Vertex Connectivity via Local Computation: Breaking Quadratic Time, Poly-logarithmic Max-flows, and Derandomization

Sorrachai Yingchareonthawornchai
Vertex Connectivity via Local Computation: Breaking Quadratic Time, Poly-logarithmic Max-flows, and Derandomization

Sorrachai Yingchareonthawornchai

A doctoral thesis completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall T2, Konemiehentie 2, of the school on 25 July 2023 at 12.00.

Aalto University
School of Science
Department of Computer Science
Combinatorics of Efficient Computations
Supervising professor
Asst. Prof. Parinya Chalermsook, Aalto University, Finland

Thesis advisor
Prof. Danupon Nanongkai, Max Planck Institute for Informatics and Saarland University, Germany

Preliminary examiners
Prof. Robert Krauthgamer, The Weizmann Institute of Science, Israel
Prof. Kurt Mehlhorn, Max Planck Institute for Informatics and Saarland University, Germany

Opponent
Prof. Robert Krauthgamer, The Weizmann Institute of Science, Israel

Aalto University publication series
DOCTORAL THESES 104/2023

© 2023 Sorrachai Yingchareonthawornchai

ISBN 978-952-64-1339-6 (printed)
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)

Images: Taken from private Carcassonne tournaments

Unigrafia Oy
Helsinki 2023

Finland
Abstract

Vertex connectivity is a classic graph-theoretic notion, that roughly measures the robustness of a network against vertex failure. Vertex connectivity k of an n-vertex m-edge graph is the minimum number of vertices to be removed to disconnect the graph. A major open problem asked almost 50 years ago is whether or not vertex connectivity can be computed in linear time [Aho, Hopcroft, and Ullman 1974, Problem 5.30].

Vertex connectivity can be solved in linear time when k = 1 and k = 2 using a depth-first search tree [Tarjan’72] and a basic data structure called SPQR tree [Hopcroft and Tarjan’73], respectively. When k > 2, it remains an open problem. For general connectivity, the fastest known algorithms take time O(mn) [Henzinger, Rao and Gabow FOCS’96] and O(T(n)+nT(k)) [Linial, Lovász, Wigderson FOCS’86] where T(n) is the time to multiply two n-by-n matrices. Even for the special case when k = O(1) and m = O(n), both algorithms run at least n² time. In this case, the quadratic time barrier was known 50 years ago [Kleitman’69].

In this thesis, we break the long-standing quadratic time barrier and affirmatively resolve the open problem by Aho, Hopcroft, and Ullman (up to a sub-polynomial factor). We show the following results:
1. An O(m+nk²)-time randomized algorithm. The algorithm takes O(m+n) time whenever k is sufficiently small. Here, the poly-logarithmic factor in the running time is omitted.
2. A randomized reduction to poly-logarithmic number of many calls to a max-flow algorithm. Thus, vertex connectivity can be solved in almost linear time by using the almost linear time max-flow algorithm [Chen, Kyng, Liu, Peng, Probst and Sachdeva FOCS’22].
3. An almost linear time deterministic algorithm whenever k is sufficiently small.

To achieve these results, we introduce local computation frameworks that could be potentially extended beyond vertex connectivity. We show that fast local cut computation implies fast vertex connectivity and develop fast local cut detection algorithms that lead to the O(m+nk²)-time vertex connectivity algorithm. By making a connection to the theory of kernelization, we solve vertex connectivity in poly-logarithmic max-flows. Finally, we use vertex expanders and vertex cut sparsifiers to derandomize the vertex connectivity algorithm for small vertex connectivity.
It has been almost a decade of my Ph.D. journey with twists and turns. I started my Ph.D. study in computer science in 2014 at Michigan State University on networking and distributed systems and later switched to a new Ph.D. study in theoretical computer science in 2019 at Aalto University. The transition to theoretical study at Aalto University was the most critical decision in my academic life. It was considered an unusual choice because at the time I was in the fifth year of my former Ph.D. study and I had little background and experience in theoretical computer science research. Nonetheless, I decided to make the transition due to kind encouragement from Dr. Danupon Nanongkai and Dr. Thatchaphol Saranurak when we started working on the vertex connectivity problem, which was the starting point of the works developed in the thesis.

The transition to theoretical research was immensely challenging, but rewarding at the same time. I live in the dream that I had 5 years ago, the dream where I can pursue research in theoretical computer science and make significant contributions to the field. I want to express my immense gratitude to Dr. Danupon Nanongkai and Dr. Thatchaphol Saranurak who made this work possible. Thank you for introducing me to theoretical computer science research, and mentoring me over the last four years. I would like to thank Dr. Parinya Chalemsook, my advisor, for his support and assistance throughout my academic journey. Your guidance has been instrumental in my development as a researcher. I am especially grateful for the academic freedom to explore research directions that I am excited about the most no matter how difficult it looks, which has made a significant impact on my research and academic progress. I would like to thank Dr. Eric Torn, my former advisor at Michigan State University, who has inspired my love and passion for theoretical computer science.

I thank Dr. Chien-Chung Huang for his patient support and constructive discussion during the research visits. I would like to thank Dr. Jason Li and Dr. Debmalya Panigrahi for their insightful discussion and fruitful collaborations. I am thankful to have collaborated with Dr. Manoj Gupta. Thank you for your encouragement and stimulating discussion. I am
thankful to Dr. Omri Weinstein for his kind support and encouragement to pursue ongoing projects. Throughout my years of research, I am fortunate to have collaborated with brilliant researchers including (but not limited to) Dr. Bundit Laekhanukit, Dr. Seth Pettie, Pattara Sukprasert, Wanchote Jiamjittrak, Akask Pareek, Nidia Obscura Acosta, and Max Franck. I also thank the Nokia Foundation scholarship for supporting my research.

My time in Finland has been enjoyable thanks to my friends and colleagues. I would like to give very special thanks to P’Ed and P’Fääng for their kind support and wisdom and enjoyable board game nights. I thank Jan for inviting me to many enjoyable events during the Covid period. I thank Victoria for being with me in difficult moments. I thank Cintia for the enjoyable coffee times. I thank Francesco for the enjoyable lunch times. I thank P’Po, Ly, Yifan, and Yok for the good times hanging out together. I also thank Nancy, N’Paew, N’Ning, N’First, N’Tangmo, and N’Bing for the occasional lunch and dinner times together. I would like to also thank my friends and co-workers from the Combinatorics of Computation group, Nidia, Ameet, Michał, Kamyar, Minoo, Natalia, Gorav, and Sumedha for the friendly research environment and the time we spent together during the group meeting and discussion. I thank Poppy for being my dear friend for more than a decade even though we do not live in the same country anymore. I thank N’Jajar for the good memories when I visited Krabi. I thank P’Louise, Natty, Earth, N’Gun, N’James, N’Pim, N’Palmy, and N’Jaja for keeping in touch with me even though we are in different countries.

Lastly, I would like to thank my family for their unconditional love and support for whatever path I choose in life. Mom and Dad, thank you for allowing me to study and for doing whatever it took to ensure my access to education. I love you. My special thanks go to my brother, Kwanchai. You have been the best inspiration with your burning passion. I am extremely proud of you. As you always say, we will see each other at the top of our success.

Espoo, Finland, June 19, 2023,

Sorrachai Yingchareonthawornchai
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>1</td>
</tr>
<tr>
<td>Contents</td>
<td>3</td>
</tr>
<tr>
<td>List of Publications</td>
<td>5</td>
</tr>
<tr>
<td>Author's Contribution</td>
<td>7</td>
</tr>
<tr>
<td><strong>1. Introduction</strong></td>
<td>9</td>
</tr>
<tr>
<td>1.1 Contribution</td>
<td>11</td>
</tr>
<tr>
<td>1.2 An Overview of the Techniques</td>
<td>12</td>
</tr>
<tr>
<td>1.2.1 Breaking Quadratic Time</td>
<td>13</td>
</tr>
<tr>
<td>1.2.2 New LocalVC Algorithms</td>
<td>15</td>
</tr>
<tr>
<td>1.2.3 Sublinear Kernelization for Poly-logarithmic Max-flows</td>
<td>16</td>
</tr>
<tr>
<td>1.2.4 Fast Derandomization via Vertex Expanders and Vertex Cut Sparsifiers</td>
<td>20</td>
</tr>
<tr>
<td>1.3 Open Problems</td>
<td>22</td>
</tr>
<tr>
<td>1.4 Organization</td>
<td>23</td>
</tr>
<tr>
<td><strong>2. Preliminaries</strong></td>
<td>25</td>
</tr>
<tr>
<td>2.1 Basic Graph Notations</td>
<td>25</td>
</tr>
<tr>
<td><strong>3. Breaking Quadratic Time Bounds for Small Vertex Connectivity</strong></td>
<td>29</td>
</tr>
<tr>
<td>3.1 Local Edge Cut Detection</td>
<td>29</td>
</tr>
<tr>
<td>3.1.1 An $O(vk^2)$-Time Randomized LocalEC Algorithm</td>
<td>30</td>
</tr>
<tr>
<td>3.1.2 An $\tilde{O}(v^2k)$-Time Deterministic LocalEC Algorithm</td>
<td>32</td>
</tr>
<tr>
<td>3.2 Local Vertex Cut Detection</td>
<td>33</td>
</tr>
<tr>
<td>3.3 Vertex Connectivity Algorithm</td>
<td>36</td>
</tr>
<tr>
<td>3.3.1 An $\tilde{O}(mk^2)$-Time Vertex Connectivity Algorithm</td>
<td>36</td>
</tr>
<tr>
<td>3.3.2 Algorithm for Detecting Unbalanced Vertex Cuts</td>
<td>36</td>
</tr>
<tr>
<td>3.3.3 Algorithm for Detecting Balanced Vertex Cuts</td>
<td>37</td>
</tr>
</tbody>
</table>
This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.


Author’s Contribution

Publication I: “Breaking Quadratic Time For Small Vertex Connectivity and Approximation Scheme”

The problem formulation and directions have been suggested by Thatchaphol Saranurak and Danupon Nanongkai. The final technical result is a product of collaborative work by everyone in the paper. The works involve weekly regular meetings where the author of this thesis has attended all the meetings and performed most of the detail verifications and contributed partially to the ideas. The first draft of the manuscript is done almost solely by the author of this thesis.

Publication II: “Computing and Testing Small Connectivity in Near-Linear Time and Queries via Fast Local Cut Algorithm”

The development of the new local cut algorithms was a joint effort among everyone in the paper. Initially, the new local cut algorithms were independently developed by Forster and Yang [16], and Nanongkai, Saranurak, and the author of this thesis [37]. Shortly after, we decided to merge the paper that appeared in SODA 2020. The preparation of the article is the contribution of everyone in the paper. The author of this thesis played a key role in preparing the technical sections of the article.

Publication III: “Vertex Connectivity in Poly-logarithmic Max-flows”

Everyone in the paper made a crucial technical contribution to the work. A direction of kernelization was suggested by Thatchaphol Saranurak and the final technical result was a product of collaborative work by everyone. The author of this thesis participated in regular meetings over the course
of the project and verified most of the details. The preparation of the article was contributed by everyone. The author of this thesis played a key role in preparing the technical sections of the article.

**Publication IV: “Deterministic Small Vertex Connectivity in Almost Linear Time”**

The problem formulation and directions have been suggested by Thatchaphol Saranurak. The final technical result was a product of collaborative work by everyone. The author of this thesis participated in regular meetings and verified technical details. The preparation of the article was a combined effort by everyone. The author of this thesis played a central role in preparing the technical sections of the article.

**Publication V: “Engineering Nearly Linear-time Algorithms for Small Vertex Connectivity”**

The author of this thesis suggested the plan for conducting the experiments and extensions. Max Franck developed new heuristics that outperformed the basic implementations without the heuristics. He also conducted extensive experiments. The preparation of the article was a joint effort from Max Franck and the author of this thesis.
1. Introduction

Vertex connectivity is a classic connectivity notion in graph theory. The *vertex connectivity* $\kappa$ of an $n$-node $m$-edge undirected graph is the minimum number of vertices to be removed in order for the remaining graph to be disconnected or to become a singleton. Vertex connectivity finds applications in many areas including network design and sociology. In fault-tolerant network design, vertex connectivity measures the robustness of the graph against vertex failures. In sociology, vertex connectivity measures social cohesion [48].

There has been a long line of research on efficient algorithms for computing vertex connectivity [28, 41, 14, 13, 18, 33, 5, 31, 11, 34, 10, 23, 24, 17, 6], see Table 1.1 for the summary. A major open problem is whether or not vertex connectivity can be solved in linear time. Aho, Hopcroft, and Ullman [1, Problem 5.30] asked in 1974 if there exists an $O(m)$-time algorithm for computing vertex connectivity. So far, linear time algorithms are known only when $\kappa \leq 2$ using depth-first search tree (Tarjan’72 [45]), and a basic data structure called SPQR tree (Hopcroft and Tarjan’73 [25]). However, $\kappa \geq 3$, no algorithms to date are faster than $O(n^2)$ time.

We will focus mainly on deciding $k$-vertex connectivity, i.e., given an integer $k$, and the task is to decide if $\kappa \geq k$; if $\kappa < k$, we must output the corresponding set of vertices, called *vertex cut*. If $\kappa \geq k$, we say that the graph is $k$-vertex connected. The decision version and vertex connectivity problem are equivalent (up to an $O(\log n)$ factor) because $\kappa$ can be computed by a binary search on parameter $k$ of the decision version.

The $O(n^2)$-time algorithm for deciding $k$-vertex connectivity was known half a century ago by Kleitman [28] in 1969 if the graph is sparse, i.e., $m = O(n)$. They presented an algorithm with running time of the form $O(kn \text{VC}_k(n,m))$ where $\text{VC}_k(n,m)$ is the running time to decide if $\kappa(s,t) \geq k$ where $\kappa(s,t)$ is the minimum number of vertices to remove to disconnect a pair of vertices $s$ and $t$ on an $n$-vertex $m$-edge graph$^1$. It was known that $\text{VC}_k(n,m) = O(mk)$ by Ford and Fulkerson [15] in 1956. So when $m = O(n)$

$^1$We are not allowed to delete $s$ and $t$; if there is an edge between $s$ and $t$, then $\kappa(s,t) = n - 1$. 

and $k = O(1)$, this gives $O(kn \text{VC}_k(n,m)) = O(k^2 mn) = O(n^2)$ time.

### Table 1.1. Algorithms for deciding $k$-vertex connectivity for undirected graphs.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Running Time</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trivial</td>
<td>$O(n^2 \cdot \text{VC}(n,m))$</td>
<td></td>
</tr>
<tr>
<td>Kleitman’69 [28]</td>
<td>$O(kn \cdot \text{VC}_k(n,m))$</td>
<td></td>
</tr>
<tr>
<td>Podderyugin’73 [41]; Even, Tarjan’75 [14]</td>
<td>$O(kn \cdot \text{VC}(n,m))$</td>
<td></td>
</tr>
<tr>
<td>Even’75 [13] (cf. [18, 12, 33])</td>
<td>$O((k^2 + n) \cdot \text{VC}_k(n,m))$</td>
<td></td>
</tr>
<tr>
<td>Becker et al.’82 [5]</td>
<td>$O(n \cdot \text{VC}(n,m))$</td>
<td>Monte Carlo</td>
</tr>
<tr>
<td>Linial, Lovász, Wigderson’88 [31]</td>
<td>$O((n^w + nk^w) \log n)$</td>
<td>Monte Carlo</td>
</tr>
<tr>
<td>Nagamochi, Ibaraki’92 [34]; Cheriyan, Thurimella’91 [11]</td>
<td>$O(k^3 n^{1.5} + k^2 n^2)$</td>
<td>Las Vegas</td>
</tr>
<tr>
<td>Henzinger’97 [23]</td>
<td>$O(\min(\sqrt{n}, k)n^2)$</td>
<td>2-approx.</td>
</tr>
<tr>
<td>Henzinger, Rao, Gabow’96 [24]</td>
<td>$O((kn^w \log n) = \tilde{O}(mn)$</td>
<td>Monte Carlo</td>
</tr>
<tr>
<td>Gabow’00 [17]</td>
<td>$O(\min(n^{3/4}, k^{1.5})k^2 n + kn^2)$</td>
<td></td>
</tr>
<tr>
<td>Censor-Hillel, Ghaffari, Khun’14 [6]</td>
<td>$\tilde{O}(m)$</td>
<td>Monte Carlo, $O(\log n)$-approx.</td>
</tr>
<tr>
<td><strong>This thesis</strong></td>
<td>$\tilde{O}(m + nk^{4})$</td>
<td>Monte Carlo</td>
</tr>
<tr>
<td><strong>This thesis</strong></td>
<td>$\tilde{O}(\text{VC}(m, m)) \leq m^{1+o(1)}$</td>
<td>Monte Carlo</td>
</tr>
<tr>
<td><strong>This thesis</strong></td>
<td>$m^{1+o(1) + O(k^2)}$</td>
<td></td>
</tr>
</tbody>
</table>

The quadratic time bound has been a long-standing barrier for over 50 years. Subsequent works improve over Kleitman’69 [28] on higher graph density (larger $m$) and larger connectivity $k$, but none of them could get below quadratic time $O(n^2)$ even when $m = O(n)$ and $k = 4$. When $k = 4$, the bound $O(n^2)$ for any $m$ was shown by Kanevsky and Ramachandran [26] in 1991. When $k = O(1)$, the bound $O(n^2)$ for any $m$ was shown by Nagamochi and Ibaraki [34] in 1992. The state-of-the-art algorithms for general graphs run in $\tilde{O}(n^w + nk^w)$ by Linial, Lovász, and Wigderson [31] and in $\tilde{O}(n^2 k) = \tilde{O}(mn)$ time² by Henzinger, Rao, and Gabow [24] where $\tilde{O}(\cdot)$ hides polylog factors, and $\omega < 2.37287$ is the matrix multiplication exponent. The state-of-the-art deterministic algorithm is due to Gabow [17] in 2000 with the running time of the form $O(\min(n^{3/4}, k^{1.5})k^2 n + kn^2)$. In 1997, Henzinger [23] presents a deterministic 2-approximation $O(\min(\sqrt{n}, k)n^2)$-time algorithm³. More recently, a randomized $O(\log n)$-approximation $\tilde{O}(m)$-time algorithm was introduced by Censor-Hillel, Ghaffari, and Kuhn [6].

²$m \geq nk$ because we can assume WLOG that the minimum degree is at least $k$.

³We say that an algorithm is $t$-approximation if it outputs a vertex cut of size at most $t$ factor from the size of a minimum vertex cut.
In terms of techniques, the state-of-the-art algorithms are based on either flow methods or algebraic methods. For flow methods, Even and Tarjan (1975) [14] show that computing $\kappa(s,t)$ can be solved in a single call to a maximum flow algorithm, i.e., $\text{VC}(m,n)$ is at most the time to solve a max flow instance on a directed unit-capacity graph of $O(m)$ edges and $O(n)$ vertices. Subsequent works have been exploiting the optimal solution's structure to reduce the number of calls of max-flows [13, 18, 12, 33]. The best reduction is due to Becker et al. [5] in 1982 where they showed a reduction to $\tilde{O}(n)$ calls to max-flows. The algorithms by Henzinger Rao and Gabow [23] are based on the push-relabel algorithm. They consider a single-source version of vertex connectivity: given a fixed source $s$, compute $\min_t \kappa(s,t)$. Trivially, we can compute $\min_t \kappa(s,t)$ by $O(n)$ calls of max-flows. They show that single-source vertex connectivity can be solved in a single call to preflow-based algorithms in $\tilde{O}(mn)$ time. Using random sampling, they show that vertex connectivity can be solved in $O(\log n)$ calls of single source vertex connectivity. This implies their $\tilde{O}(mn)$ time randomized algorithm. For algebraic methods, Linial, Lovász, Wigderson [31] showed in 1988 a reduction to matrix multiplication when $k = O(1)$, and a larger number of calls to matrix multiplication when $k$ is large. Subsequently, in 1994, Cheriyan, Reif [10] showed the same bounds for directed graphs.

For $k \geq 4$, all previous exact algorithms could not go below $O(n^2)$ for a common reason: They have to solve the $k$-single-source vertex connectivity, i.e., deciding if $\min_t \kappa(s,t) \geq k$ for a given source $s$. In all previous algorithms, there is always a vertex $s$ such that they need to decide if $\min_t \kappa(s,t) \geq k$. So far, there is no $o(n^2)$-time algorithm for $k$-single-source vertex connectivity even if $k = 4$ and $m = O(n)$.

1.1 Contribution

In this thesis, we break the long-standing quadratic time barrier and affirmatively resolve the open problem by Aho, Hopcroft and Ullman [1, Problem 5.30] up to a sub-polynomial factor. This thesis consists of three main contributions.

Contribution 1: Breaking Quadratic Time Barrier. In Publication I and Publication II, we design a randomized algorithm for deciding $k$-vertex connectivity that runs in $\tilde{O}(m + nk^3)$, which is $\tilde{O}(m)$ time whenever $k = O(\log^{O(1)}n)$. This breaks $O(n^2)$-time bound that was known since 1969 by Kleitman [28] for the special case when $k = O(1)$, and $m = O(n)$. The techniques are based on local vertex cut detection, which is different from all previous algorithms. The algorithm is simple and easy to implement. Furthermore, the algorithm has been shown to outperform the $\tilde{O}(mn)$-algorithm by Hezinger, Rao, and Gabow [24] on both synthetic and real-world datasets in Publication V even for small graphs.
Contribution 2: A Reduction to Poly-logarithmic Max-flows. Vertex connectivity can be trivially solved by computing $O(n^2)$ calls of max flows on vertex-unit-capacity graphs (by calling max-flows for all pairs) since $\kappa = \min_{x,y} \kappa(x,y)$. The number of calls can be reduced to $O(n)$ calls using randomization by Becker et al. [5], which was the best-known reduction to max-flows. In Publication III, we show that vertex connectivity can be solved using only $\log^{O(1)}(n)$ calls to max-flows using random sampling with our new tool called sublinear kernelization, which roughly means compressing a graph into a smaller graph of sublinear size while preserving the vertex connectivity. By using the $m^{1+o(1)}$-time maximum flow algorithm [9], we achieve an $m^{1+o(1)}$-time algorithm for vertex connectivity, answering the long-standing open problem asked by Aho, Hopcroft, and Ullman [1, Problem 5.30] (up to a sub-polynomial factor).

Contribution 3: Derandomization. One may ask if the power of randomization is necessary to achieve an almost linear time algorithm. There is still a big gap in the complexity between deterministic algorithms and randomized algorithms for vertex connectivity. For deterministic algorithms, the quadratic time $O(n^2)$ was known in 1969 by Kleitman [28] when $m = O(n)$ and $k = O(1)$, and the-state-of-the-art is due to Gabow [17] with the running time $O(\min\{n^{3/4}, k^{1.5}k^2n + kn^2\})$, which is still at least quadratic time $\Omega(n^2)$ even if $k = O(1)$. In Publication IV, we make progress towards derandomization. Specifically, we show that vertex connectivity can be solved in $m^{1+o(1)} \cdot 2^{O(k^2)}$ time, breaking the long-standing quadratic time barrier without randomization. One can view this algorithm as an attempt to derandomize the $\tilde{O}(m + nk^3)$-time algorithm for small vertex connectivity. The key tools for derandomization are a reduction framework to vertex expanders and vertex cut sparsifier.

1.2 An Overview of the Techniques

We develop new tools based on the local/sublinear algorithm paradigm to obtain these results in this thesis. Using known local computation algorithms, we explain how to break the quadratic time bound in Section 1.2.1 for any constant connectivity $k$. To obtain the $\tilde{O}(m + nk^3)$ time, we introduce an improved local computation algorithm In Section 1.2.2. The new local algorithm is essentially a depth-first search (DFS) with early termination. Furthermore, we show that the new local algorithm can be used to develop stronger property testers for testing $k$-edge/vertex connectivity. We explain the sublinear kernelization techniques to solve vertex connectivity in poly-logarithmic max-flows in Section 1.2.3. Finally, we explain the techniques for fast deterministic vertex connectivity in Section 1.2.4.
Notations. For any subset of vertices $L$, the volume of $L$, denoted by $\text{vol}(L)$, is $\text{vol}(L) = \sum_{v \in L} \text{deg}(v)$ where the degree of a vertex $v$, $\text{deg}(v)$, is the number of edges incident to $v$ in the graph, and the neighbors of $L$, denoted by $N(L)$, is the set of vertices (outside of $L$) having an edge incident to $L$. For any two vertex sets $S, T \subseteq V$, we denote $E(S, T)$ as the set of edges $(u, v)$ where $u \in S$ and $v \in T$. A vertex cut $(L, S, R)$ in $G = (V, E)$ is a partition of the vertex set into $L, S, R$ such that there is no edge between $L$ and $R$. We say that a vertex cut $(L, S, R)$ is of size $k$ if $|S| = k$. If $|S| = \kappa$, we say that $(L, S, R)$ is a vertex mincut and $S$ is a min separator. WLOG, we can assume that $|L| \leq |R|$ (otherwise, we can swap $L$ and $R$). We also say that the vertex set $S$ is a vertex separator (or simply a separator) in $G$ if $G$ becomes disconnected after removing all vertices in $S$. Sometimes we put a graph $G$ as a subscript (e.g., $E_G(S, T), N_G(L)$, etc.) to specify the graph we are referring to and omit it if the context is clear. We say that an algorithm succeeds with high probability if the success probability is at least $1 - n^c$ for any constant $c \geq 1$.

1.2.1 Breaking Quadratic Time

For simplicity of discussion, we assume $k = O(1)$ and $m = O(n)$. This simplification is for the overview only. Even for this case, the fastest known algorithm takes $O(n^2)$ time. The goal, for now, is to obtain a subquadratic time algorithm for deciding $k$-vertex connectivity. Now, assume that the graph $G$ has a vertex cut $(L, S, R)$ of size $< k$. The goal, for now, is to output a vertex cut of size $< k$ in $\tilde{O}(n^{1.99})$ time.

Finding a Balanced Vertex Cut Is Easy. We first demonstrate that the difficult case is only when $|L| \leq n^{0.01}$ (i.e., the vertex cut is unbalanced). In other words, using techniques in prior works, we can break $O(n^2)$ time if the optimal cut is somewhat balanced, i.e., $|L| \geq n^{0.01}$. The algorithm for balanced cuts is as follows: Sample the set $T$ of $10n^{0.99}\log n$ random pairs of vertices followed by checking if $\kappa(s, t) < k$ for all $(s, t) \in T$. Since checking if $\kappa(s, t) < k$ can done in $O(mk) = O(n)$ time using Ford-Fulkerson max-flow algorithm, the algorithm runs in $O(n^{1.99}k \log n) = \tilde{O}(n^{1.99})$ time. To see the correctness, it is enough to show that there is a pair $(s, t)$ in the sample set such that $s \in L$ and $t \in R$ with high probability. Indeed, the probability that a random vertex is in $R$ is $|R|/n \geq \frac{n - |L| - k}{n} \geq \Omega(1)$. So, the probability that a random pair $(x, y)$ hits $L$ and $R$ (so that $L \ni x$ and $R \ni y$) is $\frac{|L|}{n} \cdot \frac{|R|}{n} \geq \Omega(n^{0.01})$. Thus, sampling $10n^{0.99}\log n$ pairs suffices. However, the same approach fails when $|L| = O(1)$ because we would need $\Omega(n)$ random pairs, and each pair would take $O(n)$ time to run the Ford-Fulkerson algorithm.

Local Vertex Cut Detection. To handle the unbalanced vertex cut, we introduce our new key algorithmic tool called local vertex cut detection. The local vertex cut problem can be phrased as follows. Suppose we are given a
vertex $x$ in a graph $G$ where $x$ is in a vertex set $L \subseteq V$ with volume $\text{vol}(L) \leq v$ and $|N(L)| < k$ where $v \ll m/k$. Can we output $N(L)$ or a separator of size $< k$ using time that only depends on parameters $v$ and $k$, but not $n$? Intuitively, the algorithm is local in that we need to find a small separator “near” $x$ using the time that depends only on $v$ and $k$. We say that a vertex cut $(L, S, R)$ is $(v, k)$-local w.r.t. $x$ if $x \in L, |S| < k$ and $\text{vol}(L) \leq v$. This motivates the following definition.

**Definition 1.2.1.** An algorithm is LocalVC if it takes an initial vertex $x$ of a graph $G = (V, E)$ and two (proximity) parameters $v$ and $k$ where $v = O(m/k)$ and output either a separator of size $< k$ or the symbol $\perp$ certifying that there is no $(v, k)$-local vertex cut w.r.t. $x$.

Note that the returned separator does not have to be a $(v, k)$-local vertex cut w.r.t. $x$. That is, we allow to output a separator that could have large volume than $v$ or a min-degree cut. This definition is sufficient for the purpose of computing (global) vertex connectivity; it also gives more flexibility to design an algorithm than restricting to only output $(v, k)$-local vertex cut w.r.t. $x$.

Previously, there was rich literature on local algorithms for a different cut problem called low conductance cuts. The conductance of a cut $(S, V \setminus S)$ is $\Phi(S) = \frac{\sum_{e \in E(S, V \setminus S)} |e|}{\min \{\text{vol}(S), \text{vol}(V \setminus S)\}}$. This notion was first studied by Spielman and Teng [43] in 2004. Later, several other techniques have been developed using spectral-based techniques [44, 2, 3, 4, 19] or flow-based techniques [38, 22, 47, 46]. Applications of these techniques for finding low conductance cuts are found in various contexts (e.g. balanced cuts [44, 42]), edge connectivity [27, 22], and dynamically maintaining expanders [49, 35, 36, 42]).

Furthermore, a similar version of local vertex cut detection appears in the context of property testing [20], see Section 3.1 for elaborated discussion. Also, several property testing algorithms for testing $k$-vertex connectivity (and also for $k$-edge connectivity) were introduced [21, 40, 50, 39] (see Section 1.2.2 for more discussion). Our contribution is to establish a connection from local computation to global computation for vertex connectivity.

**Local-to-Global Vertex Connectivity.** To demonstrate the power of local computation in vertex connectivity, we sketch the ideas for how to break the $O(n^2)$ time barrier for any constant $k$. By adapting the local cut detection algorithm from Chechik et al. [8], we obtain the LocalVC algorithm with the running time $O(vk^{O(k)})$. Recall that the difficult case is when the vertex cut of $(L, S, R)$ of size $< k$ is unbalanced, i.e., $|L| \leq n^{0.01}$. The local computation approach is suitable for this situation. That is, we run the $O(vk^{O(k)})$ LocalVC algorithm on every single vertex with parameters $v = |L|^2 + k|L| = \Theta(n^{0.02})$ and the same parameter $k$ as the connectivity. The total running time is $n \cdot O(vk^{O(k)}) = O(n^{1.02})$. To see the correctness, observe that there is at least one execution where $x \in L$ and $v \geq |L|^2 + |S||L| \geq \text{vol}(L)$. By Definition 1.2.1, it must output a separator of size $< k$. 

14
To improve the running time further, we sample a set of vertices to run the LocalVC algorithm instead of running from every vertex. The full algorithm is slightly more complex, and with some effort, we can prove that the algorithm that runs in $\tilde{O}(m + nk^{O(k)})$ time by this framework.

### 1.2.2 New LocalVC Algorithms

Using existing a local algorithm (from Chechik et al. [8]) together with the framework in Section 1.2.1, we obtain $\tilde{O}(m + nk^{O(k)})$-time vertex connectivity algorithm, which is $\tilde{O}(m)$ if $k = O(1)$. To handle larger $k$, we introduce a faster randomized vertex connectivity algorithm that runs in $\tilde{O}(mk^2)$ time, which is $\tilde{O}(m)$ whenever $k = \log^{O(1)}(n)$. To do so, we design a faster LocalVC algorithm with running time $O(vk^2)$ (improving from $O(vk^{O(k)})$ from Chechik et al. [8]). Applying the same framework in Section 1.2.1 along with the faster LocalVC algorithm, we obtain a randomized $\tilde{O}(m + nk^{3})$-time vertex connectivity algorithm. If $k \leq \log^{O(1)} n$, then the algorithm runs in $\tilde{O}(m)$ time. We present the full algorithm in Chapter 3.

### Experimental Results

We conducted experiments on planted cuts where we can control the size and volume of the smaller side of the vertex cut, and real-world datasets with four algorithms denoted as LOCAL1, LOCAL1+, LOCAL2+, and HRG. LOCAL1 is a vanilla implementation of the $\tilde{O}(mk^2)$-time algorithm described in Section 3.3. LOCAL1+ and LOCAL2+ are the optimized versions of LOCAL1 with additional heuristics called degree counting. HRG is the $\tilde{O}(mn)$-time algorithm by Henzinger, Rao and Gabow [24]. HRG is the fastest alternative to the new algorithms for small connectivity. The key finding of our experimental results can be summarized as follows. For graphs with planted cuts, LOCAL1, LOCAL1+, and LOCAL2+ scale much better than HRG for a fixed $\kappa_G \leq 8$; they start to perform better than HRG on graphs as small as $n \leq 500$. For the real-world datasets, LOCAL1+ and LOCAL2+ are the fastest among the four algorithms where LOCAL2+ is slightly faster than LOCAL1+. See Section 3.4 for details.

### Applications to Property Testing

Furthermore, our new technique for LocalVC answers open problems about efficient property testing algorithms for $k$-vertex and $k$-edge connectivity. To state the results precisely, we introduce the testing model and related notations. The study of property testing has been initiated by Goldreich, Goldwasser, and Ron [21]. For any $\epsilon > 0$, we say that an $m$-edge graph is $\epsilon$-far from having a property $P$ if we need to insert or delete for at least $\epsilon m$ edges to make the graph satisfy $P$. The key question in property testing is how many edges do we need to query (or access) to decide whether or not the input graph satisfies property $P$ or is $\epsilon$-far from having property $P$. The tester is allowed to have a one-sided error in that if the graph is $\epsilon$-far from having $P$, then the tester reports “no” with probability at least $2/3$. Here, the property $P$ in
consideration is either $k$-edge connected or $k$-vertex connected.

The query complexity for testing $k$-edge connectivity for undirected graphs is well understood: The state-of-the-art tester for undirected $k$-edge connectivity takes $\tilde{O}(\frac{1}{\epsilon^4})$ queries\footnote{Parnas and Ron \cite{parnas2012testing} provided a tester with $\tilde{O}(\frac{k}{c_d^4})$ query complexity for undirected $k$-edge connectivity where $d$ is the average degree. By an observation in \cite{publicationii}, we can assume WLOG that $d \geq k/4$, and thus the bound becomes $\tilde{O}(\frac{1}{\epsilon^4})$.} \cite{parnas2012testing}. However, the query complexity for directed $k$-edge connectivity is less understood. The state-of-the-art tester takes $\tilde{O}\left(\left(\frac{\epsilon}{k}\right)^{O(k)}\right)$ queries \cite{orenstein2019property,parnas2012testing} for some constant $c > 1$. Furthermore, The state-of-the-art testers for $k$-vertex connectivity in both directed and undirected graphs take $\tilde{O}\left(\left(\frac{\epsilon}{k}\right)^{O(k)}\right)$ queries \cite{parnas2012testing} for some constant $c > 1$. Orenstein and Ron \cite{orenstein2019property} asked whether or not the exponential (in $k$) bounds from \cite{orenstein2019property,parnas2012testing} can be made polynomial.

In this thesis, we answer this question affirmatively. We design property testers for directed $k$-edge connectivity and (un)directed $k$-vertex connectivity with $\tilde{O}(\frac{k}{\epsilon^2})$ queries, which is nearly linear in $k$. The results are shown in Table 1.2. We remark that we show only the simplified bounds and only for the unbounded-degree incident-lists model. See \cite{publicationii} for more discussion regarding bounded degree incident-lists model and the precise bounds.

<table>
<thead>
<tr>
<th>Property Testing Problem</th>
<th>Query Complexity</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undirected $k$-edge connectivity</td>
<td>$\tilde{O}\left(\frac{1}{\epsilon^4}\right)$ \cite{parnas2012testing}</td>
<td>See footnoote</td>
</tr>
<tr>
<td>Directed $k$-edge connectivity</td>
<td>$\tilde{O}\left(\left(\frac{\epsilon}{\frac{k}{4}}\right)^{O(k)}\right)$ \cite{orenstein2019property,parnas2012testing}</td>
<td>Simplified</td>
</tr>
<tr>
<td>(Un)directed $k$-vertex connectivity</td>
<td>$\tilde{O}\left(\left(\frac{\epsilon}{\frac{k}{4}}\right)^{O(k)}\right)$ \cite{parnas2012testing}</td>
<td>Simplified</td>
</tr>
</tbody>
</table>

### 1.2.3 Sublinear Kernelization for Poly-logarithmic Max-flows

We explain how to solve vertex connectivity in poly-logarithmic calls of $(s,t)$-max-flows where $s$ is the source and $t$ is the sink of the network.

**Background.** The main goal for the reduction to max-flows is to obtain a pair of vertices $s$ and $t$ such that $s$ and $t$ are on the different sides of the vertex mincut $(L,S,R)$, i.e., $s \in L, t \in R$. Given such $s$ and $t$, a vertex mincut can be extracted from an $(s,t)$-max-flow on a vertex-unit-capacity graph by Menger’s theorem (or more generally by max-flow min-cut theorem). Trivially, one can obtain $\kappa$ by computing the minimum of all $(s,t)$-max-flows for all pairs of vertices $s$ and $t$. This takes $O(n^2)$ calls to max-flows. Using random sampling, Becker et al. \cite{becker2020property} show a reduction to $O(n)$ calls to

### Table 1.2. Query complexity in unbounded degree model.
max-flows, which was the best-known reduction.

**Our Approach.** In our reduction, we do not attempt to reduce the number of iterations, but instead, we bound the total size \( \sum_i |E(G_i)| \leq \tilde{O}(m) \) where we run \((s_i, t_i)\)-max-flows on a graph \( G_i \) at iteration \( i \) and \( E(H) \) denotes the set of edges of a graph \( H \). The key is that we make calls to the “compressed instances” \( G_i \)'s of sublinear sizes. Thanks to the sublinear size of max-flow instances, we can boost the chance to obtain the “correct” max-flow instance by sampling a lot more trials than those without compression.

**Demonstration.** To demonstrate the power of this approach, let us consider the following situation for connectivity \( k \geq n/100 \). The input graph contains a vertex mincut \((L, S, R)\) where \(|S| = k - 1 \) and \(|L| \leq |R|\). This is the case where all previous algorithms run \( \Omega(n^3) \) time where \( k = \Omega(n) \). We explain how to generate max-flow instances of \( \tilde{O}(n^2) = \tilde{O}(m) \) edges in total\(^5\).

The first step is to sample a vertex in \( L \). Since the probability that a random vertex is in \( L \) is \( |L|/n \), it is enough to sample \( \tilde{O}(n|L|) \) vertices\(^6\). With high probability, there is one vertex in the sample \( x \in L \). Now, assume that we have \( x \in L \). The key task is the following.

*Given a vertex \( x \in L \), describe an algorithm that outputs a max-flow instance with \( \tilde{O}(n|L|) \) edges where a vertex mincut in the original graph can be extracted from the max-flow instance.*

In other words, we want to construct a graph \( H \) and a vertex \( y \) such that \((x, y)\)-max-flow in \( H \) tells us about the vertex mincut in the original graph. The graph \( H \) is also known as kernel in parameterized algorithms. The construction of \( H \) is called kernelization. This would imply that the total size of all max-flow instances is

\[
\tilde{O}(\frac{n}{|L|} \cdot n|L|) = \tilde{O}(n^2) = \tilde{O}(m).
\] (1.1)

**Warmup.** As a warmup, we show the existence of a kernel of \( O(n|L|) \) edges. First, we contract \( R \) in graph \( G \) into a single vertex \( y \). The \((x, y)\)-max-flow value remains \(|S|\) in the contracted graph. However, the graph can have many edges. Next, we remove all edges inside \( S \) (i.e., both endpoints are incident to \( S \)). Call this graph \( H \). This reduction does not change the \((x, y)\)-max-flow. Thus, \( H \) is a kernel of \( G \). Furthermore, the number of edges become \( O(n|L|) \) since \( |E(H)| \leq |L|^2 + |L||S| + |S| = O(n|L|) \). The key challenge is we do not know the set \( R \) in the beginning. It turns out that our kernelization essentially follows this algorithm, but we contract a (random) subset of vertices in \( R \) instead.

\(^5\)We can assume WLOG that the minimum degree is at least \( k \). Otherwise, we can output the min-degree vertex cut. Thus, \( m \geq nk \geq \Omega(n^2) \).

\(^6\)One can estimate the number \(|L|\) up to a factor of 2 by trying for all \( 2^i \) for integers \( i \leq \log n \).
Sampling Strategy and Kernelization. Our strategy starts with sampling a random subset of vertices so that it contains only vertices in $R$ before applying the kernelization method. Define the set $T$ where each vertex $v \in V$ is selected into $T$ with probability $1/|L|$. Let $T_x = T - N_G[x]$ where $N_G[x]$ denotes the set of neighbors of $x$ including $x$ itself. We drop subscription $G$ when the context is clear. We claim that $T_x \subseteq R$ with probability $\Omega(1)$. Observe that $|L \cup S| \leq k + |L|$. Since $N(x) \subseteq L \cup S$, and $|N(x)| \geq k$, we conclude that

\[
|(L \cup S) - N_G[x]| \leq |L|.
\]  

(1.2)

So, $T_x \subseteq R$ if and only if none of the vertices in $(L \cup S) - N_G[x]$ is sampled\(^7\). Since $|(L \cup S) - N_G[x]| \leq |L|$ and each node is sampled with probability $1/|L|$, $T_x \subseteq R$ with probability $\geq (1 - 1/|L|)^{|L|} = \Omega(1)$.

Therefore, by repeating for $\Theta(\log n)$ time there is a trial $T$ such that $T_x := T - N[x] \subseteq R$ with high probability. We now assume $T_x \subseteq R$. That is, $x \in L$ and $T_x \subseteq R$. Next, we describe kernelization. First, we contract $T_x$ into a vertex $t_x$. Observe that we can extract vertex mincut in the original graph from $(x, t_x)$-max-flow in this graph. However, the number of edges can be large. To reduce the size of the graph further, we employ the following reduction rules. We sketch the correctness proofs showing that the vertex mincut is "preserved" after the reduction. The formal proofs can be found in Section 4.2.2.

- **Reduction 1.** Remove all edges inside the neighbors of $x$ and all edges inside the neighbors of $t_x$
- **Correctness.** The max-flow between $x$ and $t_x$ does not change because there exists a collection of $|S|$ vertex-disjoint paths between $x$ and $t_x$ where each path contains exactly one neighbor from $x$ and one neighbor from $t_x$.
- **Reduction 2.** Remove $N(x) \cap N(t_x)$ all common neighbors of $x$ and $t_x$.
  Let $Z = N(x) \cap N(t_x)$.
- **Correctness.** Any vertex neighboring to both $x$ and $t_x$ must be in $S$. We can put $Z$ back to the vertex mincut later.
- **Reduction 3.** Remove degree-1 neighbors of $t_x$ (i.e., they are adjacent to only $t_x$).
- **Correctness.** They cannot be part of an $(x, t_x)$-vertex mincut.

After the reduction, we obtain the kernel $H$ of $G$. We can obtain vertex mincut of $G$ from $H$ as follows. Let $C$ be a min $(x, t_x)$-separator (i.e., the graph $H$ after removing $C$ does not have a path from $x$ to $t_x$ and $C$ is of the minimum size) obtained from an $(x, t_x)$-max-flow in $H$. By the correctness of the reduction rules and the fact that $x \in L$, $T_x \subseteq R$, $C \cup Z$ is a min separator in $G$.

\(^7\)We also need to prove that $T_x \cap R \neq \emptyset$ with probability $\Omega(1)$, but this intuitively follows because $|R| \geq |L|$ is large. We defer the proof to Section 4.2.1.
**Kernel Size.** It remains to prove that $H$ has $O(n|L|)$ edges with high probability. We have two distinguished vertices $x$ and $t_x$ in $H$. We classify the set of vertices in $H$. First, let $Z = N_G(x) \cap N_G(T_x)$ be the set of common neighbors of $x$ and $T_x$ in $G$, which is the same set defined in Reduction 2. By design, the set of neighbors of $x$ in $H$ is $N_x = N_G(x) - Z$, and the set of neighbors of $t_x$ in $H$ is $N_t = N_G(T_x) - Z$. We denote $F$ to be the remaining vertices (being “far” from $x$ and $t_x$). The set of vertices in the kernel can be partitioned as follows:

$$V(H) = \{x, t_x\} \cup N_x \cup N_t \cup F.$$  

See Figure 1.1 for illustration. Denote $G - N[x]$ to be the graph after deleting all vertices from $N_G[x]$. We first show that the number of edges in $H$ can be charged to $N_x \cup F$.

![Figure 1.1. Kernel $H$ of graph $G = (V,E)$. Note that there is no edge such that both two endpoints are in $N_x$ or both in $N_t$, i.e., $E_H(N_x,N_x) = E_H(N_t,N_t) = \emptyset$.](image)

**Lemma 1.2.1.** $|E(H)| \leq \sum_{v \in N_x \cup F} \deg_{G - N[x]}(v) + |N_x| + |N_t|$.

**Proof.** Recall that we have discarded all internal edges inside $N_x$ and $N_t$. Thus, we have three types of edges in $H$:

- (E1) edges in $N_x \times (F \cup N_t)$, i.e. edges with one endpoint in $N_x$ and the other in $F \cup N_t$,
- (E2) edges in $F \times (F \cup N_t)$, i.e. edges with one endpoint in $F$ and the other in $F \cup N_t$, and
- (E3) edges incident to $x$ and $t_x$.

The number of edges in (E3) is at most $|N_x| + |N_t| \leq n$. The number of edges in (E1) and (E2) is $\sum_{v \in N_x \cup F} \deg_{G - N[x]}(v)$. \qed

Next, we show that every vertex in $N_x \cup F$ has low degree with high probability.

**Lemma 1.2.2.** For any $v \neq x$, if $\deg_{G - N[x]}(v) \geq 10|L|\log n$, then $v \notin N_x \cup F$ with high probability.
Proof. Recall that $T$ was constructed by sampling each vertex with probability $1/|L|$. Since $|N_G(v) - N_G[x]| > 10|L|\log n$, i.e., $v$ has many neighbors outside $N_G(x)$, the neighbors of $x$, one of these neighbors must have been sampled to $T$ with high probability, and thus $v \in N_G(T_x) \cup T_x$. If $v \in T_x$, then $v$ becomes $t_x$. Otherwise, $v \in N_G(T_x)$. In any case, we have that $v \not\in N_x \cup F$ with high probability.

Therefore, Lemma 1.2.1 and Lemma 1.2.2 imply that, with high probability, the kernel size is

$$|E(H)| \leq \sum_{v \in N_x \cup F} \deg_{G - N_G(x)}(v) + O(n) \leq |N_x \cup F| \cdot O(|L| \log n) + n = \tilde{O}(n|L|).$$

By Equation 1.1, this concludes the overview of the reduction to polynomial max-flows for $k = \Omega(n)$. Observe that to construct a kernel efficiently, we cannot afford to touch every edge in the graph since the kernel size is sublinear. To handle the challenge, we apply techniques from sublinear algorithms such as linear sketching and sparse recovery in order to build the kernel in time proportional to the size of the kernel.

For general $k$, we prove a better kernel of size $\tilde{O}(k|L|)$. The smaller kernel is required to prove that the overall kernel size (including the $\tilde{O}(n/|L|)$ term from the sampling so that $x \in L$) is

$$\tilde{O}(k|L|) \cdot \tilde{O}(n/|L|) = \tilde{O}(nk) = \tilde{O}(m).$$

To do so, we conduct a more fine-grained analysis of the same kernelization and apply new techniques including isolating vertex cut lemma. We present the reduction algorithm that works for all $k$ and the data structure for fast kernel construction in Chapter 4.

### 1.2.4 Fast Derandomization via Vertex Expanders and Vertex Cut Sparsifiers

We discuss how to obtain a fast deterministic algorithm for small connectivity. The challenge is to avoid sampling a pair of vertices used by previous randomized algorithms. To address the challenge, we develop two algorithmic tools and show that the combination of the tools almost immediately implies a fast deterministic algorithm for small connectivity. For now, we focus on $k = O(1)$, and the goal is to decide $k$-vertex connectivity without randomization.

Our starting point is that deciding $k$-vertex connectivity for vertex expander can be done efficiently without sampling a pair of vertices. For any vertex cut $(L,S,R)$, its expansion is $h(L,S,R) = \frac{|S|}{\min(|L|,|R|)+|S|}$. Note that $h(L,S,R) \in (0,1)$. We say that $(L,S,R)$ is $\phi$-sparse if $h(L,S,R) < \phi$. We say that a graph $G$ is a $\phi$-vertex expander if $\min(L,S,R) h(L,S,R) \geq \phi$. We will use $\phi = 1/n^{o(1)}$, and simply call $G$ a vertex expander. The key property vertex expander is that if a vertex mincut $(L,S,R)$ has size $< k$, then
min(|L|,|R|) ≳ k\nu^{\Theta(1)}, i.e., the vertex mincut must be unbalanced. Since the vertex mincut is unbalanced, we can apply the same algorithm as discussed in Section 1.2.1: for every vertex x in G, run O(\nu k^{O(1)})-LocalVC algorithm with parameters \nu = k\nu^{\Theta(1)} and connectivity parameter k. Since k = O(1), and \nu = k\nu^{\Theta(1)} = O(n^{\Theta(1)}), the overall running time is n \cdot O(\nu k^{O(1)}) = n^{1+\Theta(1)} to decide k-vertex connectivity in vertex expanders.

However, the graph in general may not be an expander. We introduce two algorithmic tools to handle general graphs. The first tool is a reduction framework to vertex expanders and a terminal set.

“Almost” Reduction to Vertex Expanders. Given a graph G and parameters \phi and k, the algorithm EXPANDER\textsc{OR\textsc{TERMINAL}}(G,k,\phi) runs in m^{1+\Theta(1)/\phi} time and outputs a collection \mathcal{X} of \phi-expanders and a terminal set T \subset V the following guarantee:

1. G is k-vertex connected if and only if every expander in \mathcal{X} is k-vertex connected and \kappa_G(x,y) \geq k for all x,y \in T,
2. \mathcal{X} has almost linear in total size. That is, \sum_{X \in \mathcal{X}} |V(X)| \leq n^{1+\Theta(1)}, and
3. |T| \leq n^{1+\Theta(1)}\phi.

The framework allows us to focus on checking connectivity between a pair inside T (i.e., to decide if \kappa_G(x,y) < k for some pair x,y \in T). Indeed, we first call EXPANDER\textsc{OR\textsc{TERMINAL}}(G,k,\phi) where \phi = 1/n^{\Theta(1)} to obtain a collection of expanders \mathcal{X} and a terminal set T with aforementioned properties. Since we can solve k-vertex connectivity in O(|V(X)|^{1+\Theta(1)}) time for each expander X \in \mathcal{X}, we spend O(\sum_{X \in \mathcal{X}} |V(X)|^{1+\Theta(1)}) = O(n^{1+\Theta(1)}) time in total to verify k-vertex connectivity of every expander in \mathcal{X}. If we find a small vertex cut in one of the expanders, then we are done. Otherwise, every expander is k-vertex connected. By the first property, it remains to check the connectivity between a pair inside T. Our next algorithmic tool is suitable for this situation.

Vertex Cut Sparsifiers The goal is to compress the graph G into a smaller graph H where H preserves all vertex mincuts between a pair inside T where the number of edges in H only depends on |T| and k. We make the notion of compression precise.

Let G and H be two graphs that share the same terminal set T. For A,B \subseteq T, we say that S is an (A,B)-weak separator if there is no path from A \setminus S to B \setminus S in G \setminus S. We denote \mu_G(A,B) as the minimum (A,B)-weak separator in G. Note that by definition \mu_G(A,B) \leq \min(|A|,|B|). We say that G and H are (T,k)-equivalent if for all A,B \subseteq T, we have

\min(\mu_G(A,B),k) = \min(\mu_H(A,B),k).

Intuitively, this means up to connectivity k, the min (A,B)-weak separator has the same size. We say that H is k-cut recoverable from G if V(H) \subseteq V(G) and every separator of size < k in H is a separator in G. By definition
of $k$-cut recoverable, $\kappa_H \geq \min\{k, \kappa_G\}$. We are ready to define $k$-vertex cut sparsifier.

**Definition 1.2.2.** Let $G$ and $H$ be two graphs that contain the same terminal set $T$. We say that $H$ is a $(T,k)$-sparsifier for $G$ if $H$ and $G$ are $(T,k)$-equivalent and $H$ is $k$-cut recoverable from $G$.

Kratsch and Wahlström [29] prove that there exists a vertex cut sparsifier (for any value of $k$) of $O(|T|^3)$ vertices which can be constructed in polynomial time. The construction is randomized and succeeds with high probability. Also, Liu [32] shows that vertex cut sparsifier of $|T|k^{O(1)}$ vertices can be constructed in polynomial time. For our purpose, however, the construction must be deterministic, and fast. That is, we would need a fast deterministic algorithm for constructing a $(T,k)$-sparsifier for $G$ of $|T|f(k)$ vertices for some function $f$ that depends on $k$. Indeed, we show $f(k) = 2^{O(k^2)}$ can be done in almost linear time deterministically. That is, we provide an algorithm that constructs a $(T,k)$-sparsifier $H$ for $G$ in $m^{1+o(1)}2^{O(k^2)}$ time where $|E(H)| = O(|T|)$. To achieve the result, we adapt some of the techniques from the edge version [7] into the vertex version. We discuss the technical challenges and the key differences in Section 5.3.

Using our fast algorithm for computing a vertex sparsifier of small size, we can decide $k$-vertex connectivity as follows. Let $T$ be the terminal set obtained from EXPANDORTERMINAL($G,\phi,k$). Then, we compute a $(T,k)$-sparsifier $H$ for $G$ in almost linear time. Since $H$ preserves all vertex mincut between $T$ and $\kappa_H \geq \min\{k, \kappa_G\}$, it is enough to check if $H$ is $k$-connected, i.e., we recurse on $H$ and repeat the same framework. Since $\phi = 1/(2^{O(k^2)}n^{o(1)})$, and $|T| \leq n^{1+o(1)}\phi$, we have $|E(H)| = O(|T|) = O(n^{1+o(1)}\phi) \leq n/10$ (with appropriate parameters). Therefore, we have reduced the instance of vertex connectivity by at least a constant fraction per recursion. We can repeat this framework for $O(\log n)$ time and we are done. This concludes the high-level overview of our framework. We present the proofs in Section 5.4.

### 1.3 Open Problems

We believe our results and techniques open new directions and perspectives regarding vertex cuts. We discuss intriguing open problems for vertex connectivity.

**Deterministic Vertex Connectivity Algorithms.** An outstanding open problem is whether there is a linear-time deterministic vertex connectivity algorithm as asked by Aho, Hopcroft, and Ullman in 1974 [1, Problem 5.30]. In this thesis, we resolve this question up to a sub-polynomial factor when the connectivity is $k = o(\sqrt{\log n})$. An immediate open problem

---

8For technical reason, we would need $T' = \bigcup_{v \in T} N^k(v)$ instead of $T$ where $N^k(v)$ is an arbitrary set of $k$ neighbors of $v$. See also Section 5.4
is to improve the dependency on $k$ to be $m^{1+o(1)}k^{O(1)}$. This would follow immediately from our framework if $(T,k)$-sparsifier of size $|T|k^{O(1)}$ can be computed in $m^{1+o(1)}k^{O(1)}$ time. For general connectivity $k$, obtaining an $\tilde{O}(mn)$-time deterministic vertex connectivity algorithm asked by Gabow [17] in 2006 remains open.

**Weighted Vertex Connectivity.** A natural extension of vertex connectivity is the weighted vertex mincut problem: Given a vertex-weighted graph, the *weighted vertex connectivity* is to compute a vertex separator with the minimum total weight in the separator. The state-of-the-art algorithm is due to Henzinger, Rao and Gabow [24]. They provide $\tilde{O}(mn)$-time randomized (Monte Carlo) algorithm, and $\tilde{O}(\kappa_0 mn)$-time deterministic algorithm where $\kappa_0$ is the undirected vertex connectivity. It is an astounding open problem to break the $\tilde{O}(mn)$ bound.

**Vertex Connectivity in Directed Graphs.** What is the landscape for directed vertex connectivity? In directed graphs, vertex connectivity is the minimum number of vertices to be removed so that the remaining graph is not strongly connected. The state-of-the-art algorithms prior to our work run in $\tilde{O}(mn)$ time by Henzinger, Rao and Gabow [24] and $\tilde{O}(n^{\omega} + nk^{\omega})$ time by Cheriyan, Reif [10]. An intriguing open problem is to obtain an almost linear time for directed vertex connectivity. Our results in this thesis can be extended partially to directed graphs to get almost linear time for either small or large connectivity. Namely, the algorithm in Chapter 3 gives $\tilde{O}(mk^2)$ time for directed vertex connectivity, which is nearly linear when $k = \log^{O(1)}(n)$. Furthermore, our kernelization techniques in Chapter 5 result in an $n^{2+o(1)}$ time for directed vertex connectivity, which is almost linear when $k = \Omega(n)$.

### 1.4 Organization

This thesis is organized as follows. In Chapter 2, we present background and notations that will be used throughout the rest of the chapters. In Chapter 3, we present simple algorithms that break the quadratic time bounds for small connectivity along with experimental results. We then present the algorithm that solves vertex connectivity in poly-logarithmic max-flows in Chapter 4. Finally, we present the fast deterministic algorithm for small vertex connectivity in Chapter 5.
2. Preliminaries

In this chapter, we explain basic graph notations. For the convenience of set notations, for any two sets \( A, B \), we denote \( A - B \) as the set difference of \( A \) and \( B \).

2.1 Basic Graph Notations

Let \( G = (V, E) \) be an undirected graph where \( V \) denotes the set of vertices and \( E \) denotes the set of edges. We also denote \( V(G) \) and \( E(G) \) as a set of vertices and edges of \( G \), respectively. By default, we denote \( m = |E| \) and \( n = |V| \). We use subscripts to denote the graph of interest, and we omit them if the context is clear. For any two sets of vertices \( A, B \subseteq V \), we denote \( E_G(A, B) \) as the set of edges having one endpoint in \( A \) and the other endpoint in \( B \). The degree of a vertex \( v \in V \), denoted as \( \deg_G(v) \) is the number of edges incident to \( v \). The volume of a vertex set \( S \subseteq V \), denoted as \( \text{vol}_G(S) \), is the total degree, i.e., \( \text{vol}_G(S) := \sum_{v \in S} \deg_G(S) \). For any vertex set \( S \subseteq V \), we denote \( G[S] \) as an induced subgraph of \( G \).

**Neighborhoods.** Let \( S \subseteq V \) and \( v \in V \). The cut-set of \( S \), \( \delta_G(S) \), is the set of edges having exactly one endpoint in \( S \). We denote \( \delta_G(v) := \delta_G(\{v\}) \) is the set of edges incident to \( v \). The external neighbors (or simply neighbors) of \( S \), \( N_G(S) \), is the set of vertices outside \( S \) that have an edge to \( S \). We denote \( N_G(v) := N_G(\{v\}) \), which is the set of vertices having an edge to \( v \). We denote the inclusive neighbors of \( S \) as \( N_G[S] := N_G(S) \cup S \).

**Paths.** Let \( x, y \in V \) and \( A, B \subseteq V \). We say that a path is an \((x, y)\)-path if it starts with \( x \) and ends with \( y \). We also say that a path is an \((A, B)\)-path if it is an \((x, y)\)-path for some \( x \in A, y \in B \).

**Cut-Sets.** For any edge set \( F \subseteq E \), denote \( G - F \) as the graph after removing edges \( F \) from \( G \). A set of edges \( F \subseteq E \) is called a cut-set if \( G - F \) is not connected.

**Vertex Cuts and Separators.** For any vertex set \( S \subseteq V \), we denote \( G - S \) as the graph after removing vertices \( S \). Also, we say that \( S \) is a vertex
We say that $S$ with $A$ (\cite{34})

**Theorem 2.1.1**

is an vertex cut. An is $G$ directed graph if the graph is directed, then $E$ sparsification algorithm for Edge Sparsification.

Let $S \subseteq V$, two disjoint subsets $A, B \subseteq V$, and two different vertices $s, t \in V$. We say that $S$ is an $(A, B)$-separator if $S \cap A = \emptyset$, $S \cap B = \emptyset$ and $G - S$ does not have an $(A, B)$-path. We say that $S$ is an $(s, t)$-separator if it is an $(\{s\}, \{t\})$-separator. A vertex cut $(L, S, R)$ is an $(A, B)$-vertex cut if $S$ is an $(A, B)$-separator, i.e., $A \subseteq L$ and $B \subseteq R$. It is an $(s, t)$-vertex cut if it is $(\{s\}, \{t\})$-vertex cut. An $(A, B)$-vertex mincut is a $(A, B)$-vertex cut with minimum size. Note that, by the definition of separators, $S$ does not allow to intersect with $A, B$. We also consider the situation where $S$ can also delete $A$ and $B$. We say that $S$ is an $(A, B)$-weak separator if $A \subseteq S$ or $B \subseteq S$ or $G - S$ does not have an $(A - S, B - S)$-path. For $x, y \in V$, we denote $\kappa_G(x, y)$ is the size of a minimum $(x, y)$-separator or $n - 1$ if it does not exist. The vertex connectivity of $G$ is $\kappa_G = \min_{x, y} \kappa_G(x, y)$. We denote $\mu_G(A, B)$ as the size of a minimum $(A, B)$-weak separator. By definition, $\mu_G(A, B) \leq \min(|A|, |B|)$.

For the purpose of vertex connectivity, we can assume WLOG that the graph is simple, i.e., no-self loop and no parallel edges. We say that a graph $G$ is $k$-vertex connected if $\kappa_G \geq k$.

**Edge Sparsification.** We state a linear time sparsification algorithm for undirected graphs that allow us to assume $m \leq nk$ WLOG for undirected graphs.

**Theorem 2.1.1** (\cite{34}). There is an $O(m)$-time algorithm that takes an undirected graph $G = (V, E_G)$ and a parameter $k$ as inputs, and outputs another graph $H = (V, E_H)$ satisfying the following properties:

- $G$ is $k$-vertex connected if and only if $H$ is $k$-vertex connected. Moreover, for all $S \subseteq V$ such that $|S| < k$, $S$ is a separator in $H$ if and only if $S$ is a separator in $G$, and

- $H$ has arboricity $k$, i.e., $|E_H(S, S)| \leq k|S|$ for all $S \subseteq V$. In particular, $|E_H| \leq nk$.

**Directed Graphs.** If the graph is directed, then $E_G(A, B)$ denotes the set of edges $(u, v)$ where $u \in A, v \in B$. We use the superscript out for

$$\deg^\text{out}(v), \vol^\text{out}(S), \delta^\text{out}(S), N^\text{out}_G(S)$$

to emphasize that we restrict the set of edges in consideration to those outgoing edges from $S$ (or $v$). Similarly, we use in for

$$\deg^\text{in}(v), \vol^\text{in}(S), \delta^\text{in}(S), N^\text{in}_G(S)$$
to restrict the set of edges to be incoming edges to $S$ (or $v$).

In a directed graph $G$, a vertex set $S$ is a separator if $G - S$ is not strongly connected. Vertex connectivity in a directed graph is to compute a minimum separator. A vertex cut $(L, S, R)$ is a vertex partition $L, S, R$ of $V$ such that there is no edge from $L$ to $R$. Notice that unlike the undirected case, there may be an edge from $R$ to $L$. For undirected graphs $G = (V, E)$, we can replace each edge $(u, v) \in E$ with directed edges in both directions $(u, v)$ and $(v, u)$ to turn to a directed graph without changing the vertex connectivity.
3. Breaking Quadratic Time Bounds for Small Vertex Connectivity

The goal of this chapter is to prove the following: Given a graph $G$ and a connectivity parameter $k$, there is a randomized (Monte Carlo) $k$-vertex connectivity algorithm that runs in $\tilde{O}(mk^2)$ time if $G$ is directed, and $\tilde{O}(m + nk^3)$ time if $G$ is undirected. The key tool is the local cut detection algorithm. We discuss fast local edge cut detection algorithms in Section 3.1, and show a black box reduction from local vertex cut detection to local edge cut detection in Section 3.2. In Section 3.3, we use local vertex cut to decide $k$-vertex connectivity in $\tilde{O}(mk^2)$ time for directed graphs. The bounds for undirected can be obtained by replacing $m = \Theta(nk)$ using linear-time edge sparsification algorithm by Nagamochi and Ibaraki [34]. Finally, we discuss experimental results in Section 3.4.

3.1 Local Edge Cut Detection

Similar to the local vertex cut detection problem (Definition 1.2.1), the local edge cut detection problem can be phrased as follows. Given an initial vertex $x$ in a graph $G$ where $x$ belongs to a component $S$ where $\text{vol}^\text{out}(S) \leq \nu$ and $|\delta^\text{out}(S)| < k$ where $\nu \ll m/k$, can we output the cut-set $\delta^\text{out}(S)$ or any cut-set of size $< k$ using time that only depends on $\nu$ and $k$? We say that a vertex set $S$ is $(\nu, k)$-local w.r.t. $x$ if $x \in S, \text{vol}^\text{out}(S) \leq \nu$ and $|\delta^\text{out}(S)| < k$. We call such an algorithm as local edge cut detection (LocalEC for short).

**Definition 3.1.1.** An algorithm is LocalEC if it takes an initial vertex $x$ of a directed graph $G = (V, E)$ and two (proximity) parameters $\nu$ and $k$ where $\nu = O(m/k)$ and outputs either a set $S$ where $|\delta^\text{out}(S)| < k$ or the symbol $\bot$ certifying that there is no $(\nu, k)$-local vertex set w.r.t. $x$. In the randomized (Monte Carlo) version, the certification is allowed to be incorrect with probability at most $1/3$. More precisely, the algorithm outputs either a set $S$ where $|\delta^\text{out}(S)| < k$ or $\bot$. The algorithm has a property that if there is a $(\nu, k)$-local vertex set w.r.t. $x$, then it outputs $\bot$ with probability at most $1/3$.

Table 3.1 summarizes the bounds for LocalEC algorithms. Goldreich [20]
Table 3.1. Complexity of LocalEC algorithms

<table>
<thead>
<tr>
<th>Reference</th>
<th>Time</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goldreich [20]</td>
<td>$O(dks^{3-2/k})$</td>
<td>Undirected, $</td>
</tr>
<tr>
<td>Chechik et al. [8], This thesis</td>
<td>$O(\nu k^{O(k)})$</td>
<td></td>
</tr>
<tr>
<td>This thesis</td>
<td>$O(\nu^2 k \log(\nu k))$</td>
<td></td>
</tr>
<tr>
<td>This thesis</td>
<td>$O(\nu k^2)$</td>
<td>Monte Carlo</td>
</tr>
<tr>
<td>This thesis</td>
<td>$O(\nu^{1.5} k)$</td>
<td>Monte Carlo</td>
</tr>
</tbody>
</table>

considers a slightly different guarantee on $\perp$ with parameter $s$ (for the size of the set instead of volume) so that if $\perp$ is returned, then it certifies that there is no vertex set $S$ containing $x$ such that $|\delta^{\text{out}}(S)| < k, |S| \leq s$ and the induced subgraph $G[S]$ is $k$-edge-connected. They present an $O(dks^{3-2/k})$-time LocalEC algorithm (w.r.t. their setting) on undirected graphs [20, Theorem 9.11] where $d$ is the maximum degree of the graph. More recently, Chechik et al. [8] propose an $O(\nu k^{O(k)})$-time LocalEC algorithm. In this thesis, we design two randomized and one deterministic LocalEC algorithms with the running time $O(\nu k^2), O(\nu^{1.5} k)$ and $O(\nu^{1.5} k \log(\nu k)) = \tilde{O}(\nu^2 k)$, respectively.

We describe two LocalEC algorithms with running time $O(\nu k^2)$ and $\tilde{O}(\nu^2 k)$ in Section 3.1.1 and Section 3.1.2, respectively. We will use the $O(\nu k^2)$-time randomized LocalEC algorithm in this chapter, and the $O(\nu^{1.5} k)$-time deterministic LocalEC algorithm in Chapter 5. As the $O(\nu^{1.5} k)$-time LocalEC algorithm is not used for solving vertex connectivity in this thesis, we defer to proof to Publication I instead. In Section 3.2, we show a generic reduction from LocalVC algorithms to (directed) LocalEC algorithms with essentially no overhead.

3.1.1 An $O(\nu k^2)$-Time Randomized LocalEC Algorithm

Recall that we start from an initial vertex $x$, and we have two parameters $\nu$ and $k$ where $\nu k \ll m$.

**Algorithm.** Run the depth-first search algorithm (DFS) starting at $x$ and early stop whenever we have explored $8\nu k$ edges (i.e., the number of distinct edges accessed during the DFS process has reached $8\nu k$). The DFS will run until either we find a component or we early stop after having explored $8\nu k$ edges. If a component $S$ is found by the DFS, then we return $S$. Otherwise, the DFS must early stop, and we sample a random edge $(u, v)$ among the set of $8\nu k$ explored edges. Then, we reverse every edge along the path from $x$ to $u$ in the DFS tree and repeat the same procedure for $k$ iterations. If we do not find a connected component by the end of $k$-th iteration, we return $\perp$.  

30
Running Time. The algorithm takes $O(vk^2)$ time because there are $k$ iterations, and DFS takes $O(vk)$ time per iteration.

Correctness. The following two lemmas imply that the algorithm is indeed LocalEC.

Lemma 3.1.1. If the algorithm outputs a component $S$, then $|\delta^\text{out}_G(S)| < k$.

Proof. Let $S$ be the returned component. We claim that at any iteration of DFS, the set of outgoing edges ($\delta^\text{out}(S)$) from $S$ can be reduced at most one after reversing a path. If true, then $|\delta^\text{out}_G(S)| < k$ because $S$ is returned at one of the $k$ iterations of DFS (where $\delta^\text{out}(S) = \emptyset$). We now prove the claim. At the beginning of an iteration, let $H$ be a graph and let $P$ be the path from $x$ to $u$ obtained in the algorithm. Denote $E(P)$ as the set of directed edges in $P$. By design, $x \in S$. Let $H'$ be the graph $H$ after reversing all edges in $P$. By design, the number of out-going edges from $S$ can change by only the set of edges having exactly one endpoint in $S$ in the path $P$ (either out-going or in-coming edges). Now, there are two cases:

- If $u \in S$, then $|\delta^\text{out}_H(S) \cap E(P)| = |\delta^\text{in}_H(S) \cap E(P)|$. Thus, $|\delta^\text{out}_H(S)| = |\delta^\text{out}_H(S)| - 1$. That is, the number of outgoing edges from $S$ after reversing $P$ does not change because the number of edges along the path $P$ going out of $S$ equals the number of edges going in $S$ along the path $P$.
- If $u \not\in S$, then $|\delta^\text{out}_H(S) \cap E(P)| = |\delta^\text{in}_H(S) \cap E(P)| + 1$. Thus, $|\delta^\text{out}_H(S)| = |\delta^\text{out}_H(S)| - 1$. That is, the number of outgoing edges is reduced by one after reversing $P$ because the number of edges along the path $P$ going out of $S$ is one more than the number of edges going in $S$ along the path $P$.

\[ \square \]

Lemma 3.1.2. If there is a vertex set $S \subseteq V$ such that $x \in S, |\delta^\text{out}_G(S)| < k, \text{vol}_G(S) \leq \nu$, then the algorithm return $\perp$ with probability at most a constant.

Proof. We assume WLOG that $|\delta^\text{out}_G(S)| = k - 1$ and there is no cut-set of size $< k - 1$. It is enough to prove that the probability that we reverse the path from $x$ to some vertex outside $S$ for all $k - 1$ iterations is at least a constant. If this event happens, it guarantees that a component is returned by iteration $k$. We say that $i$-th iteration is successful if we obtain a path from $x$ to some vertex outside $S$. Let $v_i$ be the volume of $S$ at iteration $i$. Observe that $v_i \leq \nu$ because the volume of $S$ can only decrease after reversing a path that starts in $S$. The probability of success at $i$-th iteration is $1 - \frac{v_i}{\nu k} \geq 1 - \frac{1}{\frac{\nu}{k}}$. Therefore, the probability of success at every iteration is $\prod_{i \leq k - 1} (1 - \frac{v_i}{\nu k}) \geq (1 - \frac{1}{\frac{\nu}{k}})^{k - 1} = O(1)$.

\[ \square \]
3.1.2 An $O(v^2k)$-Time Deterministic LocalEC Algorithm

Recall that we start from an initial vertex $x$, and we have two parameters $v$ and $k$ where $vk \ll m$. We first define the absorbing graph and its structure. Then we sketch the implementation in $O(v^2k)$ time.

**Definition 3.1.2.** Given a graph $G$ with parameters $x, v, k$, the absorbing graph $G'$ is defined as follows. We start with graph $G$. For each edge $(u, v)$ in the graph $G$, we define its capacity to be $v$. We add new two vertices $s, t$. Intuitively, $s$ is a source, and $t$ is a sink. Then, we add a new edge $(s, x)$ with capacity $vk + 1$. For each vertex $v$, we add a new edge $(v, t)$ with capacity $\deg_{G'}^+(v)$. Intuitively, $G'$ is absorbing in that the amount of flow injecting to $s$ will be absorbed in every vertex.

We introduce notations to simplify the discussion. We say that a vertex set $S$ is $(v, k)$-local w.r.t. $x$ in $G$ if $\text{vol}_{G}(S) \leq v$, and $|\delta_{G}(S)| < k$. For any edge $e$ in $G'$, denote $c_{G'}(e)$ as the capacity of $e$ in $G'$. For any set of edges $E$ in $G'$, we define the capacity of $E$ to be $c_{G'}(E) = \sum_{e \in E} c_{G'}(e)$. For any vertex set $S$ in $G'$, we denote the capacity $c_{G'}(S) = c_{G'}(\delta_{G'}^+(S))$. A vertex set $S$ in $G'$ is an $(s, t)$-cut (or simply a cut) if $s \in S$ but $t \notin S$. An $(s, t)$-mincut is a cut with minimum capacity. Let $S^*$ be an $(s, t)$-mincut of the absorbing graph $G'$.

**Lemma 3.1.3.** If there is a $(v, k)$-local cut in $G$ that contains $x$, then $c_{G'}(S^*) \leq vk$.

**Proof.** Let $S$ be a $(v, k)$-local vertex set w.r.t. $x$ in $G$. We prove that the capacity of the cut $S' = S \cup \{s\}$ in $G'$ is at most $vk$, i.e., $c_{G'}(S') \leq vk$. Since $x \in S$, every edge $(u, v)$ in the cut-set $\delta_{G'}^+(S \cup \{s\})$ either ends at $t$ or not. If it ends at $t$ (meaning $v = t$), then its capacity is $\deg_{G'}^+(u)$. Otherwise, $c_{G}(u, v) = v$. The total capacity due to those ending at $t$ is $\sum_{u \in S} \deg_{G'}^+(u) = \text{vol}_{G}(S)$, and the total capacity due to the rest is $v|\delta_{G}(S)|$. Therefore, the total capacity of the cut $S'$ in $G'$ is $c_{G'}(S') = \text{vol}_{G}(S) + v|\delta_{G}(S)| \leq v + v(k - 1) \leq vk$. \qed

**Lemma 3.1.4.** Let $S = S^* - \{s\}$. If $c_{G'}(S^*) \leq vk$, then $x \in S$ and $S$ is a $(v, k)$-local cut in $G$.

**Proof.** Since $S^*$ has capacity $\leq vk < m$, its cutset cannot contain an edge $(s, x)$, and it cannot be a trivial cut, i.e., $\delta_{G}^+(S^*) \neq \{s\}$ and $\delta_{G}^+(S^*) \neq \{t\}$. Thus, $S$ contains $x$ and every edge $(u, v)$ in the cut-set $\delta_{G}^+(S \cup \{s\})$ in $G'$ either ends at $t$ or not. If it ends at $t$ (meaning $v = t$), then its capacity is $\deg_{G}^+(u)$. Otherwise, $c_{G}(u, v) = v$. Thus, the total capacity of the $(s, t)$-mincut $S^*$ is $|\delta_{G}^+(S)| + \text{vol}_{G}(S) = c_{G}(S^*) \leq vk$. Since $|\delta_{G}(S)| \geq 0$, we have $\text{vol}_{G}(S) \leq vk$. Since $\text{vol}_{G}(S) \geq \deg_{G}^+(x) > 0$, we have $|\delta_{G}^+(S)| \nu < vk$. Therefore, $|\delta_{G}^+(S)| < k$, and $\text{vol}_{G}(S) \leq vk$. \qed

**Algorithm and Correctness.** We first compute an $(s, t)$-mincut $S^*$ in the absorbing graph $G'$. If $c_{G'}(S^*) \leq vk$, then output $S^* - \{s\}$. By Lemma 3.1.4,
$S^* - \{s\}$ is a $(v,k)$-local cut. Otherwise, output $\bot$. By Lemma 3.1.3, there is no $(v,k)$-local cut that contains $x$.

**Running Time.** It remains to show that we can compute $S^*$ deterministically in $\tilde{O}(v^2 k)$ time. We will explain the main idea for the implementation in which we follow the argument from Orecchia and Zhu [38].

**Lemma 3.1.5.** Given vertex $s$ as an initial vertex, we can deterministically compute an $(s,t)$-mincut in the absorbing graph $G'$ in $O(v^2 k \log(vk))$ time.

**Remarks on the proof of Lemma 3.1.5.** We argue that the absorbing graph has a special property that allows us to implement Dinic’s blocking flow algorithm in local time. The key intuition is the physical process of how water flows in the absorbing graph. We start by injecting the water from $s$ into $x$. It will continue to flow through the neighborhood of $x$ whenever the edge from $x$ to $t$ is saturated (meaning the capacity is full). A vertex $b$ in $G$ is saturated if the edge $(b,t)$ has been used at full capacity. Let $B$ be the set of the saturated set at the current time step. The rule is that the watering process will only saturate the neighbors of $B$. Over time, the flow will saturate the neighborhood of $B$ and so on. The flow will stop after injecting at most $v k + 1$ unit of flow (because $(s,x)$ is a cut-set with capacity $v k + 1$).

A key challenge is to implement this process without reading the entire graph. We briefly argue that the process can be simulated “locally” in a sense that the next vertex set to be saturated only depends on the current saturated set. Initially, the saturated set is only $\{x\}$. What is the biggest size of the saturated set in this process? Let $B$ be the set of saturated nodes. By design, every node $b \in B$ absorbs $\deg_{G}(b)$ unit of flow. Collectively, $B$ have absorbed $\sum_{b \in B} \deg_{G}(b) = \text{vol}_{G}(B)$ unit of flow. Since the total injection is at most $v k + 1$, $B$ will receive at most $v k + 1$ unit of flow, and thus $\text{vol}(B) \leq v k + 1$. Therefore, the size of the saturated set is bounded by $O(vk)$. For the implementation, we carefully implement Dinic’s blocking flow algorithm on the absorbing graph. The key property of blocking flows is we augment the flow along the shortest augmenting paths; this mimics the physical process as stated.

### 3.2 Local Vertex Cut Detection

We start with the definition of LocalVC on a directed graph.

**Definition 3.2.1.** We call an algorithm local vertex cut detection (LocalVC for short) if it takes an initial vertex $x$ in the adjacency list of a directed graph $G$, volume parameter $v$ and cut parameter $k$, and either output a vertex separator of size at most $k - 1$ or $\bot$, certifying that there is no vertex...
set $L$ such that $x \in L$, $|N^{out}(L)| < k$ and $\text{vol}^{out}(L) \leq \nu$. In the randomized algorithm, it is allowed to output $\perp$ incorrectly with probability at most $1/3$. That is, the algorithm outputs either a separator of size at most $k - 1$ or $\perp$; if there is a set $L$ such that $x \in L$, $|N^{out}(L)| < k$ and $\text{vol}^{out}(L) \leq \nu$, then the algorithm may output $\perp$ with probability at most $1/3$.

The main technical lemma in this section is we automatically obtain efficient LocalVC algorithms from efficient LocalEC algorithms for directed graphs. The reduction step uses the standard split graph construction.

**Lemma 3.2.1** (Reduction to LocalEC). *If there is a LocalEC algorithm (on directed graphs) using parameters $\nu$ and $k$ where $\nu = O(m/k)$ with running time $f(\nu, k)$ for some function $f$, then there is a LocalVC algorithm with running time $f(O(\nu), k)$.*

Using the bounds from directed LocalEC algorithms, Lemma 3.2.1 implies $O(\nu^2 k)$-time deterministic, $O(\nu k^2)$-time randomized and $O(\nu k^O(k))$-time randomized LocalVC algorithms. For the rest of the section, we discuss the high-level proof of Lemma 3.2.1. The key ingredient is the classic notion of split graphs.

**Definition 3.2.2.** Given a graph $G$, we define the split graph $H$ of $G$ as follows. For each vertex $v$ in $G$, we add in- and out- vertices $v_{in}$ and $v_{out}$ to $H$ with an edge $(v_{in}, v_{out})$ called split edge. For each edge $(u, v)$ in $G$, we add the edge $(u_{out}, v_{in})$ to $H$.

By construction, for any $a, b \in V(G)$, there is a bijection between the set of $(a, b)$-separators in $G$ and the set of $(a_{out}, b_{in})$-cutsets in $H$ consisting of only split edges. Let $h$ be the function that takes a vertex $v$ in $G$ and outputs the corresponding split edge $(v_{in}, v_{out})$ in $G'$. Thus, $S$ is an $(a, b)$-separator in $G$ if and only if the cutset $\bigcup_{s \in S} \{h(s)\}$ is an $(a_{out}, b_{in})$-cut in $H$. Also, $|E(H)| = O(|E(G)|)$.

We are now ready to describe the reduction.

**Inputs.** We are given initial vertex $x \in V$ and two parameters $\nu$ and $k$, and also we have an access to the LocalEC algorithm with running time $f(\nu, k)$. We implement a LocalVC algorithm as follows.

**Algorithm.** The following algorithm can be implemented in $f(O(\nu), k)$ time.

1. Run LocalEC on the split graph $H$ starting from the initial vertex $x_{out}$ using parameters $(O(\nu), k)$. It is important that we do not construct the entire graph $H$, but rather we compute a partial graph of $H$ on-the-fly starting from $x$.

2. If $\perp$ is returned from the LocalEC execution, then we return $\perp$.

3. Otherwise, a vertex set $L'$ is returned from the LocalEC algorithm.
We show that the algorithm is indeed LocalVC. If there is a Correctness.

output $S$.

(a) We define the set $S$ given $L'$ as follows. Every edge $e \in \delta_H(L')$ is of the form $(u_{in}, u_{out})$ (a split edge) or $(u_{out}, w_{in})$ (a non-split edge). We define $E'$ based on $\delta_H(L')$ as follows. Initially, $E'$ consists of every split edge in $\delta_H(L')$ (of the form $(u_{in}, u_{out})$). For each edge $e \in \delta_H(L')$ of the form $(u_{out}, w_{in})$, we add $(w_{in}, w_{out})$ to $E'$. By design, $E'$ consists of only split edges. Since $vol(H, \delta_H(L')) \leq f(O(v), k) \ll m$, $E'$ is an $(x_{out}, t_{in})$-cutset for some $t_{in}$. Let $g$ be the function that takes a split edge of the matching in- and out-vertices in $H$ and outputs the corresponding vertex in $G$, i.e., $g(v_{in}, v_{out}) = v$. Let $S = \bigcup_{(w_{in}, w_{out}) \in E'} \{g(w_{in}, w_{out})\}$. See Figure 3.1 for illustration.

(b) We argue in the correctness proof that $S$ is an $(x, t)$-separator in $G$.

4. Return $S$.

**Correctness.** We show that the algorithm is indeed LocalVC. If there is a vertex set $L$ in $G$ such that $x \in L$ and $vol_G(L) \leq v, |N_G(L)| < k$, then there is a set $L'$ in split graph $H$ that is $(O(v), k)$-local w.r.t. $x$ in the split graph $H$. Therefore, the LocalEC algorithm on $H$ will output $L'$ with probability at most $1/3$. On the other hand, if a vertex set $S'$ is returned at the final step, then $S'$ is an $(x, t)$-separator in $G$ for some vertex $t$ by the bijection between $(x, t)$-separator in $G$ and $(x_{out}, t_{in})$-cutsets (of only split edges) in $H$. Furthermore, $|S'| = |E'| = |\delta_H(S)| \ll k - 1$ because of the guarantee about the output $S$ from the LocalEC algorithm on the split graph $H$. 

\[ \]
3.3 Vertex Connectivity Algorithm

We are now ready to present the $\tilde{O}(mk^2)$-time vertex connectivity algorithm in Section 3.3.1. This implies $\tilde{O}(m+nk^3)$-time vertex connectivity algorithm for undirected graphs using the edge-sparsification algorithm in Theorem 2.1.1.

3.3.1 An $\tilde{O}(mk^2)$-Time Vertex Connectivity Algorithm

We prove that we can decide $k$-vertex connectivity in $\tilde{O}(mk^2)$ time. We assume WLOG that $k \leq \sqrt{n}/10$; otherwise, the algorithm takes longer than $O(mn)$, which is the running time of the previous algorithms.

We say that a vertex cut $(L, S, R)$ is unbalanced if $\text{vol}_{G}^{\text{out}}(L) \leq m/100k$ or $\text{vol}_{G}^{\text{in}}(R) \leq m/100k$. Otherwise, we say that the cut is balanced. We present two algorithms that find unbalanced and balanced vertex cuts efficiently.

**Lemma 3.3.1** (Detecting an unbalanced vertex cut). There is an algorithm that takes as inputs a directed graph $G$, a parameter $k$, and outputs a $(L, S, R)$. If $G$ contains an unbalanced vertex cut, then $|S| < k$ with high probability. The algorithm runs in $\tilde{O}(mk^2)$ time.

**Lemma 3.3.2** (Detecting a balanced vertex cut). There is an algorithm that takes as inputs a directed graph $G$, a parameter $k$, and outputs a $(L, S, R)$. If $G$ contains a balanced vertex cut, then $|S| < k$ with high probability. The algorithm runs in $\tilde{O}(mk^2)$ time.

We prove Lemma 3.3.1 and Lemma 3.3.2 in Section 3.3.2 and Section 3.3.3, respectively. Since a vertex cut of size $< k$ is either unbalanced or balanced, we can decide $k$-vertex connectivity in $\tilde{O}(mk^2)$ time.

3.3.2 Algorithm for Detecting Unbalanced Vertex Cuts

We will use the $O(\nu k^2)$-time LocalVC algorithm described in Section 3.2. WLOG, we assume that LocalVC is correct with high probability (this can be done by repeating LocalVC calls for $\Theta(\log n)$ times), and also we assume that the LocalVC algorithm requires that the parameters $\nu$ and $k$ satisfy $\nu k < m/100$; the constant on $m$ can change depending on the specific implementation of the algorithm.

To prove Lemma 3.3.1, we present an algorithm and analysis.

**Algorithm.** For all positive integer $i \leq \log(m/100k)$, define $\nu_i = 2^i$. We run the following for all $i$. We first sample for $\frac{m}{\nu_i} \Theta(\log n)$ random edges. For each edge $(u, v)$ in the samples, run LocalVC on $u$ with volume and cut parameters $\nu_i$ and $k$ respectively. If one of the LocalVC executions outputs a vertex set $L$ then we output $N(L)$. We repeat the same algorithm for the reverse graph $G^R$ of $G$ where $G^R$ is the graph $G$ after reversing all edges. If
LocalVC executions always output ⊥, then we return \( N^{\text{out}}(v_{\text{min}}) \) where \( v_{\text{min}} \) is a min-out-degree vertex.

**Running Time.** For each \( i \), we have \( \frac{m}{\nu} \log n \) calls of LocalVC with parameters \((\nu_i, k)\). This takes \( \frac{m}{\nu} \log n \cdot \nu_i k^2 \log n = O(\log^2 n \cdot m k^2) \) total for each \( i \). Since \( i \leq \log \frac{m}{100k} \), total running time is \( O(\log m) \cdot O(\log^2 n \cdot m k^2) = \tilde{O}(mk^2) \).

**Correctness.** The algorithm always returns a vertex cut. We prove that if \( G \) contains an unbalanced vertex cut, then the algorithm outputs a vertex cut of size \(< k \) with high probability.

**Lemma 3.3.3.** If there is a vertex cut \((L, S, R)\) in \( G \) of size \(< k \) where \( \text{vol}_L^\text{out}(L) \leq \frac{m}{100k} \) or \( \text{vol}_R^\text{in}(R) \leq \frac{m}{100k} \), then the algorithm for unbalanced vertex cut outputs a vertex cut of size \(< k \) with high probability.

**Proof.** We first prove for the case \( \text{vol}_L^\text{out}(L) \leq \frac{m}{100k} \). It is enough to prove that there is one execution of LocalVC starting on some vertex in \( L \) using the volume parameter at least \( \text{vol}_L^\text{out}(L) \) and the same cut parameter \( k \). If this happens, then the execution of LocalVC (Definition 3.2.1) will output a separator of size \(< k \) with high probability.

We now show that such execution of LocalVC with the correct volume parameter occurs with high probability. Denote \( \nu = \text{vol}_L^\text{out}(L) \). Since we binary search on volume \( \nu_i \) up to \( \frac{m}{100k} \) and \( \nu \leq \frac{m}{100k} \), there is \( i^* \) such that \( \frac{\nu_i}{2} \leq \nu \leq \nu_{i^*} \). Let \( E_{i^*} \) be the set of sampled random edges at \( i^* \)-th iteration. Denote \( \tilde{\nu} = \nu_{i^*} \), and so \( \tilde{\nu}/2 \leq \nu \leq \tilde{\nu} \). First observe that \( \tilde{\nu} \geq \nu \), meaning that \( \tilde{\nu} \) is a correct volume parameter for running LocalVC since \( \tilde{\nu} \geq \nu = \text{vol}_L^\text{out}(L) \). It remains to prove that among the sample edges \( E_{i^*} \) there is an edge \((u, v)\) such that \( u \in L \) with high probability. If true, then we have the execution of LocalVC starting on \( u \in L \) with volume parameter \( \tilde{\nu} \geq \text{vol}_L^\text{out}(L) \) since every LocalVC execution on \( i^* \)-th iteration uses volume parameter \( \tilde{\nu} \). We now prove the claim. The probability that a random edge \((u, v)\) belongs to the set \( E_G(L, V) \) is \( \text{vol}_L^\text{out}(L) / m = \nu/m \). Thus, the probability that none of the sampled edges in \( E_{i^*} \) belongs to the set \( E_G(L, V) \) is

\[
(1 - \nu/m)^{|E_{i^*}|} = (1 - \nu/m)^{\Theta(\log n)} \leq (1 - \nu/m)^{\Theta(\log n)} \leq 1/n^{\Theta(1)}
\]

Therefore, there is an edge \((u, v) \in E_{i^*} \) such that \( u \in L \) with high probability. To handle to case \( \text{vol}_R^\text{in}(R) \leq \frac{m}{100k} \), observe that \((R, S, L)\) is a vertex cut of size \(< k \) in the reversed graph \( G^R \) and \( \text{vol}_L^\text{out}(L) = \text{vol}_R^\text{in}(R) \leq \frac{m}{100k} \). Thus, the same analysis for the same algorithm in the reversed graph goes through.

**3.3.3 Algorithm for Detecting Balanced Vertex Cuts**

An unbalanced vertex cut \((L, S, R)\) has \( \text{vol}_L^\text{out}(L) \geq \frac{m}{(100k)} \) and \( \text{vol}_R^\text{in}(R) \geq \frac{m}{(100k)} \). Observe there could be an edge from \( R \) to \( L \) since the graph is
So we can sample a pair of random edges $u, v$ of vertices such that $u \in L$ and $v \in R$. If we manage to obtain such a pair, then we can run the Ford-Fulkerson max-flow algorithm to obtain a vertex cut of size $< k$ in $O(mk)$ time.

We prove Lemma 3.3.2 by showing an algorithm and its analysis.

**Algorithm.** Sample a pair of random edges $(u_1, v_1)$ and $(u_2, v_2)$, and define $\Pi$ to be the set of all pairs of vertices from the set $\{u_1, v_1, u_2, v_2\}$. For all $(a, b) \in \Pi$ ($(a, b)$ does not need to be an edge), we run Fold-Fulkerson max-flow to decide if there is an $(a, b)$-separator of size $< k$ or not. We repeat for $\Theta(k \log m)$ iterations. We return the minimum separator found so far. If we do not find any vertex cut, then return the min-degree vertex cut.

**Running Time.** The total time is $O(mk^2)$. The Ford-Fulkerson max-flow algorithm takes $O(mk)$ time to decide if the min $(s, t)$-separator has size at least $k$ or not. The number of calls is $\Theta(k \log m)$ because we run $\Theta(k \log m)$ iterations and for each iteration, we call 24 times (there are 24 pairs from the set $\Pi$).

**Correctness.** The algorithm outputs a vertex cut. We prove that if $G$ contains a balanced vertex cut, then the algorithm returns a separator of size $< k$ with high probability.

**Lemma 3.3.4.** If there is a vertex cut $(L, S, R)$ of size $< k$ where $\text{vol}_\text{out}(L) \geq m/(100k)$ and $\text{vol}_\text{in}(R) \geq m/(100k)$, then the algorithm outputs a vertex cut of size $< k$ with high probability.

**Proof.** We prove that with high probability there is a pair of vertices $(u, v)$ where $u \in L$ and $v \in R$ that we run the Fold-Fulkerson max-flow algorithm to decide if a min $(u, v)$-separator is at least $k$ or not.

As a warmup, let $e_1, e_2$ be two random edges. If we do not optimize for $k$, then the probability that $e_1 \in E(L, V)$ and $e_2 \in E(V, R)$ is $\frac{\text{vol}_\text{out}(L)}{m} \cdot \frac{\text{vol}_\text{in}(R)}{m} \geq \Omega(1/k^2)$. This means it is enough to sample for $\Theta(k^2 \log n)$ pairs of random edges.

The rest of the proof is to show that it is enough to sample for $\tilde{\Theta}(k)$ pairs of random edges. The set of directed edges $E$ can be partitioned into

$$E = E(L, V) \cup E(S, S) \cup E(V, R) \cup E(R, L),$$

and thus

$$|E| = \text{vol}_\text{out}(L) + |E(S, S)| + \text{vol}_\text{in}(R) + |E(R, L)|
\leq 2 \max(\text{vol}_\text{out}(L), \text{vol}_\text{in}(R)) + |E(S, S)| + |E(R, L)|.$$

Therefore,

$$\max(\text{vol}_\text{out}(L), \text{vol}_\text{in}(R)) + |E(R, L)| \geq \frac{|E| - |E(S, S)|}{2} \geq \frac{m - k^2}{2} \geq m/10. \quad (3.1)$$
The last inequality follows since \( k \leq \sqrt{n}/10 \leq \sqrt{m}/10 \). We remark that the set \( E(R,L) \) is empty if the graph is undirected.

Let \( e_1 \) and \( e_2 \) be two random edges. We have two cases.

- If \( \text{vol}^\text{out}(L) \leq \text{vol}^\text{in}(R) \), then the probability that \( e_1 \in E(L,V) \) and \( e_2 \in E(V,R) \cup E(R,L) \) is

\[
\frac{\text{vol}^\text{out}(L) \cdot \text{vol}^\text{in}(R) + |E(R,L)|}{m} \geq \frac{1}{100k} \cdot \frac{1}{10} = \Omega(1/k).
\]

- If \( \text{vol}^\text{out}(L) \geq \text{vol}^\text{in}(R) \), the probability that \( e_1 \in E(V,R) \) and \( e_2 \in E(L,V) \cup E(R,L) \) is

\[
\frac{\text{vol}^\text{in}(R) \cdot \text{vol}^\text{out}(L) + |E(R,L)|}{m} \geq \frac{1}{100k} \cdot \frac{1}{10} = \Omega(1/k).
\]

Therefore, in any case, after \( \Theta(k \log n) \) samples, we obtain such an event with high probability, and thus there is a pair \((u,v)\) of vertices among \( e_1, e_2 \) such that \( u \in L \) and \( v \in R \) as desired.

\[\square\]

### 3.4 Experimental Results

We implement four algorithms: LOCAL1, LOCAL1+, LOCAL2+, and HRG. LOCAL1 is a basic implementation of the algorithm described in Section 3.3. LOCAL1+ and LOCAL2+ are the optimized version where we apply additional heuristics called degree counting to speed up detecting an unbalanced vertex cut. The algorithms LOCAL1, LOCAL1+, LOCAL2+ are identical except for the implementation of the LocalVC algorithm. HRG is the \( \tilde{O}(mn) \)-time algorithm by Henzinger, Rao and Gabow [24]. The algorithms were implemented with C++17 and compiled with g++.

We ran all experiments on an Ubuntu computer with i7-7700HQ CPU (2.80GHz) and 2x8 GB DDR4-2400 RAM.

In this thesis, we highlight the experimental results on planted cuts in undirected graphs. For undirected graphs, we apply the edge sparsification algorithm by Nagamochi Ibaraki [34] as preprocessing before running each algorithm. We ignore the time to sparsify the graph as it is the same for all algorithms. As a result, the running time on the sparsified graph only depends on \( n \) and \( k \). Therefore, we report the graph size in terms of the number of vertices when \( k \) is fixed.

In Publication V, we conducted extensive experiments on additional datasets including random hyperbolic graphs, and real-world datasets, and also we conducted internal comparisons among LOCAL1, LOCAL1+, and LOCAL2+. We also conduct directed graphs at a large scale.
3.4.1 Planted Cuts

The planted cut datasets are undirected graphs with a vertex mincut \((L, S, R)\) where the sizes of \(L, S, R\) are configurable.

Generating Planted Cuts. We plant the vertex mincut \((L, S, R)\) as follows. Fix a vertex partition \((L, S, R)\) of a vertex set \(V\). Let \(K\) be a complete graph with no edges between \(L\) and \(R\). We run Nagamochi-Ibaraki algorithm [34] on \(K\) to produce a sequence of forests \(F_1, \ldots, F_\ell\) where \(\ell\) is a parameter in the algorithm. The final graph is \(G = (V, E)\) where \(E = \bigcup_{i \leq \ell} F_i\). By the property of greedy forests, for any pair \(x, y\) in graph \(G\), \(\kappa_G(x, y) \geq \ell\) iff \(x\) and \(y\) are separated by \(S\). We use \(\ell = 64\) in our experiments. The algorithm by Nagamochi-Ibaraki is deterministic and based on choosing greedy forests. To obtain diverse datasets, we add randomness in the construction by choosing a random greedy forest instead of an arbitrary greedy forest.

Balanced vs. Unbalanced Cuts. We compare the performance of the four algorithms with respect to the balanceness of the vertex mincut \((L, S, R)\), i.e. when we vary \(|L|\). We use \(n = 10000\) and \(\kappa_G = 4\), and vary \(|L|\). For each algorithm, we show the running time where we normalize by the average running time over all instances of the same algorithm in Figure 3.2. According to Figure 3.2, all four algorithms perform reasonably well with some variations. The key finding is that every algorithm runs faster when the vertex cut is balanced, i.e., when \(|L|\) is large. The difference between the highest and lowest running time is a factor of 1.71, 1.71, 1.79, and 1.89 for LOCAL1, LOCAL1+, LOCAL2+, and HRG, respectively. This suggests that the unbalanced cuts are the difficult case. For most of the experiments, we will focus on unbalanced vertex cuts.

![Figure 3.2](image_url)  
**Figure 3.2.** Undirected Planted Cuts with variable \(|L|\) or \(\kappa\) (Decision version with \(k = \kappa + 1\)).

A vertical line marks \(|L| = 5\) in Figure 3.2a.

Running Time Comparison. We study the practical running times of the four algorithms on undirected graphs with vertex connectivity \(\kappa \in \{4, 8\}\) for up to 16 million edges. Here, we focus on the unbalanced cut as it is the
most interesting (and difficult) case. In this setting, we use planted vertex cut where \(|L| = 5, |S| \in \{4, 8\}\). According to Figure 3.3, LOCAL1, LOCAL1+, and LOCAL2+ outperform HRG even on small graphs. When \(\kappa_G = 8\), HRG is slower than LOCAL1 at approximately 300 vertices with roughly 30 ms in running time. LOCAL1+ and LOCAL2+ are even faster than HRG. In terms of scalability with \(n\), HRG takes about 50 mins at \(n = 50000\) and \(\kappa_G = 8\) whereas LOCAL1 takes 73 seconds. Furthermore, LOCAL1+ and LOCAL2+ take less than 8 seconds. As the number of vertices increases to a million, both LOCAL1+ and LOCAL2+ can still run under an hour. We do not run HRG on larger \(n\) because of the time limit.

The degree counting heuristics are quite effective according to the results. LOCAL1+ and LOCAL2+ are faster than LOCAL1 as we increase the number of vertices. The speedup for LOCAL1+ relative to LOCAL1 is 2.8, 5.7, and 8.6 for \(\kappa = 4\) and \(n = 500,50000\), and 1000000, respectively. The speedup for LOCAL2+ relative to LOCAL1 is 2.6, 5.8, and 9.6 for \(\kappa = 4\) and \(n = 500,50000\), and 1000000, respectively. For \(\kappa = 8\), the speedups are around 1.3-2 times higher with larger speedups on large graphs.

Figure 3.3. Undirected Planted Cuts with fixed \(|L| = 5\) and \(\kappa\)
4. Vertex Connectivity in Poly-logarithmic Max-flows

The goal of this chapter is to prove that we can solve vertex connectivity of undirected graphs in poly-logarithmic max-flows. To state the result more concisely, we define the running time in terms of poly-logarithmic max-flows.

**Definition 4.0.1.** Given an $m$-edge graph, we say that an algorithm runs in poly-logarithmic max-flows if it makes calls to st-max-flow algorithm on unit-vertex-capacity graphs with $\tilde{O}(m)$ total number of vertices and edges and takes $\tilde{O}(m)$ time outside the max-flow calls.

The main result of this chapter is the following.

**Theorem 4.0.1.** There is a randomized (Monte Carlo) algorithm that takes an undirected graph as input and returns a minimum vertex cut in poly-logarithmic max-flows.

This immediately implies almost linear time vertex connectivity algorithm using the recent almost linear time max-flow algorithm [9]. Thus, we affirmatively resolve the open problem by Aho, Hopcroft, and Ullman [1, Problem 5.30] (up to a sub-polynomial factor).

**Theorem 4.0.2.** There is a randomized (Monte Carlo) algorithm that takes an undirected graph and returns a minimum vertex cut in $m^{1+o(1)}$ time.

We prove Theorem 4.0.1 in Section 4.1.

4.1 The Algorithm

We first design an algorithm for deciding $k$-vertex connectivity. In Section 1.2.3, we outline the sublinear kernelization technique to decide $k$-vertex connectivity when $k = \Omega(n)$ in poly-logarithmic max-flows. Specifically, we show the existence of a kernel of size $\tilde{O}(n|L|)$ if we obtain a vertex $x \in L$ where $(L, S, R)$ is a vertex mincut. Since we sample $\tilde{O}(n/|L|)$ many vertices to obtain $x \in L$, the total kernel size is $\tilde{O}(n/|L|) \cdot \tilde{O}(n|L|) = \tilde{O}(n^2) = \tilde{O}(m)$.
However, for general $k$, we need a smaller kernel of size $\tilde{O}(k|L|)$ instead of $\tilde{O}(n|L|)$ so that the total size is $\tilde{O}(n/|L|) \cdot \tilde{O}(k|L|) = \tilde{O}(nk) = \tilde{O}(m)$ (using the same algorithm). Recall that WLOG the minimum degree is at least $k$, which means $m \geq nk$.

It turns out that the same kernelization algorithm generates a kernel of size $\tilde{O}(k|L|)$ with high probability if $(L, S, R)$ is unbalanced and contains many of low degree vertices in $S$. More precisely, the same algorithm gives a kernel of size $\tilde{O}(k|L|)$ if $|S| < k/\log(n)$ and $|S_{\text{low}}| \geq 300|L|\log n$, where $S_{\text{low}} = \{v \in S : \deg_G(v) \leq 8k\}$. Such a vertex cut is called $k$-scratch.

**Definition 4.1.1.** We say that a vertex cut $(L, S, R)$ is $k$-scratch if $|S| < k$, $|L| \leq k/(100\log n)$ and $|S_{\text{low}}| \geq 300|L|\log n$.

Thus, by using sublinear kernelization, we can handle the case where $G$ contains a $k$-scratch. We prove the following lemma in Section 4.2.

**Lemma 4.1.1** (Using Sublinear Kernelization). There is an algorithm that takes a graph $G = (V, E)$, and parameter $k$, and outputs a separator $S$ in $G$. If $G$ contains a $k$-scratch, then $|S| < k$ with high probability. The algorithm runs in poly-logarithmic max-flows.

If the vertex mincut (of size $< k$) is not $k$-scratch, we call it $k$-non-scratch.

**Definition 4.1.2.** A vertex cut $(L, S, R)$ is $k$-non-scratch if $|S| < k$ but $|L| > k/(100\log n)$ or $|S_{\text{low}}| < 300|L|\log n$.

To handle the remaining case, we develop a new tool called isolating vertex cut lemma by adapting the proof for the isolating cut lemma for the edge cut version [30].

**Lemma 4.1.2** (Isolating Vertex Cut Lemma). Given a graph $G = (V, E)$, and an independent set $I \subseteq V$ of size at least 2 as inputs, there is an algorithm that outputs, for each vertex $v \in I$, a $(v, I \setminus \{v\})$-min separator. The algorithm makes $st$-maxflow calls on unit-vertex-capacity graphs with $O(m \log |I|)$ total number of vertices and edges, and takes $O(m)$ time outside of maxflow calls.

We use Lemma 4.1.2 to handle a $k$-non-scratch vertex cut. In Section 4.3, we prove the following.

**Lemma 4.1.3** (Using Isolating Vertex Cut Lemma). There is an algorithm that takes a graph $G = (V, E)$, and parameter $k$ where $m \leq nk$, and outputs a separator $S$ in $G$. If $G$ contains a $k$-non-scratch, then $|S| < k$ with high probability. The algorithm runs in poly-logarithmic max-flows.

We are now ready to prove Theorem 4.0.1.

**Proof of Theorem 4.0.1.** By binary search on $k$, it is enough to decide $k$-vertex connectivity. We now present the algorithm for deciding $k$-vertex connectivity.
Algorithm. First, we apply Nagamochi-Ibaraki’s algorithm (Theorem 2.1.1) to sparsify the graph so that $m \leq nk$. Let $S_1$ and $S_2$ be separators obtained from the algorithms in Lemma 4.1.1, and Lemma 4.1.3, respectively. If $\min\{|S_1|, |S_2|\} < k$, then output the smallest of the two separators. Otherwise, declare that the input graph is $k$-vertex connected.

Running Time. Nagamochi-Ibaraki’s algorithm (Theorem 2.1.1) runs in $O(m)$ time. The algorithms in Lemma 4.1.1 and Lemma 4.1.3 run in poly-logarithmic max-flows.

Correctness. The algorithm always outputs a separator. If $G$ is $k$-vertex connected, then the separator has size at least $k$. Otherwise, $G$ contains either a $k$-scratch or $k$-non-scratch vertex cut. Therefore, Lemma 4.1.1, and Lemma 4.1.3 imply that $\min\{|S_1|, |S_2|\} < k$ with high probability.

4.2 Using Sublinear Kernelization

We prove Lemma 4.1.1 in this section. In Section 4.2.1, we show that the same kernelization algorithm as described in Section 1.2.3 yields a smaller kernel of size $\tilde{O}(k|L|)$, rather than $\tilde{O}(n|L|)$ provided that $G$ contains a $k$-scratch. In Section 4.2.2, we formally prove the correctness of the kernel construction via reduction rules. Finally, we prove Lemma 4.1.1 by providing the full algorithm in Section 4.2.3.

4.2.1 Sublinear Kernelization

In this section, we show that the same algorithm described in Section 1.2.3 yields a kernel of size $O(k|L|)$ if $G$ contains a $k$-scratch vertex cut $(L, S, R)$. More precisely, assuming that $G$ contains a $k$-scratch vertex cut $(L, S, R)$, and we are given $x \in L^1$, we describe an algorithm that outputs a graph $H$ of $O(k|L|)$ edges and a vertex $y$ such that we can extract vertex mincut of $G$ from an $(x, y)$-max-flow in $H$. We assume WLOG that the minimum degree is at least $k$. Otherwise, we can output a vertex cut from the minimum degree vertex.

The algorithm is the same as in Section 1.2.3. Namely, define $T$ as the set of vertices where each vertex in $G$ is selected with probability $1/|L|$. Let $T_x = T - N_G[x]$. Observe that $|L \cup S| \leq k + |L|$. Since $N(x) \subseteq L \cup S$, and $|N(x)| \geq k$, we conclude that

$$|L \cup S| - N_G[x] \leq |L|. \tag{4.1}$$

This implies the following lemma.

Lemma 4.2.1. $\emptyset \neq T_x \subseteq R$ with probability $\Omega(1)$.

$^1$We can obtain a vertex $x \in L$ with high probability by sampling $\tilde{O}(n/|L|)$ vertices.
Proof. Observe that $\emptyset \neq T_x \subseteq R$ if and only if none of the vertices in $(L \cup S) - N_G[x]$ is sampled and at least one vertex in $R$ is sampled. Since the two events are independent, it is enough to prove them separately. Since $|R| \geq |L|$, the probability that at least one vertex in $R$ is sampled is

$$1 - (1 - 1/|L|)^{|R|} \geq 1 - (1 - 1/|L|)^{|L|} = \Omega(1)^2.$$  

By Equation 4.1, we have $|(L \cup S) - N_G[x]| \leq |L|$. Therefore, the probability that none of the vertices in $(L \cup S) - N_G[x]$ is sampled is

$$\left(1 - 1/|L|\right)^{|(L \cup S) - N_G[x]|} \geq 1 - (1 - 1/|L|)^{|L|} = \Omega(1).$$

This completes the proof. \hfill \Box

By Lemma 4.2.1, $\emptyset \neq T_x \subseteq R$ with probability $\Omega(1)$. Thus, by repeating $\Theta(\log n)$ times, there is one trial where $\emptyset \neq T_x \subseteq R$ with high probability.

Now, we assume $x \in L$ and $T_x \subseteq R$. We now describe the reduction steps.

- **Reduction 0.** Contract $T_x$ into $t_x$.
- **Reduction 1.** Remove all edges between the neighbors of $x$ and neighbors of $t_x$.
- **Reduction 2.** Remove $N(x) \cap N(t_x)$ all common neighbors between $x$ and $t_x$. Denote $Z = N(x) \cap N(t_x)$.
- **Reduction 3** Remove degree-1 neighbors of $t_x$.

**Correctness.** Let $H$ be the resulting graph after applying all the reductions. Then, $H$ is a kernel of $G$ in the following sense. Let $C$ be an min $(x, t_x)$-separator in $H$. By the correctness of the reduction rules (Section 4.2.2), $C \cup Z$ is an $(x, T_x)$-separator in $G$. Furthermore, since $x \in L$ and $T_x \subseteq R$, $C \cup Z$ is a min separator in $G$. We can compute $C$ by solving $(x, t_x)$-max-flow on $H$, a graph with unit vertex capacity.

**Kernel size.** We prove that the number of edges of $H$ is $\tilde{O}(k|L|)$. The kernel $H$ has $x$ and $t_x$ as two distinguished vertices. The set of vertices in $H$ can be classify into $N_x := N_H(x)$, $N_t := N_H(t_x)$ and $F$, which is the remaining vertices. The set $Z$ in Reduction 2 can be expressed as $Z = N_G(x) \cap N_G(T_x)$. By design, we can write $N_x, N_t$ in terms of the set of vertices in $G$. Namely, $N_x = N_G(x) \setminus Z$ and $N_t = N_G(T_x) \setminus Z$. In other words,

$$V(H) = \{x, t_x\} \cup N_x \cup N_t \cup F \quad (4.2)$$

See Figure 1.1 for illustration. Note that the classification of the vertices in $H$ is the same as that in Section 1.2.3.

2Note that $|L| \geq 2$ since the minimum degree is at least $k$. 

46
We now bound the size of $H$. We start with a simple observation. Since $x \in L$ and $(L, S, R)$ is $k$-scratch, we have

$$|N_x| \leq |N_G(x)| \leq k + |L| \leq 2k. \quad (4.3)$$

For any $v$, denote $\deg_{G - N[x]}(v) = |N_G(v) - N_G[x]|$ be the degree of $v$ excluding $N_G[x]$. The following lemma says that we can charge the number of edges in $H$ to the vertices in $N_x \cup F$.

**Lemma 4.2.2.**

$$|E(H)| \leq \sum_{v \in N_x \cup F} 2 \cdot \deg_{G - N[x]}(v) + 2k.$$

**Proof.** By Lemma 1.2.1, we have

$$|E(H)| \leq \sum_{v \in N_x \cup F} \deg_{G - N[x]}(v) + |N_x| + |N_t| \leq \sum_{v \in N_x \cup F} \deg_{G - N[x]}(v) + 2k + |N_t| \leq \sum_{v \in N_x \cup F} 2 \cdot \deg_{G - N[x]}(v) + 2k.$$

The last inequality follows since every vertex in $N_t$ must have an edge from $N_x \cup F$ in $H$. \hfill \Box

By Lemma 1.2.2, every vertex in $N_x \cup F$ has low degree (excluding $N[x]$) with high probability. That is, with high probability,

for all $v \in N_x \cup F$, $\deg_{G - N[x]}(v) \leq 10|L| \log n.$ \quad (4.4)

Since every vertex in $N_x \cup F$ has degree $|L| \log n$ (outside of $N_G(x)$), it remains to show that $|F| = O(k)$. If true, then Lemma 4.2.2 implies that

$$|E(H)| \leq \sum_{v \in N_x \cup F} 2 \cdot \deg_{G - N[x]}(v) + 2k \leq 2 |N_x \cup F| \cdot 10 |L| \log n + 2k \overset{(4.3)}{=} \tilde{O}(k|L|).$$

We now prove the final claim.

**Lemma 4.2.3.** If Equation 4.4 holds, then $|F| = O(k)$.

**Proof.** We show that every vertex $v \in F$ has at least $\Omega(|S_{low}|)$ neighbors in $S_{low}$ where $S_{low} = \{ v \in S: \deg_G(v) \leq 8k \}$. If true, then

$$8k |S_{low}| \geq |E_G(S_{low}, V)| \geq \Omega(|S_{low}||F|).$$

Therefore, $|F| = O(k)$ as desired. Fix $v \in F$. For any vertex set $A$, define $\deg_G(v, A) = |E_G(v, A)|$ to be the set of edges from $v$ to $A$. Since $\deg_G(v) \geq k$
and \( \deg_{G-N[x]}(v) \leq 10L|\log n \), the vertex \( v \) has \( k - 10L|\log n \) edges incident to \( N_G(x) \) \(^3\).

\[
\deg_G(v, N_G(x)) \geq k - 10L|\log n. \tag{4.5}
\]

Define \( N_{\text{low}} := N_G(x) \cap S_{\text{low}}. \) By Equation 4.1, all but \( |L| \) vertices in \( S_{\text{low}} \) are in \( N_G(x) \). Indeed,

\[
|S_{\text{low}} - N_G(x)| \leq |S - N_G(x)| \leq |(L \cup S) - N_G(x)| \leq |L|. \tag{4.1}
\]

Hence, we have

\[
|N_{\text{low}}| \geq |S_{\text{low}}| - |L|. \tag{4.6}
\]

We are now ready to bound the number of edges from \( v \) to \( S_{\text{low}} \). That is,

\[
\deg_G(v, S_{\text{low}}) \geq \deg_G(v, N_{\text{low}})
\]

\[
= \deg_G(v, N_G(x)) - |N_G(x) - N_{\text{low}}|
\]

\[
= \deg_G(v, N_G(x)) - |N_G(x)| + |N_{\text{low}}|
\]

\[
(4.3),(4.5),(4.6) \geq (k - 10L|\log n| - (k + |L|) + (|S_{\text{low}}| - |L|)
\]

\[
= |S_{\text{low}}| - 10L|\log n - 2|L|
\]

\[
\geq |S_{\text{low}}|/2
\]

The second equality follows because \( N_{\text{low}} \subseteq N_G(x) \). The last inequality follows since \( |S_{\text{low}}| \geq 300|L|\log n \) by definition of \( k \)-scratch vertex cut (Definition 4.1.1). This is the place where we crucially need \( S_{\text{low}} \) to be large in order to bound the size of the kernel to be \( \tilde{O}(k|L|) \). \( \square \)

### 4.2.2 Reduction Rules

We describe the generic reduction rules that reduce the instance of \((s,t)\)-vertex mincut problem. We fix a graph \( H = (V,E) \), and a source vertex \( s \in V \), and a sink \( t \in V \) throughout this section.

**Proposition 4.2.4 (Identify Rule).** Let \( Z = N_H(s) \cap N_H(t) \), and \( H' = H - Z \). We have that \( C \subseteq V(H) \) is a min \((s,t)\)-separator in \( H' \) if and only if \( C \cup Z \) is a min \((s,t)\)-separator in \( H \).  

**Proof.** Fix \( v \in N_H(s) \cap N_H(t) \). For any \((s,t)\)-separator \( S \), \( v \in S \) since \( v \in N_H(s) \cap N_H(t) \). Therefore, \( C \) is a min \((s,t)\)-separator in \( H - v \) if and only if \( C \cup \{v\} \) is a min \((s,t)\)-separator in \( H \). The result follows by repeating the same argument for all \( v \in Z \). \( \square \)

\(^3\)Observe that \( k - 10|L|\log n > 0 \) since \((L,S,R)\) is \( k \)-scratch.
The next rule describes how to get rid of “useless” edges and vertices in the graph without affecting the \((s, t)\)-vertex mincut.

**Proposition 4.2.5** (Filter Rule). There is a maximum set of \((s, t)\)-vertex disjoint paths \(\mathcal{P}\) such that every path in \(\mathcal{P}\) does not contain any of the following vertices and edges:

1. an edge with both endpoints in \(N(s)\) or both in \(N(t)\),
2. a vertex \(v\) where \(t \in N(v) \subseteq N[t]\), and
3. a vertex \(v\) where \(s\) cannot reach \(v\) in \(H - N[t]\).

Therefore, by max-flow min-cut theorem, the size of \((s, t)\)-vertex mincut in the graph \(H\) after removing vertices and edges according to (1), (2), and (3) stays the same.

**Proof.** Let \(\mathcal{P}\) be a maximum set of \((s, t)\)-vertex (internally) disjoint paths. We describe the process for modifying every path in \(\mathcal{P}\) to satisfy the desired properties while they are pairwise \((s, t)\)-vertex disjoint.

1. If there is a path \(P \in \mathcal{P}\) using both endpoints in \(N(s)\), then \(P\) is of the form \((s, \ldots, u, v, \ldots, t)\) where \((u, v) \in E_H(N(s), N(s))\). We can replace \(P\) with \(P'\) where we take the shortcut from \(s\) to \(v\) directly and use the same remaining suffix of \(P\) starting from \(v\); that is \(P' = (s, v, \ldots, t)\). Since we strictly shortcut \(P\) \((P'\) is a subpath of \(P\), \(P'\) does not intersect any other path in \(\mathcal{P}\). The argument for \(N(t)\) is similar.

2. Consider \(v\) such that \(t \in N(v) \subseteq N[t]\). After applying (1) in the previous step, \(N(v) = \{t\}\). Thus, there is no simple path from \(s\) to \(t\) through \(v\).

3. If \(P \in \mathcal{P}\) contains a vertex \(v\) where \(s\) cannot reach in \(H - N[t]\), then \(P\) is of the form \((s, \ldots, t', \ldots, v, \ldots, t)\) for some \(t' \in N(t)\). We can replace \(P\) with \(P' = (s, \ldots, t', t)\), which does not intersect any other path in \(\mathcal{P}\).

\[\Box\]

### 4.2.3 An Algorithm for Detecting \(k\)-Scratch Vertex Cuts

Let us first recap what we learned so far. As described in Section 4.2.1, if \(G\) contains a \(k\)-scratch \((L, S, R)\), and we are given \(x \in L\), then we can sample a vertex set \(T\) so that \(T_x := T - N_G[x] \subseteq R\) with high probability. Furthermore, the sampling procedure and the fact that \(x \in L\) and \(T_x \subseteq R\) allow us to apply the reduction rules to obtain a kernel \(G_{x,T}\) of size \(\mathcal{O}(k|L|)\) with high probability. By the correctness of reduction rules in Section 4.2.2, we can extract vertex mincut in \(G\) by solving an \((x, t_x)\)-max-flow for the kernel \(G_{x,T}\).

The next challenge is efficient kernel construction. That is, we need to design a data structure that can construct such a kernel in \(\mathcal{O}(k|L|)\) time,
which is sublinear in $n$. We can formulate the data structure problem as follows. After sampling the vertex $T$ where we select each vertex $v \in V$ with probability $1/|L|$, we can preprocess in $\tilde{O}(m)$ time so that given a vertex $x$ as a query, we output a kernel $G_{x,T}$ that preserves a min $(x,T_x)$-separator in $G$ in $\tilde{O}(k|L|)$ time.

Indeed, we design a data structure that operates on the random vertex set $T$ and an estimate $\ell$ of $|L|$ as a preprocessing step, and supports the operation $\text{KERNELIZE}(x)$ for any vertex $x$ that constructs a kernel $G_{x,T}$ in $\tilde{O}(k\ell)$ time or certifies useful properties about a $k$-scratch. More precisely, we have the following.

**Lemma 4.2.6.** We are given a graph $G = (V,E)$ with minimum degree $\geq k$, and an integer $\ell \leq k/(100\log n)$. Let $T$ be a vertex set where we select each vertex $v \in V$ with probability $1/\ell$. We can construct a data structure $\mathcal{D}_{\ell,T}$ that supports the following operation with high probability:

- **KERNELIZE**($x \in V$): Let $T_x := T - N_G[x]$. The operation takes $\tilde{O}(k\ell)$ time, and either
  - outputs a kernel $G_{x,T}$, containing $x$ and $t_x$, of size $\tilde{O}(k\ell)$ edges and a vertex set $Z_{x,T}$ such that a vertex set $C$ is a min-$(x,t_x)$ separator in $G_{x,T}$ if and only if $C \cup Z_{x,T}$ is a min $(x,T_x)$-separator in $G$ or
  - certifies that either $T_x = \emptyset$ or there is no $k$-scratch $(L,S,R)$ where $x \in L, \ell \in [|L|/2,|L|]$ and $T_x \subseteq R$. \hspace{1cm} (4.7)

The construction takes $\tilde{O}(m)$ time.

We use linear sketching and sparse recovery to implement the data structure in Lemma 4.2.6, which we will discuss the proof in Section 4.2.4. We are now ready to show an algorithm that proves Lemma 4.1.1. We describe an algorithm for detecting $k$-scratch separator and analysis.

**Algorithm.** The inputs are a graph $G = (V,E)$ and a parameter $k$.

1. Guess an integer $\ell$. We try all the guesses $\ell \leftarrow 2^i$ for each integer $i$ as long as $2^i \leq k/100\log n$.

2. Let $\mathcal{T} = \{T_1, \ldots, T_{\Theta(\log n)}\}$ where, for each $i$, $T_i$ selects each vertex in $V$ with probability $1/\ell$.

3. For each $T \in \mathcal{T}$:
   (a) Construct the data structure $\mathcal{D}_{\ell,T}$ defined in Lemma 4.2.6.
   (b) For each $x \in X$ where $X$ is a set of $\Theta(\frac{n}{\ell} \log n)$ random vertices,
      i. Call $\mathcal{D}_{\ell,T}.\text{KERNELIZE}(x)$.
ii. If the kernel $G_{x,T}$ containing a vertex $t_x$ and $Z_{x,T}$ are returned, then obtain $C \cup Z_{x,T}$ where $C$ is a min $(x,t_x)$-separator in $G_{x,T}$, which can be computed by an $(x,t_x)$-max-flow in $G_{x,T}$. By Lemma 4.2.6, $C \cup Z_{x,T}$ a min $(x,T_x)$-separator in $G$.

4. Return the smallest separator found so far or $N(v)$ where $v$ is a min-degree vertex.

**Correctness.** The algorithm always returns some separator in $G$. Now, assume that $G$ contains a $k$-scratch $(L,S,R)$. Since we binary search on the guess $\ell$. There is $\ell \in [\lfloor L/2 \rfloor, \lfloor L \rfloor]$. Since we try $\Theta(n \log n/\ell) = \tilde{\Theta}(n/|L|)$ random vertices in step 3(b), there is $x \in L$ from the samples with high probability. Given $x \in L$, Lemma 4.2.1 implies that $\emptyset \neq T_x \subseteq R$ with probability $\Omega(1)$ for any $T$ where we select each vertex into $T$ with probability $1/\ell = \Theta(1/|L|)^4$. Since $|\mathcal{T}| = \Theta(n \log n)$, there is $T \in \mathcal{T}$ such that $\emptyset \neq T_x \subseteq R$ with high probability. Therefore, a min $(x,T_x)$-separator in $G$ must have size $< k$, which will be obtained at step 3(b)ii by Lemma 4.2.6.

**Running Time.** For each $\ell$ in Step 1, we prove that the algorithm generates max-flow instances of size $\tilde{O}(m)$ in total, and runs in $\tilde{O}(m)$ time outside the max-flow calls. Since we run on $O(\log n)$ values of $\ell$, the total running time and max-flow sizes including all $\ell$'s is still $\tilde{O}(m)$.

Now, fix an $\ell$ in Step 1. We first argue the running time outside the max-flow calls. At step 2, the set $\mathcal{T}$ can be constructed in $O(n \log n)$ time. We next bound the time for step 3. Fix $T \in \mathcal{T}$. Step (3a) takes $\tilde{O}(m)$ time to preprocess the data structure given $T$ and $\ell$ (Lemma 4.2.6). The step (3b) takes $|X| \cdot \tilde{O}(k \ell) = \tilde{O}(n/\ell) \cdot \tilde{O}(k \ell) = \tilde{O}(nk) = \tilde{O}(m)$ time. Indeed, by Lemma 4.2.6, the step (3b) takes $\tilde{O}(k \ell)$ time per call to Kernelize$(x)$ for each sample $x \in X$, and $|X| = \tilde{O}(n/\ell)$. Therefore, at step 3, it takes time:

$$\tilde{O}(\log n) \cdot \tilde{O}(m) + \tilde{O}(n/\ell) \cdot \tilde{O}(k \ell) = \tilde{O}(m + nk) = \tilde{O}(m).$$

Next, we argue the number of edges in the max-flow instances, which is the total size of all kernels produced by the algorithm. By Lemma 4.2.6, at step (3bi), the kernel has size at most $\tilde{O}(k \ell)$. Thus, the total instance size is

$$\tilde{O}(\log n) \cdot \tilde{O}(n/\ell) \cdot \tilde{O}(k \ell) = \tilde{O}(nk) = \tilde{O}(m).$$

\(^4\)Since $\ell \in [\lfloor L/2 \rfloor, \lfloor L \rfloor]$, the success probability analysis in Lemma 4.2.1 will change by only a constant factor.
4.2.4 Proof of Lemma 4.2.6

We are given a graph $G = (V,E)$ and an integer $\ell \leq k/100\log n$. Let $T$ be a random set where we select each vertex $v \in V$ with probability $1/\ell$. We explain how to implement KERNELIZE operation in $\tilde{O}(k\ell)$ time.

Let $x \in V$ be the input to KERNELIZE query. Let $T_x := T - N_G[x]$. Before building the kernel, we test the preconditions in the following.

**Testing Preconditions** We denote the symbol $\bot$ as the certification in Lemma 4.2.6.

- **Test 1.** If $\deg_G(x) > k + 2\ell$, then return $\bot$.
- **Correctness.** Suppose $x \in L$ in some $k$-scratch $(L,S,R)$ where $\ell \in [\vert L/2,|L|]$, then $\deg_G(x) \leq k + |L| \leq k + 2\ell$. Thus, $x \not\in L$ for any $k$-scratch $(L,S,R)$ where $\ell \in [\vert L/2,|L|]$.
- **Test 2.** If $N_G[x] \subseteq T$, then return $\bot$.
- **Correctness.** If true, then $T_x = \emptyset$. By Test 1, $\deg_G(x) \leq k + 2\ell$. We can test it in $O(\deg_G(x)\log n) = O((k + \ell)\log n)$ time by making $\deg_G(x)$ membership queries on the set $T$: for every $v \in N_G[x]$ query if $v \in T$ (using a balanced binary search tree).

After passing the tests, we know that $T_x \neq \emptyset$ and

$$x \deg_G(x) \leq k + 2\ell \leq 3k. \tag{4.8}$$

A key challenge for constructing the kernel is that we cannot afford to touch every edge in the entire graph since the kernel size is sublinear. In particular, we need to avoid touching the edges in the set $E_G(N_G(x), N_G(x))$ and we need to find an efficient way of checking if $\deg_{G - N_G[x]}(v)$ is too large (as in Equation 4.4) for a given vertex $v \in V - \{x\}$. To do so, we use the techniques from sublinear algorithms, namely linear sketching and sparse recovery, to implement the outneighbor oracle.

**Lemma 4.2.7.** [Outneighbor Oracle] Given a graph $G = (V,E)$ and parameters $k, \ell$, we can construct the data structure that supports the following operation with high probability.

- OUTNEIGHBOR$(x,v)$ query where $\deg_G(x) \leq k + 2\ell$ and $v \in V - \{x\}$: either output $N_G(v) - N_G[x]$ in $\tilde{O}(\ell)$ time or report “too large” in $O(\log n)$ time.
  If $\vert N_G(v) - N_G[x] \vert \leq 40\ell \log n$, then the set $N_G(v) - N_G[x]$ is returned. If $\vert N_G(v) - N_G[x] \vert \geq 100\ell \log n$, then “too large” is reported.

The data structure can be constructed in $\tilde{O}(m)$ time.

We will sketch the proof at the end of the section We construct the outneighbor oracle as a preprocessing step in $\tilde{O}(m)$ time. Next, we describe how to construct the kernel using the outneighbor oracle in $\tilde{O}(k\ell)$ time.
Building the Kernel. We grow a breadth-first search (BFS) tree starting from \( x \) with the following modification. Let \( v \) be a node being visited in the BFS process. We query \( \text{OUTNEIGHBOR}(x, v) \) that either reports “too large” in \( O(\log n) \) time or returns \( N_G(v) - N_G[x] \) in \( \tilde{O}(\ell) \) time. If “too large” is reported or \( (N_G(v) - N_G[x]) \cap T_x \neq \emptyset \), then we stop expanding \( v \) and treat it as a leaf in the tree. Otherwise, we expand \( v \) as in the BFS manner (by putting the non-visited nodes in the set \( N_G(v) - N_G[x] \) into the queue of the BFS algorithm).

Lemma 4.2.8. Let \( N_x, Z, F, N_l \) be the set of vertices of the kernel defined in Equation 4.2. For any non-root node \( v \) in the BFS tree, we have the following classification.

\[
\begin{align*}
\bullet & \text{ If } v \in N_G(x) \text{ and } v \text{ is an internal node, then } v \in N_x. \\
\bullet & \text{ If } v \in N_G(x) \text{ and } v \text{ is a leaf node, then } v \in Z. \\
\bullet & \text{ If } v \notin N_G(x), \text{ and } v \text{ is an internal node, then } v \in F. \\
\bullet & \text{ If } v \notin N_G(x), \text{ and } v \text{ is a leaf node, then } v \in N_l.
\end{align*}
\]

Proof. We assume for simplicity that we never visit the vertices in the sample set \( T_x \) during the BFS process. It turns out that this assumption is without loss of generality\(^5\). Consider a vertex \( v \) being visited during the BFS process. By Equation 4.8, the precondition in \( \text{OUTNEIGHBOR}(x, v) \) operation is satisfied. By Lemma 4.2.7, if “too large” is reported, then \( |N_G(v) - N_G[x]| > 40\log n \), which means \( v \notin N_x \cup F \) (Lemma 1.2.2). Otherwise, every vertex in \( N_G(v) - N_G[x] \) is returned, and we check if \( v \in N_G(T_x) \) by checking if \( (N_G(v) - N_G[x]) \cap T_x \neq \emptyset \). Therefore, we identify every node in the BFS tree into the set \( N_x, Z, F \) and \( N_l \) correctly.

During the BFS process, we return \( \perp \) as soon as the number of internal nodes in the BFS tree is greater than \( |N_x \cup F| \). We know the upper bound of \( |N_x \cup F| \) by the following. By Lemma 4.2.3 and Equation 4.8, \( |N_x \cup F| \leq O(k) \). Therefore, we can set the threshold to early terminate as soon as the number internal nodes exceeds \( \Omega(k) \) (using an appropriate constant).

If the algorithm finishes the BFS process, then the number of internal nodes is \( O(k) \). Given the BFS tree, we construct the kernel by identifying the (subset of) vertex sets \( N_x, Z, F, \) and \( N_l \) of the kernel according to Lemma 4.2.8. Finally, we add a sink vertex \( t_x \) and its edge to every vertex in \( N_l \) in the kernel.

Correctness. We prove that either we successfully construct a kernel or we return \( \perp \) correctly certifying that there is no \( k \)-scratch \((L, S, R)\) satisfying Equation 4.7. If such a \( k \)-scratch exists, \( |F| \leq O(k) \) because \( x \in L, \phi \neq T_x \leq R \) and Lemma 4.2.3. Thus, by the contrapositive form and Lemma 4.2.8, whenever we find that the number of internal nodes exceeds\(^6\) \( \Omega(k) > |N_x \cup \)

\(^5\) This can be shown by induction on the level of the BFS tree that we always stop expanding at the neighbor of \( T_x \) before visiting any vertex in \( T_x \).

\(^6\) using an appropriate constant
Therefore, the set of slow nodes is $I = \bigcup_{x \in N_t} \{v \in N_G(v) - N_G[x] \mid \deg_G(x) \leq \Omega(n) \}$. Otherwise, the algorithm builds a kernel defined in Section 4.2.1. Note that we end up constructing a subgraph of the kernel because we do not touch the set of vertices in $F$ whose are not reachable from $x$ in $G - N_t$. By Proposition 4.2.5(3), we can safely remove those vertices.

**Running Time.** We prove that the BFS process takes $\tilde{O}(k \ell)$ time. Let $v$ be a non-root node in the BFS tree at the end. We say that $v$ is fast if $\text{OUTNEIGHBOR}(x, v)$ returned “too large”. Otherwise, we say that $v$ is slow.

**Lemma 4.2.9.** The algorithm spent $O(\log n)$ time on each fast node, and $\tilde{O}(\ell)$ time on each slow node.

**Proof.** Let $v \neq x$ be a node in the BFS tree. If $v$ is a fast node, then the $\text{OUTNEIGHBOR}(x, v)$ returned “too large”, which took $O(\log n)$ time by Lemma 4.2.7. If $v$ is a slow node, then $N_G(v) - N_G[x]$ of size at most $\tilde{O}(\ell)$ was returned by $\text{OUTNEIGHBOR}(x, v)$. The algorithm then checks if $(N_G(v) - N_G[x]) \cap T_x = \emptyset$ in $\tilde{O}(\ell)$ time. Indeed, this can be implemented by checking for each $w \in N_G(v) - N_G[x]$ if $w \in T_x$, which is equivalent to checking if $w \in T$ (since $w \notin N_G[x]$). We can check $w \in T$ using a balanced binary search tree on $T$ in $O(\log n)$ time. Therefore, the total time to check if $(N_G(v) - N_G[x]) \cap T_x = \emptyset$ is $\tilde{O}(\ell)$.

**Lemma 4.2.10.** The number of slow nodes is $O(k)$, and the number of fast nodes is $\tilde{O}(k \ell)$.

**Proof.** Let $I, Z, N_t$ be the set of internal nodes, the set of leaf nodes neighboring to $x$, and the set of leaf nodes not neighboring to $x$ in the BFS tree, respectively. We claim that the set of slow nodes is $I \cup Z$, and the set of fast nodes is $N_t$. By Lemma 4.2.8, $I = N_x \cup F, Z = Z$ and $N_t \subseteq N_t$. By Lemma 1.2.2, those with high degree outside $N[x]$ must be in $Z \cup N_t$, and those with low degree outside $N[x]$ must be in $N_x \cup F$. If $v$ has high degree outside $N[x]$, then $\text{OUTNEIGHBOR}(x, v)$ returns “too large” with high probability. Therefore, the set of slow nodes is $I \cup Z$, and the set of fast nodes is $N_t$.

The number of slow nodes is $|I| + |Z| \leq O(k)$ because we early stop whenever the number of internal nodes exceed $\Omega(k)$, and $|Z| \leq \deg_G(x) \leq O(k)$ by Equation 4.8. The number of fast nodes is $|N_t|$, which is at most the number of leaf nodes which is at most the number of edges listed by each internal node. Since the number of internal nodes is $O(k)$, and the number of new edges is $\tilde{O}(\ell)$ per internal node, the number of leaf nodes is $\tilde{O}(k \ell)$.

By Lemma 4.2.9 and 4.2.10, the total running time to run a BFS tree using the outneighbor oracle is $O(k) \cdot \tilde{O}(\ell) + \tilde{O}(k \ell) \cdot O(\log n) = \tilde{O}(k \ell)$.

**Proof Sketch of Lemma 4.2.7.** We use norm estimation and $s$-sparse recovery sketches to implement the oracle. $s$-sparse recovery is a linear
mapping from a vector $a \in \mathbb{Z}^n$ to a smaller vector $s_k(a) \in \mathbb{Z}^{\tilde{O}(s)}$ in $\tilde{O}(\|a\|_0)$ time. Furthermore, if $\|a\|_0 \leq s$, then we can recover $a$ from $s_k(a)$ in $\tilde{O}(s)$ time.

We preprocess the oracle as follows. We set $s = \tilde{O}(\ell \log n)$. For each node $v$, denote $\vec{I}_{N(v)}$, $\vec{I}_{N[v]} \in \{0,1\}^n$ as indicator vectors of $N(v)$ and $N[v]$, respectively. We precompute $s_k(\vec{I}_{N(v)})$ and $s_k(\vec{I}_{N[v]})$ for all $v$. This takes $\sum_v \tilde{O}(\deg v) = \tilde{O}(m)$ time.

We can support the operation $\text{OUTNEIGHBOR}(x,v)$ where $\deg(x) \leq k + 2\ell$ as follows. We compute $s_k(\vec{I}_{N(v)}) - s_k(\vec{I}_{N[x]}) = s_k(\vec{I}_{N(v)} - \vec{I}_{N[x]})$. The equality follows because the sketching is linear. Observe that the non-zero entries of $\vec{I}_{N(v)} - \vec{I}_{N[x]}$ corresponds to the symmetric difference $N(v) \triangle N[x]$. Using the fact that $\deg(x) \leq k + 2\ell$, we can also show that $|N(v) \triangle N[x]| = \Theta(|N(v) - N[x]| + \ell)$. If $\deg_G - N[x](v) \leq \tilde{O}(\ell)$, then $\vec{I}_{N(v)} - \vec{I}_{N[x]}$ has at most $s$ non-zero entries, and thus we can recover $N(v) \triangle N[x]$ from $s_k(\vec{I}_{N(v)} - \vec{I}_{N[x]})$ in $O(s) = \tilde{O}(\ell)$ time. Therefore, we can obtain $N(v) - N[x]$ in $\tilde{O}(\ell)$ time.

The condition $\deg_G - N[x](v) \leq \tilde{O}(\ell)$ can be checked in $O(\log n)$ time using another sketching (which is norm estimation). Since each entry of $\vec{I}_{N(v)} - \vec{I}_{N[x]}$ is either $-1,0$ or $1$, it is enough to estimate $\|\vec{I}_{N(v)} - \vec{I}_{N[x]}\|_2^2 = \|\vec{I}_{N(v)} - \vec{I}_{N[x]}\|_0$, which is proportional to $|N(v) - N[x]| + \ell$. Although norm estimation is not exact, we set the approximation guarantee to be a small constant, which suffices for our purpose.

### 4.3 Using Isolating Vertex Cut Lemma

We prove Lemma 4.1.3 in this section. We start with proving isolating vertex cut lemma in Section 4.3.1. To prove Lemma 4.1.3, we present an algorithm that uses isolating vertex lemma in Section 4.3.2.

#### 4.3.1 Isolating Vertex Cut Lemma

We start with the proof of isolating vertex cut lemma. We restate the lemma for convenience.

**Lemma 4.1.2** (Isolating Vertex Cut Lemma). Given a graph $G = (V,E)$, and an independent set $I \subseteq V$ of size at least 2 as inputs, there is an algorithm that outputs, for each vertex $v \in I$, a $(v,I \setminus \{v\})$-min separator. The algorithm makes $st$-maxflow calls on unit-vertex-capacity graphs with $O(m \log |I|)$ total number of vertices and edges, and takes $O(m)$ time outside of maxflow calls.

**Proof Sketch.** We first assign binary representation to each vertex in $I$ by assigning distinct $\lceil \log_2 |I| \rceil$ bits to each vertex in $I$. At $i$-th bit, we partition the set $I$ into $A_i, B_i$: For each vertex $v \in I$, we put $v$ into $A_i$ if its $i$-th bit is zero, and into $B_i$ otherwise. Let $C_i$ be a min $(A_i, B_i)$-separator which can be computed using one max-flow call. Note that $C_i$ exists because
I is an independent set. We compute such a cut \( C_i \) for all \( i \) in \( O(\log_2 |I|) \) max-flow calls. The key claim is that every vertex in \( I \) belongs to a different connected component in \( G - \bigcup_i C_i \). Indeed, for any \( u, v \in I \), there is \( i \) such that the \( i \)-th bit of \( u \) are different from that of \( v \), and thus \( u \) and \( v \) will be separated by the \((A_i, B_i)\)-separator \( C_i \).

Next, we argue that for all \( v \in I \), a min \((v, I - \{v\})\)-separator in \( G \) can be found in the connected component \( U_v \) containing \( v \) in \( G - \bigcup_i C_i \). Fix \( v \in I \). Let \( U_v^* \) be an inclusion-wise minimal min-\((v, I - \{v\})\)-separator in \( G \) (where the side containing \( v \) is the smallest). We can show that \( U_v^* \subseteq U_v \) using submodularity of vertex cuts. Therefore, a min \((v, I - \{v\})\)-separator can be found in the induced subgraph \( G[U_v \cup N_G(U_v)] \) followed by removing all edges with both endpoints in \( N_G(U_v) \) and adding a sink vertex \( t \) connecting to all vertices in \( N_G(U_v) \). We compute \((v, t)\)-vertex mincut (via \((v, t)\)-max-flow) in \( G_v \) which corresponds to a min \((v, I - \{v\})\)-separator in \( G \). By definition of \( G_v \), one can argue that the total size \( \sum_{v \in I} |E(G_v)| = O(m) \).

**Definition 4.3.1.** We say that a vertex cut \((L, S, R)\) of \( G = (V, E) \) isolates a vertex set \( T \) if

\[
|L \cap T| = 1, S \cap T = \emptyset, \text{ and } R \cap T \neq \emptyset.
\]

If we obtain the set \( T \) (e.g., by a sampling algorithm) that is isolated by a vertex cut of size \( < k \), then we can find a separator of size \( < k \) efficiently.

**Corollary 4.3.1.** If we are given a vertex set \( T \) that is isolated by a vertex cut \((L, S, R)\) of size \( < k \) in \( G \), then we can output a separator of size \( < k \) in poly-logarithmic max-flows.

**Proof.** Let \( I \) be a maximal independent set from \( T \). Since \((L, S, R)\) isolates \( T \), it follows that \((L, S, R)\) isolates \( I \) as well. Since \( I \) is an independent set isolated by \((L, S, R)\), there is \( v \in I \) such that \( v \in L \) and \( I - \{v\} \subseteq R \). Thus, Lemma 4.1.2 outputs \((v, I - \{v\})\)-min separator \( C \) in \( G \). Since \( v \in L \) and \( I - \{v\} \subseteq R \), we have \( |C| \leq |S| < k \).

### 4.3.2 Algorithms for Detecting \( k \)-Non-Scratch Vertex Cuts

Recall Definition 4.1.2 that a vertex cut \((L, S, R)\) is \( k \)-non-scratch if \(|S| < k \), but either \(|L| > k/(100 \log n) \) or \(|S_{\text{low}}| < 300|L| \log n \). Also, \( V_{\text{low}} = \{v \in V : \deg_G(v) \leq 8k \} \), and \( S_{\text{low}} = S \cap V_{\text{low}} \).

Our goal is to develop an algorithm to output a vertex cut of size \( < k \) with high probability if a \( k \)-non-scratch vertex cut exists. Our strategy is to sample many random subset so that a \( k \)-non-scratch isolates one of the random sets with high probability. If so, we can extract a separator of size \( < k \) by Corollary 4.3.1.

We describe the sampling strategy. We denote \( V(p) \) as a random subset of vertices \( V \) where we sample each vertex in \( V \) with probability \( p \). Also,
we denote $V_{\text{low}}(p)$ as a random subset of $V_{\text{low}}$ where we sample each vertex in $V_{\text{low}}$ with probability $p$.

**Lemma 4.3.2.** If $G$ contains a $k$-non-scratch $(L, S, R)$ where $|L| > k/(100 \log n)$, then it isolates $T = V(p)$ with probability at least $\Omega(1/\log^2 n)$ for all $p$ satisfying $1 \leq p(2|L| + |S|) \leq 2$.

**Lemma 4.3.3.** If $G$ contains a $k$-non-scratch $(L, S, R)$ where $|L| \leq k/(100 \log n)$, then it isolates $T = V_{\text{low}}(p)$ with probability at least $\Omega(1/\log^2 n)$ for all $p$ satisfying $1 \leq p(2|L| + |S_{\text{low}}|) \leq 2$.

We defer the proofs to the end of the section. Intuitively, Lemma 4.3.2 and Lemma 4.3.3 say that if we sample random subsets $V(p)$ and $V_{\text{low}}(p')$ using probability $p = 1/\Theta(|L| + |S|)$ and and $p' = 1/\Theta(|L| + |S_{\text{low}}|)$, then a $k$-non-scratch isolates either $V(p)$ or $V_{\text{low}}(p')$ with probability $\Omega(1/\log^2 n)$. We repeat $\Theta(\log^3 n)$ time to obtain such an event with high probability. By applying Corollary 4.3.1, we obtain a separator of size $< k$ with high probability in poly-logarithmic max-flows. We now make the intuition precise.

We first describe the algorithm, and we prove Lemma 4.1.3.

**Algorithm.** The inputs are a graph $G = (V,E)$ and a parameter $k$ where $m \leq nk$.

- Let $p_i = 1/2^i$ for $i \in \{1,2,\ldots,\log n\}$.
- For each $i$, we repeat the following for $\Theta(\log^3 n)$ times:
  - Let $I$ and $I'$ be maximal independent sets (MIS) from the random sets $V(p_i)$ and $V_{\text{low}}(p_i)$, respectively.
  - Apply Lemma 4.1.2 on $(G, I)$ if $|I| \geq 2$ and $(G, I')$ if $|I'| \geq 2$.
- Finally, we output the smallest separator found so far in the algorithm (or a min-degree cut if none of the separators was found).

**Running Time.** The algorithm makes $s,t$-max-flow calls on unit-vertex-capacity graphs of $O(m \log^5 n)$ vertices and edges in total because the number of calls to the isolating vertex cut lemma (Lemma 4.1.2) is $O(\log^4 n)$. In addition, the algorithm takes $\tilde{O}(m)$ time outside the max-flow calls.

**Correctness.** The algorithm outputs some separator by design. We prove that if $G$ contains a $k$-non-scratch, then a separator of size $< k$ is returned with high probability. Indeed, assume there is a $k$-non-scratch $(L, S, R)$. There are two cases. In any case, we obtain a separator of size $< k$ with high probability.

- If $|L| > k/(100 \log n)$, then there is $i$ such that $1 \leq p_i(2|L| + |S|) \leq 2$.
  By Lemma 4.3.2, $(L, S, R)$ isolates $V(p_i)$ with probability at least
(in)equality follows since the events are independent. The second inequality follows since $\Omega(1/\log^2 n)$. Since we repeat for $\Theta(\log^3 n)$ times, with high probability, there is an iteration where $(L,S,R)$ isolates $V(p_i)$. By applying isolating vertex cut lemma (Lemma 4.1.2) on $(G,I)$ where $I$ is an MIS of $V(p_i)$, we obtain a separator of size $< k$.

• If $|L| \leq k/(100\log n)$, then there is $i$ such that $1 \leq p_i(2|L| + |S_{low}|) \leq 2$. By Lemma 4.3.3, $(L,S,R)$ isolates $V_{low}(p_i)$ with probability at least $\Omega(1/\log^2 n)$. Since we repeat for $\Theta(\log^3 n)$ times, with high probability, there is an iteration where $(L,S,R)$ isolates $V_{low}(p_i)$. By applying isolating vertex cut lemma (Lemma 4.1.2) on $(G,I')$ where $I'$ is an MIS of $V_{low}(p_i)$, we obtain a separator of size $< k$.

It remains to prove Lemma 4.3.2 and Lemma 4.3.3.

**Proof of Lemma 4.3.2.** Let $p$ be a real number satisfying $1 \leq p(2|L| + |S|) \leq 2$. The probability that $(L,S,R)$ isolates $T := V(p)$ is

$$\mathbb{P}(|T \cap L| = 1, |T \cap S| = 0, |T \cap R| \geq 1) = \mathbb{P}(|T \cap L| = 1) \cdot \mathbb{P}(|T \cap S| = 0) \cdot \mathbb{P}(|T \cap R| \geq 1)$$

$$\geq \mathbb{P}(|T \cap L| = 1)^2 \cdot \mathbb{P}(|T \cap S| = 0)$$

$$= (|L|p(1-p)|L|-1)^2 \cdot (1-p)^{|S|}$$

$$\geq (|L|p)^2(1-p)^{|L|+|S|} = \Omega(1/\log^2 n).$$

The first equality follows since the events are independent. The second (in)equality follows since $|R| \geq |L|$. The third equality follows by the definition of $V(p)$. Since $|S| < k < |L|100\log n$, $|L|p = \Omega(1/\log n)$. The fourth inequality follows since $2|L| + |S| = \Theta(1/p)$, and $|L|p = \Omega(1/\log n)$. \(\square\)

Next, we prove Lemma 4.3.3. Recall that (by the preconditions of the graph $G$ in Lemma 4.1.3)

$$m \leq nk. \quad (4.9)$$

**Proof of Lemma 4.3.3.** Let $(L,S,R)$ be such a $k$-scratch. By the precondition, $|S_{low}| < 300|L|\log n$. Since we sample from the set $V_{low}$, we need to argue about the property of the sets $L_{low} := L \cap V_{low}$ and $R_{low} := R \cap V_{low}$.

We claim that $L_{low} = L$, and $|R_{low}| \geq |L_{low}|$. To see that $L_{low} = L$, any vertex $v \in L$ has $N[v] \subseteq L \cup S$, and thus $\deg(v) \leq |L| + |S| \leq 2k$. Next we show that $|R_{low}| \geq |L_{low}|$. If $k \geq n/8$, then $v \in V_{low}$ for every vertex $v$ in the graph, and thus $|R_{low}| = |R| \geq |L| = |L_{low}|$. Otherwise, $k < n/8$. In this case, $|L \cup S| \leq 2k \leq n/4$, and thus $|R| = n - |L \cup S| \geq 3n/4$. Since $8k|V - V_{low}| \leq \sum_v \deg(v) \leq 2nk$, $|V - V_{low}| \leq n/4$. Therefore, $|R_{low}| \geq |R| - |V - V_{low}| \geq 3n/4 - n/4 \geq n/2 \geq k/2 \geq k/(100\log n) \geq |L| \geq |L_{low}|$.

Using a similar argument as in the proof of Lemma 4.3.2, the probability
that \((L, S, R)\) isolates \(T_{\text{low}} := V_{\text{low}}(p)\) is

\[
\Pr(|T_{\text{low}} \cap L| = 1, |T_{\text{low}} \cap S| = 0, |T_{\text{low}} \cap R| \geq 1)
\]

\[
= \Pr(|T_{\text{low}} \cap L_{\text{low}}| = 1) \cdot \Pr(|T_{\text{low}} \cap S_{\text{low}}| = 0) \cdot \Pr(|T_{\text{low}} \cap R_{\text{low}}| \geq 1)
\]

\[
\geq \Pr(|T \cap L_{\text{low}}| = 1)^2 \cdot \Pr(|T_{\text{low}} \cap S_{\text{low}}| = 0)
\]

\[
= (|L|p(1-p)^{|L|-1})^2 \cdot (1-p)^{|S_{\text{low}}|}
\]

\[
\geq (|L|p)^2(1-p)^2|S_{\text{low}}| = \Omega(1/\log^2 n).
\]

\(\square\)
5. Almost Linear Time Deterministic Algorithms for Small Vertex Connectivity

The goal of this chapter is to provide an overview of fast deterministic algorithms for small vertex connectivity: Given an undirected graph $G$ and a connectivity parameter $k$, we can decide $k$-vertex connectivity in $m^{1+o(1)}2^{O(k^2)}$ time. The approach is to derandomize the $O(mk^2)$-time algorithm in Chapter 3. The challenge for derandomization is to avoid sampling random vertices/edges as predominantly used in the randomized algorithms.

The intuition is as follows. Roughly speaking, random sampling is not necessary when the vertex cut is sufficiently unbalanced, which is the case for vertex expanders. If the vertex cut is balanced, then we can reduce the balancedness by contracting vertices outside the cut until it becomes unbalanced or sufficiently small. The challenging part is to know which vertices are safe to contract. The combination of the two important tools can tell us which vertices are safe to contract: a reduction framework to vertex expander and vertex cut sparsifier.

We make the intuition precise. In Section 5.1, we show that vertex expanders have unbalanced vertex cut, and thus $k$-vertex connectivity can be solved in near-linear time when $k$ is small. In general, the graph may be not an expander. In Section 5.2, we explain the reduction framework to vertex expanders. The framework will reduce a graph into a collection of expanders and a terminal set that characterizes the $k$-vertex connectivity of the graph. In Section 5.3, we discuss vertex cut sparsifier given a terminal set. Finally, we describe the full vertex connectivity algorithm in Section 5.4.

5.1 Deciding $k$-Vertex Connectivity in Vertex Expanders

In this section, we solve $k$-vertex connectivity in expanders using the deterministic LocalVC algorithm developed in Chapter 3. Let $(L,S,R)$ be a vertex cut of the graph $G$. The expansion of the cut is $h(L,S,R) = \frac{|S|}{|S| + \min(|L|,|R|)} \in [0,1)$. If $h(L,S,R) < \phi$, we say that the vertex cut is $\phi$-sparse.
The expansion of the graph is \( h(G) = \min_{(L,S,R)} h(L,S,R) \). We say that a graph \( G \) is a \( \phi \)-vertex expander if \( h(G) \geq \phi \), i.e., there is no \( \phi \)-sparse vertex cut.

**Lemma 5.1.1.** If a \( \phi \)-vertex expander has a vertex cut \( (L,S,R) \) of size \( < k \) (where \( |L| \leq |R| \)), then \( |L| \leq k/\phi \).

**Proof.** Since \( h(L,S,R) = \frac{|S|}{|L|+|S|} \geq \phi \), we have \( k/\phi \geq |S|/\phi \geq |L| + |S| \geq |L| \). \( \square \)

**Lemma 5.1.2.** There is a deterministic algorithm that solves \( k \)-vertex connectivity for \( \phi \)-vertex expanders in \( O(nk^7/\phi^4) \) time.

**Proof.** We present the algorithm and analysis.

**Algorithm for \( \phi \)-expanders.** Let \( \ell = k/\phi \). For each vertex \( v \), we call \( \text{LocalVC} \) starting on \( v \) with volume parameters \( v = \ell^2 + k\ell \), and connectivity parameter \( k \). We return a vertex cut if one of the calls returns a vertex set. Otherwise, we return the symbol \( \perp \), certifying that \( G \) does not have a vertex cut of size \( < k \).

**Running Time.** It takes \( O(nk^7/\phi^4) \) time in total. There are \( n \) calls to \( \text{LocalVC} \) with the same volume and connectivity parameters, and each call to \( \text{LocalVC} \) takes \( \tilde{O}(v^2 k) = \tilde{O}(\ell^4 k^3) = \tilde{O}(k^7/\phi^4) \) time.

**Correctness.** The algorithm either outputs a vertex cut of size \( < k \) or the symbol \( \perp \). We prove that if \( G \) has a vertex cut of size \( < k \), then the algorithm will output some vertex cut of size \( < k \). Fix a vertex cut \( (L,S,R) \) of size \( < k \). Let \( x \in L \) be one of the starting vertices for the \( \text{LocalVC} \) algorithm. Since \( G \) is a \( \phi \)-expander, Lemma 5.1.1 implies that \( |L| \leq k/\phi = \ell \). Therefore, \( \text{vol}^\text{out}(L) \leq |L|^2 + |L||S| \leq \ell^2 + \ell k \leq v \). Thus, \( \text{LocalVC} \) on \( x \in L \) with parameters \( v,k \) will output a separator of size \( < k \). \( \square \)

### 5.2 Reduction Framework to Vertex Expanders and Terminal Set

In Section 5.1, we show that we can decide \( k \)-vertex connectivity in a \( \phi \)-vertex expander in \( \tilde{O}(nk^7/\phi^4) \) time. One can think of \( \phi^{-1} = n^{o(1)} \). In general, the graph may not be a vertex expander. In this section, we describe the reduction framework to a collection of expanders and terminal sets. Before stating the framework, we discuss intuition.

If the graph is not a \( \phi \)-vertex expander, there is a \( \phi \)-sparse vertex cut \( (L,S,R) \). We will contract \( L \) and \( R \) in the following sense. Define left graph \( G_L \) and right graph \( G_R \) of \( G \) w.r.t. \( (L,S,R) \) as follows. Define \( G' \) as \( G \) after removing the set of edges in \( E_G(S,S) \). Define \( G_L \) as \( G' \) after replacing \( R \) with a clique of size \( c \) followed by adding biclique edges between \( S \) and \( R \). \( G_R \) is defined symmetrically. That is, \( G_R \) is \( G' \) after replacing \( L \) with a clique of size \( c \) followed by adding biclique edges between \( L \)
and $S$. For a set $S$ of vertices, we define the $S$-Steiner connectivity as $\kappa_G(S) = \min_{x,y \in S} \kappa_G(x,y)$.

**Lemma 5.2.1.** Let $(L,S,R)$ be a vertex cut in $G$. Then, $G$ is $k$-connected if and only if

1. $S$-Steiner connectivity $\kappa_G(S) \geq k$, and

2. The left and right graph of $G$ w.r.t. the vertex cut $(L,S,R)$, $G_L$ and $G_R$ are both $k$-connected.

**Proof.** We prove that if $G$ has a vertex mincut $(L^*,S^*,R^*)$ of size $<k$, then either $S^*$ is a separator in $G_L$ or $G_R$ or $\kappa_G(S) < k$ (the proof of the other direction is easier, and we omitted it for brevity). If $|S| < k$, then $S$ is a separator of size $<k$ in both $G_L$ and $G_R$ and we are done. If $\kappa_G(S) < k$, then we are done. Now, we assume $\kappa_G(S) \geq k$ and $|S| \geq k$. Let $(L^*,S^*,R^*)$ be a vertex mincut of size $<k$ in $G$. We first prove that $S \cap L^* = \emptyset$ or $S \cap R^* = \emptyset$. Indeed, suppose otherwise. $S^*$ is an $(s,t)$-separator where $s \in S \cap L^*$ and $t \in S \cap R^*$, and thus $\kappa_G(s,t) \leq |S^*| < k$. Since $s,t \subseteq S$, we have $\kappa_G(S) < k$, contradicting to the fact that $\kappa_G(S) \geq k$. Since $|S| \geq k$, we have $S \not\subseteq S^*$, and thus both $S \cap L^*$ and $S \cap R^*$ cannot be empty at the same time. Therefore, exactly one of them is non-empty. WLOG assume $S \cap L^* \neq \emptyset$ and $S \cap R^* = \emptyset$. Let $x \in S \cap L^*$. See Figure 5.1 for illustration. Since $R^* \neq \emptyset$ and $S \cap R^* = \emptyset$, $L \cap R^* \neq \emptyset$ or $R \cap R^* \neq \emptyset$. If $L \cap R^* \neq \emptyset$, then we prove that $S^*$ is a separator in $G_L$. If $R \cap R^* \neq \emptyset$, then $S^*$ will be a separator in $G_R$ instead. Now, we assume $L \cap R^* \neq \emptyset$. Let $y \in L \cap R^*$. Since $R^* \cap S = \emptyset$, we have $N_G(L \cap R^*) \subseteq S^*$. Since $S^*$ is a minimum separator, $R \cap S^* = \emptyset$, meaning that $S^* \subseteq L \cup S$. Since $S^* \subseteq L \cup S$ and $S \cap R^* = \emptyset$, $S^*$ remains a separator in $G_L$. $\Box$

**Figure 5.1.** A crossing-diagram for two vertex cuts $(L,S,R)$ and $(L^*,S^*,R^*)$ before and after transformation from $G$ to $G_L$. If $L \cap R^* \neq \emptyset$, we show that $S^*$ is a vertex cut in $G_L$.

Lemma 5.2.1 naturally implies a recursive algorithm as follows. Fix $\phi = 1/|V(G)|^{o(1)}$ throughout the algorithm.

**Recursive algorithm.** Let $H$ be the graph at the current recursion. If $H$ is a $\phi$-vertex expander, then run the algorithm in Section 5.1. Otherwise, let $(L,S,R)$ be a $\phi$-sparse vertex cut. If $\kappa_H(S) < k$, then we output the $S$-Steiner mincut. Otherwise, we recurse on the left and right graphs of $H$. 
Almost Linear Time Deterministic Algorithms for Small Vertex Connectivity

Unrolling the recursion. Alternatively, we can unroll the recursion all the way to the base cases without computing \( S \)-Steiner connectivity. The non-recursion form of Lemma 5.2.1 can be viewed as unrolling the recursive algorithm in the following manner. Let \( \mathcal{F}(G) \) be the recursion tree of the recursive algorithm. The execution of the recursive algorithm starts at the root with the input graph \( G \) in the recursion tree. Each node \( v \) in the recursion tree has graph \( H \) as an input. If \( H \) is a \( \phi \)-vertex expander, then \( v \) becomes the leaf (representing the base case). Otherwise, \( v \) becomes an internal node. We compute a \( \phi \)-sparse vertex cut \((L,S,R)\), and construct left graph \( H_L \) and right graph \( H_R \) of \( H \) w.r.t. the sparse vertex cut \((L,S,R)\), and recurse. In other words, the internal node \( v \) has two children: one for \( H_L \) and the other one for \( H_R \).

Let \( I \) be the set of internal nodes of the recursion tree \( \mathcal{F}(G) \). Let \( T = \bigcup_{v \in I} S_v \) where \( S_v \) is the separator of the \( \phi \)-sparse vertex cut obtained at node \( v \) in the recursion tree \( \mathcal{F}(G) \). Let \( \mathcal{X} \) be a collection of \( \phi \)-vertex expanders at leaf nodes of the recursion tree \( \mathcal{F}(G) \).

Corollary 5.2.2. \( G \) is \( k \)-vertex connected if and only if \( T \)-Steiner connectivity \( \kappa_G(T) \geq k \) and every expander in \( \mathcal{X} \) is \( k \)-vertex connected.

We are now ready to state the expander reduction framework.

Theorem 5.2.3. Given an undirected graph \( G = (V,E) \) with \( n \) vertices and \( m \) edges and parameters \( \phi \in (0, 1/2k \log^2 n) \) and \( k > 0 \), there is a \( m^{1+o(1)/\phi} \)-time deterministic algorithm, denoted as EXPANDERSORTERMINAL\((G,k,\phi)\), that outputs a collection of \( \phi \)-expanders and a terminal set \( T \subseteq V \) satisfying the following:

1. \( G \) is \( k \)-vertex connected if and only if \( \kappa_G(T) \geq k \) and every expander in \( \mathcal{X} \) is \( k \)-vertex connected,
2. \(|T| \leq n^{1+o(1)} \phi \), and
3. \( \sum_{X \in \mathcal{X}} |V(X)| \leq n^{1+o(1)} \).

To decide \( k \)-vertex connectivity on a general graph, we apply the expander reduction framework as follows. Given a graph \( G \), the algorithm in Theorem 5.2.3 produces a collection of expanders \( \mathcal{X} \) and a terminal set \( T \) satisfying the three properties stated in the statement in almost linear time. By the first property, it is enough to verify \( k \)-vertex connectivity for each expanders and solve \( T \)-Steiner mincut. By Lemma 5.1.2, deciding \( k \)-vertex connectivity in an expander can be done in \( n^{1+o(1)}k^7 \) time. It remains to focus on computing \( T \)-Steiner connectivity. In general, \( T \)-Steiner connectivity is harder than vertex connectivity because we can set \( T = V(G) \), and the problem becomes vertex connectivity of \( G \). Thanks to the
second property that \( |T| \leq n^{1+o(1)}\phi \), we can control the parameter \( \phi \) for our purpose.

Instead of solving \( T \)-Steiner connectivity directly, we compress the graph \( G \) into a smaller graph \( H \) while preserving \( \min T \)-Steiner cuts up to connectivity \( k \). The intuition is that if \( T \)-Steiner connectivity in \( G \) is \(< k \), then there is a separator of size \(< k \) that separates \( T \). We want to compress graph \( G \) into \( H \) in a way that \( H \) preserves the same separator. The notion of cut-preserving compression is known as \( k \)-vertex cut sparsifier, or \( k \)-connectivity mimicking network. We discuss this in the next section.

**Remarks on the method of the proof of Theorem 5.2.3.** We compute the recursion tree \( \mathcal{T}(G) \) and output \( \mathcal{X} \) the collection of \( \phi \)-vertex expanders at leaf nodes of the recursion tree, and \( T = \bigcup_{v \in I} S_v \) the union of the separators from the sparse cut obtained from internal nodes of the recursion tree. By Corollary 5.2.2, \( G \) is \( k \)-vertex connected if and only if \( \kappa_G(T) \geq k \) and every expander in \( \mathcal{X} \) is \( k \)-vertex connected. It remains to show the last two properties.

To do so, we briefly argue the running time to compute the recursion tree \( \mathcal{T}(G) \). It depends on the balancedness of \( \phi \)-sparse vertex cuts at each internal node. For example, if \( \phi \)-sparse vertex cuts are always balanced, then the recursion depth is \( O(\log n) \). However, we cannot guarantee balancedness always. Instead, among those sparse cut \((L,S,R)\), we find the one \((L,S,R)\) such that \( \min(|L|,|R|) \) is \( n^{o(1)} \)-approximation to the most balanced one. This can be computed in almost linear time (see ). By the standard argument in expander decomposition (see ), one can expect that the recursion depth is \( n^{o(1)} \). Since the total number of vertices is roughly linear per level, and there are \( n^{o(1)} \) levels, we can show that the total number of nodes at the leaves, \( \sum_{X \in \mathcal{X}} |V(X)| \) is roughly \( n^{1+o(1)} \). Next, if a vertex cut \((L,S,R)\) is \( \phi \)-vertex sparse, then \( |S| \leq n\phi \). We can expect \( |T| \leq n^{1+o(1)}\phi \) since every sparse cut in the algorithm is \( \phi \)-sparse.

### 5.3 Vertex Cut Sparsifier

Given a graph \( G = (V,E) \) and a terminal set \( T \subseteq V \), we want to output a smaller graph \( H \) of size proportional to \( T \) and \( H \) preserves all minimum vertex cuts between terminal \( T \) to up connectivity \( k \). We now define \( H \) more precisely. Fix \( A,B \subseteq T \). A separator \( S \) is \((A,B)\)-weak separator if there is no path from \( A \) to \( B \) in \( G - S \) where \( S \) may contain some vertices in \( A \) and \( B \). We also denote \( \mu_G(A,B) \) to be the minimum \((A,B)\)-weak separator in \( G \). By definition, \( \mu_G(A,B) \leq \min(|A|,|B|) \).

We say that \( H \) is \( k \)-cut recoverable from \( G \) if \( V(H) \subseteq V(G) \) and every separator of size \(< k \) in \( H \) is a separator in \( G \). By definition of \( k \)-cut recoverable, \( \kappa_H \geq \min(k,\kappa_G) \). We are ready to define \( k \)-vertex cut sparsifier.
Definition 5.3.1. Given a graph \( G = (V,E) \), a terminal set \( T \subseteq V \) and a parameter \( k > 0 \), a \( k \)-vertex cut sparsifier for \( G \) is a (smaller) graph \( H \) where \( V(H) \subseteq V(G) \) satisfying:

1. For all \( A,B \subseteq T \), \( H \) preserves a min \( (A,B) \)-weak separator of size at most \( k \), i.e., \( \min(\mu_G(A,B),k) = \min(\mu_H(A,B),k) \), and
2. \( H \) is \( k \)-cut recoverable from \( G \). In particular, \( \kappa_H \geq \min(k,\kappa_G) \).

Our main result is the following.

Theorem 5.3.1. Given an undirected graph \( G = (V,E) \), terminal set \( T \subseteq V \), and a parameter \( k > 0 \), there is an algorithm, \( \text{VERTEXSPARSIFY}(G,T,k) \), that outputs a \( (T,k) \)-sparsifier for \( G \) where \( |E(H)| \leq |V(H)|k \) and \( |V(H)| \leq |T|2^{O(k^2)} \). The algorithm takes \( m^{1+o(1)}2^{O(k^2)} \) time.

We explain how to apply Theorem 5.3.1 in the next section. At high level, we obtain the terminal set \( T \) from \( \text{EXPLANDEROR TERMINAL}(G,k,\phi) \), and obtain \( H = \text{VERTEXSPARSIFY}(G,T,k) \). Since we can check \( k \)-connectivity of all expanders in almost linear time, we focus on \( T \)-Steiner connectivity of \( G \). Since \( H \) preserves all min vertex cuts between \( T \) and \( H \) is \( k \)-cut recoverable, it is enough to check if \( H \) is \( k \)-vertex connected. By setting \( \phi = 1/(2^{O(k^2)}n^{o(1)}) \), we have \( |T| \leq n/2^{O(k^2)} \), and thus \( |E(H)| \leq n/10 \). Therefore, we have reduced the vertex connectivity problem to the graph of half of the size of the original graph. We repeat this process for \( O(\log n) \) time, and we are done.

For the rest of the section, we provide a high-level overview of the proof of Theorem 5.3.1. We adapt the same strategy for the algorithm in Chalermsook et al. [7] to the vertex cut setting. We start with the notion of cut covering set. We say that a set \( S \) covers another set \( T \) if \( T \subseteq S \).

Definition 5.3.2. A vertex set \( Z \) of an undirected graph \( G \) is \( (T,k) \)-cut covering for some terminal set \( T \), and connectivity parameter \( k \) if for all \( A,B \subseteq T \), if \( \mu(A,B) < k \), then \( Z \) covers some min \( (A,B) \)-weak separator.

We can construct \( (T,k) \)-sparsifier from the \( (T,k) \)-cut covering set \( Z \) as shown in the following lemma.

Lemma 5.3.2. Given an undirected graph \( G = (V,E) \), a terminal set \( T \subseteq V \), and parameter \( k > 0 \), and a \( (T,k) \)-cut covering set \( Z \), there is an \( O(mk) \)-time algorithm that outputs a \( (T,k) \)-sparsifier \( H \) for \( G \) where \( |E(H)| \leq |V(H)|k \) and \( V(H) = |Z \cup T| \).

We discuss the proof of Lemma 5.3.2 in Section 5.3.1. Next, the following result says that we can compute a small \( (T,k) \)-covering set in almost linear time.

Theorem 5.3.3. Given an undirected graph \( G = (V,E) \), a terminal set \( T \subseteq V \), and parameter \( k > 0 \), we can compute a \( (T,k) \)-covering set of size \( |T|2^{O(k^2)} \) in \( m^{1+o(1)}2^{O(k^2)} \) time.
We discuss the proof of Theorem 5.3.3 in Section 5.3.2. Theorem 5.3.1 follows immediately from Lemma 5.3.2 and Theorem 5.3.3.

5.3.1 Remarks on the method of the proof of Lemma 5.3.2

We prove that there is an algorithm that outputs a \((T,k)\)-sparsifier \(H\) for \(G\) where \(|V(H)| = |Z \cup T|\) and \(|E(H)| \leq |V(H)|k\). Then, we give a proof sketch of the fast implementation. The key operation is vertex closure operation.

**Vertex Closure Operation.** Given a graph \(G = (V,E)\), and a vertex \(x \in V\), the closure of \(x\), denoted as \(\text{cl}(G,x)\), is the graph \(G\) after adding cliques among \(N_G(x)\), followed by removing \(x\). For a set \(X \subseteq V\), we define \(\text{cl}(G,X)\) as a sequence of closure operations of vertices in \(X\) in an arbitrary order. The following lemma says that the ordering of closure operations does not change the final graph.

**Lemma 5.3.4.** If \(G[X]\) is connected, then \(\text{cl}(G,X)\) is the same graph as \(G\) after adding the clique edges \(N_G(X)\) followed by removing \(X\) from \(G\).

**Proof.** We show that there is an edge \((u,v)\) in \(\text{cl}(G,X)\) for all \(u,v \in N_G(X)\). Fix \(u,v \in N_G(X)\). Since \(G[X]\) is connected, there is a path \(P\) from \(u\) to \(v\) using only vertices in \(X \cup \{u,v\}\). For any sequence of closure operations on \(X\), the path \(P\) is updated accordingly and remains a path from \(\{u,v\}\) that uses only \(X\). The invariant holds until the end of the sequence. Since \(X\) is empty at the end of the sequence and the invariant holds, the path becomes an edge between \(u\) and \(v\). \(\square\)

We can view any vertex set as a union of connected components. Thus, we have the following corollary.

**Corollary 5.3.5.** For any vertex set \(X \subseteq V\) of \(G\), the graph \(\text{cl}(G,X)\) is the same as the following transformation on \(G\). Let \(S_1,...,S_\ell\) be connected components in \(G[X]\). For each \(i \leq \ell\), we add clique edges between all pairs of vertices in \(N_G(S_i)\). Finally, we remove \(X\) from the graph.

**Lemma 5.3.6** (Monotonicity). For any \(x \in V\), if \(S\) is a separator in \(\text{cl}(G,x)\), then \(S\) is also a separator in \(G\). In particular, \(\kappa_{\text{cl}(G,x)} \geq \kappa_G\).

**Proof.** Let \((L,S,R)\) be a vertex cut in \(\text{cl}(G,x)\). By construction, there is a clique edge on \(N_G(x)\) (i.e., there is an edge \((u,v)\) for all pairs \(u,v \in N_G(x)\)) in \(\text{cl}(G,x)\). Therefore, \(N_G(x) \subseteq L \cup S\) or \(N_G(x) \subseteq S \cup R\). WLOG, assume \(N_G(x) \subseteq L \cup S\). To obtain \(G\) from \(\text{cl}(G,x)\), we add the vertex \(x\) back to \(\text{cl}(G,x)\), add the set of edges from \(x\) to all vertices in \(N_G(x)\), and remove extra edges from the clique in \(N_G(x)\). Since \(N_G(x) \subseteq L \cup S\) and we add only edges from \(x\) to \(N_G(x)\) to obtain \(G\) from \(\text{cl}(G,x)\), we conclude that \((L \cup \{x\},S,R)\) is a vertex cut in \(G\). \(\square\)

We are now ready to provide the algorithm.
Algorithm. We are given an undirected graph $G = (V, E)$ with terminal set $T$, and a $(T, k)$-cut covering set $Z$ as inputs. Let $G' = \text{cl}(G, V - (Z \cup T))$. Then, we run Nagamochi-Ibaraki algorithm (Theorem 2.1.1) on $G'$, and return the resulting graph $H$.

Correctness. We prove that the graph $H$ is a $(T, k)$-sparsifier for $G$ where $|V(H)| = |Z \cup T|$ and $|E(H)| \leq |V(H)| k$. By Lemma 5.3.6, every separator in $G'$ is a separator in $G$, and thus $\kappa_G \geq \kappa_G$. We claim that $G'$ is $(T, k)$-equivalent to $G$ where $|V(G')| = |Z \cup T|$. If true, then Nagamochi-Ibaraki algorithm will output a $(T, k)$-sparsifier where $|E(H)| \leq |V(H)| k$, and $|V(H)| = |Z \cup T|$. We now prove the claim. We show that $G'$ is $(T, k)$-equivalent to $G$, i.e., for all $A, B \subseteq T$, $\min(\mu_G(A, B), k) = \min(\mu_G(A, B), k)$. Fix $A, B \subseteq T$, if $\mu_G(A, B) \geq k$, then $\mu_G(A, B) = \mu_G(A, B) \geq k$ (by monotonicity), and we are done. Now we assume that $\mu_G(A, B) < k$. We show that $G'$ contains a min $(A, B)$-weak separator in $G$. Since $Z$ is a $(T, k)$-covering set, $Z$ contains a min $(A, B)$-weak separator $S$ in $G$. Since $Z \cup T \subseteq V(G')$, and $S \subseteq Z$, $S$ is a $(A, B)$-weak separator in $G'$. Furthermore, $S$ is a min $(A, B)$-weak separator in $G'$ because of monotonicity. Therefore, $c > \mu_G(A, B) = |S| = \mu_G'(A, B)$.

Fast Implementation. The algorithm above may take longer than linear time because the graph may become dense at some point after vertex closure operations. To handle this issue, we replace each connected component $S$ in $G[X]$ where $X = V - (Z \cup T)$ with a $k$-partial clique defined as follows. If $|S| < k$, then we add edges to every pair inside $S$. Otherwise, we select arbitrary $k$ vertices in $S$. For each selected vertex, we add an edge to every other vertex in $S$. After replacement, the graph will have $O(mk)$ edges, and we apply Nagamochi-Ibaraki algorithm (Theorem 2.1.1), which runs in $O(mk)$ time. The $k$-partial clique replacement also has a monotonicity property up to a separator of size $< k$. The argument for correctness is similar.

5.3.2 Remarks on the method of the proof of Theorem 5.3.3

We adapt the technique from Chalermsook et al. [7] for their construction for preserving $k$-edge connectivity. We first briefly review their approach and explain the modification for the vertex cut version.

Computing a $(T, k)$-Covering Set for Edge Cuts. At high level, they show a reduction to $(T, k)$-reducing set that is the set of edges $F$ such that for all $A, B \subseteq T$, if a min $(A, B)$-cut has size $\leq k$, then either $F$ intersects the cut-set or the cut-set will be in different components in $G - F$ (i.e., the cut-set is partially contained by at least two components). Suppose we can compute a small $(T, k)$-reducing set efficiently. The algorithm for $(T, k)$-covering set is as follows. First, compute a $(T, k)$-reducing set $F$, then for each component $X$ of $G - F$, compute a $(T \cap X, k - 1)$-reducing set on each component $X$ of $G - F$ recursively. They return the union of reducing sets of all subproblems.
in the recursion. Using inductive arguments, the output is \((T, k)\)-covering set for \(k\)-edge connectivity. The overhead for \((T, k)\)-covering set is small because the recursion depth is at most \(k\). To see this, fix an \((A, B)\)-cut \(F'\) in the original graph. Each level of the recursion is guaranteed to reduce the size of \(F'\) by at least one by either intersecting \(F'\), or splitting \(F'\) into different components.

**Computing a \((T, k)\)-Reducing Set for Edge Cuts.** Now, to compute a \((T, k)\)-reducing set, they find a \(T\)-Steiner mincut, i.e., a minimum cut-set that splits \(T\) into different components. The key properties are: (1) \(T\)-Steiner mincut will separate the graph into two connected components and, (2) if an \((A, B)\)-cut has size \(\leq k\), then either it intersects \(F\) or it is split by \(F\) into different components or it will be strictly in one of the components. If it intersects \(F\) or it is split by \(F\), then we are done. Now, we assume that it is strictly in one component. In this case, we can recurse on both graphs with appropriate definitions of the new terminal sets. Using inductive arguments, they show that the union of all Steiner cuts in the subproblems (plus the results from the base cases) is a \((T, k)\)-reducing set. Using potential function analysis, they show that the number of subproblems in the recursion depth is \(O(|T|k)\). We briefly discuss the size of the \((T, k)\)-reducing set. Since min Steiner cut on each subproblem has \(O(k)\) edges, the total number of edges from the subproblems (excluding base cases) in the recursion is \(O(|T|k^2)\). However, in the base case (when there are only \(O(k)\) terminals), they need to enumerate all cuts, which takes \(|T|2^{O(k)}\) number of edges from the base cases. For fast implementation, they show that computing Steiner cuts can be done efficiently on expanders, and they use expander decomposition to handle the general graph.

**Computing a \((T, k)\)-Covering Set for Vertex Cuts.** At a high level, we also show a reduction to \((T, k)\)-reducing set where the notion of reducing set in the vertex cuts is more complex than that of the edge version. We say that a vertex set \(F\) is \((T, k)\)-reducing if for all pair \(A, B \subseteq T\) where \(\mu_G(A, B) \leq k\), it either (1) splits a min \((A, B)\)-weak separator \(S\) into different components, i.e., \(|S \cap N_G[X]| \leq k - 1\) for all component \(X\) in \(G - Z\) or (2) it intersects the non-terminal part of a min \((A, B)\)-weak separator \(S\) whose \(|S - T|\) is minimized, i.e., \(|(S \cap X) - T| \leq |S - T| - 1\) for all component \(X\) in \(G - Z\). The algorithm for \((T, k)\)-covering set is similar to the edge version, but with one important detail: While we only recurse on each component in the edge version, we need to recurse on the component \(X\) in \(G - Z\) and its neighborhood \(N_G(X) \subseteq Z\) for the vertex version (i.e., recurse on \(N_G(X)\) with appropriate terminal sets on each component \(X\) in \(G - Z\)). We briefly argue that the recursion depth is still \(O(k)\). Consider the graph \(G\) with terminal set \(T\). Fix an \((A, B)\)-weak separator \(S\) in \(G\) such that \(|S - T|\) is minimized. A \((T, k)\)-reducing set \(Z\) either splits \(S\) into different components or intersects the non-terminal part of it. In the first case, the remaining size of \(S\) in
the subproblem is reduced by at least one. This case can happen at most \( k \) times, i.e., at most \( k \) levels. Otherwise, the second case will ensure that the non-terminal part is getting intersected more as we recurse. This can happen at most \( k \) levels as well. Overall, the recursion depth is at most \( 2k \), which is \( O(k) \). It requires more work and additional structural properties for the set \( Z \) to show that the total instance size is not too large.

**Computing a \((T,k)\)-Reducing Set for Vertex Cuts.** It turns out that a minimum \( T \)-Steiner vertex cut allows us to obtain a similar recursive structure as in the edge version. We define notations before stating the key property of the Steiner cut. A vertex cut \((L,S,R)\) is \( T \)-Steiner if \( L \cap T \neq \emptyset \) and \( R \cap T \neq \emptyset \). It is the minimum when \(|S|\) is the smallest. Given a min \( T \)-Steiner cut \((L,S,R)\), define left and right graphs denoted by \( G_L \) and \( G_R \), respectively. The left graph \( G_L \) is \( cl(G,(S-T) \cup (R-\{t_R\})) \) where \( t_R \in T \cap R \) is an arbitrary terminal node in \( R \), and the right graph \( G_R \) is \( cl(G,(S-T) \cup (L-\{t_L\})) \) where \( t_L \in T \cap L \). Fix \( A,B \subseteq T \) such that \( \mu_G(A,B) \leq k \). Let \( S^* \) be a min \((A,B)\)-weak separator such that \(|S^* - T|\) is minimized. If \( S \) splits \( S^* \) into a different component or intersects the non-terminal vertices of \( S^* \), then we are done. Otherwise, the structure of a min \( T \)-Steiner cut \((L,S,R)\) implies that \( S^* \) must exist in either left graph \( G_L \) or right graph \( G_R \). In this case, we can recurse on both \( G_L \) and \( G_R \) using appropriate terminal sets. In this sense, the algorithm is similar to the edge version. Using similar inductive arguments, one can show that the union of the Steiner cuts (the separator) overall subproblems (plus the base cases) is a \((T,k)\)-reducing set for vertex cuts. One can also use potential function analysis to argue that the number of subproblems is at most \( O(|T|^k) \), but we need to use the base case when the number of terminals is \( O(k) \). By enumerating all the cuts in the base cases, we obtain \( O(|T|^2 2^{O(k^2)}) \) number of vertices in the \((T,k)\)-reducing set.

**Fast Implementation by Lifting to Hypergraphs** For fast implementation, we also show that min Steiner cuts can be computed efficiently in vertex expanders, and we use vertex expander decomposition to handle the general graphs. The key difference in vertex setting from the edge setting is that the left and right graphs \( G_L \) and \( G_R \) require closure operations, which must be done in an online manner. Given a graph \( G \) and \( v \), the closure \( cl(G,v) \) is to add the clique edges between the neighbors of \( v \) and remove \( v \) from \( G \). This operation can take a lot of time when the vertex has high degree. To handle this situation, we “lift” the problem to hypergraphs. We maintain the invariant that every hyperedge represents a clique in the original graph. Using this invariant, the equivalent operation on a hypergraph is as follows: Closing a vertex \( v \) in a hypergraph is to merge all hyperedges containing \( v \) into a single hyperedge, i.e., insert \( e' = \bigcup_{e \ni v} e \) and remove all \( e \neq e' \) that contains \( v \). This operation allows us to simulate vertex closure operation on hypergraphs. The closure operation on hypergraph can be implemented more efficiently using union-find data structures.
5.4 Deterministic Vertex Connectivity Algorithm

We describe the algorithm to decide $k$-vertex connectivity and analysis. Recall that EXPANDERORTERMINAL is the algorithm in Theorem 5.2.3 and VERTEXSPARSIFY is the algorithm in Theorem 5.3.1. We will use $\phi = 1/(10n^{o(1)}2^{O(k^2)})$. By Lemma 5.1.2, we can solve $k$-vertex connectivity for $\phi$-vertex expander $X$ in $\tilde{O}(nk^7/\phi^4) = n^{1+o(1)}2^{O(k^2)}$ time.

**Algorithm.** Let $G$ be the input undirected graph. If there is a vertex with degree $< k$, then return false. Otherwise, we run EXPANDERORTERMINAL($G, k, \phi$), where $\phi = 1/(10n^{o(1)}2^{O(k^2)})$, to obtain a collection of expanders, $\mathcal{X}$, and the terminal set $T$. If one of the expanders is not $k$-vertex connected, then return false. Otherwise, let $T' = \bigcup_{v \in T} N^k_G(v)$ where $N^k_G(v)$ be an arbitrary $k$ neighbors of $v$ in $G$. Let $H = VERTEXSPARSIFY(G, T', k)$. We recurse on $H$ until the size is $\log^{O(1)}n$. That is, we return true if and only if $H$ is $k$-vertex connected. With appropriate bookkeeping, it is possible to output the corresponding vertex cut.

**Running Time.** We use the parameter $\phi^{-1} = 10n^{o(1)}2^{O(k^2)}$. By Theorem 5.2.3, EXPANDERORTERMINAL takes $m^{1+o(1)}\phi = m^{1+o(1)}2^{O(k^2)}$ time. Furthermore, $|T| \leq n^{1+o(1)}\phi$, and the total number of vertices in expanders is $\sum_{X \in \mathcal{X}} |V(X)| = n^{1+o(1)}$. By Lemma 5.1.2, deciding $k$-vertex connectivity in every expander in $\mathcal{X}$ takes $O(\sum_{X \in \mathcal{X}} |V(X)|^{1+o(1)}k^{O(1)}\phi^{-4}) = n^{1+o(1)}2^{O(k^2)}$ time. By Theorem 5.3.1, VERTEXSPARSIFY runs $m^{1+o(1)}2^{O(k^2)}$ time. It remains to prove that we repeat the algorithm for $O(\log n)$ time. By Theorem 5.3.1, we have $|E(H)| \leq |V(H)|c$ and $|V(H)| \leq |T|2^{O(k^2)}$. Since $|T| \leq n^{1+o(1)}\phi$, and $\phi^{-1} = 10n^{o(1)}2^{O(k^2)}$, we have that $|V(H)| \leq n/10$ and $|E(H)| \leq |V(H)|k \leq nk/10 \leq m/10$ (by using appropriate constants in the parameters). The fact that $nk \leq m$ follows since the minimum degree is at least $k$. Therefore, we can repeat at most $O(\log n)$ time.

**Correctness.** We prove that the algorithm outputs true if and only if $G$ is $k$-connected. Assume that $G$ is $k$-connected. By Theorem 5.2.3, every expander is $k$-connected, and $\kappa_G(T) \geq k$. We prove that $H$ is also $k$-connected. Suppose otherwise. $H$ has a separator $S$ of size $< k$. Since $H$ is $k$-cut recoverable, $S$ is also a separator for $G$ of size $< k$, a contradiction. Since $H$ is $k$-connected, we repeat the same argument on $H$ for $O(\log n)$ time until we get to the base case for which the input graph is $k$-connected. Therefore, the algorithm eventually outputs true correctly.

Now, we assume that $G$ is not $k$-connected. If the degree is $< k$, or one of the expanders is not $k$-connected, then the algorithm outputs false, and we are done. We now assume that every expander in $\mathcal{X}$ is $k$-connected. By Theorem 5.2.3, $\kappa_G(T) < k$. Next, we claim that $H$ is not $k$-connected. If true, then we repeat the same argument for $H$ for $O(\log n)$ times until it becomes the base case for which the input graph is not $k$-connected. Thus, the algorithm eventually outputs false correctly.
It remains to prove the claim. Since \( \kappa_G(T) < k \), there is a vertex cut \((L,S,R)\) of size \(< k\) where \( x \in L \cap T \) and \( y \in R \cap T \). Let \( A = N^k_G(x) \) and \( B = N^k_G(y) \). Observe that \( A,B \subseteq T' \), and furthermore \( A \subseteq L \cup S \) and \( B \subseteq S \cup R \). Thus, \( S \) is a weak \((A,B)\)-separator in \( G \) of size \(< k\), and so \( \mu_G(A,B) < k \). Since \( A,B \subseteq T' \), and \( H \) is \((T',k)\)-equivalent to \( G \), we have \( \min\{\mu_H(A,B), k\} = \min\{\mu_G(A,B), k\} = \mu_G(A,B) < k \). Therefore, \( \mu_H(A,B) < k \), and so there is a separator \( S' \) in \( H \) such that there is no path from \( A \) to \( B \) in \( H - S' \). \( S' \) may contain some vertices from \( A \) and \( B \). However, \( S' \) cannot entirely contain \( A \) (and \( B \)) because \( |S'| < k \), but \( |A| \geq k + 1, |B| \geq k + 1 \). Therefore, \( S' \) is a separator of size \(< k\) in \( H \). This completes the claim.
References


