Supplementary materials: Discovering Dense Correlated Subgraphs in Dynamic Networks

No Author Given

No Institute Given

1 Complete description of all algorithms

1.1 Creation of the Correlation Graph

The correlation graph $\mathcal{G}$ can be built exactly, by computing the correlation $c(e_1, e_2)$ between each pair of edges $e_1, e_2 \in E$ and retaining those pairs satisfying $c(e_1, e_2) \geq \sigma$. However, when D is large, comparing each pair of edges is prohibitively expensive, and thus we propose also an approximate solution based on *min-wise hashing* [1] which is described in Algorithm 1. Here we exploit the fact that that a strong correlation between two edges implies a high Jaccard similarity, and use min-wise hashing to identify sets of candidate correlated edges. Specifically, we use a variant of the TAPER algorithm [4], which repeats a min-wise hashing procedure r times, each time using h independent hash functions $h_i : T \to \mathbb{N}$. In each run, the algorithm computes h hash values for each edge and concatenates them to create a *hash code* for the edge (line 7). For each edge e and each hash function h_i, the hash value $H[e][i]$ is the minimum among the values $h_i(j)$, for timestamps $j \in T$ where e exists, i.e., $t_j(e) = 1$. For efficiency purposes, the $r\,h$ hash values are computed all together by traversing once the set of edges of D (lines 2–3). Edges with the same *hash code* are inserted into the same bucket (line 8) and the Pearson correlation is calculated for each pair of edges in the same bucket (line 12). If the correlation is greater than the correlation threshold σ, the pair is inserted in the edge set of the correlation graph $\mathcal{G}$ (line 13).

The algorithm requires two parameters that specify the number of runs r and the number of hash functions h. Larger r means less false negatives, and larger h means more effective pruning.

1.2 Enumeration of the Maximal Cliques

After the creation of the correlation graph $\mathcal{G}$, the maximal groups of correlated edges are enumerated by identifying the maximal cliques in $\mathcal{G}$. To this aim, we use our implementation of the *GP* algorithm of Wang et al. [3].

Algorithm 2 recursively partitions the graph into two disjoint parts and then examines each one independently using Procedure EnumCliques. In each step,

Algorithm 1 CreateCorrelationGraph (approximate)

Input: Dynamic network $D = (V, E)$
Input: Threshold: Correlation σ
Output: Correlation graph $\mathcal{G} = (E, \mathcal{E})$
1: $cand \leftarrow \emptyset$; $\mathcal{E} \leftarrow \emptyset$
2: **for each** $e \in E$; $i \in [0, h\,r]$ **do**
3: $\quad H[e][i] \leftarrow min\{h_i(j) \mid t_j(e) = 1\}$
4: **for each** $i \in [0, r]$ **do**
5: $\quad B[i] \leftarrow \emptyset$
6: $\quad$ **for each** $e \in E$ **do**
7: $\quad\quad code \leftarrow H[e][i\,h : (i+1)\,h - 1]$
8: $\quad\quad B[i][code] \leftarrow B[i][code] \cup \{e\}$
9: **for each** $i \in [0, r]$; $b \in B[i]$ **do**
10: $\quad$ **for each** $e_1, e_2 \in B[i][b]$ **do**
11: $\quad\quad cand \leftarrow cand \cup \{(e_1, e_2)\}$
12: **for each** $(e_1, e_2) \in cand$ **such that** $c(e_1, e_2) \geq \sigma$ **do**
13: $\quad \mathcal{E} \leftarrow \mathcal{E} \cup \{(e_1, e_2)\}$
14: **return** $\mathcal{G} \leftarrow$ CreateGraph$(E, \mathcal{E})$

it maintains three sets of vertices, *anchor*, *cand*, and *not*. The set *anchor*, initially empty, is recursively extended by adding a new vertex such that, at every step, the vertices in *anchor* are all connected in the input graph $\mathcal{G}$. The set *cand*, initially set to E, contains the vertices that can still be used to extend *anchor*, i.e., vertices that do not belong in *anchor* and are connected to every vertex in *anchor*. Finally, the set *not*, initially empty, contains vertices already used as extensions for *anchor* in the previous steps.

When examining the set *cand* in a graph, the algorithm first checks if the vertices in *cand* form a clique, hence returning the maximal clique $cand \cup anchor$. Otherwise, it recursively takes the vertex with smallest degree in *cand* (line 9), creates a new set *nxCand* of all the vertices in *cand* adjacent to v (line 11), and updates *cand* removing v (line 15). Set *not* is updated adding vertex v, as it cannot be used as a partitioning anchor anymore (line 16), and a new set *nxNot* is initialized adding all the vertices in *not* adjacent to v (line 12). If some vertex in *nxNot* is connected to all the vertices in *nxCand*, the recursion stops (line 13), since the vertices *nxCand* cannot generate a maximal clique. When *cand* becomes a clique, the recursion stops (line 17) and the algorithm checks if the clique is maximal before inserting it into the output set (line 19).

1.3 Discovery of the Dense Subgraphs

The goal of this step is to find connected groups of edges that form a dense subgraph, using either ρ_m^k or ρ_a^k as density function ρ^k. Algorithm 3 receives as input a set of maximal cliques $\mathcal{C}$, each of which represents a maximal group of correlated edges. Since some of the edges in a clique may not be connected in the network D, the algorithm extracts all the distinct connected components

Algorithm 2 FindMaximalCliques

Input: Graph $\mathcal{G} = (E, \mathcal{E})$
Output: Set of maximal cliques $\mathcal{C}$

```
1: anc ← ∅; not ← ∅; cand ← E
2: C ← EnumCliques(anc, cand, not)
3: return C

4: function EnumCliques(anc, cand, not)
5:     C ← ∅
6:     if isAClique(cand) then
7:         return {anc ∪ cand}
8:     repeat
9:         v ← vertex with smallest degree in cand
10:        nxAnc ← anc ∪ {v}
11:        nxCand ← cand ∩ adj[v]
12:        nxNot ← not ∩ adj[v]
13:        if ∄u ∈ nxNot s.t. ∀w ∈ nxCand, u ∈ adj[w] then
14:            C ← C ∪ EnumCliques(nxAnc, nxCand, nxNot)
15:        cand ← cand \ {v}
16:        not ← not ∪ {v}
17:    until isAClique(cand)
18:    if ∄u ∈ not s.t. ∀w ∈ cand, u ∈ adj[w] then
19:        C ← C ∪ {anc ∪ cand}
20:    return C
```

from the cliques (line 2), before computing the density values. To allow a faster discovery of the maximal groups of dense edges, the connected components are sorted in descending order of their size and processed iteratively. If no larger or similar dense set of the current candidate X has been discovered yet (line 5), and if the size of X does not exceed the threshold s_M, the density of X is computed by Algorithm 4 (line 6). We recall that the similarity between two sets is the Jaccard similarity, and two sets are dissimilar if the similarity is below the threshold ϵ. On the other hand, the threshold on the maximum size controls the size of the subgraphs in the result set $\mathcal{S}$, as well as the complexity of Algorithm 3.

Algorithm 4 describes the steps for determining if a set of edges X is dense or if at least contains some dense subset. It uses either Procedure isMinDense or Procedure isAvgDense in Algorithm 5 to compute the density of X and thus assessing if X is dense. Procedure isAvgDense computes ρ_a^k to determine if the average density of X is above the threshold δ. The average density of X is computed as the average among the average node degrees of the subgraphs H induced by X in all the snapshots where at least k edges of X are present.

On the other hand, Procedure isMinDense computes ρ_m^k, which expresses the density of a group of edges as the minimum between the average degrees of the induced subgraph in all the snapshots where at least k edges of the group are present, and then checks if its values is above δ. The minimum average degree is computed by iterating over the set K of snapshots at least k edges of

Algorithm 3 FINDDIVERSEDENSEEDGES

Input: Dynamic Network $D = (V, E)$
Input: Set of maximal cliques $\mathcal{C}$, Density function ρ^k
Input: Thresholds: Density δ, Size s_M
Input: Thresholds: Edges-per-snapshot k, Similarity ϵ
Output: Set of diverse dense maximal subgraphs $\mathcal{S}$

1: $\mathcal{S} \leftarrow \emptyset$; $\mathcal{P} \leftarrow \emptyset$
2: $CC \leftarrow$ EXTRACTCC($\mathcal{C}$)
3: **for each** $X \in CC$ **do**
4: **if** $X.size < s_M$ **and** ISMAXIMAL(X, $\mathcal{S} \cup \mathcal{P}$)
5: **and** ISDIVERSE(X, $\mathcal{S}$) **then**
6: $(flag, R) \leftarrow$ ISDENSE(D, X, k, ρ^k, δ)
7: **if** $flag = 1$ **then**
8: $\mathcal{S} \leftarrow \mathcal{S} \cup \{X\}$
9: **else if** $flag = 0$ **then**
10: $\mathcal{P} \leftarrow \mathcal{P} \cup R$
11: **for each** $X \in \mathcal{P}$ **do**
12: **if** ISMAXIMAL($X, \mathcal{S}$) **and** ISDIVERSE(X, $\mathcal{S}$) **then**
13: $\mathcal{S} \leftarrow \mathcal{S} \cup \{X\}$
14: **return** $\mathcal{S}$

15: **function** ISDIVERSE(X, S)
16: **if** $S = \emptyset$ **then return true**
17: **for each** $C_j \in S$ **do**
18: **if** $|C_j \cap X|/|C_j \cup X| > \epsilon$ **then return false**
19: **return true**

X present (line 9). Nonetheless, we can stop the iteration as soon as we find a snapshot t where the subgraph H induced by X in t has average degree below the density threshold δ (line 10), because the minimum average degree is now guaranteed to be lower than δ. Thanks to the optimizations described in the next paragraph, the implementation of Procedure ISAVGDENSE is more efficient than that of Procedure ISMINDENSE, and thus we call the latter only when the former returns **true**, given that the average is an upper bound on the minimum.

When the density $\rho_a^k(H)$ ($\rho_m^k(H)$ respectively) of the subgraph H induced by X is above the threshold δ, X is inserted in the result set $\mathcal{S}$ (Algorithm 3 line 8). When the density is below the threshold δ, the set X is not dense; though some subset $X' \subseteq X$ may satisfy the condition $\rho_a^k(H') \geq \delta$. Since examining all the possible subsets of X is a costly operation especially when the size of X is large, Algorithm 4 uses Procedure CONTAINSDENSE, which is based on a 2-approximation algorithm for the densest subgraph problem [2], to prune the search space. In details, Procedure CONTAINSDENSE iteratively removes the vertex with lowest degree from the induced subgraph H, until it becomes empty or its density is greater than $\delta/2$. Every time a vertex is removed, its outgoing edges are removed as well (line 11), and thus the set of valid snapshots K must be updated (line 12). If K becomes empty, any subset of X will have zero

Algorithm 4 isDense

Input: Dynamic Network $D = (V, E)$
Input: A set of edges X, Density function ρ^k
Input: Thresholds: Density δ, Edges-per-snapshot k
Output: $(1, \emptyset)$ if X is dense; $(0, R)$ if X contains the dense subsets R; $(-1, \emptyset)$ o.w.

```
1: K ← kEdgeSnapshots(X, k)
2: if ρ^k(X, K, δ) then return (1, ∅)
3: if containsDense(X, K, k) = ∅ then return (−1, ∅)
4: return (0, extractDense(X, K, k))

5: function containsDense(X, K, k)
6:     while ρ^k(X, K, δ/2) = false do
7:         if X = ∅ or K = ∅ then return false
8:         max ← getMaxDeg(X)
9:         n ← getMinDegNode(X)
10:        if max < δ/2 then return false
11:        X ← X \ adj(n)
12:        K ← kEdgeSnapshots(X, k)
13:    return true
14: function extractDense(X, K, k)
15:    R ← ∅; Q ← {(X, K)}
16:    while Q ≠ ∅ do
17:        (Y, K′) ← Q.pop
18:        if ρ^k(Y, K′, δ) then
19:            R ← R ∪ {Y}
20:        else if K′ ≠ ∅ then
21:            max ← getMaxDeg(Y)
22:            N ← getMinDegNodes(Y)
23:            if max < δ then
24:                continue
25:            for each n ∈ N do
26:                Y ← Y \ adj(n)
27:                K′ ← kEdgeSnapshots(Y, k)
28:                if Y ≠ ∅ then
29:                    Q ← Q ∪ {(Y, K′)}
30:    return R
```

density, and thus the algorithm returns **false** (line 7). If the maximum value of density calculated during the execution of this algorithm is below the threshold $\delta/2$, it holds that X cannot contain a subset X' with density above δ [2], and thus containsDense returns **false**. Therefore, Procedure extractDense, which extracts all the dense subsets in the set X, is invoked (line 4) only when Procedure containsDense returns **true**.

When Procedure containsDense returns **true**, Procedure extractDense iteratively searches for all the dense subsets in X. At each iteration, a subset of edges Y is extracted from the queue Q and its density is checked. If Y is not

Algorithm 5 Density Functions

Input: A set of edges X, A set of snapshots K
Input: Density threshold δ
Output: *true* if X is dense; *false* otherwise

```
1: function isAvgDense(X, K, δ)
2:     if K = ∅ then return false
3:     let H be the subgraph induced by X
4:     d ← 1/K Σ_{t∈K} ρ(G_t(H))
5:     return d ≥ δ

6: function isMinDense(X, K, δ)
7:     if K = ∅ then return false
8:     let H be the subgraph induced by X
9:     for each t ∈ K do
10:        if ρ(G_t(H)) < δ then return false
11:    return true
```

dense but the set of valid snapshots is not empty (line 20), a new candidate is created for each vertex n with lowest degree in the subgraph induced by Y. These candidates are then inserted into Q. On the other hand, when Y is dense, it is inserted into the result set R. At the end of Algorithm 3, the maximal subsets in the set $\mathcal{P}$, which contains the elements of all the R sets computed during the search, are checked for similarity with the subsets already in $\mathcal{S}$. Those with Jaccard similarity below ϵ with any subsets in $\mathcal{S}$ are finally added to $\mathcal{S}$ (lines 11–13).

2 Extended experimental evaluation

2.1 Effectiveness of the approximate solution

Our approximate algorithm trades accuracy for performance by approximating the set of σ-correlated edges. As described above, the approximate set is obtained by the min-wise hashing technique with parameters r and h. The parameter r indicates the number of repetitions, so larger values of r increase the quality of the result, at the cost of additional computation. For the number of hash functions h, larger numbers generate more informative hash codes, meaning that the algorithm will cluster the edges into smaller groups. As the number of comparisons decreases, the running time decreases as well.

We tested different combinations of (r, h), starting from $h{=}\ r = 3$ and increasing their value up to $h = r = 15$, and counted the number of σ-correlated edges discovered. Figure 1 shows that after a period of decrease, the number of correlated pairs reached a plateau at $h = 9$, thus prompting us to use combinations of small values for both r and h.

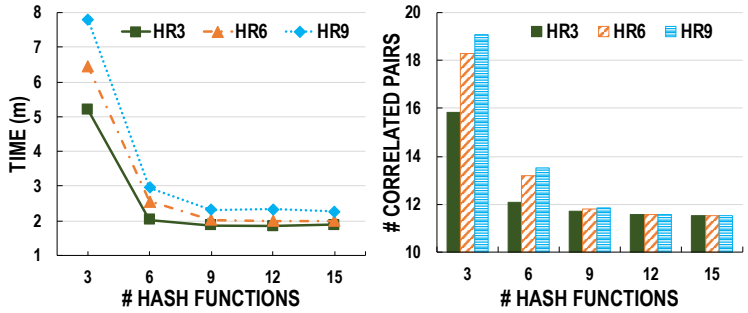

Fig. 1: Tuning of the min-wise hashing parameters r (HR) and h (HASH FUNCTIONS) running DCS *approximate* in TWITTER-L.

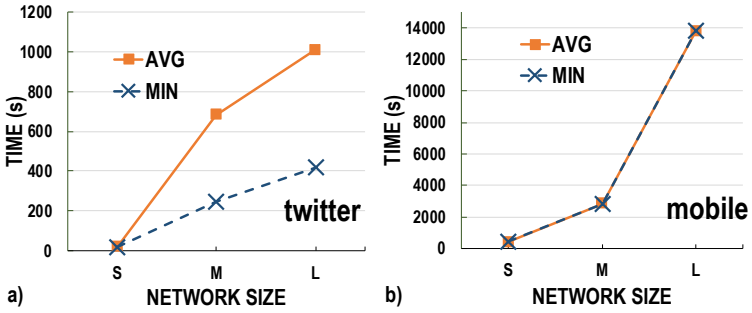

Fig. 2: Scalability of DCS in TWITTER with $\sigma = 0.7$ and $\delta = 2.6$ (a), and in MOBILE with $\sigma = 0.9$ and $\delta = 4.1$ (b), using both ρ_a^k and ρ_m^k.

2.2 Scalability

We tested the scalability of DCS using samples of increasing size, extracted from both the TWITTER and the MOBILE network. In the TWITTER-X samples we set $\sigma = 0.7$ and $\delta = 2.6$, while in the MOBILE-X samples we used $\sigma = 0.9$ and $\delta = 4.1$. Figure 2 shows that the running time increases exponentially for both ρ_a^k and ρ_m^k in the MOBILE-X samples (b), while it increases slightly slower for ρ_m^k in the TWITTER-X samples. This exponential growth is due to the **NP**-complete nature of the candidate generation task, which requires the enumeration of the maximal cliques in the correlation graph. Nonetheless, these cliques can be stored and used for all the experiments where σ is kept fixed, hence allowing a significant speed up in the performance when the user is interested in examining different combinations of the other parameters. We also note that the different behavior of ρ_m^k in the TWITTER-X samples is mainly due to the sparsity of the snapshots in TWITTER and the early pruning strategy in Algorithm 4 line 10 that discards a candidate group as soon as it finds a snapshot in which it is not dense. Finally, we mention that, while DCS terminated in under 12 minutes in the TWITTER-

M sample, CIFORAGER took 2.6 minutes per window (for a total of 89h using $w_l = 10$) to produce 12296 results.

References

1. Broder, A.Z., Charikar, M., Frieze, A.M., Mitzenmacher, M.: Min-wise independent permutations. JCSS **60**(3) (2000)
2. Charikar, M.: Greedy approximation algorithms for finding dense components in a graph. In: APPROX (2000)
3. Wang, Z., Chen, Q., Hou, B., Suo, B., Li, Z., Pan, W., Ives, Z.G.: Parallelizing maximal clique and k-plex enumeration over graph data. Journal of Parallel and Distributed Computing **106** (2017)
4. Zhang, J., Feigenbaum, J.: Finding highly correlated pairs efficiently with powerful pruning. In: CIKM (2006)