F)$ of a $PQ$-forest $F$ are*

- $\Pi(^P(v_1, \cdots, v_n)) = \mathfrak{S}(v_1, \cdots, v_n)$
- $\Pi(^Q[c_1, \cdots, c_n]) = \{(\Pi(c_1) \cdots \Pi(c_n))\}$
- $\Pi(^F\{q_1, \cdots, q_n\}) = \mathfrak{S}(\Pi(q_1), \cdots, \Pi(q_n))$

*Notice that the only difference with Definition 12 is that children of a $Q$-node cannot be flipped.*

**Lemma 5** (Transformations that preserve equivalence)**.**
*Two forests $F_1$ and $F_2$ that have the same $P$-nodes are equivalent iff a subset of the $Q$-nodes in $F_1$ can have their children flipped around to turn $F_1$ into $F_1'$ such that $\Pi(F_1') = \Pi(F_2)$.*

*Proof.*
($\Longrightarrow$) If neither $F_1$ nor $F_2$ have $Q$-nodes with two or more children, then both are of the form $^F\{^Q[^P(\cdots)], \cdots, ^Q[^P(\cdots)]\}$. Since they have the same $P$-nodes they must differ only by the order of the children of the root. Since $\Pi(^F\{q_1, \cdots, q_n\}) = \mathfrak{S}(\Pi(q_1), \cdots, \Pi(q_n))$ then $\Pi(F_1) = \Pi(F_2)$.
Assume without loss of generality that $F_1$ has a $Q$-node $q_1 = ^Q[c_1, \cdots, c_n]$ with $n \geqslant 2$. Let $v^1, v^n$ be vertices in $c_1$ and $c_n$ respectively. If there exists $v'$ not a descendant of $c_1, \cdots, c_n$ then in all associated permutations either $\sigma(v^1) < \sigma(v^n) < \sigma(v')$ or $\sigma(v^n) < \sigma(v^1) < \sigma(v')$ or $\sigma(v') < \sigma(v^1) < \sigma(v^n)$ or $\sigma(v') < \sigma(v^n) < \sigma(v^1)$. Since this must be true of any $v^1, v^n, v'$ then it follows that $F_2$ must have a $Q$-node whose leaves are the same as those of $q_1$.
By induction on the height of the tree this shows that both forests must have $Q$-nodes that contain the same children. From there it can be shown that the corresponding $Q$-nodes must have children that are either in the same order or reversed. In the first case there is nothing to do, in the second case these nodes can be reversed.

($\Longleftarrow$) It suffices to prove that flipping the children of any $Q$-node preserves the associated permutations of the whole forest. By $\Sigma(^Q[c_1, \cdots, c_n]) = \{(\Sigma(c_1) \cdots \Sigma(c_n))\} \cup \{(\Sigma(c_n) \cdots \Sigma(c_1))\} = \Sigma(^Q[c_n, \cdots, c_1])$, the property is obvious by induction on the structure of the forest. □

**Definition 16** (Adjacent nodes)**.**
*Two elements $e_1, e_2$ of $F$ a $PQ$-forest (vertices, $P$-nodes, $Q$-nodes) are said to be "adjacent" in $\sigma$ a permutation if some vertex within $e_1$ is adjacent in $\sigma$ to some vertex within $e_2$.*
*It is said that $e_1$ and $e_2$ are adjacent (resp. can be made adjacent) in $F_q$ when there exists $\sigma \in \Pi(F_q)$ (resp. $\sigma \in \Sigma(F_q)$) such that $e_1$ and $e_2$ are adjacent in $\sigma$.*

By Lemma 5, $e_1$ and $e_2$ *can be made adjacent* iff by flipping the children of some $Q$-nodes they become adjacent.

**Definition 17** (Extremity)**.**
*$c$ a node or vertex is said to be at the extremity (resp. Right, Left) of $\sigma$ a permutation if one of its vertices is the first or last (resp. last, first) element of $\sigma$.*
*For dir one of "extremity", "Right", Left", $c$ is at (resp. can be placed at) the dir of $q$ a $Q$-node if there exists $\sigma \in \Pi(F_q)$ (resp. $\sigma \in \Sigma(F_q)$) such that $c$ is at the dir of $\sigma$.*

Again by Lemma 5, $c$ *can be placed at the extremity* of $q$ iff by flipping the children of some $Q$-nodes $c$ can become an extremity of $q$.

**Lemma 6** (Extremity of class)**.**
*Given $\sigma$, $G$, $G'$, where $G$ and $G'$ differ only by the event $\{v_1, v_2\}$. Let $\sigma'$ such that $\sigma \xrightarrow{G} \sigma'$ and $\sigma'$ is compatible with the neighborhoods of $G'$.*
*Then $\sigma'$ satisfies the additional constraint that both $v_1$ and $v_2$ are at an extremity of their class.*

This will be particularly important to show that all permutations are represented at all times, since there will be an intermediate step during which $v_1$ and $v_2$ are at an extremity of their class. By only merging together adjacent $P$-nodes, which will correspond to a permutation within a class, all remaining configurations will be obtained.

*Proof.*
Consider a permutation in which $\sigma(v_1) < \sigma(v_2)$, the other case can be handled symetrically.
If the event is a LinkUp, $v_1$ and $v_2$ cannot have the same class in $G$, hence $\sigma \xrightarrow{G} \sigma'$ must not exchange $v_1$ and $v_2$ in order to be compatible with neighborhoods of $G$. This shows $\sigma(v_1) < \sigma(v_2)$.
All $\sigma(v_1) < \sigma(v') < \sigma(v_2)$ must both initially and finally be linked with $v_1$ and $v_2$, otherwise a different event would occur.
In addition, vertices $v'$ that have the same class in $G'$ as $v_2$ (resp. $v_1$) must initially already be linked to $v_1$ (resp. $v_2$), and hence must satisfy $\sigma(v_1) < \sigma(v') < \sigma(v_2)$. Thus in $\sigma'$, $v_1$ and $v_2$ are at an extremity of their class as defined in Definition 17.

When a LinkDown occurs, this time the classes in $G'$ must be different.
If $\sigma(v_1) < \sigma(v_2)$, once again all $\sigma(v_1) < \sigma(v') < \sigma(v_2)$ must be neighbors of both $v_1$ and $v_2$. Let $v'$ that has the same class in $G$ as $v_1$. Then in both $G$ and $G'$, $v'$ is linked to $v_2$. Hence $\sigma'(v') > \sigma'(v_1)$ which yields that $v_1$ is indeed to the extremity of its class. The same applies to $v_2$.
If $\sigma(v_1) > \sigma(v_2)$ then $v_1$ and $v_2$ must belong to the same class in $G$ since they are exchanged during $\sigma \xrightarrow{G} \sigma'$. If the class contains a vertex other than $v_1$ or $v_2$, such vertex must in $G'$ be linked to both $v_1$ and $v_2$ and thus be between them. Otherwise $v_1$ and $v_2$ being the only ones in their class are necessarily at extremities.
□

# VI Algorithm

All $P$-nodes, $Q$-nodes and vertices have references to their parents that are implicitly updated when the structure of the tree is modified.
In all procedures that follow, the $PQ$-forest is understood to be implicitly passed as a global mutable variable, and so is the adjacency matrix of the current snapshot.
The input to the algorithm is the number of vertices and the sequence of events $\mathcal{L} = (L_t)_{0 \leqslant i \leqslant \tau}$.

## VI.1 Primitives list

It should be self-evident that even with a naive implementation all the following can run in polynomial time.
An additional complexity bound is estimated for an implementation where $P$-nodes are `HashSets`, the root is an array, $Q$ nodes are doubly-linked lists, and all nodes contain a pointer to their direct parent ($n$ is the number of vertices).

$P$ is the set of $P$-nodes, $Q$ is the set of $Q$-nodes. $V$ is the set of vertices.

- Parent : $(Q \to Q \cup \{\mathsf{Root}\}) \cup (P \to Q)$ $O(1)$
- IsRightMostChild : $Q \times (Q \cup P) \to$ bool $O(1)$
- IsLeftMostChild : $Q \times (Q \cup P) \to$ bool $O(1)$
- ReverseChildren : $Q \to ()$ $O(1)$
  constant time since it suffices to toggle a boolean in the parent node (see Section VIII for details)
- RemoveChild : $(\{\mathsf{Root}\} \times Q) \cup (Q \times (Q \cup P)) \cup (P \times V) \to ()$ $O(1)$
- RemoveChildIfEmpty : $Q \times P \to ()$ $O(1)$
- ToggleLink : $V \times V \to ()$ $O(1)$
  updates the global adjacency matrix, which starts off empty
- IsLink : $V \times V \to$ bool $O(1)$
  queries the adjacency matrix
- NewP : $V$ set $\to P$ $O(1)$
- NewQ : $(P \cup Q)$ doubly-linked-list $\to Q$ $O(1)$
- Reject : $() \to \emptyset$
  interrupts the execution
- Accept : $() \to \emptyset$
- Unreachable : $() \to \emptyset$
  indicates that this specific configuration cannot happen
- InsertChild : $Q \times$ index $\times (Q \cup P) \to ()$ $O(1)$
  constant time because the index contains a pointer to the location in the doubly linked list
- TransferChildren : $Q \times$ index $\times (Q \cup P)$ doubly-linked-list $\to ()$ $O(1)$
- IsNextChild : $Q \times (Q \cup P)^2 \to$ bool $O(1)$
- IsToTheLeft : $Q \times P \times P \to$ bool $O(n)$
- Neighborhood : $(P \cup V) \to$ range $O(n)$
  calculates the neighborhood **before** the link is toggled
  range is a pair of pointers to the leftmost and rightmost elements
- LeftNeighborhood : $(P \cup V) \to$ range $O(n)$
- RightNeighborhood : $(P \cup V) \to$ range $O(n)$
  does not include the $P$-node given as parameter
- LeftNeighbor : $(P \cup V) \to P$ option $O(1)$

- RightNeighbor : $(P \cup V) \to P$ option $O(1)$
- DirectNeighbors : $(P \cup V) \to P$ set $O(1)$
  simply returns the set $\{\text{LeftNeighbor}(c), \text{RightNeighbor}(c)\}$
- LeftOf : $(Q \cup P) \to$ index $O(1)$
- RightOf : $(Q \cup P) \to$ index $O(1)$
- MergeP : $P \times P \to ()$ $O(n)$
  $O(1)$ if one of the two is of size 1
- LeftMostPNeighbor : $(P \cup V) \to P$ $O(n)$
- RightMostPNeighbor : $(P \cup V) \to P$ $O(n)$
- LowestCommonAncestor : $P \times P \to (Q \cup \{\text{Root}\}) \times (Q \cup P) \times (Q \cup P)$ $O(n)$ ($O(1)$ amortized)
  The extra return values are the first different ancestors: direct children of the lowest common ancestor that are parents of the first and second $P$ nodes respectively

## VI.2 Handle LinkUp

```
1: procedure BringToExtremity(q: Q, c: Q ∪ P, dir: Right or Left)
2:     if q = c then
3:         return
4:     else
5:         a ← Parent(c)                                   ▷ a must be a Q node
6:         if dir = Right and IsRightMostChild(a, c)
7:         or dir = Left and IsLeftMostChild(a, c) then
8:             skip                                        ▷ already correctly positioned
9:         else if dir = Left and IsRightMostChild(a, c)
10:        or dir = Right and IsRightMostChild(a, c) then
11:            ReverseChildren(a)
12:        else
13:            Reject()   ▷ p is in the middle of a Q-node, it cannot be the right-most or left-most child
14:        end if
15:        BringToExtremity(q, a)                          ▷ recursively handle the parent
16:    end if
17: end procedure                                          ▷ total O(n) (O(1) amortized)
```

**Lemma 7.**
BringToExtremity *takes $q$, $c$, $dir$ and rejects if $c$ cannot be placed at the extremity of $q$, otherwise it transforms $F_q$ without changing its associated permutations, so that $c$ is to the $dir$ of $F_q$.*

*Proof.* By induction on the depth of the recursive calls.
The initialization is for $q = c$: $\Sigma(T) = \Sigma(q) = \Sigma(c)$ has $c$ itself at the edge.
Assume that $q = \text{Parent}(c')$ and $c'$ is an ancestor of $c$, and that $\text{BringToExtremity}(c', c)$ was executed, hence $c$ is to the $dir$ of $c'$.

$q = [x_1, \cdots, x_n, c', y_1, \cdots, y_m]$ with $n, m > 0$, then all elements of $\Sigma(T)$ have at least one element to the left of elements of $c'$ and another one to the right.
There is no satisfactory permutation and the algorithm rejects.

Otherwise either $q = [c', x_1, \cdots, x_n]$ or $q = [x_n, \cdots, x_1, c']$, both of which have the same associated permutations. Transforming one into the other therefore preserves the associated permutations but ensures $c'$ is to the $dir$ of $q$. Since $c$ itself is to the $dir$ of $c'$ then $c$ is to the $dir$ of $q$, which concludes. □

```
1: procedure MakeAdjacent(p1, p2: P) → (Q ∪ {Root}) × (Q ∪ P) × P × (Q ∪ P) × P
2:     lca, c1, c2 ← LowestCommonAncestor(p1, p2)                    ▷ O(n) (O(1) amortized)
                                  ▷ with c1 ≠ c2 the children of lca that are ancestors of p1 and p2 respectively
3:     if lca ≠ Root then
4:         if IsNextChild(lca, c1, c2) then
5:             skip
6:         else if IsNextChild(lca, c2, c1) then
7:             p1, p2 ← p2, p1
8:             c1, c2 ← c2, c1
9:         else
10:            Reject()
11:        end if
12:    end if
                                                  ▷ now c1 is directly to the left of c2
13:    BringToExtremity(c1, p1, Right)                               ▷ O(n) (O(1) amortized)
14:    BringToExtremity(c2, p2, Left)                                ▷ O(n) (O(1) amortized)
15:    return (lca, c1, p1, c2, p2)
16: end procedure                                                    ▷ total O(n) (O(1) amortized)
```

**Lemma 8.**
MakeAdjacent$(p_1, p_2)$ *rejects all cases where* $p_1$ *cannot be made adjacent to* $p_2$ *and that satisfy* Parent$(p_1) \neq$ Parent$(p_2)$*, otherwise it applies reversals of* $Q$*-nodes so that the associated permutations do not change but* $p_1$ *becomes adjacent to* $p_2$ *in the new forest.*

*Proof.*
Since the only operation that modifies the tree (BringToExtremity) has been proven not to change the associated permutations, it follows that MakeAdjacent preserves the associated permutations.

Note that $c_1$ and $c_2$ are well defined because Parent$(p_1) \neq$ Parent$(p_2)$ so there exists $c_1 \neq c_2$ children of $lca$ that are ancestors of $p_1$ and $p_2$ respectively.

If $lca$ is the root then there exists a permutation in which $p_1$ is adjacent to $p_2$ iff $p_1$ and $p_2$ can be made to be on the edges of $c_1$ and $c_2$.

Otherwise the $lca$ is some $Q$ node $[x_1, \cdots, x_n, c_1, y_1, \cdots, y_m, c_2, z_1, \cdots, z_k]$ or $[z_k, \cdots, z_1, c_2, y_m, \cdots, y_1, c_1, x_n, \cdots, x_1]$. If $m > 0$ then all associated permutations have at least one element that separates any element of $c_1$ (including $p_1$) from any element of $c_2$ (including $p_2$). This proves that in no permutation can $p_1$ be adjacent to $p_2$.

If $lca = [z_k, \cdots, z_1, c_2, c_1, x_n, \cdots, x_1]$ it suffices to exchange the names $p_1$ and $p_2$ so that $lca$ is of the form $[x_1, \cdots, x_n, c_1, c_2, z_1, \cdots, z_k]$ as required. □

```
1: procedure ContractQTower(c, q: Q)
2:     if c = q then
3:         return
4:     else
5:         a ← Parent(q)
6:         TransferChildren(a, at:RightOf(q), from:q)
7:         RemoveChild(a, q)
8:         ContractQTower(c, a)                          ▷ O(n) recursive calls (O(1) amortized)
9:     end if
10: end procedure                                        ▷ total O(n) (O(1) amortized)
```

**Lemma 9** (Contraction preserves the current permutations)**.**
*If* $F$ *is a forest, and* $c$ *is an ancestor of* $q$ *in* $F$*, then applying* ContractQTower$(c, q)$ *yields* $F'$ *that has the same current permutations as* $F$ *and in which children of* $q$ *are now children of* $c$*.*

*Proof.*
By induction on the depth of recursive calls.
Initialization is for $c = q$ which already satisfies the requirements.
It now suffices to show that the insertion and removal of children make it so that before the recursive call, the current permutations have not changed and children of $q$ have become children of $a$.

Let $a = {}^{Q}[x_1, \cdots, x_n, q, y_1, \cdots, y_m]$ with $n, m \geqslant 0$.
With $q = {}^{Q}[c_1, \cdots, c_k]$, the call to TransferChildren turns $a$ into ${}^{Q}[x_1, \cdots, x_n, q = {}^{Q}[], c_1, \cdots, c_k, y_1, \cdots, y_m]$ then $a' = {}^{Q}[x_1, \cdots, x_n, c_1, \cdots, c_k, y_1, \cdots, y_m]$.
The set of current permutations of $a$ is $\Pi(a) = \{(\Pi(x_1) \cdots \Pi(x_n)\Pi(q)\Pi(y_1) \cdots \Pi(y_m))\}$ which by definition is equal to $(\Pi(x_1) \cdots \Pi(x_n)\Pi(c_1) \cdots \Pi(c_k)\Pi(y_1) \cdots \Pi(y_m))$ which coincides with $\Pi(a')$.

This concludes the proof that ContractQTower does not change the current permutations of $F$ and "transfers" children to a parent $Q$-node. □

```
1: procedure ReorderLinkUp(v_1, v_2: V)
2:     p_1 ← Parent(v_1)
3:     p_2 ← Parent(v_2)
4:     assert p_1 ≠ p_2                ▷ if they were in the same P-node they would have to be already connected
5:     if Parent(p_1) = Parent(p_2) then
                                       ▷ they are within the same Q-node at the last level
6:         a ← Parent(p_1)
7:         i_1^- ← LeftMostPNeighbor(p_1)                                   ▷ O(n)
8:         i_2^- ← LeftMostPNeighbor(p_2)                                   ▷ O(n)
9:         i_1^+ ← RightMostPNeighbor(p_1)                                  ▷ O(n)
10:        i_2^+ ← RightMostPNeighbor(p_2)                                  ▷ O(n)
11:        if IsNextChild(a, p_1, i_2^-) and IsNextChild(a, i_1^+, p_2) then   ▷ case (1)
12:            RemoveChild(p_1, v_1)
13:            RemoveChild(p_2, v_2)
14:            InsertChild(a, at:RightOf(p_1), NewP((v_1)))
15:            InsertChild(a, at:LeftOf(p_2), NewP((v_2)))
16:            RemoveChildIfEmpty(a, p_1)
17:            RemoveChildIfEmpty(a, p_2)
18:        else if IsNextChild(a, i_2^+, p_1) and IsNextChild(a, p_2, i_1^-) then   ▷ case (1′)
19:            RemoveChild(p_1, v_1)
20:            RemoveChild(p_2, v_2)
21:            InsertChild(a, at:LeftOf(p_1), NewP((v_1)))
22:            InsertChild(a, at:RightOf(p_2), NewP((v_2)))
23:            RemoveChildIfEmpty(a, p_1)
24:            RemoveChildIfEmpty(a, p_2)
25:        else                                                             ▷ case (2)
26:            Reject()
27:        end if
28:    else                                                                 ▷ case (3)
29:        lca, c_1, p_1, c_2, p_2 ← MakeAdjacent(p_1, p_2)                 ▷ O(n) (O(1) amortized)
30:        if p_1 ≠ Parent(v_1) then
31:            v_1, v_2 ← v_2, v_1
32:        end if
                                       ▷ lca = Q[···, c_1 = Q[Q[···Q[···, p_1]]], c_2 = Q[Q[Q[p_2, ···]···]], ···]
33:        ContractQTower(c_1, Parent(p_1))                                 ▷ O(n) (O(1) amortized)
34:        ContractQTower(c_2, Parent(p_2))                                 ▷ O(n) (O(1) amortized)
                                       ▷ lca = Q[···, c_1 = Q[···, p_1], c_2 = Q[p_2, ···], ···]
35:        RemoveChild(p_1, v_1)
36:        RemoveChild(p_2, v_2)
37:        RemoveChild(lca, c_1)
38:        InsertChild(c_2, at:Left, NewP((v_2)))
39:        InsertChild(c_2, at:Left, NewP((v_1)))
40:        TransferChildren(c_2, at:Left, from:c_1)
                                       ▷ lca = Q[···, c_2 = Q[···, p_1 \ v_1, P(v_1), P(v_2), p_2 \ v_2, ···], ···]
41:    end if
42: end procedure                                                           ▷ total O(n)
```

**Lemma 10** (Link up)**.**
ReorderLinkUp *produces the set of permutations that are compatible with the neighborhoods of both* $G$ *and* $G'$ *if the event is a* LinkUp.

*Proof.*
**Case 1: Same $Q$ parent, adjacent**
In particular $i_1 = i_2^- - 1$ requires that $p_1$ is to the left of $p_2$.
The case (1′) is the symetrical configuration and can be handled in the same way.
$a$ must be of the form ${}^Q[\cdots, {}^P(v_1, v_1', \cdots, v_m'), c_1, \cdots, c_n, {}^P(v_2, v_1'', \cdots, v_k''), \cdots]$.
In addition since $v_2$ has all of $c_1, \cdots, c_n$ as neighbors, they must be $P$-nodes.
Since $v_1$ and $v_2$ should be connected to each other but not $v_1$ to the rest of $p_2$ or $v_2$ to the rest of $p_1$, then should a solution exist it must have $v_1$ to the right of $p_1 \setminus v_1$ and $v_2$ to the left of $p_2 \setminus v_2$.

Hence this case transforms $a$ into ${}^{Q}[\cdots, {}^{P}(v'_1, \cdots, v'_m), {}^{P}(v_1), c_1, \cdots, c_n, {}^{P}(v_2), {}^{P}(v''_1, \cdots, v''_k), \cdots]$, which represents exactly the permutations compatible with the neighborhoods of both $G$ and $G'$ and in which $v_1$ and $v_2$ are at an extremity of their class. By Lemma 6 this corresponds to all permutations compatible with both $G$ and $G'$.

**Case 2: Same $Q$ parent, not adjacent**
Since $i_j^-$ and $i_j^+$ are extremal, this case can only be reached by having either $i_1$ outside of $[i_2^- - 1, i_2^+ + 1]$ or $i_2$ outside of $[i_1^- - 1, i_1^+ + 1]$.
Assume without loss of generality that $i_1 < i_2^- - 1 < i_2$.
In all associated permutations, $i_1$ and $i_2$ are separated by $i_2^- - 1$.
Therefore it is impossible that all neighbors of $p_2$ be contiguous without reordering the nodes.

**Case 3: Different direct $Q$ parents**
If $p_1$ and $p_2$ cannot be made adjacent then it is impossible that both all neighbors of $p_1$ be contiguous and no neighborhood spans more than a single $Q$-node.
MakeAdjacent rejects in these cases.
If they can be made adjacent, then after the call to MakeAdjacent, $lca$ is of the form ${}^{Q}[\cdots, c_1 = {}^{Q}[{}^{Q}[\cdots {}^{Q}[\cdots, p_1]]], c_2 = {}^{Q}[{}^{Q}[{}^{Q}[p_2, \cdots]\cdots]], \cdots]$.
The associated permutations in which $p_1$ is adjacent to $p_2$ are exactly the current permutations and their reverse, hence the calls to ContractQTower which preserve the current permutation do not remove allowed permutations from the set of associated permutations.
The insertions and removals of children at lines 40 to 45 make $v_1$ adjacent to $v_2$, and them being at extremities of their nearest $Q$ parent guarantees that their class contains no other vertex.
Hence the forest describes all permutations in which $v_1$ and $v_2$ are at extremities of their class. □

## VI.3 Handle LinkDown

```
1: procedure ReorderLinkDown(v_1, v_2: V)
2:     p_1, p_2 ← Parent(v_1), Parent(v_2)
3:     assert Parent(p_1) = Parent(p_2)   ▷ since to be linked they must belong to the same direct Q parent
4:     q ← Parent(p_1)
5:     RemoveChild(p_1, v_1)
6:     RemoveChild(p_2, v_2)
7:     N_1^-, N_1^+ ← LeftNeighborhood(p_1), RightNeighborhood(p_1)                    ▷ O(n)
8:     N_2^-, N_2^+ ← LeftNeighborhood(p_2), RightNeighborhood(p_2)                    ▷ O(n)
9:     if N_1^- ∩ N_2^- ≠ ∅ or N_1^+ ∩ N_2^+ ≠ ∅ then                  ▷ O(1) intersection for ranges
10:        Reject()                                                    ▷ claw-like structure, case (1)
11:    else if N_1^- ∩ N_2^+ ≠ ∅ and N_1^+ ∩ N_2^- ≠ ∅ then
12:        Unreachable()                                               ▷ case (2)
13:    else if (N_1^+ = ∅ and N_2^- = ∅) or (N_1^- = ∅ and N_2^+ = ∅) then
14:        Unreachable()                                               ▷ case (3)
15:    else
                                                  ▷ determine the order between v_1 and v_2
16:        if i_1 > i_2
17:        or IsToTheLeft(q, p_2, RightNeighbor(p_1))
18:        or IsToTheLeft(q, LeftNeighbor(p_2), p_1) then
                          ▷ note that the LeftNeighbor or RightNeighbor may not even exist,
                                          ▷ in which case the comparison is false
19:            v_1, v_2 ← v_2, v_1
20:            p_1, p_2 ← p_2, p_1
21:            i_1, i_2 ← i_2, i_1
22:            N_1^-, N_2^- ← N_2^-, N_1^-
23:            N_1^+, N_2^+ ← N_2^+, N_1^+
24:        end if
                                                  ▷ v_1 can be placed to the left of v_2
25:        if N_1^- ∪ N_2^- ∪ N_1^+ ∪ N_2^+ = ∅ then                   ▷ case (4)
26:            assert p_1 = p_2
27:            i ← LeftOf(p_1)
28:            RemoveChild(q, p_1)
29:            q' ← NewQ([NewP((v_1)), p_1, NewP((v_2))]))
30:            InsertChild(q, at:i, q')
31:            RemoveChildIfEmpty(q', p_1)
32:        else                                                        ▷ case (5)
33:            InsertChild(q, at:LeftOf(p_1), NewP((v_1)))
34:            InsertChild(q, at:RightOf(p_2), NewP((v_2)))
35:            RemoveChildIfEmpty(q, p_1)
36:            RemoveChildIfEmpty(q, p_2)
37:        end if
38:    end if
39: end procedure
```

**Lemma 11** (Link down)**.**
ReorderLinkDown *produces the set of permutations that are compatible with the neighborhoods of both $G$ and $G'$ if the event is a* LinkDown

*Proof.*
**Case 1: Claw**
Assume without loss of generality that $v' \in N_1^- \cap N_2^- \neq \emptyset$, the other case can be handled by symmetry.
As $v'$ is not in the same class as $v_1$ and $v_2$, they may not switch positions during a discrete transition. Therefore after the discrete transition, both $v_1$ and $v_2$ must be to the right of $v'$. This contradicts compatibility with neighborhoods of either $v_1$ or $v_2$ since one of them separates the other from $v'$ and they are not connected.
Therefore there can be no discrete transition, which implies that there exists no temporal model and the algorithm must reject.

**Case 2: Loop**
Let $v \in N_1^- \cap N_2^+$, $v' \in N_1^+ \cap N_2^-$.
All associated permutations $\sigma$ must satisfy $\sigma(v) < \sigma(v_1) < \sigma(v') < \sigma(v_2) < \sigma(v)$ which is impossible.
Thus this case in unreachable.

**Case 3: Other incompatibilities**
Since $v_1$ and $v_2$ are linked, compatibility of neighborhoods must imply that all $v \in N_1^+ \cup N_2^+$, $v' \in N_1^- \cup N_2^-$ satisfy $\sigma(v) < \sigma(v_1), \sigma(v_2) < \sigma(v')$. If both $N_1^+ \neq \emptyset$ and $N_2^+ \neq \emptyset$ then $v_1$ or $v_2$ separates the other from its neighborhood. The same can be said if $N_1^- \neq \emptyset$ and $N_2^- \neq \emptyset$.
If $N_1^- \neq \emptyset$ and $N_1^+ \neq \emptyset$ then $v_2$ separates $v_1$ from its neighborhood. The reverse holds if $N_2^- \neq \emptyset$ and $N_2^+ \neq \emptyset$.
Therefore all of these cases are unreachable.

**Case 4: Empty left- and right-neighborhoods**
If $v_1$ and $v_2$ are both in the same class as their whole neighborhood, is particular they must share the same class hence $p_1 = p_2$.
Permutations in which $v_1$ and $v_2$ are at an extremity of their class are $(v_1, p_1 \setminus \{v_1, v_2\}, v_2)$ and $(v_2, p_1 \setminus \{v_1, v_2\}, v_1)$, hence the creation of a $Q$-node produces all possible permutations.

**Case 5: Default**
If $v_1$ is to the right of $v_2$ or $v_1$ has a right neighbor that is to the right of $v_2$ or $v_2$ has a left neighbor that is to the left of $v_1$, then exchange $v_1$ and $v_2$.
Hence the following assumes that $v_1$ is not initially to the right of $v_2$, that any left neighbors of $v_1$ are to the left of $v_2$, and that any right neighbors of $v_2$ are to the right of $v_1$.
By the previous case it also assumes that there exists at least one such neighbor.
Therefore the nearest $Q$-node is of the form either ${}^Q[x_1, \cdots, x_n, {}^P(v_1, v_1', \cdots, v_k'), \cdots, {}^P(v_2, v_1'', \cdots, v_h''), y_1, \cdots, y_m]$ or ${}^Q[x_1, \cdots, x_n, {}^P(v_1, v_2, v_1', \cdots, v_k'), y_1, \cdots, y_m]$ and in both cases all associated permutations whose neighborhood is compatible with the snapshots both before and after the event are reachable only by exchanging vertices that are in the same class from associated permutations of
${}^Q[x_1, \cdots, x_n, {}^P(v_1), {}^P(v_1', \cdots, v_k'), \cdots, {}^P(v_1'', \cdots, v_h''), {}^P(v_2), y_1, \cdots, y_m]$ and
${}^Q[x_1, \cdots, x_n, {}^P(v_1), {}^P(v_1', \cdots, v_k'), {}^P(v_2), y_1, \cdots, y_n]$ respectively. □

## VI.4 Main loop

```
1: procedure MergeClass(v: V)
2:     for n in DirectNeighbors(v) do                          ▷ O(1) iterations
3:         if Neighborhood(v) = Neighborhood(n) then           ▷ O(n)
                                  ▷ can only be true in at most one of the iterations of the loop
                          ▷ since the left and right direct neighbors of v do not have the same neighborhood
4:             MergeP(Parent(v), n)                            ▷ O(|Parent(v)|)
5:         end if
6:     end for
7: end procedure                                               ▷ Total O(n)
```

**Lemma 12** (Merge classes)**.**
*If $v$ is to the edge of its class in $G'$ after a single event $\{v, v'\}$, then* MergeClass($v$) *restores the property that the $P$-node of $v$ is exactly its class, with the possible exception of $v'$.*

*Proof.*
If only the $v - v'$ link changes state, then only the vertices $v$ and $v'$ can change neighborhoods. Since it is assumed that before the modification, the forest satisfied the condition that two vertices share the same $P$-node iff they belong to the same class, then it follows that by having their neighborhood change by at most one vertex, $v$ can now belong to the same class as at most one directly ajacent $P$-node on each side, excluding $v'$.
Hence after merging the $P$-node of $v$ with both of its adjacent $P$-nodes if they share the same class, $v$ is now in the same $P$-node as all other elements of its class, except possibly $v'$.

□

**Lemma 13** (Double merge)**.**
*In the conditions of Lemma 12, consecutive calls to* MergeClass($v$) *and* MergeClass($v'$) *fully restore the property that the P-nodes of $v$ and $v'$ are exactly their class.*

*Proof.*
Notice that if $v$ and $v'$ do not share the same class then Lemma 12 is already sufficient.
If $v$ and $v'$ do have the same final class, then either the class has another vertex $v''$ and the above lemma applied to $v - v''$ and $v' - v''$ separately yields that $v$ and $v'$ indeed share the same neighborhood after calls to both MergeClass($v$) and MergeClass($v'$), or the class is otherwise empty, in which case they must be adjacent, and will be merged as early as the first call to MergeClass($v$) (MergeClass($v'$) will afterwards be a no-op). □

---

```
1:  procedure Main(n: int, ℒ: event list)
2:      global T ← {NewQ([NewP((1))]), ··· , NewQ([NewP((n))])}   ▷ O(n) (O(1) with lazy initialization)
                                          ▷ adjacency matrix also lazily initialized to empty in O(1)
3:      for L in ℒ do                                              ▷ O(|E|) iterations
4:          match L with
5:              case LinkUp(v1, v2)
6:                  assert not IsLink(v1, v2)
7:                  ReorderLinkUp(v1, v2)                          ▷ O(n)
8:              case LinkDown(v1, v2)
9:                  assert IsLink(v1, v2)
10:                 ReorderLinkDown(v1, v2)                        ▷ O(n)
11:         end match
12:         ToggleLink(v1, v2)
                                          ▷ v1 and v2 are alone in their P-node
13:         MergeClass(v1)                                         ▷ O(1)
14:         MergeClass(v2)                                         ▷ O(1)
15:     end for
16:     Accept()
17: end procedure                                                  ▷ total O(n · |E|)
```

---

**Theorem 3** (Complete representation)**.**
*At each end of a loop, either there exists a temporal model compatible with events so far, in which case $T$ represents all possible permutations of the last embedding, or there exists no temporal model and the algorithm has rejected.*

*Proof.*
The first part inside the loop (lines 4 to 11) turns the initial $PQ$-forest into one that is compatible with both the initial and final snapshots, only by permutations of vertices that share the same class. By the previous theorems this corresponds to a discrete transition. The second part (lines 12 to 14) also corresponds to a discrete transition since it only merges adjacent $P$-nodes and hence doesn't change the ordering of vertices which do not have the same class.
By Lemma 13 the property that $P$-nodes are exactly classes is restored at the end of the loop iteration.
Both of these together guarantee that at all times the associated permutations of the forest are valid permutations of the dynamic UDG.
The subcall to ReorderLinkUp and ReorderLinkDown hence rejects configurations where there exists no temporal model.
Also notice that ReorderLinkUp has the effect of placing the two nodes in the same direct $Q$ parent, which preserves another of the announced invariants.

Since at the end of line 11 it provides all permutations compatible with both snapshots that have $v_1$ and $v_2$ as an extremity of their class, then by Lemma 6 it provides all permutations compatible with both snapshots. Theorem 2 and Lemma 13 then prove that the state of the forest at the end of the loop iteration represents all possible final orderings of a model. □

# VII Extensions to the algorithm

## VII.1 Constraints imply bounded speed

Consider a sequence $\sigma_0 \xrightarrow{G_0} \sigma_1 \xrightarrow{G_1} \cdots \xrightarrow{G_\tau} \sigma_{\tau+1}$.
There exists a temporal model $\iota_0 \xrightarrow{L_1} \iota_1 \xrightarrow{L_2} \cdots \xrightarrow{L_\tau} \iota_\tau$ such that the speed of all vertices is bounded:

$$\exists K, \forall i, \forall v,\ |\iota_i(v) - \iota_{i+1}(v)| \leqslant K$$

Take in particular $K = 1$, and consider the event $\{v_1, v_2\}$, assuming without loss of generality that $\sigma_i(v_1) < \sigma_i(v_2)$.
During the construction of a model of $\sigma_i$ in Theorem 1, positions of $v'$ such that $\sigma_i(v') < \sigma_i(v_2)$ do not change. $\sigma_i(v_2)$ itself either must be moved to the right by at most 1 if it has other neighbors. Moving all $\iota_{i+1}(v) \triangleq \iota_i(v)+1$ for $\sigma_i(v) \geqslant \sigma_i(v')$ displaces them by a bounded distance, and links can afterwards be restored only by moving them to the left. Therefore a continuous transition with a single event can induce only a bounded displacement.

This provides that as long as the frequency of events is bounded, the temporal model constructed by the algorithm can remain physically plausible.

## VII.2 Handling (or not) of simultaneous events

An argument for disallowing simultaneous events is the following: $O(k)$ events occurring simultaneously during $O(1)$ snapshots can produce temporal models in which at least one node must move by a distance $\Omega(k)$.

In other words, if an unbounded number of events can happen at the same time, there may be a vertex that is forced to acquire an unbounded speed.

Let $n > 2$, $V = \{v_1, \cdots, v_n\}$, and consider the three consecutive snapshots:

- $G_0 : \{v_i, v_j\} \in E_0 \iff |i - j| \leqslant 2$
- $G_1 : \{v_i, v_j\} \in E_1 \iff |i - j| \leqslant 1$
- $G_2 : E_2 = \emptyset$

There exists a model: $\iota_0 : v_i \mapsto \frac{i}{2}$; $\iota_1 : v_i \mapsto i$; $\iota_2 : v_i \mapsto 2i$.
Naive interpolation will indeed make events simultaneous.
Adding all inequalities of the form $|\iota_0(v_i) - \iota_0(v_{i+2})| \leqslant 1$ will provide $|\iota_0(v_1) - \iota_0(v_n)| \leqslant n/2$, and the combination of all $|\iota_2(v_i) - \iota_2(v_{i+1})|$ for $i \neq j$ as well as the ordering imposed to remain compatible with the neighborhoods of $G_1$ implies $|\iota_2(v_1) - \iota_2(v_n)| > n$.

Since in two snapshots $v_1$ and $v_n$ travel $\Theta(n)$ relative to each other, at least one of them must travel $\Omega(n)$ in absolute distance.
It leaves open the question of deciding algorithmically whether given events produce this phenomenon or not.
However, if simultaneous events occur only to vertices that share the same class, it is easy to adapt the algorithm so that ReorderLinkUp and ReorderLinkDown can handle a pair of subsets of vertices contained in one or two $P$-nodes, instead of just a single vertex. It suffices to treat all these vertices as a single one during the relevant steps. This may however make some steps of the algorithm not run in $O(1)$ anymore.

## VII.3 Initializing the algorithm with existing links

The algorithm assumes that the initial configuration has no links. However, it would be useful to be able to adapt the algorithm were that not the case.
An algorithm for creating a $PQ$-tree from an interval graph is already described in [**Consecutive_Ones**]. Although the definition therein is a bit different in that (among other things) it allows $P$-nodes to not be leaves, their construction produces (in linear time) $PQ$-trees for unit interval graphs in which each connected component has the form ${}^{Q}[{}^{P}(\cdots), \cdots, {}^{P}(\cdots)]$.

A $PQ$-forest can easily be created from these by only adding the root as a parent of the tree of each connected component.

## VII.4 Computing compatible initial configurations

The previous algorithm is only concerned with deciding whether the given temporal graph is unit interval, and it produces as a by-product a description of all possible orderings of the vertices in their final configuration.

However, in order to know the actual possible initial configurations, it is required to at least perform a second pass in reverse order of events, starting from the $PQ$-forest obtained at the end of the first pass.
It happens that this second pass is indeed sufficient to know not only all initial configurations compatible with the events that follow, but also all intermediate configurations.

Notice that if $\iota \xrightarrow{L} \iota'$ then $\iota' \xrightarrow{L} \iota$. Indeed, conditions 1 to 6 are easily verifiable for $(t, v) \mapsto \varphi(1 - t, v)$.
The same can be said of discrete transitions: $\sigma \xrightarrow{G} \sigma'$ implies $\sigma' \xrightarrow{G} \sigma$.
In particular since the final permutations are $\sigma_{\tau+1}$ such that there exists $\sigma_0, \cdots, \sigma_\tau$ for which $\sigma_0 \xrightarrow{G_0} \sigma_1 \xrightarrow{G_1} \cdots \xrightarrow{G_\tau} \sigma_{\tau+1}$, this is equivalent to the fact that $\sigma_{\tau+1} \xrightarrow{G_\tau} \cdots \xrightarrow{G_1} \sigma_1 \xrightarrow{G_0} \sigma_0$.

Two passes of the algorithm can thus determine all such possible $\sigma_\tau$ and $\sigma_0$, as well as the intermediate steps.

Similarly, the algorithm can be adapted if the initial configuration is known: instead of initializing to ${}^F\{{}^Q[{}^P(v_1)], \cdots, {}^Q[{}^P(v_n)]\}$, start with ${}^F\{{}^Q[{}^P(v_1), \cdots, {}^P(v_n)]\}$, which does not allow reorderings at the toplevel.

## VII.5 Amortized complexity analysis

Notice that BringToExtremity and MakeAdjacent are always followed by calls to ContractQTower which removes all but the last of the $Q$-nodes explored during calls to the former two functions. Hence a given $Q$-node is only ever visited by BringToExtremity and MakeAdjacent once in its lifetime, which justifies the $O(1)$ amortized time.

LowestCommonAncestor also performs as many loop iterations as the distance to the common ancestor in question, making it run in $O(1)$ amortized as well.

```
1:  procedure LowestCommonAncestor(p1, p2: P) → (Q ∪ {Root}) × (Q ∪ P) × (Q ∪ P)
2:      A1 ← HashMap[(Q ∪ Root) → (Q ∪ P)]
3:      A2 ← HashMap[(Q ∪ Root) → (Q ∪ P)]
4:      a1 ← p1
5:      a2 ← p2
6:      loop
7:          if Parent(a1) ∈ A2 then return (Parent(a1), a1, A2[Parent(a1)])
8:          else if Parent(a2) ∈ A1 then return (Parent(a2), A1[Parent(a2)], a2)
9:          else
10:             A1[Parent(a1)] ← a1;  a1 ← Parent(a1)          ▷ stop at the root
11:             A2[Parent(a2)] ← a2;  a2 ← Parent(a2)
12:         end if
13:     end loop                                  ▷ O(n) iterations (O(1) amortized)
14: end procedure                                     ▷ total O(n) (O(1) amortized)
```

## VII.6 Efficient neighborhood query

Primitives which take $O(n)$ time are only neighborhood queries (Neighborhood, LeftNeighborhood, RightNeighborhood, LeftMostPNeighbor, RightMostPNeighbor) as well as IsToTheLeft.

The improvement proposed here makes all of these run in amortized time $O(\lg n)$ at the cost of increasing the complexity of IsRightMostChild, IsLeftMostChild, RemoveChild, RemoveChildIfEmpty, InsertChild, TransferChildren to $O(\lg n)$ as well.

Organize the children of a $Q$-node as a self-balancing AVL tree ([**AVL**]) of $P$- and $Q$-nodes. Consecutive children are in order, as nodes of the AVL tree. There are no real "keys" to order the nodes by, but insertions and deletions are only ever done at extremities of the AVL tree or next to a preexisting entry to which a direct pointer is available, so the keys are not required to know where to insert the element.
This change already makes most of the functions from the previous paragraph run in $O(\lg n)$ since they are only deletions, insertions, and queries to the smallest and greatest elements.

TransferChildren is a matter of inserting an AVL tree into another, which can be done in $O(\lg n)$ by first splitting in two the outer tree in $O(\lg n)$ time, then joining the pieces also in $O(\lg n)$.

The functions IsNextChild, LeftNeighbor, RightNeighbor, DirectNeighbors still take amortized time $O(1)$ since such is the cost of accessing the next and previous elements in a balanced binary tree.
IsToTheLeft can be implemented by only exploring the two chains of parents until the nearest common ancestor, which is at a distance $O(\lg n)$ thanks to the AVL tree remaining balanced.
Neighborhood queries can all be implemented only in terms of RightMostPNeighbor and LeftMostPNeighbor:

Neighborhood($p$) = [LeftMostPNeighbor($p$), RightMostPNeighbor($p$)]
LeftNeighborhood($p$) = [LeftMostPNeighbor($p$), LeftNeighbor($p$)]
RightNeighborhood($p$) = [RightNeighbor($p$), RightMostPNeighbor($p$)]

RightMostPNeighbor is of course the symetric of LeftMostPNeighbor, and thanks to the neighborhood of each vertex being contiguous, one can perform a dichotomic search to find the extremity of the neighborhood.

```
1:  procedure RightMostPNeighbor(p: P) → P
2:      best ← p
3:      curr ← p
4:      loc ← []
5:      while AvlParent(curr) ≠ null do                          ▷ until the root
6:          if curr = AvlLeftChild(AvlParent(curr)) then
7:              loc ← Left :: loc
8:          else
9:              loc ← Right :: loc
10:         end if
11:         curr ← AvlParent(curr)
12:     end while                                                ▷ O(lg n) iterations
                                  ▷ curr is now the root, and loc indicates where we are relative to p
13:     while curr ≠ null do                                     ▷ until the leaves
14:         match IsLink(p, curr), loc with          ▷ assume that IsLink returns false if curr is a Q-node
15:             case false, Left :: rest                         ▷ too much to the right
16:                 curr ← AvlLeftChild(curr)
17:                 loc ← rest
18:             case false, Right :: rest                        ▷ too much to the left
19:                 curr ← AvlRightChild(curr)
20:                 loc ← rest
21:             case false, []                  ▷ too much to the right, can't be a Q-node
22:                 curr ← AvlLeftChild(curr)
23:             case true, Left :: rest
24:                 best ← curr                          ▷ currently to the right, new best
25:                 curr ← AvlRightChild(curr)
26:                 loc ← []                                     ▷ loc is no longer relevant
27:             case true, Right :: rest        ▷ too much to the left, can't be a Q-node
28:                 curr ← AvlRightChild(curr)
29:                 loc ← rest
30:             case true, []
31:                 best ← curr
32:                 curr ← AvlRightChild(curr)
33:         end match
34:     end while                                                ▷ O(lg n) iterations
35:     return best
36: end procedure                                                ▷ total O(lg n)
```

This procedure is essentially a dichotomy to find the edge of the neighborhood, with a small trick to handle the fact that the property is not strictly monotonous but a specific case of bitonic: when looking at all nodes in increasing order, the property is false then true then false again. Knowledge of a specific node for which the property is true is required.

The procedure is divided in 3 phases:

1. climb towards the root and record the path;
2. follow the path in the reverse direction until the first node belonging to the neighborhood;
3. from there do a dichotomy: go right if the current node is in the neighborhood, left otherwise.

Figure 11 shown a sample illustration of the path taken by the procedure, with the three phases distinguished. $p$ is the initial node, the neighborhood $N[p]$ is in orange (nodes not part of the neighborhood in black), and nodes circled red are successive values of *best*.

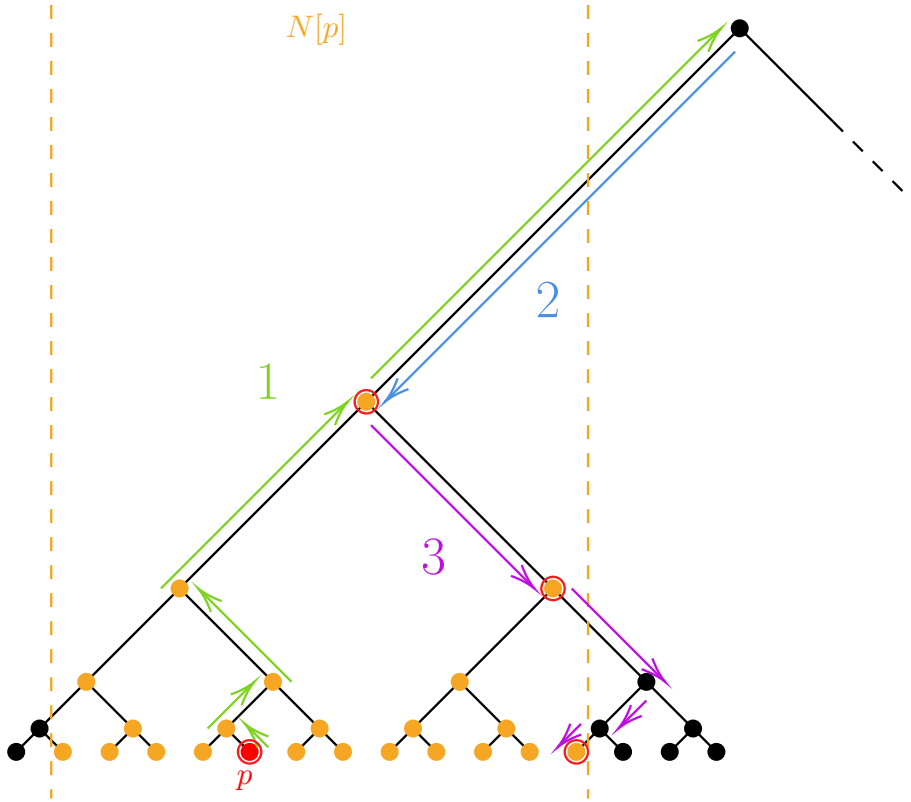


Figure 11: Three-phase dichotomy in a binary search tree for a false-true-false property

This improvement makes every single primitive function run in time at most $O(\lg n)$ amortized. The algorithm having no loops executed more than $O(1)$ times at each event, it means that handling each event takes $O(\lg n)$ amortized time.
The whole procedure therefore runs in time $O(\tau \cdot \lg n)$, which is also the expected size of the input, since each event being represented by two identifiers makes it of size $\lg n$.

With this improvement, the algorithm therefore runs in linear time, as is the case for many recognition procedures for the static problem.

# VIII Orientation

The requirement to enable flipping children of $Q$-nodes is without consequence for time complexity, but it makes implementations more difficult.

An implementation that would disregard efficiency could define $\overline{\bullet} : \text{Tree} \to \text{Tree}$ as:

$$\overline{\varepsilon} = \varepsilon \qquad \overline{\left(\begin{array}{c} A \\ \alpha \quad \beta \end{array}\right)} = \begin{array}{c} A \\ \overline{\beta} \quad \overline{\alpha} \end{array}$$

Instead, a method will be used here to simulate this behavior without having to actually execute $\overline{\bullet}$ (this would take linear time, which is too much for our purposes).
Each node of the AVL trees involved should have an `orientation` field that can be either $\leftarrow$ or $\rightarrow$.
The semantics of the order in which the nodes are is written $[\![\bullet]\!] : \text{Tree} \to \mathfrak{S}(\text{Node})$ and defined as follows:

$$[\![\varepsilon]\!] = \varepsilon \qquad \left[\!\!\left[\begin{array}{c} \overrightarrow{A} \\ \alpha \quad \beta \end{array}\right]\!\!\right] = [\![\alpha]\!]\, A\, [\![\beta]\!] \qquad \left[\!\!\left[\begin{array}{c} \overleftarrow{A} \\ \alpha \quad \beta \end{array}\right]\!\!\right] = \overline{[\![\beta]\!]} A \overline{[\![\alpha]\!]} = \overline{[\![\alpha]\!]\, A\, [\![\beta]\!]}$$

Where $\overline{\bullet} : \mathfrak{S}(\text{Node}) \to \mathfrak{S}(\text{Node})$ sends $a_1 a_2 \cdots a_{n-1} a_n$ to $a_n a_{n-1} \cdots a_2 a_1$.
If $\alpha$ is a tree, $\overleftarrow{\alpha}$ is defined as $\alpha$ where the root has had its orientation flipped. Notice that $[\![\overleftarrow{\alpha}]\!] = \overline{[\![\alpha]\!]} = [\![\overline{\alpha}]\!]$.
It must now be shown that all operations on AVL trees can be adapted to preserve semantics w.r.t. orderings of nodes. The balance factor $\text{bf}(\overleftarrow{\alpha})$ is naturally defined to be equal to $\text{bf}(\overline{\alpha}) = -\text{bf}(\alpha)$.

## VIII.1 LeftChild, RightChild and derivatives

LeftChild($n$) just needs to check $v$.`orientation`: if it is $\rightarrow$ return LeftChild($v$.`left`), otherwise return RightChild($v$.`right`).
RightChild is symetric.

All other functions that only rely on LeftChild and RightChild need not be modified (e.g. LeftMostChild, LeftMostNeighbor, IsNextChild, IsToTheLeft,...).

## VIII.2 InsertChild and TransferChildren

In the case of InsertChild, the newly created node can be given orientation $\rightarrow$ by default.

TransferChildren($q, i, c$) needs a bit more care. The actual orientation of $c$ and $i$ must be calculated relative to the lowest common ancestor of $q$ and $c$.
If the parity of the number of nodes having $\leftarrow$ is the same in the paths from $i$ to $lca$ or from the former parent of $c$ to $lca$ nothing needs to be done. Otherwise the root of the AVL tree representing $c$ should have its orientation flipped.

## VIII.3 Tree rotations

AVL trees need left and right rotations for balancing. They must be adapted to preserve ordering. Only the left rotation will be studied here, the right rotation is symetrical. There are *a priori* 4 cases to consider depending on the orientation of the root and the pivot.
Notice that all rotated trees have $A$ and $B$ oriented to the right, so what these modified rotations actually do is amortize the cost of $\overline{\bullet}$ by incrementally propagating $\overleftarrow{\bullet}$ to the leaves.

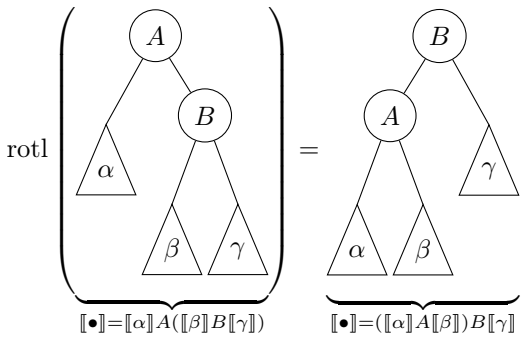


Figure 12: Left rotation in a standard binary tree

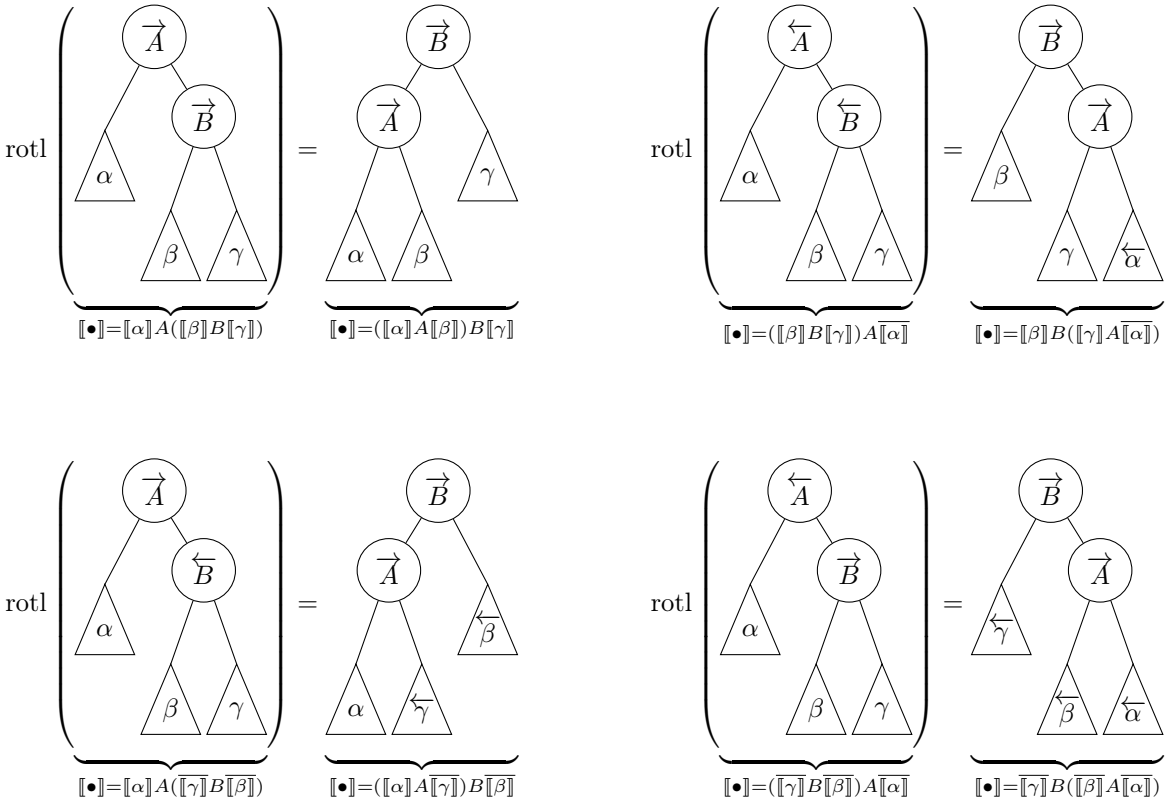


Figure 13: Left rotations adapted to oriented trees

# IX Study of forbidden patterns

It is known [**PIG_Forbidden**] that static proper interval graphs are exactly (sun, hole, net, claw)-free graphs, i.e. graphs that contain neither of the patterns shon in Figure 14 as an induced subgraph. The purpose of this section is to study how this result translates to dynamic PIG.

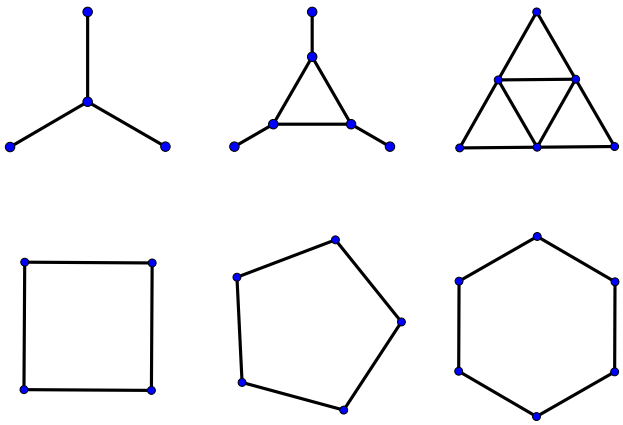

Figure 14: The forbidden static patterns
Top (left to right): claw, net, sun
Bottom: $C_4$, $C_5$, $C_6$, etc. (collectively hole or $C_{n+4}$)

**Conjecture 1.**
*All forbidden temporal patterns contain in their footprint a forbidden static pattern*

An induced subgraph $G_{V',[i^-,i^+]} = \bigcup_{i^- \leqslant i \leqslant i^+} G_{V'}$ is given by a subset $V' \subseteq V$ and a timeframe $[i^-, i^+] \subseteq [0, i_{\max}]$.
Since the focus is on temporal patterns, an additional constraint $i^+ - i^- \geqslant 1$ is added.

Section IX.1 argues that sun does not have to be counted among forbidden dynamic patterns and Section IX.2 corresponds to the four locations in the algorithm from Section VI where Reject is used. Section IX.3 is an overview of which forbidden static patterns can be footprints.
In this way the algorithm provides insights about the patterns that must not appear, but there may be a more direct characterization.

## IX.1 sun is redundant

sun can never occur alone in an induced subgraph, in the sense that if a temporal graph has a sun induced subgraph then it also has an induced subgraph with one of the other forbidden patterns.

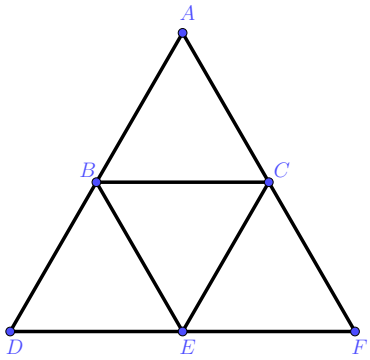


Figure 15: sun

Since these links must be created one by one, there must be either a last link added, or an extra link deleted to leave the 9 above links active.
More precisely, consider $[t_1, t_2]$ the earliest (lexicographically) time interval for which sun appears in an induced subgraph of the footprint.
Either $t_1 - 1$ contains an additional link, or the footprint of $[t_1, t_2 - 1]$ is missing one link for it to contain sun.

If $BC$ (as labeled in fig. 15 is the last link added, there is already a 4-cycle $ABEC$ and the graph has been rejected.
The same holds for $BE$ or $EC$.

If $DB$ is the last link added, there is already a claw $EDBF$ which has hence already been rejected.
The same holds for $DE$, $BA$, $AC$, $FC$, $CE$.

Otherwise all links are present and one extra link is active which must be removed.
If that link is $AE$ then there is a claw $EADF$ (also for $DC$ and $BF$).
If the extra link is $AD$ then there is a 4-cycle $ADEC$ (also for $AF$ and $DF$).

Since this covers all possible cases, there can never be an sun induced subgraph encountered without another of the forbidden static patterns.

## IX.2 Case analysis for rejections

This is a proof sketch of Conjecture 1.

**Lemma 14** (Connectivity of $Q$-nodes)**.**
*If at instant $i_0$, $v, v'$ are descendants of a common $Q$-node then they belong to the same connected component in $G_{[0,i_0]}$.*

*Proof.*
By induction on the history of the $Q$-node: either it is created at the initialization in which case there is only one disk and the property is obvious, or it is created in ReorderLinkDown case 4, in which case all of its descendants come from a single $P$-node and are thus connected.
When disks that are outside of the $Q$-node in question are added, it is only in ReorderLinkUp case 3, which merges two different $Q$-nodes. Since separately these $Q$-nodes are parts of connected components in $G_{[0,i_0]}$, the new link between one of each makes their fusion a connected component in $G_{[0,i_0+1]}$. □

**Lemma 15** (Extraction of a path)**.**
*If there exists a path from $v$ to $v'$ in $G_{[i^-,i^+]}$ then there exists a path from $v$ to $v'$ in an induced subgraph of $G_{[i^-,i^+]}$.*

*Proof.*
Consider a shortest path from $v$ to $v'$ in $G_{[i^-,i^+]}$. This path cannot have an induced $K_3$ otherwise removing one

of the vertices would shorten it. Therefore the induced subgraph with the vertices from this shortest path is a path graph. □

### BringToExtremity

In the situation of Line 13, configurations shown in Figure 16.

Assume by symmetry that $v_1$ is not at an extremity of a $Q$-node of which $v_1$ is a descendant but not $v_2$.
There must exist at least one disk on each side, name them $u$ and $u'$.
If $u$ and $u'$ are not directly connected more recently than the paths from $v_1$, then the new link from $v_1$ to $v_2$ would create a claw $v_1uu'v_2$.

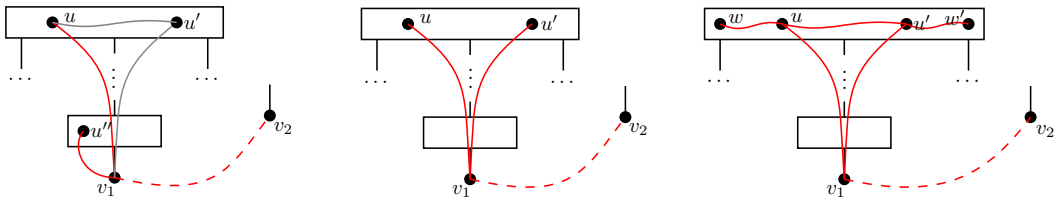


Figure 16: Case analysis for BringToExtremity Line 13

Otherwise if $u$ and $u'$ are linked, there must be other connections more recent that the paths from $v_1$ so that $u$ and $u'$ are not in the same class as $v_1$. Either $v_1$ is connected to some $u''$ which forms a claw, or $u$ and $u'$ each are connected to different $w$ and $w'$ as to not have the same class as each other and as $v_1$. These two situations form either a claw $v_1uu''v_2$ or a net $v_1v_2uwu'w'$.

### MakeAdjacent

In the situation of Line 10, configurations shown in Figure 17.

Assume $v_1$ and $v_2$ are not in the middle of any $Q$-node between them and their nearest common ancestor but there exists at least one disk between them.

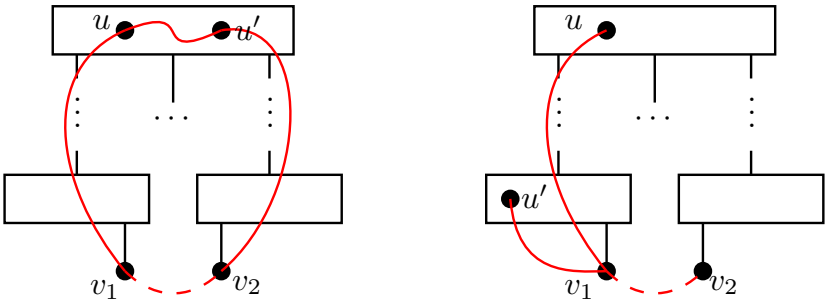


Figure 17: Case analysis for MakeAdjacent Line 10

Let $u$ such a node in the middle. In particular neither $v_1$ nor $v_2$ are connected to $u$, but there exists paths from $v_2$ to $u$ and from $v_1$ to $u$.
Assume without loss of generality that $u$ was separated from $v_2$ earlier than from $v_1$.
For $u$ not to become within the same neighborhood as $v_1$, there has to be either a disk connected to $u$ but not $v_1$, or the opposite.
In the first case, such a disk $u'$ must have at some point had a path to $v_2$, hence there exists a hole $v_1uu'v_2$.
In the second case, $v_1uv_2$ forms a claw.

### ReorderLinkUp case 2

In the situation of Line 26, configurations shown in Figure 18.

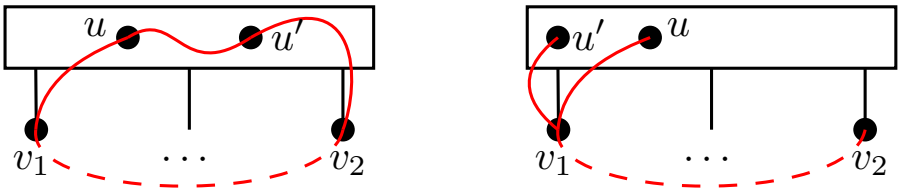


Figure 18: Case analysis for ReorderLinkUp Line 26

There exists some $u$ between $v_1$ and $v_2$. If there exists a second vertex $u'$ in between, then paths $v_2u'$, $u'u$, $uv_1$ would form a hole with the newly added link $v_1v_2$.
Otherwise, in order for $u$ to not have the same neighborhood as $v_1$ without there being an additional $Q$-node, $u$ and $v_1$ must never have had the same neighborhood. Hence there must be $u'$ neighbor of either $u$ but not $v_1$, or $v_1$ but not $u$. In the first case this forms a hole, in the second case a claw.

#### ReorderLinkDown case 1

In the situation of Line 10, configurations shown in Figure 19.

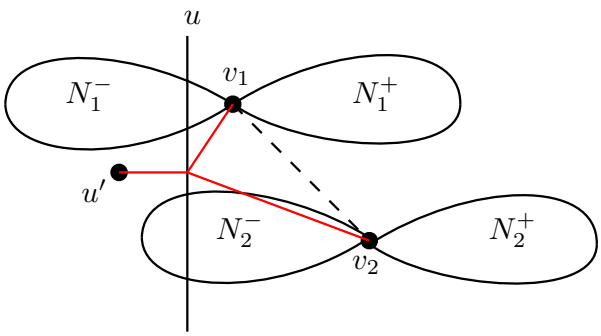


Figure 19: Case analysis for ReorderLinkDown Line 10

Calls to Reject during ReorderLinkUp are based on the structure of the tree, the single call in ReorderLinkDown however is based on the neighborhoods of the two vertices being separated. The structure of the tree is always almost the same anyway, since the two vertices belong to the same $Q$-node.

Assume by symmetry that $N_1^- \cap N_2^- \neq \emptyset$: it contains some $u$.
Since $u$ does not have the same neighborhood as $v_1$, there must be some $u'$ in $N[u] \setminus N[v_1]$.
For the neighborhood of $v_1$ to be contiguous, it must be that $u'$ is itself to the left of $u$.
Removing the link $v_1v_2$ hence produces a claw $uu'v_1v_2$.

## IX.3 Possible patterns as footprints

The previous section explains why forbidden temporal patterns seem to correspond to forbidden static patterns, but says nothing of which patterns are actually allowed.
The following is an attempt to fill some gaps by studying which forbidden patterns may or may not appear in the footprint of the whole graph.

#### claw is realizable

The temporal graph shown in Figure 20 has a claw footprint.

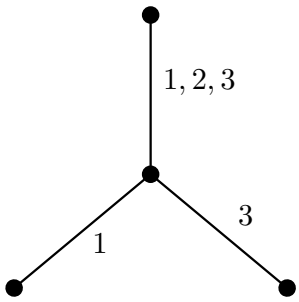


Figure 20: Labeled edges for a footprint with a claw
Numbers next to edges represent times at which they are active.

A temporal model is easy to construct, with the three extremal vertices immobile, and the middle one travelling from one side to the other.

More generally, all complete bipartite graphs $K_{m,n}$ can be footprints of dynamic PIG: two groups of $m$ and $n$ vertices crossing each other. claw $= K_{1,3}$ is a special case of this.

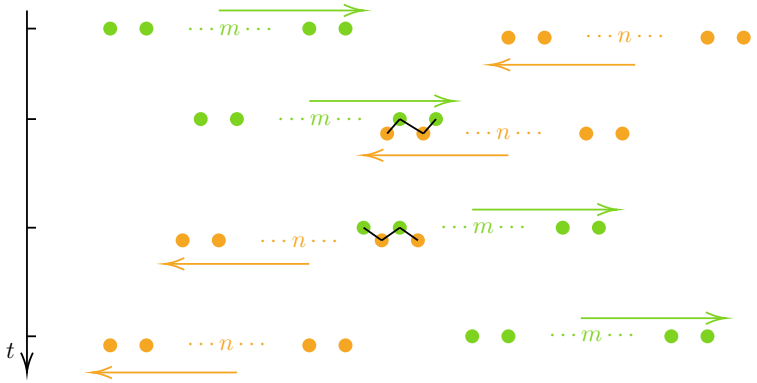


Figure 21: A sequence of movements that produces the footprint $K_{m,n}$

**net is not realizable**

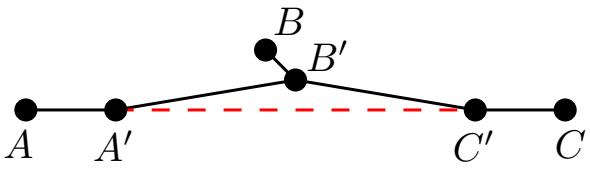


Figure 22: Only possible configuration to produce a net

Assume without loss of generality that initially $\iota(A) < \iota(B) < \iota(C)$.
Since $A$,$B$,$C$ are never in contact, they can never have the same neighborhood, and the above inequality must hold for all instants.

By contiguity of the neighborhoods of $A$, $B$, and $C$, it must be that $\iota(A') < \iota(B) < \iota(C')$, and since $A'$ and $C'$ are never in contact with $B$ it must remain so.
Moreover this implies $\iota(A') + 1 < \iota(B) < \iota(C') - 1$, which in particular makes $|\iota(A') - \iota(C')| \leqslant 1$ impossible.
Hence a net cannot be an induced subgraph of the footprint.

**sun is not realizable**

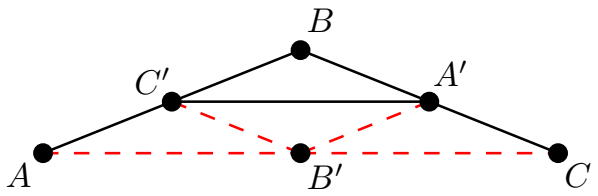


Figure 23: Only possible configuration to produce a sun

Once again assume $\iota(A) < \iota(B) < \iota(C)$.
Connections with $C'$ and $A'$ imply $\iota(A) < \iota(C') < \iota(B) < \iota(A') < \iota(C')$.
This leaves no room for $B'$ to be connected to both $C'$ and $A'$ but not $B$.

**$C_4$ is realizable**

$C_4 = K_{2,2}$ has already been shown in Figure 21 to be a possible footprint.

**$C_{n+5}$ is not realizable**

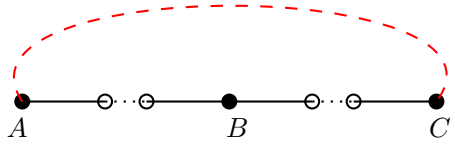


Figure 24: Only possible configuration to produce a $C_{n+5}$

Consider the leftmost and rightmost vertex $A$ and $C$. Since the cycle is of length at least 5, there exists $B$ that is in contact with neither $A$ nor $C$, but there exists paths from $A$ to $B$ and $B$ to $C$, hence $\iota(A)+1 < \iota(B) < \iota(C)-1$ at all times. This prevents the existence of a path between $A$ and $C$ that does not pass through $B$ without any of the vertices between $A$ and $C$ touching $B$.

## IX.4 Conclusion

This is far from establishing an equivalence between dynamic PIG and graphs that do not contain certain patterns, but at least it suggests that if there exists such a characterization of dynamic UDG, then it needs not use patterns that are not forbidden in the static case. It also shows that there is no obvious equivalence between forbidden static patterns and forbidden temporal patterns.