

ciForager: Incrementally Discovering Regions of Correlated Change in Evolving Graphs

JEFFREY CHAN and JAMES BAILEY and CHRISTOPHER LECKIE, University of Melbourne
MICHAEL HOULE, National Institute of Informatics

Categories and Subject Descriptors: H.2.8 [Database Management]: Database Applications-Data Mining

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Dynamic graph mining, correlated change, shortest path distance, connected components, fault detection

1. INTRODUCTION

There is growing interest in the data mining community with regard to the discovery of important patterns in dynamic graphs. The focus in this area has included analysing how global properties of graphs change with time [Leskovec et al. 2005], detecting anomalous changes in evolving graphs [Shoubridge et al. 2002], and analysing community evolution in social networks [Kumar et al. 2003].

One important and natural type of pattern for dynamic graphs is the discovery of compact subgraphs which evolve in a similar manner over time [Chan et al. 2008][Chan et al. 2009]. Such patterns are known as *regions of correlated change* and the region discovery problem involves grouping together similar sequences of changes that occur i) over the same period of time (are temporally correlated) and ii) within the same region of the graph (are spatially correlated). Each group of changes is called a **region of correlated change**. These regions provide a compact profile of the change behaviour of a dynamic graph and can allow the user to focus on significant trends, rather than being confused by a multitude of (possibly noisy) individual changes. Furthermore, further processing and linking of regions can aid in fault diagnosis, where the goal is to infer underlying ‘faults’ which are responsible for the evolution of a dynamic graph.

Application areas: One interesting potential application is in the field of medical imaging. Different parts of the human brain become excited under different stimuli [Clare 1997]. However, it may be difficult to determine which regions of the brain are changing in a significant manner, for a given sequence of stimuli. One can represent the brain as a dynamic graph, with vertices corresponding to different points on the brain, and edges corresponding to points that are close to each other. Vertices are present at a particular time if the activity level is above an excitation minimum. Seeking correlated vertex changes, a region of correlated change in this dynamic 3D

Correspondence email: jeffrey.chan@unimelb.edu.au

Author’s addresses: J. Chan and J. Bailey and C. Leckie, Department of Computing and Information Systems, University of Melbourne, Australia;

M. Houle, National Institute of Informatics, Tokyo, Japan.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© YYYY ACM 1556-4681/YYYY/01-ARTA \$10.00

DOI 10.1145/0000000.0000000 <http://doi.acm.org/10.1145/0000000.0000000>

graph then represents a set of vertices that have correlated heightened activity and are geometrically close to each other. Therefore, the discovery of regions of correlated changes is equivalent to finding cohesive regions of the brain being excited at the same time.

Another important area of application is the inference of unknown alliances and communities in online virtual worlds [Thon et al. 2008]. Finding these communities can be used to better understand the interactions and needs of players, make intelligent computer agents more realistic, making the game more enjoyable for the human players, and allow new players to better understand the social dynamics of the game world and see if they wish to join some of the existing communities and alliances.

As an example, consider an online strategy game called Travian¹ with multiple servers around the world. Each server hosts a number of game worlds, and each world consists of about 30,000 players each. Players control villages, can attack and trade with other villages, and can form alliances with each other. Because travelling long distances takes a long time, all these social activities tend to be restricted to local neighbourhoods, particularly in the case of attacks. In addition, players tend to attack as an alliance, typically against another alliance. If we represent a game world as a dynamic graph, with vertices representing the villages and edges representing an attack from one of the incident villages against the other incident village, then correlated edge behaviour within a local neighbourhood can indicate the villages of one alliance attacking the villages of another alliance. Therefore, we can find regions of correlated change in the dynamic graph representing the attacks and use these regions to discover the alliances.

A third application area is multi-layered computer networks [Steinder and Sethi 2004], where the goal is to identify underlying faults that affect the dynamic behavior of the network. We demonstrate the fault diagnosis application using the Border Gateway Protocol Internet routing network. In particular, we discover regions of correlated change representing failure events in the European parts of the network during the landfall of Hurricane Katrina in 2005.

Other potential areas of application include the tracking of groups of objects, like animal migration, in spatio-temporal databases [Elnekave et al. 2007] and detecting flash crowds at websites based on the fluctuating popularity of individual webpages accessed [Arlitt and Jin 1999].

We next provide a more concrete example, to help explain the notion of regions of correlated change. Note that we regard a region as a set of edges (as opposed to a set of vertices)².

Illustrative Example To illustrate the concept of regions of correlated change, consider an example of five sequential snapshots of a dynamic graph, shown in Figure 1. The regions of correlated change in this dynamic graph are highlighted in Figure 1f. Consider the set of edges in region A. The edges in region A are either all present, or all absent over each of the five snapshots. This is an example of correlated temporal behaviour. In addition, consider the shortest path distance between the edges in region A - all the distances are relatively close. This is an example of spatial correlation. Because the edges of region A have correlated temporal behaviour as well as spatial correlation, they form a region of spatio-temporal correlated change.

In contrast, note that the edge $e_{4,5}$ is assigned to a separate region B. Edge $e_{4,5}$ has the same temporal behaviour as the edges in region A, but it is assigned to region B because of the difference in spatial correlation. The shortest path distances from $e_{4,5}$ to

¹www.travian.com

²Structure changes in vertices induce structure changes in all the incident edges, hence there is no decrease in the information we can discover from just analysing changes to edges.

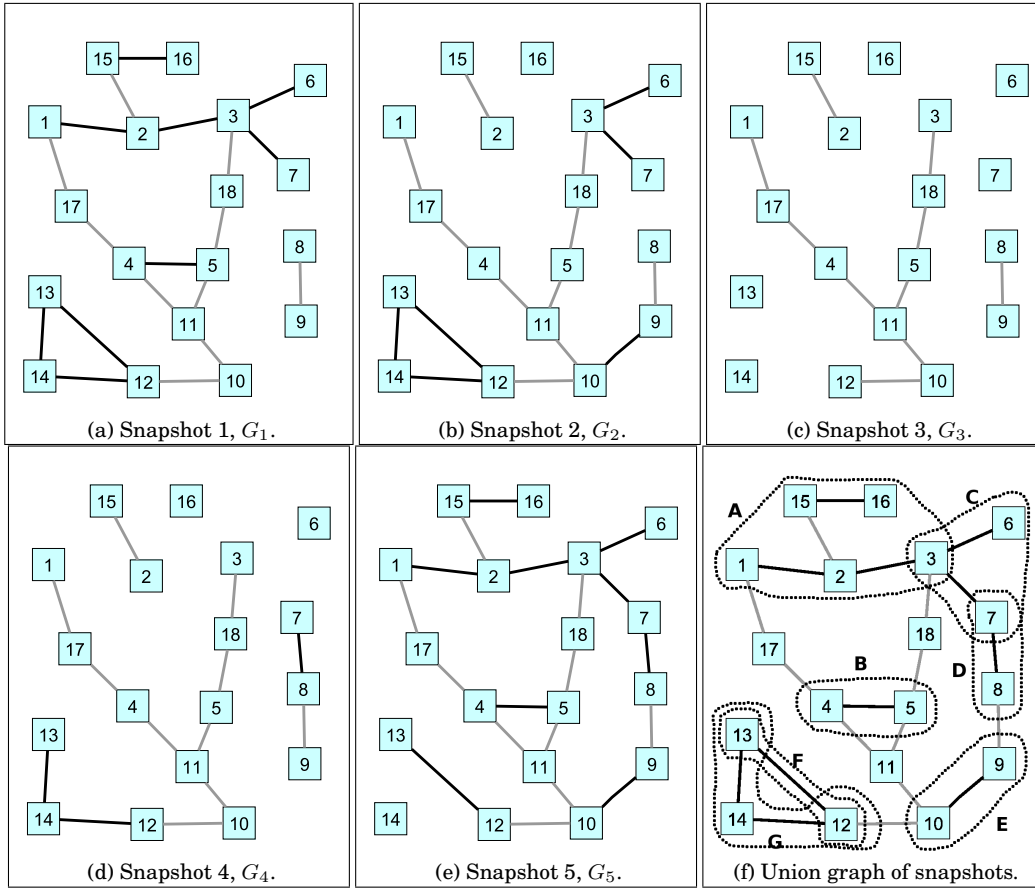


Fig. 1: An example of a dynamic graph with five snapshots. Bold edges highlight edges that have experienced change in the five snapshots. The changed edges belonging to each region are circled and labelled in Figure 1f.

the edges of region A ($e_{1,2}$, $e_{2,3}$, $e_{15,16}$) are large in comparison to the distances between $e_{1,2}$, $e_{2,3}$ and $e_{15,16}$. Although the edges of regions A and B are temporally correlated, they are not spatially correlated, and therefore form two distinct regions. The regions of Figure 1, their edge members and their change behaviours are summarised in Table I. The change behaviours are represented as waveforms. We will explain this representation in Section 4.1.

Challenges: There are two key technical challenges in discovering regions of correlated change in dynamic graphs.

- *Scalability:* Designing scalable algorithms that can efficiently process very large graphs, sampled over many time points. Scalability here refers to both time efficiency and memory efficiency.
- *Accuracy:* It is important that the regions that are discovered are of high quality. This is difficult to achieve, due to the multi-objective combinatorial nature of the problem. One must determine the optimal length of time for each region, such that the total

Region	Edge Set	Change Waveform				
		G_1	G_2	G_3	G_4	G_5
A	$\{e_{1,2}, e_{2,3}, e_{15,16}\}$					
B	$\{e_{4,5}\}$					
C	$\{e_{3,6}, e_{3,7}\}$					
D	$\{e_{7,8}\}$					
E	$\{e_{9,10}\}$					
F	$\{e_{12,13}\}$					
G	$\{e_{12,14}, e_{13,14}\}$					

Table I: Regions of correlated spatio-temporal change and their associated change waveforms of the dynamic graph from Figure 1. The change waveforms represent the change behaviour experienced by each region. We explain this representation in Section 4.1.

temporal and spatial correlation over all the regions is maximised. Furthermore, the optimal number of regions is unknown.

As we shall see, our work in this paper focuses on improving the first of these challenges - scalability.

Limitations of existing approaches: There are two existing approaches for discovering regions of correlated change. One based on greedy search [Chan et al. 2008] and the other based on global optimisation [Chan et al. 2009]. Although both these approaches can discover high quality regions, they are not able to efficiently process very large graphs. For example, neither of these algorithms is able to analyse the entire BGP routing graph or a massive web graph such as [Gibson et al. 2005]. A key bottleneck of these algorithms is that they need to repeatedly compute the temporal and topological correlation/distances between all pairs of changing edges in the network. Hence, an open problem is how to reduce the number of distance calculations in order to improve the scalability of these approaches, while maintaining their high accuracy. The increased scalability will allow analysis of very large graphs.

Our approach (ciForager): To address this important scalability problem, we propose in this paper a new algorithm called ciForager (*Incremental Change Forager*). This algorithm is specifically designed to improve the speed and scalability of region discovery, by reducing the number of distance calculations. As its name implies, part of its efficiency relies on an ability to operate in an incremental manner as new data (graphs) are processed. To reduce the number of distance calculations, ciForager introduces two new modelling concepts, *Synchronised Connected Components* and *Equivalence Classes*. Synchronised Connected Components (synCC) are sets of changing edges

that are connected to each other and have the same temporal evolution, such as edges $e_{1,2}$ and $e_{2,3}$ in Figure 1. Rather than computing the temporal and shortest path distances between all pairs of changing edges, we instead compute distances between the much smaller set of synchronised connected components. We further reduce the time to compute the temporal distances between the synCCs by the introduction of equivalence classes.

To address the second part of the problem, i.e., reducing the number of distance recalculations across time, we introduce the idea of a graph Voronoi [Erwig 2000]. Although this type of representation has previously been used in graph theory, here we use and adapt it to an entirely different context: to determine which shortest path distances between synCCs are likely to have changed due to underlying graph changes and hence need to be recomputed. The idea is to partition the graph into a number of cells around the synCCs. Each cell contains the edges whose closest synCC is in that cell. When changes occur, we can then restrict the shortest path computations to those cells that are affected by the change.

Finally, these new modelling concepts will become bottlenecks themselves if they are not efficiently computed and maintained. Hence, ciForager also proposes efficient methods to determine a) which synCCs and cells of the graph Voronoi have been affected when the underlying graph changes; and b) incrementally update the affected synCCs and cells. The new modelling concepts and the efficient incremental updating of these concepts make up the technical contributions of ciForager.

To evaluate the effectiveness of these approaches for improving the scalability of region discovery, we have evaluated ciForager on a variety of synthetic and real-life data sets. We show that for graphs where the degree of change is mostly localised, we can achieve speedups of over 10^6 times over previous work [Chan et al. 2008], whilst having comparable accuracy. We also evaluate our algorithm on graphs where changes are more widespread, such as the Border Gateway Protocol (BGP) Internet Routing and the 1998 World Cup website access graphs. In such graphs, when compared to previous work, we are able to achieve speedups of up to 70 times for ciForager, whilst using substantially less memory and achieving higher accuracy. Moreover, this strong memory efficiency also means we are able to apply ciForager to the BGP graph for the entire Internet, something which was not achievable by previous work. We demonstrate similar performance improvements when analysing a dynamic graph of web accesses to a popular website.

Contributions: In summary, the contributions of this paper are:

- We introduce a new, incremental algorithm called *ciForager* to find regions of spatio-temporal correlated change. This method introduces the concepts of synchronised connected components, equivalence classes and uses a graph Voronoi to significantly reduce the number of distance calculations and recalculations and the general running time.
- We design efficient algorithms to incrementally maintain the sets of synCCs and equivalence classes and the graph Voronoi as new graph snapshots arrive.
- Using real and synthetic data sets, we show that ciForager scales linearly with the size of the graphs analysed. In addition, we show that ciForager is from ten times to 10^6 times faster than an existing approach [Chan et al. 2008], with comparable or better accuracy and substantially reduced memory usage. This speed advantage of ciForager grows as the size of the graph analysed increases.
- We show that we can apply ciForager to discover regions of correlated change in the global BGP connectivity graph, which has until now been out of reach for previous methods.

The rest of the paper is organised as follows. In Section 2 we survey related work. Then in Section 3, the region discovery problem is formally presented. In Section 4 we provide an overview of ciForager. In Section 5, we introduce the concept of synchronised connected components. Then we describe an efficient algorithm to compute and incrementally maintain the set of synCCs for each window. In Section 6, we describe how the spatial distances between synCCs can be efficiently maintained via graph Voronoi diagrams. The time complexity of ciForager is presented in Section 7. Then in Section 8, we present an evaluation of ciForager against an existing method, using both synthetic and real datasets. Finally, in Section 9, we conclude and present possible future work.

2. RELATED WORK

In this section, we first describe the two previous approaches to finding regions of correlated change, namely cSTAG and regHunter, and highlight their strengths and weaknesses. Then we outline work in stream clustering, spatio-temporal mining and dynamic subgraph graph mining and analysis that are related to region discovery.

cSTAG (*clustering for spatio-temporal analysis of graphs*) [Chan et al. 2008] is a greedy approach. It segments a sequence of snapshots into a number of overlapping subsequences, then finds the sets of regions for each subsequence. The set of changed edges for each window are grouped into regions based on their temporal and spatial distances. To find regions that span multiple window lengths, the temporal and spatial correlation of regions in adjacent windows are computed. Those regions that are highly correlated are merged to form regions that span multiple consecutive windows. Although cSTAG is incremental in grouping the edges, it still computes all the pairwise distances among the changed edges for every snapshot in the whole snapshot sequence, regardless of whether the distance has changed or has been computed before. ciForager avoids much of this redundant and duplication distance computation using the synCC and graph Voronoi concepts.

In contrast, regHunter (*Region Hunter*) [Chan et al. 2009], employs a more global approach to discover regions in order to discover minimally separated regions that cSTAG cannot accurately find. regHunter solves the region discovery problem as a graph partitioning problem. The evolving set of changed changes are modeled as a set of vertices, and the temporal or spatial distance between changed edges are modeled as edge weights. Each partition of this time evolving graph corresponds to a region. regHunter has comparable or worse running times than cSTAG, because similar to cSTAG, regHunter also computes many duplicate distances across the snapshots. The concepts of ciForager can be used to reduce these inefficiencies.

We now outline related work in other areas. Most of this related work solves problems that are similar to region discovery problem, but are different enough such that their solutions cannot be directly used to discover regions in dynamic graphs. For example, in the area of data stream clustering [Aggarwal et al. 2003][Zhou et al. 2007], the aim is to cluster records that arrive in a continuous stream. The emphasis is to perform clustering online, while keeping memory consumption within reasonable limits. Aggregate statistics of the objects are updated online, and clusters can be produced anytime from these aggregates. However, these statistics are formulated for aggregating flat, non-relational data. It is not obvious or intuitive to extend these aggregation statistics to graphs. Hence, current stream clustering algorithms [Aggarwal et al. 2003][Zhou et al. 2007] cannot be used to solve the region discovery problem.

In spatio-temporal clustering and pattern mining, the main objective is to find objects or incidents that are in close geographical proximity, and occur frequently together. One example of spatio-temporal pattern mining is given in [Celik et al. 2006], where Celik et al. define the problem of mining mixed-drove spatio-temporal

co-occurrence patterns. These patterns are objects in spatio-temporal databases that are spatially near each other for an extended period of time.

Another example of spatio-temporal mining is [Yang et al. 2005]. Yang et al. [Yang et al. 2005] extended spatio-temporal co-location mining to scientific data. They introduced a method to discover spatio-temporal episodes, which are defined as spatial patterns that frequently occur together across a number of snapshots.

Finally, Lauw et al. [Lauw et al. 2005] used spatio-temporal co-occurring patterns to construct affiliations between people. The idea is that people or entities who are co-located frequently are mostly likely to be affiliated to each other. Places where there is a high concentration of people for a period of time are classified as events. For example, an event could be a data mining conference. Lauw et al. then uses the frequency people are at the same events to determine the affiliation between people. The more frequently two people are at the same events/places, the more likely they are to be affiliated with each other.

All these examples illustrate that spatio-temporal mining is concerned with finding clusters of entities that co-occur frequently. However, the region discovery problem is concerned with groups of entities that report strong temporal change correlation over a window of time, which may or may not co-occur frequently. For example, in the application of localising brain excitation, a part of the brain can be excited for a short burst of time. These excitations do not last long and are not very frequent, but the affected parts do produce the bursts around the same time. Hence, these excitations can be detected as a region of correlated change, but will not be considered as a frequent, co-located cluster.

In dynamic subgraph mining and analysis, subgraphs of interest are extracted. Similar to discovering regions of spatio-temporal correlated change, the emphasis is on extracting patterns in graphical data. However, each of the works in this area have one or two significant differences with the region discovery problem. We shall highlight each of their differences in the following paragraphs.

Borgwardt et al. [Borgwardt et al. 2006] defined the novel problem of finding frequent subgraphs in dynamic graphs. In addition to the traditional definition of being topologically frequent, these subgraphs must also exhibit similar temporal evolution over a period of time. Although very similar to our work, the frequent subgraphs sought by Borgwardt do not have any topological or spatial constraints, whilst in our work, we require changed edges to be topologically close. For example, in finding faults in computer network, two areas of the network that is caused by different faults but having same change signature will be incorrectly classified as one by Borgwardt's definition, while correctly classified as two areas of change by the region definition.

In [Lahiri and Berger-Wolf 2010], Lahiri and Berger-Wolf proposed the problem of mining frequent, periodic subgraph patterns from dynamic graphs. These subgraph patterns are periodic with a certain frequency, span a continuous period of time (i.e., no gaps in the periodicity), are maximal in size and in time (i.e., the subgraph pattern will no longer be a valid pattern if a vertex is added or the time spanned is extended), and in the period of time it spans, it must appear a minimum number of times (minimum support). The authors proposed an efficient pattern tree that keeps track of all potential patterns, and demonstrated some interesting patterns like periodic movement of a herd of zebras and the appearance of the same set of actors at an annual awards show. Finding periodic subgraph patterns is an appropriate step towards finding general correlated subgraph patterns, but it can be considered as a specialisation of finding the general regions of correlated changes.

Bogdanov [Bogdanov et al. 2011] recently proposed the problem of mining dynamic heavy subgraphs. Heavy subgraphs are subgraphs that are have maximal total edge weights, and dynamic heavy subgraphs are heavy subgraphs that span a continuous

interval of time. As an example to motivate the mining of heavy subgraphs, the same framework can be used to identify heavy, short traffic congestion spikes, or moderate congestion that persist for long time. An algorithm called MEDEN was proposed to find a single dynamic heavy subgraph per interval. Although similar to finding regions of correlated changes, MEDEN only identifies one subgraph candidate, and has no concept of correlation in edge weight changes.

In [Kumar et al. 2003], Kumar et al. explored the evolution of community structure and behaviour in several collections of weblogs. They introduced the notion of a time graph to model the evolution of a collection of weblogs and the links between them. Edges in the time graph are labelled with their creation time. Using these time graphs, they extracted weblog communities and analysed the degree distribution, evolution of node distributions, and burstiness of communities. In [Kumar et al. 2006], they performed a similar analysis on the structure and evolution of two online social networks, namely Flickr and Yahoo! 360. Similar to Leskovec et al. [Leskovec et al. 2005], their focus is on analysing how the frequency distributions of vertex and local substructures evolve with time and then designing generative models to replicate the observed distributions. In addition, these global based graph analysis do not analyse how correlated the changes are, nor find regions whose change is correlated across a limited span of time.

In [Ali et al. 2005], Ali et al. introduced the idea of phenomena detection and tracking (PDT) in sensor networks. PDT involves detecting sensors that are geographically near to each other and have abnormal readings over a certain time period. The motivating example was to track oil spills from a sensor network deployed at sea. Again, although similar in aim to our work, the PDT algorithm involves finding areas where the readings are all high. It does not analyse how the readings change nor whether the reading changes are correlated.

Sun et al. [Sun et al. 2006] introduced tensor analysis to dynamic graphs. Tensors are basically a generalisation of a matrix to multiple dimensions. Tensor decomposition techniques can reveal important factors that explain the underlying structure and/or variance observed. Sun et al. proposed two efficient algorithms to perform the tensor decomposition. A dynamic graph can be considered as a 3 dimensional tensor (adjacency matrix and time). As such, it can be used to find prominent regions that can explain most of the structure or variation seen in the evolving graph. However, it is more difficult for such an approach to find regions that might not be prominent yet possibly interesting (e.g., an anomalous attack pattern that tries to remain hidden).

Similar to [Sun et al. 2006], Du et al. [Du et al. 2010] employed matrix decomposition in the idea of EigenNetworks. Similar to our work, EigenNetworks represents and tracks the evolution of edges as a matrix of edges and their activity across time (each entry in the matrix has a value of '1' if the corresponding edge had some activity in that time instance, '0' otherwise). Then singular value decomposition is applied to determine the edge groupings and the activity levels of each group. Analysing the energy values of the singular values gives an indication of the fluctuations in the activity of a group, while each grouping allows their subgraphs and the evolution in their graph neighbourhoods to be tracked across time. EigenNetworks is a very useful analytic tool, but it is different from finding regions of correlated change because it does not model the connectivity information, i.e., which edges are connected to each other.

In [Sun et al. 2007], Sun et al. proposed an information theoretic algorithm to discover communities in evolving, relational graphs. Communities are defined as groups of objects (vertices) that have the same set of connections among themselves over a period of time. Using lossless encoding schemes, groups of objects are encoded, and the ones with the smallest entropy (i.e., are most homogeneous or have the same connections among themselves over a continuous span of time) are chosen as the communi-

ties. In theory, it is possible to design an encoding scheme to find regions of correlated changes. However, unlike finding communities, it is much more difficult to design an efficient coding scheme and optimising heuristic for regions of correlated change. For example, if a region consists of a change pattern of present, absent, absent, absent, present, then to find this pattern, all dense subgraphs of all sizes would need to be kept in the initial snapshot, then the number of candidates that could be extended with absent edges could be very large for sparse graphs. This can become very expensive very quickly. However, there is promise in this approach, and certainly worth considering in future work.

Similar to [Sun et al. 2007], Chi et al. [Chi et al. 2007] extended spectral clustering to finding evolving subgraph communities. Spectral approaches are a popular and successful method for partitioning static graphs into communities. To extend spectral clustering to dynamic graphs, Chi et al. used the idea of temporal smoothing from evolutionary clustering [Chakrabarti et al. 2006]. Evolutionary clustering is an incremental approach, where at each time snapshot, clusters are found that, on one hand, are similar to their previous history, but on the other hand, maximise cluster quality for the current snapshot. Similar to [Sun et al. 2007], the main focus of [Chi et al. 2007] is the discovery of dynamic communities and not the discovery of regions of correlated change.

In summary, the two previous solutions [Chan et al. 2008][Chan et al. 2009] to discovering regions in dynamic graphs cannot scale to very large graphs because of their inefficiencies and redundant computations. In Section 4, we introduce ciForager that tackles this efficiency challenge. In addition, each of the areas of stream clustering, spatio-temporal mining and dynamic subgraph mining and analysis have similarities with the region discovery problem. However, each of them have one or more significant differences with the region discovery problem. Hence algorithms developed in these areas cannot be directly used to solve the region discovery problem.

3. PROBLEM STATEMENT

In this section, we formally define what is a region of spatio-temporal correlated change and the problem of discovering these regions. For ease of reference, Table II provides a summary of the main symbols used in this paper.

Definition 3.1. A graph $G(V, E)$ consists of a set of vertices V , and a set of edges E , $E : V \times V$, representing the pairwise relationships over V .

Definition 3.2. A **dynamic graph** is represented as a sequence of snapshots $\langle G_1, \dots, G_t, \dots, G_T \rangle$ of the graph G , $1 < t < T$. Note that T can be infinity. A subsequence (or window) $\langle G_{ts}, \dots, G_{te} \rangle$ is denoted by $W^{ts,te}$, $1 \leq ts \leq te \leq T$. Let wn be the number of windows. For fixed length windows, we also denote a window by $W_{[k]}$, $1 \leq k \leq wn$, where the index k is ordered in ascending temporal order; i.e., the first snapshot of $W_{[k]}$ is earlier than the first snapshot of $W_{[k+1]}$. Note that the index k does not designate the starting point of a window. Finally, if both the window ordering and the snapshots spanned are important, then the notation $W_k^{ts,te}$ is used.

For example, the window $W^{1,2}$ for the graph illustrated in Figure 1 represents the sequence $\langle G_1, G_2 \rangle$. If $wn = 4$, and the set of windows is $\{W^{1,2}, W^{2,3}, W^{3,4}, W^{4,5}\}$, then W_1 is $W^{1,2}$, W_2 is $W^{2,3}$, and so on.

Definition 3.3. A **structural change** of an edge e is defined as the appearance or disappearance of e between any two consecutive graph snapshots. We call an edge that has experienced any structural change a **changed edge**, and we define $E_C^{ts,te}$ (E_C^k) as the set of changed edges over window $W^{ts,te}$ (W_k).

Symbol	Description
$G(V, E)$	A graph, with vertex and edge set V and E .
$W^{ts,te}(W_k)$	A window of snapshots $\langle G_{ts}, \dots, G_{te} \rangle$, with index k .
$E_C^{ts,te}(E_C^k)$	Set of changed edges over window $W^{ts,te}(W_k)$.
$q^{ts,te}(e_i)$	The change waveform of edge e_i over window $W^{ts,te}$.
$R_r^{ts,te}$	A region of spatio-temporal correlated change, defined over $W^{ts,te}$.
\mathbf{R}	A set of regions of spatio-temporal correlated change.
ω	The sliding window size.
T	The length of the total sequence of graph snapshots.
synMG	The acronym for a synchronised maximal graph.
synCC	The acronym for a synchronised connected component.
s_i	An synCC with label i .
\mathcal{S}_k	The set of synCCs for window W_k .
eq_i	An equivalence class with label i .
\mathcal{EQ}_k	The set of equivalence classes for window W_k .
$cell_i$	The cell of synCC s_i .
$N^k(i)$	The set of synCC neighbours of synCC s_i .
$B^k(i, j)$	The set of boundary vertices and edges between synCCs s_i and s_j .
$SPD^k(i, j)$	The shortest path distance between synCCs s_i and s_j .
U_k	The number of synCC operations required to update \mathcal{S}_k to \mathcal{S}_{k+1} .
$ V_{\mathcal{S}_k Avg} $	The average size of the synCCs in \mathcal{S}_k .

Table II: Summary of the main symbols and parameters used in this paper.

For example, the edge $e_{3,7}$ for the graph shown in Figure 1 is a changed edge over $W^{1,2}$. The set of changed edges for $W^{1,2}$, $E_C^{1,2}$, is $\{e_{1,2}, e_{2,3}, e_{4,5}, e_{9,10}, e_{15,16}\}$.

A **region of spatio-temporal correlated change** $R_r^{ts,te}$, $R_r^{ts,te} \subseteq E_C^{1,T}$, is a set of changed edges that have the following characteristics:

- Over the snapshots spanning ts to te , all edges are highly correlated in their change behaviour, as well as being topologically close.
- The temporal and spatial distances within each region is minimised.

Intuitively, the edges within a region should have minimal temporal and spatial distances between them, while edges in different regions should have maximal temporal and/or spatial distances between them. For example, in the computer network fault diagnosis context, let there be two simultaneous faults occurring: an unstable router, and a cut physical link. All connections that transit through an underlying, unstable router will experience similar, flapping changes and be near each other in the connectivity graph. All connections that transit through the cut link will experience a failure

change and be near each other. The two sets of connections are temporally and spatially similar within themselves, and temporal and possibly spatial different from each other. Another example is in the localisation of brain excitations. Each excited portion of the brain will be excited at different periods hence have different high temporal difference and be in different areas of the brain, and hence have high spatial distances. Therefore to use regions to localise faults and brain excitations, we define a region as having minimal temporal and spatial distance among its member edges.

Next we define the relations d_{tem} and d_{spa} , the objectives f_{tem} and f_{spa} and the region discovery problem.

Definition 3.4. The temporal relation $d_{tem} : E \times E \times W_k \rightarrow [0, 1]$ measures the difference between the temporal change behaviour of pairs of edges over the window W_k . The spatial relation $d_{spa} : E \times E \times W_k \rightarrow [0, 1]$ measures the topological distance between pairs of edges over the window W_k .

In this paper, we represent the temporal behaviour of an edge by a waveform. In unweighted graphs, the waveform is a sequence of '0' and '1', where '1' means the edge was present at that time instance, '0' means absent. To measure the difference, we use the modified euclidean distance, first proposed in Definition 6 of [Chan et al. 2008]. For the spatial distance measure, we use the shortest path distance between the edges, computed over the union graph of the snapshots in the window. These are described in more detail in Section 4.1

Definition 3.5. Let the **set of regions of spatio-temporal correlated change** be denoted by $\mathbf{R} = \{R_1^{ts_1, te_1}, \dots, R_L^{ts_L, te_L}\}$, where $R_r^{ts_r, te_r} \subseteq E_C$, L is the number of regions defined over the whole snapshot sequence $W^{1,S}$ and a changed edge can only belong in one region at any particular time.

Definition 3.6. Given \mathbf{R} and d_{tem} , the temporal objective function, $f_{tem}(\mathbf{R}, d_{tem})$ is defined as

$$f_{tem}(\mathbf{R}, d_{tem}) = \sum_{R_r^{ts_r, te_r} \in \mathbf{R}} \sum_{e_i, e_j \in R_r^{ts_r, te_r}} d_{tem}(e_i, e_j, W^{ts_r, te_r})$$

where $f_{tem}(\mathbf{R}, d_{tem})$ measures how temporally correlated the regions \mathbf{R} are, under the temporal distance relation d_{tem} . The spatial objective function $f_{spa}(\mathbf{R}, d_{spa})$ is similarly defined.

This measure sums up the temporal (spatial) distances within each region, and minimising it would minimise the temporal and spatial distances among the member edges of each region.

Definition 3.7. Given a sequence of snapshots $\langle G_1, \dots, G_t, \dots, G_T \rangle$ and its set of changed edges $E_C^{1,T}$, the problem of discovering regions of spatio-temporal correlated change is to partition $E_C^{1,T}$ into a **set of regions of spatio-temporal correlated change** $\mathbf{R} = \{R_1^{ts_1, te_1}, \dots, R_L^{ts_L, te_L}\}$, to minimise the objective functions $f_{tem}(\mathbf{R}, d_{tem})$ and $f_{spa}(\mathbf{R}, d_{spa})$.

From its definition, the region discovery problem is a bi-objective optimisation problem. In general, it is infeasible to optimise multiple objectives simultaneously, hence there is no "standard" definition for multi-objective optimisation. Different formulations are used depending on the application. Hence we intentionally present the problem using this general form. For example, in [Chan et al. 2008][Chan et al. 2009], different formulations such as optimising a single weighted sum of f_{tem} and f_{spa} were introduced and evaluated. We describe formulations in Section 4.1. The focus of this

paper is to improve the overall efficiency of the two existing algorithms, hence we use the same formulations and weights as in [Chan et al. 2008]. In Section 8.3.1, we evaluate different approaches and different weightings and their effect on the running time and accuracy.

4. OVERVIEW OF CIFORAGER

The ciForager algorithm improves the distance calculation efficiency of the two existing frameworks, cSTAG [Chan et al. 2008] and regHunter [Chan et al. 2009]. ciForager achieves this via the three new modeling concepts introduced in Section 1, and the efficient incremental maintenance of these concepts. The new concepts are:

- (1) the concept of *synchronised connected components* (synCC), designed to reduce the number of pairwise temporal and spatial comparisons;
- (2) the concept of *equivalence classes*, designed to reduce the number of pairwise temporal comparisons;
- (3) the concept of a *graph Voronoi diagram*, designed to reduce the number of spatial recalculations triggered by new snapshots;

The concepts are centred around using synchronised connected components to represent regions of correlated change. In the evaluation of cSTAG and regHunter [Chan et al. 2008][Chan et al. 2009], we found regions of correlated change can be broken down into a number of disjoint groups of edges that form connected components and experience the same temporal evolution over the region's lifespan. From the definition of a synchronised connected component, these edge groups are synchronised connected components (synCCs). Therefore, a region of correlated change can be considered as a set of synCCs. For example, in the fault diagnosis example presented earlier, the edges in the region representing the unstable router would have the same unstable change behaviour, and are likely to form a connected component as many of the connections have the same incident vertices. Therefore, we can group the edges together into a synCC and avoid computing the spatial or temporal distances among the edges. To further illustrate the idea of a synCC, consider region A of Figure 1, which consists of three changed edges: $e_{15,16}$, $e_{1,2}$ and $e_{2,3}$. But region A can also be considered as consisting of two synchronised connected components (synCC): one synCC consisting of the edge $e_{15,16}$, and the other consisting of edges $e_{1,2}$ and $e_{2,3}$.

Hence we can reduce the problem of discovering regions from grouping a set of changed edges to grouping a set of synchronised connected components. Since the number of synCCs is usually far smaller (about a reduction of 100 to 1000 times), the time to compute the pairwise temporal and spatial distances is reduced. In addition, we can also reduce the time to cluster/partition. As we will show in our worst case complexity analysis (Section 7) and empirical evaluation (Section 8), the time savings from calculating distances between synCCs outweigh the time required to construct and maintain the synCCs.

The ideas of equivalence classes and graph Voronoi further enhance the speedups using synchronised connected components. Although the introduction of synCCs reduces the number of temporal distance calculations, there are still redundant calculations. Multiple synCCs can have the same change waveform/equivalence class because there can be groups of edges that have the same change evolution but form multiple connected components. Restricting temporal distance calculations among unique waveforms or equivalence classes will further reduce the calculation time. For example, in the detection of flash crowds in web access graphs, there are many changes, particularly of the type where the users access the website then leave after a period of time. Therefore, many of the change waveforms are the same, and the idea of only

computing distances among the equivalence classes will reduce the time to compute the temporal distances.

A graph Voronoi divides a graph into a number of cells. As we show in Section 6, shortest path distance recalculations can be restricted to only the ones that pass through cells affected by new changes. The synCCs themselves do not restrict the shortest path distance recalculations; the addition of the graph Voronoi and its cell structure help restrict these recalculations. As an example of the potential speed improvements achieved by restricting recalculations, consider the localisation of brain excitations. Each part of the brain has specific functionalities, hence the excitations due to different activities are very localised. Therefore, many of the existing synCCs and the shortest paths between them do not change, and so using a graph Voronoi to restrict recalculations to affected distances will improve the speed.

In addition, the ciForager algorithm incrementally maintains and updates the synchronised connected components, equivalence classes and graph Voronoi. This is to prevent the maintenance of these structures becoming a bottleneck. We show in Section 7 that we can restrict updates to these structures to only those affected by new changes. This ensures the updating can be performed in linear time to the number of synCCs.

The ciForager algorithm improves the overall efficiency of both the cSTAG and regHunter algorithms. Both algorithms clusters the set of changing edges, hence can benefit from the idea of synchronised connected components. Both frameworks calculate temporal and spatial distances, hence can benefit from the equivalence class and graph Voronoi concepts. However, due to the similarity of the required implementation, we demonstrate how ciForager improves the distance calculation efficiency of the cSTAG framework in this paper. The procedure and efficiency benefits of extending regHunter are similar to those for cSTAG. In the following subsections, we first introduce the algorithm of cSTAG, then how ciForager extends cSTAG.

4.1. cSTAG Algorithm

cSTAG [Chan et al. 2008] divides the sequence of snapshots into a sequence of consecutive, overlapping windows of snapshots. Regions of correlated change are discovered for each window, then these locally discovered regions are merged to form regions that span multiple windows.

The temporal relation d_{tem} used in cSTAG, and regHunter, is based on representing change behaviour of edges as change waveforms. More formally,

Definition 4.1. For structural changes to edge e_i , over the subsequence $W^{ts,te}$, the changes are represented by a binary valued **change waveform** $q^{ts,te}(e_i) = q(e_i)[1]q(e_i)[2] \dots q(e_i)[te - ts + 1]$, where

$$q(e_i)[k] = \begin{cases} 0 & e_i \notin G_{ts+k-1} \\ 1 & e_i \in G_{ts+k-1} \end{cases}, 1 \leq k \leq te - ts + 1$$

As an example, the change waveforms for each changed edge in Figure 1 are shown in Table I.

Using this representation, any existing waveform measure can be used to compute the temporal distance between the edges. The measure used in [Chan et al. 2008] is the *modified Euclidean* distance, which is the Euclidean distance measure that also takes the shape of the waveforms into consideration. The spatial relation used in cSTAG was the shortest path distance measure as it is a general measure of topological proximity.

To discover the local regions for each window, a bi-objective clustering method is used to partition the set of changed edges for that window, using the temporal and spatial distances between the edges. There are two parts to the bi-objective cluster-

ing: a) incorporating both distance measures; and b) grouping the changed edges into regions.

Several methods were proposed in [Chan et al. 2008] to combine and incorporate the distances, including *hard modification*, *soft modification* and *sequential*. In *hard modification*, the temporal and spatial distances are combined into one measure, based on using one of the distance measures as a constraint. If the constraining distance is less than a user set threshold, then the combined distance is set as the other distance. Otherwise, it is set to the maximum distance, usually 1 if distances are normalised to range $[0, 1]$. In *soft modification*, the temporal and spatial distances are combined as a weighted linear sum. In *sequential*, also known as lexicographical optimisation, one of the distance measures is used to initially partition the regions. Then the other distance measure is used to further partition the regions discovered in the first partitioning step.

To group the changed edges into local regions, a clustering method was used. These include *leaderFollower*, *singleLinkage* and *averageLinkage* clustering algorithms. *leaderFollower* [Chan et al. 2008] is an incremental clustering algorithm that inserts each new point into an existing cluster if its average distance to the points in that cluster is below some user-specified threshold. New clusters are created for new points that do not satisfy the threshold for all existing clusters. *singleLinkage* and *averageLinkage* are the single and average linkage (hierarchical) agglomerative clustering algorithms [Jain and Dubes 1998].

The last step is to merge the local regions across windows. cSTAG computes the inter-region distances between the discovered regions of adjacent windows, and merges regions that have average inter-region temporal and spatial distances below user specified minimum similarity thresholds.

Figure 2a shows the process diagram for cSTAG.

4.2. ciForager Algorithm

ciForager uses the concepts of equivalence classes, synchronised connected components and graph Voronoi diagrams to reduce unnecessary calculations during region discovery. In Figure 2, we show the process diagrams of cSTAG and ciForager side by side. Figure 2b shows the process diagram of ciForager, and how the new concepts are incorporated. ciForager maintains the overall structure of cSTAG and regHunter. The main differences are associated with calculating the temporal and spatial distances and clustering/partitioning. These differences are highlighted by the gray shaded boxes in Figure 2b. We elaborate on these differences in the following paragraphs.

After the waveforms are updated in the *Update waveforms* step, ciForager incorporate the new changes into the current set of synCCs and equivalence classes in step *Update synCCs, equivalence classes* (we elaborate the process in Section 5). Then we compute the temporal distances among the updated equivalence classes (step *Compute temporal distance*). The updates in the synCCs are also used to update the graph Voronoi in the *Update graph Voronoi* step (Section 6). Then the shortest path distances that have changed are recomputed in step *Compute SPD*. After all necessary distances are recomputed, the updated distances are used to cluster the current set of synCCs in step *Cluster synCCs*. Finally, the member edges of each synCC are extracted in step *Extract edges* because the region association step is inherited from cSTAG and requires regions to consist of edges, not synCCs. The rest of the steps are the same between ciForager and cSTAG.

In the following sections, we describe the key concepts behind these steps in more detail. We first formally define the concepts of an equivalence class and a synchronised connected component, and describe how it can be efficiently maintained. Then we define the concept of a graph Voronoi diagram, and explain how they can be efficiently

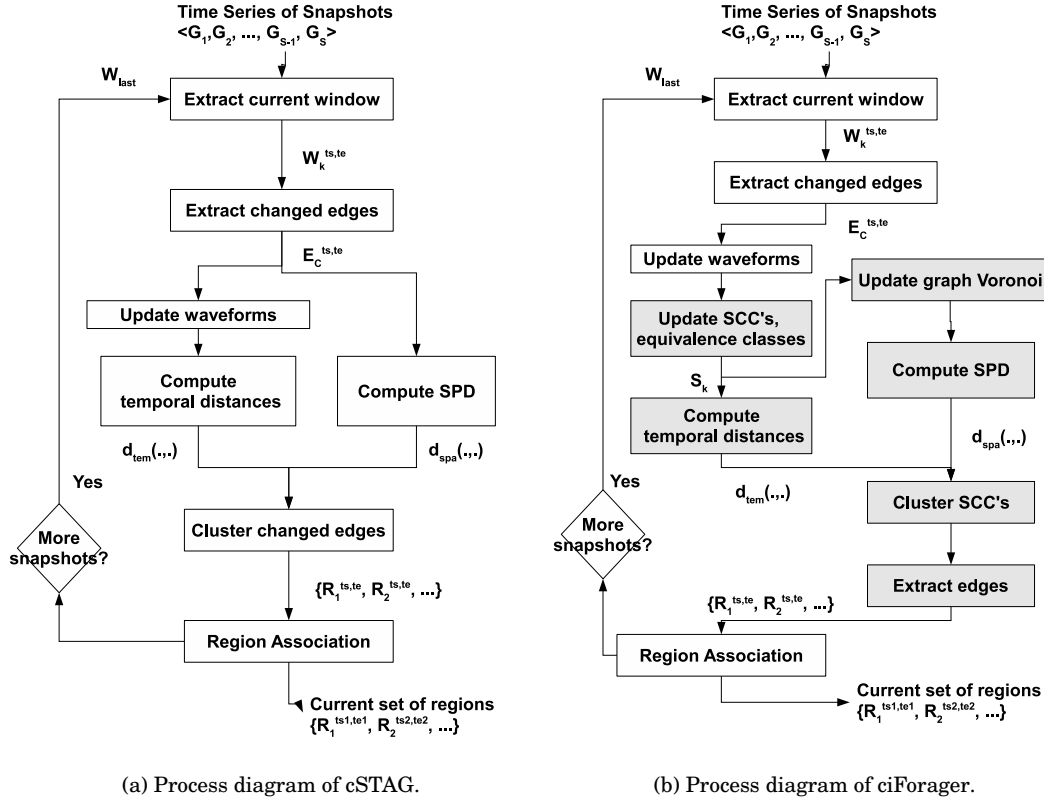


Fig. 2: Process diagram of cSTAG and ciForager. The boxes shaded in gray in Figure 2b highlight the main process differences between cSTAG and ciForager.

maintained and used to compute shortest path distances. Finally, we present the time complexity of ciForager.

5. EQUIVALENCE CLASSES AND SYNCHRONISED CONNECTED COMPONENTS (SCC)

In this section we first describe the concept of an equivalence class. To simplify the description of discovering and maintaining a set of synchronised connected components (synCC), we then introduce the more general concept of *Synchronised Maximal Subgraphs* (synMG). Finally, we formally introduce the synchronised connected component concept and describe the algorithms used to incrementally maintain a set of synMGs and a set of synCCs.

Definition 5.1. An **equivalence class** is a unique waveform among the set of change waveforms of a window. A changed edge is linked to an equivalence class if it has the same change waveform as the equivalence class.

We now introduce the concepts of a synchronised maximal subgraph, a connected component and a synchronised connected component.

Definition 5.2. A **synchronised maximal subgraph** (synMG) for a window W_k is a set of changed edges. It is associated with exactly one equivalence class of W_k . An

edge belongs to the synMG if it is linked to the synMG's associated equivalence class. The set of synMGs for window W_k is denoted by \mathcal{S}_k .

Definition 5.3. A **connected component** of a graph G is a subgraph G' such that for any pair of edges $e_1, e_2 \in E_{G'}$, it is possible to find a connected path between them in G . In addition, the connected component G' is maximal in size, i.e., it contains all edges in G that satisfy the connected path property.

Definition 5.4. A **synchronised connected component** (synCC) for window W_k is a set of changed edges such that a) all the edges in the synCC are linked to the same equivalence class, and b) the edges form a maximal connected component, over the union graph of the snapshots in W_k . Note that the concept of an synCC is a special case of an synMG, but not vice versa.

For an example illustrating a synchronised maximal subgraph (synMG), a synchronised connected component (synCC) and their difference, consider Figure 1. The edges in regions C and F together form one synMG, because the edges in $C \cup F$ consist of all the changed edges with the same change waveform (i.e., they are in the same equivalence class). However, the edges in regions C and F form two separate synCCs, $\{e_{3,8}, e_{3,7}\}$ and $\{e_{12,13}\}$, because the edges form two different connected components. Next, we describe how the set of synMGs can be incrementally maintained.

5.1. Incrementally Maintaining the Set of Synchronised Maximal Subgraphs

A naive approach to discovering and maintaining the set of synchronised maximal subgraphs (synMGs) is to discover the set of changed edges for each window, then find the maximal groups of edges with the same equivalence class. But since consecutive windows are overlapping, the sets of synMGs can be much more efficiently discovered and maintained using an incremental approach. In this subsection, we first examine the operations that are required to maintain the set of synMGs, then state a lemma that shows that these operations are sufficient to maintain the set of synMGs. With some minor modification, the same set of operations are sufficient to maintain a set of synCCs. Then in the next subsection, we shall introduce an algorithm that implements these operations to efficiently maintain the set of synMGs and synCCs as new snapshots arrive.

Consider the following set of possible operations on a set of synMGs.

Definition 5.5. Shrink: Let \mathcal{EQ}_k denote the set of equivalence classes for window W_k . Given an equivalence class $eq_j \in \mathcal{EQ}_k$ defined over $\langle G_{ts}, G_{ts+1}, \dots, G_{ts+\omega-1} \rangle$, the shrink operation, $\text{shrink}: \mathcal{EQ}_k \rightarrow \mathcal{Q}$ (\mathcal{Q} is a set of waveforms), drops its earliest element; i.e., $eq_j[0]$ is deleted. After shrinking, the length of eq_j reduces to $\omega - 1$ and eq_j is subsequently defined over $\langle G_{ts+1}, G_{ts+2}, \dots, G_{ts+\omega-1} \rangle$.

Note that a shrunk equivalence class might not be a unique waveform anymore, hence the range of the shrink operation is the set \mathcal{Q} of waveforms of length $\omega - 1$.

Definition 5.6. Extend: Given an equivalence class eq_j defined over $\langle G_{ts}, G_{ts+1}, \dots, G_{ts+\omega-1} \rangle$ and a value $y \in \{0, 1\}$, the extend operation, $\text{extend}: \mathcal{EQ}_k \times \{0, 1\} \rightarrow \mathcal{EQ}_{k+1}$ appends y to the end of eq_j . After extension, the length of eq_j increases by one and eq_j is subsequently defined over $\langle G_{ts}, G_{ts+1}, \dots, G_{ts+\omega-1}, G_{ts+\omega} \rangle$.

As an example of the extend and shrink operations, consider Figure 3. Figure 3 shows an example of an equivalence class (labelled original), and the result of extending, shrinking and applying both operations (labelled extension, shrink and extension + shrink respectively).

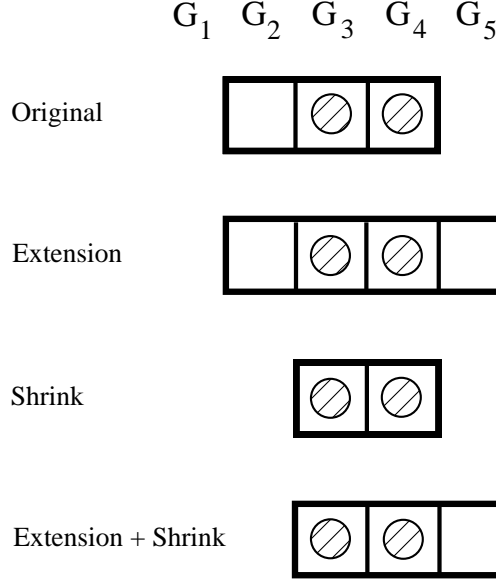


Fig. 3: An example of the result of shrinking and extending an equivalence class. Each rectangular block represent a change waveform, and the circles within represent the value of '1' and blank squares represent '0'. The original waveform of the equivalence class spans the snapshots G_2 , G_3 and G_4 . After extension, it spans G_2 to G_5 . After shrinking, it spans G_3 to G_4 . And after extension and shrinking, it spans G_3 to G_5 .

Definition 5.7. Merge: Given two sets of edges, s_i and s_j , that are associated with the same equivalence class eq_i , the merge operation, $\text{merge}: \Delta \rightarrow S_k$ (Δ is a set of edge sets), combines the two groups into one synMG, s_c , where $s_i \cup s_j = s_c$. The equivalence class of s_c is set to eq_i .

Definition 5.8. Split: Given a set of edges, s_p , where a subset of the edges, $s_i \in s_p$, is associated with an equivalence class eq_i of \mathcal{EQ}_{k+1} , the other subset, $s_j \in s_p$, is associated with a different equivalence class eq_j of \mathcal{EQ}_{k+1} , and s_i and s_j are disjoint sets, the split operation, $\text{split}: \Delta \rightarrow S_{k+1}$, partitions s_i into the two separate synMGs s_j and s_l , where $s_p = s_i \cup s_j$.

Before presenting the theorem that shows that these four operations are sufficient to maintain a set of synMGs or synCCs, we need to introduce an alternative notation for equivalence classes. The alternative notation defines the time alignment of two equivalence classes.

Definition 5.9. An equivalence class can alternatively be represented using a “list” like notation. An equivalence class eq_i of length ω can be represented as $x|_tAy$, where $x, y \in \{0, 1\}$ are equivalence classes of length 1, and A is an equivalence class of length $\omega - 2$. x denotes the value of eq_i at snapshot G_{t-1} , y the value at snapshot $G_{t+\omega-1}$, and A is defined over the subsequence $\langle G_t, G_{t+1}, \dots, G_{t+\omega-2} \rangle$. Note that using this temporal alignment notation, $x|_tAy$ can also be equivalently denoted by $|_{t-1}xAy$ or $xA|_{t+\omega-2}y$. In addition, where it is unambiguous to do so and only an indication of the temporal alignment is required, $|A$ and $|B$ will be used to denote $|_tA$ and $|_tB$ respectively.

Now we present a main theorem showing that the four operations are sufficient to maintain a set of synMGs or a set of synCCs.

THEOREM 5.10. *Let the set of synMGs for window W_k and W_{k+1} be denoted by S_k and S_{k+1} respectively. Then between two consecutive windows $W_k = \langle G_{ts}, \dots, G_{ts+\omega-1} \rangle$ and $W_{k+1} = \langle G_{ts+1}, \dots, G_{ts+\omega} \rangle$, the operations shrink, extend, merge, and split are sufficient to update S_k to S_{k+1} .*

PROOF. To prove Theorem 5.10, we first construct a process to update S_k to S_{k+1} that only uses the four operations. This process consists of an intermediate state S_{int} , where S_{int} is the set of synMGs over $\langle G_{ts}, \dots, G_{ts+\omega} \rangle$ (i.e., $W_k \cup W_{k+1}$), and it involves two sub-processes. The first sub-process applies the extend, then split operation on the synMGs of S_k to obtain the synMGs of S_{int} , and the second sub-process applies the shrink, then merge operation on the synMGs of S_{int} to obtain the synMGs of S_{k+1} . The process can be summarised as:

$$S_k \xrightarrow[\text{split}]{\text{extend}} S_{int} \xrightarrow[\text{merge}]{\text{shrink}} S_{k+1}$$

As S_k , S_{int} and S_{k+1} are assumed to be correct and complete over their respective windows, we just need to show the process is correct and complete. To show the process is correct and complete, hence proving Theorem 5.10, we need to show the sub-processes are correct and complete.

We add the synMGs associated with equivalence classes of all '0' or '1' values to S_k , S_{int} and S_{k+1} ; i.e., those equivalence classes that do not change. This special case is handled separately in the implementation, but it simplifies the proof to consider them as being part of those synCCs.

Consider the first sub-process, S_k to S_{int} , which uses the extend and split operations. Consider two synMGs in S_{int} , $s_{\beta 0}$ and $s_{\beta 1}$, associated with equivalence classes $x|A0$ and $x|A1$ respectively, where $x \in \{0, 1\}$ and A is a waveform of length $\omega - 1$. All the edges in $s_{\beta 0} \cup s_{\beta 1}$ must have once been linked to the equivalence class $x|A$. Let the synMG associated with $x|A$ be denoted by s_i ; therefore all the edges in $s_{\beta 0} \cup s_{\beta 1}$ must also have once belonged in s_i . The synMG s_i must exist and be a synMG of window W_k , and hence must be in S_k , as $x|A$ is a valid equivalence class over W_k . This shows every synMG in S_{int} must have originated from a synMG in S_k .

To obtain $s_{\beta 0}$ and $s_{\beta 1}$ from s_i , the change waveform of the edges of $s_{\beta 0}$ are *extended* with '0' to form the equivalence class $x|A0$ and the change waveform of the edges of $s_{\beta 1}$ are *extended* with '1' to form the equivalence class $x|A1$. Then s_i is *split* into $s_{\beta 0}$ and $s_{\beta 1}$. This shows every synMG in S_{int} can be obtained from a synMG in S_k by extension and splitting.

Consider the second sub-process, S_{int} to S_{k+1} , which uses the shrink and merge operations. Consider a synMG in S_{k+1} , s_α , associated with equivalence class Ay , $y \in \{0, 1\}$. The edges in s_α must have been linked to either $x|A0$ and $x|A1$ (depending on the value of y), which are the equivalence classes of $s_{\beta 0}$ and $s_{\beta 1}$. Therefore, the edges in s_α must have belonged in either $s_{\beta 0}$ or $s_{\beta 1}$, and $s_{\beta 0}$ and $s_{\beta 1}$ must be synMGs in S_{int} . This shows every synMG in S_{k+1} must have originated from a synMG in S_{int} .

To obtain s_α from $s_{\beta 0}$ or $s_{\beta 1}$, the equivalence classes of $s_{\beta 0}$ and $s_{\beta 1}$ are first *shrunk*, resulting in $x|A$. Then the edges in $s_{\beta 0}$ and $s_{\beta 1}$ are *merged* to form s_α , since both sets of edges have the same equivalence class after shrinking. This shows every synMG in S_{k+1} can be obtained from a synMG in S_{int} by shrinking and merging. \square

5.2. Algorithm to update the set of synMGs

Algorithm 1 describes the algorithm to update the set of synMGs and the set of equivalence classes given the existing sets of equivalence classes and synMGs and the new

changes. The algorithm assumes that: a) the waveforms of all the edges have been extended with their new values for the snapshot $G_{t,s+\omega}$; and b) the set of edges that have appeared or disappeared (*appeared* and *disappeared* respectively) between the last two snapshots and are not part of any existing synMGs have already been extracted; and c) the initial set of synMGs, S_k , is available. Assumptions a) and b) can be easily performed prior to each invocation of Algorithm 1. Assumption c) is always valid except for the initial set of synCC S_1 . S_1 can be obtained via bootstrapping, as it is a special case of Theorem 5.10. S_1 can be obtained from extending and splitting the synMGs obtained for the first $\omega - 1$ snapshots, which in turn can be obtained from extending and splitting synMGs obtained from the first $\omega - 2$ snapshots, and so on until the base case of two snapshots. The set of synMGs over the initial two snapshots can be obtained by snapshot comparison. Thus S_1 can be obtained by bootstrapping from the set of synMGs obtained from the first two snapshots.

ALGORITHM 1: Updating of the set of synMGs and equivalence classes for window W_k to the set of synMGs and equivalence classes for W_{k+1} . S_k and \mathcal{EQ}_k are updated inplace.

Input: S_k - the set of synMGs associated with window W_k .

\mathcal{EQ}_k - the set of equivalence classes for window W_k .

appeared - edges that have appeared but are not in any of the existing synMGs.

disappeared - edges that have disappeared but are not in any of the existing synMGs.

Output: S_{k+1} - the set of synMGs associated with window W_{k+1} .

\mathcal{EQ}_{k+1} - the set of equivalence classes for window W_{k+1} .

```

1 // Extend the equivalence classes. This might cause some synMGs to be split.
2 for each  $s_j \in S_k$  do
3   if  $s_j$ .edges are all extended with the same value  $y$  then
4     | Extend  $s_j$ .eq with  $y$ ;
5   else
6     | split( $s_j$ );
7   end
8 end
9 eqHash = {};
10 // Shrink all equivalence classes and neighbourhood vectors in  $\mathcal{EQ}_k$ .
11 for each  $eq_j \in \mathcal{EQ}_k$  do
12   shrink( $eq_j$ );
13   if  $eq_j$  is a constant equivalence class then
14     |  $\mathcal{EQ}_k$ .delete( $eq_j$ );
15   end
16   eqHash.insert( $eq_j$ );
17 end
18 // Create new synMGs
19 nScc1 = newScc(appeared, eq01);
20 nScc2 = newScc(disappeared, eq10);
21  $S_k \leftarrow \{nScc1, nScc2\}$ ;
22 eqHash.insert(eq01); eqHash.insert(eq10);
23 // Determine which equivalence classes need to be merged
24 for each key  $eq_j$  in eqHash do
25   if |eqHash[ $eq_j$ ] > 1 then
26     | merge(eqHash[ $eq_j$ ]);
27   end
28 end

```

Algorithm 1 performs each of the four operations of Theorem 5.10 in turn. It first extends the equivalence classes of all existing synMGs. Those synMGs that are associ-

ated with two extended equivalence classes are then split. After that, the equivalence classes are shrunk, and those synMGs associated with constant equivalence classes (i.e., do not possess a change anymore) are deleted. New synMGs are then created, and any synMGs that have the same equivalence classes are merged.

5.3. Maintaining a Set of Synchronised Connected Components (synCC)

In this section, we examine how to extend Algorithm 1 to maintaining a set of synCCs. Then we evaluate the complexity of this synCC algorithm.

First, consider a lemma. Recall that the connected components and shortest path distances are computed over the graph resulting from the union of the snapshots in a window. This property leads to the following lemma.

LEMMA 5.11. *After extension, an existing synCC s_j can only split due to its member edges been linked to different equivalence classes.*

PROOF. See Appendix A. \square

Lemma 5.11 shows that an existing synCC can only split because new changes cause its member edges to belong to different equivalence classes. This means the algorithm does not have to test whether topological changes to the edges will cause any existing synCCs to split. However, this case does not apply to merging. To merge two existing synCCs, the equivalence classes and whether the two sets of edges of the synCCs form a connected component after merging have to be evaluated.

The simplest method to test whether two sets of edges form a connected component after their union is to run a connected component algorithm over the union set of edges. The time complexity of this is linear in the number of edges in the merged synCCs.

The other difference with the algorithm to maintain the set of synMGs is that each equivalence class can be associated with multiple synCCs. No significant changes are necessary to handle this difference, except for some implementation details relating to splitting. When splitting, an additional check must be performed to test if an equivalence class already exists. If so, we do not create a new equivalence class, but link the existing one to one of the newly split partitions/synCCs. Algorithm 2 illustrates the process to maintain the set of synCCs. It is similar to Algorithm 1, apart from lines 2–12 and 22–30 of Algorithm 2 replacing the lines 2–8 and 18–20 of Algorithm 1.

5.3.1. Complexity

LEMMA 5.12. *The worst case complexity to update S_k to S_{k+1} (Algorithm 2) is $O(|E_C^k|) + O(|V_{S_k, Avg}| \cdot |S_k|) + O(|E_C^{k+1}|)$.*

PROOF. We analyse each step of Algorithm 2 and show that the total complexity to update S_k to S_{k+1} is $O(|E_C^k|) + O(|V_{S_k, Avg}| \cdot |S_k|) + O(|E_C^{k+1}|)$.

First, each of the edges in the synCCs of S_k are extended with either a value of '0' or '1' (lines 3 to 8 of Algorithm 2). Extending a change waveform is a $O(1)$ operation and there is a total of $|E_C^k|$ number of extensions, hence the complexity of extension is $O(|E_C^k|)$.

For splits (line 10), in the worst case, each extension of a synCC results in a split. Each split requires finding the connected components among the two resulting set of edges. Let s_p be the set of edges in synCC, and s_0 and s_1 denote the two sets of edges after the split (i.e., $s_0 \cup s_1 = s_p$). Using the disjoint set-find-union data structure [Cormen et al. 2001], the cost to determine the connected components is $O(\mu)$, where μ is the size of a set of edges in which the connected components are sought. Therefore the complexity to perform a split is $O(s_0) + O(s_1) = O(s_p)$. As it is difficult to analytically determine the size of synCCs without making assumptions about the

ALGORITHM 2: Updating of the set of synCCs and equivalence classes for window W_k to the set of synCCs and equivalence classes for W_{k+1} .

Input: S_k - the set of synCCs associated with window W_k .

\mathcal{EQ}_k - the set of equivalence classes for window W_k .

appeared - edges that have appeared but are not in any of the existing synMGs.

disappeared - edges that have disappeared but are not in any of the existing synMGs.

Output: S_{k+1} - the set of synCCs associated with window W_{k+1} .

\mathcal{EQ}_{k+1} - the set of equivalence classes for window W_{k+1} .

```

1 // Extend the equivalence classes associated with  $S_k$ . This might cause some synCCs to be split.
2 for each  $s_j \in S_k$  do
3   if  $s_j$ .edges are all extended with the same value  $y$  then
4     if  $s_j$ .eq extended with  $y$  exists already then
5       link  $s_j$ .eq to existing extended equivalence class;
6     else
7       extend  $s_j$ .eq with  $y$ ;
8     end
9   else
10    split( $s_j$ );
11  end
12 end
13 eqHash = {};
14 // Shrink all equivalence classes.
15 for each  $eq_j \in \mathcal{EQ}_k$  do
16   shrink( $eq_j$ );
17   if  $eq_j$  is a constant equivalence class then
18     |  $\mathcal{EQ}_k$ .delete( $eq$ ); delete associated synCCs;
19   end
20   eqHash.insert( $eq_j$ );
21 end
22 // Create new synCCs
23 Conna = connectedComponents(appeared);
24 for each  $cc \in Conn_a$  do
25   |  $S_k \leftarrow$  newSCC(cc, eq01);
26 end
27 Connb = connectedComponents(disappeared);
28 for each  $cc \in Conn_b$  do
29   |  $S_k \leftarrow$  newSCC(cc, eq10);
30 end
31 eqHash.insert(eq01); eqHash.insert(eq10);
32 // Determine which equivalence classes need to be merged
33 for each key  $eq_j$  in eqHash do
34   if |eqHash[ $eq_j$ ] > 1 then
35     | merge(eqHash[ $eq_j$ ]);
36   end
37 end

```

underlying graph structure and the distribution of changed edges, we use the average size of synCCs, $|V_{S_k Avg}|$, to parameterise this aspect. Therefore the total complexity of performing splits is $O(|V_{S_k Avg}| \cdot |S_k|)$.

Next, the equivalence classes are shrunk and possibly deleted (lines 15 to 21). Each equivalence class is shrunk, and each shrinking operation takes $O(1)$ time, hence the complexity for shrinking is $O(|\mathcal{E}Q_k|)$. Some of the shrinking operations might require a deletion, which can be performed in $O(1)$ time as the equivalence classes, and the associated synCCs can be looked up and deleted in constant time using a hash map. Hence, the total complexity for shrinking and deleting is $O(|\mathcal{E}Q_k|)$.

S_k are created in lines 22 to 30. At worst, the maximum number of creations is $|S_{k+1}|$ when all existing synCCs in S_k are deleted and all synCCs in S_{k+1} are new. To maintain the data structure for these synCCs, two new equivalence classes will be created eq01 and eq10, which has complexity of $O(1)$. However, they would require finding the connected components, which would have the worst case total complexity of $O(|E_C^{k+1}|) (|s_1| + |s_2| + \dots + |s_n| = O(|E_C^{k+1}|)$, where $s_1, s_2, \dots, s_n \in S_{k+1}$). The complexity for creation is then $O(|E_C^{k+1}|)$.

Finally, some of the synCCs are merged (line 35). The maximum number of merges possible is equal to the number of equivalence classes $|\mathcal{E}Q_k|$. Each merge, using a union-set-join data structure, requires $O(1)$ time. Hence the complexity of merging the equivalence classes and the associated synCCs is $O(|\mathcal{E}Q_k|)$.

Therefore, the total time complexity to update S_k to S_{k+1} is $O(|E_C^k|) + O(|V_{S_k Avg}| \cdot |S_k|) + O(|\mathcal{E}Q_k|) + O(|E_C^{k+1}|)$. Each changed edge can be associated with one distinct changed waveform, hence at most, there is one equivalence classes associated with each changed edge, or $|\mathcal{E}Q_k| < |E_C^{k+1}|$. Therefore we can simplify the total complexity to $O(|E_C^k|) + O(|V_{S_k Avg}| \cdot |S_k|) + O(|E_C^{k+1}|)$ \square

This shows the incremental maintenance of a set of synCCs, $S_k \rightarrow S_{k+1}$, is only linear in the number of synCCs, the number of changing edges and the average size of a synCC. When the changes in the graph are localised, there are generally fewer synCCs and equivalence classes to update, hence the running time is less. In contrast, when the changes to the graph are more randomly distributed, there are more synCCs and equivalence classes to update, hence the running time is larger. But for all datasets in our evaluation, the number of synCCs and new changing edges are still several orders of magnitude smaller than the total number of existing changing edges, hence even for the distributed change case, ciForager is still faster than cSTAG. We show in Section 7 and 8 that the time to maintain the set of synCCs and calculate the distances between them is much less than the time to compute the pairwise distances in cSTAG and regHunter.

6. USING GRAPH VORONOI DIAGRAMS TO COMPUTE AND MAINTAIN THE SHORTEST PATH DISTANCES BETWEEN SYNCCS

In this section, we describe how a Voronoi partition of a dynamic graph can be used to efficiently compute and maintain the shortest path distances between a dynamic set of synCCs (synchronised connected components).

Voronoi diagrams [de Berg et al. 2008] are an important concept in computational geometry. Given a set of Voronoi points and the space the points are embedded in, the space is partitioned into cells. A cell envelopes each Voronoi point, and other points in the space belongs to the cell whose Voronoi point it is closest to. In ciForager, we extend the Voronoi diagram to dynamic graphs. In the literature [Erwig 2000], graph Voronoi are used to partition the graph into cells to quickly detect what are the closest vertices to a set of pre-selected vertices. Algorithms [Erwig 2000] have been designed to up-

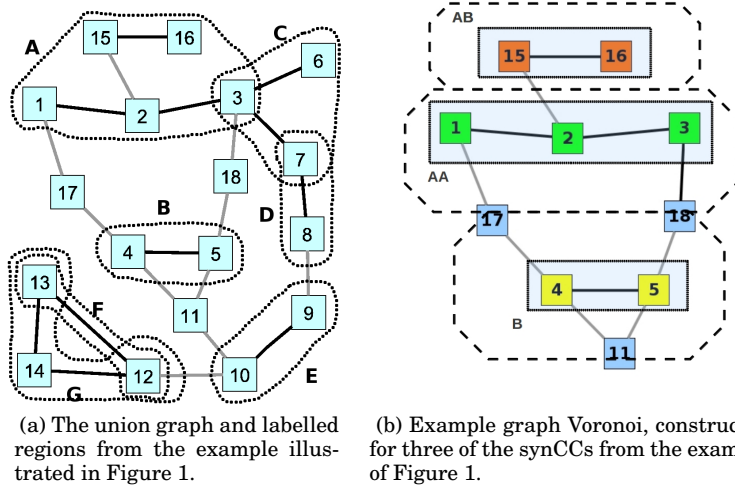


Fig. 4: An example graph Voronoi, constructed from part of the graph in Figure 1 (we have re-included Figure 1f for easier reference). Each synCC is highlighted with a solid rectangle and labelled according the regions to which their corresponding synCCs belong. Region A of Figure 1 consists of two synCCs, so vertex labelled “Aa” corresponds to $\{e_{1,2}, e_{2,3}\}$ and “Ab” corresponds to $\{e_{15,16}\}$. The cell of each synCC is drawn with a dotted line. Note that the cell boundaries can occur on an edge or a vertex.

date the cells as the graph changes. However, graph Voronois have not been used to determine which existing shortest paths need to be recomputed due to changes in the underlying graph. Nor has previous work used synCCs (synchronised connected components) as the set of Voronoi points, or analysed how changes to the graph Voronoi can be solely considered in terms of synCCs changes. This type of consideration simplifies the maintenance of the graph Voronoi and the other structures, as all changes in ciForager can be considered as synCC changes.

In ciForager, the Voronoi points are the synchronised connected components, and the cells are the vertices and edges that are closest to the associated synchronised connected component. As an example of a graph Voronoi and how it is extended in ciForager, consider an example graph Voronoi (Figure 4b), constructed for a part of the graph from Figure 1. We only illustrate part of the graph to avoid cluttering the example. Each synCC is highlighted by a solid rectangle, and the corresponding cells are the dotted octagons around the synCCs. Where the octagons meet are the cell boundaries. As Figure 4b shows, cells are constructed around each synCC, and the boundaries of the cells are equidistant to their respective synCCs.

Using this graph Voronoi structure, we can reduce the complexity of computing shortest path distances on a smaller meta-graph. Furthermore changes to the underlying graph can be isolated to a subset of cells, and only those synchronised connected components with affected cells need their shortest path distances updated. This also helps reduce the number of shortest path recomputations.

The meta-graph represents the shortest path distances between neighbouring synchronised connected components. synCCs are neighbouring if their cells are neigh-

hours of each other. A vertex in the constructed meta-graph represents an synCC and an edge between two vertices indicates the corresponding synCCs are neighbours. The weight of the edges is the shortest path distance between neighbouring synCCs, and the weight of the vertices is the average shortest path distance within the associated synCC. We shall shortly explain why vertex weights are necessary. The shortest path distance between a pair of synCCs is then the shortest path distance between the corresponding vertices in the meta-graph. As an example of a meta graph, consider Figure 5b. It shows the meta graph constructed from the example of Figure 1.

In the rest of this section, we first define some further notation needed for graph Voronoi, then describe how to calculate the spatial distances using the Voronoi and its meta-graph. After that, we show that only localised parts of the graph and the associated graph Voronoi need to be updated when changes occur to the underlying graph. Finally, we present the time complexity to update the graph Voronoi.

First, we introduce the graph Voronoi related notation.

Definition 6.1. The shortest path distance between a vertex v_a and a synCC s_i is defined as $d(v_a, s_i) = \min_{v_i \in s_i} d(v_a, v_i)$, where $d(v_a, v_i)$ is the shortest path distance between the vertices v_a and v_i . Similarly, the shortest path distance between an edge $e_{a,b}$ (v_a, v_b are the incident vertices) and a synCC s_i is defined as $\min_{v_i \in s_i} (d(v_a, v_i), d(v_b, v_i))$.

Definition 6.2. Let $cell_i^k$ denote the cell of synCC s_i for window W_k . A cell consists of vertices and edges that have s_i as their closest synCC. It is defined as $cell_i^k = \{v_a | d(v_a, s_i) \leq d(v_a, s_j), \forall j \neq i\} \cup \{e_{a,b} | d(e_{a,b}, s_i) \leq d(e_{a,b}, s_j), \forall j \neq i\}$. Where it is unambiguous to do so, the k superscript will be dropped.

For example, $e_{4,17}$ and $e_{5,18}$ belong to the cell of region B in Figure 4a (see Figure 4b).

Definition 6.3. The set of neighbours of an synCC s_j is the set of synCCs that have their cells adjacent to $cell_j$. The set of neighbours of s_j is denoted by $N(s_j)$.

Definition 6.4. The boundary list between two neighbouring synCCs s_i and s_j is the set of vertices and edges that lie on the boundary of $cell_i$ and $cell_j$ for window W_k . The boundary list of s_i and s_j is denoted by $B^k(i, j)$. More formally, $B^k(i, j) = \{v_a | d(v_a, s_i) = d(v_a, s_j), v_a \in s_i, s_j\} \cup \{e_{a,b} | d(e_{a,b}, s_i) = d(e_{a,b}, s_j), v_a \in s_i, v_b \in s_j\}$

Definition 6.5. Given the union graph over window W_k and a set of corresponding synCCs S_k , a graph Voronoi $G_v^k(V_v^k, E_v^k)$ divides the union graph into a set of cells satisfying Definition 6.2, where for each synCC s_i , there exists an associated $cell_i^k$, and the boundaries between neighbouring cells satisfy Definition 6.4.

With these definitions, we now explain how the shortest path distances are computed using the graph Voronoi diagram.

6.1. Computing the Shortest Path Distances using the Graph Voronoi Diagram

We define the shortest path distance between a pair of synCCs s_i and s_j as the minimum shortest path distance between a vertex of s_i and a vertex of s_j . Formally, it is defined as $\min_{v_a \in s_i, v_b \in s_j} d(v_a, v_b)$. The graph Voronoi diagram allows us to compute the shortest path distance between neighbouring synCCs quickly. In addition, the Voronoi meta-graph built from the graph Voronoi diagram enables us to quickly compute an approximate shortest path distance between non-neighbouring synCCs. In this subsection, we first describe how we can efficiently compute the shortest path distances of neighbouring synCCs, then explain how these distances can be used to compute the distances for non-neighbouring synCCs.

The shortest path distance between neighbouring synCCs s_i and s_j can be calculated from the boundary lists of the synCCs. As the vertices and edges on the lists are on

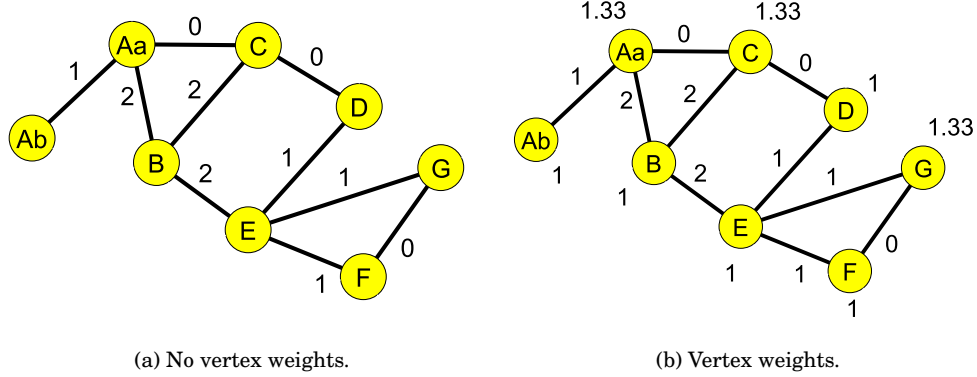


Fig. 5: The graph Voronoi meta-graph constructed for the example of Figure 1. The vertices are the synCCs, or the Voronoi points. The vertices are labelled according to the regions to which their corresponding synCCs belong. Region A of Figure 1 consists of two synCCs, so vertex labelled “Aa” corresponds to $\{e_{1,2}, e_{2,3}\}$ and “Ab” corresponds to $\{e_{15,16}\}$.

the boundary, they are equidistant from the associated synCCs. Therefore the shortest path distance is simply double of the distance from the boundary to one of the synCCs sharing that boundary. Formally, let the shortest path distance between synCCs s_i and s_j (of window W_k) be denoted by $SPD^k(i, j)$. Then the shortest path distance between neighbouring synCCs s_i and s_j can be computed as:

$$SPD^k(i, j) = \min_{v_a \in B^k(i,j), e_b \in B^k(i,j)} (2d(v_a, s_i), 2d(e_b, s_i) + 1) \quad (1)$$

Note that we can equally define $SPD^k(i, j)$ in terms of the distance between the s_j and the boundary vertices and edges, as $d(v_a, s_i) == d(v_a, s_j)$.

The distances between neighbouring synCCs are the edge weights in the meta-graph. Using these distances, the shortest path distance between non-neighbouring synCCs is computed as the shortest path distance on the meta-graph. The shortest path distances of the meta-graph consists of the sum of the edge and vertex weights. Before we explain the reasoning of the vertex weights, we first formally define the notation for the Voronoi meta-graph.

Definition 6.6. Let $G^*(V^*, E^*)$ denote the Voronoi meta-graph of graph $G(V, E)$. Let $v_i^* \in V^*$ be associated with synCC s_i and $w(v_i)$ and $w(e_{i,j})$ denote the weights of v_i and $e_{i,j}$ respectively.

A path and shortest path distance between two vertices in this meta-graph are defined as follows:

Definition 6.7. Let $P^*(i, j) = \langle v_i^*, v_l^*, \dots, v_p^*, v_j^* \rangle$ denote a path between v_i^* and v_j^* . Then the path distance $d(P^*(i, j)) = w(e_{i,l}) + w(v_l^*) + \dots + w(v_p^*) + w(e_{p,j})$.

Now we can define the shortest path distance between two synCCs as:

Definition 6.8. The shortest path distance between synCCs s_i and s_j is

$$SPD^k(i, j) = \min_{P^*(i,j)} d(P^*(i, j))$$

The vertex weights of the meta-graph are calculated as the average shortest path distance among the vertices of each synCC and represent the average cost to transit through a synCC. The vertex weights are included in the shortest path distance calculations to prevent two non-neighbouring synCCs being connected via a chain of adjacent synCCs, resulting in an incorrect total distance of 0. For an example, Figure 5a shows the meta-graph with no vertex weights, built from the example of Figure 1 (see Figure 4a). Consider the shortest path distance between vertices Aa and E, which is 1 when no vertex weights are considered. Now consider Figure 5b, which is the meta-graph with vertex weights. The shortest path distance between Aa and E is 3.33, which is much closer to the true shortest path distance of 3. Hence, using the vertex weights help to ensure the meta-graph shortest path distance is close to the true distance.

6.2. Constructing the Graph Voronoi Diagram

In order to construct the graph Voronoi, we grow the cells from each synCC simultaneously. Each cell expands at the same rate, and keep growing until either it hits the edge of the graph or the boundary(ies) of other cells. When all cells stop growing, the graph would be partitioned into a number of cells. To implement this, we run Dijkstra's algorithm from multiple roots. The roots are the synCCs, which are considered as an indivisible entity in the Voronoi diagram. We can use the standard implementation of Dijkstra's algorithm using a priority heap, as the heap used ensures the vertices are expanded in the correct order. Algorithm 3 shows the pseudo code for this process.

Computing this Voronoi structure is no worse than visiting every vertex and edge in the whole graph, which is the lower bound of any shortest path distance algorithm. The dominant running time cost is the cost of running Dijkstra's algorithm. This has a complexity of $O(|E| + |V|\log|V|)$ if implemented with Fibonacci heaps [Erwig 2000].

6.3. Incrementally Maintaining the Graph Voronoi Diagram and Shortest Path Distances

In this section we describe our approach to maintaining and updating the graph Voronoi and the shortest path distances between synCCs when changes occur. First, we introduce a theorem that states that changes to the underlying graph can solely be considered in terms of changes in the set of synCCs. The consequence of this theorem is that we can analyse how the graph Voronoi and associated shortest path distances change and how they are affected solely in terms of synCC changes. This simplifies the analysis. After introducing the theorem, we introduce a series of lemmas that show what part of the graph Voronoi and shortest path distances change, and present how we update the affected parts and distances.

Definition 6.9. Given a synCC s_i , the no-change operation, no-change: $\mathcal{S}_k \rightarrow \mathcal{S}_{k+1}$, maintains the edge membership of s_i .

THEOREM 6.10. *Changes to the underlying dynamic graph can be considered solely as a sequence of merge, split, create, delete and no-change operations to the set of synCCs.*

PROOF. Parts of the underlying graph that are non-changing in the last window but experience change in the latest snapshots can be considered as part of new synCCs (*create*, see Appendix D, Lemma D.1). Parts of the graph that no longer change over the current window can be considered as deletions from the existing synCCs (*delete*, see Appendix E, Lemma E.1). All other changes must involving existing synCCs, as the set of synCCs encapsulates all existing change behaviour in the dynamic graph. Therefore, all other changes either involving merging, splitting or not changing existing synCCs. Changes that just extend an existing synCC with the same value does not change the edge membership of that synCC, hence can be consider as a *no-change* operation. The

ALGORITHM 3: Construction of the graph Voronoi diagram.

Input: S_k - The set of synCCs for window W_k .
 $G_{union,k}$ - The union graph computed over the snapshots in W_k .
Output: P_k - Parent list for all vertices in $G_{union,k}$.
 D_k - Vector of distance of each vertex to its closest synCC.
 B_k - Boundary list between all pairs of synCCs for window W_k .
 N_k - synCC Neighbourhood lists for each synCC for window W_k .

$B_k = \{\}; N_k = \{\}; P_k = \{\};$
 $D[v] = \infty, \text{marked}[v] = \text{false}, \forall v \in V_{union,k}.$
construct empty heap H ;
// Initialise and mark vertices in the synCCs **for each** $s_j \in S_k$ **do**
 for each $(x, y) \in s_j$ **do**
 $D[x] = D[y] = 0; P[x] = P[y] = \{j\};$
 $\text{marked}[x] = \text{marked}[y] = \text{true};$
 $H.\text{insert}(x, 0); H.\text{insert}(y, 0);$
 end
end
// Start Dijkstra's algorithm.
while H is not empty **do**
 $v = \text{deleteMin}(H); \text{marked}[v] = \text{true};$
 // Find all adjacent vertices of v .
 $A_v = \text{adjacent}(v)$, where $v_a \in A_v$, if $e_{v,v_a} \in E_{union,k} \wedge \text{marked}[v_a]$;
 for each $v_a \in A_v$ **do**
 $\text{newD} = d[v] + w(v, v_a);$
 if $d[v_a] = \infty$ **then**
 $d[v_a] = \text{newD}; p[v_a] \leftarrow p(v);$
 $H.\text{insert}(v_a, \text{newD});$
 else
 // Vertex boundary case.
 if $\text{newD} == d[v_a]$ **then**
 add $p[v]$ to $p[v_a]$;
 add v_a to $B_k(p(v), p(v_a))$;
 add $p[v_a]$ to $N_k[p(v)]$; add $p(v)$ to $N_k[p(v_a)]$;
 end
 // Edge boundary case.
 else if $\text{newD} == d[v_a] + 1$ **then**
 add $e(v, v_a)$ to $B_k(p(v), p(v_a))$;
 add $p[v_a]$ to $N_k[p(v)]$; add $p(v)$ to $N_k[p(v_a)]$;
 end
 else if $\text{newD} < d[v_a]$ **then**
 $d[v_a] = \text{newD}; p[v_a] = \{p(v)\};$
 $H.\text{decrease}(v_a, d[v_a] - \text{newD});$
 end
 end
end
end

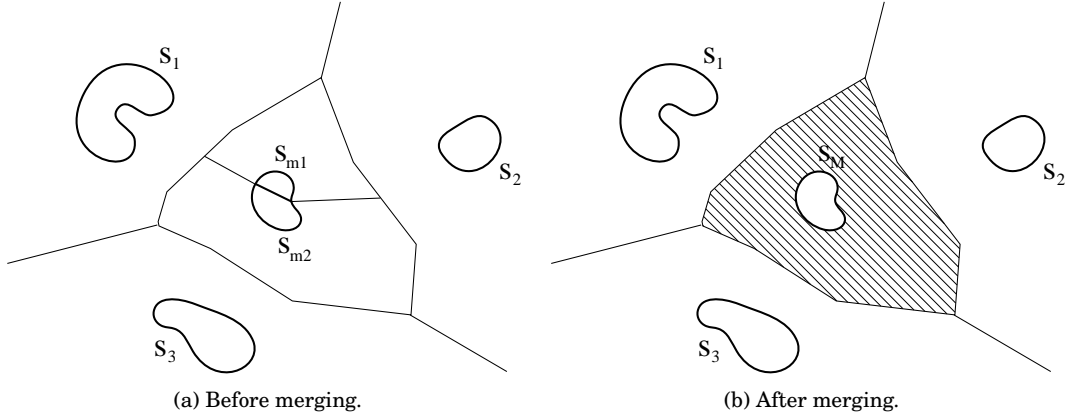


Fig. 6: An example from a portion of a graph Voronoi diagram, showing a subset of the embedded synCCs and the cell boundaries before and after synCCs s_{m1} and s_{m2} are merged. The solid black lines represent the cell boundaries, and the shaded, diagonally striped area represent the affected area.

other types of change either cause synCCs to *merge* (see Lemma 6.11), or a synCC to *split* (see Appendix C, Lemma C.1). \square

Theorem 5.10 indicates that the changes to the graph, graph Voronoi and the associated shortest path distances can be considered as changes to synCCs: two existing synCCs merging, an existing synCC being split into two synCCs, creation of a new synCC, or deletion of an existing synCC (we do not need to do anything for the no-change operation). Therefore, we only need to consider how to handle these four synCC changes when updating the graph Voronoi and the associated shortest path distances.

In the following, we introduce a lemma that shows which cells of the graph Voronoi are affected and which shortest path distances change when two synCCs merge. The lemmas for splitting, creating and deleting synCCs are available in Appendices C–E. These lemmas are used to design procedures to only update the necessary parts of the graph Voronoi, and the changed shortest path distances. In the four lemmas, we make the assumption that only one change is incorporated at a time. Therefore, it is assumed that synCCs and their respective cells that are not part of the operation or explicitly mentioned in the lemmas do not experience change during the update operation.

To illustrate the logic of the lemmas, we constructed examples of the graph Voronoi before and after the change in the synCC. These examples are illustrated in Figures 6 and 14–16. We will describe them in more detail in the appropriate subsection. The figures show the graph Voronoi, the synCCs, and the cell partition before and after the synCC change. Dotted lines are used to represent a pending synCC split or the location of previous cell boundaries.

6.3.1. Merging:

LEMMA 6.11. *Let s_{m1} and s_{m2} be a pair of synCCs to be merged into s_m ; i.e., $s_m = s_{m1} \cup s_{m2}$. Then:*

- (1) *The resulting cell of s_m is the union of the cells of s_{m1} and s_{m2} ; i.e., $cell_m = cell_{m1} \cup cell_{m2}$.*

(2) *Only the shortest paths between synCCs that were neighbours to both s_{m1} and s_{m2} change. More formally, $\forall s_p \in S_{k+1}$,*

$$SPD^{k+1}(m, p) = \begin{cases} SPD^k(m1, p) & s_p \in N^k(m1), s_p \notin N^k(m2) \\ SPD^k(m2, p) & s_p \notin N^k(m1), s_p \in N^k(m2) \\ \min(SPD^k(m1, p), SPD^k(m2, p)) & s_p \in N^k(m1), s_p \in N^k(m2) \end{cases} \quad (2)$$

PROOF. See Appendix B. \square

Consider Figure 6, which shows a portion of an example graph Voronoi before and after two synCCs merge. The diagonal, striped area represent the area of the Voronoi affected by the merge. Lemma 6.11 and Figure 6 show that changes to membership of the cells are restricted to vertices and edges in s_m . In addition, the lemma shows that the only distances that need to be updated are the ones where s_{m1} and s_{m2} share a neighbour.

Therefore the update can be performed by first relabelling the parent of all the vertices and edges in $cell_{m1}$ and $cell_{m2}$ to s_m (i.e., they become part of $cell_m$). Then, the shortest path distances are updated using Equation 2.

6.3.2. Algorithm. The algorithm to update an existing graph Voronoi takes a sequence of changes to the synCCs, and applies the updating techniques outlined previously, one by one. At the end of the process, the graph Voronoi is updated.

6.3.3. Complexity. In this subsection, we derive the time complexity to perform any of the four graph Voronoi updates. This result will be used for constructing the overall complexity of ciForager.

LEMMA 6.12. *The worst case complexity of performing any synCC update operation is $O(|E_{cellAvg}| + |V_{cellAvg}| \log(|V_{cellAvg}|))$, where $|V_{cellAvg}|$ and $|E_{cellAvg}|$ are the average number of vertices and edges of a voronoi cell respectively.*

PROOF. We analyse each of the update operations and show that they have complexity equal to or less than $O(|E_{cellAvg}| + |V_{cellAvg}| \log(|V_{cellAvg}|))$.

The time complexity to perform merging is dominated by the time to update the membership/parent and distance information of the merging cells. As we need to update the membership of one of the merging cells to the id of the other, this has linear complexity with the number of vertices in a cell. As the number of vertices (and edges) in a cell can vary [Honiden et al. 2009], we use the average number of vertices and edges of a voronoi cell. Hence the (average) complexity is $O(|V_{cellAvg}|)$.

After splitting a synCC into two, the vertices and edges in the cell of the original synCC need to be allocated between the two new synCCs. Hence Dijkstra's algorithm needs to be run over the cell of the split synCC. Recall that Dijkstra's algorithm has a complexity of $O(|E| + |V| \log|V|)$ for a graph with vertex set V and edge set E . Using the average vertex and edge sizes, the complexity to split a synCC has complexity $O(|E_{cellAvg}| + |V_{cellAvg}| \log(|V_{cellAvg}|))$.

The time complexity to create a new synCC is again dominated by the time to reassign and rerun Dijkstra's algorithm over the cell of the new synCC. This is over S_{k+1} rather than S_k but assuming the average size of a cell is calculated over a long time, we can approximate the size of the cells by $|V_{cellAvg}|$ and $|E_{cellAvg}|$. Therefore the complexity of creating a synCC is $O(|E_{cellAvg}| + |V_{cellAvg}| \log(|V_{cellAvg}|))$.

The time complexity to delete a synCC is again dominated by the time to reassign and rerun Dijkstra's algorithm over the deleted cell. Therefore, the complexity is $O(|E_{cellAvg}| + |V_{cellAvg}| \log(|V_{cellAvg}|))$.

Therefore, the worst case complexity of performing any synCC update operation is $O(|E_{cellAvg}|) + |V_{cellAvg}| \log(|V_{cellAvg}|)$. \square

7. OVERALL TIME COMPLEXITY OF CIFORAGER

In this section, we analyse the time complexity to compute the regions for W_{k+1} , given the data structures of W_k . Then we analyse the time complexity of the previous method cSTAG [Chan et al. 2008] and compare the complexities of ciForager and cSTAG.

THEOREM 7.1. *The overall complexity of ciForager to update the regions of correlated change from window k to window $k + 1$ is:*

$$\begin{aligned} & O(|E_C^k|) + O(|V_{S_kAvg}| \cdot |S_k|) + O(|E_C^{k+1}|) + O(|E_{cellAvg}| \\ & + |V_{cellAvg}| \log(|V_{cellAvg}|)) * U_k + O(|\mathcal{E}Q_k|^2) + O(|S_k|^2 + |S_k| \log(|S_k|)) + ciFor_clus_cost. \end{aligned} \quad (3)$$

ciFor_clus_cost is the clustering cost, while U_k is the number of synCC update operations between windows k and $k + 1$.

PROOF. To determine the time complexity of ciForager, we analyse the sub-steps required to perform the update (see Figure 2b). As a reminder these sub-steps are: extracting the changed edges, updating the waveforms and synCCs, computing the temporal distances between the equivalence classes, updating the graph Voronoi, computing the shortest path distances between the synCCs, clustering the synCCs, and extracting the edges and performing region association.

Extracting the changed edges involves comparing the two edge sets, which can be done in $O(\max(|E^k|, |E^{k+1}|))$ if performed as a separate operation. However, this step can be performed when the graph snapshots are read in. A hash table of edges is maintained for the previous graph snapshot and the comparison is performed when reading in the next graph snapshot from a file. Hence, we do not need to directly include this cost into the analysis. Note that this step is also shared with cSTAG.

The time to update the set of synCCs from S_k to S_{k+1} is $O(|E_C^k|) + O(|V_{S_kAvg}| \cdot |S_k|) + O(|E_C^{k+1}|)$ (see Section 5.3.1).

The time to compute the pairwise temporal distances among the equivalence classes $\mathcal{E}Q_k$ is $O(|\mathcal{E}Q_k|^2)$, as ciForager incrementally computes the waveform distances, which can be performed in $O(1)$ for each comparison [Chan et al. 2009].

The time to update the graph Voronoi is equal to the time to compute an operation (see Section 6.3.3) multiplied by the number of synCC updates. Recall that the number of synCC updates between windows k and $k + 1$ is denoted by U_k . Then the time to update the graph Voronoi is $O(|E_{cellAvg}| + |V_{cellAvg}| \log(|V_{cellAvg}|)) * U_k$.

Dijkstra's algorithm is used to compute the shortest path distances on the meta-graph. The time to compute the shortest path distances for a graph $G(V, E)$ is $O(|E| + |V| \log(|V|)) = O(|V|^2 + |V| \log(|V|))$. As $|V| = |S_k|$ for the meta-graph, then the time to compute the pairwise spatial distances among the synCCs of S_k is $O(|S_k|^2 + |S_k| \log(|S_k|))$.

The clustering cost is dependent on the clustering method employed, but is dependent on $|S_{k+1}|$. We denote it by *ciFor_clus_cost*.

The time to extract the changed edges from the clusters of synCCs is $O(|E_C^{k+1}|)$.

Finally, the time to compute the region association is dominated by the spatial inter-region time, which is the time to compute the set intersection over the two sets of regions, which have total sizes of $|E_C^k|$ and $|E_C^{k+1}|$. Keeping an inter-region confusion matrix, the algorithm makes a linear scan through the smaller of the two changed edge sets, and updates the intersection count of the confusion matrix. Again, this is

actually is part of the underlying algorithm (cSTAG) and not a part of the algorithm we are focusing on in this paper, but we include it for completeness.

Therefore, the total complexity for ciForager is

$$O(|E_C^k|) + O(|V_{S_k Avg}| \cdot |S_k|) + O(|E_C^{k+1}|) + O(|E_{cellAvg}| + |V_{cellAvg}| \log(|V_{cellAvg}|)) * U_k \\ + O(|\mathcal{E}Q_k|^2) + O(|S_k|^2 + |S_k| \log(|S_k|)) + ciFor_clus_cost + O(\min(|E_C^k|, |E_C^{k+1}|)) \quad (4)$$

which can be simplified to

$$O(|E_C^k|) + O(|V_{S_k Avg}| \cdot |S_k|) + O(|E_C^{k+1}|) + O(|E_{cellAvg}| + |V_{cellAvg}| \log(|V_{cellAvg}|)) * U_k \\ + O(|\mathcal{E}Q_k|^2) + O(|S_k|^2 + |S_k| \log(|S_k|)) + ciFor_clus_cost \quad (5)$$

□

Equation 5 indicates the running time of ciForager depends mainly on the number of changed edges ($|E_C^k|$ and $|E_C^{k+1}|$) and the number of synCCs ($|S_k|$). The term, $O(|E_{cellAvg}| + |V_{cellAvg}| \log(|V_{cellAvg}|)) * U_k$, has terms $E_{cellAvg}$ and $V_{cellAvg}$, whose size is generally inversely proportional to the number of synCCs (the more synCCs, the smaller their sizes on average). While the second term, $O(|S_k|^2 + |S_k| \log(|S_k|))$ is quadratic with the number of synCCs. Hence there is a trade-off between more synCCs against the average size of cells. We explore this trade-off in Section 8.

Using a similar analysis, we can show that the complexity of cSTAG is dominated by the shortest path comparisons between all pairs of changed edges ($O(|E_C^k|^2 \cdot V)$), which is generally much larger than the number of synCCs. From this complexity analysis, it can be seen ciForager is much faster than cSTAG. In the evaluation section that follows, we show that ciForager is up to 10^6 times faster than cSTAG

8. EVALUATION

In this section we evaluate the running time and accuracy of ciForager. We show that ciForager is up to 10^6 times faster than cSTAG, with the speed advantage growing a) as the changes become more localised and b) as the size of the graphs analysed increases. In addition, we show that ciForager has equal accuracy to cSTAG, and has linear complexity with respect to the size of the analysed graphs. Because of the faster running speed and more efficient memory usage, ciForager can analyse very large graphs like the global BGP connectivity graph that cSTAG cannot.

In the rest of this section, we first describe the datasets used in the evaluation. Then we describe the accuracy measures used. Finally, we evaluate the running times and accuracy of ciForager and cSTAG for both synthetic and real datasets, and present two new high quality regions discovered by ciForager when analysing the global BGP connectivity graph.

8.1. Datasets

In this section, we describe two sets of datasets used to evaluate ciForager and cSTAG, one synthetic and one real-life. The synthetic datasets have more localised changes. In contrast, the real-life datasets have more bursty, distributed changes. We introduce these two types of datasets to evaluate the efficiency of ciForager for different types of changes.

8.1.1. Synthetic Graphs (Localised Changes). The aim of the synthetic graphs is to generate a number of pseudo random graphs that test the efficiency and accuracy of ciForager. On one hand, we wish to vary the number of synCCs and other parameters that affect the efficiency of ciForager. On the other hand, we also wish to evaluate and

compare the accuracy of ciForager against cSTAG. Hence we use two synthetic dataset generation methods, first used in [Chan et al. 2009], to evaluate cSTAG and regHunter.

The first method generates regions of change, then generates paths between them to form the dynamic graph. The second method generates the graph, then randomly select subgraphs to become regions. The first method allows more control over the separation criteria between the regions, but it is difficult to control the number of synCCs and the percentage of the graph that experiences change. Hence, we introduce the second method, which allows more control over these two factors.

The graphs generated tend to have changes that are more localised. Localised changes are realistic for datasets like the online strategy game Travian, where players tend to ally and attack within a local neighbourhood.

Region-then-link Method. The difficulty of finding high quality regions in a dataset depends on the separability of the regions. Separability is measured by three factors: **minSpaSep**, the minimum spatial separation between any pair of edges that are in different regions; ii) **minTemSep**, the minimum temporal separation between the change waveforms of any pair of edges in different regions; and iii) **minEvtSep**, the minimum temporal separation between consecutive, but independent and separate windows of changes affecting the same set of edges. By varying these factors, we can generate a variety of synthetic dynamic graphs to test the accuracy and efficiency of ciForager and cSTAG.

To generate the datasets using this method a number of random subgraphs are generated. A number of random paths, at least minSpatSep long, are generated between each pair of subgraphs. This ensures that each subgraph is at least minSpatSep from all the other extracted subgraphs. Next, a random sequence of changes are generated for each subgraph, such that the distance between any pair of sequences is at least minTemSep. In addition, each sequence consists of a random number of subsequences of high change, separated by at least minEvtSep of periods of no change. Each subgraph, along with a subsequence of change, constitutes a region. A simulation is used to generate the snapshots and regions by applying the sequences of changes to the original random graph. The names of datasets generated by this method have the prefix *synGen*.

Graph-then-region Method. To generate datasets using this method, a graph is first generated. Any graph model generator can be used to generate this graph. Next, non-overlapping subgraphs are selected for each region. Finally, random change sequences are assigned to each region, and additional graph snapshots are generated based on the original graph, the set of regions, and their associated change sequences. The names of the datasets generated by this method will have the prefix *introGen*.

8.1.2. Border Gateway Protocol (BGP) Connectivity Graphs (Widely Distributed Changes). BGP is a routing protocol used to establish the forwarding tables between the routers of organisations, known as Autonomous Systems (ASs), on the Internet. The vertices in the BGP connectivity graph represent the ASs, and the edges represent the existence of a routing path between the ASs. The BGP connectivity graph represents the top-level routing topology of the Internet.

The RouteViews project³ at the University of Oregon collects BGP routing information by passively peering with a number of distributed ASs. From each table obtained from RouteViews, we built a snapshot of the BGP connectivity graph using the AS PATH path entries.

³<http://www.routeviews.org>

In [Chan et al. 2008], the Katrina event was analysed because it has been reported that its effect on the Internet was mostly localised around Louisiana and several other southern states. In this paper, we analyse the US portion of the BGP graph. In addition, we also analyse the whole (worldwide) BGP graph over the same period, which was not performed in previous work [Chan et al. 2008][Chan et al. 2009].

In August 2005, the US BGP graph consisted of around 9,000-10,000 vertices and 36,000 edges, and the whole BGP graph consisted of around 20,000 vertices and 42,000 edges. We analysed three and a half days of snapshots, from 28 August, 13:19 to 31 August, 22:32. This period included the landfall of Hurricane Katrina (around 29 August, 10:00).

There are two types of changes occurring within the BGP connectivity graph. The changes associated with the failure events tend to be of a more localised nature. In addition, there is a steady amount of distributed, background changes of a more random nature. Overall, there are many distributed areas of background changes, but strong areas of localisation are present when failure events like the arrival of Hurricane Katrina occur.

8.1.3. 1998 World Cup Website Access. In 1998, the 16th FIFA World Cup was held in France. To study the workload characteristics of the official web site, *www.france98.com*⁴, access logs⁵ of the web site were analysed by Arlitt and Jin [Arlitt and Jin 1999]. It was reported by Arlitt and Jin that the website experienced flash crowds - sudden, large increases in the number of unique, legitimate clients accessing the website. This coincided with the time of weekday matches. The 1998 World Cup was the first world cup where live scores were available online. Therefore, a significant number of fans, who cannot watch the football matches on television, monitored the live scores via the website during the matches, producing the flash crowds.

We construct a dynamic graph of the website accesses and find regions of correlated change that correspond to different types of access, e.g., a group of accesses relating to online viewing of a particular match. In previous work [Chan et al. 2008][Chan et al. 2009], we had manually matched discovered regions with the matches, based on the time the regions were defined and the websites contained in them. Like the BGP dataset, this dynamic web access graph has localised and distributed background changes. For comparison, we used the same snapshots as [Chan et al. 2008], which varied from 3417 to 5891 vertices and 110 to 849 edges. Please refer to [Chan et al. 2008] for details on the snapshot extraction process.

8.2. Accuracy Validation Methods

Accuracy was evaluated using external and internal cluster validation methods. In the external methods, we compare the obtained set of regions with a known set of true regions, while in the internal methods, we compare against a set of measures that are based on other partitioning objectives. We will describe the external and internal validation measures next.

8.2.1. External Accuracy Validation Measure. In external accuracy validation, we have a set of true (generated) regions, R^{tru} , and a set of detected regions R^{det} . This problem has been partially studied in the field of external cluster validation [Halkidi et al. 2001]. Existing validation methods can be divided into three groups: a) those that count the number of pairs [Halkidi et al. 2001]; b) those that compare set membership [Meila 2003]; and c) those that compare membership distribution [Bae et al. 2010][Zhou et al. 2005]. However, none of them consider both the temporal behaviour

⁴As of August 2007, the address is still valid, but links to a general soccer promotion website.

⁵Available at [wc98trace].

Name	Attribute varied	Attribute set
<i>synGen003</i>	–	Same as <i>synGen003</i> of [Chan et al. 2009]
<i>synGenSeq</i>	Total sequence length (T)	30, 70, 110, 150
<i>introGenScc</i>	# of regions	6, 17, 34, 46, 77, 147
<i>introGenPerc</i>	% of edges changing	5%, 10%, 20%, 30%, 50% 70%
<i>introGenSize</i>	Number of edges of graphs	2000, 4000, 16000, 32000, 65000

Table III: Summary of the synthetic datasets used in the evaluation.

and membership of regions when comparing sets of regions. Hence, in prior work [Chan et al. 2009], we introduced a matching-based method that incorporates both temporal and membership considerations in [Chan et al. 2009]. The method is called *extRegCompare*.

Each region in \mathbf{R}^{tru} is matched with one or more regions in \mathbf{R}^{det} . *extRegCompare* finds the matching that minimises the total distance between the matched regions, by solving it as a mass transportation linear programming problem [Luenberger 2003]. A distance of 0 means the two sets of regions are the same, and a distance of 1 means the two sets are completely different.

8.2.2. Internal Accuracy Validation Measure. Similar to [Chan et al. 2009], we use the average intra-region distance and the average inter-region distance as the internal validation measures. Average intra-region distance measures how compact the regions are. Average inter-region distance measures how well separated the regions are.

Both of these measures depend on an intra or inter-region distance measure. We use separate spatial and temporal distances, so effectively, there are four different measures (two for temporal, two for spatial). More formally, if R_y and R_z are two of the discovered regions, then

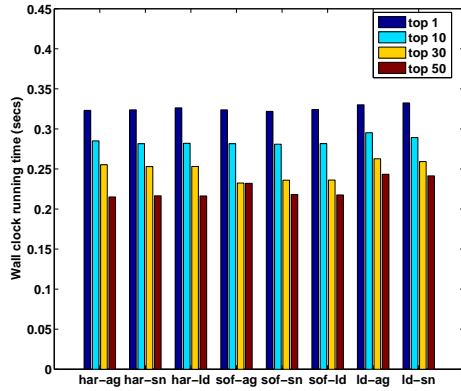
$$d_{\text{spat}}(R_y, R_z) = \frac{1}{|R_y||R_z|} \sum_{e_i \in R_y} \sum_{e_j \in R_z} d_{\text{spd}}(e_i, e_j, W^{1,T})$$

where T is the number of graph snapshots in the sequence and d_{spd} is the shortest path distance measure. The average *intra* and *inter-region spatial* distances are defined as $d_{\text{spat}}(R_y, R_y)$ and $\frac{1}{|\mathbf{R}-1|} d_{\text{spd}}(R_y, R_z)$, $R_y, R_z \in \mathbf{R}$, $R_y \neq R_z$ respectively. The average intra and inter-region temporal distances are defined similarly, except the modified Euclidean distance measure replaces d_{spd} .

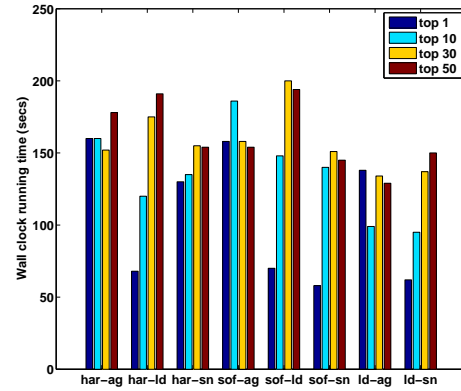
8.3. Synthetic (Localised Changes) Graph Comparison

In this subsection, we investigate the performance of ciForager and cSTAG using synthetic datasets. We compare the performance of ciForager and cSTAG across different clustering methods and different synthetic datasets. The various datasets vary the length of the sequence, the number of synCCs (synchronised connected components), the percentage of changing edges, and the size of the generated graphs. All these parameters affect the running speed of the algorithms. The synthetic datasets used are summarised in Table III. To reduce the bias, for each of the synthetic dataset, we generated three different sets of data for each parameter setting and present the average (and standard deviation) of the results. More details will be provided about each set of datasets in the subsections that follow. Apart from the clustering methods evaluation, we use the best clustering algorithm of cSTAG (*leaderFollower-singleLinkage*) as the benchmark to compare against ciForager. We first present the running time results, then the accuracy results.

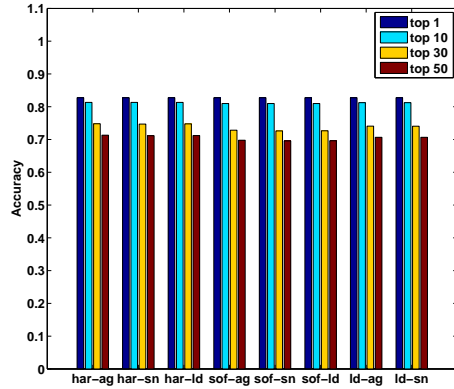
8.3.1. Running Time Evaluation.



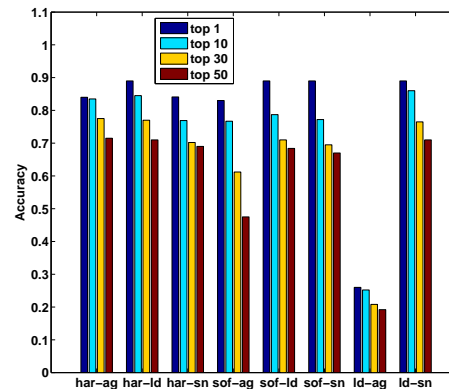
(a) Timing comparison for ciForager.



(b) Timing comparison for cSTAG.



(c) Accuracy comparison for ciForager.



(d) Accuracy comparison for cSTAG.

Fig. 7: Timing and accuracy comparison of ciForager against cSTAG. Note that the time scales in Figures 7a and 7b are different. The clustering algorithms labelled *har-ag*, *har-ld*, *har-sn* are the hard approach with {averageLinkage, leaderFollower, singleLinkage} clustering methods; *sof-ag*, *sof-ld*, *sof-sn* are the soft approach with {averageLinkage, leaderFollower, singleLinkage} clustering methods; *ld-ag* and *ld-sn* are the sequential approach with {leaderFollower + averageLinkage, leaderFollower + singleLinkage}. As cSTAG and ciForager consist of a number of parameter configurations, we report the average of the best, top 10, top 30 and top 50 results, in terms of accuracy, with 100-120 configurations for each clustering algorithm.

Varying the clustering methods. Any clustering algorithm can be used in cSTAG and ciForager, hence it is important to evaluate the sensitivity of ciForager and cSTAG across different clustering methods. We evaluate the eight different clustering algorithms proposed for cSTAG [Chan et al. 2008].

The synthetic datasets used to evaluate these results were the same ones used to compare cSTAG against regHunter in [Chan et al. 2009]. The parameters for generat-

ing these datasets are $\text{seqLen} = 30$, $\text{minSpaSep} = 3$, $\text{minTemSep} = 3$ and $\text{minEvtSep} = 3$.

cSTAG and ciForager have a number of parameters. To avoid bias, we varied the parameters and report the average performance of the top, top 10, top 30 and top 50 most accurate results for each clustering method (100-120 parameter configurations per method). The timing results are averaged over the same ordering. We do not show the error bars in these bar charts as we found that they clutter the charts.

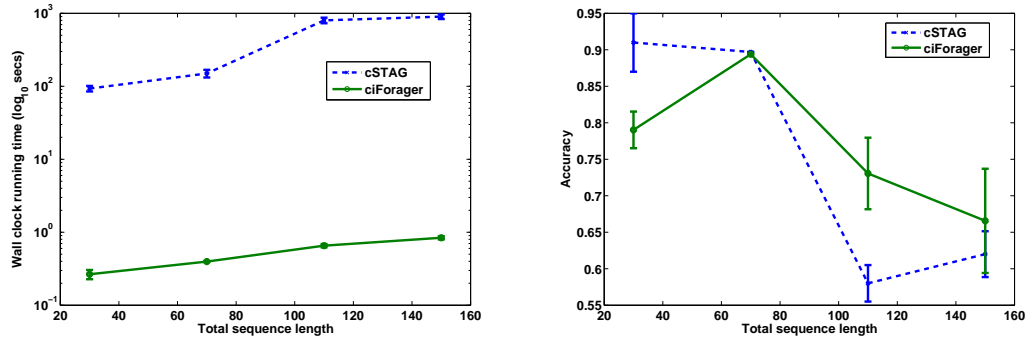
Consider the running time results for ciForager and cSTAG, illustrated in Figures 7a and 7b respectively. Note that the timing scales for Figures 7a and 7b are different. ciForager is 100-800 times faster than cSTAG, across all results and across all clustering methods. These results show that ciForager is considerably faster than cSTAG across different clustering methods and parameter settings.

In addition, note that the timing of ciForager is consistent against the clustering methods, suggesting that ciForager and the idea of synCCs is agnostic to the clustering approach employed. On the other hand, the results of cSTAG does not show this consistency across clustering methods.

Varying the number of snapshots in the sequence. As the sequence length increases, the number of windows analysed grow linearly, hence the running time of the algorithms should increase linearly also. To investigate the effect of increasing the sequence length on running time and accuracy, we generated datasets with four different sequence lengths (30, 70, 110, 150 snapshots). In addition, to maintain the average number of changing edges between any pair of consecutive snapshots, we linearly increase the life span of the generated regions in line with the increase in the sequence length. This dataset is labelled *synGenSeq* in Table III.

Consider Figure 8a, which shows the total running time of each algorithm. First, we confirm that the running time of ciForager is linear. The running times for the sequence lengths 30, 70, 110, 150 are 0.266s, 0.397s, 0.655s and 0.840s respectively. We did a linear regression fit, and obtained the equation $y = 0.0049x + 0.0940$, with $R^2 = 0.9863$, which shows that the running time of ciForager is indeed linear with the sequence length.

Next, we compare the running times of ciForager and cSTAG; ciForager is at least two orders of magnitude faster than cSTAG, with the gap growing as the sequence length increases. This is because the average number of synCCs per snapshot is increasing much more slowly than the average number of changing edges. As Equations 5 show, the running time of ciForager is dependent on the number of synCCs per snapshot while the running time of cSTAG [Chan et al. 2008] is dependent on the number of changing edges per snapshot. As the number of changing edges grows much faster than the number of synCCs, this results in the increasing gap between the running times of the two algorithms.



(a) Timing comparison between ciForager and cSTAG.

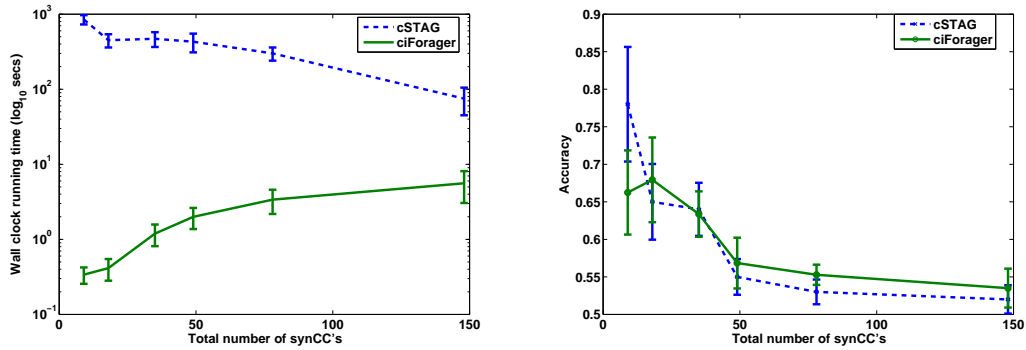
(b) Accuracy comparison between ciForager and cSTAG.

Fig. 8: Timing and accuracy comparison of ciForager against cSTAG when the total sequence length of the dynamic graph is varied. The size of the snapshots is approximately 500 vertices and 1500 edges. The total number of regions varied from 22 to 38, and the total number of unique changed edges is approximately 200 to 215.

Varying the average number of synCCs embedded in each snapshot. To investigate the effect of varying the average number of embedded synCCs on the performance of ciForager and cSTAG, we generated a random, connected graph of size 500 vertices and 5000 edges, and introduced an increasing number of regions into it. The number of regions is the lower bound on the number of synCCs. The sequence length was set at 30, $\text{minTempSep} = 0$, $\text{minEvtSep} = 3$, for all the generated dynamic graphs in this dataset. This dataset is labelled *introGenSec* in Table III.

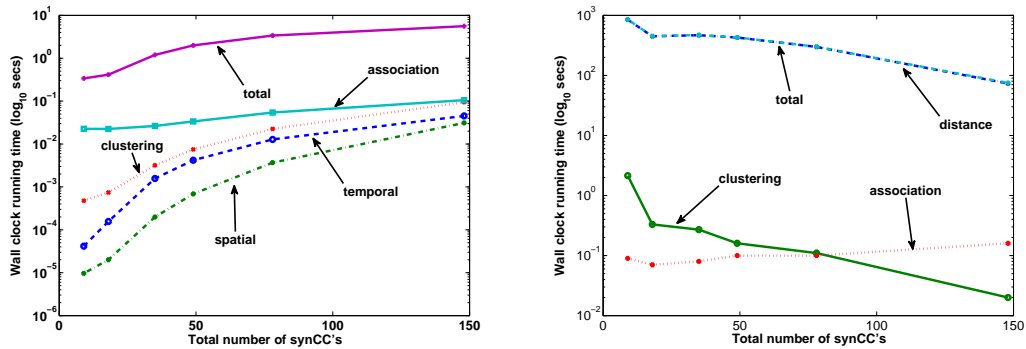
Figure 9a shows the running time of ciForager and cSTAG for this dataset. Again, ciForager is 15 to over 1000 times faster than cSTAG. However, as the number of synCCs increases, the gap in timing performance between ciForager and cSTAG decreases. Consider Figures 9c and 9d, which show the timing breakdowns of ciForager and cSTAG, respectively (to avoid clutter, we do not show the error bars). The running times of each algorithm are broken down into the time to compute the clustering (labelled *clustering* in the figures), the region association (labelled *association*), the temporal and spatial distances (labelled *temporal* and *spatial* in Figure 9c and combined together and labelled *distance* in Figure 9d) and the total running time (labelled *total*). For cSTAG, the average size of the regions found after the first stage of the sequential clustering (i.e., the regions are only temporally correlated, not necessarily spatially correlated) decreases as the number of synCCs increases; this results in fewer spatial distances calculated overall, as confirmed by the distance line in Figure 9d.

For ciForager, the total running time of ciForager shows an increasing trend. This is due to the increasing time to update the set of synCCs and the graph Voronoi, and the increasing time to compute the distances, clustering and association, where the first three are dependent on the number of synCCs. This is confirmed by Equation 5, which shows increasing time with respect to the number of synCCs. This experiment suggests that for this scenario, as the number of synCCs increases, the time to maintain the synCCs and compute the distances grow faster than the decrease in time due to the average size of cells decreasing.



(a) Timing comparison between ciForager and cSTAG.

(b) Accuracy comparison between ciForager and cSTAG.



(c) Timing breakdown of ciForager.

(d) Timing breakdown of cSTAG.

Fig. 9: Timing and accuracy comparison of ciForager against cSTAG when the total number of synCCs is varied. The size of the snapshots is approximately 500 vertices and 5000 edges. The sequence length is 30 for all datasets. Total number of unique changed edges is approximately 500.

Varying the average number of edges changing between any pair of consecutive snapshots. To determine the effect of the average number of changing edges between pairs of consecutive snapshots on timing and accuracy, we generated a graph of 500 vertices and 5000 edges, and varied the number of changed edges introduced, from 50 (5%) to 3500 (70%) of edges experiencing some change. The sequence length was set at 30 for all the generated datasets. This set of datasets is labelled *introGenPerc* in Table III.

Figure 10a illustrates the running times of ciForager and cSTAG. Again, the total running time of ciForager is two to four orders of magnitude faster than cSTAG. The difference in performance increases as more edges change. The rate at which the running time increases is gentler for ciForager than cSTAG. This is because the distance calculations dominate the running times for cSTAG, as the amount of spatial distance calculations increase at least quadratically with the number of changed edges. However, the time to compute the spatial distances is much less for ciForager than cSTAG, due to the fact that spatial distances are calculated on the much smaller meta-graph for ciForager. In addition, we found that the time to perform the clustering is about one

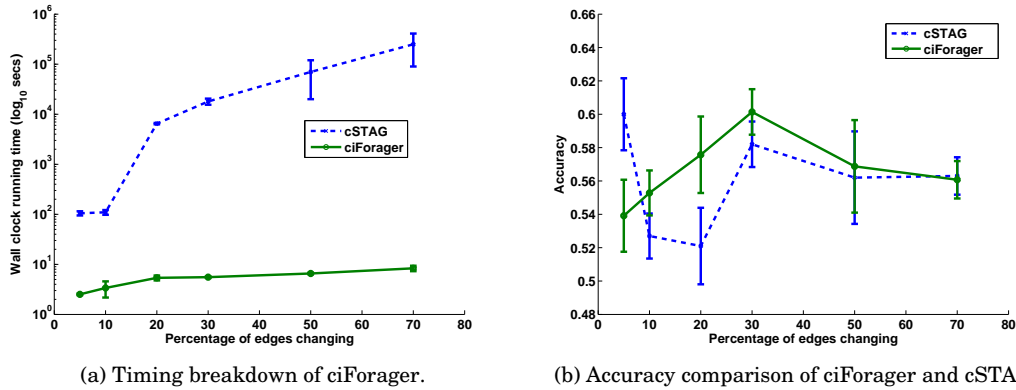


Fig. 10: Timing and accuracy comparison of ciForager against cSTAG when the percentage of edges in the original graph is varied. The size of the snapshots is approximately 500 vertices and 5000 edges. The sequence length is 30 for all datasets.

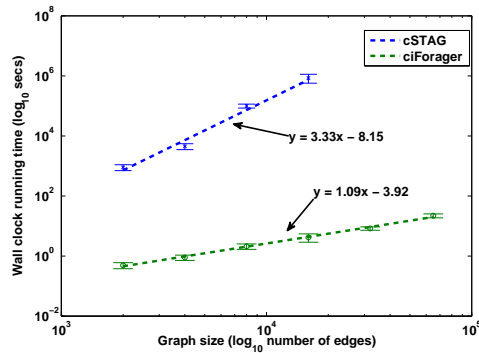


Fig. 11: Timing comparison of ciForager vs. cSTAG as the size of the scale-free graph increases. Sequence length was 30, and 20% of edges experienced change over the snapshot sequence.

order of magnitude faster for ciForager than cSTAG, reinforcing the fact that grouping into synCCs vastly decreases the number of actual points the clustering algorithms need to compare and group.

Varying the number of edges in the dynamic graphs. To investigate how the number of edges affects the running time of ciForager against cSTAG, we generated scale-free dynamic graphs whose edge count range from 2,000 to 65,000 edges. We generated scale-free graphs because many real-life graphs have scale-free characteristics. In addition, we wanted to evaluate the shortest path distances using a difficult dataset. Scale-free graphs cause shortest path searches to expand over much of the graph, resulting in long average running times. The number of vertices was kept at 30% of the number of edges, and the generated graphs were all connected. For each dynamic graph, 20% of the edges were changing and their sequence length was 30. This set of datasets is labelled *introGenSize* in Table III.

We attempted to run cSTAG on the 32,000 and 65,000 edge datasets, but cSTAG had not finished after 5.8 million seconds, therefore we terminated the tests. Figure 11 shows the running time of ciForager and cSTAG as the size of the graph increases, in log-log scale. We show the lines of best fit for the results of both algorithms, and the slope of these lines indicate that cSTAG scales cubically with the number of edges, while ciForager has almost linear scalability. For cSTAG, we found that the time to perform the region associations and calculate the spatial distances were the dominant factors in the increasing running time. For ciForager, it was the time to compute the spatial distances between synCCs, but it is still a fraction of the time it takes cSTAG to compute.

8.3.2. Accuracy Evaluation. In this subsection, we evaluate the accuracy of ciForager and cSTAG for the *synGen003*, *synGenSeq*, *synGenScc* and *synGenPerc* datasets. Generally, the accuracy of ciForager is comparable to the accuracy of cSTAG.

Varying the clustering methods. Consider Figures 7c and 7d, which show the accuracy results for ciForager and cSTAG respectively. The best accuracy result for ciForager is a few percent lower than cSTAG for some of the clustering methods. The reason for this is as follows. Some of the dataset generated have regions that have a small diameter (i.e., small shortest path distance among the changed edges). In cSTAG, when clustering the edges in the spatial dimension, we are considering the pairwise distance between edges. Because the regions have small diameter, they are clustered correctly. In ciForager, we first group edges into synCCs and then cluster the synCCs. To ensure a direct comparison, we used the same parameter settings, including the same set of clustering parameters. These clustering parameters were designed for the clustering of cSTAG and not ciForager, hence when clustering synCCs rather than edges, the synCCs can be mistakenly grouped into the same cluster or region, occasionally leading to more accurate results for cSTAG. Even with this disadvantage, ciForager still achieves accuracies of over 80%.

In addition, the ciForager results show significantly less variability within any of the clustering methods. Also, across all clustering methods, the results for ciForager vary much less than cSTAG (compare the results of ciForager, leaderFollower-averageLinkage against cSTAG, leaderFollower-averageLinkage for example). This indicates that ciForager is less sensitive to the choice of parameters and clustering method than cSTAG, making it simpler to use and optimise.

Varying the number of snapshots in the sequence. Consider Figure 8b, which shows the accuracy comparison of ciForager and cSTAG. Apart from the seqLen = 30 result, ciForager is of comparable or better accuracy than cSTAG. This indicates that the incremental nature of ciForager does not result in a loss of accuracy, even for longer sequence lengths.

Varying the average number of synCCs embedded in each snapshot. Consider Figure 9b, which shows the accuracy comparison of the two algorithms. ciForager has comparable accuracy to cSTAG, apart from the nine synCC result.

Varying the average number of edges changing between any pair of consecutive snapshots. Consider Figure 10b, which shows the accuracy results for ciForager and cSTAG when the percentage of edges changing is varied. The accuracy of ciForager is slightly higher than cSTAG almost over all edge percentage values, apart from the 5% value. Despite not optimising the parameters for ciForager, this suggests that the average effect of first grouping changed edges into synCCs can lead to accuracy improvements.

8.3.3. Synthetic (Localised Changes) Dataset Evaluation Summary. These results indicate that ciForager is highly scalable, and its running time is dependent on the average

Algorithm	Intra		Inter		Running time	
	Temporal	Spatial	Temporal	Spatial	Time (secs)	% of cSTAG
<i>ciForager</i>	0.011859	0.340575	0.968289	0.473339	859	1.59%
<i>cSTAG</i>	0.27145	0.39723	0.962554	0.62065	34402	100%

Table IV: Results for the best intra-region distance, for the effect of Hurricane Katrina on the BGP connectivity graph. Lower intra and higher inter-region distances are more accurate.

Algorithm	Intra		Inter		Running time	
	Temporal	Spatial	Temporal	Spatial	Time (secs)	% of cSTAG
<i>ciForager</i>	0.011859	0.340575	0.968289	0.473339	859	1.55%
<i>cSTAG</i>	0.46870	0.51435	0.96283	0.64748	35209	100%

Table V: Results for the best inter-region distance, for the effect of Hurricane Katrina on the BGP connectivity graph. Lower intra and higher inter-region distances are more accurate.

number of synCCs per snapshot, rather than the sequence length. In addition, ciForager is up to 10^6 times faster than cSTAG, primarily because of large reductions in the time required to compute the shortest path distances and to perform the clustering. The speed advantage of ciForager over cSTAG grows as the size of the analysed graphs increases. Because ciForager is much faster than cSTAG, it was able to process larger graphs that cSTAG cannot handle, as shown by the inability of cSTAG to complete the 32,000 and 64,000 edge tests.

8.4. BGP Graph Evaluation

In this subsection, we evaluate the timing and accuracy performance of ciForager and cSTAG on the Internet routing connectivity BGP graph. We analyse two versions of the connectivity BGP graphs - the US and global versions. We use the US portion of the BGP graph to compare the timing and accuracy of ciForager and cSTAG. cSTAG was unable to analyse the global version of the BGP graph due to its high memory usage and long running times. Therefore, we use ciForager to analyse the global portion to demonstrate a useful application of the improved efficiency of ciForager over cSTAG.

The BGP graphs do not have a known set of regions, hence we use the internal validation methods to evaluate the accuracy of ciForager and cSTAG. We compare the intra/inter-region accuracy and running time results for the US and global portions of the Internet. Table IV shows the best intra-regional distance results for the US BGP graph for ciForager and cSTAG. Table V shows the best inter-regional distance results. Both sets of results show that ciForager is significantly more accurate in terms of intra-region distance and approximately 40–41 times faster. The inter-region spatial distance for ciForager is lower than cSTAG due to synCC formation, which might inadvertently group edges together that would not be grouped together if considered individually.

The lower running time savings of ciForager over cSTAG are due to the large number of background changes in the graph. The speed advantage of ciForager over cSTAG is due to the ability of ciForager to localise change updates. However, the background changes in the BGP graphs are generally randomly distributed throughout the graph, which limits the ability to localise updates, leading to the diminished efficiency advantage of ciForager over cSTAG for this scenario.

Consider Figure 12a, which shows the number of changed vertices and edges experienced across subsequences that are six snapshots long. The figure shows that the num-

Algorithm	Running time (secs)		
	Pre-landfall	Landfall	Post-landfall
<i>ciForager</i>	25.7	131.0	76.1
<i>cSTAG</i>	849.4	10781.4	4912.8
Speedup (times)	33.1	81.3	64.6

Table VI: A breakdown comparison of the running times of *ciForager* and *cSTAG* for the pre-landfall, landfall and post-landfall periods of Hurricane Katrina.

ber of connectivity failures spiked when Hurricane Katrina made landfall. In addition, in [Chan et al. 2008], we found the failure regions have strong topological locality. This means that the updates required due to changes in the regions tend to be localised, hence *ciForager* should be fastest over periods of focused failure activity.

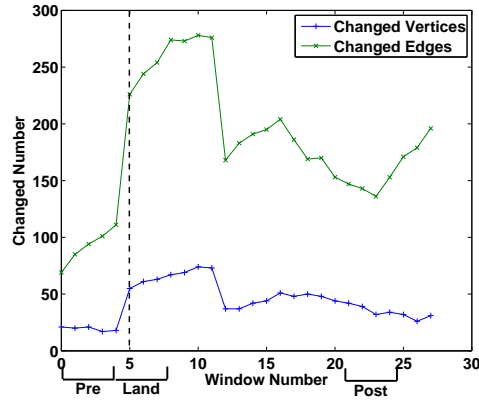
Hence, to evaluate if the speed advantage of *ciForager* over *cSTAG* is indeed restricted by the random background changes, we segmented the sequence of snapshots into three periods (see Figure 12a): before the landfall of Hurricane Katrina (*pre-landfall*), during the landfall of Hurricane Katrina (*landfall*), and after the landfall (*post-landfall*). The number of random, background changes is dominant in the pre-landfall period, hence among the three periods, we expect the advantage of *ciForager* over *cSTAG* to be the least. The number of localised changes is dominant in the landfall period, hence we expect *ciForager* to have greatest speed advantage over *cSTAG*. Finally, the network is in a recovery phase in the post-landfall period and therefore has fewer localised changes than during the landfall period, but still higher than the pre-landfall period. Note that the landfall and post periods are deliberately non-continuous. We want to choose a period whose amount of localised change fall between the two other periods.

These observations are evident in Figures 12b and 12c. Figure 12b shows the number of synCCs for each time window and Figure 12c shows the size of the three largest synCCs for each time window. Figures 12b and 12c indicate that the number of synCCs is largest and the size of the three largest synCCs are relatively small during the pre-landfall period. This suggests that most of the synCCs during this period are of a random nature, and most likely to be background changes. Compare this with the number of synCCs and the size of the three largest synCCs during the landfall period. The number of synCCs is minimal, and the size of the three largest synCCs are relatively large. This suggests that the changes tend to form connected components and are of a more localised nature. Finally, the number of synCCs and the size of the three largest synCCs during the post-landfall period is in-between the other two periods, suggesting that during this period, some of the changes are localised, but there are also a significant number of random background changes.

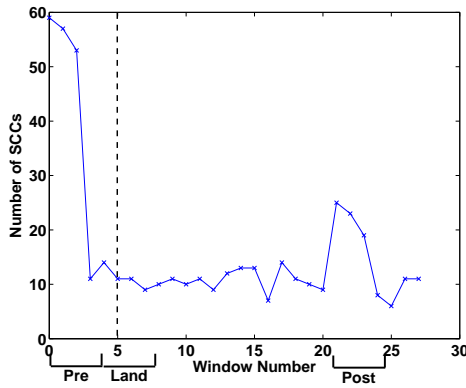
The running time results for each period are shown in Table VI. As Table VI shows, the speed advantage of *ciForager* over *cSTAG* is maximal over the landfall period (81.3 times), while least over the pre-landfall period (33.1 times) and 64.6 times for the post-landfall period. This further reinforces our hypothesis, and shows that *ciForager* has maximal advantage over *cSTAG* when localised changes are the dominant type of change.

In summary, even with a larger, real-life graph that experiences significant change, *ciForager* is still much faster than *cSTAG*, as well as actually being more accurate than *cSTAG*.

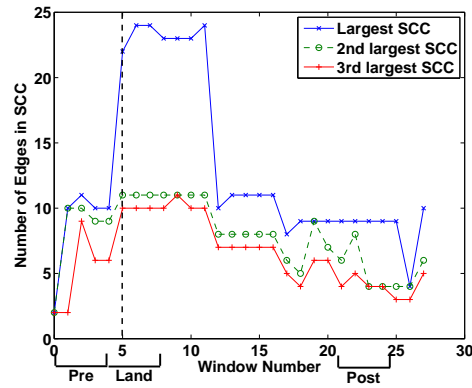
Global BGP Graph Results. In this section, we present two regions that were discovered by *ciForager* when analysing the whole BGP graph. *cSTAG* was not about to analyse the whole BGP graph due to its slow speed and very high memory consump-



(a) Number of changed edges and vertices in each time window.



(b) Number of synCCs found over each time window.



(c) Sizes of the three largest synCC found in each time window.

Fig. 12: Number of changed edges and vertices, number of synCCs and the sizes of the three largest synCCs found in each time window in the US portion of the BGP graph during the period 28 August to 31 August 2005. The dotted line signifies the landfall of Hurricane Katrina. The labels *pre*, *land* and *post* represent the periods *pre-landfall*, *landfall* and *post-landfall* respectively.

tion. Table VII summarises the two new regions found. Both represent instability regions, one during the time of the landfall of Hurricane Katrina (between snapshots 11 and 12), and the other was during the subsequent unstable, recovery periods after the landfall. Both regions consist of incident vertices that are predominately European ASs, and indicates connectivity instability in the US cascading to Europe. These regions show that a failure event in the US will not only affect the connectivity in the US portion of the network [Chan et al. 2008], but also cause instability in the European and other parts of the network. In addition, the large size of the regions indicate the fragility of the connectivity network to significant localised changes to its US core. In contrast, these regions were undiscovered by cSTAG because it cannot scale up to the global BGP graph. This demonstrates the added benefit of increased efficiency of ciForager.

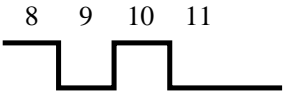
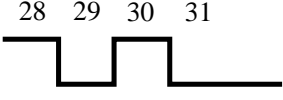
Region	No. of Edges	Change Waveform	Comments
Sat-C1	114		Instability region, mostly consisting of European ASs. Number of ASs by region – European: 95, US: 26, Asia Pacific: 7, other: 40.
Sat-C2	123		Instability region, mostly consisting of European ASs. Number of ASs by region – European: 101, US and Canadian: 36, Asia Pacific: 2, other: 24.

Table VII: Characteristics of two new discovered regions of correlated change for the whole BGP graph over the period of the Hurricane Katrina landfall.

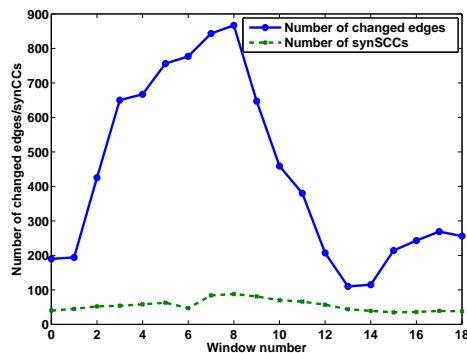


Fig. 13: Number of changed edges and synCCs in each time window, for the World Cup dataset.

8.5. 1998 World Cup Graph Evaluation

In this section, we evaluate another real dataset with a large number changes that occur in bursts.

Consider Figure 13, which shows the number of changed edges per window and the number of synCCs per window. The window size is 6 (same as in [Chan et al. 2008]). As this figure illustrate, even though this World Cup dataset is much smaller than the BGP dataset, it has many more changes. In addition, the ratio of synCCs to changed edges is higher than the BGP dataset, suggesting this dataset has many distributed changes. Analysing the algorithms on this dataset should give us further confidence of the performance of ciForager in this distributed change data type of data.

Tables VIII and IX show the timing breakdown of ciForager and cSTAG respectively. Despite the bursty nature of the dataset and the relatively large number of synCCs to changed edges, ciForager is still more than 70 times faster than cSTAG. This suggests that the computation of the shortest paths over the meta-graph has a significant

Temporal	Spatial	Clustering	Association	Total
0.022417	0.068927	0.102661	0.064997	11.7674

Table VIII: Running time breakdown (secs) for ciForager on the World Cup dataset.

Distance	Clustering	Association	Total
547.597	321.877	0.525479	884.981

Table IX: Running time breakdown (secs) for cSTAG on the World Cup dataset.

speedup effect, as most of the speedup resulted from savings in the distance and clustering calculations. This is another demonstration that by first grouping changed edges into synCCs and then incrementally updating them and the shortest path distance, the running time can be reduced significantly.

In conclusion of the evaluation, ciForager is up to 10^6 times faster than cSTAG for graphs with more localised changes. For distributed, random changes, ciForager can still be up to 70 times faster than cSTAG. In addition, the speed advantage of ciForager over cSTAG grows as the size of the graphs analysed increases. This means ciForager can scale up to very large graphs that cSTAG cannot analyse. This was demonstrated by the inability of cSTAG to analyse the global BGP graph in a timely and memory limited manner.

9. CONCLUSION

In this paper, we have introduced a new framework called ciForager to discover regions of correlated change. We introduced several major efficiency improvements to existing region discovery frameworks. These improvements include the introduction of synchronised connected components to reduce the number of temporal and spatial distances that need to be computed, the introduction of a graph Voronoi structure to reduce the number of shortest path recalculations required when new snapshots are considered, and the introduction of equivalence classes among the change waveforms to reduce the number of shortest path distance computations. Furthermore, ciForager proposed efficient methods to incrementally update the set of synchronised connected components and the graph Voronoi as new snapshots arrive.

Using datasets with localised changes, we have shown that the improvements of ciForager resulted in up to six orders of magnitude speedup over a previous method (cSTAG), with ciForager scaling linearly with the size of the graphs analysed. We have evaluated the effect that the length of the graph sequence, the average size of the synCCs, the number of synCCs and the size of the graphs have on the running time and accuracy of ciForager and cSTAG. Furthermore, we have shown that ciForager takes 1.55% of the time cSTAG takes to analyse the US portion of the BGP connectivity graphs, and only 1.2% of the time cSTAG takes to analyse the sub-period covering the landfall of Katrina, which had more localised changes. When analysing the global BGP graph, we have also discovered two regions corresponding to the instability of European ASs, caused by the instability in the US. cSTAG and regHunter were unable to analyse the whole BGP graph due to its inability to scale to very large graphs well. In addition, ciForager is 70 times faster when analysing the bursty World Cup 98 website access graph. The high scalability of ciForager means it is a practical tool for discovering regions of correlated changes in very large graphs and on challenging real world problems.

For future work, we would like to incorporate dynamic shortest path algorithms and evaluate the trade-off between the possible decrease in running time and increased memory usage, and also derive with an upper bound on using graph Voronoi to com-

pute the shortest path distances. In addition, we would like to investigate other measures of spatial correlation. One example is the amount of shared boundary between a pair of synCCs. Finally, we would like to extend our analysis to weighted graphs. This can be easily accommodated into the ciForager framework by extending the change waveforms distance measure to incorporate weights.

ACKNOWLEDGMENTS

This research was funded by NICTA and the National Institute of Informatics. Furthermore, we would like to thank the University of Oregon for making the RouteView data publicly available, and Prof. Christos Faloutsos and the two anonymous reviewers for their helpful comments.

REFERENCES

- AGGARWAL, C. C., HAN, J., WANG, J., AND YU, P. S. 2003. A framework for clustering evolving data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases*. 81–92.
- ALI, M. H., MOKBEL, M. F., AREF, W. G., AND KAMEL, I. 2005. Detection and tracking of discrete phenomena in sensor-network databases. In *Proceedings of the 17th International Conference on Scientific and Statistical Database Management*. 163–172.
- ARLITT, M. AND JIN, T. 1999. Workload characterization of the 1998 World Cup website. Tech. Rep. HPL-99-35R1, Hewlett-Packard Labs. September.
- BAE, E., BAILEY, J., AND DONG, G. 2010. A clustering comparison measure using density profiles and its application to the discovery of alternate clusterings. *Data Mining and Knowledge Discovery* 21, 427–471.
- BOGDANOV, P., MONGIOVÌ, M., AND SINGH, A. K. 2011. Mining heavy subgraphs in time-evolving networks. In *Proceedings of the 11th International Conference on Data Mining*. 81–90.
- BORGWARDT, K. M., KRIEGEL, H.-P., AND WACKERSREUTHER, P. 2006. Pattern mining in frequent dynamic subgraphs. In *Proceedings of the 6th International Conference on Data Mining*. 818–822.
- CELIK, M., SHEKHAR, S., ROGERS, J. P., SHINE, J. A., AND YOO, J. S. 2006. Mixed-drove spatio-temporal co-occurrence pattern mining: A summary of results. In *Proceedings of the 6th International Conference on Data Mining*. 119–128.
- CHAKRABARTI, D., KUMAR, R., AND TOMKINS, A. 2006. Evolutionary clustering. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 554–560.
- CHAN, J., BAILEY, J., AND LECKIE, C. 2008. Discovering correlated spatio-temporal changes in evolving graphs. *Knowledge and Information Systems* 16, 1, 53–96.
- CHAN, J., BAILEY, J., AND LECKIE, C. 2009. Using graph partitioning to discover regions of correlated change spatio-temporal change in evolving graphs. *Intelligent Data Analysis* 13, 5, 755–793.
- CHI, Y., SONG, X., ZHOU, D., HINO, K., AND TSENG, B. L. 2007. Evolutionary spectral clustering by incorporating temporal smoothness. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 153–162.
- CLARE, S. 1997. Functional mri : Methods and applications. Ph.D. thesis, University of Nottingham.
- CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms*. MIT Press.
- DE BERG, M., CHEONG, O., VAN KREVELD, M., AND OVERMARS, M. 2008. *Computational Geometry: Algorithms and Applications*. Springer-Verlag.
- DU, N., WANG, H., AND FALOUTSOS, C. 2010. Analysis of large multi-modal social networks: Patterns and a generator. In *Machine Learning and Knowledge Discovery in Databases*. Lecture Notes in Computer Science Series, vol. 6321. Springer Berlin / Heidelberg, 393–408.
- ELNEKAVE, S., LAST, M., AND MAIMON, O. 2007. Incremental clustering of mobile objects. *IEEE Computer Society*, Los Alamitos, CA, USA, 585–592.
- ERWIG, M. 2000. The graph voronoi diagram with applications. *Networks* 36, 3, 156–163.
- GIBSON, D., KUMAR, R., AND TOMKINS, A. 2005. Discovering large dense subgraphs in massive graphs. In *Proceedings of the 31st International Conference on Very Large Data Bases*. 721–732.
- HALKIDI, M., BATISAKIS, Y., AND VAZIRGIANNIS, M. 2001. On clustering validation techniques. *Journal of Intelligent Information Systems* 17, 2–3, 107–145.
- HONIDEN, S., HOULE, M. E., AND SOMMER, C. 2009. Balancing graph voronoi diagrams. In *Proceedings of the 2009 Sixth International Symposium on Voronoi Diagrams*. *IEEE Computer Society*, 183–191.

- JAIN, A. K. AND DUBES, R. C. 1998. *Algorithms for Clustering Data*. Prentice-Hall, Inc.
- KUMAR, R., NOVAK, J., RAGHAVAN, P., AND TOMKINS, A. S. 2003. On the bursty evolution of blogspace. In *Proceedings of the 12th International Conference on World Wide Web*. 568–576.
- KUMAR, R., NOVAK, J., AND TOMKINS, A. S. 2006. Structure and evolution of online social networks. In *Proceedings of the 12th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (poster)*.
- LAHIRI, M. AND BERGER-WOLF, T. Y. 2010. Periodic subgraph mining in dynamic networks. *Knowledge and Information Systems* 24, 467–497.
- LAUW, H. W., LIM, E.-P., TAN, T.-T., AND PANG, H.-H. 2005. Mining social networks from spatio-temporal events. In *Workshop on Link Analysis, Counterterrorism and Security*.
- LESKOVEC, J., KLEINBERG, J., AND FALOUTSOS, C. 2005. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. 177–187.
- LUENBERGER, D. 2003. *Linear and Nonlinear Programming*. Kluwer Academic Publishers.
- MEILA, M. 2003. Comparing clusterings by the variation of information. In *Proceedings of the Conference on Learning Theory and Kernel Machines*. 173–187.
- SHOUBRIDGE, P. J., KRAETZL, M., WALLIS, W. D., AND BUNKE, H. 2002. Detection of abnormal change in a time series of graphs. *Journal of Interconnection Networks* 3, 1-2, 85–101.
- STEINDER, M. AND SETHI, A. S. 2004. A survey of fault localization techniques in computer networks. *Science of Computer Programming* 53, 2, 165–194.
- SUN, J., PAPADIMITRIOU, S., YU, P. S., AND FALOUTSOS, C. 2007. Graphscope: Parameter-free mining of large time-evolving graphs. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 687–696.
- SUN, J., TAO, D., AND FALOUTSOS, C. 2006. Beyond streams and graphs: dynamic tensor analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, USA, 374–383.
- THON, I., LANDWEHR, N., AND RAEDT, L. D. 2008. A Simple Model for Sequences of Relational State Descriptions. In *Proceedings of the 19th European Conference on Machine Learning*. 506–521.
- wc98trace. 1998 World Cup website access traces. <http://ita.ee.lbl.gov/html/contrib/WorldCup.html>
- YANG, H., PARTHASARATHY, S., AND MEHTA, S. 2005. A generalized framework for mining spatio-temporal patterns in scientific data. In *Proceedings of the eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. 716–721.
- ZHOU, A., CAO, F., QIAN, W., AND JIN, C. 2007. Tracking clusters in evolving data streams over sliding windows. *Knowledge and Information Systems*, 181–214.
- ZHOU, D., LI, J., AND ZHA, H. 2005. A new mallows distance based metric for comparing clusterings. In *Proceedings of the 22nd International Conference on Machine Learning*. 1028–1035.

A. PROOF OF LEMMA 5.11 (SYNCC SPLITTING DUE TO DIFFERENT EQUIVALENCE CLASSES)

From its definition, after extension, an existing synCC can split due to its member edges a) linked to different equivalence classes; and/or b) no longer forming a connected component.

To show the proof, we show that case b) is not possible after an existing synCC is extended.

From the definition of a change waveform, the equivalence class of s_j must contain at least one $0 \rightarrow 1$ or $1 \rightarrow 0$ change. Hence, the waveform of the equivalence class must contain a '1' in one of its values. From the definition of a synCC, the edges in s_j must form a connected component before extension. After extending the equivalence class with '0' or '1', the extended equivalence class must still contain a '1' in one of its values. Therefore the member edges will still form a connected component over the union graph. Therefore, case b) is not a possible state after extension. \square

B. PROOF OF LEMMA 6.11 (MERGING OF TWO SYNCCS)

We first prove $cell_{m1} \cup cell_{m2} \rightarrow cell_m$. Let $a \in cell_{m1}$. Therefore, $d(a, s_{m1}) \leq d(a, s_p), \forall s_p \in S_{k-1}, p \neq i$. Since $d(a, s_{m1}) \leq d(a, s_p)$, then $d(a, s_{m1}) = d(a, s_{m1} \cup s_{m2}) = d(a, s_m)$, hence $d(a, s_m) \leq d(a, s_p)$ and $a \in cell_m$. A similar derivative can be used

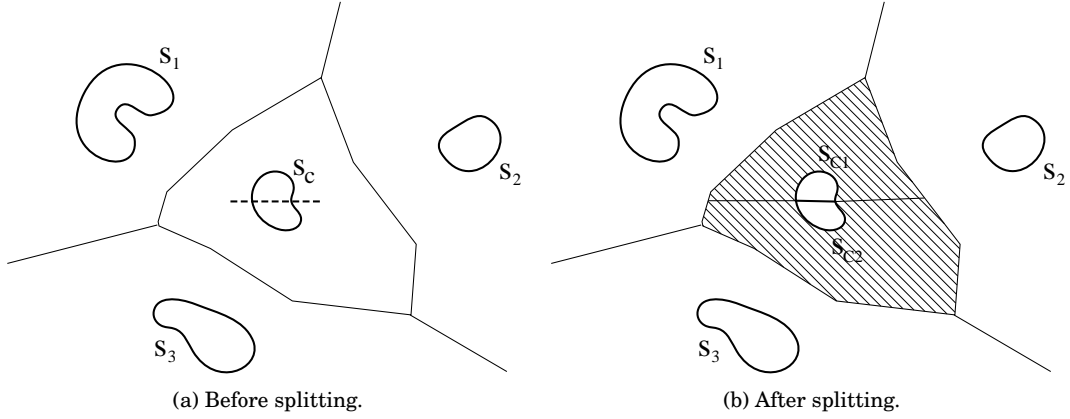


Fig. 14: An example from a portion of a graph Voronoi diagram, showing a subset of the embedded synCCs and the cell boundaries before and after synCC s_c is split. The solid black lines represent the cell boundaries, and the shaded, diagonally striped area represent the affected area. In addition, the dotted line in Figure 14a represent where the split in s_c is about to occur.

to show that $b \in cell_{m2}$ is true. Therefore, $a \in cell_m$ and $b \in cell_m$ and hence, $cell_{m1} \cup cell_{m2} \rightarrow cell_m$.

Now we prove the other direction, by proving the contrapositive ($\neg(cell_{m1} \cup cell_{m2}) \rightarrow \neg cell_m$). Let $a \notin cell_{m1}, a \notin cell_{m2}$, hence there exists a synCC s_p in $S[k]$ where $d(a, s_{m1}) \geq d(a, s_p)$ and $d(a, s_{m2}) \geq d(a, s_p)$. Recall $s_m = s_{m1} \cup s_{m2}$, hence $d(a, s_m) \geq d(a, s_p)$. From the definition of a cell, a therefore cannot be in $cell_m$.

As $cell_m = cell_{m1} \cup cell_{m2}$ and there are no changes to the cells then:

$$B^{k+1}(m, p) = \begin{cases} B^k(m1, p) & s_p \in N^{k+1}(m1), s_p \notin N^{k+1}(m2) \\ B^k(m2, p) & s_p \notin N^{k+1}(m1), s_p \in N^{k+1}(m2) \\ B^k(m1, p) \cup B^k(m2, p) & s_p \in N^{k+1}(m1), s_p \in N^{k+1}(m2) \end{cases}$$

If there are no changes to the boundaries, then the shortest path distances do not change. For the case $s_p \in N^k(m1), s_p \in N^k(m2)$, the shortest path distance is the minimum of the two existing distances. Hence, the result for $SPD(., .)$ follows from the results from $B(., .)$.

C. INCREMENTAL UPDATING OF THE GRAPH VORONOI TO ADDRESS THE SPLITTING OF A SYNCC

LEMMA C.1. Let s_c be split into s_{c1} and s_{c2} , where $s_c = s_{c1} \cup s_{c2}$. Then:

- (1) The elements of the cell of s_c is redistributed between the cells of s_{c1} and s_{c2} after the split occurs; i.e., $cell_{c1} \cup cell_{c2} = cell_c$.
- (2) Only the shortest paths distances between synCCs that are neighbours to both s_{c1} and s_{c2} after the split might change. More formally, $\forall s_p \in S[k+1]$,

$$SPD^{k+1}(c1, p) = \begin{cases} SPD^k(c, p) & s_p \in N^{k+1}(c1), s_p \notin N^{k+1}(c2) \\ \geq SPD^k(c, p) & s_p \in N^{k+1}(c1), s_p \in N^{k+1}(c2) \end{cases}$$

$$SPD^{k+1}(c2, p) = \begin{cases} SPD^k(c, p) & s_p \notin N^{k+1}(c1), s_p \in N^{k+1}(c2) \\ \geq SPD^k(c, p) & s_p \in N^{k+1}(c1), s_p \in N^{k+1}(c2) \end{cases}$$

PROOF. We first prove $cell_c \rightarrow cell_{c_1} \cup cell_{c_2}$. Let $a \in cell_c$. Then $d(a, s_c) = d(a, s_{c_1} \cup s_{c_2}) = \min(d(a, s_{c_1}), d(a, s_{c_2})) \leq d(a, s_p), \forall s_p \in \mathcal{S}_k, p \neq c_1, p \neq c_2$. From the definition of a cell, a must belong to either $cell_{c_1}$ or $cell_{c_2}$, hence $cell_c \rightarrow cell_{c_1} \cup cell_{c_2}$.

We now prove the other direction by proving the contrapositive ($\neg cell_c \rightarrow \neg(cell_{c_1} \cup cell_{c_2})$). Let $a \notin cell_c$. Therefore, there exists a synCC s_p where $d(a, s_p) \leq d(a, s_c)$. Since $d(a, s_c) \leq d(a, s_{c_1})$ and $d(a, s_c) \leq d(a, s_{c_2})$, then a cannot be in $cell_{c_1}$ nor $cell_{c_2}$.

To prove the shortest path distance relationships, consider the changes in the boundaries:

$$B^{k+1}(c_1, p) = \begin{cases} B^k(c, p) & s_p \in N^{k+1}(c_1), s_p \notin N^{k+1}(c_2) \\ 0 & s_p \notin N^{k+1}(c_1), s_p \in N^{k+1}(c_2) \\ \subseteq B^k(c, p) & s_p \in N^{k+1}(c_1), s_p \in N^{k+1}(c_2) \end{cases}$$

From the definition of a boundary, $B^{k+1}(c, p) = B^k(c_1 \cup c_2, p) = B^k(c_1, p) \cup B^k(c_2, p)$. Therefore, $B^k(c_1, p) = B^{k+1}(c, p) - B^k(c_2, p) \cup (B^k(c_1, p) \cap B^k(c_2, p))$. For $s_p \in N^{k+1}(c_1)$ and $s_p \notin N^{k+1}(c_2)$, $B^k(c_2, p) = \emptyset$, hence $B^k(c_1, p) = B^k(c, p)$. For $s_p \in N^{k+1}(c_1)$ and $s_p \in N^{k+1}(c_2)$, $B^k(c_1, p) \subset B^{k+1}(c, p)$. Similar results hold for $B^{k+1}(c_2, p)$.

If the boundaries shrink, then the shortest path distances can only increase. Hence, the definitions of $SPD(., .)$ follows from the changes to the boundaries. \square

From Lemma C.1 and Figure 14, only the vertices and edges that were in $cell_c$ are reassigned to either $cell_{c_1}$ or $cell_{c_2}$. All other cells remain the same. In addition, the lemma shows that only the shortest path distances between synCCs that are neighbours to both s_{c_1} and s_{c_2} need to be recomputed.

Therefore, to handle splitting s_c , the distances and the parent markers of all the vertices and edges in $cell_c$ are first invalidated. We need to invalidate all the vertices and edges in the cells because we do not know which of s_{c_1} or s_{c_2} the edges or vertices formerly in s_c are closest to. Then Dijkstra's algorithm is rerun from s_j and s_i simultaneously, until the search reaches the former boundary of s_c . The boundaries of s_i and s_j will then be available from the Dijkstra search. From the boundary and cell information, the necessary neighbours and distances of s_i and s_j and their neighbours can be computed.

D. INCREMENTAL UPDATING OF THE GRAPH VORONOI TO ADDRESS THE CREATION OF A SYNCC

LEMMA D.1. *Let s_n be the new synCC. Let $\mathcal{S}[\gamma]$ denote the set of synCCs whose cells, before the appearance of s_n , would have overlapped with s_n ; i.e., $\mathcal{S}[\gamma] = \{s_i | cell_i^k \cap s_n \neq \emptyset\}$. Let $\mathcal{S}[aff]$ denote the set of synCCs that are neighbours of $\mathcal{S}[\gamma]$, including the synCCs in $\mathcal{S}[\gamma]$; i.e., $\mathcal{S}[aff] = \mathcal{S}[\gamma] \cup N^k(\mathcal{S}[\gamma])$. Then:*

- (1) *The set of cells affected by the creation of s_n is restricted to $\mathcal{S}[aff]$; i.e., $\bigcup_{s_i \in \mathcal{S}[aff]} cell_i^{k+1} \cup cell_n = \bigcup_{s_i \in \mathcal{S}[aff]} cell_i^k$.*
- (2) *Only the shortest path distances of synCCs that a) are between the neighbours of s_n , and b) were in $\mathcal{S}[aff]$, might change due to the creation of s_n . More formally, $\forall s_i \in N^{k+1}(n)$,*

$$SPD^{k+1}(i, p) = \begin{cases} SPD^k(i, p) & s_p \notin \mathcal{S}[aff] \\ \geq SPD^k(i, p) & s_p \in \mathcal{S}[aff] \end{cases}$$

PROOF. We first prove $\bigcup_{s_i \in \mathcal{S}[aff]} cell_i^k \rightarrow \bigcup_{s_i \in \mathcal{S}[aff]} cell_i^{k+1} \cup cell_n$. Let $a \in \bigcup_{s_i \in \mathcal{S}[aff]} cell_i^k$. Let s_i be the synCC in $\mathcal{S}[aff]$ that a is closest to. Therefore, $d(a, s_i) \leq d(a, s_l), \forall s_l \in \mathcal{S}[k]$.

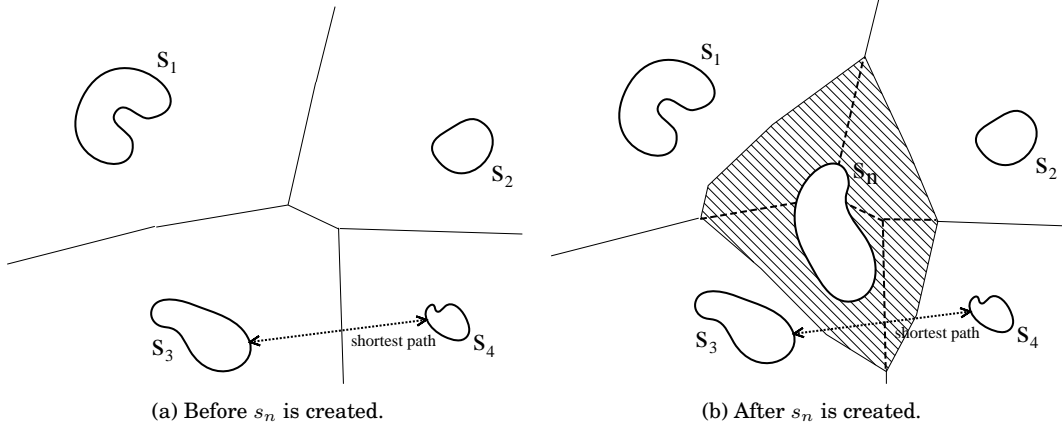


Fig. 15: An example from a portion of a graph Voronoi diagram, showing a subset of the embedded synCCs and the cell boundaries before and after the synCC s_n is created. The solid black lines represent the cell boundaries, and the shaded, diagonally striped area represent the affected area. In addition, the existing shortest path distance between s_3 and s_4 is represented by the dotted line with two arrowheads. The example shows how the creation of s_n can affect s_4 , even though its cell before the creation of s_n does not overlap with s_n itself.

In addition, $d(a, s_i \cup s_n) \leq d(a, s_i), \forall s_i \in (\mathcal{S}[k] \cup s_n) = \mathcal{S}[k+1]$. Therefore $a \in cell_i^{k+1} \cup cell_n$, and $a \in \bigcup_{s_i \in S_{aff}} cell_i^{k+1} \cup cell_n$.

We now prove the other direction ($\bigcup_{s_i \in S_{aff}} cell_i^{k+1} \cup cell_n \rightarrow \bigcup_{s_i \in S_{aff}} cell_i^k$). Let b be a member of $cell_n$ or a cell in $\mathcal{S}[aff]$, i.e., $b \in \bigcup_{s_i \in S_{aff}} cell_i^k \cup cell_n$.

If $b \in \bigcup_{s_i \in S_{aff}} cell_i^{k+1}$, then $b \in \bigcup_{s_i \in S_{aff}} cell_i^k$, as $cell_i^{k+1} \subseteq cell_i^k$.

Consider the other case, $b \in cell_n$. We wish to show that $cell_n = \bigcup_{s_i \in S_{aff}} cell_i^k \cap cell_n$, or $cell_n$ can be partitioned among the cells of S_{aff} . This statement is equivalent to $\bigcup_{s_i \in S_{aff}} cell_i^k \cap cell_n \neq \emptyset$ and $\bigcup_{s_j \notin S_{aff}} cell_j^k \cap cell_n = \emptyset$. Hence, we show by contradiction that $\bigcup_{s_j \notin S_{aff}} cell_j^k \cap cell_n = \emptyset$ holds to show $cell_n$ can be partitioned among the cells of S_{aff} and $cell_n = \bigcup_{s_i \in S_{aff}} cell_i^k \cap cell_n$.

Assume the opposite is true, i.e., $\bigcup_{s_j \notin S_{aff}} cell_j^k \cap cell_n \neq \emptyset$, or there exists a synCC s_j that is not in $\mathcal{S}[aff]$ and whose cell have a non-empty overlap with $cell_n$, i.e., $cell_j^k \cap cell_n \neq \emptyset$. Let $h \in cell_j^k \cap cell_n$. Let s_p be a neighbouring scc of s_j , i.e., $s_p \in N^{k+1}(j)$, therefore $d(h, s_p) < d(h, s_n)$. s_p must exist, as between s_n and s_j there must be a synCC in $N(\mathcal{S}_\gamma)$ that is closer to s_j than s_n . Therefore $h \in cell_p^{k+1}$, which contradicts the initial assumption. Hence $cell_j^{k+1} \cap cell_n = \emptyset, \forall s_j \notin S_{aff}, \bigcup_{s_i \in S_{aff}} cell_i^k \cap cell_n \neq \emptyset$, and $cell_n = \bigcup_{s_i \in S_{aff}} cell_i^k \cap cell_n$. Therefore $b \in \bigcup_{s_i \in S_{aff}} cell_i^k$, and $\bigcup_{s_i \in S_{aff}} cell_i^{k+1} \cup cell_n \rightarrow \bigcup_{s_i \in S_{aff}} cell_i^k$ is true.

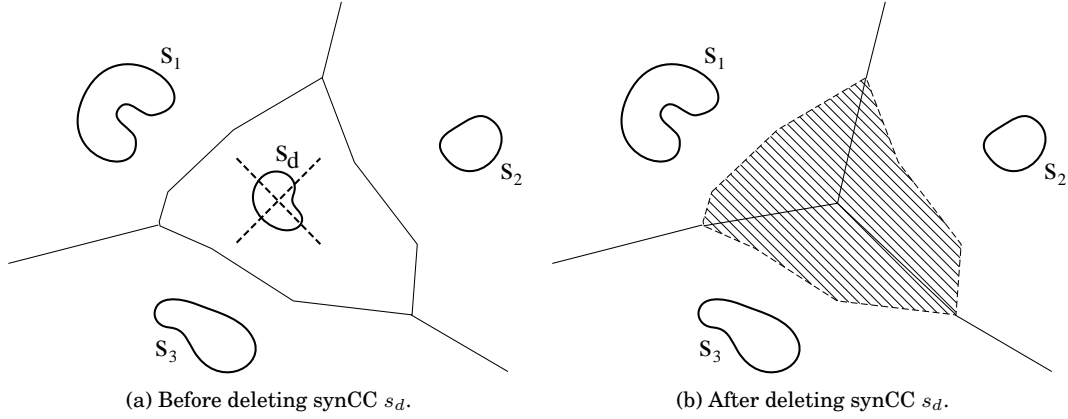


Fig. 16: An example from a portion of a graph Voronoi diagram, showing a subset of the embedded synCCs and the cell boundaries before and after the synCC s_d is deleted. The solid black lines represent the cell boundaries, and the shaded, diagonally striped area represent the affected area.

To prove the shortest path distance relationships, consider the changes in the boundaries:

$$B^{k+1}(i, p) = \begin{cases} B^k(i, p) & s_p \notin S_{aff} \\ \subseteq B^k(i, p) & s_p \in S_{aff} \end{cases}$$

The results of $B^{k+1}(i, p)$ are obvious from $cell_i^{k+1} \subseteq cell_i^k, \forall s_i \in S_{aff}$ and $cell_j^{k+1} = cell_j^k, \forall s_j \notin S_{aff}$.

The result for $SPD(., .)$ follows from $B^{k+1}(i, p)$. \square

Lemma D.1 and Figure 15 shows that the changes are confined to the cells of the synCCs in S_{aff} . S_{aff} is the set of synCCs whose cells either partially overlaps the new synCC s_n , or is a neighbour of an overlapping synCC. To handle creation of a new synCC s_n , we update the parents and distances of the vertices and edges in s_n and some of the affected synCCs in S_{aff} . The Dijkstra's algorithm is rerun from the new s_n . Existing distances and parent information are overwritten if the current distance of search is smaller than the existing distance. If not smaller, terminate that branch of search. When the search finishes, we would have grown a new cell around the new synCC. From the new cell $cell_n$, neighbours, boundaries and distances of s_n can be determined. Finally, all the boundaries and shortest path distances between all pairs of synCCs s_i and s_j , where $s_i, s_j \in N(s_n)$, must also be updated, because the cells of s_i and s_j can also be affected.

E. INCREMENTAL UPDATING OF THE GRAPH VORONOI TO ADDRESS THE DELETION OF A SYNCC

LEMMA E.1. *Let s_d be the deleted synCC. Then:*

- (1) *The elements in the cell of s_d are redistributed among the cells of the former neighbours of s_d ; i.e., $\bigcup_{s_i \in N^k(d)} cell_i^{k+1} = \bigcup_{s_i \in N^k(d)} cell_i^k \cup cell_d$;*

(2) *Only the shortest path distances between the former neighbours of s_d might change. More formally, $s_i \in N^k(d)$,*

$$SPD^{k+1}(i, p) = \begin{cases} SPD^k(i, p) & s_p \notin N^k(d) \\ \leq SPD^k(i, p) & s_p \in N^k(d) \end{cases}$$

PROOF. We first prove $\bigcup_{s_i \in N^k(d)} cell_i^k \cup cell_d \rightarrow \bigcup_{s_i \in N^k(d)} cell_i^{k+1}$. Let a be a member of either $cell_d$, or one of its neighbouring cells in $\mathcal{S}[k]$, i.e., $a \in \bigcup_{s_i \in N^k(d)} cell_i^k \cup cell_d$. If a is in a neighbouring cell of $cell_d$ in $\mathcal{S}[k]$, i.e., $a \in \bigcup_{s_i \in N^k(d)} cell_i^k$, then a must be in the same cell ($a \in \bigcup_{s_i \in N^k(d)} cell_i^{k+1}$), as $d(a, s_i) \leq d(a, s_p)$ still holds for $\forall s_p \in \mathcal{S}[k]$. For the case where a in $cell_d$ ($a \in cell_d$), note that $s_i \in N^k(d)$, hence $d(a, s_d) \leq d(a, s_i) \leq d(a, s_p)$, $\forall s_i \in N^k(d)$, $s_p \notin N^k(d)$, $i \neq p \neq d$. Hence when s_d and its associated $cell_d$ are deleted, a is closest to one of the synCCs in $N^k(d)$, or $a \in \bigcup_{s_i \in N^k(d)} cell_i^{k+1}$.

We now prove the other direction ($\bigcup_{s_i \in N^k(d)} cell_i^{k+1} \rightarrow \bigcup_{s_i \in N^k(d)} cell_i^k \cup cell_d$). Let $b \in \bigcup_{s_i \in N^k(d)} cell_i^{k+1}$. b must be have been closest to either s_d or one of its neighbouring synCC s_i , because $d(b, s_i) \leq d(b, s_p)$ or $d(b, s_d) \leq d(b, s_p)$, $\forall s_p \in \mathcal{S}[k]$. Therefore b must be in either $cell_d$ or one of its neighbouring cells, i.e., $b \in \bigcup_{s_i \in N^k(d)} cell_i^k \cup cell_d$.

To prove the shortest path distance relationships, consider the changes in the boundaries:

$$B^{k+1}(i, p) = \begin{cases} B^k(i, p) & s_p \notin N^k(d) \\ \supseteq B^k(i, p) & s_p \in N^k(d) \end{cases}$$

If a cell is not neighbouring s_d in $\mathcal{S}[k]$, then there is no change to its cell membership, and no changes to its boundaries with its neighbours. However, if a cell is a neighbour of $cell_d$, i.e., in $N^k(d)$, then it gets a portion of the redistribution of s_d , hence its boundaries with the other neighbouring cells of $cell_d$ can either remain constant or grow. The result for the changes in boundaries follows from these cases.

The results stem from $cell_i^{k+1} \supseteq cell_i^k$. The result for $SPD(.,.)$ follows from $B^{k+1}(i, p)$.
□

Lemma E.1 and Figure 16 show that the edges and vertices of the deleted cell must be redistributed among its neighbouring synCCs. Because of the redistribution, there can be changes to the neighbourhood information among the neighbours of the deleted synCC. Hence, the boundaries and shortest path distances among the neighbours need to be recalculated also.

Therefore, to handle deletion, we invalidate all the information in the vertices and edges in $cell_d$ and s_d . Then from the boundaries of s_d , we rerun Dijkstra's algorithm. When the algorithm terminates, the vertices and edges of $cell_d$ and s_d will be redistributed among its former neighbours, and their boundaries and cells will have been updated. The shortest path distances among the former neighbours can then be easily recomputed from the new boundary information.