Project Survey for the Paper:
Efficient Progressive Minimum k-Core Search

Group Members: Deepak Kukkapalli [vk23p], Karthik Reddy Vemireddy [kv23b], and Muqeet Mohsin Shaik [ms23ch]

Paper Authors: Conggai Li , Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, Xuemin Lin

Published in: Association for Computing Machinery Proceedings of the VLDB Endowment, 2019-11, Vol.13 (3), p.361-374

1 Problem definition

What is a k-core? A k-core is a subgraph where each node has at least k connections to other nodes in the same subgraph. The authors are trying to find the largest k-core subgraph (maximum k-core) because it helps with things like measuring user engagement in social networks.
That is: A subgraph S of G is a k-core if for every vertex u in S, its degree in S is at least k, i.e.,

$$\deg_S(u) \geq k, \quad \forall u \in S$$

where:
Neighbor set of a vertex u, denoted as $N(u)$, is the set of all vertices directly connected to u.
Degree of a vertex u, denoted as $\deg(u)$, is the number of neighbors u has.
Largest degree in the graph, denoted as $d_{\max}$, is the highest degree among all vertices.
A subgraph S of G is a smaller graph formed by selecting a subset of V and E from G.
The **size** of a subgraph S, denoted as $|S|$, is the number of vertices in S.

This means that every node in the k-core must be connected to at least k other nodes within the subgraph.

Difference Between Maximum and Minimum k-Core

- **Maximum k-core:** The largest possible k-core subgraph in G that satisfies the k-core constraint. It is unique for a given k and has been widely studied.
- **Minimum k-core:** The smallest k-core subgraph that contains a given query set Q. This is the focus of this paper.

For Example: Suppose we have an undirected, unweighted graph $G = (V, E)$. A degree constraint with $k = 3$:
Here The **maximum k-core** might include all nodes that have at least 3 neighbors.
If we only care about a specific node q, we might find a much smaller k-core subgraph that still

includes q but removes unnecessary nodes.

Problem Definition: Minimum k-Core Search A query set Q (which consists of one or more important vertices).

The **Goal** is to find the smallest possible k-core subgraph that includes all nodes in Q.

Why is this problem hard? Finding the smallest k-core is NP-hard, as there's no easy, fast way to always find the perfect solution. The search space is huge, making it impractical to find the exact minimum k-core by brute force. Existing methods mostly focus on greedy approaches, which may not find the best possible solution.

2 Existing Solutions for Minimum k-Core Search

The authors discussed previous approaches for solving the minimum k-core search problem along with their limitations and classified them into two main categories:

- **Global Search (Shrink Strategy)**
- **Local Search (Expansion Strategy)**

The main Challenge is a Large Search Space: Like we discussed above, finding the exact minimum k-core is **NP-hard**, it means that brute-force methods are impractical due to the massive search space. Because of this, existing solutions use heuristic approaches to find a near-optimal solution efficiently.

2.1 Global Search (Shrink Strategy)

Key Idea: Start with a large k-core and shrink it by removing unnecessary vertices.

How it works

1. First, compute the **maximal k-core** of G.
2. The maximal k-core is a k-core of the largest possible size, and it can be computed in linear time $O(n + m)$.
3. Then, iteratively remove vertices (except the query vertex q) while ensuring that the remaining subgraph still satisfies the k-core condition.
4. Stop when further removals would violate the k-core constraint.

Why is it ineffective?

- The maximal k-core is usually very large, leading to an unnecessarily big initial search space.
- The shrinking process may still retain a large number of nodes that are not needed, making the final result not optimal.
- Studies show that global search methods do not perform well compared to local search approaches.

2.2 Local Search (Expansion Strategy)

Key Idea: Start with a small set (just the query vertex) and expand it by adding the most useful nodes.

How it works

1. Begin with the query vertex q as the initial subgraph P.
2. Maintain a **candidate set** C, which contains neighbors of vertices in P that are not yet included.
3. Iteratively add the most useful vertex from C to P, updating C accordingly.
4. Stop when all vertices in P satisfy the k-core condition.

This incremental approach helps control the size of the resulting subgraph and avoids the inefficiencies of the global search.

2.3 State-of-the-Art: S-Greedy Algorithm

A popular local search algorithm was introduced called **S-Greedy**.

How S-Greedy Works

1. Initialize P with just the query vertex q.
2. Maintain a candidate set C, consisting of neighbors of vertices in P that are not yet included.
3. In each step:
   - Pick the most promising vertex u from C.
   - Add u to P.
   - Update C by adding new candidate neighbors.
4. Stop when all nodes in P satisfy the k-core constraint.

How does S-Greedy choose which vertex to add?

S-Greedy uses a **scoring function** to measure the usefulness of a candidate vertex u. The score of u is based on two factors:

- **Positive impact** $p^+(u)$: Number of neighbors in P that currently have degree $< k$.
- Higher $p^+(u)$ means u can help more vertices in P reach the required degree.
- **Negative impact** $p^-(u)$: Number of additional vertices needed in P to ensure u has at least k neighbors.
- Higher $p^-(u)$ means u is harder to integrate into P.

The final score is computed as:

$$\text{Score}(u) = p^{+}(u) - p^{-}(u)$$

A higher score is preferred.

This greedy heuristic ensures that each added vertex maximally helps the current subgraph grow efficiently.

2.3.1 Time Complexity of S-Greedy

- Computing the score for each vertex takes $O(d_{\max})$, since we only examine neighboring vertices.
- Maintaining the best candidate in a priority queue takes $O(\log n)$ time.
- The overall complexity is $O(s(d_{\max} + \log n))$, where s is the final subgraph size.

Since s is bounded by the maximal k-core size, S-Greedy is relatively efficient compared to global search methods.

2.4 Limitations of S-Greedy and Existing Solutions

- **Not optimal:** The greedy approach does not guarantee that the final subgraph is the smallest possible k-core.
- **Quality gap:** Empirical studies show that the subgraphs produced are still much larger than the true minimum k-core.
- **No quality guarantee:** There is no way to control the trade-off between solution size and search time.
- **Users have no flexibility:** If a user wants a smaller k-core, they cannot adjust parameters to improve the result.

2.5 Need for a New Approach [our implementation solution]

Since existing methods lack control over result quality and do not guarantee an optimal or near-optimal subgraph, this paper proposes:

- A **progressive algorithm** that balances quality and efficiency.
- A method that allows users to control the trade-off between result size and computation time.
- A more systematic way of exploring the search space rather than relying purely on greedy heuristics.

(a) Intelligent Search Tree Construction

Unlike S-Greedy, which expands blindly based on a greedy heuristic, PSA builds a search tree, where:

- The root node is the query vertex.
- Each child node represents adding a new unique neighbor to the subgraph.
- Branches are explored based on their potential to form a small k-core.

(b) Using Lower and Upper Bounds

PSA refines the search using bounds:

Lower Bound Driven Search

Calculates a lower bound $s^{-}(t)$ on the smallest possible k-core containing the partial solution V_t. Expands only the branch with the smallest lower bound.

Upper Bound Driven Search

Uses a Depth-First Search (DFS) heuristic to compute an upper bound $s^{+}(t)$ on the minimum k-core size. The global upper bound is updated dynamically.

Why This Matters

- Existing methods (S-Greedy) only focus on adding good vertices but do not optimize for a small subgraph.
- PSA's bounding approach ensures a controlled expansion, minimizing unnecessary searches.

4. Comparing existing solutions with PSA

Algorithms Compared in Experiments

To validate PSA's effectiveness, we compare:

Algorithm	**Lower Bound Method**	**Optimal**	**Upper Bound Method**	**Search Strategy**
S-Greedy	None	No	Greedy heuristic	Greedy expansion
L-Greedy	None	No	Upper-bound heuristic (DFS)	Greedy expansion
PSA (Proposed Solution)	Two lower bounds (L_{sr}, L_{ie})	Yes (Approx.)	L-Greedy	Best-First Search (BesFS)
PSA-S	Greedy lower bound (L_g)	No	S-Greedy	Hybrid search
PSA-L	Greedy lower bound (L_g)	No	L-Greedy	Hybrid search

Key Findings

- **PSA outperforms S-Greedy**
  - Produces significantly smaller k-cores than S-Greedy.
  - Intelligent branch selection avoids unnecessary expansion.

- **PSA-L and PSA-S show trade-offs**
  - PSA-S (using S-Greedy) favors speed but sacrifices solution quality.
  - PSA-L (using L-Greedy) provides a better balance but lacks theoretical guarantees.
- **PSA is the best overall**
  - Balances search time and result quality.
  - Provides a provable approximation ratio, ensuring users can control the trade-off.

5. Why PSA is Fundamentally New and Better

A Novel Approach to a Known Problem Instead of relying on greedy heuristics, PSA systematically refines the search using progressive bounds. It is the first method to provide a provable trade-off between search time and solution quality.

Overcoming the Limitations of Existing Work

- Global Search is too large → PSA avoids unnecessary computation.
- S-Greedy expands blindly → PSA guides expansion using a search tree.
- No guarantees in existing methods → PSA provides an approximate guarantee.

Theoretical Justification PSA can approximate the optimal k-core within a user-defined ratio c. PSA reduces computational complexity by prioritizing promising subgraphs first.

Conclusion

PSA is a major improvement over existing solutions for the minimum k-core search problem. By introducing a progressive search with upper and lower bounds, PSA achieves:

- More accurate results (smaller k-cores).
- Better efficiency (avoiding unnecessary searches).
- A provable trade-off between quality and runtime.

PSA represents a fundamentally new approach to solving this problem and sets a new benchmark for k-core search techniques.