CoCooN: A Graph Compression Framework with Bounded Neighborhood Loss

Shubhadip Mitra
Manufacturing Research,
Tata Consultancy Services Ltd.,
Bengaluru, India
shubhadip.mitra@tcs.com

Sona Elza Simon
Centre for Machine Intelligence and Data Science,
Indian Institute of Technology Bombay
Mumbai, India
sonasimonp@gmail.com

C Oswald
Dept. of Computer Science and Engg.,
National Institute of Technology Tiruchirappalli
Tiruchirapalli, India
oswald@nitt.edu

Arnab Bhattacharya
Dept. of Computer Science and Engg.,
Indian Institute of Technology Kanpur
Kanpur, India
arnabb@cse.iitk.ac.in

Arindam Pal
Data61, Commonwealth Scientific and Industrial Research Organisation
Sydney, New South Wales, Australia
arindamp@gmail.com

ABSTRACT

Graph compression is a method to produce a compressed representation of a graph that allows reconstruction of the original graph with minimal or no loss. Despite the existence of several graph compression techniques, there does not exist a general purpose unified compression framework that offers high compression along with *all* the following desirable capabilities: (1) it allows the user to choose between lossless and lossy compression, (2) it allows the user to control the amount of neighborhood loss that it can tolerate, (3) it allows processing of wide range of graph queries on the compressed graph (without decompressing it), (4) it allows the user to specify the type of loss it can tolerate, i.e., whether it can tolerate *false positive edges* (i.e., extra edges), *false negative edges* (i.e., missing edges), or neither, in the decompressed graph or query answers. To overcome these limitations, we propose a novel graph compression framework CoCooN that builds upon the idea of *common neighborhoods*. The proposed framework consists of three compression variants, CCN-E, CCN-I and CCN-U. While CCN-E is a lossless compression scheme, CCN-I and CCN-U are lossy compression schemes that allow reconstruction of the input graph with *no false positive* edges and *no false negative* edges, respectively. To bound the graph reconstruction loss, we introduce a user-specified parameter *neighborhood loss tolerance threshold*, that bounds the maximum loss allowed in the neighborhood of any given node. This allows reconstruction of the input graph and evaluation of neighborhood queries with *bounded quality guarantees*. Empirical evaluation on 9 real-world graphs shows that CoCooN offers superior compression than most of its competitors. Further, they also confirm that CoCooN can answer a wide range of graph queries with fairly high accuracy and efficiency.

A THE COCOON ALGORITHM

This section presents the remaining pseudocodes of the CoCooN algorithm that could not be accommodated in the main body of the paper due to lack of space. The DiscoverCN procedure (Alg. 5) discovers the pairs of nodes that are eligible to be merged. The ComputeCG procedure (Alg. 6) computes the compression gain of any such pair of nodes. The IsSafeMerge procedure (Alg. 7 validates whether merging a given pair (u, v) is safe, i.e., it would not violate the neighborhood loss constraint (Eq. 5). Once a given pair is merged, the UpdateDegree procedure (Alg. 8) updates the degree of each node $u \in G_d$, given by $\deg_{G_d}(u)$. The UpdateCG procedure (Alg. 9) is responsible for updating the heap H after a given pair of nodes is merged. Finally, the Anc procedure (Alg 10) computes the ancestors of a node $u \in G_c$.

Algorithm 5: DiscoverCN

```
1  Function DiscoverCN (G, var)
2      Create an empty binary max-heap H.
3      for each pair (u, v) such that u ∈ G and v ∈ N_G^2(u) do
4          cg ← ComputeCG(G, var, u, v)
5          if cg > 0 then
6              Insert-Key (H, u, v, cg) // inserts a node into the heap
                 H with key = cg and value = (u, v).
7
8      return H
```

Algorithm 6: ComputeCG

```
1  Function ComputeCG (G_c, var, u, v)
2      if var = I then
3          cg ← 2|N_Gc(u) ∪ N_Gc(v)| − 3
4          if edge (u, v) ∈ G_c then
5              cg ← cg − 2
6      else if var = E ∨ var = U then
7          cg ← 2|N_Gc(u) ∩ N_Gc(v)| − 3
8      for each w ∈ {u, v} do
9          if |P_Gc(w)| = E then
10             if var = I then
11                 cg ← cg + 1
12             else if var = U ∧ edge (u, v) ∉ G_c then
13                 cg ← cg + 1
14             else if var = E then
15                 w' ← {u, v} − {w}
16                 W ← N_Gc(w) − N_Gc(w')
17                 if |W| = E then
18                     cg ← cg + 1
19     return cg
```

The $NeighborhoodQuery$ algorithm returns the neighborhood set $N_{G_d}(u)$, i.e., the neighborhood set of u in the decompressed graph

Algorithm 7: IsSafeMerge

```
1  Function IsSafeMerge (G_c, var, u*, v*, Δ)
2      if var = E then
3          return true
4      U* ← Anc(G_c, u*)
5      V* ← Anc(G_c, v*)
6      if var = I then
7          U' ← {u' ∈ Anc(G_c, u'') | u'' ∈ (N_{G_c}(u*) − N_{G_c}(v*))}
8          V' ← {v' ∈ Anc(G_c, v'') | v'' ∈ (N_{G_c}(v*) − N_{G_c}(u*))}
9          for each u ∈ U* do
10             if (deg_{G_d}(u) − |U'|) / deg_G(u) < 1 − δ_u then
11                 return false
12         for each v ∈ V* do
13             if (deg_{G_d}(v) − |V'|) / deg_G(v) < 1 − δ_v then
14                 return false
15         for each u ∈ U' − V* do
16             if (deg_{G_d}(u) − |U*|) / deg_G(u) < 1 − δ_u then
17                 return false
18         for each v ∈ V' − U* do
19             if (deg_{G_d}(v) − |V*|) / deg_G(v) < 1 − δ_v then
20                 return false
21         return true
22     else if var = U then
23         U' ← {u' ∈ Anc(G_c, u'') | u'' ∈ (N_{G_c}(u*) − {v*} − N_{G_c}(v*))}
24         V' ← {v' ∈ Anc(G_c, v'') | v'' ∈ (N_{G_c}(v*) − {u*} − N_{G_c}(u*))}
25         for each u ∈ U* do
26             if (deg_{G_d}(u) + |V'|) / deg_G(u) > 1 + δ_u then
27                 return false
28         for each v ∈ V* do
29             if (deg_{G_d}(v) + |U'|) / deg_G(v) > 1 + δ_v then
30                 return false
31         for each u ∈ U' do
32             if (deg_{G_d}(u) + |V*|) / deg_G(u) > 1 + δ_u then
33                 return false
34         for each v ∈ V' do
35             if (deg_{G_d}(v) + |U*|) / deg_G(v) > 1 + δ_v then
36                 return false
37         return true
```

Algorithm 8: UpdateDegree

```
1  Function UpdateDegree (G_c, var, u*, v*)
2      U* ← Anc(G_c, u*)
3      V* ← Anc(G_c, v*)
4      if var = I then
5          U' ← {u' ∈ Anc(G_c, u'') | u'' ∈ (N_{G_c}(u*) − N_{G_c}(v*))}
6          V' ← {v' ∈ Anc(G_c, v'') | v'' ∈ (N_{G_c}(v*) − N_{G_c}(u*))}
7          for each u ∈ U* do
8              deg_{G_d}(u) ← deg_{G_d}(u) − |U'|
9          for each v ∈ V* do
10             deg_{G_d}(v) ← deg_{G_d}(v) − |V'|
11         for each u ∈ U' − V* do
12             deg_{G_d}(u) ← deg_{G_d}(u) − |U*|
13         for each v ∈ V' − U* do
14             deg_{G_d}(v) ← deg_{G_d}(v) − |V*|
15     else if var = U then
16         U' ← {u' ∈ Anc(G_c, u'') | u'' ∈ (N_{G_c}(u*) − {v*} − N_{G_c}(v*))}
17         V' ← {v' ∈ Anc(G_c, v'') | v'' ∈ (N_{G_c}(v*) − {u*} − N_{G_c}(u*))}
18         for each u ∈ U* do
19             deg_{G_d}(u) ← deg_{G_d}(u) + |V'|
20         for each v ∈ V* do
21             deg_{G_d}(v) ← deg_{G_d}(v) + |U'|
22         for each u ∈ U' do
23             deg_{G_d}(u) ← deg_{G_d}(u) + |V*|
24         for each v ∈ V' do
25             deg_{G_d}(v) ← deg_{G_d}(v) + |U*|
```

Algorithm 9: UpdateCG

```
1  Function UpdateCG (G_c, var, H, s, u*, v*)
2      for each pair (s, u) such that u ∈ N²_{G_c}(s) do
3          cg ← ComputeCG (G_c, var, s, u)
4          if cg > 0 then
5              Insert-Key (H, s, u, cg)
6      if var = I ∨ var = U then
7          for each pair (u, v*) ∈ H do
8              Decrease-Key (H, u, v*, ∞)
9          for each pair (u*, v) ∈ H do
10             Decrease-Key (H, u*, v, ∞)
11     if var = I then
12         for each pair (u, v) ∈ H, such that either u or v or both lie in
             N_{G_c}(u*) ∪ N_{G_c}(v*) do
13             Decrease-Key (H, u, v, 2)
14     else if var = U then
15         for each pair (u, v) ∈ H, such that u ∈ (N_{G_c}(u*) − N_{G_c}(v*)),
             v ∈ (N_{G_c}(v*) − N_{G_c}(u*)) do
16             Increase-Key (H, u, v, 2)
17         for each pair (u, v) ∈ H, such that both u, v ∈ (N_{G_c}(u*) ∩ N_{G_c}(v*)) do
18             Decrease-Key (H, u, v, 2)
19     else if var = E then
20         for each pair (u, v*) ∈ H do
21             value ← 2|N_{G_c}(u*) ∩ N_{G_c}(v*) ∩ N_{G_c}(u)|
22             Decrease-Key (H, u, v*, value)
23         for each pair (u*, v) ∈ H do
24             value ← 2|N_{G_c}(u*) ∩ N_{G_c}(v*) ∩ N_{G_c}(v)|
25             Decrease-Key (H, u*, v, value)
26         for each pair (u, v) such that either u or v or both lie in
             N_{G_c}(u*) ∩ N_{G_c}(v*) do
27             Decrease-Key (H, u, v, 2)
```

G_d. Initially, it sets $N_{G_d}(u)$ to the empty set. Then it makes a call to the Desc procedure that reports all the *descendents* of u. A node s is said to be a *descendent* of u, if u is an ancestor of s. The Desc procedure (Algorithm 12) with input parameters (G_c, u) reports all the descendents of u in G_c. To realize this, it uses the *Child* sets. Finally, the $NeighborhoodQuery$ algorithm returns the neighborhood set $N_{G_d}(u) = \{Anc(G_c, w) | w \in N_{G_c}(v),\ v \in Desc(G_c, u)\}$.

B CORRECTNESS OF COCOON

The following result helps to establish the correctness of the CoCooN algorithm.

THEOREM 8. *Given an input graph $G = (V, E)$ such that $|E| > |V|(|V| - 1)/4$. Suppose G is compressed by CoCooN to produce the compressed graph $G_c = (V_c, E_c)$ that is later decompressed to produce the decompressed graph $G_d = (V_d, E_d)$. Then, $V_d = V$. Also,* CCN-U, CCN-E *and* CCN-I *respectively guarantee that $E_d \supseteq E$, $E_d = E$ and $E_d \subseteq E$. Additionally, G_d respects the neighborhood loss constraint stated in Eq. 5.*

PROOF. If $|E| > |V|(|V| - 1)/4$, the CoCooN algorithm invokes the GreedyCN procedure with parameters: $(\overline{G}, \overline{var}, \overline{\Delta})$ where $\overline{G}$ is the complementary graph of G and $\overline{\Delta} = \{\overline{\delta_u} | u \in G\}$ and $\overline{\delta_u}$ is as stated in Eq. 9. Let G_c be the compressed graph returned by the GreedyCN procedure that when decompressed by the Decompression procedure returns the decompressed graph G_d. Since in this case, $G_c.complement = true$, therefore, the Decompression procedure generates a graph $\overline{G_d}$ (at the end of line 7) before returning its complementary graph G_d in line 11.

Suppose $var = I$. This implies the GreedyCN procedure is invoked with parameter $var = U$ and for each node $u \in G$, $\overline{\delta_u} = \frac{\delta_u \deg_G(u)}{n - \deg_G(u)}$. From Th. 1, it follows that for each node $u \in \overline{G}$, $N_{\overline{G}}(u) \subseteq$

Algorithm 10: Anc

```
Function Anc (G_c, s)
    Nodes ← ∅
    Let Stack be an empty stack.
    Stack.push(s)
    while Stack is non-empty do
        u = Stack.pop()
        if u ∉ G_c ∨ P_{G_c}(u) = ∅ then
                                        // u is a simple node.
            Nodes ← Nodes ∪ {u}
        else
                                        // u is a super-node.
            for each v ∈ P_{G_c}(u) do
                Stack.push(v)
    return Nodes
```

Algorithm 11: *NeighborhoodQuery*

```
Function NeighborhoodQuery (G_c, u)
    N_{G_d}(u) ← ∅
    Desc ← Desc(G_c, u)
    for each node v ∈ Desc do
        for each node w ∈ N_{G_c}(v) do
            N_{G_d}(u) ← N_{G_d}(u) ∪ Anc(G_c, w)
    return N_{G_d}(u)
```

Algorithm 12: Desc

```
Function Desc (G_c, u)
    Nodes ← ∅
    Let Stack be an empty stack.
    Stack.push(u)
    while Stack is non-empty do
        v = Stack.pop()
        for each w ∈ Child_{G_c}(v) do
            Nodes ← Nodes ∪ {w}
            Stack.push(w)
    return Nodes
```

$N_{\overline{G_d}}(u)$. Further, from Eq. 5, it follows that

$$\frac{\deg_{\overline{G_d}}(u)}{\deg_{\overline{G}}(u)} \leq 1 + \overline{\delta_u} \leq 1 + \frac{\delta_u \deg_G(u)}{n - \deg_G(u)}$$

Given that for each node $u \in G$, $N_{\overline{G}}(u) = V - N_G(u)$ and $N_{\overline{G_d}}(u) = V - N_{G_d}(u)$, hence, $\deg_{\overline{G}}(u) = n - \deg_G(u)$ and $\deg_{\overline{G_d}}(u) = n - \deg_{G_d}(u)$. This implies

$$\begin{aligned} \frac{n - \deg_{G_d}(u)}{n - \deg_G(u)} &\leq 1 + \frac{\delta_u \deg_G(u)}{n - \deg_G(u)} \\ n - \deg_{G_d}(u) &\leq n - (1 - \delta_u)\deg_G(u) \\ \deg_{G_d}(u) &\geq (1 - \delta_u)\deg_G(u) \end{aligned}$$

Further, since $N_{\overline{G}}(u) \subseteq N_{\overline{G_d}}(u)$, it follows that $N_{G_d}(u) \subseteq N_G(u)$. Therefore, the neighborhood loss constraint is satisfiedfor $var = I$.

Suppose $var = U$. This implies the GreedyCN procedure is invoked with parameter $var = I$ and for each node $u \in G$, $\overline{\delta_u} = \frac{\delta_u \deg_G(u)}{n - \deg_G(u)}$. From Th. 1,, it follows that for each node $u \in \overline{G}$, $N_{\overline{G_d}}(u) \subseteq N_{\overline{G}}(u)$. Further, from Eq. 5, it follows that

$$\begin{aligned} \frac{\deg_{\overline{G_d}}(u)}{\deg_{\overline{G}}(u)} &\geq 1 - \overline{\delta_u} \geq 1 - \frac{\delta_u \deg_G(u)}{n - \deg_G(u)} \\ \frac{n - \deg_{G_d}(u)}{n - \deg_G(u)} &\geq \frac{n - (1 + \delta_u)\deg_G(u)}{n - \deg_G(u)} \\ n - \deg_{G_d}(u) &\geq n - (1 + \delta_u)\deg_G(u) \\ \deg_{G_d}(u) &\leq (1 + \delta_u)\deg_G(u) \end{aligned}$$

Since $N_{\overline{G_d}}(u) \subseteq N_{\overline{G}}(u)$, it follows that $N_G(u) \subseteq N_{G_d}(u)$. Therefore, the neighborhood loss constraint is satisfiedfor $var = U$. □ □

C GRAPH PROPERTIES THAT GUARANTEE COMPRESSION BY THE COCOON FRAMEWORK

The following results analyze the necessary and sufficient conditions to guarantee that $|G_c| < |G|$, i.e., $cr < 1$, for each of the three algorithms, namely, CCN-E, CCN-I and CCN-U.

THEOREM 9. *Let G_c be the compressed graph of G as produced by the* CCN-E *algorithm. Then, $|G_c| < |G|$ if and only if there exists at least a pair of nodes $u, v \in G$ such that either $|N_G(u) \cap N_G(v)| \geq 2$ or $|N_G(u) \cap N_G(v)| = 1 \ \wedge N_G(u) = N_G(v)$.*

PROOF. Suppose the graph G has a pair of nodes $u, v \in G$ such that $|N_G(u) \cap N_G(v)| \geq 2$. It is sufficient to show that the CCN-E algorithm executes at least one MergeNodes operation, and eventually produces a graph G_C such that $|G_c| < |G|$. For this to happen, it is sufficient to show that the heap H has at least one pair of nodes that offer positive compression gain. If such a pair exists, the CCN-E algorithm would eventually extract and merge such a pair. We show that the pair (u, v) satisfies the above requirement.

From the ComputeCG algorithm (Alg. 6), we observe that for the given pair (u, v), $cg(u, v) \geq 2|N_G(u) \cap N_G(v)| - 3 + x$ where x is given as follows:

$$x = \begin{cases} 0 & \text{if } |N_G(u) - N_G(v)| \geq 1 \wedge |N_G(v) - N_G(u)| \geq 1 \\ 1 & \text{if either } N_G(u) \subset N_G(v) \text{ or } N_G(v) \subset N_G(u) \\ 2 & \text{if } N_G(u) = N_G(v) \end{cases} \tag{11}$$

If $|N_G(u) \cap N_G(v)| \geq 2$ then $cg(u, v) \geq 1$. If $|N_G(u) \cap N_G(v)| = 1$ and $N_G(u) = N_G(v)$, then $x = 2$. Consequently, $cg(u, v) = 1$. Thus, in either of the above cases, $cg(u, v) > 0$. It follows that the CCN-E algorithm would have executed at least one merge operation, thereby producing a graph G_c such that $|G_c| < |G|$.

Next consider the case $|G_c| < |G|$. This is only possible if the graph G undergoes at least one iteration of MergeNodes. This can only happen if there exist at least a pair of nodes $u, v \in G$ such that $cg(u, v) > 0$. From the ComputeCG algorithm (Alg. 6), we know that $cg(u, v) \geq 2|N_G(u) \cap N_G(v)| - 3 + x$ where x is given in Eq. 11. Thus, if $cg(u, v) > 0$, it implies that G has a pair of nodes $u, v \in G$ such that $|N_G(u) \cap N_G(v)| \geq 2$ or $|N_G(u) \cap N_G(v)| = 1 \ \wedge N_G(u) = N_G(v)$.

□ □

THEOREM 10. *Let G_c be the compressed graph of G as produced by the* CCN-I *algorithm. Then, $|G_c| < |G|$ if and only if there exists at least a pair of nodes $u, v \in G$ such that $v \in N_G^2(u)$,*

$|N_G(u)| \leq |N_G(u) \cap N_G(v)|/(1-\delta_u)$ *and* $|N_G(v)| \leq |N_G(u) \cap N_G(v)|/(1-\delta_v)$.

PROOF. Suppose the graph G has a pair of nodes $u, v \in G$ such that $v \in N_G^2(u)$, $|N_G(u) \cup N_G(v)| \geq 2$, $|N_G(u)| \leq |N_G(u) \cap N_G(v)|/(1-\delta_u)$ and $|N_G(v)| \leq |N_G(u) \cap N_G(v)|/(1-\delta_v)$. It is sufficient to show that the CCN-I algorithm executes at least one MergeNodes operation, and eventually produces a graph G_C such that $|G_c| < |G|$. For this to happen, it is sufficient to show that the heap H has at least one pair of nodes that offer positive compression gain and their merging does not violate the neighborhood loss constraint. If such a pair exists, the CCN-I algorithm would eventually extract and merge such a pair. We show that the pair (u, v) satisfies the above requirement.

Firstly, from the ComputeCG algorithm (Alg. 6), we observe that for a given pair $u, v \in G$, $cg(u, v) \geq 2|N_G(u) \cup N_G(v)| - 3 + y$ where y is given as follows:

$$y = \begin{cases} 0 & \text{if } (u, v) \in E \\ 2 & \text{otherwise} \end{cases} \tag{12}$$

Since $v \in N_G^2(u)$, therefore, it follows that $|N_G(u) \cup N_G(v)| \geq 1$. Moreover, if $(u, v) \in E$, then, $|N_G(u) \cup N_G(v)| \geq 2$. Hence, for either of the cases: $(u, v) \in E$ or $(u, v) \notin E$, $cg(u, v) > 0$.

Next, we note that $rl(u) = |N_G(u) - N_{G_d}(u)|/|N_G(u)| = (|N_G(u) - (N_G(u) \cap N_G(v))|)/|N_G(u)|$. Since $(N_G(u) \cap N_G(v)) \subseteq N_G(u)$, therefore, the above expression simplifies to $rl(u) = 1 - |N_G(u) \cap N_G(v)|/|N_G(u)|$. From the condition stated above, $|N_G(u) \cap N_G(v)| \geq (1-\delta_u)|N_G(u)|$. Hence, $rl(u) \leq \delta_u$. In a similar manner, it can be shown that $rl(v) \leq \delta_v$. Therefore, if the pair of nodes (u, v) is merged, it does not violate the the neighborhood loss constraint. It follows that the CCN-I algorithm would have executed at least one merge operation, thereby producing a graph G_c such that $|G_c| < |G|$.

Next consider the case $|G_c| < |G|$. This is only possible if the graph G undergoes at least one iteration of MergeNodes. This can only happen if there exists at least a pair of nodes $u, v \in G$ such that $cg(u, v) > 0$ and their merging does not violate the neighborhood loss constraint. From the ComputeCG algorithm (Alg. 6), we know that $cg(u, v) \geq 2|N_G(u) \cup N_G(v)| - 3 + y$ where y is given by Eq. 12. Thus, if $cg(u, v) > 0$, then it implies that $v \in N_G^2(u)$. Further, if the merging of the pair (u, v) does not violate the neighborhood loss constraint, then, $rl(u) \leq \delta_u$ and $rl(v) \leq \delta_v$. It follows that the graph G has a pair of nodes $u, v \in G$ such that $v \in N_G^2(u)$, $|N_G(u)| \leq |N_G(u) \cap N_G(v)|/(1-\delta_u)$ and $|N_G(v)| \leq |N_G(u) \cap N_G(v)|/(1-\delta_v)$.

□ □

THEOREM 11. *Let G_c be the compressed graph of G as produced by the* CCN-U *algorithm. Then, $|G_c| < |G|$ if and only if there exists at least a pair of nodes $u, v \in G$ such that either $|N_G(u) \cap N_G(v)| \geq 2$ or $|N_G(u) \cap N_G(v)| = 1 \wedge (u, v) \notin E$; $|N_G(u)| \geq |N_G(u) \cup N_G(v)|/(1+\delta_u)$ and $|N_G(v)| \geq |N_G(u) \cup N_G(v)|/(1+\delta_v)$.*

PROOF. Suppose the graph G has a pair of nodes $u, v \in G$ such that $|N_G(u) \cap N_G(v)| \geq 2$, $|N_G(u)| \geq |N_G(u) \cup N_G(v)|/(1+\delta_u)$ and $|N_G(v)| \geq |N_G(u) \cup N_G(v)|/(1+\delta_v)$. It is sufficient to show that the CCN-U algorithm executes at least one MergeNodes operation, and eventually produces a graph G_C such that $|G_c| < |G|$. For this to happen, it is sufficient to show that the heap H has at least one pair of nodes that offer positive compression gain and their merging does not violate the neighborhood loss constraint. If such a pair exists, the CCN-U algorithm would eventually extract and merge such a pair. We show that the pair (u, v) satisfies the above requirement.

Firstly, from the ComputeCG algorithm (Alg. 6), we observe that for a given pair (u, v), $cg(u, v) \geq 2|N_G(u) \cap N_G(v)| - 3 + z$ where z is given as follows:

$$z = \begin{cases} 0 & \text{if } (u, v) \in E \\ 2 & \text{otherwise} \end{cases} \tag{13}$$

Since $v \in N_G^2(u)$, therefore, $|N_G(u) \cap N_G(v)| \geq 1$. Note that if $(u, v) \notin E$, then $z = 2$. Consequently, $cg(u, v) > 0$. Else, if $|N_G(u) \cap N_G(v)| \geq 2$, then $cg(u, v) > 0$.

Next, we note that $rl(u) = |N_{G_d}(u) - N_G(u)|/|N_G(u)| = (|(N_G(u) \cup N_G(v) - N_G(u))|)/|N_G(u)|$. Since $N_G(u) \subseteq (N_G(u) \cup N_G(v))$, therefore, the above expression simplifies to $rl(u) = |N_G(u) \cup N_G(v)|/|N_G(u)| - 1$. From the condition stated above, $|N_G(u) \cup N_G(v)| \leq (1+\delta_u)|N_G(u)|$. Hence, $rl(u) \leq \delta_u$. In a similar manner, it can be shown that $rl(v) \leq \delta_v$. Therefore, if the pair of nodes (u, v) is merged, it does not violate the the neighborhood loss constraint. It follows that the CCN-U algorithm would have executed at least one merge operation, thereby producing a graph G_c such that $|G_c| < |G|$.

Next consider the case $|G_c| < |G|$. This is only possible if the graph G undergoes at least one iteration of MergeNodes. This can only happen if there exist at least a pair of nodes $u, v \in G$ such that $cg(u, v) > 0$ and their merging does not violate the neighborhood loss constraint. From the ComputeCG algorithm (Alg. 6), we know that $cg(u, v) \geq 2|N_G(u) \cap N_G(v)| - 3 + z$ where z is given in Eq. 13. Thus, if $cg(u, v) > 0$, then it implies that either $(u, v) \notin E$ or $|N_G(u) \cap N_G(v)| \geq 2$. Further, if the merging of the pair (u, v) does not violate the neighborhood loss constraint, then, $rl(u) \leq \delta_u$ and $rl(v) \leq \delta_v$. It follows that the graph G has a pair of nodes $u, v \in G$ such that either $|N_G(u) \cap N_G(v)| \geq 2$ or $|N_G(u) \cap N_G(v)| = 1 \wedge (u, v) \notin E$; $|N_G(u)| \geq |N_G(u) \cup N_G(v)|/(1+\delta_u)$ and $|N_G(v)| \geq |N_G(u) \cup N_G(v)|/(1+\delta_v)$.

□ □

D COMPLEXITY OF COCOON

Next, we analyze the time complexity of the CoCooN algorithm. Suppose the maximum degree of any node $u \in G$ is d, i.e., $|N_G(u)| \leq d$. Assume the neighborhood sets $N_G(u), N_{G_c}(u)$ are kept sorted for each node u. Further, let $|V| = n$, and $|E| = m$. Let $\delta = \max\{\delta_u | u \in G\}$. The complexity of the CoCooN algorithm is established through the following series of results.

LEMMA 1. *Given a graph G of n nodes, the number of pairs of nodes that are eligible to be merged is $O(\min(md, \binom{n}{2}))$.*

PROOF. A pair of nodes $u, v \in G$ is eligible to be merged if $v \in N_G^2(u)$. The number of 2-hop neighbors of $u \in G$, is $|N_G^2(u)| = \sum_{v \in N_G(u)} (\deg_G(v) - 1)$. Thus, the total number of pairs of nodes that are eligible to be merged is at most $\sum_{u \in G} |N_G^2(u)| \leq \sum_{u \in G} \sum_{v \in N_G(u)} (\deg_G(v) - 1) \leq \sum_{u \in G} \deg_G(u)(\deg_G(u) - 1) \leq 2md$. Given that the graph G has n nodes, the total number of possible pairs of nodes is $\binom{n}{2}$. Therefore, the number of eligible pairs is $O(\min(md, \binom{n}{2}))$. □ □

LEMMA 2. *The DiscoverCN procedure creates a binary max-heap H, whose size, i.e., the number of nodes in H, is $|H| = O(md)$.*

PROOF. From Lem. 1, the number of pairs eligible to be merged is $O(md)$. Note that the DiscoverCN procedure inserts one node h into the heap H for each such pair (u, v) whose $cg(u, v) > 0$. Thus, the total number of nodes inserted by the DiscoverCN procedure is $|H| = O(md)$. □ □

LEMMA 3. *The DiscoverCN procedure runs in $O(md^2)$ time.*

PROOF. Firstly, we observe that the ComputeCG procedure runs in $O(d)$ time. This is because each neighborhood set of size at most d is kept sorted, and the intersection and union of such sorted sets can be computed in linear time. From Lem. 1, the number of pairs eligible to be merged is $O(md)$. Since ComputeCG is called for each such pair, the DiscoverCN procedure requires $O(md^2)$ time. □ □

LEMMA 4. *Let S be the set of super-nodes in the compressed graph G_c. Then,*

$$|S| \leq \begin{cases} m & \text{if var} = E \\ n-1 & \text{if var} \in \{I, U\} \end{cases} \tag{14}$$

PROOF. Consider the case $var = E$. A given pair (u, v) is merged to form exactly one super-node if $cg(u, v) > 0$. For this to happen, referring to Eq. 7, it follows that the number of edges in G_c must reduce by at least 1 after the merging of (u, v). Given that G_c is initialized to G which has m edges, the number of super-nodes that are added into G_c is at most $m - |E_c| \leq m$.

Now suppose $var \in \{I, U\}$. In this scenario, each super-node has exactly 2 parents, and mor importantly, each node has at most one child. In view of this fact, the evolution of super-nodes during the course of the CoCooN algorithm, can be modeled as a binary tree where the nodes in the input graph G form the leaf nodes, and the super-nodes are represented by the internal nodes. Since a binary tree with n leaf nodes has at most $n - 1$ internal nodes, therefore, the number of super-nodes produced by the CoCooN algorithm is at most $n - 1$. □ □

LEMMA 5. *At any stage of the CoCooN algorithm, the size of the 1-hop neighborhood of any node $u \in G_c$ is bounded as follows:*

$$|N_{G_c}(u)| \leq \begin{cases} d & \text{if var} \in \{I, E\} \\ d(1+\delta) & \text{if var} = U \end{cases} \tag{15}$$

PROOF. At the begining of the CoCooN algorithm, the GreedyCN procedure is invoked that initializes the graph G_c to G. At this stage, the neighborhood of any node $u \in G_c$ ia bounded as follows: $|N_{G_c}(u)| \leq |N_G(u)| \leq d$. Next consider any subsequent stage of the algorithm.

Consider any node $u \in G_c$ which could either be a simple node or a super-node. Firstly, suppose u is a simple node. Referring to the MergeNodes procedure (Algorithm 3), one finds that during the course of the algorithm, the size of the neighborhood set $N_{G_c}(u)$ either shrinks or remains unchanged. If a new neighbor is added, simultaneously, one or more existing neighbors are removed. Thus, for any such node, $|N_{G_c}(u)| \leq d$. Next, suppose u is a super-node and v is one of its ancestors, i.e., $v \in Anc(G_c, u)$.

Consider the case $var \in \{I, E\}$. From the MergeNodes procedure, it is clear that $N_{G_c}(u) \subseteq N_G(v)$. From this it follows that $|N_{G_c}(u)| \leq |N_G(v)| \leq d$. Further, during the course of the algorithm, the size of the neighborhood $|N_{G_c}(u)|$ never increases. Thus, at each stage of the algorithm, $|N_{G_c}(u)| \leq d$.

Next, consider $var = U$. Suppose $|N_{G_c}(u)| > d(1+\delta)$. At this stage, suppose G_c is decompressed. From the Decompression procedure, it follows that $|N_{G_d}(v)| \geq |N_{G_c}(u)| > d(1+\delta)$. This violates the neighborhood loss constraint stated in Eq. 5. This contradicts the correctness of the CoCooN algorithm, established in Th. 3. □ □

LEMMA 6. *At any stage of the CoCooN algorithm, the maximum size of the heap H is given as follows:*

$$|H| = \begin{cases} O(md + nd^2) & \text{if var} \in \{I, U\} \\ O(md^2) & \text{if var} = E \end{cases} \tag{16}$$

PROOF. From Lem. 2, it follows that at the end of the DiscoverCN procedure, the size of the heap H is $O(md)$. Subsequently, suppose the algorithm produces k super-nodes. From Lem. 4, it follows that if $var \in \{I, U\}$, $k = O(n)$; and if $var = E$, then $k = O(m)$. For each such super-node s, and for each $u \in N^2_{G_c}(s)$, the UpdateCG procedure inserts a node into the heap H. From Lem. 5, it follows that $|N^2_{G_c}(s)| = O(d^2)$. Combining these facts, lead to the above result. □ □

LEMMA 7. *The number of iterations of the while loop stated in line 7 of the GreedyCN procedure is given as follow:*

$$O(|H|) \text{ for var} \in \{I, U\}$$
$$O(m) \text{ for var} = E$$

where $|H|$ is given in Eq. 16.

PROOF. Consider the case $var \in \{I, U\}$. The iterative phase of the GreedyCN procedure (starting from line 7) continues as long as there exists a pair $(u, v) \in H$ such that $cg(u, v) > 0$. Therefore, the number of iterations of the while loop (stated in line 7) of the GreedyCN procedure is at most the maximum size of the heap H that is stated in Eq. 16.

Next, consider the case $var = E$. In each iteration, the GreedyCN procedure extracts a pair of nodes and merges them. Since each merge operation produces exactly one super-node, and the number of super-nodes is at most m (Lem. 4), hence the number of iterations is at most $O(m)$. □ □

LEMMA 8. *The IsSafeMerge procedure and the UpdateDegree procedure run in $O(n)$ time.*

PROOF. Firstly, we show that the Anc procedure runs in $O(n)$ time. Referring to the Anc procedure, one finds that if s is a super-node that has k ancestors, then $Anc(G_c, s$ takes $O(k)$ time. Since $k \leq n$, hence, the Anc procedure runs in $O(n)$ time.

Next, referring to the IsSafeMerge procedure and the UpdateDegree procedure, it is clear that the size of each of the sets U^*, V^*, U' and V' is at most n and their computation takes $O(n)$ time. The remaining steps of these procedures run in $O(n)$ time. This leads to the above result. □ □

LEMMA 9. *The UpdateCG procedure runs in $O(d^2(d + \log n))$ time.*

PROOF. The UpdateCG procedure begins by considering all pairs of nodes $s, u \in G_c$, such that (s, u) can be merged. From Lem. 5, it follows that $|N^2_{G_c}(s)| = O(d^2)$. Thus, the number of pairs (s, u) is $O(d^2)$. For each such pair (s, u), the ComputeCG procedure is called that takes $O(d)$. The total time spent in computation of $cg(s, u)$ for each pair (s, u) is thus, $O(d^3)$. For each pair (s, u) if $cg(s, u) > 0$, then the pair (s, u) is inserted into the heap H. This insertion step takes $O(\log |H|)$ for a given pair. Using Eq. 16, and the relations: $m = O(n^2)$ and $d = O(n)$, it follows that $O(\log |H|) = O(\log n)$. In the subsequent steps, the UpdateCG procedure scans $O(d^2)$ pairs of nodes in the heap H. For each such pair, it invokes a Decrease-Key or an Increase-Key operation that takes $O(\log |H|) = O(\log n)$ time. Additionally, there are neighborhood set operations such as union, intersection and set difference, each of which requires $O(d)$ time. Summing up all these computation costs lead to the above lemma. □ □

LEMMA 10. *The MergeNodes procedure runs in:*

$$O(n + d^2(d + \log n)) \text{ for } \text{var} \in \{I, U\}$$
$$O(d^2(d + \log n)) \text{ for } \text{var} = E$$

PROOF. The most expensive step of the MergeNodes procedure that is common to all variants of the CoCooN algorithm, that are given by $var \in \{I, E, U\}$ is the UpdateCG procedure. From Lem. 9, this step runs in $O(d^2(d + \log n))$ time. For $var \in \{I, U\}$, there is an additional expensive step in the form of the UpdateDegree procedure. From Lem. 8, this step runs in $O(n)$ time. This leads to the stated result. □ □

LEMMA 11. *The GreedyCN procedure runs in:*

$$O(mnd + nd^3 + nd^2 \log n) \text{ for } \text{var} \in \{I, U\}$$
$$O(md^3 + md^2 \log n) \text{ for } \text{var} = E$$

PROOF. The GreedyCN procedure invokes the DiscoverCN procedure that takes $O(md)$ time (Lem. 3). Next, the procedure invokes the iterative stage from line 7.

Consider the case $var \in \{I, U\}$. From Lem. 7, the number of iterations is $O(md)$. In each iteration, the procedure extracts a pair $(u, v) \in H$. This takes $O(\log n)$ time. Further, if $cg(u, v) > 0$, then it is checked whether the merging of the pair (u, v) is safe, using the IsSafeMerge procedure. Since the IsSafeMerge procedure runs in $O(n)$ time (Lem. 8), the total computation cost of IsSafeMerge across all the iterations, is $O(mnd)$. If it is safe to merge (u, v), the pair is merged by calling the MergeNodes procedure. Following Lem. 4, the number of super-nodes produced is at most $O(n)$. From Lem. 10, it follows that the total computation cost of MergeNodes procedure across all the iterations, is $O(n^2 + nd^3 + nd + ^2 \log n)$. Finally, summing up all the above computation costs leads to $O(mnd + nd^3 + nd^2 \log n)$.

Next consider the case $var = E$. Following Lem. 7, the number of iterations is $O(m)$. In each iteration, the procedure extracts a pair $(u, v) \in H$. This takes $O(\log n)$ time. Subsequently, the pair is merged by calling the MergeNodes procedure. Following Lem. 4, the number of super-nodes produced is at most $O(m)$. From Lem. 10, it follows that the total computation cost of MergeNodes procedure across all the iterations, is $O(md^3 + md + ^2 \log n)$. Finally, summing up all the above computation costs leads to $O(md^3 + md + ^2 \log n)$. □ □

THEOREM 12. *The CoCooN algorithm runs in:*

$$O(mnd + nd^3 + nd^2 \log n) \text{ for } \text{var} \in \{I, U\}$$
$$O(n^2 + md^3 + md^2 \log n) \text{ for } \text{var} = E$$

where $n = |V|$, $m = |E|$ *and* $d = \max\{|N_G(u)||u \in G\}$.

PROOF. The CoCooN algorithm begins by checking if $m \leq n(n-1)/4$. If this is true, then it considers the graph G; else, it considersthe complementary graph $\overline{G}$. Computing the complementary graph requires $O(n^2)$ time. Following this step, the GreedyCN procedure is invoked that takes time as stated in Lem. 11. Summing up the above computation costs leads to the above theorem. □ □