Efficient Core Maintenance of Dynamic Graphs

Wen Bai[1], Yuxiao Zhang[1], Xuezheng Liu[1], Min Chen[3], and Di Wu[1,2(✉)]

[1] School of Data and Computer Science, Sun Yat-Sen University, Guangzou, China
{baiw6,zhangyx27}@mail2.sysu.edu.cn, {liuxzh36,wudi27}@mail.sysu.edu.cn
[2] Guangdong Key Laboratory of Big Data Analysis and Processing, Guangzhou, China
[3] School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
minchen@ieee.org

Abstract. $k-$core is one type of cohesive subgraphs such that every vertex has at least k degree within the graph. It is widely used in many graph mining tasks, including but not limited to community detection, graph visualization and clique finding. Frequently decomposing a dynamic graph to get its $k-$cores brings expensive cost since $k-$cores evolve as the dynamic graph changes. To address this problem, previous studies proposed several maintenance solutions to update $k-$cores based on a single inserted (removed) edge. Unlike previous studies, we maintain affected $k-$cores from the sparsest to the densest, so the cost of our method is determined by the largest core number of a graph. Experimental results show that our approach can significantly outperform the previous algorithms up to 3 order of magnitude for real graphs tested.

Keywords: $k-$core · Core maintenance · Dynamic graph

1 Introduction

$k-$core [10] is defined as the maximal subgraph of a simple graph G such that every vertex in the subgraph has at least k degree. The problem of finding the core number of all vertices in G is called core decomposition [4], which is widely used in many real-world applications, including large graph visualization [1], community detection [5], and network analysis [2].

Most graphs in our life are highly dynamic, whose edges are inserted into or removed from the graph over time. The core number of vertices should be updated to reveal the up-to-date structure of the graph. Clearly, it is uneconomical to recalculate the core number of all vertices while a few edges change. Instead, core maintenance [6,9] is proposed, whose goal is to update the core number of influenced vertices rather than decompose the entire graph. Unfortunately, existing solutions can only deal with a single edge each time, which leads to high cost when a graph with numerous inserted (removed) edges.

© Springer Nature Switzerland AG 2020
Y. Nah et al. (Eds.): DASFAA 2020, LNCS 12113, pp. 658–665, 2020.
https://doi.org/10.1007/978-3-030-59416-9_42

To overcome above drawbacks, we provide a novel solution to maintain core number of vertices with multiple inserted (removed) edges simultaneously. Compared with previous studies, our solution is relevant to the maximum core number of a graph. We conduct extensive experiments to evaluate the performance of our method and the existing solution. Experimental results show that our method can significantly outperform the previous algorithm up to 3 orders of magnitude for large real graphs tested.

The main contributions of our paper can be summarized as follows:

- With the aid of quasi−k−core, a similar but more loose concept to k−core, we can estimate the vertices affected by a set of inserted (removed) edges.
- Unlike existing approaches, our maintenance solution can update the core number of vertices in affected k−cores from the sparsest to the densest.
- Through executing extensive experiments on real graphs, our solution performs better than the existing approach.

The rest of this paper is organized as follows: Sect. 2 provides some preliminaries. The details of our solution are introduced in Sect. 3. Section 4 reports experimental results and Sect. 5 describes the related work about our paper. Finally, Sect. 6 concludes the paper.

2 Preliminaries

Usually, G represents a simple graph, which consists of a vertex set $V(G)$ and an edge set $E(G)$ such that $E(G) \subseteq V(G) \times V(G)$. For convenience, $|G| = |V(G)| + |E(G)|$ is used to represent the size of $|G|$, where $|V(G)|$ and $|E(G)|$ are the size of vertices and edges respectively. Additionally, K_0 indicates an empty graph without vertices or edges.

Given an arbitrary vertex $v \in V(G)$, we define $N(G, v) = \{u : (u, v) \in E(G)\}$ as the set of neighbors of v. Clearly, $|N(G, v)|$ is the degree of v in G, denoted by $d(G, v)$. For convenience, we also use $v \in G$ $((u, v) \in G)$ to replace $v \in V(G)$ $((u, v) \in E(G))$, where u and v are two adjacent vertices of an edge in G.

To clearly illustrate the relation between two graphs G_1 and G_2, we generalize four set notations on graphs: $G_1 \subseteq G_2$ represents G_1 is a subgraph of G_2; $G_1 \cap G_2$ refers to the intersection graph of G_1 and G_2; $G_1 \cup G_2$ is the union graph of G_1 and G_2; $G_1 \setminus G_2$ depicts the difference graph of G_1 and G_2 such that $E(G_1 \setminus G_2) = E(G_1) \setminus E(G_2)$.

k−core [10] is a well-established metric to evaluate the importance of vertices as well as their connections in the graph. Besides, k−core has two important properties: uniqueness and nestedness [4,8].

Definition 1. *A k−core is the largest subgraph of a graph G, denoted by $C(G, k)$, such that $d(C(G, k), v) \geq k$ for an arbitrary vertex $v \in C(G, k)$.*

Generally, we require $k \geq 1$. When $k = 0$, 0−core is the graph itself. If not specified, we assume isolated vertices have been removed from G. Besides, we use $C(G, k) = K_0$ to represent an empty k−core.

Similar to existing studies [4], we define the core number of vertices in G. If a vertex v is located in $k-$core but not contained in $(k+1)-$core, then its core number is k, denoted by $\phi(G, v) = k$. Additionally, the maximum core number of vertices in G is denoted as $\phi(G)$.

3 Solution

Existing methods obey the core update theorem [6,9], while an edge is inserted into (removed from) a graph, the vertices affected by this edge will change their core number at most 1. When numerous edges change, existing methods repeatedly identify influenced vertices for each edge and some of them may change their core number many times. If the number of edges is very large, the maintenance cost will become expensive.

To address the above issues, we propose a novel solution, which updates core number of influenced vertices from the sparsest $k-$core to the densest. To this end, we first propose the quasi$-k-$core to estimate the candidate vertices for each influenced $k-$core. Secondly, we identify the partial$-k-$core of each affected $k-$core and update their core number. Lastly, we increase k until all affected $k-$cores are updated.

For convenience, we use G_c to represent the current graph and G_p to indicate the previous graph before changing. Correspondingly, we define $S_i = G_c \setminus G_p$ ($S_r = G_p \setminus G_c$) as the insertion (removal) graph.

3.1 Quasi$-k-$core

Consider that most graphs are sparse, not all $k-$cores will be affected by inserted (removed) edges. To find influenced $k-$cores, an intuitive idea is to decompose S_i (S_r). Since some vertices of S_i (S_r) lack adjacent edge information in G_c (G_p), we decompose S_i (S_r) to a set of quasi$-k-$cores with the aid of G_c (G_p).

Consider that the steps of quasi core decomposition on S_i and S_r are similar, we use S (e.g. S_i or S_r) and G (e.g. G_c or G_p) to represent two arbitrary graphs for ease of presentation. To supplement extra edge information of vertices in S, we define the neighborhood graph S on G, which consists of vertices in S and their adjacent neighbors within one step in G. Similar to $k-$core, quasi$-k-$core is also unique and nested. Otherwise, it contradicts to the maximal property of quasi$-k-$core.

Definition 2. *$S(G) = (V(S(G)), E(S(G)))$ is the neighborhood graph of S on G such that $V(S(G)) = V(S) \cup \{v : v \in N(G, u) \wedge u \in V(S)\}$ and $E(S(G)) = E(S) \cup \{(u, v) : u \in S \wedge v \in N(G, u)\}$. Specially, if $S \cap G = K_0$, then $S(G) = S$.*

Definition 3. *The quasi$-k-$core $\hat{C}(S, G, k)$ is the largest subgraph of S on G such that $d(\hat{C}(G), v) \geq k$ for an arbitrary $v \in \hat{C}(S, G, k)$, where $\hat{C}(G)$ is the neighborhood graph of $\hat{C}(S, G, k)$ on G.*

To get a set of quasi$-k-$cores, we can revise the decomposition method of $k-$cores. Through recursively removing unsatisfied vertices from S for each k, we can get a set of quasi$-k-$cores. Besides, we can also define quasi core number for each vertex, which is similar to core number.

3.2 Insertion Case

Our insertion algorithm has four steps: firstly, we decompose S_i to a set of quasi$-k-$cores with the aid of G_c; secondly, we expand each quasi$-k-$core to a candidate graph; thirdly, we get the partial$-k-$core from the candidate graph; lastly, we update the core number of vertices in the partial$-k-$core and continue the next loop until all affected $k-$cores are updated.

Note that inserted edges may increase the core number of adjacent vertices of the quasi$-k-$core, but they are not contained in the quasi$-k-$core. To find all affected vertices, we expand the quasi$-k-$core to a candidate graph, which contains all possible affected vertices. Additionally, we terminate the search path when edges are contained in the previous $k-$core since they must belong to the current $k-$core.

Definition 4. *$F(k)$ is a candidate graph whose vertex $v \in C(G_c, k-1)$ satisfying $d(C(G_c, k-1), v) \geq k$ is reachable from $u \in \hat{C}(S_i, G_c, k)$ via a path and satisfies $(u', v') \notin C(G_p, k)$ for an arbitrary edge $(u', v') \in F(k)$.*

We can observe that $F(k)$ contains all vertices that may be contained in $C(G_c, k) \setminus C(G_p, k)$. So, $C(G_c, k) \subseteq F(k) \cup C(G_p, k)$ holds. Since $F(k)$ contains some redundant vertices, we identify the partial$-k-$core from $F(k)$, denoted by $P(k) = C(G_c, k) \setminus C(G_p, k)$, which is the difference graph of $C(G_c, k)$ and $C(G_p, k)$. Since $\hat{C}(P(k), C(G_p, k), k)$ is a subgraph of $P(k)$ and for any $v \in P(k)$, $d(P(C(G_p, k)), v) \geq k$ holds, we have $\hat{C}(P(k), C(G_p, k), k) = P(k)$.

To get $P(k)$ from $F(k)$, we observe that $P(k) = \hat{C}(P(k), C(G_p, k), k)$. Since $P(k) \subseteq F(k)$, we have $P(k) \subseteq \hat{C}(F(k), C(G_p, k), k)$. On the contrary, if a vertex $v \in \hat{C}(F(k), C(G_p, k), k) \setminus P(k)$, there must be a vertex $u \in C(G_p, k)$ which can be reachable from v such that $d(F(C(G_p, k)), u) < k$ via a path. Again, this is a contradiction. Consider another case $(u, v) \in \hat{C}(F(k), C(G_p, k), k) \setminus P(k)$, since $u, v \in P(k)$, $P(k) \subseteq P(k) \cup (u, v)$. When $C(G_p, k) = K_0$, we can directly get $C(G_c, k)$ from $F(k)$ since $C(G_c, k) \subseteq F(k)$.

Based on above discussions, we implement Algorithm 1 to maintain $k-$cores for the insertion case. In detail, it first decomposes S_i to a map of quasi core numbers and corresponding vertex sets. Then the algorithm updates the core number of influenced vertices from $k = 2$ to $\hat{\phi}(S_i, G_c)$ (the maximal quasi core number of S_i on G_c) according to two cases mentioned above.

Clearly, the time complexity of Algorithm 1 is $O(|S_i| + \sum_{k=1}^{\hat{\phi}(S_i, G_c)} 2|F(k)|)$, where $O(2|F(k)|)$ is the cost to get $F(k)$ and the partial$-k-$core. Note that $\hat{\phi}(S_i, G_c)$ is much less than $|V(G_c)|$, where $|V(G_c)|$ is the number of vertices in G_c. As for the space complexity, it only costs $O(|G_c|)$ to store the entire graph.

Algorithm 1: Insertion Case (IC)

Input: S_i: the insertion graph, G_c: the current graph, $\boldsymbol{\phi}$: a map of vertices and their core number.
Output: $\boldsymbol{\phi}$: a map of vertices and their core number.

```
1  decompose S_i to a set of quasi−k−cores;
2  φ(v) ← 1 for v ∈ S_i;
3  k ← 2;
4  while k ≤ φ̂(S_i, G_c) do
5  |  expand the quasi−k−core to a candiadte graph;
6  |  if k ≤ φ(G_p) then
7  |  |  get Ĉ(F(k), C(G_p, k), k) from F(k);
8  |  |  φ(v) ← k for v ∈ Ĉ(F(k), C(G_p, k), k);
9  |  |  k ← k + 1;
10 |  else
11 |  |  execute core decomposition on F(k);
12 |  |  for v ∈ F(k) do
13 |  |  |  φ(v) ← φ(F(k), v) if φ(F(k), v) ≥ k;
14 |  |  break;
15 return φ;
```

3.3 Removal Case

Our removal algorithm contains three steps: firstly, we decompose S_r to a set of quasi$-k-$cores with the aid of G_p; secondly, we delete the common edges in $\hat{C}(S_r, G_p, k) \cap C(G_p, k)$ and recursively remove influenced vertices that cannot be located in $C(G_p, k)$; thirdly, we continue the loop until all affected $k-$cores are updated.

The implementation of Algorithm 2 is relatively simple. Firstly, it decomposes S_r to a map of vertices and their quasi core number (line 1). Secondly, for each influenced $k-$core, it deletes common edges in $\hat{C}(S_r, G_p, k) \cap C(G_p, k)$, recursively removes influenced vertices and updates the core number of affected vertices. The time complexity of Algorithm 2 is $O(|S_r| + \sum_{k=1}^{\hat{\phi}(S_r, G_p)} |C(G_p, k)|)$. Since it at most traverses the entire affected $k-$core for each k, and the space complexity is $O(|G_p|)$.

4 Experiments

Our real graphs are downloaded from Koblenz Network Collection[1], including 10 real graphs (seen Table 1), where *Stanford* is a direct graph and *Youtube* is a temporal graph. For the directed graph, we ignore the edge direction and regard it as a simple graph. Then for the temporal graph, we sort their edges by the timestamp. While for the remainder graphs, we keep the initial order of edges as the corresponding graph files.

[1] http://konect.uni-koblenz.de/.

Algorithm 2: Removal Case (RC)

Input: S_r: the removal graph, G_p: the previous graph, ϕ: a map of vertices and their core number.
Output: ϕ: a map of vertices and their core number.

```
1  decompose S_r to a set of quasi−k−cores;
2  φ̂(S_r, G_p) ← φ(G_p) if φ̂(S_r, G_p) > φ(G_p);
3  k ← 1;
4  while k ≤ φ̂(S_r, G_p) do
5      let Q be an empty queue;
6      for (u, v) ∈ Ĉ(S_r, G_p, k) ∩ C(G_p, k) do
7          remove (u, v) from C(G_p, k);
8          push u (v) into Q if d(C(G_p, k), u) < k (d(C(G_p, k), v) < k);
9      adjust C(G_p, k) by removing vertices in Q;
10     update core number of affected vertices in φ;
11     k ← k + 1;
12 return φ;
```

All algorithms are implemented in C++ and compiled with GCC 7.4.0 at -O2 optimization level. All experiments are executed sequentially on the Linux operating system Ubuntu 18.04, which is running on a machine with two Xeon E5-2683v4@2.1 GHz CPUs and 128 GB RAM.

Table 1. The detail of graphs

G	Amazon	Douban	Flixster	Gowalla	Hyves	Livemocha	Skitter	Stanford	Wordnet	Youtube
$\|V(G)\|$	334, 863	154, 908	2, 523, 386	196, 591	1, 402, 673	104, 103	1, 696, 415	281, 903	146, 005	3, 223, 585
$\|E(G)\|$	925, 872	327, 162	7, 918, 801	950, 327	2, 777, 419	2, 193, 083	11, 095, 298	1, 992, 636	656, 999	9, 375, 374
$\|\phi(G)\|$	6	15	68	51	39	92	111	71	31	88

Similar to the most of existing studies, we adopt the execution time is as the metric of our experiments. In our experiments, we select the traversal approach [9] as the baseline, which contains **Trav-I** for the insertion case and **Trav-R** for the removal case. To support numerous edges, we recursively execute the traversal approach multiple times. Before experiments, we set some necessary parameters. We set the 2-hops for **Trav-I** and 1-hop for **Trav-R** in experiments, the details of these algorithms can be seen in [9].

For the insertion case, the last m edges are used to construct the insertion graph S_i and the remainder are used to construct G_p. As for the removal case, the first m edges are used to construct removal graph S_r and all edges are used to construct G_p. Generally, we vary m from $100,000$ to $200,000$ for tracing the evolution of the performance of two approaches.

Table 2 shows the execution time on all graphs for the insertion case. Compared with **Trav-I**, **IC** achieves the best performance on all graphs. Table 3 shows the execution time on all graphs for the removal case. Since both two

Table 2. The execution time for the insertion case (unit: second)

m	100, 000		120, 000		150, 000		180, 000		200, 000	
Methods	IC	Trav-I	IC	Trav-I	IC	Trav-I	IC	Trav-I	IC	Trav-I
Amazon	**6.02**	254.66	**6.79**	296.50	**7.52**	344.93	**8.34**	377.53	**8.92**	392.49
Douban	**3.36**	30.22	**3.75**	33.77	**4.79**	49.18	**5.41**	67.17	**5.89**	90.68
Gowalla	**4.72**	11.21	**5.87**	13.00	**7.23**	14.65	**8.75**	16.65	**10.39**	18.22
Stanford	**59.79**	154.22	**62.36**	177.53	**66.70**	220.50	**70.94**	257.38	**72.09**	275.71
Wordnet	**6.76**	14.96	**7.68**	18.45	**8.95**	24.78	**10.24**	29.22	**10.97**	33.29
Flixster	**48.62**	1222.00	**56.03**	1474.87	**64.67**	1674.03	**73.05**	2365.18	**78.25**	2657.49
Hyves	**7.43**	1715.22	**8.42**	2070.69	**9.32**	2616.87	**10.41**	3164.22	**10.93**	3419.63
Livemocha	**66.84**	73.95	**73.09**	88.01	**81.30**	104.80	**89.32**	144.65	**94.50**	176.27
Skitter	**4.27**	10.88	**5.02**	11.53	**6.70**	12.81	**8.69**	14.48	**10.36**	15.57
Youtube	**284.04**	336.86	**297.46**	394.72	**312.28**	488.06	**326.43**	596.92	**334.13**	731.56

Table 3. The execution time for the removal case (unit: second)

m	100, 000		120, 000		150, 000		180, 000		200, 000	
Methods	RC	Trav-R	RC	Trav-R	RC	Trav-R	RC	Trav-R	RC	Trav-R
Amazon	**2.51**	2.96	**3.02**	3.54	**3.75**	4.39	**4.58**	5.25	**5.13**	5.79
Douban	**3.25**	6.08	**3.93**	7.54	**4.55**	8.93	**5.32**	10.29	**5.84**	10.80
Gowalla	**7.88**	10.33	**9.67**	12.38	**12.14**	16.38	**15.05**	19.92	**16.53**	22.79
Stanford	**8.37**	10.33	**10.21**	12.34	**12.54**	14.64	**14.93**	16.81	**16.65**	18.87
Wordnet	**4.26**	5.56	**5.18**	6.64	**6.41**	7.90	**7.50**	8.79	**8.30**	9.80
Flixster	**12.93**	27.29	**15.71**	33.67	**19.14**	41.27	**22.14**	49.07	**24.35**	52.11
Hyves	**5.85**	12.80	**6.59**	14.11	**7.72**	16.12	**9.41**	19.13	**10.35**	20.68
Livemocha	**20.32**	30.18	**24.71**	38.63	**31.25**	48.22	**37.37**	58.04	**40.66**	63.12
Skitter	**8.77**	9.94	**10.15**	11.91	**12.74**	15.51	**15.30**	19.21	**16.95**	21.50
Youtube	**9.82**	14.18	**12.09**	17.84	**14.24**	20.34	**16.74**	22.53	**17.84**	24.44

algorithms do not search candidate vertices in the removal case, the execution time is obviously less than that of the insertion case. Even so, the performance of **RC** is still better than **Trav-R** on all graphs.

5 Related Work

$k-$core decomposition, which assigns each vertex v with a core number to reveal the connected state of v and its neighbors, is strongly related to graph degeneracy [10]. Numerous $k-$core decomposition algorithms are proposed to handle different cases. To handle $(k,h)-$core of a temporal graph, which is an extension of $k-$core, Wu *et al.* [11] proposed two distributed algorithms to deal with massive temporal graphs. Besides, the probabilistic core decomposition was also studied recently in [3], where $(k,\eta)-$cores were proposed.

$k-$core and its extensions have been extensively used in numerous applications. To solve the maximal clique problem, Lu *et al.* [7] devised a randomized algorithm by utilizing $k-$core and $k-$truss. With the aid of $k-$core, variants of

community detection problems are addressed such as local communities detection [5]. Alvarezhamelin *et al.* [2] used $k-$core as a tool to analyze large scale graphs such as social network and Internet graph.

6 Conclusions

In this paper, we propose a novel solution to tackle $k-$core maintenance of dynamic graphs, which provides an effective solution to maintain the core number of vertices affected by multiple inserted (removed) edges simultaneously. We confirm our approaches by conducting extensive experiments on 10 real graphs. The results show that our solution can outperform the existing algorithm up to 3 order of magnitude for real graphs tested.

Acknowledgment. This work was supported by the National Natural Science Foundation of China under Grant U1911201, Guangdong Special Support Program under Grant 2017T X04X148, the Fundamental Research Funds for the Central Universities under Grant 19LGZD37, 19LGYJS57.

References

1. Alvarez-Hamelin, J.I., Dall'Asta, L., Barrat, A., Vespignani, A.: Large scale networks fingerprinting and visualization using the k-core decomposition. In: Advances in Neural Information Processing Systems, pp. 41–50 (2006)
2. Alvarezhamelin, J.I., Dall'Asta, L., Barrat, A., Vespignani, A.: K-core decomposition of internet graphs: hierarchies, self-similarity and measurement biases. Netw. Heterogen. Media **3**(2), 371–393 (2017)
3. Bonchi, F., Gullo, F., Kaltenbrunner, A., Volkovich, Y.: Core decomposition of uncertain graphs. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1316–1325. ACM (2014)
4. Cheng, J., Ke, Y., Chu, S., Özsu, M.T.: Efficient core decomposition in massive networks. In: 2011 IEEE 27th International Conference on Data Engineering (ICDE), pp. 51–62. IEEE (2011)
5. Cui, W., Xiao, Y., Wang, H., Wei, W.: Local search of communities in large graphs. In: ACM SIGMOD International Conference on Management of Data, pp. 991–1002. ACM (2014)
6. Li, R.H., Yu, J.X., Mao, R.: Efficient core maintenance in large dynamic graphs. IEEE Trans. Knowl. Data Eng. **26**(10), 2453–2465 (2014)
7. Lu, C., Yu, J.X., Wei, H., Zhang, Y.: Finding the maximum clique in massive graphs. Proc. VLDB Endowment **10**(11), 1538–1549 (2017)
8. Montresor, A., De Pellegrini, F., Miorandi, D.: Distributed k-core decomposition. IEEE Trans. Parallel Distrib. Syst. **24**(2), 288–300 (2013)
9. Sarıyüce, A.E., Gedik, B., Jacques-Silva, G., Wu, K.L., Çatalürek, Ü.V.: Incremental k-core decomposition: algorithms and evaluation. VLDB J. **25**(3), 425–447 (2016)
10. Seidman, S.B.: Network structure and minimum degree. Soc. Netw. **5**(3), 269–287 (1983)
11. Wu, H., Cheng, J., Lu, Y., Ke, Y., Huang, Y., Yan, D., Wu, H.: Core decomposition in large temporal graphs. In: 2015 IEEE International Conference on Big Data (Big Data), pp. 649–658. IEEE (2015)